# PROJ coordinate transformation software library

***Release 9.6.2***

**PROJ contributors**

**02 Jun 2025**

# CONTENTS


**Contents** i

**1 About** 1
- 1.1 Citation . . . 1
- 1.2 License . . . 2

**2 News** 3
- 2.1 9.6.2 . . . 3
  - 2.1.1 Updates . . . 3
  - 2.1.2 Bug Fixes . . . 3
- 2.2 9.6.1 . . . 3
  - 2.2.1 Updates . . . 3
  - 2.2.2 Bug Fixes . . . 3
- 2.3 9.6.0 . . . 4
  - 2.3.1 Updates . . . 4
  - 2.3.2 Bug Fixes . . . 5
- 2.4 9.5.1. Release Notes . . . 5
  - 2.4.1 Updates . . . 5
  - 2.4.2 Bug fixes . . . 5
- 2.5 9.5.0 Release Notes . . . 6
  - 2.5.1 Updates . . . 6
  - 2.5.2 Bug fixes . . . 6
- 2.6 9.4.1 Release Notes . . . 7
  - 2.6.1 Updates . . . 7
  - 2.6.2 Bug fixes . . . 7
- 2.7 9.4.0 Release Notes . . . 7
  - 2.7.1 Updates . . . 7
  - 2.7.2 Bug fixes . . . 8
- 2.8 9.3.1 Release Notes . . . 8
  - 2.8.1 Updates . . . 8
  - 2.8.2 Bug fixes . . . 8
- 2.9 9.3.0 Release Notes . . . 9
  - 2.9.1 Updates . . . 9
  - 2.9.2 Bug fixes . . . 9
- 2.10 9.2.1 Release Notes . . . 10
  - 2.10.1 Database updates . . . 10
  - 2.10.2 Bug fixes . . . 10
- 2.11 9.2.0 Release Notes . . . 11
  - 2.11.1 Updates . . . 11
  - 2.11.2 Bug fixes . . . 12

2.12 9.1.1 Release Notes . . . 13
2.12.1 Updates . . . 13
2.12.2 Bug fixes . . . 13
2.13 9.1.0 Release Notes . . . 14
2.13.1 Updates . . . 14
2.13.2 Bug fixes . . . 15
2.14 9.0.1 Release Notes . . . 15
2.14.1 Database updates . . . 15
2.14.2 Bug fixes . . . 15
2.15 9.0.0 Release Notes . . . 16
2.15.1 Breaking Changes . . . 16
2.15.2 Updates . . . 16
2.15.3 Bug fixes . . . 17
2.16 8.2.1 Release Notes . . . 17
2.16.1 Updates . . . 17
2.16.2 Bug fixes . . . 17
2.17 8.2.0 Release Notes . . . 17
2.17.1 Announcements . . . 18
2.17.2 Updates . . . 18
2.17.3 Bug fixes . . . 18
2.18 8.1.1 Release Notes . . . 19
2.18.1 Updates . . . 19
2.18.2 Bug Fixes . . . 19
2.19 8.1.0 Release Notes . . . 20
2.19.1 Updates . . . 20
2.19.2 Bug fixes . . . 21
2.20 8.0.1 Release Notes . . . 21
2.20.1 Updates . . . 21
2.20.2 Bug fixes . . . 21
2.21 8.0.0 Release Notes . . . 22
2.21.1 Updates . . . 22
2.21.2 Bug fixes . . . 22
2.22 7.2.1 Release Notes . . . 23
2.22.1 Updates . . . 23
2.22.2 Bug fixes . . . 23
2.23 7.2.0 Release Notes . . . 23
2.23.1 Updates . . . 23
2.23.2 Bug fixes . . . 24
2.24 7.1.1 Release Notes . . . 24
2.24.1 Updates . . . 25
2.24.2 Bug fixes . . . 25
2.25 7.1.0 Release Notes . . . 25
2.25.1 Updates . . . 25
2.25.2 Bug fixes . . . 27
2.26 7.0.1 Release Notes . . . 27
2.26.1 Updates . . . 27
2.26.2 Bug fixes . . . 27
2.27 6.3.2 Release Notes . . . 28
2.27.1 Bug fixes . . . 28
2.28 7.0.0 Release Notes . . . 29
2.28.1 Updates . . . 29
2.28.2 Bug fixes . . . 30
2.28.3 Breaking changes . . . 30
2.29 6.3.1 Release Notes . . . 30

2.29.1 Updates . . . 31
2.29.2 Bug fixes . . . 31
2.30 6.3.0 Release Notes . . . 31
2.30.1 Updates . . . 31
2.30.2 Bug fixes . . . 32
2.31 6.2.1 Release Notes . . . 32
2.31.1 Updates . . . 32
2.31.2 Bug fixes . . . 32
2.32 6.2.0 Release Notes . . . 33
2.32.1 Updates . . . 33
2.32.2 Bug Fixes . . . 33
2.33 6.1.1 Release Notes . . . 33
2.33.1 Updates . . . 34
2.33.2 Bug Fixes . . . 34
2.34 6.1.0 Release Notes . . . 34
2.34.1 Updates . . . 34
2.34.2 Bug fixes . . . 34
2.35 6.0.0 Release Notes . . . 35
2.35.1 Updates . . . 35
2.35.2 Bug fixes . . . 36
2.36 PROJ 5.2.0 . . . 37
2.36.1 Updates . . . 37
2.36.2 Bug fixes . . . 37
2.37 PROJ 5.1.0 . . . 37
2.37.1 Updates . . . 37
2.37.2 Bug fixes . . . 38
2.38 PROJ 5.0.1 . . . 38
2.38.1 Bug fixes . . . 38
2.39 PROJ 5.0.0 . . . 39
2.39.1 Versioning and naming . . . 39
2.39.2 Updates . . . 40
2.39.3 Bug fixes . . . 41

**3 Download** **43**
3.1 Current Release . . . 43
3.2 Past Releases . . . 43

**4 Installation** **47**
4.1 Installation from package management systems . . . 47
4.1.1 Cross platform . . . 47
4.1.1.1 Conda . . . 47
4.1.1.2 Docker . . . 47
4.1.2 Windows . . . 47
4.1.3 Linux . . . 48
4.1.3.1 Debian . . . 48
4.1.3.2 Fedora . . . 48
4.1.3.3 Red Hat . . . 48
4.1.4 Mac OS X . . . 49
4.2 Compilation and installation from source code . . . 49
4.2.1 Requirements . . . 49
4.2.1.1 Build requirements . . . 49
4.2.1.2 Test requirements . . . 49
4.2.2 Build steps . . . 50
4.2.3 CMake configure options . . . 50

4.2.4 Building on Windows with vcpkg and Visual Studio 2017 or 2019 . . . 54
4.2.4.1 Install git . . . 54
4.2.4.2 Install Vcpkg . . . 54
4.2.4.3 Install PROJ dependencies . . . 54
4.2.4.4 Checkout PROJ sources . . . 54
4.2.4.5 Build PROJ . . . 54
4.2.4.6 Run PROJ tests . . . 55
4.2.5 Building on Windows with Conda dependencies and Visual Studio 2017 or 2019 . . . 55
4.2.5.1 Install git . . . 55
4.2.5.2 Install miniconda . . . 55
4.2.5.3 Install PROJ dependencies . . . 55
4.2.5.4 Checkout PROJ sources . . . 55
4.2.5.5 Build PROJ . . . 55
4.2.5.6 Run PROJ tests . . . 56

**5 Using PROJ** **57**
5.1 Quick start . . . 57
5.2 Cartographic projection . . . 58
5.2.1 Units . . . 58
5.2.2 False Easting/Northing . . . 59
5.2.3 Longitude Wrapping . . . 59
5.2.4 Prime Meridian . . . 59
5.2.5 Axis orientation . . . 60
5.2.6 Order of applications of parameters . . . 61
5.3 Geodetic transformation . . . 61
5.3.1 Transformation pipelines . . . 62
5.3.2 PROJ 4.x/5.x paradigm . . . 63
5.3.3 Grid Based Datum Adjustments . . . 65
5.3.3.1 Skipping Missing Grids . . . 66
5.3.3.2 The null Grid . . . 66
5.3.3.3 Caveats . . . 67
5.4 Ellipsoids . . . 67
5.4.1 Ellipsoid size parameters . . . 67
5.4.2 Ellipsoid shape parameters . . . 68
5.4.3 Ellipsoid spherification parameters . . . 68
5.4.4 Built-in ellipsoid definitions . . . 68
5.4.5 Transformation examples . . . 69
5.5 Environment variables . . . 69
5.6 Known differences between versions . . . 71
5.6.1 Version 4.6.0 . . . 71
5.6.2 Version 5.0.0 . . . 71
5.6.2.1 Longitude wrapping when using custom central meridian . . . 71
5.6.3 Version 6.0.0 . . . 72
5.6.3.1 Removal of `proj_def.dat` . . . 72
5.6.3.2 Changes to deformation . . . 72
5.6.4 Version 6.3.0 . . . 72
5.6.4.1 projinfo . . . 72
5.6.5 Version 7.0.0 . . . 72
5.6.5.1 proj . . . 72
5.6.5.2 cs2cs . . . 72
5.6.5.3 UTF-8 adoption . . . 72
5.7 Network capabilities . . . 73
5.7.1 CDN of GeoTIFF grids . . . 73
5.7.2 How to enable network capabilities ? . . . 73

5.7.3 Setting endpoint . . . 73
5.7.4 Caching . . . 73
5.7.5 Download API . . . 73
5.7.6 Download utility . . . 74
5.7.7 Mirroring . . . 74
5.7.8 Acknowledgements . . . 74

**6 Applications** **75**
6.1 cct . . . 75
6.1.1 Synopsis . . . 75
6.1.2 Description . . . 76
6.1.3 Use of remote grids . . . 77
6.1.4 Examples . . . 77
6.1.5 Background . . . 78
6.2 cs2cs . . . 78
6.2.1 Synopsis . . . 78
6.2.2 Description . . . 79
6.2.2.1 Use of remote grids . . . 82
6.2.3 Examples . . . 82
6.2.3.1 Using PROJ strings . . . 82
6.2.3.2 Using EPSG CRS codes . . . 83
6.2.3.3 Using EPSG CRS names . . . 83
6.3 geod . . . 83
6.4 invgeod . . . 83
6.4.1 Synopsis . . . 83
6.4.2 Description . . . 83
6.4.3 Examples . . . 85
6.4.4 Further reading . . . 85
6.5 gie . . . 85
6.5.1 Synopsis . . . 85
6.5.2 Description . . . 85
6.5.3 Examples . . . 86
6.5.4 gie command language . . . 87
6.5.5 Strict mode . . . 90
6.5.6 Background . . . 90
6.6 proj . . . 90
6.7 invproj . . . 90
6.7.1 Synopsis . . . 90
6.7.2 Description . . . 91
6.7.3 Example . . . 92
6.8 projinfo . . . 93
6.8.1 Synopsis . . . 93
6.8.2 Description . . . 95
6.8.3 Examples . . . 99
6.9 projsync . . . 104
6.9.1 Synopsis . . . 104
6.9.2 Description . . . 104
6.9.3 Examples . . . 105

**7 Coordinate operations** **107**
7.1 Projections . . . 107
7.1.1 Adams Hemisphere in a Square . . . 107
7.1.1.1 Parameters . . . 107
7.1.2 Adams World in a Square I . . . 109

7.1.2.1 Parameters . . . 109
7.1.3 Adams World in a Square II . . . 111
7.1.3.1 Parameters . . . 111
7.1.4 Albers Equal Area . . . 113
7.1.4.1 Options . . . 114
7.1.5 Azimuthal Equidistant . . . 115
7.1.5.1 Options . . . 115
7.1.6 Airocean . . . 117
7.1.6.1 Parameters . . . 119
7.1.7 Airy . . . 119
7.1.7.1 Options . . . 120
7.1.8 Aitoff . . . 121
7.1.8.1 Parameters . . . 122
7.1.9 Modified Stereographic of Alaska . . . 123
7.1.9.1 Options . . . 123
7.1.10 Apian Globular I . . . 124
7.1.10.1 Options . . . 124
7.1.11 August Epicycloidal . . . 125
7.1.11.1 Parameters . . . 126
7.1.12 Bacon Globular . . . 127
7.1.12.1 Parameters . . . 127
7.1.13 Bertin 1953 . . . 128
7.1.13.1 Usage . . . 128
7.1.13.2 Parameters . . . 128
7.1.13.3 Further reading . . . 129
7.1.14 Bipolar conic of western hemisphere . . . 130
7.1.14.1 Parameters . . . 130
7.1.15 Boggs Eumorphic . . . 131
7.1.15.1 Parameters . . . 131
7.1.16 Bonne (Werner lat_1=90) . . . 133
7.1.16.1 Parameters . . . 133
7.1.17 Cal Coop Ocean Fish Invest Lines/Stations . . . 134
7.1.17.1 Usage . . . 136
7.1.17.2 Options . . . 136
7.1.17.3 Mathematical definition . . . 136
7.1.17.4 Further reading . . . 136
7.1.18 Cassini (Cassini-Soldner) . . . 136
7.1.18.1 Usage . . . 137
7.1.18.2 Options . . . 137
7.1.18.3 Mathematical definition . . . 139
7.1.18.4 Further reading . . . 140
7.1.19 Central Cylindrical . . . 141
7.1.19.1 Parameters . . . 142
7.1.20 Central Conic . . . 142
7.1.20.1 Usage . . . 143
7.1.20.2 Parameters . . . 143
7.1.20.3 Mathematical definition . . . 144
7.1.20.4 Reference values . . . 144
7.1.21 Equal Area Cylindrical . . . 145
7.1.21.1 Named specializations . . . 145
7.1.21.2 Parameters . . . 146
7.1.21.3 Further reading . . . 147
7.1.22 Chamberlin Trimetric . . . 147
7.1.22.1 Parameters . . . 147

7.1.23 Collignon . . . 150
7.1.23.1 Parameters . . . 150
7.1.24 Colombia Urban . . . 151
7.1.24.1 Parameters . . . 151
7.1.25 Compact Miller . . . 152
7.1.25.1 Parameters . . . 152
7.1.26 Craster Parabolic (Putnins P4) . . . 154
7.1.26.1 Parameters . . . 154
7.1.27 Denoyer Semi-Elliptical . . . 155
7.1.27.1 Parameters . . . 155
7.1.28 Eckert I . . . 156
7.1.28.1 Parameters . . . 156
7.1.29 Eckert II . . . 158
7.1.29.1 Parameters . . . 158
7.1.30 Eckert III . . . 159
7.1.30.1 Parameters . . . 159
7.1.31 Eckert IV . . . 160
7.1.31.1 Parameters . . . 160
7.1.32 Eckert V . . . 162
7.1.32.1 Parameters . . . 162
7.1.33 Eckert VI . . . 163
7.1.33.1 Parameters . . . 163
7.1.34 Equidistant Cylindrical (Plate Carrée) . . . 164
7.1.34.1 Usage . . . 164
7.1.34.2 Parameters . . . 165
7.1.34.3 Mathematical definition . . . 167
7.1.34.4 Further reading . . . 167
7.1.35 Equidistant Conic . . . 167
7.1.35.1 Parameters . . . 167
7.1.36 Equal Earth . . . 170
7.1.36.1 Usage . . . 170
7.1.36.2 Parameters . . . 171
7.1.36.3 Further reading . . . 171
7.1.37 Euler . . . 172
7.1.37.1 Parameters . . . 172
7.1.38 Fahey . . . 174
7.1.38.1 Parameters . . . 174
7.1.39 Foucaut . . . 175
7.1.39.1 Parameters . . . 175
7.1.40 Foucaut Sinusoidal . . . 176
7.1.40.1 Parameters . . . 177
7.1.41 Gall (Gall Stereographic) . . . 178
7.1.41.1 Usage . . . 179
7.1.41.2 Parameters . . . 179
7.1.41.3 Mathematical definition . . . 180
7.1.41.4 Further reading . . . 180
7.1.42 Geostationary Satellite View . . . 180
7.1.42.1 Usage . . . 181
7.1.42.2 Parameters . . . 183
7.1.43 Ginsburg VIII (TsNIIGAiK) . . . 184
7.1.43.1 Parameters . . . 184
7.1.44 General Sinusoidal Series . . . 185
7.1.44.1 Parameters . . . 186
7.1.45 Gnomonic . . . 186

7.1.45.1 Parameters . . . 188
7.1.46 Goode Homolosine . . . 189
7.1.46.1 Parameters . . . 189
7.1.47 Modified Stereographic of 48 U.S. . . . 190
7.1.47.1 Parameters . . . 190
7.1.48 Modified Stereographic of the 50 U.S. states . . . 191
7.1.48.1 Parameters . . . 191
7.1.49 Guyou . . . 192
7.1.49.1 Parameters . . . 192
7.1.50 Hammer & Eckert-Greifendorff . . . 194
7.1.50.1 Parameters . . . 194
7.1.51 Hatano Asymmetrical Equal Area . . . 196
7.1.51.1 Parameters . . . 196
7.1.52 HEALPix . . . 197
7.1.52.1 Usage . . . 198
7.1.52.2 Parameters . . . 198
7.1.52.3 Further reading . . . 199
7.1.53 rHEALPix . . . 199
7.1.53.1 Usage . . . 200
7.1.53.2 Parameters . . . 200
7.1.53.3 Further reading . . . 201
7.1.54 Interrupted Goode Homolosine . . . 201
7.1.54.1 Parameters . . . 202
7.1.55 Interrupted Goode Homolosine (Oceanic View) . . . 203
7.1.55.1 Parameters . . . 203
7.1.56 Interrupted Mollweide . . . 204
7.1.56.1 Parameters . . . 204
7.1.57 Interrupted Mollweide (Oceanic View) . . . 205
7.1.57.1 Parameters . . . 206
7.1.58 International Map of the World Polyconic . . . 207
7.1.58.1 Parameters . . . 207
7.1.59 Icosahedral Snyder Equal Area . . . 209
7.1.59.1 Parameters . . . 209
7.1.60 Kavrayskiy V . . . 211
7.1.60.1 Parameters . . . 211
7.1.61 Kavrayskiy VII . . . 212
7.1.61.1 Parameters . . . 213
7.1.62 Krovak . . . 214
7.1.62.1 Parameters . . . 214
7.1.63 Laborde . . . 216
7.1.63.1 Parameters . . . 216
7.1.64 Lambert Azimuthal Equal Area . . . 218
7.1.64.1 Parameters . . . 218
7.1.65 Lagrange . . . 220
7.1.65.1 Parameters . . . 220
7.1.66 Larrivee . . . 223
7.1.66.1 Parameters . . . 223
7.1.67 Laskowski . . . 224
7.1.67.1 Parameters . . . 224
7.1.68 Lambert Conformal Conic . . . 225
7.1.68.1 Parameters . . . 226
7.1.68.2 Further reading . . . 228
7.1.69 Lambert Conformal Conic Alternative . . . 228
7.1.69.1 Parameters . . . 228

7.1.70 Lambert Equal Area Conic . . . 230
7.1.70.1 Parameters . . . 231
7.1.71 Lee Oblated Stereographic . . . 232
7.1.71.1 Parameters . . . 232
7.1.72 Loximuthal . . . 232
7.1.72.1 Parameters . . . 234
7.1.73 Space oblique for LANDSAT . . . 235
7.1.73.1 Parameters . . . 235
7.1.73.2 Further reading . . . 237
7.1.74 McBryde-Thomas Flat-Polar Sine (No. 1) . . . 237
7.1.74.1 Parameters . . . 237
7.1.75 McBryde-Thomas Flat-Pole Sine (No. 2) . . . 238
7.1.75.1 Parameters . . . 239
7.1.76 McBride-Thomas Flat-Polar Parabolic . . . 239
7.1.76.1 Parameters . . . 239
7.1.77 McBryde-Thomas Flat-Polar Quartic . . . 241
7.1.77.1 Parameters . . . 241
7.1.78 McBryde-Thomas Flat-Polar Sinusoidal . . . 242
7.1.78.1 Parameters . . . 242
7.1.79 Mercator . . . 243
7.1.79.1 Usage . . . 243
7.1.79.2 Parameters . . . 245
7.1.79.3 Mathematical definition . . . 246
7.1.79.4 Further reading . . . 247
7.1.80 Miller Oblated Stereographic . . . 247
7.1.80.1 Parameters . . . 249
7.1.81 Miller Cylindrical . . . 249
7.1.81.1 Usage . . . 249
7.1.81.2 Parameters . . . 249
7.1.81.3 Mathematical definition . . . 251
7.1.81.4 Further reading . . . 251
7.1.82 Space oblique for MISR . . . 251
7.1.82.1 Parameters . . . 251
7.1.83 Modified Krovak . . . 253
7.1.83.1 Parameters . . . 255
7.1.84 Mollweide . . . 255
7.1.84.1 Parameters . . . 256
7.1.85 Murdoch I . . . 257
7.1.85.1 Parameters . . . 257
7.1.86 Murdoch II . . . 259
7.1.86.1 Parameters . . . 259
7.1.87 Murdoch III . . . 261
7.1.87.1 Parameters . . . 261
7.1.88 Natural Earth . . . 263
7.1.88.1 Usage . . . 263
7.1.88.2 Parameters . . . 263
7.1.88.3 Further reading . . . 264
7.1.89 Natural Earth II . . . 264
7.1.89.1 Parameters . . . 265
7.1.90 Nell . . . 266
7.1.90.1 Parameters . . . 266
7.1.91 Nell-Hammer . . . 267
7.1.91.1 Parameters . . . 267
7.1.92 Nicolosi Globular . . . 268

7.1.92.1 Parameters . . . 268
7.1.93 Near-sided perspective . . . 269
7.1.93.1 Parameters . . . 271
7.1.94 New Zealand Map Grid (EPSG:27200) . . . 272
7.1.94.1 Parameters . . . 272
7.1.95 General Oblique Transformation . . . 273
7.1.95.1 Usage . . . 273
7.1.95.2 Parameters . . . 273
7.1.96 Oblique Cylindrical Equal Area . . . 275
7.1.96.1 Parameters . . . 275
7.1.97 Oblated Equal Area . . . 277
7.1.97.1 Parameters . . . 277
7.1.98 Oblique Mercator . . . 279
7.1.98.1 Usage . . . 280
7.1.98.2 Parameters . . . 281
7.1.98.3 Caveats . . . 282
7.1.99 Ortelius Oval . . . 282
7.1.99.1 Parameters . . . 283
7.1.100 Orthographic . . . 284
7.1.100.1 Parameters . . . 285
7.1.101 Patterson . . . 286
7.1.101.1 Parameters . . . 286
7.1.102 Perspective Conic . . . 288
7.1.102.1 Parameters . . . 288
7.1.103 Peirce Quincuncial . . . 290
7.1.103.1 Parameters . . . 290
7.1.104 Polyconic (American) . . . 294
7.1.104.1 Parameters . . . 294
7.1.105 Putnins P1 . . . 295
7.1.105.1 Parameters . . . 295
7.1.106 Putnins P2 . . . 296
7.1.106.1 Parameters . . . 297
7.1.107 Putnins P3 . . . 298
7.1.107.1 Parameters . . . 298
7.1.108 Putnins P3' . . . 299
7.1.108.1 Parameters . . . 299
7.1.109 Putnins P4' . . . 300
7.1.109.1 Parameters . . . 300
7.1.110 Putnins P5 . . . 301
7.1.110.1 Parameters . . . 302
7.1.111 Putnins P5' . . . 303
7.1.111.1 Parameters . . . 303
7.1.112 Putnins P6 . . . 304
7.1.112.1 Parameters . . . 304
7.1.113 Putnins P6' . . . 305
7.1.113.1 Parameters . . . 305
7.1.114 Quartic Authalic . . . 306
7.1.114.1 Parameters . . . 307
7.1.115 Quadrilateralized Spherical Cube . . . 308
7.1.115.1 Usage . . . 309
7.1.115.2 Parameters . . . 310
7.1.115.3 Further reading . . . 311
7.1.116 Robinson . . . 311
7.1.116.1 Parameters . . . 311

7.1.117 Roussilhe Stereographic . . . 313
7.1.117.1 Parameters . . . 313
7.1.118 Rectangular Polyconic . . . 314
7.1.118.1 Parameters . . . 315
7.1.119 S2 . . . 316
7.1.119.1 Usage . . . 317
7.1.119.2 Parameters . . . 317
7.1.119.3 Further reading . . . 318
7.1.120 Spherical Cross-track Height . . . 319
7.1.120.1 Parameters . . . 319
7.1.121 Sinusoidal (Sanson-Flamsteed) . . . 320
7.1.121.1 Parameters . . . 321
7.1.122 Space Oblique Mercator (SOM) . . . 321
7.1.122.1 Parameters . . . 323
7.1.122.2 Further reading . . . 324
7.1.123 Swiss Oblique Mercator . . . 324
7.1.123.1 Parameters . . . 324
7.1.124 Spilhaus . . . 325
7.1.124.1 Parameters . . . 327
7.1.125 Stereographic . . . 328
7.1.125.1 Parameters . . . 328
7.1.126 Oblique Stereographic Alternative . . . 330
7.1.126.1 Parameters . . . 331
7.1.127 Gauss-Schreiber Transverse Mercator (aka Gauss-Laborde Reunion) . . . 332
7.1.127.1 Parameters . . . 333
7.1.128 Transverse Central Cylindrical . . . 334
7.1.128.1 Parameters . . . 334
7.1.129 Transverse Cylindrical Equal Area . . . 335
7.1.129.1 Parameters . . . 337
7.1.130 Times . . . 337
7.1.130.1 Parameters . . . 338
7.1.131 Tissot . . . 339
7.1.131.1 Parameters . . . 339
7.1.132 Transverse Mercator . . . 341
7.1.132.1 Usage . . . 341
7.1.132.2 Parameters . . . 342
7.1.132.3 Mathematical definition . . . 343
7.1.132.4 Further reading . . . 347
7.1.133 Tobler-Mercator . . . 347
7.1.133.1 Usage . . . 347
7.1.133.2 Parameters . . . 348
7.1.133.3 Mathematical definition . . . 349
7.1.134 Two Point Equidistant . . . 350
7.1.134.1 Parameters . . . 350
7.1.135 Tilted perspective . . . 351
7.1.135.1 Parameters . . . 351
7.1.136 Universal Polar Stereographic . . . 354
7.1.136.1 Parameters . . . 354
7.1.137 Urmaev V . . . 355
7.1.137.1 Parameters . . . 355
7.1.138 Urmaev Flat-Polar Sinusoidal . . . 357
7.1.138.1 Parameters . . . 358
7.1.139 Universal Transverse Mercator (UTM) . . . 358
7.1.139.1 Usage . . . 359

7.1.139.2 Parameters . . . 360
7.1.139.3 Further reading . . . 360
7.1.140 van der Grinten (I) . . . 360
7.1.140.1 Parameters . . . 362
7.1.141 van der Grinten II . . . 362
7.1.141.1 Parameters . . . 362
7.1.142 van der Grinten III . . . 364
7.1.142.1 Parameters . . . 364
7.1.143 van der Grinten IV . . . 366
7.1.143.1 Parameters . . . 366
7.1.144 Vitkovsky I . . . 368
7.1.144.1 Parameters . . . 368
7.1.145 Wagner I (Kavrayskiy VI) . . . 370
7.1.145.1 Parameters . . . 370
7.1.146 Wagner II . . . 371
7.1.146.1 Parameters . . . 372
7.1.147 Wagner III . . . 372
7.1.147.1 Parameters . . . 373
7.1.148 Wagner IV . . . 374
7.1.148.1 Parameters . . . 374
7.1.149 Wagner V . . . 375
7.1.149.1 Parameters . . . 376
7.1.150 Wagner VI . . . 377
7.1.150.1 Parameters . . . 377
7.1.151 Wagner VII . . . 378
7.1.152 Web Mercator / Pseudo Mercator . . . 379
7.1.152.1 Usage . . . 379
7.1.152.2 Parameters . . . 379
7.1.152.3 Mathematical definition . . . 380
7.1.152.4 Further reading . . . 380
7.1.153 Werenskiold I . . . 380
7.1.153.1 Parameters . . . 380
7.1.154 Winkel I . . . 382
7.1.154.1 Parameters . . . 382
7.1.155 Winkel II . . . 383
7.1.155.1 Parameters . . . 383
7.1.156 Winkel Tripel . . . 385
7.1.156.1 Parameters . . . 385
7.2 Conversions . . . 386
7.2.1 Axis swap . . . 386
7.2.1.1 Usage . . . 387
7.2.1.2 Parameters . . . 387
7.2.2 Geodetic to cartesian conversion . . . 387
7.2.2.1 Usage . . . 387
7.2.2.2 Parameters . . . 388
7.2.3 Geocentric Latitude . . . 388
7.2.3.1 Mathematical definition . . . 388
7.2.3.2 Usage . . . 389
7.2.3.3 Parameters . . . 389
7.2.4 Lat/long (Geodetic alias) . . . 389
7.2.4.1 Parameters . . . 389
7.2.5 No operation . . . 390
7.2.6 Pop coordinate value to pipeline stack . . . 390
7.2.6.1 Examples . . . 391

7.2.6.2 Parameters . . . 391
7.2.6.3 Further reading . . . 391
7.2.7 Push coordinate value to pipeline stack . . . 391
7.2.7.1 Examples . . . 392
7.2.7.2 Parameters . . . 392
7.2.7.3 Further reading . . . 393
7.2.8 Set coordinate value . . . 393
7.2.8.1 Example . . . 393
7.2.8.2 Parameters . . . 393
7.2.9 Geocentric to topocentric conversion . . . 394
7.2.9.1 Usage . . . 395
7.2.9.2 Parameters . . . 396
7.2.10 Unit conversion . . . 397
7.2.10.1 Parameters . . . 397
7.2.10.2 Distance units . . . 398
7.2.10.3 Angular units . . . 398
7.2.10.4 Time units . . . 398
7.3 Transformations . . . 399
7.3.1 Affine transformation . . . 399
7.3.1.1 Parameters . . . 400
7.3.2 Multi-component time-based deformation model . . . 401
7.3.2.1 Parameters . . . 401
7.3.2.2 Example . . . 401
7.3.3 Kinematic datum shifting utilizing a deformation model . . . 401
7.3.3.1 Example . . . 402
7.3.3.2 Parameters . . . 403
7.3.3.3 Mathematical description . . . 404
7.3.3.4 See also . . . 404
7.3.4 Geographic offsets . . . 404
7.3.4.1 Examples . . . 405
7.3.4.2 Parameters . . . 405
7.3.5 General grid shift . . . 405
7.3.5.1 Example . . . 406
7.3.6 Helmert transform . . . 406
7.3.6.1 Examples . . . 407
7.3.6.2 Parameters . . . 407
7.3.6.3 Mathematical description . . . 409
7.3.7 Horner polynomial evaluation . . . 410
7.3.7.1 Examples . . . 412
7.3.7.2 Parameters . . . 413
7.3.7.3 Further reading . . . 415
7.3.8 Molodensky transform . . . 415
7.3.8.1 Examples . . . 415
7.3.8.2 Parameters . . . 415
7.3.9 Molodensky-Badekas transform . . . 416
7.3.9.1 Example . . . 416
7.3.9.2 Parameters . . . 417
7.3.9.3 Mathematical description . . . 417
7.3.10 Horizontal grid shift . . . 418
7.3.10.1 Temporal gridshifting . . . 418
7.3.10.2 Parameters . . . 419
7.3.11 Triangulated Irregular Network based transformation . . . 420
7.3.11.1 Parameters . . . 420
7.3.11.2 Example . . . 420

7.3.12 Vertical Offset And Slope . . . 423
7.3.12.1 Examples . . . 423
7.3.12.2 Parameters . . . 423
7.3.12.3 Formula . . . 424
7.3.13 Vertical grid shift . . . 424
7.3.13.1 Temporal gridshifting . . . 425
7.3.13.2 Parameters . . . 426
7.3.14 Geocentric grid shift . . . 426
7.3.14.1 Example . . . 427
7.4 The pipeline operator . . . 427
7.4.1 Rules for pipelines . . . 428
7.4.2 Parameters . . . 429
7.4.2.1 Required . . . 429
7.4.2.2 Optional . . . 429
7.5 Computation of coordinate operations between two CRS . . . 430
7.5.1 Introduction . . . 430
7.5.2 Geographic CRS to Geographic CRS, with known identifiers . . . 431
7.5.3 Filtering and sorting of coordinate operations . . . 431
7.5.4 Geodetic/geographic CRS to Geodetic/geographic CRS, without known identifiers . . . 432
7.5.5 Geodetic/geographic CRS to Geodetic/geographic CRS, without direct transformation . . . 435
7.5.6 Projected CRS to any target CRS . . . 438
7.5.7 Vertical CRS to a Geographic CRS . . . 439
7.5.8 Vertical CRS to a Vertical CRS . . . 439
7.5.9 Compound CRS to a Geographic CRS . . . 440
7.5.10 CompoundCRS to CompoundCRS . . . 442
7.5.11 When the source or target CRS is a BoundCRS . . . 444
7.5.12 Using Derived Projected for source or target CRS . . . 445

**8 Resource files** **447**
8.1 Where are PROJ resource files looked for ? . . . 447
8.2 `proj.db` . . . 448
8.3 `proj.ini` . . . 448
8.4 Transformation grids . . . 450
8.5 External resources and packaged grids . . . 450
8.5.1 proj-data . . . 450
8.5.2 proj-datumgrid . . . 450
8.5.3 Regional packages . . . 450
8.5.4 World package . . . 451
8.5.5 -latest packages . . . 451
8.6 Other transformation grids . . . 451
8.6.1 Free grids . . . 451
8.6.1.1 Hungary . . . 451
8.6.2 Non-Free Grids . . . 451
8.6.2.1 Austria . . . 451
8.6.2.2 Brazil . . . 451
8.6.2.3 Netherlands . . . 451
8.6.2.4 Portugal . . . 451
8.6.3 HTDP . . . 451
8.6.3.1 Getting and building HTDP . . . 452
8.6.3.2 Getting `crs2crs2grid.py` . . . 452
8.6.3.3 Usage . . . 452
8.6.3.4 See Also . . . 453
8.7 Init files . . . 453

**9 Geodesic calculations** **455**
9.1 Introduction . . . 455
9.2 Solution of geodesic problems . . . 455
9.3 Additional properties . . . 455
9.4 Multiple shortest geodesics . . . 456
9.5 Area of a polygon . . . 456
9.6 Background . . . 456

**10 Development** **459**
10.1 Development practices . . . 459
10.1.1 Commit messages . . . 459
10.1.2 Commit hooks . . . 459
10.1.3 Blame ignore file . . . 459
10.2 Quick start for C API usage . . . 459
10.3 Quick start for C++ API usage . . . 465
10.4 Error handling . . . 468
10.5 Reference . . . 469
10.5.1 Macros . . . 469
10.5.2 Data types . . . 470
10.5.2.1 Transformation objects . . . 470
10.5.2.2 2 dimensional coordinates . . . 471
10.5.2.3 3 dimensional coordinates . . . 471
10.5.2.4 Spatiotemporal coordinate types . . . 472
10.5.2.5 Ancillary types for geodetic computations . . . 473
10.5.2.6 Complex coordinate types . . . 474
10.5.2.7 Projection derivatives . . . 475
10.5.2.8 List structures . . . 476
10.5.2.9 Info structures . . . 478
10.5.2.10 Error codes . . . 480
10.5.2.11 Logging . . . 481
10.5.2.12 Setting custom I/O functions . . . 482
10.5.2.13 Network related functionality . . . 483
10.5.2.14 C API for ISO-19111 functionality . . . 484
10.5.3 Functions . . . 494
10.5.3.1 Threading contexts . . . 494
10.5.3.2 Transformation setup . . . 495
10.5.3.3 Area of interest . . . 498
10.5.3.4 Coordinate transformation . . . 499
10.5.3.5 Error reporting . . . 503
10.5.3.6 Logging . . . 505
10.5.3.7 Info functions . . . 506
10.5.3.8 Lists . . . 506
10.5.3.9 Distances . . . 507
10.5.3.10 Various . . . 508
10.5.3.11 Setting custom I/O functions . . . 511
10.5.3.12 Network related functionality . . . 513
10.5.3.13 Cleanup . . . 518
10.5.3.14 C API for ISO-19111 functionality . . . 518
10.5.4 C++ API . . . 581
10.5.4.1 General documentation . . . 581
10.5.4.2 common namespace . . . 583
10.5.4.3 util namespace . . . 591
10.5.4.4 metadata namespace . . . 597
10.5.4.5 cs namespace . . . 606

10.5.4.6 datum namespace . . . 623
10.5.4.7 crs namespace . . . 638
10.5.4.8 operation namespace . . . 662
10.5.4.9 io namespace . . . 732
10.6 Using PROJ in CMake projects . . . 762
10.7 Language bindings . . . 762
10.7.1 Excel . . . 762
10.7.2 Fortran . . . 762
10.7.3 Go (Golang) . . . 762
10.7.4 Java . . . 762
10.7.5 Julia . . . 762
10.7.6 MySQL . . . 762
10.7.7 Perl . . . 763
10.7.8 Python . . . 763
10.7.9 Ruby . . . 763
10.7.10 Rust . . . 763
10.7.11 TCL . . . 763
10.7.12 Visual Basic . . . 763
10.8 Version 4 to 6 API Migration . . . 763
10.8.1 Code example . . . 763
10.8.2 Function mapping from old to new API . . . 765
10.8.3 Backward incompatibilities . . . 765
10.8.4 Feedback from downstream projects on the PROJ 6 migration . . . 766
10.9 Version 4 to 5 API Migration . . . 766
10.9.1 Background . . . 766
10.9.2 Code example . . . 767
10.9.3 Function mapping from old to new API . . . 768

**11 Specifications** **771**
11.1 PROJJSON . . . 771
11.1.1 Introduction . . . 771
11.1.2 Normative references . . . 771
11.1.3 Definitions . . . 771
11.1.4 Schema . . . 771
11.1.5 History of the schema . . . 772
11.1.6 Specification . . . 772
11.1.6.1 High level objects . . . 773
11.1.6.2 Identifiers . . . 773
11.1.6.3 Object usages . . . 774
11.1.6.4 Units . . . 774
11.1.6.5 Omitted units in measured parameters . . . 775
11.1.6.6 Coordinate system . . . 775
11.1.7 Examples . . . 775
11.1.7.1 GeographicCRS . . . 775
11.1.7.2 ProjectedCRS . . . 780
11.1.7.3 CompoundCRS . . . 782
11.1.7.4 BoundCRS . . . 785
11.1.7.5 Transformation . . . 789
11.1.8 Deviations with the WKT2:2019 specification . . . 791
11.1.8.1 PROJJSON extensions . . . 791
11.1.8.2 PROJJSON omissions . . . 791
11.1.9 Reference implementation . . . 791
11.1.10 Online CRS definitions as PROJJSON . . . 791
11.1.11 Copyright and License information . . . 791

11.2 Geodetic TIFF grids (GTG) . . . 791
11.2.1 Introduction . . . 792
11.2.2 General description . . . 792
11.2.3 Example . . . 798
11.2.4 Multi-grid storage . . . 800
11.2.5 Examples of multi-grid dataset . . . 802
11.2.6 Revisions . . . 802

**12 Community** **803**
12.1 Communication channels . . . 803
12.1.1 Mailing list . . . 803
12.1.2 GitHub . . . 803
12.1.3 Gitter . . . 803
12.2 Contributing . . . 804
12.2.1 Help a fellow PROJ user . . . 804
12.2.2 Adding bug reports . . . 804
12.2.3 Feature requests . . . 804
12.2.4 Write documentation . . . 805
12.2.5 Code contributions . . . 805
12.2.5.1 Legalese . . . 805
12.2.6 Additional Resources . . . 805
12.2.7 Acknowledgements . . . 806
12.3 Guidelines for PROJ code contributors . . . 806
12.3.1 Code contributions. . . . 806
12.3.1.1 Making Changes . . . 806
12.3.1.2 Submitting Changes . . . 806
12.3.1.3 Coding conventions . . . 806
12.3.2 Tools . . . 807
12.3.2.1 Reformatting C++ code . . . 807
12.3.2.2 cppcheck static analyzer . . . 807
12.3.2.3 Clang Static Analyzer (CSA) . . . 807
12.3.2.4 Typo detection and fixes . . . 808
12.3.2.5 Include What You Use (IWYU) . . . 808
12.4 Code of Conduct . . . 808
12.4.1 Our Pledge . . . 808
12.4.2 Our Standards . . . 808
12.4.3 Our Responsibilities . . . 809
12.4.4 Scope . . . 809
12.4.5 Enforcement . . . 809
12.4.6 Attribution . . . 809
12.5 PROJ Service Providers . . . 809
12.5.1 Core Contributors . . . 810
12.5.2 Contributors . . . 810
12.5.3 Other Service Providers . . . 810
12.5.4 How to Add a Company . . . 810
12.6 Request for Comments . . . 810
12.6.1 PROJ RFC 1: Project Committee Guidelines . . . 811
12.6.1.1 Summary . . . 811
12.6.1.2 List of PSC Members . . . 811
12.6.1.3 Detailed Process . . . 812
12.6.1.4 When is Vote Required? . . . 812
12.6.1.5 Observations . . . 813
12.6.1.6 Committee Membership . . . 813
12.6.1.7 Membership Responsibilities . . . 813

12.6.1.8 Updates . . . 813
12.6.2 PROJ RFC 2: Initial integration of "GDAL SRS barn" work . . . 813
12.6.2.1 Summary . . . 814
12.6.2.2 Related standards . . . 814
12.6.2.3 Details . . . 814
12.6.2.4 Code repository . . . 818
12.6.2.5 Database . . . 818
12.6.2.6 Utilities . . . 820
12.6.2.7 Impacted files . . . 824
12.6.2.8 C API . . . 826
12.6.2.9 Documentation . . . 827
12.6.2.10 Testing . . . 827
12.6.2.11 Build requirements . . . 827
12.6.2.12 Runtime requirements . . . 827
12.6.2.13 Backward compatibility . . . 827
12.6.2.14 Future work . . . 828
12.6.2.15 Adoption status . . . 828
12.6.3 PROJ RFC 3: Dependency management . . . 828
12.6.3.1 Summary . . . 828
12.6.3.2 Background . . . 829
12.6.3.3 C and C++ . . . 829
12.6.3.4 Software dependencies . . . 829
12.6.3.5 Bootstrapping . . . 829
12.6.3.6 Adoption status . . . 830
12.6.4 PROJ RFC 4: Remote access to grids and GeoTIFF grids . . . 830
12.6.4.1 Motivation . . . 830
12.6.4.2 Summary of work planned by this RFC . . . 831
12.6.4.3 Network access to grids . . . 831
12.6.4.4 Grids in GeoTIFF format . . . 838
12.6.4.5 Dropping grid catalog functionality . . . 841
12.6.4.6 Backward compatibility issues . . . 841
12.6.4.7 Potential future related work . . . 841
12.6.4.8 Documentation . . . 841
12.6.4.9 Testing . . . 841
12.6.4.10 Proposed implementation . . . 841
12.6.4.11 Adoption status . . . 842
12.6.5 PROJ RFC 5: Adopt GeoTIFF-based grids for grids delivered with PROJ . . . 842
12.6.5.1 Motivation . . . 842
12.6.5.2 Summary of work planned by this RFC and related decisions . . . 842
12.6.5.3 Backward compatibility . . . 843
12.6.5.4 Testing . . . 843
12.6.5.5 Proposed implementation . . . 843
12.6.5.6 Adoption status . . . 844
12.6.6 PROJ RFC 6: Triangulation-based transformations . . . 844
12.6.6.1 Summary . . . 844
12.6.6.2 Details . . . 844
12.6.6.3 Backward compatibility . . . 848
12.6.6.4 Testing . . . 848
12.6.6.5 Documentation . . . 848
12.6.6.6 Proposed implementation . . . 849
12.6.6.7 References . . . 849
12.6.6.8 Adoption status . . . 849
12.6.6.9 Funding . . . 849
12.6.7 PROJ RFC 7: Drop Autotools, maintain CMake . . . 849

12.6.7.1 Summary . . . 849
12.6.7.2 Background . . . 849
12.6.7.3 Motivation . . . 850
12.6.7.4 Why drop Autotools? . . . 850
12.6.7.5 Why use CMake? . . . 850
12.6.7.6 Why not CMake? . . . 850
12.6.7.7 Potential impacts . . . 850
12.6.7.8 Transition plan . . . 851
12.6.7.9 Adoption status . . . 851
12.6.8 PROJ RFC-8: Embedding resource files into libproj . . . 851
12.6.8.1 Summary . . . 851
12.6.8.2 Motivation . . . 851
12.6.8.3 Technical solution . . . 852
12.6.8.4 New CMake options . . . 852
12.6.8.5 Impacted code . . . 853
12.6.8.6 Out of scope . . . 853
12.6.8.7 Backward compatibility . . . 853
12.6.8.8 Documentation . . . 853
12.6.8.9 Testing . . . 853
12.6.8.10 Related issues and PRs . . . 854
12.6.8.11 Voting history . . . 854

**13 Tutorials** **855**
13.1 Transformation tutorial . . . 855
13.1.1 Introduction . . . 855
13.1.2 How to fill out the PROJ tutorial exercises . . . 855
13.1.3 Exercises . . . 856
13.1.3.1 The UTM projection . . . 856
13.1.3.2 The Lambert Conic Conformal projection . . . 857
13.1.3.3 Specifying ellipsoid models . . . 860
13.1.3.4 Gridshifting . . . 862
13.1.3.5 The Helmert transformation . . . 863
13.1.3.6 Coordinate conversions . . . 866
13.1.3.7 Transformation pipelines . . . 868
13.1.3.8 The Transverse Mercator projection . . . 870
13.1.3.9 Answers to the PROJ tutorial exercises . . . 873
13.2 Geodesics and planar approximations using PROJ and geod . . . 875
13.2.1 1. Objective . . . 876
13.2.2 2. The geodesic . . . 876
13.2.3 1. The planar approximation . . . 877
13.2.4 4. Suggested meditations . . . 878
13.2.5 Answers . . . 878

**14 FAQ** **879**
14.1 Which file formats does PROJ support? . . . 879
14.2 Can I transform from *abc* to *xyz*? . . . 879
14.3 Coordinate reference system *xyz* is not in the EPSG registry, what do I do? . . . 879
14.4 I found a bug in PROJ, how do I get it fixed? . . . 880
14.5 How do I contribute to PROJ? . . . 880
14.6 How do I calculate distances/directions on the surface of the earth? . . . 880
14.7 What is the best format for describing coordinate reference systems? . . . 880
14.8 Which CRS apply to a given location? . . . 880
14.9 Why is the axis ordering in PROJ not consistent? . . . 880
14.10 Why am I getting the error "Cannot find proj.db"? . . . 881

14.11 What happened to PROJ.4? . . . 881
14.12 Who uses PROJ ? . . . 882
14.12.1 Software and organizations using or contributing to PROJ . . . 882
14.12.1.1 Free and open source software . . . 882
14.12.1.2 PROJ bindings in other languages . . . 882
14.12.1.3 Proprietary licensed software . . . 882
14.12.1.4 Geodetic organizations . . . 882
14.12.1.5 Public organizations . . . 883
14.12.1.6 Private organizations . . . 883
14.12.1.7 Sponsors . . . 883

**15 Glossary** **885**

**Bibliography** **887**

**Index** **891**

CHAPTER

ONE

# ABOUT

PROJ is a generic coordinate transformation software that transforms geospatial coordinates from one coordinate reference system (CRS) to another. This includes cartographic projections as well as geodetic transformations. PROJ is released under the *X/MIT open source license*

PROJ includes *command line applications* for easy conversion of coordinates from text files or directly from user input. In addition to the command line utilities PROJ also exposes an *application programming interface*, or API in short. The API lets developers use the functionality of PROJ in their own software without having to implement similar functionality themselves.

PROJ started purely as a cartography application letting users convert geodetic coordinates into projected coordinates using a number of different cartographic projections. Over the years, as the need has become apparent, support for datum shifts has slowly worked its way into PROJ as well. Today PROJ supports more than a hundred different map projections and can transform coordinates between datums using all but the most obscure geodetic techniques.

## 1.1 Citation

To cite PROJ in publications use:

> PROJ contributors (2025). PROJ coordinate transformation software library. Open Source Geospatial Foundation. URL https://proj.org/. DOI: 10.5281/zenodo.5884394

A BibTeX entry for LaTeX users is

```
@Manual{,
  title = {{PROJ} coordinate transformation software library},
  author = {{PROJ contributors}},
  organization = {Open Source Geospatial Foundation},
  year = {2025},
  url = {https://proj.org/},
  doi = {10.5281/zenodo.5884394},
}
```

## 1.2 License

PROJ uses the MIT license. The software was initially released by the USGS in the public domain. When Frank Warmerdam took over the development of PROJ it was moved under the MIT license. The full text of the license follows, and can also be found in the file `COPYING`, at the top level of the source distribution package.

```
All source, data files and other contents of the PROJ package are
available under the following terms.  Note that the PROJ 4.3 and earlier
was "public domain" as is common with US government work, but apparently
this is not a well defined legal term in many countries. Frank Warmerdam placed
everything under the following MIT style license because he believed it is
effectively the same as public domain, allowing anyone to use the code as
they wish, including making proprietary derivatives.

Initial PROJ 4.3 public domain code was put as Frank Warmerdam as copyright
holder, but he didn't mean to imply he did the work. Essentially all work was
done by Gerald Evenden.

Copyright information can be found in source files.

 --------------

 Permission is hereby granted, free of charge, to any person obtaining a
 copy of this software and associated documentation files (the "Software"),
 to deal in the Software without restriction, including without limitation
 the rights to use, copy, modify, merge, publish, distribute, sublicense,
 and/or sell copies of the Software, and to permit persons to whom the
 Software is furnished to do so, subject to the following conditions:

 The above copyright notice and this permission notice shall be included
 in all copies or substantial portions of the Software.

 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
 OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
 DEALINGS IN THE SOFTWARE.
```

CHAPTER

TWO

# NEWS

## 2.1 9.6.2

*June 6th 2025*

### 2.1.1 Updates

- Update EPSG to 12.013

### 2.1.2 Bug Fixes

- Database: change ESRI deprecated records back to their deprecated state (#4519)
- ProjectedCRS::identify(): more robust identification of old ESRI names using _IntlFeet (#4518)
- Fix issue related to D_Unknown_based_on_WGS_84_ellipsoid datum name (#4514)
- proj_trans_bounds(): sample points within the source grid to avoid missing extent such as world-wide coverage from EPSG:4326 to ESRI:54099 (Spilhaus) (#4513)
- proj_trans_bounds(): avoid erroring out, in some cases, when PJ* object is directly constructed from a PROJ pipeline (#4512)

## 2.2 9.6.1

*June 2nd 2025*

### 2.2.1 Updates

- Update EPSG to 12.012 (#4498)
- Update ESRI CRS DB to ArcGIS Pro 3.5 (#4493)
- Add an option to ship PDB files (#4505)

### 2.2.2 Bug Fixes

- Deal with South Orientated TMerc presented as regular TMerc but with Scale_Factor=-1 (#4427)
- Add missing 'limits' and 'algorithm' includes (#4444)
- CMake: avoid detection of C23 `#embed` on too old compilers (#4463)
- `createOperations()`: for NAD83(CSRS)[x] to NAD83(CSRS)[y], do not go through NAD83 (#4467)
- Add missing include for `uint32_t` on Windows with gcc 15.1 (#4478)

- `getGeoidModels()`: use `vertical_crs.datum_code` to get vcrss that uses different units (#4480)
- `FileManager::open_resource_file()`: honour known URLs from `grid_alternatives` that don't point to cdn.proj.org (#4481)
- `createOperationsCompoundToGeog()`: avoid bringing non-sensical transformations (#4494)
- Bug correction in Wagner VI projection parameters (#4453)
- C++: Fix bare `isnan()` to `std::isnan()` (#4472)
- Database: fix duplicated entries with auxiliary database (#4482)
- Use `std::abs` instead of `abs` for NetBSD (#4486)
- Fix Windows build when both `EMBED_RESOURCE_FILES` and `USE_ONLY_EMBEDDED_RESOURCE_FILES` are set (#4508)

## 2.3 9.6.0

*March 15th 2025*

### 2.3.1 Updates

- *projinfo*: display whether an operation is time-dependent (#4255)
- Add CMake options `EMBED_RESOURCE_FILES` and `USE_ONLY_EMBEDDED_RESOURCE_FILES` for `proj.db` and `proj.ini` embedding (#4265, #4274)
- Database: ingest deprecated ESRI names to be able to better import old WKT ESRI (#4282)
- *proj_factors()*: enhance speed when called repeatedly on same compound or projected CRS (#4289)
- Add Airocean projection (formerly Dymaxion) (#4303)
- `createOperations()`: Optimization by avoiding database lookups for CRS (typically PROJ strings) using unknown datums (#4320)
- PROJJSON export: for a Projected CRS, add an explicit type=GeographicCRS/GeodeticCRS members to the base_crs member (#4331)
- *proj_trans_bounds()*: make it work when target CRS is a CompoundCRS (#4333)
- `createOperations()`: do Helmert transformation in 2D when one of source or target CRS is compound (#4337)
- Map EPSG method 'Vertical Offset by Grid Interpolation (asc)' to `+proj=vgridshift` (#4344)
- Add support for 'Coordinate Frame rotation full matrix (geocen)' and Coordinate Frame rotation full matrix (geog2D)' methods (#4348)
- Add a CMake `EMBED_RESOURCE_DIRECTORY` option to embed .tif/.json files in to libproj (#4349, #4373)
- Take into account new EPSG methods 'Cartesian Grid Offsets by TIN Interpolation (JSON)' and 'Vertical Offset by TIN Interpolation (JSON)' (#4353)
- Added an option `native_ca` to `proj.ini` and an environment variable *PROJ_NATIVE_CA* to be able to configure curl to use the operating system CA store. (#4356)
- Database: add an optional `step_direction` column to the `concatenated_operation_step` table (#4357)
- Database: replicate EGM2008 grid transformation record to WGS 84 realizations (#4363)
- Improve ETRFxxx to ETRFyyy, and WGS 84 (xxx) to WGS 84 (yyy) (#4364)

- Build: Update to C++17 version (#4366, #4369)
- Add bash completion script for projinfo (#4371)
- Dispatch content of `4D_api.cpp` into multiple .cpp files (#4393)
- Add *proj_trans_bounds_3D()* (#4394)
- Database: add a `engineering_datum` and `engineering_crs` tables; import few related EPSG records (#4396)
- NKG transformations: Add support for EUREF-FIN in Finish transformations (#4399)
- Add Spilhaus projection (#4401)
- Export ESRI:54099 *WGS_1984_Spilhaus_Ocean_Map_in_Square* as PROJ string (#4402)
- Database: update to EPSG v12.004 (#4413)
- *proj_download_file()*: invalidate in-memory caches related to downloaded file (for the current context) (#4414)

### 2.3.2 Bug Fixes

- `createObjectsFromName()`: be more tolerant about N/S vs North/South, absence of zone or height (#4318)
- `createFromPROJString()`: avoid repeated openings of proj.db and proj.ini and lookup of 'epsg' (#4334)
- Debugging/internals: append sub-grid index to grid name when opening a multi-grid TIFF file (#4381)
- `createOperations()`: use more appropriate operation when using a 'PROJ {grid_name}' geoid model, based on matching the vertical datum (#4379)
- *proj_clone()*: properly propagate `errorIfBestTransformationNotAvailable` and other flags from source object (#4382)
- Inverse `+proj=cass`: fix non-convergence on inputs where easting=false_easting or northing=false_northing (#4386)
- Fix typo in parameter for ITRF97 in ITRF2014 file (#4408)

## 2.4 9.5.1. Release Notes

*December 1st 2024*

### 2.4.1 Updates

- Database: update to EPSG v11.022 (#4254, #4275, #4297, #4312)
- Speed-up dramatically proj.db build time (#4280)
- Database: ingest deprecated ESRI names to be able to better import old WKT ESRI (#4282)

### 2.4.2 Bug fixes

- *findsOpsInRegistryWithIntermediate()*: Restrict to using known source/target CRS that have the same originating authority (#4252)
- *isTimeDependent()*: Do not mark "static" Coordinate Frame helmert transformations as time-dependent, only time-dependent ones (#4259)
- Fix clang-19 "warning: empty paragraph passed to '@throw' command [-Wdocumentation]" (#4263)
- WKT importer: fix `nullptr` dereference on invalid `VERTCS[]` (#4266)

- Fixed `sql_filelist.cmake` relative path issue (#4296)
- Fail consistently on network error and grid transformations (#4302)
- Fix `-Wshadow` warnings between `parammappings.cpp` and `singleoperation.cpp` with unity builds (#4304)
- Removed leftover setting of `CMAKE_REQUIRED_FLAGS` (#4322)
- WKT1 ESRI export: fix wrong mapping of Lambert Cylindrical Equal Area to Behrmann (#4328)
- Numerous documentation fixes (#4248, #4267, #4291, #4293)

## 2.5 9.5.0 Release Notes

*Septemper 15th 2024*

### 2.5.1 Updates

- Database: add constraint for unicity of CRS and operation names (#4071)
- Make a few more functions that accept a `PJ*` to actually accept a `const PJ*` (#4074)
- Database: Increase consistency in trigger checks (#4080)
- Speed-up `+proj=cart +inv` (#4087)
- Added EPSG:9656 "Cartesian Grid Offsets" operation method, and imported related records (#4094)
- Add a `WKTParser::grammarErrorList()` method so that *proj_create_from_wkt()* can behave as documented (#4108)
- `CRS::stripVerticalComponent()`: Redirect it to `demoteTo2D(std::string(), nullptr)` (#4127)
- Add `proj_context_set_user_writable_directory()` to public API (#4144)
- CMake: for Windows builds, defaults PROJ DLL to be just `proj_${PROJ_MAJOR_VERSION}.dll` (#4167)
- Add `toWGS84AutocorrectWrongValues()` method and use it in PROJ.4 and WKT1 CRS import (#4172)
- WKT import: add support for Oracle WKT "Albers_Conical_Equal_Area" spelling (#4181)
- Limit number of C++ exceptions thrown & caught internally (#4183)
- Database: update to EPSG 11.016 (#4241)
- CMake: add option `EMBED_PROJ_DATA_PATH` (#4207)
- Add Inverse to Icosahedral Snyder Equal-Area (ISEA) projection (#4211)
- *projinfo*: Add 2 args equivalent to usage of *-s* and *-t* (#4218)
- CMake: add `TESTING_USE_NETWORK` configure option (#4220)
- Add `PROJ_ERR_COORD_TRANSFM_MISSING_TIME` and *proj_coordoperation_requires_per_coordinate_input_time()* (#4221)
- Add new Conversion "Local Orthographic" (#4228, #4229)
- Add data/ITRF2020 with ITRF2020 to ITRFother, and ITRF2020 plate motion models (#4235)

### 2.5.2 Bug fixes

- Fix `imoll` and `imoll_o` zone calculations to correct inverse transformations near the "seams" (#4159)
- Fix wrong EPSG conversion code for UTM south (#4166)
- `BoundCRS::exportToPROJ()`: handle case of NADCON conus grid (#4168)

- `+proj=gridshift`: make projected grids work with *PROJ_NETWORK=ON* (#4174)
- `createOperations()`: make it work when transforming from/to a CompoundCRS with a DerivedVerticalCRS with ellipsoidal height (#4176)
- Support default context as `nullptr` in `CoordinateTransformer` (#4177)
- *cct*: emit error message and return error code when not being able to open input file (#4200)
- Fix race condition surrounding DB reopening during fork (#4231)
- `GeographicBoundingBox::create()`: accept degenerate bounding box reduced to a point or a line (#4237)
- `createOperation()`: tune so that ITRF2000->ETRS89 does not return only NKG grid based operations but also time-dependent Helmert (#4244)

## 2.6 9.4.1 Release Notes

*June 1st 2024*

### 2.6.1 Updates

- Database: update to EPSG v11.006 (#4086)
- Database: update ESRI dataset to ArcGIS Pro 3.3 (#4143)

### 2.6.2 Bug fixes

- `DatabaseContext::lookForGridInfo()`: do not trigger network activity in PROJ_GRID_AVAILABILITY_KNOWN_AVAILABLE mode when all grids are known (#4090)
- Retry "Connection reset by peer" (#4107)
- Use unique names for NKG transformations (#4111)
- `proj_create()`/`proj_create_from_database()`: recall CRS AUTH:CODE in error message (#4114)
- `DerivedProjectedCRS`: deal with lack of explicit CS in BASEPROJCRS in WKT (#4139)
- `proj_symbol_rename.h`: update with current symbols (#4145)
- Improve detection of compatible/incompatible celestial bodies (#4150)

## 2.7 9.4.0 Release Notes

*March 1st 2024*

### 2.7.1 Updates

- Add support for transformations involving coordinate epoch changes (PointMotionOperation), specifically for Canadian NAD83(CSRS)(#3884)
- `SSL_OPTIONS`: set SSL options on Curl library (#3936)
- Add support for unity builds (#3962)
- Added ability to install `*.tif` if present in data (#3970)
- `createOperationsCompoundToGeog()`: tune selection logic when `--grid-check` known_available is specified (#3990)
- Increase CMake minimum version from 3.9 to 3.16 (#3997)

- CMake: use FetchContent to get googletest-1.12.1 (#4006)
- CMake: Replace custom FindSqlite3 with FindSQLite3 built-in (#4007)
- tpeqd: use numerically stable formula for computing the central angle from (phi_1, lam_1) to (phi_2, lam_2) (#4009)
- Move content of `proj_experimental.h` to `proj.h` (#4019)
- Add `+proj=mod_krovak` projection method for Modified Krovak that applies to S-JTSK/05 in the Czech Republic (#4021)
- PROJString formatter optimizer: simplify pipelines doing [Modified]Krovak (South West) <--> [Modified]Krovak (East North) by just doing an axis swap (#4034)
- `+proj=gridshift`: enhance to support grids referenced in projected CRS, and with easting_offset/northing_offset corrections (#4038)
- Tune concatenated operation instantiation, reference CR2005 geoid for Czechia and add (disabled by default) records for Czechia S-JTSK/05 based transformations (#4044)
- Database: update to EPSG v11.004 (#4062)

### 2.7.2 Bug fixes

- Fix missing symbols at link time for Windows target in Visual Studio (#3984)
- Improve error message in axisswap (#3975)
- Avoid convergence errors in `+proj=gridshift` when using biquadratic interpolation (#3985)

## 2.8 9.3.1 Release Notes

*December 1st 2023*

### 2.8.1 Updates

- Update to EPSG 10.098 (#3968)
- Update ESRI objects to v3.2.0 (#3944)

### 2.8.2 Bug fixes

- ITRF2008: fix wrong sign for `dry` parameter of EURA and EURA_T (#3870)
- Fix build error with MSVC 2019 in `/std:c++20` on `NN_NO_CHECK()` (#3872)
- ESRI WKT import: normalize GCS_unknown to unknown and D_unknown to unknown (#3874)
- `CoordinateOperationFactory()`: deal with CompoundToCompound with a horizontal similarity transformation and a ballpark vertical (#3881)
- `Ellipsoid::_isEquivalentTo()`: fix so that an ellipsoid of semi-major axis A (and non-zero inv flattening) isn't equivalent to a sphere of radius A (#3882)
- `isEquivalentTo()`: make a datum name 'unknown' equivalent to another one (#3883)
- *cs2cs*: fix handling of input coordinates in grad (#3886)
- Make `setargv.obj` available on Universal Windows Platform (#3891)
- Allow opening proj.db with a URI (#3892)
- `createOperations()`: fix GeogCRS 3D with TOWGS84 to geocentric CRS (#3915)

- Fix test suite so that it can pass with `ENABLE_TIFF=OFF` (#3916)
- `GeographicBoundingBox::intersects()`: avoid infinite recursion and stack overflow on invalid bounding boxes (#3919)
- Fix importing `'+proj=topocentric ... +type=crs'` by using a geocentric CRS as the base CRS (#3924)
- Allow LOCAL_CS with 3 axes (#3928)
- WKT1 parser: in non-strict mode, accept missing UNIT[] in GEOGCS, GEOCCS, PROJCS and VERT_CS elements (#3933)
- `createOperations()`: fix issue with a obscure case involving CompoundCRS of unknown horizontal datum + boundCRS of vertical (#3934)
- `createOperations()`: fix bad PROJ pipeline when converting between Geog3D with non-metre height to CompoundCRS (#3943)
- `createOperations()`: Fix possible null dereference on invalid WKT input (#3946)
- `proj_factor()`: fix when input is a compound CRS of a projected CRS (#3950)
- `pj_get_suggested_operation()`: tune it to give correct result for RGAF09 to RRAF 1991 / UTM zone 20N + Guadeloupe 1988 height transformation (#3954)
- Move static `NameSpace::GLOBAL` definition in `static.cpp` to avoid 'static initialization fiasco' (#3956)
- horner: allow arbitrary input type of coordinate (#3961)

## 2.9 9.3.0 Release Notes

*September 1st 2023*

### 2.9.1 Updates

- Add C++ API to transform coordinate (#3705)
- CMake: add `PROJ_DB_CACHE_DIR` (#3711)
- Implement EPSG:1026 Mercator (Spherical) method (#3741)
- CMake: remove useless cross-compiling related checks (#3747)
- Add mapping of Equidistant Conic to new EPSG:1119 method (#3812)
- Implement proposal001 from Planetary DWG from OGC (#3816)
- Add option in *proj* CLI to use a CRS (#3825)
- *`proj_factors()`*: make it work with projected CRS with non-metre unit and/or northing/easting axis order (#3826)
- *`PROJ_DEBUG`*: make ON an alias of 2, and OFF of 1 (#3835)
- Database: update to EPSG 10.094 (#3847)

### 2.9.2 Bug fixes

- `GeographicBoundingBox::intersection()`: avoid infinite recursion and stack overflow on invalid bounding boxes (#3748)
- Various fixes related to concatenated operations (#3820)
- Projected CRS identification: fix crash when the base CRS is a non-geographic geodetic CRS (#3830)

- Avoid C++ exceptions to be thrown (and caught) when parsing strings like `+proj=longlat +datum=WGS84 +type=crs` (#3834)
- BUG: Handle prefix whitespace when guessing WKT dialiect (#3841)
- *`proj_alter_id()`*: make it replace an existing ID instead of appending a new one (#3846)
- bonne: fix inverse map projection computations when lat_1 < 0 (#3849)
- WKT1 ESRI import/export: fix GCS name for EPSG:8353 S-JTSK_[JTSK03]_Krovak_East_North (#3851)

## 2.10 9.2.1 Release Notes

*June 1st 2023*

### 2.10.1 Database updates

- Add alias for old ESRI datum/CRS names of EPSG:8353 S_JTSK_JTSK03_Krovak_East_North (#3678)
- Pseudo Mercator now supported with IAU CRS's (#3645)
- Updated to EPSG 10.088 (#3731)

### 2.10.2 Bug fixes

- Fixed build errors with Cygwin (#3642)
- PROJ string CRS parser: make sure that PROJ arguments of the rotated string are kept in the WKT representation (#3656)
- Fix slowness on `proj_trans()` when doing WGS 84 <--> NAD83 conversions (#3661)
- `vgridshift`/`gridshift`: accept `hydroid_height` as valid band description (#3675)
- WKT/PROJJSON: import/export accuracy of ConcatenatedOperation (#3676)
- Coordinate operation factory: count identified concatenated operations as a single step (#3677)
- EngineeringCRS: make `` `proj_create_engineering_crs()` `` set a datum name and relax `isEquivalentTo()` comparisons (#3685)
- PROJJSON: fix import/export of integer parameter value, and deal with interpolation CRS parameters in conversions (#3695)
- CMake: avoid imbalanced cmake_policy push/pop if TIFF or CURL dependency cannot be found (#3697)
- `proj_create_crs_to_crs()`: restore transformation selection behavior of PROJ 9.1 when grids are missing (#3707)
- ESRI WKT: improve roundtrip of name and definition for UPS WGS84 CRS (#3720)
- CRS instantiation from PROJ.4 string: set 'Unknown based on XXXX ellipsoid' datum name when instantiating from known +a, +rf (#3727)
- *cs2cs* / `proj_create_crs_to_crs()`: fix regression with geocentric CRS (#3729)
- `proj_trans()`: set `PROJ_ERR_COORD_TRANSFM_NO_OPERATION` error when failing in `ONLY_BEST=YES` mode (#3730)
- `tinshift`: raise maximum size of JSON file to 100 MB (#3736)

## 2.11 9.2.0 Release Notes

*March 1st 2023*

### 2.11.1 Updates

- Database
  - EPSG updated to EPSG 10.082 (#3609)
  - ESRI objects updated to v3.1.0 (#3575)
  - Added Slovenian geoid model SLO-VRP2016/Koper (#3610)
- Projections
  - Added generic Space Oblique Mercator (#3338)
  - Inverse `eck4`: Add checks for projection domain (#3625)
- Transformations
  - Added support for NADCON5 grids and transformation method (#3510)
  - Added option `ONLY_BEST=YES` (#3535)

    When enabled only use the best possible transformation between two CRS's can be instantiated. If the best possible transformations can't be instantiated it will fail instead of opting for the next best transformation.

    With `ONLY_BEST=YES` a transformation will fail if a needed grid is unavailable.

    The option can be toggled in `proj.ini` and with `--only-best` in *cs2cs*.
  - Implement 'Similarity transformation' (EPSG:9621) and import related transformations from EPSG (#3578)
  - Optimise transformations between GDA94, GDA2020 and WGS84 (#3621)
- Performance improvements
  - Pass *PJ_COORD* by reference for `fwd4d()` and `inv4d()` callbacks (#3503)
  - Improved of inverse meridian length calculations (#3516)
  - Avoid very poor performance on a geographic CRS whose datum name is the alias of an official one (#3624)
- Well-Known Text improvements
  - Support WKT:2019 COORDINATEMETADATA[] construct (#2711)
  - implement `alterCSLinearUnit()` for CompoundCRS (#3325)
  - Implement `alterCSLinearUnit()` for BoundCRS (#3327)
  - Added *PJ_TYPE_DERIVED_PROJECTED_CRS* (#3488)
  - Added support for ISO19111 CoordinateMetadata class (#3562)
  - Update WKT import to support WKT2 corrigendum (#3573)
  - WKT import: Make axis direction and range meaning enumeration case insensitive as mandated by spec (#3577)
  - Implement AffineCS and handle it in WKT (#3579)
  - WKT1 parser: Support `inf` as inverse flattening value (#3626)
- PROJJSON

  - Version number changed to 0.6
  - Added an additional optional `source_crs` property in `abridged_transformation` (#3454)
  - Added support for ISO19111 CoordinateMetadata class (#3562)
  - Implement AffineCS and handle it in PROJJSON (#3579)
- C/C++ API
  - C API: add *proj_get_domain_count()*, *proj_get_area_of_use_ex()* and *proj_get_scope_ex()* (#3588)
  - Added explicit `NaN` handling to *proj_trans()* and *gie* (#3603)

  **DEPRECATIONS**

  The following functions have been deprecated and will be removed in a future version.
  - C API function *proj_rtodms()*. Use *proj_rtodms2()* instead (#3431)
  - C experimental API function *proj_create_conversion_tunisia_mapping_grid()*. Use *proj_create_conversion_tunisia_mining_grid()* instead (#3559)
  - C++ API function `createTunisiaMapingGrid()`. Use `createTunisiaMiningGrid()` instead (#3559)
- CMake
  - Added `uninstall` target (#3362)
  - Adopted use of `TIFF:TIFF` and `CURL::libcurl` targets (#3374)
- Security improvements
  - Replaced uses of `sprintf()` with the safer `snprintf()` (#3421, #3431)
- Various other improvements
  - Implemented platform independent mutex for better results on lesser used platforms (#3425)
  - Improved name and remarks in descriptions of some vertical transformations (#3334)
  - Implemented stricter matching by name to avoid false positives (#3487)
  - Improved numerical accuracy of the meridian length calculations and its inverse (#3516)
  - Added `PROJ_IGNORE_CELESTIAL_BODY=YES` environment variable that allows better matching of non-earth shaped ellipsoids to other celestial bodies (#3569)

### 2.11.2 Bug fixes

- `rtodms()`: fix potential buffer overflow not dealt with (#3441)
- `NetworkFile::open()`: fix memory leak in unlikely error code path (#3443)
- When importing LOCAL_CS["foo"], generates a non-empty name for the datum (#3491)
- Avoid unit mix-up in `CRS::alterCSLinearUnit()` for `DerivedProjectedCRS` (#3499)
- WKT to PROJ conversions: Map 'Lambert Azimuthal Equal Area (Spherical)' to `+proj=laea +R_A` (#3526)
- ProjectedCRS::identify(): Relax trust in id/authority in definition to identify (#3532)
- Fix instantiation of http://www.opengis.net/def/crs/IAU/2015/xxx URNs (#3569)
- Fix *proj_normalize_for_visualization()* on a transformation with `FORCE_OVERb=YES` (#3570)
- `createBoundCRSToWGS84IfPossible()`: Do not report `+towgs84` for a CRS with multiple extents, such as EPSG:6316 (#3587)

- `pj_get_suggested_operation()`: handle longitudes outside of [-180,180] for coordinate operation selection (#3595)
- WKT1 parser: only emit warning when encountering invalid UNKNOWN WKT1 axis direction (#3618)
- *projinfo*: fix long option name `--source-crs` (#3601)

## 2.12 9.1.1 Release Notes

*December 1st 2022*

### 2.12.1 Updates

- Database: register `at_bev_AT_GIS_GRID_2021_09_28` grid (#3442)

### 2.12.2 Bug fixes

- Implement `alterGeodeticCRS` and `stripVerticalComponent` for `DerivedProjected` (#3482)
- Various improvements to handling of `DerivedProjectedCRS` (#3482, #3477 , #3342 , #3319, #3317)
- Add missing <cstdint> includes (#3459)
- cs2cs: better validate value of `-W` option (#3453)
- `DatabaseContext::lookForGridInfo()`: fix caching of filenames and set correct URLs (#3448)
- Database: register in `grid_alternatives` grids from PROJ-data that have no corresponding transformation record (#3446)
- `cass` projection: fix forward computation of easting (#3433)
- Implement `Geographic/Vertical Offset` conversions (#3413)
- `vandg` projection: handle `+over` to extend the validity domain outside of `|long|>180deg` (#3427)
- `eqdc` projection: avoid floating point division by zero in non-nominal case (#3415)
- `createOperations()`: fix issues when transforming between `Geog3D` and `DerivedGeog3D` CRS with `Geographic3D` offsets method (#3411)
- `VerticalCRS::_isEquivalentTo()`: do not consider `VerticalCRS` and `DerivedVerticalCRS` as equivalent (#3408)
- **cct and cs2cs: Avoid problems when encountering UTF-8 *BOM* ` characters at** beginning of input files (#3395)
- `createFromUserInput()`: Improved lookup when approximate name is provided (#3371)
- projinfo / cs2cs : auto promotion to 3D of CRS specified by name (#3367)
- `findsOpsInRegistryWithIntermediate()`: make it work when source/target geodetic CRS has no known id (#3365)
- `createOperations()`: emulate PROJ < 6 behavior when doing geocentric <--> geographic transformation between datum with unknown transformation (#3361)
- Fix issue when transforming from/into a WKT2 Bound `VerticalCRS` with a 'Geographic3D to GravityRelatedHeight' method (#3355)
- *`proj_normalize_for_visualization()`*: take into account `FORCE_OVER` property from source operation (#3350)
- Link geodtest against libm when available (#3341)

## 2.13 9.1.0 Release Notes

*September 1st 2022*

### 2.13.1 Updates

- Database:
  - Database: update to EPSG v10.074 (#3273)
  - Update ESRI objects to version 3.0.0 (#3257)
  - Generate null transformation between geodetic/vertical datum ensemble and its members (#3266)
  - Add `fk delete cascade` to database tables for improved customizability of the database (#3237)
  - Add Svalbard geoid grid `no_kv_arcgp-2006-sk` to grid alternatives (#3292)
  - Added French grid `fr_ign_RAF20` to grid alternatives (#3228)
- WKT & PROJJSON:
  - WKT2 parser: allow unit-less PARAMETER for coordinate operations (#3271)
  - WKT and PROJJSON adjustments (#3234)
  - Bump PROJJSON to v0.5 (#3216)
  - Change order of horizontal and vertical operations when dealing with WKT1 / PROJ4 compound CRS (#3127)
- Coordinate Operations:
  - Implement Vertical Offset and slope transformation method (#3200)
  - Add Interrupted Mollweide (land and ocean view) projections (#3139)
  - Add invertible Horner polynomials (#3133)
- Optimizations:
  - NTv2 and GTX grid readers: add a cache of lines to speed-up (#3241)
  - `IComparable::isEquivalentTo()` optimizations (#3233)
- Various Updates:
  - `createOperations()` / *`proj_trans()`*: better take into account area of interest name / do not only keep first returned coordinate operation (#3282)
  - `createOperations()`: take into account axis unit and inversion of target DerivedProjectedCRS (#3281)
  - Allow a Compound CRS with derived projected (#3283)
  - Make "Vertical Offset and Slope" method usable as a deriving conversion of a DerivedVerticalCRS (#3270)
  - PROJ pipeline generator: recognize opposite Helmert transformations using a different convention (#3265)
  - `createOperations()`: prefer simpler pipelines / affects WGS 84 to GDA94/GDA2020 (#3248)
  - Make it possible to determine which grid files were actually used during a transformation (#3180)
  - Transformation: no longer do vertical transformation when doing compound CRS to 2D CRS / add --3d to cs2cs (#3119)
  - Deprecate PROJ4 CMake name (#3094)
  - Use GNUInstallDirs for CMake install locations (#3100)

  - Introduce `PROJ_DATA` environment variable to deprecate `PROJ_LIB` (#3253)

### 2.13.2 Bug fixes

- *projinfo*: fix crash on --list-crs when proj.db cannot be opened (#3290)
- `getCRSInfoList()`: fix retrieval of projected_crs given by text_definition only (#3289)
- WKT parser: fix issue when parsing some WKT1 with Hotine_Oblique_Mercator_Azimuth_Center and ignoring rectified_grid_angle (#3280)
- WKT1 import: better deal when the angular unit of the GEOGCS[] of the PROJCS[] doesn't match the one from the database (#3274)
- `createOperations()`: avoid potential infinite recursions (#3264)
- WKT ESRI: fix related to projected CRS export and add support for geographic 3D CRS with LINUNIT node (#3256)
- *`proj_grid_info()`*: make it work again with remote grids (#3239)
- `createOperations()`: fix projected CRS with non-metre horiz&vertical unit and towgs84 to gencentric CRS (#3231)
- `createOperations()`: fix infinite recursion in a complex case (#3226)

## 2.14 9.0.1 Release Notes

*June 15th 2022*

### 2.14.1 Database updates

- Update to EPSG 10.064 (#3208)
- Add OGC:CRS84h (WGS 84 longitude-latitude-height) (#3155)

### 2.14.2 Bug fixes

- Use `CMAKE_INSTALL_MANDIR` to override default (#3081)
- Increase `MAX_ITER` so Mollweide forward projection works near the poles (#3082)
- Fix wrong results with SQLite 3.38.0 (#3091)
- Fix issue when transforming from/to BoundCRS of 3D CRS with non-Greenwich prime meridian, created from WKT (#3098)
- Fix issues with WKT of concatenated operations (#3105)
- CMake: fix installation of `proj.pc` on Windows (#3109)
- `createOperations()`: fix issue in transformation northing, easting projected CRS -> +proj=longlat +lon_wrap (#3110)
- `unitconvert`: round to nearest date when converting to yyyymmdd (#3111)
- Fix comparison of GeodeticReferenceFrame vs DynamicGeodeticReferenceFrame (#3120)
- `createOperations()`: fix transformation involving CompoundCRS, ToWGS84 and PROJ4_GRIDS (#3124)
- Fix datum names when importing from PROJ4 crs strings (affects some transformations using geoidgrids) (#3129)
- Deal with `PARAMETER["EPSG code for Interpolation CRS",crs_code]` (#3149)

- `createOperations()`: fix `CompoundCRS[BoundCRS[ProjectedCRS],BoundCRS[VerticalCRS]]` to Geog3DCrs (#3151)
- ITRF2014: fix ITRF2014:ITRF88,ITRF94 and ITRF96 definitions (#3159)
- `createBoundCRSToWGS84IfPossible()`: improve selection logic to generate +towgs84= taking into account extent (#3160)
- `createOperations()`: fix some complex cases involving TOWGS84 and CompoundCRS (#3163)
- Fix CMake CURL dependency (#3185)
- WKT import: deal with Projected CRS that is a mix of WKT1:GDAL / WKT1:ESRI (#3189)
- `createOperations()`: fix/improve result of 'BD72 + Ostend height' to 'WGS84+EGM96 height' (#3199)
- `Identifier::isEquivalentName()`: fix when ending by ' + ' which could indirectly cause an infinite stack call in master (#3202)
- WKT import: correctly deal with absence of Latitude_Of_Origin parameter in WKT1 ESRI with Stereographic projection (#3212)
- PROJJSON parser: do not error out if a datum ensemble member is unknown in the database (#3223)

## 2.15 9.0.0 Release Notes

*March 1st 2022*

### 2.15.1 Breaking Changes

- Support for the autotools build system has been removed (#3027) See RFC7 for details: https://proj.org/community/rfc/rfc-7.html

### 2.15.2 Updates

- Database updates:
  - ESRI projection engine db to version 12.9 (#2943)
  - EPSG v10.054 (#3051)
  - Vertical grid files for PL-geoid-2011, Polish geoid model (#2960)
  - Belgian geoid model hBG18 to grid alternatives (#3044)
- Add new option to *`proj_create_crs_to_crs_from_pj()`* method to force `+over` on transformation operations (#2914)
- Specify `CMAKE_INSTALL_RPATH` for macOS; use `-rpath LDFLAGS` for tests (#3009)
- Implement Geographic3D to Depth/Geog2D+Depth as used by ETRS89 to CD Norway depth (#3010)
- Allow `PROJ_LIB` paths wrapped with double quotes (#3031)
- Use external gtest by default when possible (#3035)
- CMake: make `BUILD_SHARED_LIBS=ON` the default even on Windows (#3042)
- `proj.ini`: add a `ca_bundle_path` variable (#3049)

### 2.15.3 Bug fixes

- Fix extremely long parsing time on hostile PROJ strings (#2968)
- CMake: fix warning with external googletest (#2980)
- *proj_get_crs_info_list_from_database()*: report PJ_TYPE_GEODETIC_CRS for IAU_2015 -ocentric geodetic CRS (#3013)
- peirce_q: rename +type parameter wrongly introduced in 8.2.1 to +shape (#3014)
- Set more precise error code for parsing errors in *proj_create()* (#3037)
- createOperations(): fix transformations from/to a BoundCRS of a DerivedGeographicCRS coming from WKT (#3046)
- Better deal with importing strings like +init=epsg:XXXX +over (#3055)
- Fix importing CRS definition with +proj=peirce_q and +shape different from square or diamond (#3057)

## 2.16 8.2.1 Release Notes

*January 1st 2022*

### 2.16.1 Updates

- Database updated with EPSG v. 10.041 (#2974)

### 2.16.2 Bug fixes

- BoundCRS WKT import: fix setting of name (#2917)
- PROJStringFormatter::toString(): avoid invalid iterator increment (#2932)
- Ensure CApi test are cross-platform (#2934)
- createOperations(): do not stop at the first operation in the PROJ namespace for vertical transformations (#2937)
- createOperationsCompoundToCompound(): fix null pointer dereference when connection to proj.db doesn't exist. (#2938)
- Fix windows.h conflict with Criterion::STRICT (#2950)
- Cache result of *proj_get_type()* to help for performance of *proj_factors()* (#2967)
- createOperations(): improvement for "NAD83(CSRS) + CGVD28 height" to "NAD83(CSRS) + CGVD2013(CGG2013) height" (#2977)
- WKT1 import: correctly deal with missing rectified_grid_angle parameter (#2986)
- Fix and additional options for Peirce Quincuncial projections (#2978)
- Fix build with Intel C++ compiler (#2995)

## 2.17 8.2.0 Release Notes

*November 1st 2021*

### 2.17.1 Announcements

From PROJ 9.0.0 and onwards CMake will be the only build system bundled with the PROJ package. As a consequence support for Autotools builds will stop when the 8.2 branch of PROJ reaches end of life. We encourage everyone to adjust their build workflows as soon as possible and report any discrepancies discovered between Autotools and CMake builds.

Details about the build system unification can be found in *PROJ RFC 7: Drop Autotools, maintain CMake*.

Note also that the "CMake: revise handling of symbol export and static builds" change mentioned below may require changes for users of the library on Windows.

### 2.17.2 Updates

- Added the S2 projection (#2749)
- Added support for Degree Sign on input (#2791)
- ESRI WKT: add support for import/export of (non interrupted) Goode Homolosine (#2827)
- Make filemanager aware of UWP Win32 API (#2831)
- Add *proj_create_conversion_pole_rotation_netcdf_cf_convention()* to address netCDF datasets using a pole rotation method (#2835)
- Emit better debug message when a grid isn't found (#2838)
- Add support for GeodeticCRS using a Spherical planetocentric coordinate system (#2847)
- PROJJSON: support additional properties allowed in id object (version, authority_citation, uri) for parity with WKT2:2019 (#2850)
- Database layout modified to include "anchor" field to `geodetic_datum` and `vertical_datum` tables, consequently database layout version is increased to 1.2 (#2859)
- *proj_factors()*: accept *P* to be a projected CRS (#2868)
- Add IAU_2015 CRS definitions (#2876)
- `CRS::extractGeodeticCRS()`: implement for `DerivedProjectedCRS` (#2877)
- Added *proj_trans_bounds()* (#2882)
- CMake: add a `BUILD_APPS` to be able to disable build of all applications (#2895)
- CMake: generate `invproj`/`invgeod` binaries (symlinks on Unix, copy otherwise) (#2897)
- CMake build: add `generate_wkt1_parser` and `generate_wkt2_parser manual` target, and logic to detect when they must be run (#2900)
- Add fallback strategy for tinshift transform to use closest triangle for points not in any (#2907)
- Database: update to EPSG v10.038 (#2910)
- CMake: revise handling of symbol export and static builds (#2912)

  This requires changes for users of static builds on Windows that do not use CMake config files. The empty `PROJ_DLL=` definition must now be defined when building against a static build of PROJ. For users of dynamic builds on Windows, the `PROJ_MSVC_DLL_IMPORT` definition is no longer needed.

### 2.17.3 Bug fixes

- Fix O(n^2) performance patterns where n is the number of steps of a pipeline (#2820)
- Detect ESRI WKT better in certain circumstances (#2823)
- Fix performance issue on pipeline instantiation of huge (broken) pipelines (#2824)

- Make sure to re-order projection parameters according to their canonical order if needed (#2842)
- Fix database access across fork() when SQLite3 doesn't use `pread[64]()` (#2845)
- Fix error in implementation of Inverse ellipsoidal orthographic projection that cause convergence to sometimes fail (#2853)
- Fix handling of edge-case coordinates in inverse ortho ellipsoidal oblique (#2855)
- *proj_normalize_for_visualization()*: set input and output units when there are several alternative transformations (#2867)
- `CRS::identify()`: fix ignoring CS order when identifying a geodetic CRS by a PROJ string with just the ellipsoid (#2881)
- Fix CRS Equality with PROJ parameter order (#2887)
- WKT concatenated operation parsing: fix when a axis order reversal conversion is the first or last operation (#2891)
- WKT1 parser: recognize Lambert_Conformal_Conic as projection name for LCC 1SP or 2SP (#2893)
- CMake: Always build gie if testing is requested (#2899)
- Geographic 3D CRS: allow to export to WKT1:ESRI if only the GEOGCS is known (and thus extrapolating a VERTCS) (#2902)
- `lib_proj.cmake`: add a PROJ::proj alias and add BUILD_INTERFACE include directories, so that proj can be used as a subdirectory of a larger project (#2913)

## 2.18 8.1.1 Release Notes

*September 1st 2021*

### 2.18.1 Updates

- EPSG Database updated to version 10.028 (#2773)

### 2.18.2 Bug Fixes

- Include algorithm header file to avoid build errors on Alpine Linux (#2769)
- CMake: fix installation of executables on iOS (#2766)
- Associate extents to transformations of CRS's that include GEOIDMODEL (#2769)
- Logging: avoid some overhead when logging is not enabled (#2775)
- ortho: remove useless and invalid log trace (#2777)
- CMake: remove external nlohmann_json from INTERFACE_LINK_LIBRARIES target (#2781)
- createOperations(): fix SourceTargetCRSExtentUse::NONE mode (#2783)
- GeoTIFF grid reading: perf improvements (#2788)
- `Conversion::createUTM()`: avoid integer overflow (#2796)
- Inverse laea ellipsoidal: return `PROJ_ERR_COORD_TRANSFM_OUTSIDE_PROJECTION_DOMAIN` when appropriates (#2801)
- Make sure that *proj_crs_promote_to_3D()* returns a derived CRS (#2806)
- `createOperations()`: fix missing deg<-->rad conversion when transforming with a CRS that has a fallback-to-PROJ4-string behavior and is a BoundCRS of a GeographicCRS (#2808)

- WKT2 import/export: preserve PROJ.4 CRS extension string in REMARKS[] (#2812)
- BoundCRS: accept importing/exporting in WKT2 and PROJJSON the scope/area/extent/id attributes (#2815)
- `ConcatenatedOperation::fixStepsDirection()`: fix bad chaining of steps when inverse map projection is involved in non-final step (#2819)

## 2.19 8.1.0 Release Notes

*July 1st 2021*

### 2.19.1 Updates

- **Database**
  - Update to EPSG v10.027 (#2751)
  - Decrease DB size by using `WITHOUT ROWID` tables (#2730) (#2647)
  - Add a `ANALYZE` step during `proj.db` creation allowing for faster lookups (#2729)
  - Added a `PROJ.VERSION` metadata entry (#2646)
  - Added NGO48 (EPSG:4273) to ETRS89 (EPSG:4258) triangulation-based transformation (#2554)
  - Additions to the norwegian NKG2020 transformation (#2548)
  - ESRI projection database updated to version 12.8 (#2717)
- **API additions**
  - Added *proj_get_geoid_models_from_database()* function that returns a list of geoid models available for a given CRS (#2681)
  - Added :c:func`proj_get_celestial_body_list_from_database()` that returns a list of celestial bodies in the PROJ database (#2667)
  - Added *proj_get_celestial_body_name()* (#2662)
- **Various improvements**
  - *proj_trans()*/*cs2cs*: If two operations have the same accuracy, use the one that is contained within a larger one (#2750)
  - Share SQLite database handle among all contexts (#2738)
  - Added `proj/internal/mutex.hpp` as compat layer for mingw32 for std::mutex (#2736)
  - *projsync*: make it filter out files not intended for the current version (#2725)
  - Improvements related to `DerivedVerticalCRS` using Change Unit and Height/Depth reversal methods (#2696)
  - Update internal `nlohmann/json` to 3.9.1, and add a CMake option to be able to use external `nlohmann/json` (#2686)
  - `createFromUserInput()`: change name of CRS built from URN combined references to match the convention of EPSG projected CRS (#2677)
  - Parse compound id with two authorities, like ESRI:103668+EPSG:5703 (#2669)
  - Added *projinfo* option option `--list-crs` (supports `--area`) (#2663)
  - Added support for hyperbolic Cassini-Soldner (#2637)
  - Added capability to get SQL statements to add custom CRS in the database (#2577)

### 2.19.2 Bug fixes

- Fix 'Please include winsock2.h before windows.h' warning with msys (#2692)
- Minor changes to address lint in `geodesic.c` (#2752)
- `BoundCRS::identify()`: avoid incompatible transformation for WKT1 / TOWGS84 export (#2747)
- *`proj_create()`*: do not open proj.db if string is a PROJ string, even if *`proj_context_set_autoclose_database()`* has been set (#2735)
- Fix export of transformation to PROJ string in a particular situation where CompoundCRS are involved (#2721)

## 2.20 8.0.1 Release Notes

*May 5th 2021*

### 2.20.1 Updates

- Database: update to EPSG v10.018 (#2636)
- Add transformations for CHGeo2004, Swiss geoid model (#2604)
- Additions to the norwegian NKG2020 transformation (#2600)

### 2.20.2 Bug fixes

- `pj_vlog()`: fix buffer overflow in case of super lengthy error message (#2693)
- Revert "*`proj_create_crs_to_crs_from_pj()`*: do not use PROJ_SPATIAL_CRITERION_PARTIAL_INTERSECTION if area is specified" (#2679)
- UTM: error out when value of `+zone=` is not an integer (#2672)
- `getCRSInfoList()`: make result order deterministic (by increasing auth_name, code) (#2661)
- `createOperation()`: make sure no to discard deprecated operations if the replacement uses an unknown grid (#2623)
- Fix build on Solaris 11.4 (#2621)
- Add mapping of ESRI Equal_Area projection method to EPSG (#2612)
- Fix incorrect EPGS extent code for EPSG:7789>EPSG:4976 NKG transformation (#2599)
- fix wrong capitalization of CHENyx06_ETRS.gsb (#2597)
- `createOperations()`: improve handling of vertical transforms when when compound CRSs are used (#2592)
- `CRS::promoteTo3D()`: propagate the extent from the 2D CRS (#2589)
- `createFromCRSCodesWithIntermediates()`: improve performance when there is no match (#2583)
- Fix *`proj_clone()`* to work on 'meta' coordinate operation `PJ*` objects that can be returned by *`proj_create_crs_to_crs()`* (#2582)
- add `PROJ_COMPUTE_VERSION`, `PROJ_VERSION_NUMBER`, `PROJ_AT_LEAST_VERSION` macros (#2581)
- Make *`proj_lp_dist()`* and *`proj_geod()`* work on a `PJ*` CRS object (#2570)
- Fix gcc 11 `-Wnonnull` compilation warnings (#2559)
- Fix use of uninitialized memory in gie tests (#2558)
- `createOperations()`: fix incorrect height transformation between 3D promoted RGF93 and CH1903+ (#2555)

## 2.21 8.0.0 Release Notes

*March 1st 2021*

With the release of PROJ 8 the `proj_api.h` API is finally removed. See *Version 4 to 6 API Migration* for more info on how to migrate from the old to the `proj.h` API.

With the removal of `proj_api.h` it has been possible to simplify error codes and messages given by the software. The error codes are exposed in the API.

Several improvements has been made to the command line utilities as well as tweaks in the underlying API.

### 2.21.1 Updates

- Public header file `proj_api.h` removed (#837)
- Improved accuracy of the Mercator projection (#2397)
- Copyright statement wording updated (#2417)
- Allow *cct* to instantiate operations via object codes or names (#2419)
- Allow `@filename` syntax in *cct* (#2420)
- Added *Geocentric to topocentric conversion* (`+proj=topocentric`) (#2444)
- Update GeographicLib to version 1.51 (#2445)
- Added option to allow export of Geographic/Projected 3D CRS in WKT1_GDAL (#2450)
- Added `--area` and `--bbox` options in *cs2cs* to restrict candidate coordinate operations (#2466)
- Added build time option to make `PROJ_LIB` env var tested last (#2476)
- Added `--authority` switch in *cs2cs* to control where coordinate operations are looked for. C API function *proj_create_crs_to_crs_from_pj()* updated accordingly (#2477)
- Error codes revised and exposed in the public API (#2487)
- Added `--accuracy` options to *projinfo*. C API function *proj_create_crs_to_crs_from_pj()* updated accordingly (#2488)
- Added *proj_crs_is_derived()* function to C API (#2496)
- Enabled linking against static cURL on Windows (#2514)
- Updated ESRI CRS database to 12.7 (10.8.1/2.6) (#2519)
- Allow a WKT BoundCRS to use a PROJ string transformation (#2521)
- Update to EPSG v10.015 (#2539)
- Default log level set to `PJ_LOG_ERROR` (#2542)
- CMake installs a pkg-config file `proj.pc`, where supported (#2547)

### 2.21.2 Bug fixes

- Do not restrict longitude to [-90;90] range in spherical transverse Mercator forward projection (#2471)
- `createOperations()`: fix Compound to Geog3D/Projected3D CRS with non-metre ellipsoidal height (#2500)
- Avoid error messages to be emitted log level is set to `PJ_LOG_NONE` (#2527)
- Close database connection when autoclose set to True (#2532)

## 2.22 7.2.1 Release Notes

*January 1st 2021*

### 2.22.1 Updates

- Add metadata with the version number of the database layout (#2474)
- Split `coordinateoperation.cpp` and `test_operation.cpp` in several parts (#2484)
- Update to EPSG v10.008 (#2490)
- Added the NKG 2008 and 2020 transformations in `proj.db` (#2495)

### 2.22.2 Bug fixes

- Set `CURL_ENABLED` definition on projinfo build (#2405)
- `createBoundCRSToWGS84IfPossible()`: make it return same result with a CRS built from EPSG code or WKT1 (#2412)
- WKT2 parsing: several fixes related to map projection parameter units (#2428)
- `createOperation()`: make it work properly when one of the CRS is a BoundCRS of a DerivedGeographicCRS (`+proj=ob_tran +o_proj=lonlat +towgs84=....`) (#2441)
- WKT parsing: fix ingestion of WKT with a Geocentric CRS as the base of the projected CRS (#2443)
- `GeographicCRS::_isEquivalentTo(EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS)`: make it work when comparing easting,northing,up and northing,easting,up (#2446)
- `createOperation()`: add a ballpark vertical transformation when dealing with `GEOIDMODEL[]` (#2449)
- Use same arguments to printf format string for both radians and degrees in output by cct (#2453)
- PRIMEM WKT handling: fixes on import for 'sexagesimal DMS' or from WKT1:GDAL/ESRI when GEOGCS UNIT != Degree; morph to ESRI the PRIMEM name on export (#2455)
- `createObjectsFromName()`: in exact match, make looking for 'ETRS89 / UTM zone 32N' return only the exact match (#2462)
- Inverse tmerc spherical: fix wrong sign of latitude when lat_0 is used (#2469)
- Add option to allow export of Geographic/Projected 3D CRS in WKT1_GDAL (#2470)
- Fix building `proj.db` with SQLite built with `-DSQLITE_DQS=0` (#2480)
- Include JSON Schema files in CMake builds (#2485)
- `createOperations()`: fix inconsistent chaining exception when transforming from BoundCRS of projected CRS based on NTF Paris to BoundCRS of geog CRS NTF Paris (#2486)

## 2.23 7.2.0 Release Notes

*November 1st 2020*

### 2.23.1 Updates

- **Command line tools**
  - Add multi-line PROJ string export capability, and use it by default in *projinfo* (unless `--single-line` is specified) (#2381)

- **Coordinate operations**
  - *Colombia Urban* projection, implementing a EPSG projection method used by a number of projected CRS in Colombia (#2395)
  - *Triangulated Irregular Network based transformation* for triangulation-based transformations (#2344)
  - Added ellipsoidal formulation of *Orthographic* (#2361)
- **Database**
  - Update to EPSG 10.003 and make code base robust to dealing with WKT CRS with DatumEnsemble (#2370)
  - Added Finland tinshift operations (#2392)
  - Added transformation from JGD2011 Geographic 3D to JGD2011 height using GSIGEO2011 (#2393)
  - Improve CompoundCRS identification and name morphing in VerticalCRS with ESRI WKT1 (#2386)
  - Added OGC:CRS27 and OGC:CRS83 CRS entries for NAD27 and NAD83 in longitude, latitude order (#2350)
- **API**
  - Added temporal, engineering, and parametric datum *PJ_TYPE* enumerations (#2274)
  - Various improvements to context handling (#2329, #2331)
  - *proj_create_vertical_crs_ex()*: add a `ACCURACY` option to provide an explicit accuracy, or derive it from the grid name if it is known (#2342)
  - *proj_crs_create_bound_crs_to_WGS84()*: make it work on verticalCRS/compoundCRS such as EPSG:4326+5773 and EPSG:4326+3855 (#2365)
  - `promoteTo3D()`: add a remark with the original CRS identifier (#2369)
  - Added *proj_context_clone()* (#2383)

### 2.23.2 Bug fixes

- Avoid core dumps when copying contexts in certain scenarios (#2324)
- *proj_trans()*: reset errno before attempting a retry with a new coordinate operation (#2353)
- PROJJSON schema corrected to allow prime meridians values with explicitly stating a unit (degrees assumed) (#2354)
- Adjust `createBoundCRSToWGS84IfPossible()` and operation filtering (for POSGAR 2007 to WGS84 issues) (#2357)
- `createOperations()`: several fixes affecting NAD83 -> NAD83(2011) (#2364)
- WKT2:2019 import/export: handle DATUM (at top level object) with PRIMEM
- WKT1_ESRI: fix import and export of CompoundCRS (#2389)

## 2.24 7.1.1 Release Notes

*September 1st 2020*

### 2.24.1 Updates

- Added various Brazilian grids to the database (#2277)
- Added geoid file for Canary Islands to the database (#2312)
- Updated EPSG database to version 9.8.15 (#2310)

### 2.24.2 Bug fixes

- WKT parser: do not raise warning when parsing a WKT2:2015 TIMECRS whose TIMEUNIT is at the CS level, and not inside (#2281)
- Parse '+proj=something_not_latlong +vunits=' without +geoidgrids as a Projected3D CRS and not a compound CRS with a unknown datum (#2289)
- C API: Avoid crashing due to missing SANITIZE_CTX() in entry points (#2293)
- CMake build: Check "target_clones" before use (#2297)
- PROJ string export of +proj=krovak +czech: make sure we export +czech… (#2301)
- Helmert 2D: do not require a useless +convention= parameter (#2305)
- Fix a few spelling errors related to "vgridshift" (#2307)
- Fix ability to identify EPSG:2154 as a candidate for 'RGF93_Lambert_93' (#2316)
- WKT importer: tune for Oracle WKT and 'Lambert Conformal Conic' (#2322)
- Revert compiler generated Fused Multiply Addition optimized routines (#2328)

## 2.25 7.1.0 Release Notes

*July 1st 2020*

### 2.25.1 Updates

- **New transformations**
  - Add a +proj=defmodel transformation for multi-component time-based deformation models (#2206):
- **New projections**
  - Add square conformal projections from libproject (#2148):
    - *Adams Hemisphere in a Square*
    - *Adams World in a Square I*
    - *Adams World in a Square II*
    - *Guyou*
    - *Peirce Quincuncial*
  - Adams Square II: map ESRI WKT to PROJ string, and implement iterative inverse method (#2157)
  - Added *Interrupted Goode Homolosine (Oceanic View)* projection (#2226)
  - Add *Winkel II* inverse by generic inversion of forward method (#2243)
- **Database**
  - Update to EPSG 9.8.12, ESRI 10.8.1 and import scope and remarks for conversion (#2238) (#2267)

  - Map the Behrmann projection to `cae` when converting ESRI CRSes (#1986)
  - Support conversion of Flat_Polar_Quartic projection method (#1987)
  - Register 4 new Austrian height grids (see https://github.com/OSGeo/PROJ-data/pull/13) and handle 'Vertical Offset by Grid Interpolation (BEV AT)' method (#1989)
  - Add ESRI projection method mappings for Mercator_Variant_A, Mercator_Variant_B and Transverse_Cylindrical_Equal_Area and various grid mappings (#2020) (#2195)
  - Map ESRI Transverse_Mercator_Complex to Transverse Mercator (#2040)
  - Register grids for New Caledonia (see https://github.com/OSGeo/PROJ-data/pull/16) (#2051) (#2239)
  - Register NZGD2000 -> ITRF96 transformation for NZGD2000 database (#2248)
  - Register geoid file for UK added (see https://github.com/OSGeo//PROJ-data/pull/25() (#2250)
  - Register Slovakian geoid transformations with needed code changes (#2259)
  - Register Spanish SPED2ETV2 grid for ED50->ETRS89 (#2261)

- **API**
  - Add API function *proj_get_units_from_database()* (#2065)
  - Add API function *proj_get_suggested_operation()* (#2068)
  - Add API functions *proj_degree_input()* and *proj_degree_output()* (#2144)
  - Moved *proj_context_get_url_endpoint()* & *proj_context_get_user_writable_directory()* from `proj_experimental.h` to `proj.h` (#2162)
  - `createFromUserInput()`: allow compound CRS with the 2 parts given by names, e.g. 'WGS 84 + EGM96 height' (#2126)
  - `createOperations()`: when converting CompoundCRS<-->Geographic3DCrs, do not use discard change of ellipsoidal height if a Helmert transformation is involved (#2227)

- **Optimizations**
  - `tmerc/utm`: add a +algo=auto/evenden_snyder/poder_engsager parameter (#2030)
  - Extended `tmerc` (Poder/Engsager): speed optimizations (#2036)
  - Approximate `tmerc` (Snyder): speed optimizations (#2039)
  - `pj_phi2()`: speed-up computation (and thus inverse ellipsoidal Mercator and LCC) (#2052)
  - Inverse `cart`: speed-up computation by 33% (#2145)
  - Extended `tmerc`: speed-up forward path by ~5% (#2147)

- **Various**
  - Follow PDAL's CMake RPATH strategy (#2009)
  - WKT import/export: add support for WKT1_ESRI VERTCS syntax (#2024)
  - *projinfo*: add a `--hide-ballpark` option (#2127)
  - *gie*: implement a strict mode with `<gie-strict> </gie-strict>` (#2168)
  - Allow importing WKT1 COMPD_CS with a VERT_DATUM[Ellipsoid,2002] (#2229)
  - Add runtime checking that sqlite3 is >= 3.11 (#2235)

### 2.25.2 Bug fixes

- `createOperations()`: do not remove ballpark transformation if there are only grid based operations, even if they cover the whole area of use (#2155)
- `createFromProjString()`: handle default parameters of '+krovak +type=crs', and handle `+czech` correctly (#2200)
- `ProjectedCRS::identify()`: fix identification of EPSG:3059 (#2215)
- Database: add a 'WGS84' alias for the EPSG:4326 CRS (#2218)
- Fixes related to CompoundCRS and BoundCRS (#2222)
- Avoid 2 warnings about missing database indices (#2223)
- Make `projinfo --3d --boundcrs-to-wgs84` work better (#2224)
- Many fixes regarding BoundCRS, CompoundCRS, Geographic3D CRS with non-metre units (#2234)
- Fix identification of (one of the) ESRI WKT formulations of EPSG:3035 (#2240)
- Avoid using deprecated and removed Windows API function with Mingw32 (#2246)
- `normalizeForVisualization()`: make it switch axis for EPSG:5482 (RSRGD2000 / RSPS2000) (#2256)
- Fix access violation in *`proj_context_get_database_metadata()`* (#2260)

## 2.26 7.0.1 Release Notes

*May 1st 2020*

### 2.26.1 Updates

- Database: update to EPSG v9.8.9 (#2141)

### 2.26.2 Bug fixes

- Make tests independent of proj-datumgrid (#1995)
- Add missing projection property tables (#1996)
- Avoid crash when running against SQLite3 binary built with `-DSQLITE_OMIT_AUTOINIT` (#1999)
- `createOperations()`: fix wrong pipeline generation with CRS that has `+nadgrids=` and `+pm=` (#2002)
- Fix bad copy&replace pattern on HEALPix and rHEALPix projection names (#2007)
- `createUnitOfMeasure()`: use full double resolution for the conversion factor (#2014)
- Update README with info on PROJ-data (#2015)
- `utm/ups`: make sure to set errno to `PJD_ERR_ELLIPSOID_USE_REQUIRED` if `+es==0` (#2045)
- `data/Makefile.am`: remove bashism (#2048)
- `ProjectedCRS::identify()`: tune it to better work with ESRI WKT representation of EPSG:2193 (#2059)
- Fix build with gcc 4.8.5 (#2066)
- Autotools/pkg-conf: Define datarootdir (#2069)
- *cs2cs*: don't require `+to` for '{source_crs} {target_crs} filename...' syntax (#2081)
- CMake: fix bug with `find_package(PROJ)` with macOS (#2082)

- ESRI WKT import / identification: special case for NAD_1983_HARN_StatePlane_Colorado_North_FIPS_0501 with Foot_US unit (#2088)
- ESRI WKT import / identification: special case for NAD_1983_HARN_StatePlane_Colorado_North_FIPS_0501 with Foot_US unit (#2089)
- EngineeringCRS: when exporting to WKT1_GDAL, output unit and axis (#2092)
- Use jtsk03-jtsk horizontal grid from CDN (#2098)
- CMake: prefer to use use PROJ_SOURCE_DIR and PROJ_BINARY_DIR (#2100)
- Fix wrong grids file name in esri.sql (#2104)
- Fix identification of projected CRS whose name is close but not strictly equal to a ESRI alias (#2106)
- Fix working of Helmert transform between the horizontal part of 2 compoundCRS (#2111)
- Database: fix registration of custom entries of grid_transformation_custom.sql for geoid grids (#2114)
- ESRI_WKT ingestion: make sure to identify to non-deprecated EPSG entry when possible (#2119)
- Make sure that importing a Projected 3D CRS from WKT:2019 keeps the base geographic CRS as 3D (#2125)
- `createOperations()`: improve results of compoundCRS to compoundCRS case (#2131)
- hgridshift/vgridshift: defer grid opening when grid has already been opened (#2132)
- Resolve a few shadowed declaration warnings (#2142)
- ProjectedCRS identification: deal with switched 1st/2nd std parallels for LCC_2SP(#2153)
- Fix Robinson inverse projection (#2154)
- `createOperations()`: do not remove ballpark transformation if there are only grid based operations, even if they cover the whole area of use (#2156)
- `createFromCoordinateReferenceSystemCodes()`: 'optimization' to avoid using C++ exceptions (#2161)
- Ingestion of WKT1_GDAL: correctly map 'Cylindrical_Equal_Area' (#2167)
- Add limited support for non-conformant WKT1 LAS COMPD_CS[] (#2172)
- PROJ4 string import: take into correctly non-metre unit when the string looks like the one for WGS 84 / Pseudo Mercator (#2177)
- `io.hpp`: avoid dependency to `proj_json_streaming_writer.hpp` (#2184)
- Fix support of WKT1_GDAL with netCDF rotated pole formulation (#2186)

## 2.27 6.3.2 Release Notes

*May 1st 2020*

### 2.27.1 Bug fixes

- `validateParameters()`: fix false-positive warning on Equidistant Cylindrical (#1947)
- `proj_create_crs_to_cr()`: avoid potential reprojection failures when reprojecting area of use to source and target CRS (#1993)
- `createOperations()`: fix wrong pipeline generation with CRS that has `+nadgrids=` and `+pm=` (#2003)
- Fix bad copy&replace pattern on HEALPix and rHEALPix projection names (#2006)
- `createUnitOfMeasure()`: use full double resolution for the conversion factor (#2013)

- `data/Makefile.am`: remove bashism (#2047)
- :cpp:func:`ProjectedCRS::identify`: tune it to better work with ESRI WKT representation of EPSG:2193 (#2058)
- EngineeringCRS: when exporting to WKT1_GDAL, output unit and axis (#2091)
- Add missing entries in grid_alternatives for Portugal grids coming from ESRI entries (#2103)
- Fix working of Helmert transform between the horizontal part of 2 compoundCRS (#2110)
- ESRI_WKT ingestion: make sure to identify to non-deprecated EPSG entry when possible (#2118)
- Make sure that importing a Projected 3D CRS from WKT:2019 keeps the base geographic CRS as 3D (#2124)
- `createOperations()`: improve results of compoundCRS to compoundCRS case (#2130)
- PROJ4 string import: take into correctly non-metre unit when the string looks like the one for WGS 84 / Pseudo Mercator (#2178)
- Fix support of WKT1_GDAL with netCDF rotated pole formulation (#2187)
- `io.hpp`: avoid dependency to `proj_json_streaming_writer.hpp` (#2188)

## 2.28 7.0.0 Release Notes

*March 1st 2020*

The major feature in PROJ 7 is significantly improved handling of gridded models. This was implemented in *PROJ RFC 4: Remote access to grids and GeoTIFF grids*. The main features of the RFC4 work is that PROJ now implements a new grid format, Geodetic TIFF grids, for exchanging gridded transformation models. In addition to the new grid format, PROJ can now also access grids online using a data store in the cloud.

The grids that was previously available via the proj-datumgrid packages are now available in two places:

1. As a single combined data archive including all available resource files
2. From the cloud via https://cdn.proj.org

In Addition, provided with PROJ is a utility called *projsync* that can be used download grids from the data store in the cloud.

The use of the new grid format and the data from the cloud requires that PROJ is build against `libtiff` and `libcurl`. Both are optional dependencies to PROJ but it is highly encouraged that the software is build against both.

**Warning**

PROJ 7 will be last major release version that includes the `proj_api.h` header. The functionality in `proj_api.h` is deprecated and only supported in maintenance mode. It is inferior to the functionality provided by functions in the `proj.h` header and all projects still relying on `proj_api.h` are encouraged to migrate to the new API in `proj.h`. See *Version 4 to 6 API Migration*. for more info on how to migrate from the old to the new API.

### 2.28.1 Updates

- Added new file access API to `proj.h` (#866)
- Updated the name of the most recent version of the WKT2 standard from WKT2_2018 to WKT2_2019 to reflect the proper name of the standard (#1585)
- Improvements in transformations from/to WGS 84 (Gxxxx) realizations and vertical <--> geog transformations (#1608)

- Update to version 1.50 of the geodesic library (#1629)
- Promote `proj_assign_context()` to `proj.h` from `proj_experimental.h` (#1630)
- Add rotation support to the HEALPix projection (#1638)
- Add C function *proj_crs_create_bound_vertical_crs()* (#1689)
- Use Win32 Unicode APIs and expect all strings to be UTF-8 (#1765)
- Improved name aliases lookup (#1827)
- CMake: Employ better use of CTest with the `BUILD_TESTING` option (#1870)
- Grid correction: fix handling grids spanning antimeridian (#1882)
- Remove legacy CMake target name `proj` (#1883)
- *projinfo* add `--searchpaths` switch (#1892)
- Add *+proj=set operation* to set component(s) of a coordinate to a fixed value (#1896)
- Add EPSG records for 'Geocentric translation by Grid Interpolation (IGN)' (`gr3df97a.txt`) and map them to new *+proj=xyzgridshift* (#1897)
- Remove `null` grid file as it is now a special hardcoded case in grid code (#1898)
- Add *projsync* utility (#1903)
- Make `PROJ` the CMake project name (#1910)
- Use relative directory to locate PROJ resource files (#1921)

### 2.28.2 Bug fixes

- Horizontal grid shift: fix failures on points slightly outside a subgrid (#209)
- Fix ASAN issue with SQLite3VFS class (#1902)
- tests: force use of bash for `proj_add_test_script_sh` (#1905)

### 2.28.3 Breaking changes

- Reject NTV2 files where `GS_TYPE != SECONDS` (#1294)
- On Windows the name of the library is now fixed to `proj.lib` instead of encoding the version number in the library name (#1581)
- Require C99 compiler (#1624)
- Remove deprecated JNI bindings (#1825)
- Remove -ld option from *proj* and *cs2cs* (#1844)
- Increase CMake minimum version from 3.5 to 3.9 (#1907)

## 2.29 6.3.1 Release Notes

*February 11th 2020*

### 2.29.1 Updates

- Update the EPSG database to version 9.8.6
- Database: add mapping for gg10_smv2.mnt and gg10_sbv2.mnt French grids
- Database: add mapping for TOR27CSv1.GSB

### 2.29.2 Bug fixes

- Fix wrong use of derivingConversionRef() that caused issues with use of +init=epsg:XXXX by GDAL (affecting R spatial libraries) or in MapServer
- fix exporting CoordinateSystem to PROJ JSON with ID
- projinfo: use No. abbreviation instead of UTF-8 character (#1828)
- CompoundCRS::identify(): avoid exception when horiz/vertical part is a BoundCRS
- createOperations(): fix dealing with projected 3D CRS whose Z units != metre
- WKT1_GDAL export: limit datum name massaging to names matching EPSG (#1835)
- unitconvert with mjd time format: avoid potential integer overflow (ossfuzz 20072)
- ProjectedCRS::identify(): fix wrong identification of some ESRI WKT linked to units
- Database: add a geoid_like value for proj_method column of grid_alternatives, fix related entries and simplify/robustify logic to deal with EPSG 'Geographic3D to GravityRelatedHeight' methods
- Fix ingestion of +proj=cea with +k_0 (#1881)
- Fix performance issue, affecting PROJ.4 string generation of EPSG:7842 (#1913)
- Fix identification of ESRI-style datum names starting with D_ but without alias (#1911)
- cart: Avoid discontinuity at poles in the inverse case (#1906)
- Various updates to make regression test suite pass with gcc on i386 (#1906)

## 2.30 6.3.0 Release Notes

*January 1st 2020*

### 2.30.1 Updates

- Database: tune accuracy of Canadian NTv1 file w.r.t NTv2 (#1812)
- Modify verbosity level of some debug/trace messages (#1811)
- *projinfo*: no longer call createBoundCRSToWGS84IfPossible() for WKT1:GDAL (#1810)
- `proj_trans()`: add retry logic to select other transformation if the best one fails. (#1809)
- *BoundCRS::identify()*: improvements to discard CRS that aren't relevant (#1802)
- Database: update to IGNF v3.1.0 (#1785)
- Build: Only export symbols if building DLL (#1773)
- Database: update ESRI entries with ArcGIS Desktop version 10.8.0 database (#1762)
- `createOperations()`: chain operations whose middle CRSs are not identical but have the same datum (#1734)
- import/export PROJJSON: support a interpolation_crs key to geoid_model (#1732)

- Database: update to EPSG v9.8.4 (#1725)
- Build: require SQLite 3.11 (#1721)
- Add support for GEOIDMODEL (#1710)
- Better filtering based on extent and performance improvements (#1709)

### 2.30.2 Bug fixes

- Horizontal grid shift: fix issue on iterative inverse computation when switching between (sub)grids (#1797)
- `createOperations()`: make filtering out of 'uninteresting' operations less aggressive (#1788)
- Make EPSG:102100 resolve to ESRI:102100 (#1786)
- `ob_tran`: restore traditional handling of `+to_meter` with `pj_transform()` and *proj* utility (#1783)
- CRS identification: use case insensitive comparison for authority name (#1780)
- `normalizeForVisualization()` and other methods applying on a ProjectedCRS: do not mess the deriving-Conversion object of the original object (#1746)
- `createOperations()`: fix transformation computation from/to a CRS with `+geoidgrids` and `+vunits` != m (#1731)
- Fix `proj_assign_context()`/`pj_set_ctx()` with pipelines and alternative coord operations (#1726)
- Database: add an auxiliary concatenated_operation_step table to allow arbitrary number of steps (#1696)
- Fix errors running gie-based tests in Debug mode on Window (#1688)

## 2.31 6.2.1 Release Notes

*November 1st 2019*

### 2.31.1 Updates

- Update the EPSG database to version 9.8.2

### 2.31.2 Bug fixes

- Fixed erroneous spelling of "Potsdam" (#1573)
- Calculate y-coordinate correctly in *Bertin 1953* in all cases (#1579)
- *proj_create_crs_to_crs_from_pj()*: make the PJ* arguments const PJ* (#1583)
- *PROJStringParser::createFromPROJString()*: avoid potential infinite recursion (#1574)
- Avoid core dump when setting `ctx==NULL` in functions *proj_coordoperation_is_instantiable()* and *proj_coordoperation_has_ballpark_transformation()* (#1590)
- `createOperations()`: fix conversion from/to PROJ.4 CRS strings with non-ISO-kosher options and `+towgs84`/`+nadgrids` (#1602)
- *proj_trans_generic()*: properly set coordinate time to `HUGE_VAL` when no value is passed to the function (#1604)
- Fix support for `+proj=ob_tran +o_proj=lonlat/latlong/latlon` instead of only only allowing `+o_proj=longlat` (#1601)
- Improve backwards compatibility of vertical transforms (#1613)

- Improve emulation of deprecated `+init` style initialization (#1614)
- *cs2cs*: autopromote CRS to 3D when there's a mix of 2D and 3D (#1563)
- Avoid divisions by zero in odd situations (#1620)
- Avoid compile error on Solaris (#1639)
- *proj_create_crs_to_crs()*: fix when there are only transformations with ballpark steps (#1643)
- PROJ string CRS ingester: recognize more unit-less parameters, and general handling of `+key=string_value` parameters (#1645)
- Only call pkg-config in configure when necessary (#1652)
- *Azimuthal Equidistant*: for spherical forward path, go to higher precision ellipsoidal case when the point coordinates are super close to the origin (#1654)
- *proj_create_crs_to_crs()*: remove elimination of Ballpark operations that caused transformation failures in some cases (#1665)
- `createOperations()`: allow transforming from a compoundCRS of a bound verticalCRS to a 2D CRS (#1667)
- Avoid segfaults in case of out-of-memory situations (#1679)
- `createOperations()`: fix double vertical unit conversion from CompoundCRS to other CRS when the horizontal part of the projected CRS uses non-metre unit (#1683)(#1683)
- `importFromWkt()`: fix axis orientation for non-standard ESRI WKT (#1690)

## 2.32 6.2.0 Release Notes

*September 1st 2019*

### 2.32.1 Updates

- Introduced *PROJJSON*, a JSON encoding of WKT2 (#1547)
- Support CRS instantiation of OGC URN's (#1505)
- Expose scope and remarks of database objects (#1537)
- EPSG Database updated to version 9.7.0 (#1558)
- Added C API function proj_grid_get_info_from_database() (#1494)
- Added C API function proj_operation_factory_context_set_discard_superseded() (#1534)
- Added C API function proj_context_set_autoclose_database() (#1566)
- Added C API function proj_create_crs_to_crs_from_pj() (#1567)
- Added C API function proj_cleanup() (#1569)

### 2.32.2 Bug Fixes

- Fixed build failure on Solaris systems (#1554)

## 2.33 6.1.1 Release Notes

*July 1st 2019*

### 2.33.1 Updates

- Update EPSG registry to version 9.6.3 (#1485)

### 2.33.2 Bug Fixes

- Take the passed authority into account when identifying objects (#1466)
- Avoid exception when transforming from NAD83 to projected CRS using NAD83(2011) (#1477)
- Avoid off-by-one reading of name argument if name of resource file has length 1 (#11489)
- Do not include `PROJ_LIB` in `proj_info().searchpath` when context search path is set (#1498)
- Use correct delimiter for the current platform when parsing PROJ_LIB (#1497)
- Do not confuse 'ID74' CRS with WKT2 ID[] node (#1506)
- WKT1 importer: do case insensitive comparison for axis direction (#1509)
- Avoid compile errors on GCC 4.9.3 (#1512)
- Make sure that pipelines including `+proj=ob_tran` can be created (#1526)

## 2.34 6.1.0 Release Notes

*May 15th 2019*

### 2.34.1 Updates

- Include custom ellipsoid definitions from QGIS (#1137)
- Add `-k ellipsoid` option to projinfo (#1338)
- Make cs2cs support 4D coordinates (#1355)
- WKT2 parser: update to OGC 18-010r6 (#1360 #1366))
- Update internal version of googletest to v1.8.1 (#1361)
- Database update: EPSG v9.6.2 (#1462), IGNF v3.0.3, ESRI 10.7.0 and add operation_version column (#1368)
- Add *`proj_normalize_for_visualization()`* that attempts to apply axis ordering as used by most GIS applications and PROJ <6 (#1387)
- Added noop operation (#1391)
- Paths set by user take priority over `PROJ_LIB` for search paths (#1398)
- Reduced database size (#1438)
- add support for compoundCRS and concatenatedOperation named from their components (#1441)

### 2.34.2 Bug fixes

- Have *gie* return non-zero code when file can't be opened (#1312)
- CMake cross-compilation fix (#1316)
- Use 1st eccentricity instead of 2nd eccentricity in Molodensky (#1324)
- Make sure to include grids when doing Geocentric to CompoundCRS with nadgrids+geoidgrids transformations (#1326)
- Handle coordinates outside of bbox better (#1333)

- Enable system error messages in command line automatically in builds (#1336)
- Make sure to install projinfo man page with CMake (#1347)
- Add data dir to pkg-config file proj.pc (#1348)
- Fix GCC 9 warning about useless `std::move()` (#1352)
- Grid related fixes (#1369)
- Make sure that ISO19111 C++ code sets pj_errno on errors (#1405)
- vgridshift: handle longitude wrap-around for grids with 360deg longitude extent (#1429)
- *proj*/*cs2cs*: validate value of `-f` parameter to avoid potential crashes (#1434)
- Many division by zero and similar bug fixes found by OSS Fuzz.

## 2.35 6.0.0 Release Notes

*March 1st 2019*

PROJ 6 has undergone extensive changes to increase its functional scope from a cartographic projection engine with so-called "early-binding" geodetic datum transformation capabilities to a more complete library supporting coordinate transformations and coordinate reference systems.

As a foundation for other enhancements, PROJ now includes a C++ implementation of the modelisation proposed by the ISO-19111:2019 standard / OGC Abstract Specification Topic 2: "Referencing By Coordinates", for geodetic reference frames (datums), coordinate reference systems and coordinate operations. Construction and query of those geodetic objects is available through a new C++ API, and also accessible for the most part from bindings in the C API.

Those geodetic objects can be imported and exported from and into the OGC Well-Known Text format (WKT) in its different variants: ESRI WKT, GDAL WKT 1, WKT2:2015 (ISO 19162:2015) and WKT2:2018 (ISO 19162:2018). Import and export of CRS objects from and into PROJ strings is also supported. This functionality was previously available in the GDAL software library (except WKT2 support which is a new feature), and is now an integral part of PROJ.

A unified database of geodetic objects, coordinate reference systems and their metadata, and coordinate operations between those CRS is now available in a SQLite3 database file, proj.db. This includes definitions imported from the IOGP EPSG dataset (v9.6.0 release), the IGNF (French national mapping agency) geodetic registry and the ESRI projection engine database. PROJ is now the reference software in the "OSGeo C stack" for this CRS and coordinate operation database, whereas previously this functionality was spread over PROJ, GDAL and libgeotiff, and used CSV or other ad hoc text-based formats.

Late-binding coordinate operation capabilities, that takes metadata such as area of use and accuracy into account, has been added. This can avoid in a number of situations the past requirement of using WGS84 as a pivot system, which could cause unneeded accuracy loss, or was not doable at all sometimes when transformation to WGS84 was not available. Those late-binding capabilities are now used by the proj_create_crs_to_crs() function and the cs2cs utility.

A new command line utility, projinfo, has been added to query information about a geodetic object of the database, import and export geodetic objects from/into WKT and PROJ strings, and display coordinate operations available between two CRSs.

### 2.35.1 Updates

- Removed projects.h as a public interface (#835)
- Deprecated the proj_api.h interface. The header file is still available but will be removed with the next major version release of PROJ. It is now required to define `ACCEPT_USE_OF_DEPRECATED_PROJ_API_H` before the interface can be used (#836)

- Removed support for the nmake build system (#838)
- Removed support for the `proj_def.dat` defaults file (#201)
- C++11 required for building PROJ (#1203)
- Added build dependency on SQLite 3.7 (#1175)
- Added *projinfo* command line application (#1189)
- Added many functions to `proj.h` for handling ISO19111 functionality (#1175)
- Added C++ API exposing ISO19111 functionality (#1175)
- Updated *cs2cs* to use late-binding features (#1182)
- Removed the `nad2bin` application. Now available in the proj-datumgrid git repository (#1236)
- Removed support for Chebyshev polynomials in *proj* (#1226)
- Removed `proj_geocentric_latitude()` from *proj.h* API (#1170)
- Changed behavior of *proj*: Now only allow initialization of projections (#1162)
- Changed behavior of *tmerc*: Now defaults to the Extended Transverse Mercator algorithm (`etmerc`). Old implementation available by adding `+approx` (#404)
- Changed behavior: Default ellipsoid now set to GRS80 (was WGS84) (#1210)
- Allow multiple directories in `PROJ_LIB` environment variable (#1281)
- Added *Lambert Conic Conformal (2SP Michigan)* projection (#1142)
- Added *Bertin1953* projection (#1133)
- Added *Tobler-Mercator* projection (#1153)
- Added *Molodensky-Badekas* transform (#1160)
- Added *push* and *pop* coordinate operations (#1250)
- Removed `+t_obs` parameter from helmert and deformation (#1264)
- Added *+dt* parameter to deformation as replacement for removed `+t_obs` (#1264)

### 2.35.2 Bug fixes

- Read *+towgs84* values correctly on locales not using dot as comma separator (#1136)
- Fixed file offset for reading of shift values in NTv1 files (#1144)
- Avoid problems with `PTHREAD_MUTEX_RECURSIVE` when using CMake (#1158)
- Avoid raising errors when setting ellipsoid flattening to zero (#1191)
- Fixed lower square calculations in *rHealpix* projection (#1206)
- Allow *Molodensky* transform parameters to be zero (#1194)
- Fixed wrong parameter in `ITRF2000` init file (#1240)
- Fixed use of grid paths including spaces (#1152)
- *Robinson*: fix wrong values for forward path for latitudes >= 87.5, and fix inaccurate inverse method (#1172)

## 2.36 PROJ 5.2.0

*September 15th 2018*

### 2.36.1 Updates

- Added support for deg, rad and grad in unitconvert (#1054)
- Assume `+t_epoch` as time input when not otherwise specified (#1065)
- Added inverse Lagrange projection (#1058)
- Added `+multiplier` option to vgridshift (#1072)
- Added Equal Earth projection (#1085)
- Added "require_grid" option to gie (#1088)
- Replace `transpose` option of Helmert transform with `+convention`. From now on the convention used should be explicitly written. An error will be returned when using the +transpose option (#1091)
- Improved numerical precision of inverse spherical Mercator projection (#1105)
- *cct* will now forward text after coordinate input to output stream (#1111)

### 2.36.2 Bug fixes

- Do not pivot over WGS84 when doing cs2cs-emulation with geocent (#1026)
- Do not scan past the end of the read data in `pj_ctx_fgets()` (#1042)
- Make sure `proj_errno_string()` is available in DLL (#1050)
- Respect +*to_meter* setting when doing cs2cs-emulation (#1053)
- Fixed unit conversion factors for *geod* (#1075)
- Fixed test failures related to GCC 8 (#1084)
- Improved handling of +*geoc* flag (#1093)
- Calculate correct projection factors for Webmercator (#1095)
- *cs2cs* now always outputs degrees when transformed coordinates are in angular units (#1112)

## 2.37 PROJ 5.1.0

*June 1st 2018*

### 2.37.1 Updates

- Function `proj_errno_string()` added to `proj.h` API (#847)
- Validate units between pipeline steps and ensure transformation sanity (#906)
- Print help when calling *cct* and *gie* without arguments (#907)
- *CITATION* file added to source distribution (#914)
- Webmercator operation added (#925)
- Enhanced numerical precision of forward spherical Mercator near the Equator (#928)
- Added `--skip-lines` option to *cct* (#923)

- Consistently return `NaN` values on `NaN` input (#949)
- Removed unused `src/org_proj4_Projections.h` file (#956)
- Java Native Interface bindings updated (#957, #969)
- Horizontal and vertical gridshift operations extended to the temporal domain (#1015)

### 2.37.2 Bug fixes

- Handle `NaN` float cast overflow in `PJ_robin.c` and `nad_intr.c` (#887)
- Avoid overflow when Horner order is unreasonably large (#893)
- Avoid unwanted NaN conversions in etmerc (#899)
- Avoid memory failure in *gie* when not specifying x,y,z in gie files (#902)
- Avoid memory failure when *+sweep* is initialized incorrectly in geos (#908)
- Return `HUGE_VAL` on erroneous input in ortho (#912)
- Handle commented lines correctly in cct (#933)
- Avoid segmentation fault when transformation coordinates outside grid area in deformation (#934)
- Avoid doing false easting/northing adjustments on cartesian coordinates (#936)
- Thread-safe creation of proj mutex (#954)
- Avoid errors when setting up geos with +lat_0!=0 (#986)
- Reset errno when running *proj* in verbose mode (#988)
- Do not interpolate node values at nodata value in vertical grid shifts (#1004)
- Restrict Horner degrees to positive integer values to avoid memory allocation issues (#1005)

## 2.38 PROJ 5.0.1

*March 1st 2018*

### 2.38.1 Bug fixes

- Handle ellipsoid change correctly in pipelines when `+towgs84=0,0,0` is set (#881)
- Handle the case where nad_ctable2_init returns NULL (#883)
- Avoid shadowed declaration errors with old gcc (#880)
- Expand `+datum` properly in pipelines (#872)
- Fail gracefully when incorrect headers are encountered in grid files (#875)
- Improve roundtrip stability in pipelines using `+towgs84` (#871)
- Fixed typo in gie error codes (#861)
- Numerical stability fixes to the geodesic package (#826 & #843)
- Make sure that transient errors are returned correctly (#857)
- Make sure that locally installed header files are not used when building PROJ (#849)
- Fix inconsistent parameter names in `proj.h/proj_4D_api.c` (#842)
- Make sure `+vunits` is applied (#833)

- Fix incorrect Web Mercator transformations (#834)

## 2.39 PROJ 5.0.0

*February 1st 2018*

This version of PROJ introduces some significant extensions and improvements to (primarily) the geodetic functionality of the system.

The main driver for introducing the new features is the emergence of dynamic reference frames, the increasing use of high accuracy GNSS, and the related growing demand for accurate coordinate transformations. While older versions of PROJ included some geodetic functionality, the new framework lays the foundation for turning PROJ into a generic geospatial coordinate transformation engine.

The core of the library is still the well established projection code. The new functionality is primarily exposed in a new programming interface and a new command line utility, *cct* (for "Coordinate Conversion and Transformation"). The old programming interface is still available and can - to some extent - use the new geodetic transformation features.

The internal architecture has also seen many changes and much improvement. So far, these improvements respect the existing programming interface. But the process has revealed a need to simplify and reduce the code base, in order to support sustained active development.

**Therefore we have scheduled regular releases over the coming years which will gradually remove the old programming interface.**

**This will cause breaking changes with the next two major version releases, which will affect all projects that depend on PROJ (cf. section "deprecations" below).**

The decision to break the existing API has not been easy, but has ultimately been deemed necessary to ensure the long term survival of the project. Not only by improving the maintainability immensely, but also by extending the potential user (and hence developer) community.

The end goal is to deliver a generic coordinate transformation software package with a clean and concise code base appealing to both users and developers.

### 2.39.1 Versioning and naming

For the first time in more than 25 years the major version number of the software is changed. The decision to do this is based on the many new features and new API. While backwards compatibility remains - except in a few rare corner cases - the addition of a new and improved programming interface warrants a new major release.

The new major version number unfortunately leaves the project in a bit of a conundrum regarding the name. For the majority of the life-time of the product it has been known as PROJ.4, but since we have now reached version 5 the name is no longer aligned with the version number.

Hence we have decided to decouple the name from the version number and from this version and onwards the product will simply be called PROJ.

In recognition of the history of the software we are keeping PROJ.4 as the *name of the organizing project*. The same project team also produces the datum-grid package.

In summary:

- The PROJ.4 project provides the product PROJ, which is now at version 5.0.0.
- The foundational component of PROJ is the library libproj.
- Other PROJ components include the application proj, which provides a command line interface to libproj.
- The PROJ.4 project also distributes the datum-grid package, which at the time of writing is at version 1.6.0.

### 2.39.2 Updates

- Introduced new API in `proj.h`.
  - The new API is orthogonal to the existing `proj_api.h` API and the internally used `projects.h` API.
  - The new API adds the ability to transform spatiotemporal (4D) coordinates.
  - Functions in the new API use the `proj_` namespace.
  - Data types in the new API use the `PJ_` namespace.
- Introduced the concept of "transformation pipelines" that makes possible to do complex geodetic transformations of coordinates by daisy chaining simple coordinate operations.
- Introduced *cct*, the Coordinate Conversion and Transformation application.
- Introduced *gie*, the Geospatial Integrity Investigation Environment.
  - Selftest invoked by `-C` flag in *proj* has been removed
  - Ported approx. 1300 built-in selftests to *gie* format
  - Ported approx. 1000 tests from the gigs test framework
  - Added approx. 200 new tests
- Adopted terminology from the OGC/ISO-19100 geospatial standards series. Key definitions are:
  - At the most generic level, a *coordinate operation* is a change of coordinates, based on a one-to-one relationship, from one coordinate reference system to another.
  - A *transformation* is a coordinate operation in which the two coordinate reference systems are based on different datums, e.g. a change from a global reference frame to a regional frame.
  - A *conversion* is a coordinate operation in which both coordinate reference systems are based on the same datum, e.g. change of units of coordinates.
  - A *projection* is a coordinate conversion from an ellipsoidal coordinate system to a plane. Although projections are simply conversions according to the standard, they are treated as separate entities in PROJ as they make up the vast majority of operations in the library.
- New operations
  - *The pipeline operator* (`pipeline`)
  - **Transformations**
    - *Helmert transform* (`helmert`)
    - Horner real and complex polynomial evaluation (`horner`)
    - *Horizontal gridshift* (`hgridshift`)
    - *Vertical gridshift* (`vgridshift`)
    - *Molodensky transform* (`molodensky`)
    - *Kinematic gridshift with deformation model* (`deformation`)
  - **Conversions**
    - *Unit conversion* (`unitconvert`)
    - *Axis swap* (`axisswap`)
  - **Projections**
    - *Central Conic projection* (`ccon`)

- Significant documentation updates, including
  - Overhaul of the structure of the documentation
  - A better introduction to the use of PROJ
  - *A complete reference to the new API*
  - a complete rewrite of the section on geodesic calculations
  - Figures for all projections
- New "free format" option for operation definitions, which permits separating tokens by whitespace when specifying key/value- pairs, e.g. `proj = merc lat_0 = 45`.
- Added metadata to init-files that can be read with the *`proj_init_info()`* function in the new `proj.h` API.
- Added ITRF2000, ITRF2008 and ITRF2014 init-files with ITRF transformation parameters, including plate motion model parameters.
- Added ellipsoid parameters for GSK2011, PZ90 and "danish". The latter is similar to the already supported andrae ellipsoid, but has a slightly different semimajor axis.
- Added Copenhagen prime meridian.
- Updated EPSG database to version 9.2.0.
- Geodesic library updated to version 1.49.2-c.
- Support for analytical partial derivatives has been removed.
- Improved performance in Winkel Tripel and Aitoff.
- Introduced `pj_has_inverse()` function to `proj_api.h`. Checks if an operation has an inverse. Use this instead of checking whether `P->inv` exists, since that can no longer be relied on.
- ABI version number updated to 13:0:0.
- Removed support for Windows CE.
- Removed the VB6 COM interface.

### 2.39.3 Bug fixes

- Fixed incorrect convergence calculation in Lambert Conformal Conic. (#16)
- Handle ellipsoid parameters correctly when using `+nadgrids=@null`. (#22)
- Return correct latitude when using negative northings in Transverse Mercator. (#138)
- Return correct result at origin in inverse Mod. Stereographic of Alaska. (#161)
- Return correct result at origin in inverse Mod. Stereographic of 48 U.S. (#162)
- Return correct result at origin in inverse Mod. Stereographic of 50 U.S. (#163)
- Return correct result at origin in inverse Lee Oblated Stereographic. (#164)
- Return correct result at origin in inverse Miller Oblated Stereographic. (#165)
- Fixed scaling and wrap-around issues in Oblique Cylindrical Equal Area. (#166)
- Corrected a coefficient error in inverse Transverse Mercator. (#174)
- Respect `-r` flag when calling *proj* with `-V`. (#184)
- Remove multiplication by 2 at the equator error in Stereographic projection. (#194)
- Allow +alpha=0 and +gamma=0 when using Oblique Mercator. (#195)

- Return correct result of inverse Oblique Mercator when alpha is between 90 and 270. (#331)
- Avoid segmentation fault when accessing point outside grid. (#396)
- Avoid segmentation fault on NaN input in Robin inverse. (#463)
- Very verbose use of *proj* (`-V`) on Windows is fixed. (#484)
- Fixed memory leak in General Oblique Transformation. (#497)
- Equations for meridian convergence and partial derivatives have been corrected for non-conformal projections. (#526)
- Fixed scaling of cartesian coordinates in `pj_transform()`. (#726)
- Additional bug fixes courtesy of Google's OSS-Fuzz program

CHAPTER

THREE

# DOWNLOAD

Source code for the current and previous releases can be downloaded from https://download.osgeo.org/proj/

Documentation in PDF and zipped HTML can be downloaded using the flyout menu on the left side of the browser window from https://proj.org/

## 3.1 Current Release

- **2025-06-06** proj-9.6.2.tar.gz (md5)
- **2025-06-02** proj-data-1.22.tar.gz

> **Note**
>
> The proj-datumgrid packages have been deprecated with PROJ 7.0.0. The proj-data package should be used with PROJ 7.0.0 and newer
>
> The proj-datumgrid packages should be used with PROJ releases from the 5.x and 6.x branches.

## 3.2 Past Releases

- **2025-06-02** proj-9.6.1.tar.gz
- **2025-03-15** proj-9.6.0.tar.gz
- **2024-12-01** proj-9.5.1.tar.gz
- **2024-09-15** proj-9.5.0.tar.gz
- **2024-06-01** proj-9.4.1.tar.gz
- **2024-03-01** proj-9.4.0.tar.gz
- **2023-12-01** proj-9.3.1.tar.gz
- **2023-09-01** proj-9.3.0.tar.gz
- **2023-06-01** proj-9.2.1.tar.gz
- **2023-03-01** proj-9.2.0.tar.gz
- **2022-12-01** proj-9.1.1.tar.gz
- **2022-09-01** proj-9.1.0.tar.gz
- **2022-06-15** proj-9.0.1.tar.gz
- **2022-03-01** proj-9.0.0.tar.gz

- **2022-01-01** proj-8.2.1.tar.gz
- **2021-11-01** proj-8.2.0.tar.gz
- **2021-09-01** proj-8.1.1.tar.gz
- **2021-07-01** proj-8.1.0.tar.gz
- **2021-05-05** proj-8.0.1.tar.gz
- **2021-03-01** proj-8.0.0.tar.gz
- **2021-01-01** proj-7.2.1.tar.gz
- **2020-11-01** proj-7.2.0.tar.gz
- **2020-09-01** proj-7.1.1.tar.gz
- **2020-07-01** proj-7.1.0.tar.gz
- **2020-05-01** proj-6.3.2.tar.gz
- **2020-05-01** proj-7.0.1.tar.gz
- **2020-03-01** proj-7.0.0.tar.gz
- **2020-02-11** proj-6.3.1.tar.gz
- **2020-01-01** proj-6.3.0.tar.gz
- **2019-11-01** proj-6.2.1.tar.gz
- **2019-09-01** proj-6.2.0.tar.gz
- **2019-07-01** proj-6.1.1.tar.gz
- **2019-05-15** proj-6.1.0.tar.gz
- **2019-03-01** proj-6.0.0.tar.gz
- **2018-09-15** proj-5.2.0.tar.gz
- **2018-06-01** proj-5.1.0.tar.gz
- **2018-04-01** proj-5.0.1.tar.gz
- **2018-03-01** proj-5.0.0.tar.gz
- **2016-09-02** proj-4.9.3.tar.gz
- **2015-09-13** proj-4.9.2.tar.gz
- **2015-03-04** proj-4.9.1.tar.gz
- **2025-03-15** proj-data-1.21.tar.gz
- **2024-12-01** proj-data-1.20.tar.gz
- **2024-09-15** proj-data-1.19.tar.gz
- **2024-06-01** proj-data-1.18.tar.gz
- **2024-06-01** proj-data-1.17.tar.gz
- **2023-12-01** proj-data-1.16.tar.gz
- **2023-09-01** proj-data-1.15.tar.gz
- **2023-06-01** proj-data-1.14.tar.gz
- **2023-03-01** proj-data-1.13.tar.gz

- **2022-12-01** proj-data-1.12.tar.gz
- **2022-09-01** proj-data-1.11.tar.gz
- **2022-06-15** proj-data-1.10.tar.gz
- **2022-03-01** proj-data-1.9.tar.gz
- **2021-11-01** proj-data-1.8.tar.gz
- **2021-07-01** proj-data-1.7.tar.gz
- **2021-05-05** proj-data-1.6.tar.gz
- **2021-03-01** proj-data-1.5.tar.gz
- **2021-01-01** proj-data-1.4.tar.gz
- **2020-11-01** proj-data-1.3.tar.gz
- **2020-09-01** proj-data-1.2.tar.gz
- **2020-05-01** proj-data-1.1.tar.gz
- **2020-03-01** proj-data-1.0.tar.gz
- **2018-09-15** proj-datumgrid-1.8.zip
- **2020-03-01** proj-datumgrid-europe-1.6.zip
- **2020-03-01** proj-datumgrid-north-america-1.4.zip
- **2020-03-01** proj-datumgrid-oceania-1.2.zip
- **2019-03-01** proj-datumgrid-world-1.0.zip
- **2018-03-01** proj-datumgrid-1.7.zip
- **2016-09-11** proj-datumgrid-1.6.zip
- **2019-09-01** proj-datumgrid-europe-1.5.zip
- **2019-09-01** proj-datumgrid-europe-1.4.zip
- **2019-07-01** proj-datumgrid-europe-1.3.zip
- **2019-03-01** proj-datumgrid-europe-1.2.zip
- **2018-09-15** proj-datumgrid-europe-1.1.zip
- **2018-03-01** proj-datumgrid-europe-1.0.zip
- **2019-03-01** proj-datumgrid-north-america-1.3.zip
- **2019-03-01** proj-datumgrid-north-america-1.2.zip
- **2018-09-15** proj-datumgrid-north-america-1.1.zip
- **2018-03-01** proj-datumgrid-north-america-1.0.zip
- **2018-03-01** proj-datumgrid-oceania-1.1.zip
- **2018-03-01** proj-datumgrid-oceania-1.0.zip

CHAPTER

FOUR

# INSTALLATION

These pages describe how to install PROJ on your computer without compiling it yourself. Below are guides for installing on Windows, Linux and Mac OS X. This is a good place to get started if this is your first time using PROJ. More advanced users may want to compile the software themselves.

## 4.1 Installation from package management systems

### 4.1.1 Cross platform

PROJ is also available via cross platform package managers.

#### 4.1.1.1 Conda

The conda package manager includes several PROJ packages. We recommend installing from the `conda-forge` channel:

```
conda install -c conda-forge proj
```

Using `conda` you can also install the PROJ data package. Here's how to install the *proj-data* package:

```
conda install -c conda-forge proj-data
```

Available is also the legacy packages `proj-datumgrid-europe`, `proj-datumgrid-north-america`, `proj-datumgrid-oceania` and `proj-datumgrid-world`.

> **Tip**
>
> Read more about the various datumgrid packages available *here*.

#### 4.1.1.2 Docker

A Docker image with just PROJ binaries and a full compliment of grid shift files is available on DockerHub. Get the package with:

```
docker pull osgeo/proj
```

### 4.1.2 Windows

The simplest way to install PROJ on Windows is to use the OSGeo4W software distribution. OSGeo4W provides easy access to many popular open source geospatial software packages. After installation you can use PROJ from the OSGeo4W shell. To install PROJ do the following:

**Note**

If you have already installed software via OSGeo4W on your computer, or if you have already installed QGIS on your computer, it is likely that PROJ is already installed. Type "OSGeo4W Shell" in your start menu and check whether that gives a match.

1. Download the 64 bit installer.
2. Run the OSGeo4W setup program.
3. Select "Advanced Install" and press Next.
4. Select "Install from Internet" and press Next.
5. Select a installation directory. The default suggestion is fine in most cases. Press Next.
6. Select "Local package directory". The default suggestion is fine in most cases. Press Next.
7. Select "Direct connection" and press Next.
8. Choose the download.osgeo.org server and press Next.
9. Find "proj" under "Commandline_Utilities" and click the package in the "New" column until the version you want to install appears.
10. Press next to install PROJ.

You should now have a "OSGeo" menu in your start menu. Within that menu you can find the "OSGeo4W Shell" where you have access to all the OSGeo4W applications, including proj.

For those who are more inclined to the command line, steps 2--10 above can be accomplished by executing the following command:

```
C:\temp\osgeo4w-setup.exe -q -k -r -A -s https://download.osgeo.org/osgeo4w/v2/ -P proj
```

### 4.1.3 Linux

How to install PROJ on Linux depends on which distribution you are using. Below is a few examples for some of the more common Linux distributions:

#### 4.1.3.1 Debian

On Debian and similar systems (e.g. Ubuntu) the APT package manager is used:

```
sudo apt-get install proj-bin
```

#### 4.1.3.2 Fedora

On Fedora the `dnf` package manager is used:

```
sudo dnf install proj
```

#### 4.1.3.3 Red Hat

On Red Hat based system packages are installed with `yum`:

```
sudo yum install proj
```

### 4.1.4 Mac OS X

On OS X PROJ can be installed via the Homebrew package manager:

```
brew install proj
```

PROJ is also available from the MacPorts system:

```
sudo ports install proj
```

## 4.2 Compilation and installation from source code

The classic way of installing PROJ is via the source code distribution. The most recent version is available from the *download page*.

The following guides show how to compile and install the software using CMake.

> **Note**
>
> Support for Autotools was maintained until PROJ 8.2 (see *PROJ RFC 7: Drop Autotools, maintain CMake*). PROJ 9.0 and later releases only support builds using CMake.

### 4.2.1 Requirements

#### 4.2.1.1 Build requirements

- C99 compiler
- C++17 compiler
- CMake >= 3.16
- SQLite3 >= 3.11: headers and library for target architecture, and sqlite3 executable for build architecture
- libtiff >= 4.0 (optional but recommended)
- curl >= 7.29.0 (optional but recommended)
- JSON for Modern C++ (nlohmann/json) >= 3.7.0; if not found as an external dependency then vendored version 3.9.1 from PROJ source tree is used

#### 4.2.1.2 Test requirements

These are only required if testing is built (see *BUILD_TESTING*, default ON)

- GoogleTest (GTest) >= 1.8.1; if not found and *TESTING_USE_NETWORK* is ON, then version 1.15.2 is fetched from GitHub and locally installed
- Python >= 3.7
- importlib_metadata only needed for Python 3.7
- One of either PyYAML or ruamel.yaml

### 4.2.2 Build steps

With the CMake build system you can compile and install PROJ on more or less any platform. After unpacking the source distribution archive step into the source- tree:

```
cd proj-9.6.2
```

Create a build directory and step into it:

```
mkdir build
cd build
```

From the build directory you can now configure CMake, build and install the binaries:

```
cmake ..
cmake --build .
cmake --build . --target install
```

On Windows, one may need to specify generator:

```
cmake -G "Visual Studio 15 2017" ..
```

If the SQLite3 dependency is installed in a custom location, specify *CMAKE_PREFIX_PATH*:

```
cmake -DCMAKE_PREFIX_PATH=/opt/SQLite ..
```

Tests are run with:

```
ctest
```

With a successful install of PROJ, we can now install data files using the *projsync* utility:

```
projsync --system-directory --all
```

which will download all resource files currently available for PROJ. If less than the entire collection of resource files is needed the call to *projsync* can be modified to suit the users needs. See *projsync* for more options.

**Note**

The use of *projsync* requires that network support is enabled (the default option). If the resource files are not installed using *projsync* PROJ will attempt to fetch them automatically when a transformation needs a specific data file. This requires that *PROJ_NETWORK* is set to `ON`.

As an alternative on systems where network access is disabled, the *proj-data* package can be downloaded and its content decompressed into one of the directories where PROJ looks for *resources*

Starting with PROJ 9.2, a `uninstall` target is available to remove files installed by the `install` target:

```
cmake --build . --target uninstall
```

### 4.2.3 CMake configure options

Options to configure a CMake are provided using `-D<var>=<value>`. All cached entries can be viewed using `cmake -LAH` from a build directory.

**BUILD_APPS**=ON

Build PROJ applications. Default is ON. Control the default value for BUILD_CCT, BUILD_CS2CS, BUILD_GEOD, BUILD_GIE, BUILD_PROJ, BUILD_PROJINFO and BUILD_PROJSYNC. Note that changing its value after having configured once will not change the value of the individual BUILD_CCT, ... options.

Changed in version 8.2.

**BUILD_CCT**=ON

Build *cct*, default is the value of BUILD_APPS.

**BUILD_CS2CS**=ON

Build *cs2cs*,default is the value of BUILD_APPS.

**BUILD_GEOD**=ON

Build *geod*, default is the value of BUILD_APPS.

**BUILD_GIE**=ON

Build *gie*, default is the value of BUILD_APPS.

**BUILD_PROJ**=ON

Build *proj*, default is the value of BUILD_APPS.

**BUILD_PROJINFO**=ON

Build *projinfo*, default is the value of BUILD_APPS.

**BUILD_PROJSYNC**=ON

Build *projsync*, default is the value of BUILD_APPS.

**BUILD_SHARED_LIBS**

Build PROJ library shared. Default is ON. See also the CMake documentation for BUILD_SHARED_LIBS.

Changed in version 7.0: Renamed from `BUILD_LIBPROJ_SHARED`

> **Note**
>
> before PROJ 9.0, the default was OFF for Windows builds.

**BUILD_TESTING**=ON

CTest option to build the testing tree. Default is ON, but can be turned OFF if tests are not required. See *test requirements* for other details.

Changed in version 7.0: Renamed from `PROJ_TESTS`

**CMAKE_BUILD_TYPE**

Choose the type of build, options are: None (default), Debug, Release, RelWithDebInfo, or MinSizeRel. See also the CMake documentation for CMAKE_BUILD_TYPE.

> **Note**
>
> A default build is not optimized without specifying `-DCMAKE_BUILD_TYPE=Release` (or similar) during configuration, or by specifying `--config Release` with CMake multi-configuration build tools (see example below).

**PROJ_OUTPUT_NAME**

Added in version 9.5.

Sets the name of the PROJ library (excluding extension). This generally defaults to "proj", except on Windows, where this defaults to "proj_${PROJ_MAJOR_VERSION}" if APPEND_SOVERSION is OFF.

> **Note**
>
> For PROJ >= 6.0 and up to 9.4.1, on Windows, this was hardcoded to "proj_${PROJ_MAJOR_VERSION}_${PROJ_MINOR_VERSION}".

**APPEND_SOVERSION**=OFF

Added in version 9.5.

This variable can be set to ON for MinGW builds where BUILD_SHARED_LIBS=ON, to add a "-${PROJ_SOVERSION}" suffix to the PROJ shared library name. When this variable is set, PROJ_OUTPUT_NAME defaults to "proj"

**CMAKE_C_COMPILER**

C compiler. Ignored for some generators, such as Visual Studio.

**CMAKE_C_FLAGS**

Flags used by the C compiler during all build types. This is initialized by the `CFLAGS` environment variable.

**CMAKE_CXX_COMPILER**

C++ compiler. Ignored for some generators, such as Visual Studio.

**CMAKE_CXX_FLAGS**

Flags used by the C++ compiler during all build types. This is initialized by the `CXXFLAGS` environment variable.

**CMAKE_INSTALL_PREFIX**

Default for Windows is based on the environment variable `OSGEO4W_ROOT` (if set), otherwise is `c:/OSGeo4W`. Default for Unix-like is `/usr/local/`.

**CMAKE_PREFIX_PATH**

CMake variable used to specify installation prefixes for SQLite3 and other dependencies.

**CMAKE_UNITY_BUILD**=OFF

Added in version 9.4.

Default is OFF. This can be set to ON to build PROJ using the CMAKE_UNITY_BUILD. feature. This helps speeding PROJ build times. This feature is still considered experimental for now, and could hide subtle bugs (we are not aware of any at writing time though). We don't recommend it for mission critical builds.

**ENABLE_IPO**=OFF

Build library using the compiler's interprocedural optimization (IPO), if available, default OFF.

Changed in version 7.0: Renamed from `ENABLE_LTO`.

**EXE_SQLITE3**

Path to an `sqlite3` or `sqlite3.exe` executable.

> **Note**

When cross-compiling, the executable pointed by EXE_SQLITE3 must be of the same architecture as the host, *not* of the architecture you build for. That sqlite3 binary is used to build the `proj.db` SQLite3 database from source .sql files.

Deprecated since version 9.4.0: `SQLITE3_INCLUDE_DIR` and `SQLITE3_LIBRARY` should be replaced with `SQLite3_INCLUDE_DIR` and `SQLite3_LIBRARY`, respectively. Users may also consider *CMAKE_PREFIX_PATH* instead.

**ENABLE_CURL**=ON

Enable CURL support, default ON.

**CURL_INCLUDE_DIR**

Path to an include directory with the `curl` directory.

**CURL_LIBRARY**

Path to a shared or static library file, such as `libcurl.dll`, `libcurl.so`, `libcurl.lib`, or other name.

**ENABLE_TIFF**=ON

Enable TIFF support to use PROJ-data resource files, default ON.

**TIFF_INCLUDE_DIR**

Path to an include directory with the `tiff.h` header file.

**TIFF_LIBRARY_RELEASE**

Path to a shared or static library file, such as `tiff.dll`, `libtiff.so`, `tiff.lib`, or other name. A similar variable `TIFF_LIBRARY_DEBUG` can also be specified to a similar library for building Debug releases.

**USE_CCACHE**=OFF

Configure CMake to use ccache (or clcache for MSVC) to build C/C++ objects.

**PROJ_DB_CACHE_DIR**

Path to an existing directory used to cache `proj.db` to speed-up subsequent builds without modifications to source SQL files.

**TESTING_USE_NETWORK**=ON

Added in version 9.5.

Permit use of network to fetch *test requirements* (if needed) and run network-dependent tests. Default ON.

**EMBED_PROJ_DATA_PATH**

Added in version 9.5.

Embed `PROJ_DATA` hard-coded alternative path for data files location. Disable to avoid setting this non-relocatable hard-coded path. Default ON.

**EMBED_RESOURCE_FILES**=ON/OFF

Added in version 9.6.

When ON, `proj.db`, `proj.ini` and ITRF resource files will be embedded into the PROJ library. Default is OFF for shared library builds (BUILD_SHARED_LIBS=ON), and ON for static library builds (BUILD_SHARED_LIBS=OFF).

**EMBED_RESOURCE_DIRECTORY**=<directory>

Added in version 9.6.

Embed files from <directory> ending with .tif, .json or .pol in the PROJ library itself.

The pointed directory can potentially be the full PROJ-data package (uncompressed). In that case, about 6 GB of free disk and 16 GB of RAM are required to build PROJ.

When using this parameter, EMBED_RESOURCE_FILES must be set to ON.

If the content of the directory changes, you need to run CMake again to update the list of files.

**USE_ONLY_EMBEDDED_RESOURCE_FILES**=ON/OFF

Added in version 9.6.

Even if EMBED_RESOURCE_FILES=ON, by default PROJ will still try to locate `proj.db`, `proj.ini`, ITRF resource files or grid files on the file system, and fallback to the embedded version if not found. By setting USE_ONLY_EMBEDDED_RESOURCE_FILES=ON, no attempt at locating those files on the file system is made. Default is OFF. Users will also typically want to set EMBED_PROJ_DATA_PATH=OFF if setting USE_ONLY_EMBEDDED_RESOURCE_FILES=OFF.

### 4.2.4 Building on Windows with vcpkg and Visual Studio 2017 or 2019

This method is the preferred one to generate Debug and Release builds.

#### 4.2.4.1 Install git

Install git

#### 4.2.4.2 Install Vcpkg

Assuming there is a c:\dev directory

```
cd c:\dev
git clone https://github.com/Microsoft/vcpkg.git

cd vcpkg
.\bootstrap-vcpkg.bat
```

#### 4.2.4.3 Install PROJ dependencies

```
vcpkg install sqlite3[core,tool] tiff curl --triplet=x86-windows
vcpkg install sqlite3[core,tool] tiff curl --triplet=x64-windows
```

**Note**

The tiff and curl dependencies are only needed since PROJ 7.0

#### 4.2.4.4 Checkout PROJ sources

```
cd c:\dev
git clone https://github.com/OSGeo/PROJ.git
```

#### 4.2.4.5 Build PROJ

```
cd c:\dev\PROJ
mkdir build_vs2019
cd build_vs2019
cmake -DCMAKE_TOOLCHAIN_FILE=C:\dev\vcpkg\scripts\buildsystems\vcpkg.cmake ..
cmake --build . --config Debug -j 8
```

#### 4.2.4.6 Run PROJ tests

```
cd c:\dev\PROJ\build_vs2019
ctest -V --build-config Debug
```

### 4.2.5 Building on Windows with Conda dependencies and Visual Studio 2017 or 2019

Variant of the above method but using Conda for SQLite3, TIFF and CURL dependencies. It is less appropriate for Debug builds of PROJ than the method based on vcpkg.

#### 4.2.5.1 Install git

Install git

#### 4.2.5.2 Install miniconda

Install miniconda

#### 4.2.5.3 Install PROJ dependencies

Start a Conda enabled console and assuming there is a c:\dev directory

```
cd c:\dev
conda create --name proj
conda activate proj
conda install sqlite libtiff curl cmake
```

**Note**

The libtiff and curl dependencies are only needed since PROJ 7.0

#### 4.2.5.4 Checkout PROJ sources

```
cd c:\dev
git clone https://github.com/OSGeo/PROJ.git
```

#### 4.2.5.5 Build PROJ

From a Conda enabled console

```
conda activate proj
cd c:\dev\PROJ
call "C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\
↪vcvars64.bat"
cmake -S . -B _build.vs2019 -DCMAKE_LIBRARY_PATH:FILEPATH="%CONDA_PREFIX%/Library/lib" -
↪DCMAKE_INCLUDE_PATH:FILEPATH="%CONDA_PREFIX%/Library/include"
cmake --build _build.vs2019 --config Release -j 8
```

#### 4.2.5.6 Run PROJ tests

```
cd c:\dev\PROJ
cd _build.vs2019
ctest -V --build-config Release
```

CHAPTER

FIVE

# USING PROJ

The main purpose of PROJ is to transform coordinates from one coordinate reference system to another. This can be achieved either with the included command line applications or the C API that is a part of the software package.

## 5.1 Quick start

Coordinate transformations are defined by, what in PROJ terminology is known as, "proj-strings". A proj-string describes any transformation regardless of how simple or complicated it might be. The simplest case is projection of geodetic coordinates. This section focuses on the simpler cases and introduces the basic anatomy of the proj-string. The complex cases are discussed in *Geodetic transformation*.

A proj-strings holds the parameters of a given coordinate transformation, e.g.

```
+proj=merc +lat_ts=56.5 +ellps=GRS80
```

I.e. a proj-string consists of a projection specifier, `+proj`, a number of parameters that applies to the projection and, if needed, a description of a datum shift. In the example above geodetic coordinates are transformed to projected space with the *Mercator projection* with the latitude of true scale at 56.5 degrees north on the GRS80 ellipsoid. Every projection in PROJ is identified by a shorthand such as `merc` in the above example.

By using the above projection definition as parameters for the command line utility *proj* we can convert the geodetic coordinates to projected space:

```
$ proj +proj=merc +lat_ts=56.5 +ellps=GRS80
```

If called as above *proj* will be in interactive mode, letting you type the input data manually and getting a response presented on screen. *proj* works as any UNIX filter though, which means that you can also pipe data to the utility, for instance by using the `echo` command:

```
$ echo 55.2 12.2 | proj +proj=merc +lat_ts=56.5 +ellps=GRS80
3399483.80      752085.60
```

PROJ also comes bundled with the *cs2cs* utility which is used to transform from one coordinate reference system to another. Say we want to convert the above Mercator coordinates to UTM, we can do that with *cs2cs*:

```
$ echo 3399483.80 752085.60 | cs2cs +proj=merc +lat_ts=56.5 +ellps=GRS80 +to +proj=utm␣
↪+zone=32
6103992.36      1924052.47 0.00
```

Notice the `+to` parameter that separates the source and destination projection definitions.

If you happen to know the EPSG identifiers for the two coordinates reference systems you are transforming between you can use those with *cs2cs*:

```
$ echo 56 12 | cs2cs +init=epsg:4326 +to +init=epsg:25832
231950.54      1920310.71 0.00
```

In the above example we transform geodetic coordinates in the WGS84 reference frame to UTM zone 32N coordinates in the ETRS89 reference frame. UTM coordinates

## 5.2 Cartographic projection

The foundation of PROJ is the large number of *projections* available in the library. This section is devoted to the generic parameters that can be used on any projection in the PROJ library.

Below is a list of PROJ parameters which can be applied to most coordinate system definitions. This table does not attempt to describe the parameters particular to particular projection types. These can be found on the pages documenting the individual *projections*.

| Parameter | Description |
|---|---|
| +a | Semimajor radius of the ellipsoid axis (always in meters) |
| +axis | Axis orientation |
| +b | Semiminor radius of the ellipsoid axis (always in meters) |
| +ellps | Ellipsoid name (see `proj -le`) |
| +k | Scaling factor (deprecated) |
| +k_0 | Scaling factor |
| +lat_0 | Latitude of origin (in degree if no angular unit specified) |
| +lon_0 | Central meridian (in degree if no angular unit specified) |
| +lon_wrap | Center longitude to use for wrapping (see below) |
| +over | Allow longitude output outside -180 to 180 range, disables wrapping (see below) |
| +pm | Alternate prime meridian (typically a city name, see below) |
| +proj | Projection name (see `proj -l`) |
| +units | Horizontal coordinate system units (meters, US survey feet, etc.) |
| +vunits | Vertical coordinate system units. |
| +x_0 | False easting (always in meters) |
| +y_0 | False northing (always in meters) |

In the sections below most of the parameters are explained in details.

### 5.2.1 Units

Horizontal coordinate system units can be specified using the `+units` keyword with a symbolic me for a unit (i.e. `us-ft`). Alternatively the translation to meters can be specified with the `+to_meter` keyword (i.e. 0.304800609601219 for US feet). The `-lu` argument to *cs2cs* or *proj* can be used to list symbolic unit names. The default unit for projected coordinates is the meter. A few special projections deviate from this behavior, most notably the latlong pseudo-projection that returns degrees. Note that this does *not* affect the units of linear parameters such as `+x_0`, `+y_0`, `+a` or `+b` which should always be specified in meters.

Vertical (Z) units can be specified using the `+vunits` keyword with a symbolic name for a unit (i.e. `us-ft`). Alternatively the translation to meters can be specified with the `+vto_meter` keyword (i.e. 0.304800609601219 for US feet). The `-lu` argument to *cs2cs* or *proj* can be used to list symbolic unit names. If no vertical units are specified, the vertical units will default to be the same as the horizontal coordinates.

**Note**

*proj* does not handle vertical units at all and hence the `+vto_meter` argument will be ignored.

Scaling of output units can be done by applying the +k_0 argument (unitless). The returned coordinates are scaled by the value assigned with the +k_0 parameter. This parameter is only used by projections that mention using it, and its exact effect is projection dependent.

Input units for angular parameters (+lon_0, +lat_0, +pm, etc.) are interpreted to be decimal degrees by convention. Explicit specification of input angular units can be accomplished by adding the appropriate suffix to input values.

| Suffix | Unit |
|---|---|
| d | Decimal degrees |
| D | |
| ° | |
| r | Radians |
| R | |

The sign of an angle is taken from either a - or + prefix, or the last character specifying a cardinal direction, where e/E (East) or n/N (North) are positive and w/W (West) or s/S (South) are negative.

Example of use. The longitude of the central meridian +lon_0=90, can also be expressed more explicitly with units of decimal degrees as +lon_0=90d or in radian units as +lon_0=1.570796r (approximately).

Angles can be expressed in DMS notation, using ' after arcminutes and " after arcseconds, ending with optional cardinal direction. For example, +pm=3d41'14.55"W, but character escapes +pm=3d41\'14.55\"W may be required.

Degree-minute notation does not require any quotation symbols. All of these are equivalent values: +pm=-17d40, +pm=17D40W, +pm=17°40w or +pm=17d40'W (escaped as +pm=17d40\'W).

### 5.2.2 False Easting/Northing

Virtually all coordinate systems allow for the presence of a false easting (+x_0) and northing (+y_0). Note that these values are always expressed in meters even if the coordinate system is some other units. Some coordinate systems (such as UTM) have implicit false easting and northing values.

### 5.2.3 Longitude Wrapping

By default PROJ wraps output longitudes in the range -180 to 180. The +over switch can be used to disable the default wrapping which is done at a low level in pj_inv(). This is particularly useful with projections like the *Equidistant Cylindrical (Plate Carrée)* or *Mercator* where it could be desirable for X values past -20000000 (roughly) to continue past -180 instead of wrapping to +180.

Note however that for most projections where the 180 meridian does not project to a straight line, +over will have no effect or will not lead to expected results.

The +lon_wrap option can be used to provide an alternative means of doing longitude wrapping. It has only effect with operations that output angular coordinates, such as +proj=longlat The argument to this option is a center longitude. So +lon_wrap=180 means wrap longitudes in the range 0 to 360.

```
$ echo -1 0 cs2cs +proj=longlat +to +proj=longlat +lon_wrap=180
359dE   0dN 0.000
```

Note that +over does **not** disable +lon_wrap.

### 5.2.4 Prime Meridian

A prime meridian may be declared indicating the longitude offset between the prime meridian of the declared coordinate system and that of greenwich. A prime meridian is declared using the +pm parameter, and may be assigned an angle in DMS or decimal degrees format, or as a hard-coded name.

Currently prime meridian declarations are not used by the `pj_inv()` and `pj_fwd()` calls. Consequently the user utility *cs2cs* does honour prime meridians but the *proj* user utility ignores them.

Hard-coded prime meridians can be listed with `cs2cs -lm`. Note that the following list is no longer updated, and some values may conflict with other sources.

| Meridian | Longitude |
|---|---|
| greenwich | 0dE |
| lisbon | 9d07'54.862"W |
| paris | 2d20'14.025"E |
| bogota | 74d04'51.3"W |
| madrid | 3d41'16.58"W |
| rome | 12d27'8.4"E |
| bern | 7d26'22.5"E |
| jakarta | 106d48'27.79"E |
| ferro | 17d40'W |
| brussels | 4d22'4.71"E |
| stockholm | 18d3'29.8"E |
| athens | 23d42'58.815"E |
| oslo | 10d43'22.5"E |
| copenhagen | 12d34'40.35"E |

Example of use. The location `long=0`, `lat=0` in the greenwich based lat/long coordinates is translated to lat/long coordinates with Lisbon as the prime meridian.

```
cs2cs +proj=latlong +datum=WGS84 +to +proj=latlong +datum=WGS84 +pm=lisbon
0 0
9d7'54.862"E    0dN 0.000
```

Decimal degrees can also be simply specified for the prime meridian.

```
echo 13d30E 45N | proj +proj=merc +pm=13.5
0.00    5591295.92
```

See *Units* for other examples of angular inputs.

### 5.2.5 Axis orientation

Starting in PROJ 4.8.0, the +axis argument can be used to control the axis orientation of the coordinate system. The default orientation is "easting, northing, up" but directions can be flipped, or axes flipped using combinations of the axes in the +axis switch. The values are:

- "e" - Easting
- "w" - Westing
- "n" - Northing
- "s" - Southing
- "u" - Up
- "d" - Down

They can be combined in +axis in forms like:

- `+axis=enu` - the default easting, northing, elevation.

- `+axis=neu` - northing, easting, up - useful for "lat/long" geographic coordinates, or south orientated transverse mercator.
- `+axis=wnu` - westing, northing, up - some planetary coordinate systems have "west positive" coordinate systems

**Note**

The `+axis` argument does not work with the *proj* command line utility.

### 5.2.6 Order of applications of parameters

In the forward direction (from geodetic to projected coordinates), steps are performed in the following order:

- subtracting prime meridian (`+pm`) to input coordinate longitude
- subtracting central meridian (`+lon_0`) to input coordinate longitude
- normalizing input coordinate longitude to [-180, 180], unless `+over` is specified
- application of map projection formula on a spheroid of radius 1 and with the eccentricity of the target spheroid specified. Includes taking into account projection parameters such as `+k_0`, `+lat_0`, and other projection specific parameters
- scaling of output x, y values by the semimajor axis
- addition of false easting `+x_0` to x and false northing `+y_0` to y
- scaling of output x, y by the output horizontal unit (`+units` / `+to_meter`)
- scaling of output z by the output vertical unit (`+vunits` / `+vto_meter`)
- application of axis orientation and order (`+axis`)

This is implemented mostly in fwd.cpp

For the reverse direction (from projected coordinates to geodetic coordinates), reverse steps are performed in the reverse order.

## 5.3 Geodetic transformation

PROJ can do everything from the most simple projection to very complex transformations across many reference frames. While originally developed as a tool for cartographic projections, PROJ has over time evolved into a powerful generic coordinate transformation engine that makes it possible to do both large scale cartographic projections as well as coordinate transformation at a geodetic high precision level. This chapter delves into the details of how geodetic transformations of varying complexity can be performed.

In PROJ, two frameworks for geodetic transformations exists, the *PROJ 4.x/5.x* / *cs2cs* / `pj_transform()` framework and the *transformation pipelines* framework. The first is the original, and limited, framework for doing geodetic transforms in PROJ. The latter is a newer addition that aims to be a more complete transformation framework. Both are described in the sections below. Large portions of the text are based on [EversKnudsen2017].

Before describing the details of the two frameworks, let us first remark that most cases of geodetic transformations can be expressed as a series of elementary operations, the output of one operation being the input of the next. E.g. when going from UTM zone 32, datum ED50, to UTM zone 32, datum ETRS89, one must, in the simplest case, go through 5 steps:

1. Back-project the UTM coordinates to geographic coordinates
2. Convert the geographic coordinates to 3D cartesian geocentric coordinates
3. Apply a Helmert transformation from ED50 to ETRS89

4. Convert back from cartesian to geographic coordinates
5. Finally project the geographic coordinates to UTM zone 32 planar coordinates.

### 5.3.1 Transformation pipelines

The homology between the above steps and a Unix shell style pipeline is evident. It is there the main architectural inspiration behind the transformation pipeline framework. The pipeline framework is realized by utilizing a special "projection", that takes as its user supplied arguments, a series of elementary operations, which it strings together in order to implement the full transformation needed. Additionally, a number of elementary geodetic operations, including Helmert transformations, general high order polynomial shifts and the Molodensky transformation are available as part of the pipeline framework. In anticipation of upcoming support for full time-varying transformations, we also introduce a 4D spatiotemporal data type, and a programming interface (API) for handling this.

The Molodensky transformation converts directly from geodetic coordinates in one datum, to geodetic coordinates in another datum, while the (typically more accurate) Helmert transformation converts from 3D cartesian to 3D cartesian coordinates. So when using the Helmert transformation one typically needs to do an initial conversion from geodetic to cartesian coordinates, and a final conversion the other way round, to arrive at the desired result. Fortunately, this three-step compound transformation has the attractive characteristic that each step depends only on the output of the immediately preceding step. Hence, we can build a geodetic-to-geodetic Helmert transformation by tying together the outputs and inputs of 3 steps (geodetic-to-cartesian → Helmert → cartesian-to-geodetic), pipeline style. The pipeline driver, makes this kind of chained transformations possible. The implementation is compact, consisting of just one pseudo-projection, called `pipeline`, which takes as its arguments strings of elementary projections (note: "projection" is the, slightly misleading, PROJ term used for any kind of transformation). The pipeline pseudo projection is supplemented by a number of elementary transformations, all in all providing a framework for building high accuracy solutions for a wide spectrum of geodetic tasks.

As a first example, let us take a look at the iconic *geodetic → Cartesian → Helmert → geodetic* case (steps 2 to 4 in the example in the introduction). (Note, for the rest of this book the "+"s are removed from the examples.) In PROJ it can be implemented as

```
proj=pipeline
step proj=cart ellps=intl
step proj=helmert convention=coordinate_frame
     x=-81.0703  y=-89.3603  z=-115.7526
    rx=-0.48488 ry=-0.02436 rz=-0.41321  s=-0.540645
step proj=cart inv ellps=GRS80
```

The pipeline can be expanded at both ends to accommodate whatever coordinate type is needed for input and output: In the example below, we transform from the deprecated Danish System 45, a 2D system with some tension in the original defining network, to UTM zone 33, ETRS89. The tension is reduced using a polynomial transformation (the init=./s45b... step, s45b.pol is a file containing the polynomial coefficients), taking the S45 coordinates to a technical coordinate system (TC32), defined to represent "UTM zone 32 coordinates, as they would look if the Helmert transformation between ED50 and ETRS89 was perfect". The TC32 coordinates are then converted back to geodetic(ED50) coordinates, using an inverse UTM projection, further to cartesian(ED50), then to cartesian(ETRS89), using the relevant Helmert transformation, and back to geodetic(ETRS89), before finally being projected onto the UTM zone 33, ETRS89 system. All in all a 6 step pipeline, implementing a transformation with centimeter level accuracy from a deprecated system with decimeter level tensions.

```
proj=pipeline
step init=./s45b.pol:s45b_tc32
step proj=utm inv ellps=intl zone=32
step proj=cart ellps=intl
step proj=helmert convention=coordinate_frame
      x=-81.0703  y=-89.3603  z=-115.7526
```

(continues on next page)

(continued from previous page)

```
     rx=-0.48488 ry=-0.02436 rz=-0.41321 s=-0.540645
step proj=cart inv ellps=GRS80
step proj=utm ellps=GRS80 zone=33
```

With the pipeline framework spatiotemporal transformation is possible. This is possible by leveraging the time dimension in PROJ that enables 4D coordinates (three spatial components and one temporal component) to be passed through a transformation pipeline. In the example below a transformation from ITRF93 to ITRF2000 is defined. The temporal component is given as GPS weeks in the input data, but the 14-parameter Helmert transform expects temporal units in decimalyears. Hence the first step in the pipeline is the unitconvert pseudo-projection that makes sure the correct units are passed along to the Helmert transform. Most parameters of the Helmert transform are taken from [Altamimi2002], except the epoch which is the epoch of the transformation. The last step in the pipeline is converting the coordinate timestamps back to GPS weeks.

```
proj=pipeline
step proj=unitconvert t_in=gps_week t_out=decimalyear
step proj=helmert convention=coordinate_frame
     x=0.0127 y=0.0065 z=-0.0209 s=0.00195
     rx=0.00039 ry=-0.00080 rz=0.00114
     dx=-0.0029 dy=-0.0002 dz=-0.0006 ds=0.00001
     drx=0.00011 dry=0.00019 drz=-0.00007
     t_epoch=1988.0
step proj=unitconvert t_in=decimalyear t_out=gps_week
```

### 5.3.2 PROJ 4.x/5.x paradigm

| Parameter | Description |
|---|---|
| +datum | Datum name (see `proj -ld`) |
| +geoidgrids | Filename of GTX grid file to use for vertical datum transforms |
| +geoid_crs | (Added in PROJ 9.1) Value can be "WGS84" or "horizontal_crs". Only used if geoidgrids is specified. |
| +nadgrids | Filename of NTv2 grid file to use for datum transforms |
| +towgs84 | 3 or 7 term datum transform parameters |
| +to_meter | Multiplier to convert map units to 1.0m |
| +vto_meter | Vertical conversion to meters |

**Warning**

This section documents the behavior of PROJ 4.x and 5.x. In PROJ 6.x, *cs2cs* has been reworked to use *proj_create_crs_to_crs()* internally, with *late binding* capabilities, and thus is no longer constrained to using WGS84 as a pivot (also called as *early binding* method). When *cs2cs* of PROJ 6 is used with PROJ.4 expanded strings to describe the CRS, including `+towgs84`, `+nadgrids` and `+geoidgrids`, it will generally give the same results as earlier PROJ versions. When used with AUTHORITY:CODE CRS descriptions, it may return different results.

The *cs2cs* framework in PROJ 4 and 5 delivers a subset of the geodetic transformations available with the *pipeline* framework. Coordinate transformations done in this framework were transformed in a two-step process with WGS84

as a pivot datum. That is, the input coordinates are transformed to WGS84 geodetic coordinates and then transformed from WGS84 coordinates to the specified output coordinate reference system, by utilizing either the Helmert transform, datum shift grids or a combination of both. Datum shifts can be described in a proj-string with the parameters +towgs84, +nadgrids and +geoidgrids. An inverse transform exists for all three and is applied if specified in the input proj-string. The most common is +towgs84, which is used to define a 3- or 7-parameter Helmert shift from the input reference frame to WGS84. Exactly which realization of WGS84 is not specified, hence a fair amount of uncertainty is introduced in this step of the transformation. With the +nadgrids parameter a non-linear planar correction derived from interpolation in a correction grid can be applied. Originally this was implemented as a means to transform coordinates between the North American datums NAD27 and NAD83, but corrections can be applied for any datum for which a correction grid exists. The inverse transform for the horizontal grid shift is "dumb", in the sense that the correction grid is applied verbatim without taking into account that the inverse operation is non-linear. Similar to the horizontal grid correction, +geoidgrids can be used to perform grid corrections in the vertical component. Both grid correction methods allow inclusion of more than one grid in the same transformation

The +geoid_crs=WGS84/horizontal_crs parameter was added in PROJ 9.1 to solve an ambiguity that has been rampant for a long time. Before PROJ 6, the interpolation CRS of the geoid transformation was assumed to be the one of the horizontal CRS. Starting with PROJ 6, this was inadvertently changed to be WGS84.

When transforming from a CRS that specifies a horizontal datum shift with +towgs84 or +nadgrids, and a vertical one with +geoidgrids with +geoid_crs=horizontal_crs, to WGS84, the vertical transformation is done before the horizontal one.

```
$ projinfo -s "+proj=longlat +ellps=GRS80 +nadgrids=@foo +geoidgrids=@bar +geoid_
↪crs=horizontal_crs +type=crs" -t EPSG:4979  -o PROJ -q

+proj=pipeline
  +step +proj=unitconvert +xy_in=deg +xy_out=rad
  +step +proj=vgridshift +grids=@bar +multiplier=1
  +step +proj=hgridshift +grids=@foo
  +step +proj=unitconvert +xy_in=rad +z_in=m +xy_out=deg +z_out=m
  +step +proj=axisswap +order=2,1
```

Conversely, when +geoid_crs=WGS84 (or omitting it with PROJ >= 6), the horizontal transformation is done before the vertical one.

```
$ projinfo -s "+proj=longlat +ellps=GRS80 +nadgrids=@foo +geoidgrids=@bar +geoid_
↪crs=WGS84 +type=crs" -t EPSG:4979  -o PROJ -q

+proj=pipeline
  +step +proj=unitconvert +xy_in=deg +xy_out=rad
  +step +proj=hgridshift +grids=@foo
  +step +proj=vgridshift +grids=@bar +multiplier=1
  +step +proj=unitconvert +xy_in=rad +xy_out=deg
  +step +proj=axisswap +order=2,1
```

In contrast to the *transformation pipeline* framework, transformations with the *cs2cs* framework in PROJ 4 and 5 were expressed as two separate proj-strings. One proj-string *to* WGS84 and one *from* WGS84. Together they form the mapping from the source coordinate reference system to the destination coordinate reference system. When used with the *cs2cs* the source and destination CRS's are separated by the special +to parameter.

The following example demonstrates converting from the Greek GGRS87 datum to WGS84 with the +towgs84 parameter.

```
cs2cs +proj=latlong +ellps=GRS80 +towgs84=-199.87,74.79,246.62
    +to +proj=latlong +datum=WGS84
```

(continues on next page)

(continued from previous page)

```
20 35
20d0'5.467"E    35d0'9.575"N 0.000
```

With PROJ 6, you can simply use the following:

> **Note**
>
> With PROJ 6, the order of coordinates for EPSG geographic coordinate reference systems is latitude first, longitude second.

```
cs2cs "GGRS87" "WGS 84"
35 20
35d0'9.575"N    20d0'5.467"E 0.000

cs2cs EPSG:4121 EPSG:4326
35 20
35d0'9.575"N    20d0'5.467"E 0.000
```

The EPSG database provides this example for transforming from WGS72 to WGS84 using an approximated 7 parameter transformation.

```
cs2cs +proj=latlong +ellps=WGS72 +towgs84=0,0,4.5,0,0,0.554,0.219 \
    +to +proj=latlong +datum=WGS84
4 55
4d0'0.554"E     55d0'0.09"N 0.000
```

With PROJ 6, you can simply use the following (note the reversed order for latitude and longitude)

```
cs2cs "WGS 72" "WGS 84"
55 4
55d0'0.09"N 4d0'0.554"E 0.000

cs2cs EPSG:4322 EPSG:4326
55 4
55d0'0.09"N 4d0'0.554"E 0.000
```

### 5.3.3 Grid Based Datum Adjustments

In many places (notably North America and Australia) national geodetic organizations provide grid shift files for converting between different datums, such as NAD27 to NAD83. These grid shift files include a shift to be applied at each grid location. Actually grid shifts are normally computed based on an interpolation between the containing four grid points.

PROJ supports use of grid files for shifting between various reference frames. The grid shift table formats are CTable, NTv1 (the old Canadian format), and NTv2 (`.gsb` - the new Canadian and Australian format).

The text in this section is based on the *cs2cs* framework. Gridshifting is off course also possible with the *pipeline* framework. The major difference between the two is that the *cs2cs* framework is limited to grid mappings to WGS84, whereas with *transformation pipelines* it is possible to perform grid shifts between any two reference frames, as long as a grid exists.

Use of grid shifts with *cs2cs* is specified using the `+nadgrids` keyword in a coordinate system definition. For example:

```
% cs2cs +proj=latlong +ellps=clrk66 +nadgrids=ntv1_can.dat \
    +to +proj=latlong +ellps=GRS80 +datum=NAD83 << EOF
-111 50
EOF
111d0'2.952"W   50d0'0.111"N 0.000
```

In this case the `/usr/local/share/proj/ntv1_can.dat` grid shift file was loaded, and used to get a grid shift value for the selected point.

It is possible to list multiple grid shift files, in which case each will be tried in turn till one is found that contains the point being transformed.

```
cs2cs +proj=latlong +ellps=clrk66 \
          +nadgrids=conus,alaska,hawaii,stgeorge,stlrnc,stpaul \
    +to +proj=latlong +ellps=GRS80 +datum=NAD83 << EOF
-111 44
EOF
111d0'2.788"W   43d59'59.725"N 0.000
```

#### 5.3.3.1 Skipping Missing Grids

The special prefix `@` may be prefixed to a grid to make it optional. If it not found, the search will continue to the next grid. Normally any grid not found will cause an error. For instance, the following would use the `ntv2_0.gsb` file if available, otherwise it would fallback to using the `ntv1_can.dat` file.

```
cs2cs +proj=latlong +ellps=clrk66 +nadgrids=@ntv2_0.gsb,ntv1_can.dat \
    +to +proj=latlong +ellps=GRS80 +datum=NAD83 << EOF
-111 50
EOF
111d0'3.006"W   50d0'0.103"N 0.000
```

#### 5.3.3.2 The null Grid

A special `null` grid shift file is distributed with PROJ. This file provides a zero shift for the whole world. It may be listed at the end of a nadgrids file list if you want a zero shift to be applied to points outside the valid region of all the other grids. Normally if no grid is found that contains the point to be transformed an error will occur.

```
cs2cs +proj=latlong +ellps=clrk66 +nadgrids=conus,null \
    +to +proj=latlong +ellps=GRS80 +datum=NAD83 << EOF
-111 45
EOF
111d0'3.006"W   50d0'0.103"N 0.000

cs2cs +proj=latlong +ellps=clrk66 +nadgrids=conus,null \
    +to +proj=latlong +ellps=GRS80 +datum=NAD83 << EOF
-111 44
-111 55
EOF
111d0'2.788"W   43d59'59.725"N 0.000
111dW   55dN 0.000
```

For more information see the chapter on *Other transformation grids*.

#### 5.3.3.3 Caveats

- Where grids overlap (such as conus and `ntv1_can.dat` for instance) the first found for a point will be used regardless of whether it is appropriate or not. So, for instance, `+nadgrids=ntv1_can.dat`,conus would result in the Canadian data being used for some areas in the northern United States even though the conus data is the approved data to use for the area. Careful selection of files and file order is necessary. In some cases border spanning datasets may need to be pre-segmented into Canadian and American points so they can be properly grid shifted
- Additional detail on the grid shift being applied can be found by setting the PROJ_DEBUG environment variable to a value. This will result in output to stderr on what grid is used to shift points, the bounds of the various grids loaded and so forth

## 5.4 Ellipsoids

An ellipsoid is a mathematically defined surface which approximates the *geoid*: the surface of the Earth's gravity field, which is approximately the same as mean sea level.

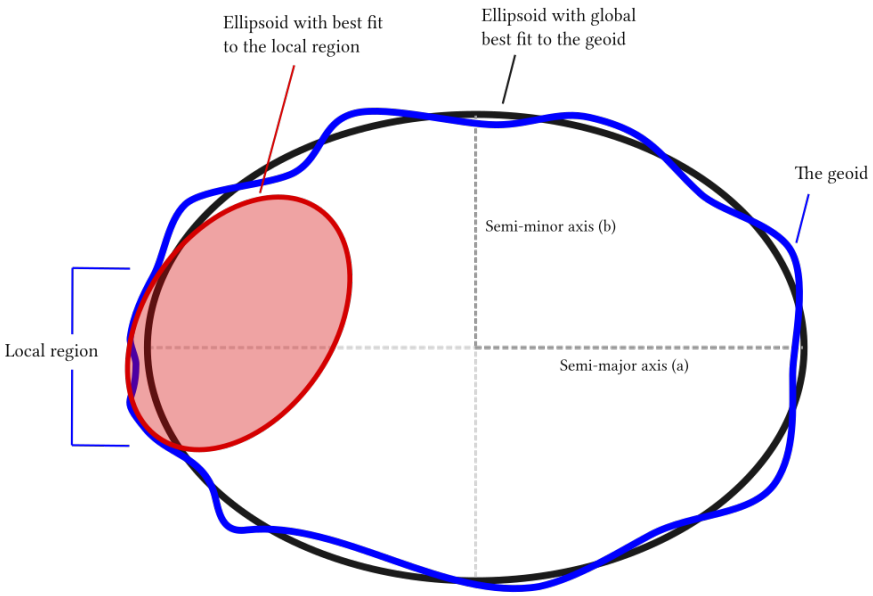



Fig. 1: Global and local fitting of the ellipsoid

A complete ellipsoid definition comprises a size (primary) and a shape (secondary) parameter.

### 5.4.1 Ellipsoid size parameters

**+R**=<value>

Radius of the sphere, $R$.

**+a**=<value>

Semi-major axis of the ellipsoid, $a$.

### 5.4.2 Ellipsoid shape parameters

**+rf**=<value>

Reverse flattening of the ellipsoid, $1/f$.

**+f**=<value>

Flattening of the ellipsoid, $f$.

**+es**=<value>

Eccentricity squared, $e^2$.

**+e**=<value>

Eccentricity, $e$.

**+b**=<value>

Semi-minor axis, $b$.

The ellipsoid definition may be augmented with a spherification flag, turning the ellipsoid into a sphere with features defined by the ellipsoid.

### 5.4.3 Ellipsoid spherification parameters

**+R_A**

A sphere with the same surface area as the ellipsoid.

**+R_V**

A sphere with the same volume as the ellipsoid.

**+R_C**

Added in version 9.3.0.

A sphere whose radius is the radius of the conformal sphere at $\phi_0$.

**+R_a**

A sphere with $R = (a + b)/2$ (arithmetic mean).

**+R_g**

A sphere with $R = \sqrt{ab}$ (geometric mean).

**+R_h**

A sphere with $R = 2ab/(a + b)$ (harmonic mean).

**+R_lat_a**=<phi>

A sphere with $R$ being the arithmetic mean of the corresponding ellipsoid at latitude $\phi$.

**+R_lat_g**=<phi>

A sphere with $R$ being the geometric mean of the corresponding ellipsoid at latitude $\phi$.

If *+R* is given as size parameter, any shape and spherification parameters given are ignored.

### 5.4.4 Built-in ellipsoid definitions

The `+ellps=xxx` parameter provides both size and shape for a number of built-in ellipsoid definitions. Default is GRS80 if not given.

| ellps | Parameters | Datum name |
|---|---|---|
| GRS80 | a=6378137.0 rf=298.257222101 | GRS 1980(IUGG, 1980) |
| airy | a=6377563.396 b=6356256.910 | Airy 1830 |
| bessel | a=6377397.155 rf=299.1528128 | Bessel 1841 |
| clrk66 | a=6378206.4 b=6356583.8 | Clarke 1866 |
| intl | a=6378388.0 rf=297. | International 1909 (Hayford) |
| WGS60 | a=6378165.0 rf=298.3 | WGS 60 |
| WGS66 | a=6378145.0 rf=298.25 | WGS 66 |
| WGS72 | a=6378135.0 rf=298.26 | WGS 72 |
| WGS84 | a=6378137.0 rf=298.257223563 | WGS 84 |
| sphere | a=6370997.0 b=6370997.0 | Normal Sphere (r=6370997) |

If size and shape are given as `+ellps=xxx`, later shape and size parameters are are taken into account as modifiers for the built-in ellipsoid definition.

While this may seem strange, it is in accordance with historical PROJ behavior. It can e.g. be used to define coordinates on the ellipsoid scaled to unit semimajor axis by specifying `+ellps=xxx +a=1`

### 5.4.5 Transformation examples

Spherical earth with radius 7000km:

```
+proj=latlon +R=7000000
```

Using the GRS80 ellipsoid:

```
+proj=latlon +ellps=GRS80
```

Expressing ellipsoid by semi-major axis and reverse flattening $(1/f)$:

```
+proj=latlon +a=6378137.0 +rf=298.25
```

Spherical earth based on volume of ellipsoid

```
+proj=latlon +a=6378137.0 +rf=298.25 +R_V
```

## 5.5 Environment variables

PROJ can be controlled by setting environment variables. Most users will have a use for the *PROJ_DATA*.

On UNIX systems environment variables can be set for a shell-session with:

```
$ export VAR="some variable"
```

or it can be set for just one command line call:

```
$ VAR="some variable" ./cmd
```

Environment variables on UNIX are usually removed with the `unset` command:

```
$ unset VAR
```

On windows systems environment variables can be set in the command line with:

```
> set VAR="some variable"
```

VAR will be available for the entire session, unless it is unset. This is done by setting the variable with no content:

```
> set VAR=
```

**PROJ_DATA**

Prior to PROJ 9.1, this environment variable was called PROJ_LIB. This older name is still supported in PROJ 9.1 as a fallback, but support for it may be removed in future release.

The location of PROJ *resource files*.

Starting with PROJ 6, multiple directories can be specified. On Unix, they should be separated by the colon (:) character. on Windows, by the semi-colon (;) character.

PROJ is hardcoded to look for resource files in other locations as well, amongst those are the installation directory (usually share/proj under the PROJ installation root) and the current folder.

You can also set the location of the resource files using proj_context_set_search_paths() in the proj.h API header.

Changed in version 6.1.0: Starting with PROJ version 6.1.0, the paths set by proj_context_set_search_paths() will have priority over the *PROJ_DATA* to allow for multiple versions of PROJ resource files on your system without conflicting.

**PROJ_AUX_DB**

Added in version 8.1.0.

To set the path to one or several auxiliary SQLite3 databases of structure identical to the main proj.db database and that can contain additional object (CRS, transformation, ...) definitions. If several paths are provided, they must be separated by the colon (:) character on Unix, and on Windows, by the semi-colon (;) character.

**PROJ_DEBUG**

Set the debug level of PROJ.

The following levels are available: - 0: no message. - 1: error messages only (default). - 2: same as 1, with debug messages. - 3: same as 2, with verbose messages. - 4: same as 3, with very verbose messages.

Starting with PROJ 9.3, ON can be used as an alias for 2, and OFF as an alias for 1.

**PROJ_NETWORK**

Added in version 7.0.0.

If set to ON, enable the capability to use remote grids stored on CDN (Content Delivery Network) storage, when grids are not available locally. Alternatively, the *proj_context_set_enable_network()* function can be used.

**PROJ_NETWORK_ENDPOINT**

Added in version 7.0.0.

Define the endpoint of the CDN storage. Normally defined through the proj.ini configuration file locale in *PROJ_DATA*. Alternatively, the *proj_context_set_url_endpoint()* function can be used.

**PROJ_CURL_CA_BUNDLE**

Added in version 7.2.0.

Define a custom path to the CA Bundle file. This can be useful if *curl* and *PROJ_NETWORK* are enabled. Alternatively, the proj_curl_set_ca_bundle_path() function can be used.

**PROJ_NATIVE_CA**

Added in version 9.6.0.

When this is set to ON, the operating systems native CA store will be used for certificate verification If you set this option to ON and also set PROJ_CURL_CA_BUNDLE then during verification those certificates are searched in addition to the native CA store.

## 5.6 Known differences between versions

Once in a while, a new version of PROJ causes changes in the existing behavior. In this section we track deliberate changes to PROJ that break from previous behavior. Most times that will be caused by a bug fix. Unfortunately, some bugs have existed for so long that their faulty behavior is relied upon by software that uses PROJ.

Behavioural changes caused by new bugs are not tracked here, as they should be fixed in later versions of PROJ.

### 5.6.1 Version 4.6.0

The default datum application behavior changed with the 4.6.0 release. PROJ will now only apply a datum shift if both the source and destination coordinate system have valid datum shift information.

The PROJ 4.6.0 Release Notes states

> MAJOR: Rework `pj_transform()` to avoid applying ellipsoid to ellipsoid transformations as a datum shift when no datum info is available.

### 5.6.2 Version 5.0.0

#### 5.6.2.1 Longitude wrapping when using custom central meridian

By default PROJ wraps output longitudes in the range -180 to 180. Previous to PROJ 5, this was handled incorrectly when a custom central meridian was set with *+lon_0*. This caused a change in sign on the resulting easting as seen below:

```
$ proj +proj=merc +lon_0=110 <<EOF
-70 0
290 0
EOF
20037508.34     0.00
-20037508.34    0.00
```

From PROJ 5 on onwards, the same input now results in same coordinates, as seen from the example below where PROJ 5 is used:

```
$ proj +proj=merc +lon_0=110 <<EOF
-70 0
290 0
EOF
-20037508.34     0.00
-20037508.34    0.00
```

The change is made on the basis that $\lambda = 290°$ is a full rotation of the circle larger than $\lambda = -70°$ and hence should return the same output for both.

Adding the `+over` flag to the projection definition provides the old behavior.

## 5.6.3 Version 6.0.0

### 5.6.3.1 Removal of `proj_def.dat`

Before PROJ 6, the `proj_def.dat` was used to provide general and per-projection parameters, when `+no_defs` was not specified. It has now been removed. In case, no ellipsoid or datum specification is provided in the PROJ string, the default ellipsoid is GRS80 (was WGS84 in previous PROJ versions).

### 5.6.3.2 Changes to deformation

#### Reversed order of operation

In the initial version of the of *deformation* operation the time span between $t_{obs}$ and $t_c$ was determined by the expression

$$dt = t_c - t_{obs}$$

With version 6.0.0 this has been reversed in order to behave similarly to the *Helmert operation*, which determines time differences as

$$dt = t_{obs} - t_c$$

Effectively this means that the direction of the operation has been reversed, so that what in PROJ 5 was a forward operation is now an inverse operation and vice versa.

Pipelines written for PROJ 5 can be migrated to PROJ 6 by adding *`+inv`* to forward steps involving the deformation operation. Similarly *`+inv`* should be removed from inverse steps to be compatible with PROJ 6.

#### Removed `+t_obs` parameter

The `+t_obs` parameter was confusing for users since it effectively overwrote the observation time in input coordinates. To make it more clear what is the operation is doing, users are now required to directly specify the time span for which they wish to apply a given deformation. The parameter *`+dt`* has been added for that purpose. The new parameter is mutually exclusive with *`+t_epoch`*. *`+dt`* is used when deformation for a set amount of time is needed and *`+t_epoch`* is used (in conjunction with the observation time of the input coordinate) when deformation from a specific epoch to the observation time is needed.

## 5.6.4 Version 6.3.0

### 5.6.4.1 projinfo

Before PROJ 6.3.0, WKT1:GDAL was implicitly calling `--boundcrs-to-wgs84`, to add a `TOWGS84[]` node in some cases. This is no longer the case.

## 5.6.5 Version 7.0.0

### 5.6.5.1 proj

Removed `-ld` option from application, since it promoted use of deprecated parameters like `+towgs` and `+datum`.

### 5.6.5.2 cs2cs

Removed `-ld` option from application, since it promoted use of deprecated parameters like `+towgs` and `+datum`.

### 5.6.5.3 UTF-8 adoption

The value of all path, filenames passed to PROJ through function calls, PROJ strings or environment variables should be encoded in UTF-8.

## 5.7 Network capabilities

Added in version 7.0.

PROJ 7.0 has introduced, per *PROJ RFC 4: Remote access to grids and GeoTIFF grids*, the capability to work with grid files that are not installed on the local machine where PROJ is executed.

This enables to transparently download the parts of grids that are needed to perform a coordinate transformation.

### 5.7.1 CDN of GeoTIFF grids

Files are accessed by default through a CDN (Content Delivery Network), accessible through https://cdn.proj.org, that contains *Geodetic TIFF grids (GTG)* datasets which are mirrored and managed by the https://github.com/OSGeo/PROJ-data/ GitHub project. Files in the CDN are designed to be used by PROJ 7 or later, but any software project wishing to use the CDN for shifting support are encouraged to participate in the project and leverage the CDN.

### 5.7.2 How to enable network capabilities ?

This capability assumes that PROJ has been build against *libcurl*, and that the user authorizes network access.

Authorizing network access can be done in multiple ways:

- enabling / uncommenting the `network = on` line of *proj.ini*
- defining the *PROJ_NETWORK* environment variable to ON
- or using the `proj_context_set_enable_network()` with a `enabled = TRUE` value.

> **Note**
>
> Instead of using the *libcurl* implementation, an application using the PROJ API can supply its own network implementation through C function callbacks with `proj_context_set_network_callbacks()`. Enabling network use must still be done with one of the above mentioned method.

### 5.7.3 Setting endpoint

When this is enabled, and a grid is not found in the various locations where *resource files are looked for*, PROJ will then attempt at loading the file from a remote server, which defaults to https://cdn.proj.org in *proj.ini*. This location can be changed with the *PROJ_NETWORK_ENDPOINT* environment variable or with `proj_context_set_url_endpoint()`.

### 5.7.4 Caching

To avoid repeated access to network, a local cache of downloaded chunks of grids is implemented as SQLite3 database, `cache.db`, stored in the *PROJ user writable directory*.

This local caching is enabled by default (can be changed in *proj.ini* or with `proj_grid_cache_set_enable()`). The default maximum size of the cache is 300 MB, which is more than half of the total size of grids available, at time of writing. This size can also be customized in *proj.ini* or with `proj_grid_cache_set_max_size()`

### 5.7.5 Download API

When on-demand loading of grid is not desirable, the PROJ API also offers the capability to download whole grids in the *PROJ user writable directory* by using the `proj_is_download_needed()` and `proj_download_file()` functions.

### 5.7.6 Download utility

*projsync* is a tool for downloading resource files.

### 5.7.7 Mirroring

If you are able, you are encouraged to mirror the grids via AWS S3 command line:

```
aws s3 sync s3://cdn.proj.org .
```

If direct S3 access is not possible, you can also use wget to locally mirror the data:

```
wget --mirror https://cdn.proj.org/
```

### 5.7.8 Acknowledgements

The s3://cdn.proj.org bucket is hosted by the Amazon Public Datasets program. CDN services are provided by the AWS Public Dataset team via CloudFront

CHAPTER

SIX

# APPLICATIONS

Bundled with PROJ comes a set of small command line utilities. The *proj* program is limited to converting between geographic and projection coordinates within one datum. The *cs2cs* program operates similarly, but allows translation between any pair of definable coordinate systems, including support for datum transformation. The *geod* program provides the ability to do geodesic (great circle) computations. *gie* is the program used for regression tests in PROJ. *cct*, a 4D equivalent to the *proj* program, performs transformation coordinate systems on a set of input points. *projinfo* performs queries for geodetic objects and coordinate operations. *projsync* is a tool for synchronizing PROJ datum and transformation support data.

## 6.1 cct

### 6.1.1 Synopsis

**cct** [**-cIostvz** [args]] +*opt[=arg]* ... file ...

or

**cct** [**-cIostvz** [args]] {object_definition} file ...

Where {object_definition} is one of the possibilities accepted by `proj_create()`, provided it expresses a coordinate operation

- a proj-string,
- a WKT string,
- an object code (like "EPSG:1671" "urn:ogc:def:coordinateOperation:EPSG::1671"),
- an object name. e.g. "ITRF2014 to ETRF2014 (1)". In that case as uniqueness is not guaranteed, heuristics are applied to determine the appropriate best match.
- a OGC URN combining references for concatenated operations (e.g. "urn:ogc:def:coordinateOperation,coordinateOperation:EPSG::3895,coordinateOperation:EPSG::1618")
- a PROJJSON string. The jsonschema is at https://proj.org/schemas/v0.4/projjson.schema.json

Added in version 8.0.0.

> **Note**
>
> Before version 8.0.0 only proj-strings could be used to instantiate operations in *cct*.

or

**cct** [**-cIostvz** [args]] {object_reference} file ...

where {object_reference} is a filename preceded by the '@' character. The file referenced by the {object_reference} must contain a valid {object_definition}.

Added in version 8.0.0.

### 6.1.2 Description

*cct* is a 4D equivalent to the *proj* projection program, performs transformation coordinate systems on a set of input points. The coordinate system transformation can include translation between projected and geographic coordinates as well as the application of datum shifts.

Note however that unlike the *proj*, angular input must be in decimal degrees. Any minutes and seconds given will be silently dropped.

The following control parameters can appear in any order:

**-c** <x,y,z,t>

Specify input columns for (up to) 4 input parameters. Defaults to 1,2,3,4.

**-d** <n>

Added in version 5.2.0.

Specify the number of decimals to round to in the output.

**-I**

Do the inverse transformation.

**-o** <output file name>, **--output**=<output file name>

Specify the name of the output file.

**-t** <time>, **--time**=<time>

Specify a fixed observation *time* to be used for all input data.

**-z** <height>, **--height**=<height>

Specify a fixed observation *height* to be used for all input data.

**-s** <n>, **--skip-lines**=<n>

Added in version 5.1.0.

Skip the first *n* lines of input. This applies to any kind of input, whether it comes from STDIN, a file or interactive user input.

**-v, --verbose**

Write non-essential, but potentially useful, information to stderr. Repeat for additional information (-vv, -vvv, etc.)

**--version**

Print version number.

The *+opt* arguments are associated with coordinate operation parameters. Usage varies with operation.

*cct* is an acronym meaning *Coordinate Conversion and Transformation*.

The acronym refers to definitions given in the OGC 08-015r2/ISO-19111 standard "Geographical Information -- Spatial Referencing by Coordinates", which defines two different classes of *coordinate operations*:

*Coordinate Conversions*, which are coordinate operations where input and output datum are identical (e.g. conversion from geographical to cartesian coordinates) and

*Coordinate Transformations*, which are coordinate operations where input and output datums differ (e.g. change of reference frame).

### 6.1.3 Use of remote grids

Added in version 7.0.0.

If the *PROJ_NETWORK* environment variable is set to ON, *cct* will attempt to use remote grids stored on CDN (Content Delivery Network) storage, when they are not available locally.

More details are available in the *Network capabilities* section.

### 6.1.4 Examples

1. The operator specs describe the action to be performed by *cct*. So the following script

```
echo 12 55 0 0 | cct +proj=utm +zone=32 +ellps=GRS80
```

will transform the input geographic coordinates into UTM zone 32 coordinates. Hence, the command

```
echo 12 55 | cct -z0 -t0 +proj=utm +zone=32 +ellps=GRS80
```

Should give results comparable to the classic *proj* command

```
echo 12 55 | proj +proj=utm +zone=32 +ellps=GRS80
```

2. Convert geographical input to UTM zone 32 on the GRS80 ellipsoid:

```
cct +proj=utm +ellps=GRS80 +zone=32
```

3. Roundtrip accuracy check for the case above:

```
cct +proj=pipeline +ellps=GRS80 +zone=32 +step +proj=utm +step +proj=utm +inv
```

4. As (2) but specify input columns for longitude, latitude, height and time:

```
cct -c 5,2,1,4 +proj=utm +ellps=GRS80 +zone=32
```

5. As (2) but specify fixed height and time, hence needing only 2 cols in input:

```
cct -t 0 -z 0 +proj=utm +ellps=GRS80 +zone=32
```

6. Auxiliary data following the coordinate input is forwarded to the output stream:

```
$ echo 12 56 100 2018.0 auxiliary data | cct +proj=merc
1335833.8895   7522963.2411      100.0000     2018.0000 auxiliary data
```

7. Coordinate operation referenced through its code

```
$ echo 3541657.3778 948984.2343 5201383.5231 2020.5 | cct EPSG:8366
3541657.9112    948983.7503  5201383.2482     2020.5000
```

8. Coordinate operation referenced through its name

```
$ echo 3541657.3778 948984.2343 5201383.5231 2020.5 | cct "ITRF2014 to ETRF2014 (1)"
3541657.9112    948983.7503  5201383.2482     2020.5000
```

### 6.1.5 Background

*cct* also refers to Carl Christian Tscherning (1942--2014), professor of Geodesy at the University of Copenhagen, mentor and advisor for a generation of Danish geodesists, colleague and collaborator for two generations of global geodesists, Secretary General for the International Association of Geodesy, IAG (1995--2007), fellow of the American Geophysical Union (1991), recipient of the IAG Levallois Medal (2007), the European Geosciences Union Vening Meinesz Medal (2008), and of numerous other honours.

*cct*, or Christian, as he was known to most of us, was recognized for his good mood, his sharp wit, his tireless work, and his great commitment to the development of geodesy -- both through his scientific contributions, comprising more than 250 publications, and by his mentoring and teaching of the next generations of geodesists.

As Christian was an avid Fortran programmer, and a keen Unix connoisseur, he would have enjoyed to know that his initials would be used to name a modest Unix style transformation filter, hinting at the tireless aspect of his personality, which was certainly one of the reasons he accomplished so much, and meant so much to so many people.

Hence, in honour of *cct* (the geodesist) this is *cct* (the program).

## 6.2 cs2cs

### 6.2.1 Synopsis

**cs2cs** [**-eEfIlrstvwW** [args]]
[[--area <name_or_code>] | [--bbox <west_long,south_lat,east_long,north_lat>]]
[--authority <name>] [--3d]
[--accuracy <accuracy>] [--only-best[=yes|=no]] [--no-ballpark]
[--s_epoch {epoch}] [--t_epoch {epoch}]
([+*opt[=arg]* ...] [+to +*opt[=arg]* ...] | {source_crs} {target_crs})
file ...

where {source_crs} or {target_crs} is one of the possibilities accepted by *proj_create()*, provided it expresses a CRS

- a proj-string,
- a WKT string,
- an object code (like "EPSG:4326", "urn:ogc:def:crs:EPSG::4326", "urn:ogc:def:coordinateOperation:EPSG::1671"),
- an Object name. e.g "WGS 84", "WGS 84 / UTM zone 31N". In that case as uniqueness is not guaranteed, heuristics are applied to determine the appropriate best match.
- a CRS name and a coordinate epoch, separated with '@'. For example "ITRF2014@2025.0". (*added in 9.2*)
- a OGC URN combining references for compound coordinate reference systems (e.g "urn:ogc:def:crs,crs:EPSG::2393,crs:EPSG::5717" or custom abbreviated syntax "EPSG:2393+5717"),
- a OGC URN combining references for references for projected or derived CRSs e.g. for Projected 3D CRS "UTM zone 31N / WGS 84 (3D)": "urn:ogc:def:crs,crs:EPSG::4979,cs:PROJ::ENh,coordinateOperation:EPSG::16031" (*added in 6.2*)
- a OGC URN combining references for concatenated operations (e.g. "urn:ogc:def:coordinateOperation,coordinateOperation:EPSG::3895,coordinateOperation:EPSG::1618")
- a PROJJSON string. The jsonschema is at https://proj.org/schemas/v0.4/projjson.schema.json (*added in 6.2*)

- a compound CRS made from two object names separated with " + ". e.g. "WGS 84 + EGM96 height" (*added in 7.1*)

Added in version 6.0.0.

> **Note**
>
> before 7.0.1, it was needed to add +to between {source_crs} and {target_crs} when adding a filename

### 6.2.2 Description

*cs2cs* performs transformation between the source and destination cartographic coordinate reference system on a set of input points. The coordinate reference system transformation can include translation between projected and geographic coordinates as well as the application of datum shifts.

The following control parameters can appear in any order:

**-I**

Method to specify inverse translation, convert from +*to* coordinate system to the primary coordinate system defined.

**-t<a>**

Where *a* specifies a character employed as the first character to denote a control line to be passed through without processing. This option applicable to ASCII input only. (# is the default value).

**-d** <n>

Added in version 5.2.0.

Specify the number of decimals to round to in the output.

**-e** <string>

Where *string* is an arbitrary string to be output if an error is detected during data transformations. The default value is a three character string: `*\t*`.

**-E**

Causes the input coordinates to be copied to the output line prior to printing the converted values.

**-l<[=id]>**

List projection identifiers that can be selected with +*proj*. `cs2cs -l=id` gives expanded description of projection *id*, e.g. `cs2cs -l=merc`.

**-lp**

List of all projection id that can be used with the +*proj* parameter. Equivalent to `cs2cs -l`.

**-lP**

Expanded description of all projections that can be used with the +*proj* parameter.

**-le**

List of all ellipsoids that can be selected with the +*ellps* parameters.

**-lm**

List of hard-coded prime meridians that can be selected with the +*pm* parameter. Note that this list is no longer updated, and some values may conflict with other sources.

**-lu**

List of all distance units that can be selected with the +*units* parameter.

**-r**

This options reverses the order of the first two expected inputs from that specified by the CRS to the opposite order. The third coordinate, typically height, remains third.

**-s**

This options reverses the order of the first two expected outputs from that specified by the CRS to the opposite order. The third coordinate, typically height, remains third.

**-f** <format>

Where *format* is a printf format string to control the form of the output values. For inverse projections, the output will be in degrees when this option is employed. If a format is specified for inverse projection the output data will be in decimal degrees. The default format is "%.2f" for forward projection and DMS for inverse.

**-w<n>**

Where *n* is the number of significant fractional digits to employ for seconds output (when the option is not specified, -w3 is assumed).

**-W<n>**

Where *n* is the number of significant fractional digits to employ for seconds output. When -W is employed the fields will be constant width with leading zeroes. Valid range: -W0 through -W8.

**-v**

Causes a listing of cartographic control parameters tested for and used by the program to be printed prior to input data.

**--area** <name_or_code>

Added in version 8.0.0.

Specify an area of interest to restrict the results when researching coordinate operations between 2 CRS. The area of interest can be specified either as a name (e.g "Denmark - onshore") or a AUTHORITY:CODE (EPSG:3237)

This option is mutually exclusive with *--bbox*.

**--bbox** <west_long,south_lat,east_long,north_lat>

Added in version 8.0.0.

Specify an area of interest to restrict the results when researching coordinate operations between 2 CRS. The area of interest is specified as a bounding box with geographic coordinates, expressed in degrees in a unspecified geographic CRS. *west_long* and *east_long* should be in the [-180,180] range, and *south_lat* and *north_lat* in the [-90,90]. *west_long* is generally lower than *east_long*, except in the case where the area of interest crosses the antimeridian.

**--only-best**[=yes|=no]

Added in version 9.2.0.

Force *cs2cs* to only use the best transformation known by PROJ. *cs2cs* will return an error if a grid needed for the best transformation is missing.

Best transformation should be understood as the most accurate transformation available among all relevant for the point to transform, and if all known grids required to perform such transformation were accessible (either locally or through network).

Note that the default value for this option can be also set with the PROJ_ONLY_BEST_DEFAULT environment variable, or with the only_best_default setting of *proj.ini* (*--only-best* when specified overrides such default value).

**--no-ballpark**

Added in version 8.0.0.

Disallow any coordinate operation that is, or contains, a *Ballpark transformation*

**--accuracy** <accuracy>

Added in version 8.0.0.

Sets the minimum desired accuracy for candidate coordinate operations.

**--authority** <name>

Added in version 8.0.0.

This option can be used to restrict the authority of coordinate operations looked up in the database. When not specified, coordinate operations from any authority will be searched, with the restrictions set in the `authority_to_authority_preference` database table related to the authority of the source/target CRS themselves. If authority is set to `any`, then coordinate operations from any authority will be searched If authority is a non-empty string different of `any`, then coordinate operations will be searched only in that authority namespace (e.g `EPSG`).

This option is mutually exclusive with *--bbox*.

**--3d**

Added in version 9.1.

"Promote" 2D CRS(s) to their 3D version, where the vertical axis is the ellipsoidal height in metres, using the ellipsoid of the base geodetic CRS. Depending on PROJ versions and the exact nature of the CRS involved, especially before PROJ 9.1, a mix of 2D and 3D CRS could lead to 2D or 3D transformations. Starting with PROJ 9.1, both CRS need to be 3D for vertical transformation to possibly happen.

**--s_epoch**

Added in version 9.4.

Epoch of coordinates in the source CRS, as decimal year. Only applies to a dynamic CRS.

**--t_epoch**

Added in version 9.4.

Epoch of coordinates in the target CRS, as decimal year. Only applies to a dynamic CRS.

The *cs2cs* program requires two coordinate reference system (CRS) definitions. The first (or primary is defined based on all projection parameters not appearing after the *+to* argument. All projection parameters appearing after the *+to* argument are considered the definition of the second CRS. If there is no second CRS defined, a geographic CRS based on the datum and ellipsoid of the source CRS is assumed. Note that the source and destination CRS can both of same or different nature (geographic, projected, compound CRS), or one of each and may have the same or different datums.

When using a WKT definition or a AUTHORITY:CODE, the axis order of the CRS will be enforced. So for example if using EPSG:4326, the first value expected (or returned) will be a latitude.

Internally, *cs2cs* uses the *proj_create_crs_to_crs()* function to compute the appropriate coordinate operation, so implementation details of this function directly impact the results returned by the program.

The environment parameter *PROJ_DATA* establishes the directory for resource files (database, datum shift grids, etc.)

One or more files (processed in left to right order) specify the source of data to be transformed. A - will specify the location of processing standard input. If no files are specified, the input is assumed to be from stdin. For input data the two data values must be in the first two white space separated fields and when both input and output are ASCII all trailing portions of the input line are appended to the output line.

Input geographic data (longitude and latitude) must be in DMS or decimal degrees format and input cartesian data must be in units consistent with the ellipsoid major axis or sphere radius units. Output geographic coordinates will normally be in DMS format (use `-f %.12f` for decimal degrees with 12 decimal places), while projected (cartesian) coordinates will be in linear (meter, feet) units.

#### 6.2.2.1 Use of remote grids

Added in version 7.0.0.

If the *PROJ_NETWORK* environment variable is set to ON, *cs2cs* will attempt to use remote grids stored on CDN (Content Delivery Network) storage, when they are not available locally.

More details are available in the *Network capabilities* section.

### 6.2.3 Examples

#### 6.2.3.1 Using PROJ strings

The following script

```
cs2cs +proj=latlong +datum=NAD83 +to +proj=utm +zone=10 +datum=NAD27 -r <<EOF
45°15'33.1" 111.5W
45d15.551666667N -111d30
+45.25919444444 111d30'000w
EOF
```

will transform the input NAD83 geographic coordinates into NAD27 coordinates in the UTM projection with zone 10 selected. The geographic values of this example are equivalent and meant as examples of various forms of DMS input. The x-y output data will appear as three lines of:

```
1402285.93  5076292.58 0.00
```

To get those exact values, you have need to have all current grids installed (in that instance the NADCON5 `us_noaa_nadcon5_nad27_nad83_1986_conus.tif` grid) locally or use networking capabilities mentioned above.

To make sure you will get the optimal result, you may add *--only-best*. Assuming the above mentioned grid is *not* available,

```
echo -111.5 45.25919444444 | cs2cs --only-best +proj=latlong +datum=NAD83 +to +proj=utm␣
↪+zone=10 +datum=NAD27
```

would return:

```
Attempt to use coordinate operation axis order change (2D) + Inverse of NAD27 to NAD83␣
↪(7) + axis order change (2D) + UTM zone 10N failed. Grid us_noaa_nadcon5_nad27_nad83_
↪1986_conus.tif is not available. Consult https://proj.org/resource_files.html for␣
↪guidance.
*   * inf
```

Otherwise, if you don't have the grid available and you don't specify *--only-best*:

```
echo -111.5 45.25919444444 | cs2cs --only-best +proj=latlong +datum=NAD83 +to +proj=utm␣
↪+zone=10 +datum=NAD27
```

would return:

```
1402224.57  5076275.42 0.00
```

which is the result when the NAD27 and NAD83 datums are dealt as identical, which is an approximation at a level of several tens of metres.

#### 6.2.3.2 Using EPSG CRS codes

Transforming from WGS 84 latitude/longitude (in that order) to UTM Zone 31N/WGS 84

```
cs2cs EPSG:4326 EPSG:32631 <<EOF
45N 2E
EOF
```

outputs

```
421184.70   4983436.77 0.00
```

#### 6.2.3.3 Using EPSG CRS names

Transforming from WGS 84 latitude/longitude (in that order) with EGM96 height to UTM Zone 31N/WGS 84 with WGS84 ellipsoidal height

```
echo 45 2 0 | cs2cs "WGS 84 + EGM96 height" "WGS 84 / UTM zone 31N" --3d
```

outputs

```
421184.70   4983436.77 50.69
```

**Note**

To get those exact values, you have need to have the EGM96 grid installed locally or use networking capabilities mentioned above.

## 6.3 geod

## 6.4 invgeod

### 6.4.1 Synopsis

**geod** +*ellps*=<*ellipse*> [**-afFIlptwW** [args]] [+*opt[=arg]* ...] file ...

**invgeod** +*ellps*=<*ellipse*> [**-afFIlptwW** [args]] [+*opt[=arg]* ...] file ...

### 6.4.2 Description

*geod* (direct) and *invgeod* (inverse) perform geodesic (Great Circle) computations for determining latitude, longitude and back azimuth of a terminus point given a initial point latitude, longitude, azimuth and distance (direct) or the forward and back azimuths and distance between an initial and terminus point latitudes and longitudes (inverse). The results are accurate to round off for $|f| < 1/50$, where $f$ is flattening.

*invgeod* may not be available on all platforms; in this case use `geod -I` instead.

The following command-line options can appear in any order:

**-I**

Specifies that the inverse geodesic computation is to be performed. May be used with execution of *geod* as an alternative to *invgeod* execution.

**-a**

Latitude and longitudes of the initial and terminal points, forward and back azimuths and distance are output.

**-t<a>**

Where *a* specifies a character employed as the first character to denote a control line to be passed through without processing.

**-le**

Gives a listing of all the ellipsoids that may be selected with the +*ellps*= option.

**-lu**

Gives a listing of all the units that may be selected with the +*units*= option. (Default units are meters.)

**-f** <format>

Where *format* is a printf format string to control the output form of the geographic coordinate and azimuth values. The default mode is DMS.

**-F** <format>

Where *format* is a printf format string to control the output form of the distance value. The default mode is "%.3f".

**-w<n>**

Where *n* is the number of significant fractional digits to employ for seconds output (when the option is not specified, -w3 is assumed).

**-W<n>**

Where *n* is the number of significant fractional digits to employ for seconds output. When -W is employed the fields will be constant width with leading zeroes.

**-p**

This option causes the azimuthal values to be output as unsigned DMS numbers between 0 and 360 degrees. Also note -f.

The +*opt* command-line options are associated with geodetic parameters for specifying the ellipsoidal or sphere to use. controls. The options are processed in left to right order from the command line. Reentry of an option is ignored with the first occurrence assumed to be the desired value.

One or more files (processed in left to right order) specify the source of data to be transformed. A - will specify the location of processing standard input. If no files are specified, the input is assumed to be from stdin.

For direct determinations input data must be in latitude, longitude, azimuth and distance order and output will be latitude, longitude and back azimuth of the terminus point. Latitude, longitude of the initial and terminus point are input for the inverse mode and respective forward and back azimuth from the initial and terminus points are output along with the distance between the points.

Input geographic coordinates (latitude and longitude) and azimuthal data must be in decimal degrees or DMS format and input distance data must be in units consistent with the ellipsoid major axis or sphere radius units. The latitude must lie in the range [-90d,90d]. Output geographic coordinates and azimuths will be in DMS (if the -f switch is not employed) to 0.001" with trailing, zero-valued minute-second fields deleted. Output distance data will be in the same units as the ellipsoid or sphere radius.

The Earth's ellipsoidal figure may be selected in the same manner as program *proj* by using +*ellps*=, +*a*=, +*es*=, etc.

*geod* may also be used to determine intermediate points along either a geodesic line between two points or along an arc of specified distance from a geographic point. In both cases an initial point must be specified with +*lat_1*=*lat* and +*lon_1*=*long* parameters and either a terminus point +*lat_2*=*lat* and +*lon_2*=*long* or a distance and azimuth from the initial point with +*S*=*distance* and +*A*=*azimuth* must be specified.

If points along a geodesic are to be determined then either +*n_S*=*integer* specifying the number of intermediate points and/or +*del_S*=*distance* specifying the incremental distance between points must be specified.

To determine points along an arc equidistant from the initial point both +*del_A*=*angle* and +*n_A*=*integer* must be specified which determine the respective angular increments and number of points to be determined.

### 6.4.3 Examples

The following script determines the geodesic azimuths and distance in U.S. statute miles from Boston, MA, to Portland, OR:

```
geod +ellps=clrk66 -I +units=us-mi <<EOF
42d15'N 71d07'W 45d31'N 123d41'W
EOF
```

which gives the results:

```
-66d31'50.141" 75d39'13.083" 2587.504
```

where the first two values are the azimuth from Boston to Portland, the back azimuth from Portland to Boston followed by the distance.

An example of forward geodesic use is to use the Boston location and determine Portland's location by azimuth and distance:

```
geod +ellps=clrk66 +units=us-mi <<EOF
42d15'N 71d07'W -66d31'50.141" 2587.504
EOF
```

which gives:

```
45d31'0.003"N 123d40'59.985"W 75d39'13.094"
```

i.e., the latitude and longitude of Portland, and the back azimuth from Portland to Boston.

> **Note**
>
> Lack of precision in the distance value compromises the precision of the Portland location.

### 6.4.4 Further reading

1. GeographicLib.
2. C. F. F. Karney, Algorithms for Geodesics, J. Geodesy **87**(1), 43–55 (2013); addenda.
3. A geodesic bibliography.

## 6.5 gie

### 6.5.1 Synopsis

**gie** [ **-hovql** [ args ] ] file[s]

### 6.5.2 Description

*gie*, the Geospatial Integrity Investigation Environment, is a regression testing environment for the PROJ transformation library. Its primary design goal is to be able to perform regression testing of code that are a part of PROJ, while not requiring any other kind of tooling than the same C compiler already employed for compiling the library.

**-h, --help**

Print usage information

**-o** <file>, **--output** <file>

Specify output file name

**-v**, **--verbose**

Verbose: Provide non-essential informational output. Repeat *-v* for more verbosity (e.g. **-vv**)

**-q**, **--quiet**

Quiet: Opposite of verbose. In quiet mode not even errors are reported. Only interaction is through the return code (0 on success, non-zero indicates number of FAILED tests)

**-l**, **--list**

List the PROJ internal system error codes

**--version**

Print version number

Tests for *gie* are defined in simple text files. Usually having the extension **.gie**. Test for *gie* are written in the purpose-build command language for gie. The basic functionality of the gie command language is implemented through just 3 command verbs: **operation**, which defines the PROJ operation to test, **accept**, which defines the input coordinate to read, and **expect**, which defines the result to expect.

A sample test file for *gie* that uses the three above basic commands looks like:

```
<gie>

-------------------------------------------
Test output of the UTM projection
-------------------------------------------
operation  +proj=utm  +zone=32  +ellps=GRS80
-------------------------------------------
accept     12  55
expect     691_875.632_14   6_098_907.825_05

</gie>
```

Parsing of a *gie* file starts at **<gie>** and ends when **</gie>** is reached. Anything before **<gie>** and after **</gie>** is not considered. Test cases are created by defining an *operation* which *accept* an input coordinate and *expect* an output coordinate.

Because *gie* tests are wrapped in the **<gie>**/**</gie>** tags it is also possible to add test cases to custom made *init files*. The tests will be ignore by PROJ when reading the init file with *+init* and *gie* ignores anything not wrapped in **<gie>**/**</gie>**.

*gie* tests are defined by a set of commands like *operation*, *accept* and *expect* in the example above. Together the commands make out the *gie* command language. Any line in a *gie* file that does not start with a command is ignored. In the example above it is seen how this can be used to add comments and styling to *gie* test files in order to make them more readable as well as documenting what the purpose of the various tests are.

Below the *gie command language* is explained in details.

### 6.5.3 Examples

1. Run all tests in a file with all debug information turned on

```
gie -vvvv corner-cases.gie
```

2. Run all tests in several files

```
gie foo bar
```

### 6.5.4 gie command language

**operation** <+args>

Define a PROJ operation to test. Example:

```
operation proj=utm zone=32 ellps=GRS80
# test 4D function
accept    12 55 0 0
expect    691875.63214  6098907.82501  0  0

# test 2D function
accept    12 56
expect    687071.4391   6210141.3267
```

**accept** <x y [z [t]]>

Define the input coordinate to read. Takes test coordinate. The coordinate can be defined by either 2, 3 or 4 values, where the first two values are the x- and y-components, the 3rd is the z-component and the 4th is the time component. The number of components in the coordinate determines which version of the operation is tested (2D, 3D or 4D). Many coordinates can be accepted for one *operation*. For each *accept* an accompanying *expect* is needed.

Note that *gie* accepts the underscore (_) as a thousands separator. It is not required (in fact, it is entirely ignored by the input routine), but it significantly improves the readability of the very long strings of numbers typically required in projected coordinates.

See *operation* for an example.

**expect** <x y [z [t]]> | <error code>

Define the expected coordinate that will be returned from accepted coordinate passed though an operation. The expected coordinate can be defined by either 2, 3 or 4 components, similarly to *accept*. Many coordinates can be expected for one *operation*. For each *expect* an accompanying *accept* is needed.

See *operation* for an example.

In addition to expecting a coordinate it is also possible to expect a PROJ error code in case an operation can't be created. This is useful when testing that errors are caught and handled correctly. Below is an example of that tests that the pipeline operator fails correctly when a non-invertible pipeline is constructed.

```
operation   proj=pipeline step
            proj=urm5 n=0.5 inv
expect      failure pjd_err_malformed_pipeline
```

See *gie --list* for a list of error codes that can be expected.

**tolerance** <tolerance>

The *tolerance* command controls how much accepted coordinates can deviate from the expected coordinate. This is handy to test that an operation meets a certain numerical tolerance threshold. Some operations are expected to be accurate within millimeters where others might only be accurate within a few meters. *tolerance* should

```
operation       proj=merc
# test coordinate as returned by ```echo 12 55 | proj +proj=merc``
tolerance       1 cm
```

(continues on next page)

(continued from previous page)

```
accept          12 55
expect          1335833.89 7326837.72

# test that the same coordinate with a 50 m false easting as determined
# by ``echo 12 55 |proj +proj=merc +x_0=50`` is still within a 100 m
# tolerance of the unaltered coordinate from proj=merc
tolerance       100 m
accept          12 55
expect          1335883.89  7326837.72
```

The default tolerance is 0.5 mm. See *proj -lu* for a list of possible units.

**roundtrip** <n> <tolerance>

Do a roundtrip test of an operation. *roundtrip* needs a *operation* and a *accept* command to function. The accepted coordinate is passed to the operation first in it's forward mode, then the output from the forward operation is passed back to the inverse operation. This procedure is done n times. If the resulting coordinate is within the set tolerance of the initial coordinate, the test is passed.

Example with the default 100 iterations and the default tolerance:

```
operation       proj=merc
accept          12 55
roundtrip
```

Example with count and default tolerance:

```
operation       proj=merc
accept          12 55
roundtrip       10000
```

Example with count and tolerance:

```
operation       proj=merc
accept          12 55
roundtrip       10000 5 mm
```

**direction** <direction>

The *direction* command specifies in which direction an operation is performed. This can either be `forward` or `inverse`. An example of this is seen below where it is tested that a symmetrical transformation pipeline returns the same results in both directions.

```
operation proj=pipeline zone=32 step
          proj=utm  ellps=GRS80 step
          proj=utm  ellps=GRS80 inv
tolerance 0.1 mm

accept 12 55 0 0
expect 12 55 0 0

# Now the inverse direction (still same result: the pipeline is symmetrical)

direction inverse
expect 12 55 0 0
```

The default direction is "forward".

**ignore** <error code>

This is especially useful in test cases that rely on a grid that is not guaranteed to be available. Below is an example of that situation.

```
operation proj=hgridshift +grids=nzgd2kgrid0005.gsb ellps=GRS80
tolerance 1 mm
ignore    pjd_err_failed_to_load_grid
accept    172.999892181021551 -45.001620431954613
expect    173                 -45
```

See *gie --list* for a list of error codes that can be ignored.

**require_grid** <grid_name>

Checks the availability of the grid <grid_name>. If it is not found, then all *accept*/*expect* pairs until the next *operation* will be skipped. *require_grid* can be repeated several times to specify several grids whose presence is required.

**echo** <text>

Add user defined text to the output stream. See the example below.

```
<gie>
echo ** Mercator projection tests **
operation +proj=merc
accept  0   0
expect  0   0
</gie>
```

which returns

```
--------------------------------------------------------------------------------
Reading file 'test.gie'
** Mercator projection test **
--------------------------------------------------------------------------------
total:  1 tests succeeded,  0 tests skipped,  0 tests failed.
--------------------------------------------------------------------------------
```

**skip**

Skip any test after the first occurrence of *skip*. In the example below only the first test will be performed. The second test is skipped. This feature is mostly relevant for debugging when writing new test cases.

```
<gie>
operation proj=merc
accept  0   0
expect  0   0
skip
accept  0   1
expect  0   110579.9
</gie>
```

### 6.5.5 Strict mode

Added in version 7.1.

A stricter variant of normal gie syntax can be used by wrapping gie commands between <gie-strict> and </gie-strict>. In strict mode, comment lines must start with a sharp character. Unknown commands will be considered as an error. A command can still be split on several lines, but intermediate lines must end with the space character followed by backslash to mark the continuation.

```
<gie-strict>
# This is a comment. The following line with multiple repeated characters too
------------------------------------------------
# A command on several lines must use " \" continuation
operation proj=hgridshift +grids=nzgd2kgrid0005.gsb \
          ellps=GRS80
tolerance 1 mm
ignore    pjd_err_failed_to_load_grid
accept    172.999892181021551 -45.001620431954613
expect    173                 -45
</gie-strict>
```

### 6.5.6 Background

More importantly than being an acronym for "Geospatial Integrity Investigation Environment", gie were also the initials, user id, and USGS email address of Gerald Ian Evenden (1935--2016), the geospatial visionary, who, already in the 1980s, started what was to become the PROJ of today.

Gerald's clear vision was that map projections are *just special functions*. Some of them rather complex, most of them of two variables, but all of them *just special functions*, and not particularly more special than the `sin()`, `cos()`, `tan()`, and `hypot()` already available in the C standard library.

And hence, according to Gerald, *they should not be particularly much harder to use*, for a programmer, than the `sin()`'s, `tan()`'s and `hypot()`'s so readily available.

Gerald's ingenuity also showed in the implementation of the vision, where he devised a comprehensive, yet simple, system of key-value pairs for parameterising a map projection, and the highly flexible *PJ* struct, storing run-time compiled versions of those key-value pairs, hence making a map projection function call, `pj_fwd(PJ, point)`, as easy as a traditional function call like `hypot(x,y)`.

While today, we may have more formally well defined metadata systems (most prominent the OGC WKT2 representation), nothing comes close being as easily readable ("human compatible") as Gerald's key-value system. This system in particular, and the PROJ system in general, was Gerald's great gift to anyone using and/or communicating about geodata.

It is only reasonable to name a program, keeping an eye on the integrity of the PROJ system, in honour of Gerald.

So in honour, and hopefully also in the spirit, of Gerald Ian Evenden (1935--2016), this is the Geospatial Integrity Investigation Environment.

## 6.6 proj

## 6.7 invproj

### 6.7.1 Synopsis

**proj** [**-beEfiIlmorsStTvVwW**] [args]] ([+*opt[=arg]* ...] | {crs}) file ...

**invproj** [**-beEfiIlmorsStTvVwW**] [args]] ([+*opt[=arg]* ...] | {crs}) file ...

### 6.7.2 Description

*proj* and *invproj* perform respective forward and inverse conversion of cartographic data to or from cartesian data with a wide range of selectable projection functions.

*invproj* may not be available on all platforms; in this case use `proj -I` instead.

The following control parameters can appear in any order

**-b**

Special option for binary coordinate data input and output through standard input and standard output. Data is assumed to be in system type double floating point words. This option is to be used when *proj* is a child process and allows bypassing formatting operations.

**-d** <n>

Added in version 5.2.0: Specify the number of decimals to round to in the output.

**-i**

Selects binary input only (see `-b`).

**-I**

Alternate method to specify inverse projection. Redundant when used with *invproj*.

**-o**

Selects binary output only (see `-b`).

**-t<a>**

Where *a* specifies a character employed as the first character to denote a control line to be passed through without processing. This option applicable to ASCII input only. (# is the default value).

**-e** <string>

Where *string* is an arbitrary string to be output if an error is detected during data transformations. The default value is a three character string: `*\t*`. Note that if the `-b`, `-i` or `-o` options are employed, an error is returned as HUGE_VAL value for both return values.

**-E**

Causes the input coordinates to be copied to the output line prior to printing the converted values.

**-l<[**=id]>

List projection identifiers that can be selected with +*proj*. `proj -l=id` gives expanded description of projection *id*, e.g. `proj -l=merc`.

**-lp**

List of all projection id that can be used with the +*proj* parameter. Equivalent to `proj -l`.

**-lP**

Expanded description of all projections that can be used with the +*proj* parameter.

**-le**

List of all ellipsoids that can be selected with the +*ellps* parameters.

**-lu**

List of all distance units that can be selected with the +*units* parameter.

**-r**

This options reverses the order of the expected input from longitude-latitude or x-y to latitude-longitude or y-x.

**-s**

This options reverses the order of the output from x-y or longitude-latitude to y-x or latitude-longitude.

**-S**

Causes estimation of meridional and parallel scale factors, area scale factor and angular distortion, and maximum and minimum scale factors to be listed between <> for each input point. For conformal projections meridional and parallel scales factors will be equal and angular distortion zero. Equal area projections will have an area factor of 1.

**-m** <mult>

The cartesian data may be scaled by the *mult* parameter. When processing data in a forward projection mode the cartesian output values are multiplied by *mult* otherwise the input cartesian values are divided by *mult* before inverse projection. If the first two characters of *mult* are 1/ or 1: then the reciprocal value of *mult* is employed.

**-f** <format>

Where *format* is a printf format string to control the form of the output values. For inverse projections, the output will be in degrees when this option is employed. The default format is "%.2f" for forward projection and DMS for inverse.

**-w<n>**

Where *n* is the number of significant fractional digits to employ for seconds output (when the option is not specified, -w3 is assumed).

**-W<n>**

Where *n* is the number of significant fractional digits to employ for seconds output. When -W is employed the fields will be constant width with leading zeroes.

**-v**

Causes a listing of cartographic control parameters tested for and used by the program to be printed prior to input data.

**-V**

This option causes an expanded annotated listing of the characteristics of the projected point. -v is implied with this option.

The +*opt* run-line arguments are associated with cartographic parameters. Additional projection control parameters may be contained in two auxiliary control files: the first is optionally referenced with the +*init=file:id* and the second is always processed after the name of the projection has been established from either the run-line or the contents of +init file. The environment parameter PROJ_DATA establishes the default directory for a file reference without an absolute path. This is also used for supporting files like datum shift files.

Added in version 9.3.0: *{crs}* is one of the possibilities accepted by :c:proj_create(), provided it expresses a projected CRS, like a WKT string, an object code (like "EPSG:32632") a PROJJSON string, etc. The projection computed will be those of the map projection implied by the transformation from the base geographic CRS of the projected CRS to the projected CRS.

One or more files (processed in left to right order) specify the source of data to be converted. A - will specify the location of processing standard input. If no files are specified, the input is assumed to be from stdin. For ASCII input data the two data values must be in the first two white space separated fields and when both input and output are ASCII all trailing portions of the input line are appended to the output line.

Input geographic data (longitude and latitude) must be in DMS or decimal degrees format and input cartesian data must be in units consistent with the ellipsoid major axis or sphere radius units. Output geographic coordinates will be in DMS (if the -w switch is not employed) and precise to 0.001" with trailing, zero-valued minute-second fields deleted.

### 6.7.3 Example

The following script

```
proj +proj=utm +zone=12 -r <<EOF
45d15'33.1" 111.5W
45d15.551666667N -111d30
+45.25919444444 111d30'000w
EOF
```

will perform UTM zone 12 forward projection. The default +ellps=GRS80 is used as no +ellps was specified. The geographic values of this example are equivalent and meant as examples of various forms of DMS input. The x-y output data will appear as three lines of:

```
460770.43     5011865.86
```

This other example

```
proj EPSG:6421 -V <<EOF
-120 35.8
EOF
```

Will perform the projection of the coordinates in "NAD83(2011) / California zone 4" (*EPSG:6421*) into its geographic system, "NAD83(2011)", showing the expanded annotated listing. The output will appear as:

```
#Lambert Conformal Conic
#   Conic, Sph&Ell
#   lat_1= and lat_2= or lat_0, k_0=
# +proj=lcc +lat_0=35.3333333333333 +lon_0=-119 +lat_1=37.25 +lat_2=36
# +x_0=2000000 +y_0=500000 +ellps=GRS80
#Final Earth figure: ellipsoid
#  Major axis (a): 6378137.000
#  1/flattening: 298.257222
#  squared eccentricity: 0.006694380023
Longitude: 120dW [ -120 ]
Latitude:  35d48'N [ 35.8 ]
Easting (x):   1909606.87
Northing (y):  552253.58
Meridian scale (h) : 1.00004382  ( 0.004382 % error )
Parallel scale (k) : 1.00004382  ( 0.004382 % error )
Areal scale (s):     1.00008765  ( 0.008765 % error )
Angular distortion (w): 0.000
Meridian/Parallel angle: 90.00000
Convergence : -0d35'47.714" [ -0.59658715 ]
Max-min (Tissot axis a-b) scale error: 1.00004 1.00004
```

## 6.8 projinfo

### 6.8.1 Synopsis

**projinfo**

[-o formats] [-k crs|operation|datum|ensemble|ellipsoid] [--summary] [-q]
[[--area name_or_code] | [--bbox west_long,south_lat,east_long,north_lat]]
[--spatial-test contains|intersects]
[--crs-extent-use none|both|intersection|smallest]
[--grid-check none|discard_missing|sort|known_available]

[--pivot-crs always|if_no_direct_transformation|never|{auth:code[,auth:code]*}]
[--show-superseded] [--hide-ballpark] [--accuracy {accuracy}]
[--allow-ellipsoidal-height-as-vertical-crs]
[--boundcrs-to-wgs84]
[--authority name]
[--main-db-path path] [--aux-db-path path]*
[--dump-db-structure]
[--identify] [--3d]
[--output-id AUTH:CODE]
[--c-ify] [--single-line]
--searchpaths | --remote-data |
--list-crs [list-crs-filter] |
--dump-db-structure [{object_definition} | {object_reference}] |
{object_definition} | {object_reference} |
(-s {srs_def} [--s_epoch {epoch}] -t {srs_def} [--t_epoch {epoch}]) |
({srs_def} {srs_def})

where {object_definition} or {srs_def} is one of the possibilities accepted by *proj_create()*

- a proj-string,
- a WKT string,
- an object code (like "EPSG:4326", "urn:ogc:def:crs:EPSG::4326", "urn:ogc:def:coordinateOperation:EPSG::1671"),
- an Object name. e.g "WGS 84", "WGS 84 / UTM zone 31N". In that case as uniqueness is not guaranteed, heuristics are applied to determine the appropriate best match.
- a CRS name and a coordinate epoch, separated with '@'. For example "ITRF2014@2025.0". (*added in 9.2*)
- a OGC URN combining references for compound coordinate reference systems (e.g "urn:ogc:def:crs,crs:EPSG::2393,crs:EPSG::5717" or custom abbreviated syntax "EPSG:2393+5717"),
- a OGC URN combining references for references for projected or derived CRSs e.g. for Projected 3D CRS "UTM zone 31N / WGS 84 (3D)": "urn:ogc:def:crs,crs:EPSG::4979,cs:PROJ::ENh,coordinateOperation:EPSG::16031" (*added in 6.2*)
- Extension of OGC URN for CoordinateMetadata. e.g. "urn:ogc:def:CoordinateMetadata:NRCAN::NAD83_CSRS_1997_MTM11_HT2_1997"
- a OGC URN combining references for concatenated operations (e.g. "urn:ogc:def:coordinateOperation,coordinateOperation:EPSG::3895,coordinateOperation:EPSG::1618")
- a PROJJSON string. The jsonschema is at https://proj.org/schemas/v0.4/projjson.schema.json (*added in 6.2*)
- a compound CRS made from two object names separated with " + ". e.g. "WGS 84 + EGM96 height" (*added in 7.1*)

{object_reference} is a filename preceded by the '@' character. The file referenced by the {object_reference} must contain a valid {object_definition}.

The usage of "{srs_def} {srs_def}" is equivalent to "-s {srs_def} -t {srs_def}" (*added in 9.5*).

### 6.8.2 Description

*projinfo* is a program that can query information on a geodetic object, coordinate reference system (CRS) or coordinate operation, when the `-s` and `-t` options are specified, and display it under different formats (PROJ string, WKT string or PROJJSON string).

It can also be used to query coordinate operations available between two CRS.

The program is named with some reference to the GDAL gdalsrsinfo utility that offers partly similar services.

The following control parameters can appear in any order:

**`-o` formats**

formats is a comma separated combination of: `all`, `default`, `PROJ`, `WKT_ALL`, `WKT2:2015`, `WKT2:2019`, `WKT1:GDAL`, `WKT1:ESRI`, `PROJJSON`, `SQL`.

Except `all` and `default`, other formats can be preceded by `-` to disable them.

> **Note**
>
> WKT2_2019 was previously called WKT2_2018.

> **Note**
>
> Before PROJ 6.3.0, WKT1:GDAL was implicitly calling --boundcrs-to-wgs84. This is no longer the case.

> **Note**
>
> When SQL is specified, *`--output-id`* must be specified.

**`-k` `crs|operation|datum|ensemble|ellipsoid`**

When used to query a single object with a AUTHORITY:CODE, determines the (k)ind of the object in case there are CRS, coordinate operations or ellipsoids with the same CODE. The default is crs.

**`--summary`**

When listing coordinate operations available between 2 CRS, return the result in a summary format, mentioning only the name of the coordinate operation, its accuracy and its area of use.

> **Note**
>
> only used for coordinate operation computation

**`-q`**

Turn on quiet mode. Quiet mode is only available for queries on single objects, and only one output format is selected. In that mode, only the PROJ, WKT or PROJJSON string is displayed, without other introduction output. The output is then potentially compatible of being piped in other utilities.

**`--area` `name_or_code`**

Specify an area of interest to restrict the results when researching coordinate operations between 2 CRS. The area of interest can be specified either as a name (e.g "Denmark - onshore") or a AUTHORITY:CODE (EPSG:3237) This option is exclusive of *`--bbox`*.

> **Note**
>
> only used for coordinate operation computation

**--bbox** west_long,south_lat,east_long,north_lat

Specify an area of interest to restrict the results when researching coordinate operations between 2 CRS. The area of interest is specified as a bounding box with geographic coordinates, expressed in degrees in a unspecified geographic CRS. *west_long* and *east_long* should be in the [-180,180] range, and *south_lat* and *north_lat* in the [-90,90]. *west_long* is generally lower than *east_long*, except in the case where the area of interest crosses the antimeridian.

> **Note**
>
> only used for coordinate operation computation

**--spatial-test** contains|intersects

Specify how the area of use of coordinate operations found in the database are compared to the area of use specified explicitly with *--area* or *--bbox*, or derived implicitly from the area of use of the source and target CRS. By default, *projinfo* will only keep coordinate operations whose are of use is strictly within the area of interest (`contains` strategy). If using the `intersects` strategy, the spatial test is relaxed, and any coordinate operation whose area of use at least partly intersects the area of interest is listed.

> **Note**
>
> only used for coordinate operation computation

**--crs-extent-use** none|both|intersection|smallest

Specify which area of interest to consider when no explicit one is specified with *--area* or *--bbox* options. By default (`smallest` strategy), the area of use of the source or target CRS will be looked, and the one that is the smallest one in terms of area will be used as the area of interest. If using `none`, no area of interest is used. If using `both`, only coordinate operations that relate (contain or intersect depending of the *--spatial-test* strategy) to the area of use of both CRS are selected. If using `intersection`, the area of interest is the intersection of the bounding box of the area of use of the source and target CRS

> **Note**
>
> only used for coordinate operation computation

**--grid-check** none|discard_missing|sort|known_available

Specify how the presence or absence of a horizontal or vertical shift grid required for a coordinate operation affects the results returned when researching coordinate operations between 2 CRS. The default strategy is `sort` (if *PROJ_NETWORK* is not defined). In that case, all candidate operations are returned, but the actual availability of the grids is used to determine the sorting order. That is, if a coordinate operation involves using a grid that is not available in the PROJ resource directories (determined by the *PROJ_DATA* environment variable), it will be listed in the bottom of the results. The `none` strategy completely disables the checks of presence of grids and this returns the results as if all the grids where available. The `discard_missing` strategy discards results that involve grids not present in the PROJ resource directories. The `known_available` strategy discards results that involve grids not present in the PROJ resource directories and that are not known of the CDN. This is the default strategy is *PROJ_NETWORK* is set to `ON`.

> **Note**
>
> only used for coordinate operation computation

**--pivot-crs** always|if_no_direct_transformation|never|{auth:code[,auth:code]*}

Determine if intermediate (pivot) CRS can be used when researching coordinate operation between 2 CRS. A typical example is the WGS84 pivot. By default, *projinfo* will consider any potential pivot if there is no direct transformation ( `if_no_direct_transformation`). If using the `never` strategy, only direct transformations between the source and target CRS will be used. If using the `always` strategy, intermediate CRS will be considered even if there are direct transformations. It is also possible to restrict the pivot CRS to consider by specifying one or several CRS by their AUTHORITY:CODE.

> **Note**
>
> only used for coordinate operation computation

**--show-superseded**

When enabled, coordinate operations that are superseded by others will be listed. Note that supersession is not equivalent to deprecation: superseded operations are still considered valid although they have a better equivalent, whereas deprecated operations have been determined to be erroneous and are not considered at all.

> **Note**
>
> only used for coordinate operation computation

**--hide-ballpark**

Added in version 7.1.

Hides any coordinate operation that is, or contains, a *Ballpark transformation*

> **Note**
>
> only used for coordinate operation computation

**--accuracy** {accuracy}

Added in version 8.0.

Sets the minimum desired accuracy for returned coordinate operations.

> **Note**
>
> only used for coordinate operation computation

**--allow-ellipsoidal-height-as-vertical-crs**

Added in version 8.0.

Allows exporting a geographic or projected 3D CRS as a compound CRS whose vertical CRS represents the ellipsoidal height.

> **Note**
>
> only used for CRS, and with WKT1:GDAL output format

**--boundcrs-to-wgs84**

When specified, this option researches a coordinate operation from the base geographic CRS of the single CRS, source or target CRS to the WGS84 geographic CRS, and if found, wraps those CRS into a BoundCRS object. This is mostly to be used for early-binding approaches.

**--authority** name

Specify the name of the authority into which to restrict looks up for objects, when specifying an object by name or when coordinate operations are computed. The default is to allow all authorities.

When used with SQL output, this restricts the authorities to which intermediate objects can belong to (the default is EPSG and PROJ). Note that the authority of the *--output-id* option will also be implicitly added.

**--main-db-path** path

Specify the name and path of the database to be used by *projinfo*. The default is `proj.db` in the PROJ resource directories.

**--aux-db-path** path

Specify the name and path of auxiliary databases, that are to be combined with the main database. Those auxiliary databases must have a table structure that is identical to the main database, but can be partly filled and their entries can refer to entries of the main database. The option may be repeated to specify several auxiliary databases.

**--identify**

When used with an object definition, this queries the PROJ database to find known objects, typically CRS, that are close or identical to the object. Each candidate object is associated with an approximate likelihood percentage. This is useful when used with a WKT string that lacks a EPSG identifier, such as ESRI WKT1. This might also be used with PROJ strings. For example, *+proj=utm +zone=31 +datum=WGS84 +type=crs* will be identified with a likelihood of 70% to EPSG:32631

**--dump-db-structure**

Added in version 8.1.

Outputs the sequence of SQL statements to create a new empty valid auxiliary database. This option can be specified as the only switch of the utility. If also specifying a CRS object and the *--output-id* option, the definition of the object as SQL statements will be appended.

**--list-crs** [list-crs-filter]

Added in version 8.1.

Outputs a list (authority name:code and CRS name) of the filtered CRSs from the database. If no filter is provided all authority names and types of non deprecated CRSs are dumped. list-crs-filter is a comma separated combination of: allow_deprecated,geodetic,geocentric, geographic,geographic_2d,geographic_3d,vertical,projected,compound. Affected by options *--authority*, *--area*, *--bbox* and *--spatial-test*

A visual alternative is the webpage CRS Explorer .

**--3d**

Added in version 6.3.

"Promote" 2D CRS(s) to their 3D version, where the vertical axis is the ellipsoidal height in metres, using the ellipsoid of the base geodetic CRS. Depending on PROJ versions and the exact nature of the CRS involved, especially before PROJ 9.1, a mix of 2D and 3D CRS could lead to 2D or 3D transformations. Starting with PROJ 9.1, both CRS need to be 3D for vertical transformation to possibly happen.

**--output-id**=AUTH:NAME

Added in version 8.1.

Identifier to assign to the object (for SQL output).

It is strongly recommended that new objects should not be added in common registries, such as EPSG, ESRI, IAU, etc. Users should use a custom authority name instead. If a new object should be added to the official EPSG registry, users are invited to follow the procedure explained at https://epsg.org/dataset-change-requests.html.

Combined with *--dump-db-structure*, users can create auxiliary databases, instead of directly modifying the main proj.db database. See the *example how to export to an auxiliary database*.

Those auxiliary databases can be specified through proj_context_set_database_path() or the *PROJ_AUX_DB* environment variable.

**--c-ify**

For developers only. Modify the string output of the utility so that it is easy to put those strings in C/C++ code

**--single-line**

Output PROJ, WKT or PROJJSON strings on a single line, instead of multiple indented lines by default.

**--searchpaths**

Added in version 7.0.

Output the directories into which PROJ resources will be looked for (if not using C API such as proj_context_set_search_paths() that will override them.

**--remote-data**

Added in version 7.0.

Display information regarding if *Network capabilities* is enabled, and the related URL.

**--s_epoch**

Added in version 9.4.

Epoch of coordinates in the source CRS, as decimal year. Only applies to a dynamic CRS.

**--t_epoch**

Added in version 9.4.

Epoch of coordinates in the target CRS, as decimal year. Only applies to a dynamic CRS.

### 6.8.3 Examples

1. Query the CRS object corresponding to EPSG:4326

```
projinfo EPSG:4326
```

Output:

```
PROJ.4 string:
+proj=longlat +datum=WGS84 +no_defs +type=crs

WKT2:2019 string:
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
```

(continues on next page)

(continued from previous page)

```
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    USAGE[
        SCOPE["unknown"],
        AREA["World"],
        BBOX[-90,-180,90,180]],
    ID["EPSG",4326]]
```

2. List the coordinate operations between NAD27 (designed with its CRS name) and NAD83 (designed with its EPSG code 4269) within an area of interest

```
projinfo -s NAD27 -t EPSG:4269 --area "USA - Missouri"
```

Output:

```
DERIVED_FROM(EPSG):1241, NAD27 to NAD83 (1), 0.15 m, USA - CONUS including EEZ

PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 +step +proj=unitconvert \
+xy_in=deg +xy_out=rad +step +proj=hgridshift +grids=conus \
+step +proj=unitconvert +xy_in=rad +xy_out=deg +step +proj=axisswap +order=2,1

WKT2:2019 string:
COORDINATEOPERATION["NAD27 to NAD83 (1)",
    SOURCECRS[
        GEOGCRS["NAD27",
            DATUM["North American Datum 1927",
                ELLIPSOID["Clarke 1866",6378206.4,294.978698213898,
                    LENGTHUNIT["metre",1]]],
            PRIMEM["Greenwich",0,
                ANGLEUNIT["degree",0.0174532925199433]],
            CS[ellipsoidal,2],
                AXIS["geodetic latitude (Lat)",north,
                    ORDER[1],
                    ANGLEUNIT["degree",0.0174532925199433]],
                AXIS["geodetic longitude (Lon)",east,
                    ORDER[2],
                    ANGLEUNIT["degree",0.0174532925199433]]]],
    TARGETCRS[
        GEOGCRS["NAD83",
            DATUM["North American Datum 1983",
                ELLIPSOID["GRS 1980",6378137,298.257222101,
                    LENGTHUNIT["metre",1]]],
            PRIMEM["Greenwich",0,
                ANGLEUNIT["degree",0.0174532925199433]],
            CS[ellipsoidal,2],
```

(continues on next page)

(continued from previous page)

```
                AXIS["geodetic latitude (Lat)",north,
                    ORDER[1],
                    ANGLEUNIT["degree",0.0174532925199433]],
                AXIS["geodetic longitude (Lon)",east,
                    ORDER[2],
                    ANGLEUNIT["degree",0.0174532925199433]]],
    METHOD["CTABLE2"],
    PARAMETERFILE["Latitude and longitude difference file","conus"],
    OPERATIONACCURACY[0.15],
    USAGE[
        SCOPE["unknown"],
        AREA["USA - CONUS including EEZ"],
        BBOX[23.81,-129.17,49.38,-65.69]],
    ID["DERIVED_FROM(EPSG)",1241]]
```

3. Export an object as a PROJJSON string

```
projinfo GDA94 -o PROJJSON -q
```

Output:

```
{
    "type": "GeographicCRS",
    "name": "GDA94",
    "datum": {
        "type": "GeodeticReferenceFrame",
        "name": "Geocentric Datum of Australia 1994",
        "ellipsoid": {
            "name": "GRS 1980",
            "semi_major_axis": 6378137,
            "inverse_flattening": 298.257222101
        }
    },
    "coordinate_system": {
        "subtype": "ellipsoidal",
        "axis": [
        {
            "name": "Geodetic latitude",
            "abbreviation": "Lat",
            "direction": "north",
            "unit": "degree"
        },
        {
            "name": "Geodetic longitude",
            "abbreviation": "Lon",
            "direction": "east",
            "unit": "degree"
        }
        ]
    },
    "area": "Australia - GDA",
    "bbox": {
```

(continues on next page)

(continued from previous page)

```
        "south_latitude": -60.56,
        "west_longitude": 93.41,
        "north_latitude": -8.47,
        "east_longitude": 173.35
    },
    "id": {
        "authority": "EPSG",
        "code": 4283
    }
}
```

4. Exporting the SQL statements to insert a new CRS in an auxiliary database.

```
# Get the SQL statements for a custom CRS
projinfo "+proj=merc +lat_ts=5 +datum=WGS84 +type=crs +title=my_crs" --output-id HOBU:MY_
↪CRS -o SQL -q > my_crs.sql
cat my_crs.sql

# Initialize an auxiliary database with the schema of the reference database
echo ".schema" | sqlite3 /path/to/proj.db | sqlite3 aux.db

# Append the content of the definition of HOBU:MY_CRS
sqlite3 aux.db < my_crs.db

# Check that everything works OK
projinfo --aux-db-path aux.db HOBU:MY_CRS
```

or more simply:

```
# Create an auxiliary database with the definition of a custom CRS.
projinfo "+proj=merc +lat_ts=5 +datum=WGS84 +type=crs +title=my_crs" --output-id HOBU:MY_
↪CRS --dump-db-structure | sqlite3 aux.db

# Check that everything works OK
projinfo --aux-db-path aux.db HOBU:MY_CRS
```

Output:

```
INSERT INTO geodetic_crs VALUES('HOBU','GEODETIC_CRS_MY_CRS','unknown','','geographic 2D
↪','EPSG','6424','EPSG','6326',NULL,0);
INSERT INTO usage VALUES('HOBU','USAGE_GEODETIC_CRS_MY_CRS','geodetic_crs','HOBU',
↪'GEODETIC_CRS_MY_CRS','PROJ','EXTENT_UNKNOWN','PROJ','SCOPE_UNKNOWN');
INSERT INTO conversion VALUES('HOBU','CONVERSION_MY_CRS','unknown','','EPSG','9805',
↪'Mercator (variant B)','EPSG','8823','Latitude of 1st standard parallel',5,'EPSG','9122
↪','EPSG','8802','Longitude of natural origin',0,'EPSG','9122','EPSG','8806','False␣
↪easting',0,'EPSG','9001','EPSG','8807','False northing',0,'EPSG','9001',NULL,NULL,NULL,
↪NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL,0);
INSERT INTO usage VALUES('HOBU','USAGE_CONVERSION_MY_CRS','conversion','HOBU',
↪'CONVERSION_MY_CRS','PROJ','EXTENT_UNKNOWN','PROJ','SCOPE_UNKNOWN');
INSERT INTO projected_crs VALUES('HOBU','MY_CRS','my_crs','','EPSG','4400','HOBU',
↪'GEODETIC_CRS_MY_CRS','HOBU','CONVERSION_MY_CRS',NULL,0);
INSERT INTO usage VALUES('HOBU','USAGE_PROJECTED_CRS_MY_CRS','projected_crs','HOBU','MY_
↪CRS','PROJ','EXTENT_UNKNOWN','PROJ','SCOPE_UNKNOWN');
```

```
PROJ.4 string:
+proj=merc +lat_ts=5 +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs +type=crs

WKT2:2019 string:
PROJCRS["my_crs",
    BASEGEOGCRS["unknown",
        ENSEMBLE["World Geodetic System 1984 ensemble",
            MEMBER["World Geodetic System 1984 (Transit)"],
            MEMBER["World Geodetic System 1984 (G730)"],
            MEMBER["World Geodetic System 1984 (G873)"],
            MEMBER["World Geodetic System 1984 (G1150)"],
            MEMBER["World Geodetic System 1984 (G1674)"],
            MEMBER["World Geodetic System 1984 (G1762)"],
            ELLIPSOID["WGS 84",6378137,298.257223563,
                LENGTHUNIT["metre",1]],
            ENSEMBLEACCURACY[2.0]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["HOBU","GEODETIC_CRS_MY_CRS"]],
    CONVERSION["unknown",
        METHOD["Mercator (variant B)",
            ID["EPSG",9805]],
        PARAMETER["Latitude of 1st standard parallel",5,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8823]],
        PARAMETER["Longitude of natural origin",0,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8802]],
        PARAMETER["False easting",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8806]],
        PARAMETER["False northing",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8807]]],
    CS[Cartesian,2],
        AXIS["(E)",east,
            ORDER[1],
            LENGTHUNIT["metre",1]],
        AXIS["(N)",north,
            ORDER[2],
            LENGTHUNIT["metre",1]],
    ID["HOBU","MY_CRS"]]
```

5. Get the WKT representation of EPSG:25832 in the WKT1:GDAL output format and on a single line

```
projinfo -o WKT1:GDAL --single-line EPSG:25832
```

Output:

```
WKT1:GDAL string:
PROJCS["ETRS89 / UTM zone 32N",GEOGCS["ETRS89",DATUM["European_Terrestrial_Reference_
↪System_1989",SPHEROID["GRS 1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],
↪AUTHORITY["EPSG","6258"]],PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",
```

(continues on next page)

(continued from previous page)

```
↪0.0174532925199433,AUTHORITY["EPSG","9122"]],AUTHORITY["EPSG","4258"]],PROJECTION[
↪"Transverse_Mercator"],PARAMETER["latitude_of_origin",0],PARAMETER["central_meridian",
↪9],PARAMETER["scale_factor",0.9996],PARAMETER["false_easting",500000],PARAMETER["false_
↪northing",0],UNIT["metre",1,AUTHORITY["EPSG","9001"]],AXIS["Easting",EAST],AXIS[
↪"Northing",NORTH],AUTHORITY["EPSG","25832"]]
```

## 6.9 projsync

### 6.9.1 Synopsis

**projsync**

[--endpoint URL]
[--local-geojson-file FILENAME]
([--user-writable-directory] | [--system-directory] | [--target-dir DIRNAME])
[--bbox west_long,south_lat,east_long,north_lat]
[--spatial-test contains|intersects]
[--source-id ID] [--area-of-use NAME]
[--file NAME]
[--all] [--exclude-world-coverage]
[--quiet | --verbose] [--dry-run] [--list-files]
[--no-version-filtering]

### 6.9.2 Description

*projsync* is a program that downloads remote resource files into a local directory. This is an alternative to downloading a proj-data-X.Y.Z archive file, or using the on-demand *networking capabilities* of PROJ.

The following control parameters can appear in any order:

**--endpoint** URL

Defines the URL where to download the master `files.geojson` file and then the resource files. Defaults to the value set in *proj.ini*

**--local-geojson-file** FILENAME

Defines the filename for the master GeoJSON files that references resources. Defaults to `${endpoint}/files.geojson`

**--user-writable-directory**

Specifies that resource files must be downloaded in the *user writable directory*. This is the default.

**--system-directory**

Specifies that resource files must be downloaded in the ${installation_prefix}/share/proj directory. The user launching projsync should make sure it has writing rights in that directory.

**--target-dir** DIRNAME

Directory into which resource files must be downloaded.

**--bbox** west_long,south_lat,east_long,north_lat

Specify an area of interest to restrict the resources to download. The area of interest is specified as a bounding box with geographic coordinates, expressed in degrees in a unspecified geographic CRS. *west_long* and *east_long* should be in the [-180,180] range, and *south_lat* and *north_lat* in the [-90,90]. *west_long* is generally lower than *east_long*, except in the case where the area of interest crosses the antimeridian.

**--spatial-test** contains|intersects

Specify how the extent of the resource files are compared to the area of use specified explicitly with *--bbox*. By default, any resource files whose extent intersects the value specified by *--bbox* will be selected. If using the `contains` strategy, only resource files whose extent is contained in the value specified by *--bbox* will be selected.

**--source-id** ID

Restrict resource files to be downloaded to those whose source_id property contains the ID value. Specifying ? as ID will list all possible values.

**--area-of-use** NAME

Restrict resource files to be downloaded to those whose area_of_use property contains the NAME value. Specifying ? as NAME will list all possible values.

**--file** NAME

Restrict resource files to be downloaded to those whose name property contains the NAME value. Specifying ? as NAME will list all possible values.

**--all**

Ask to download all files.

**--exclude-world-coverage**

Exclude files which have world coverage.

**-q** / --quiet

Quiet mode

**--verbose**

Added in version 8.1.

Verbose mode (more than default)

**--dry-run**

Simulate the behavior of the tool without downloading resource files.

**--list-files**

List file names, with the source_id and area_of_use properties.

**--no-version-filtering**

Added in version 8.1.

By default, projsync only downloads files that are compatible of the PROJ_DATA.VERSION metadata of `proj.db`, taking into account the `version_added` and `version_removed` properties of entries in `files.geojson`. When specifying this switch, all files referenced in `files.geojson` will be candidate (combined with other filters).

At least one of *--list-files*, *--file*, *--source-id*, *--area-of-use*, *--bbox* or *--all* must be specified.

Options *--file*, *--source-id*, *--area-of-use* and *--bbox* are combined with a AND logic.

### 6.9.3 Examples

1. Download all resource files

```
projsync --all
```

2. Download resource files covering specified point and attributed to an agency

```
projsync --source-id fr_ign --bbox 2,49,2,49
```

CHAPTER

SEVEN

# COORDINATE OPERATIONS

Coordinate operations in PROJ are divided into three groups: Projections, conversions and transformations. Projections are purely cartographic mappings of the sphere onto the plane. Technically projections are conversions (according to ISO standards), though in PROJ projections are distinguished from conversions. Conversions are coordinate operations that do not exert a change in reference frame. Operations that do exert a change in reference frame are called transformations.

## 7.1 Projections

Projections are coordinate operations that are technically conversions but since projections are so fundamental to PROJ we differentiate them from conversions.

Projections map the spherical 3D space to a flat 2D space.

### 7.1.1 Adams Hemisphere in a Square

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | adams_hemi |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.1.1 Parameters

**Note**

All parameters are optional.

`+lon_0`=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

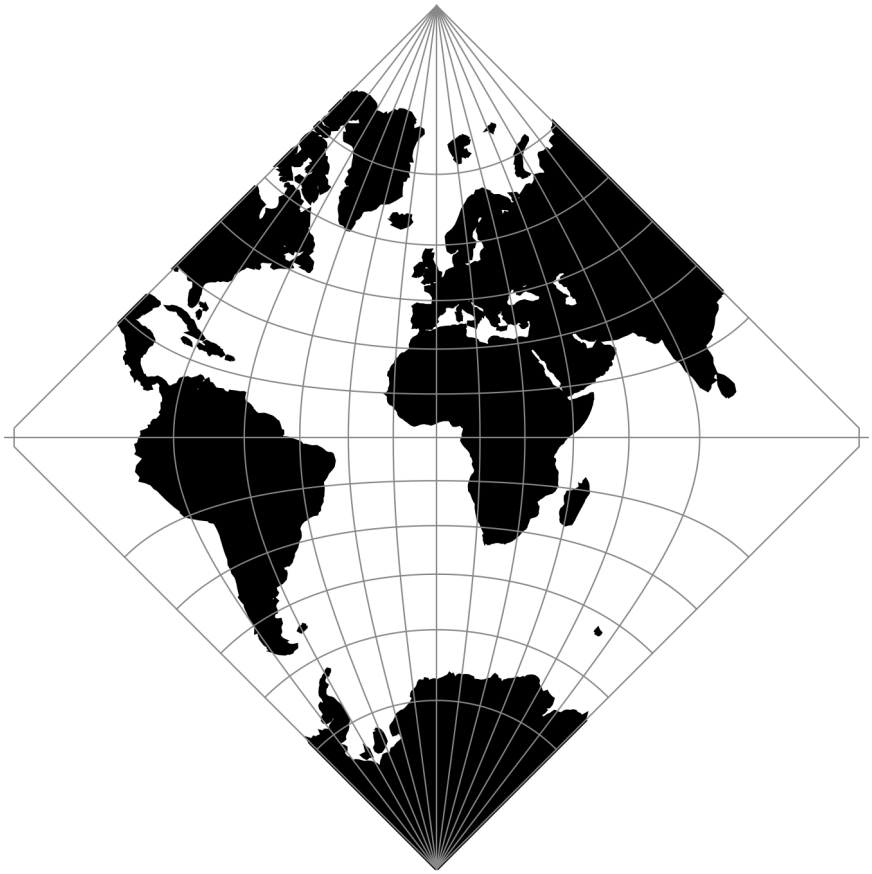


Fig. 1: proj-string: `+proj=adams_hemi`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.2 Adams World in a Square I

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | adams_ws1 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.2.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

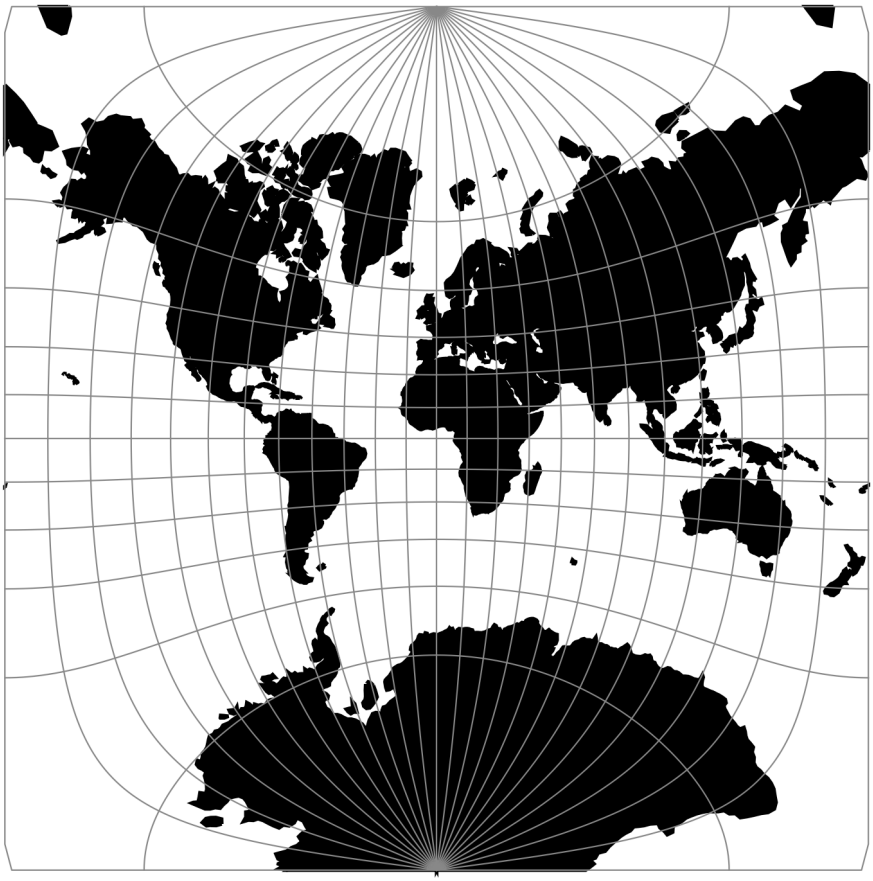


Fig. 2: proj-string: `+proj=adams_ws1`

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.3 Adams World in a Square II

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical |
| **Defined area** | Global |
| **Alias** | adams_ws2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.3.1 Parameters

**Note**

All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

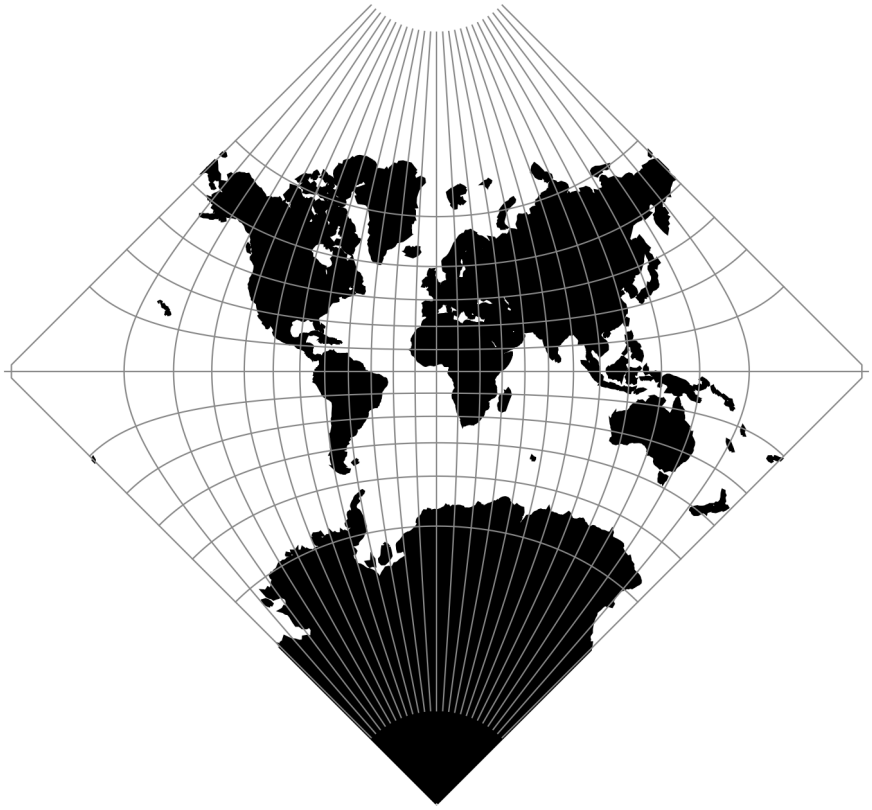


Fig. 3: proj-string: `+proj=adams_ws2`

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.4 Albers Equal Area

| | |
|---|---|
| **Classification** | Conic |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | aea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

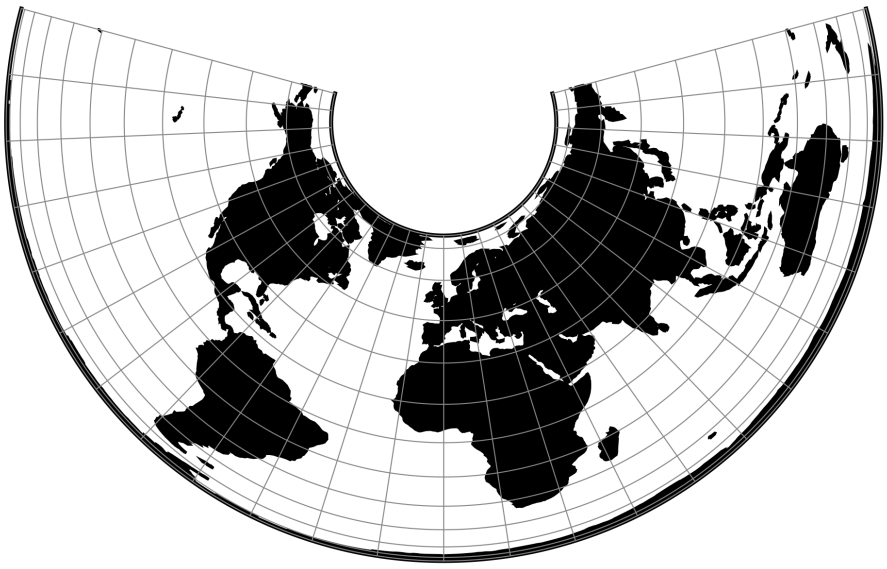


Fig. 4: proj-string: +proj=aea +lat_1=29.5 +lat_2=42.5

#### 7.1.4.1 Options

### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_2=1.570796r*
>
> See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.5 Azimuthal Equidistant

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | aeqd |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.5.1 Options

**Note**

All options are optional for the Azimuthal Equidistant projection.

**+guam**

Use Guam ellipsoidal formulas. Only accurate near the Island of Guam ($\lambda \approx 144.5°$, $\phi \approx 13.5°$)

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

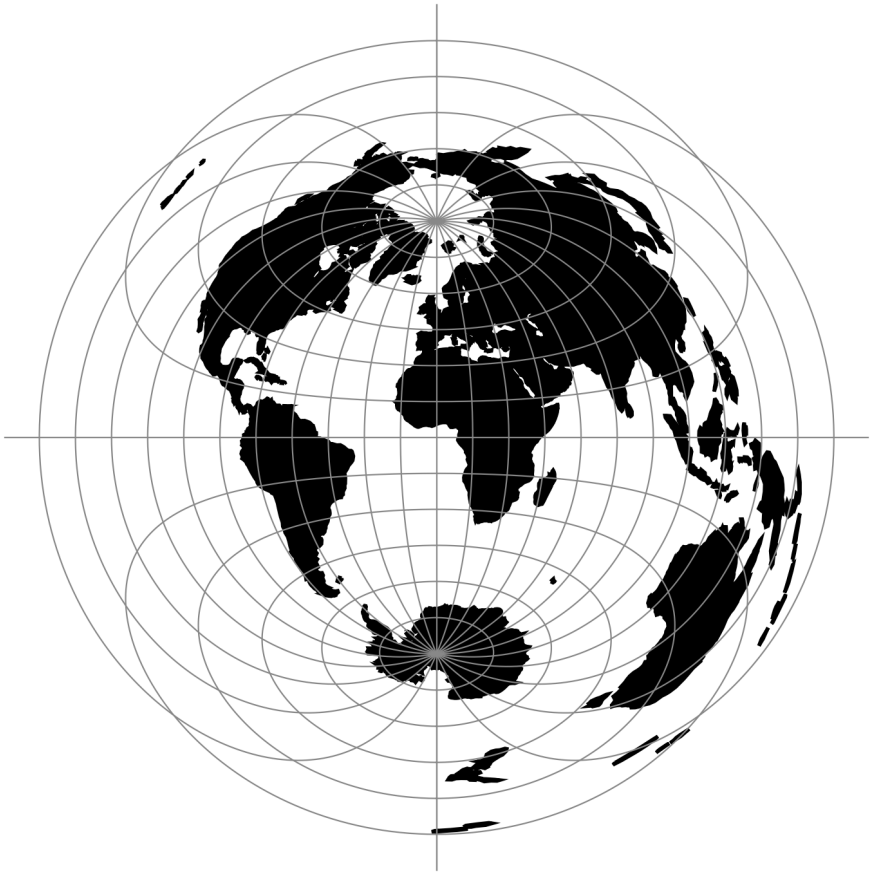


Fig. 5: proj-string: +proj=aeqd

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

### 7.1.6 Airocean

Added in version 9.6.0.

The Airocean map projection is a icosahedron polyhedral projection that intends to lay down continental landmasses into a contiguous map with minimal scale and angular distortion while preserving land surfaces from (most) interruptions. In comparison to the Snyder's Icosahedral Equal Area, this projection relies on a specific way to orient and cut the icosahedron so that most interruptions lie within the ocean. The projection was first invented by Buckminster Fuller in 1943 using a cuboctahedron as basis. The idea to use an icosahedron came later in 1954, and it wasn't until 1978 that the projection was formally mathematically defined [Gray1995].

| | |
|---|---|
| **Classification** | Polyhedral, equal area |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | airocean |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

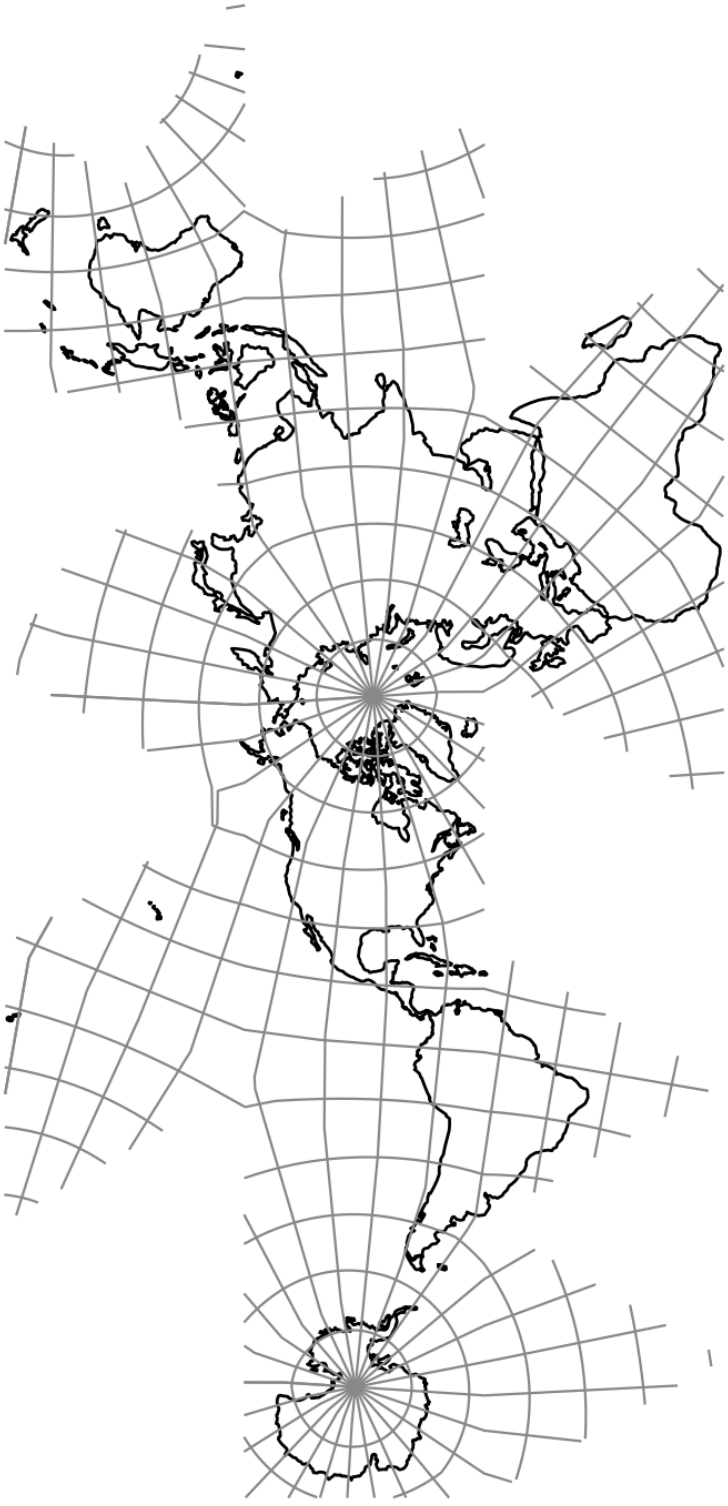


Fig. 6: proj-string: `+proj=airocean`

> **Note**
>
> The airocean map of the whole globe has an extent of approximately $[0, 0, 17404000, 36844000]$ in meters. Although the airocean map has no true up or down, the choice was made to lay it out vertically, from Antarctica to Australia.

#### 7.1.6.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+orient**=<string>

Can be set to either `vertical` or `horizontal`.

*Defaults to vertical*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.7 Airy

The Airy projection is an azimuthal minimum error projection for the region within the small or great circle defined by an angular distance, $\phi_b$, from the tangency point of the plane $(\lambda_0, \phi_0)$.

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | airy |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 7: proj-string: `+proj=airy`

#### 7.1.7.1 Options

**`+lat_b`**

Angular distance from tangency point of the plane $(\lambda_0, \phi_0)$ where the error is kept at minimum.

*Defaults to 90° (suitable for hemispherical maps).*

**`+no_cut`**

Do not cut at hemisphere limit

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

### 7.1.8 Aitoff

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse spherical projection |
| **Defined area** | Global |
| **Alias** | aitoff |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

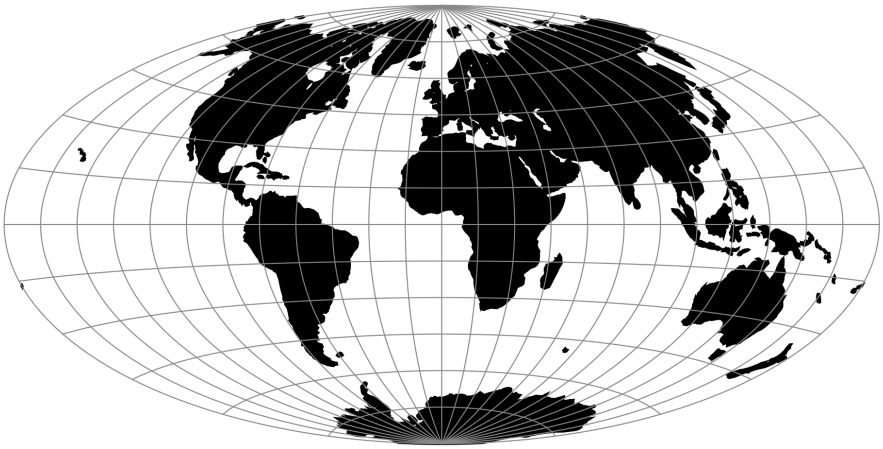


Fig. 8: proj-string: `+proj=aitoff`

#### 7.1.8.1 Parameters

> **Note**
>
> All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.9 Modified Stereographic of Alaska

| | |
|---|---|
| **Classification** | Modified azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Alaska |
| **Alsk** | alsk |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

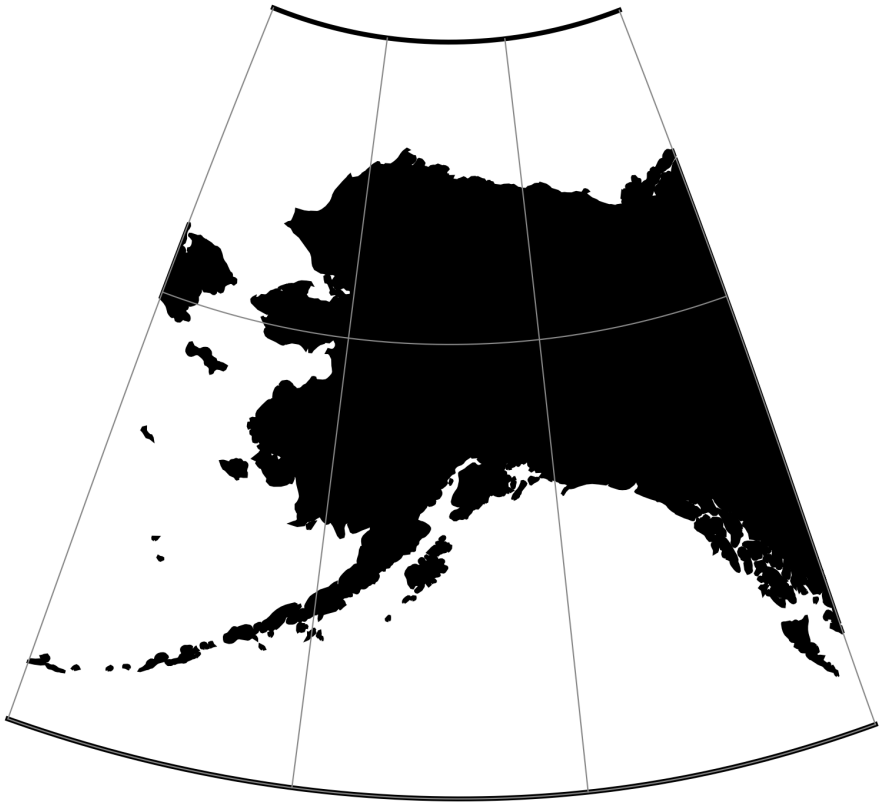


Fig. 9: proj-string: `+proj=alsk`

#### 7.1.9.1 Options

**Note**

All options are optional for the projection.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

### 7.1.10 Apian Globular I

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | apian |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

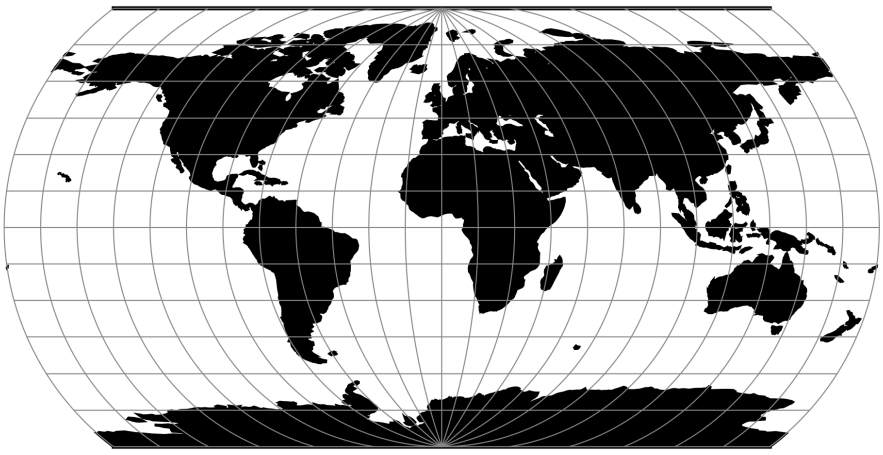


Fig. 10: proj-string: `+proj=apian`

#### 7.1.10.1 Options

> **Note**
>
> All options are optional for the Apian Globular projection.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.11 August Epicycloidal

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | august |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

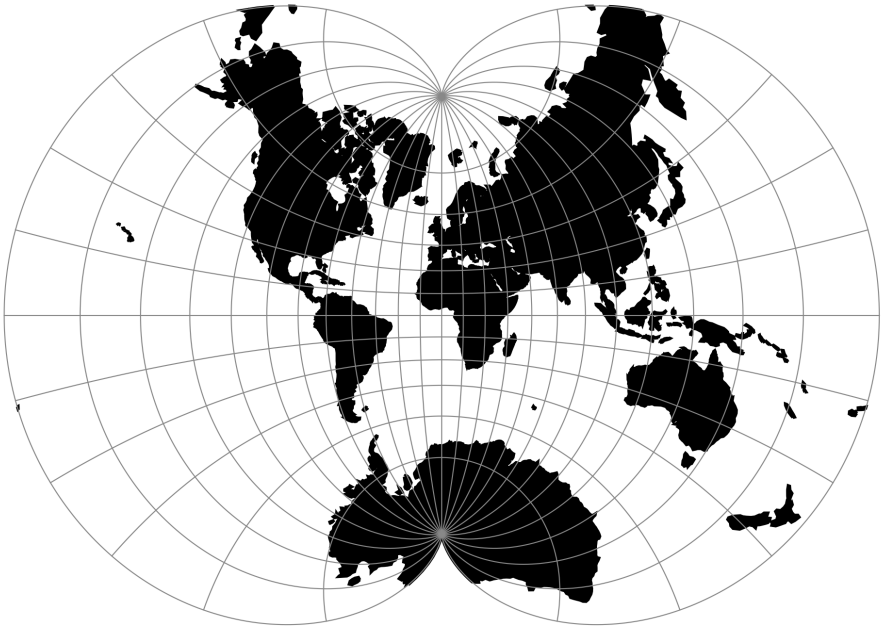


Fig. 11: proj-string: `+proj=august`

#### 7.1.11.1 Parameters

> **Note**
>
> All options are optional for the August Epicycloidal projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.12 Bacon Globular

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | bacon |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

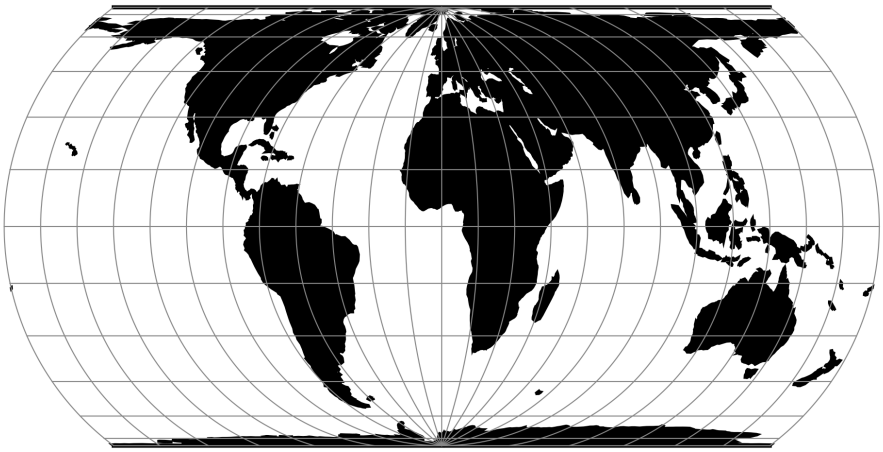


Fig. 12: proj-string: `+proj=bacon`

#### 7.1.12.1 Parameters

> **Note**
>
> All parameters are optional for the Bacon Globular projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.13 Bertin 1953

Added in version 6.0.0.

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward, spherical projection |
| **Defined area** | Global |
| **Alias** | bertin1953 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

The Bertin 1953 projection is intended for making world maps. Created by Jacques Bertin in 1953, this projection was the go-to choice of the French cartographic school when they wished to represent phenomena on a global scale. The projection was formulated in 2017 by Philippe Rivière for visionscarto.net.

#### 7.1.13.1 Usage

The Bertin 1953 projection has no special options. Its rotation parameters are fixed. Here is an example of a forward projection with scale 1:

```
$ echo 122 47 | proj +proj=bertin1953 +R=1
0.72    0.73
```

#### 7.1.13.2 Parameters

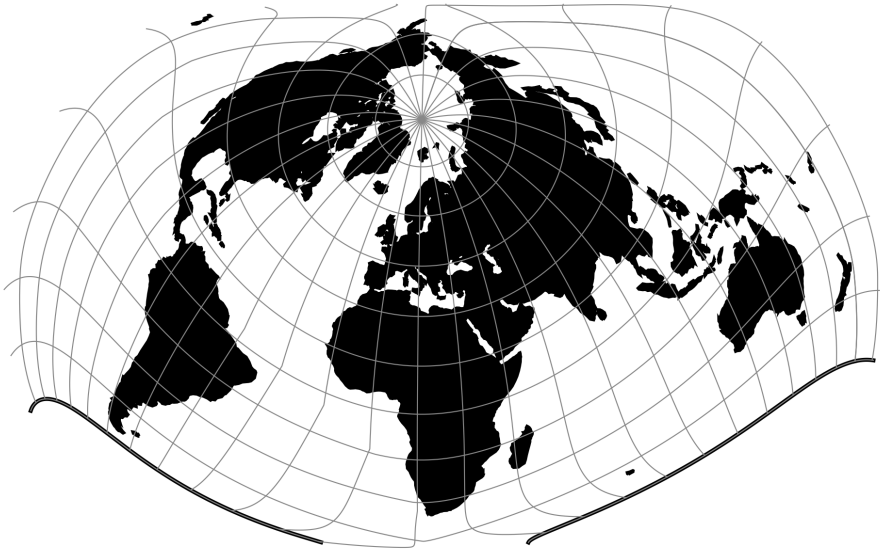


Fig. 13: proj-string: `+proj=bertin1953`

> **Note**
>
> All parameters for the projection are optional.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.13.3 Further reading

1. Philippe Rivière (2017). *Bertin Projection (1953) <https://visionscarto.net/bertin-projection-1953>*, Visionscarto.net.

### 7.1.14 Bipolar conic of western hemisphere

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse spherical projection |
| **Defined area** | Global |
| **Alias** | bipc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 14: proj-string: `+proj=bipc +ns`

#### 7.1.14.1 Parameters

**Note**

All options are optional for the Bipolar Conic projection.

**+ns**

Return non-skewed cartesian coordinates.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.15 Boggs Eumorphic

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | boggs |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.15.1 Parameters

> **Note**
>
> All options are optional for the Boggs Eumorphic projection.

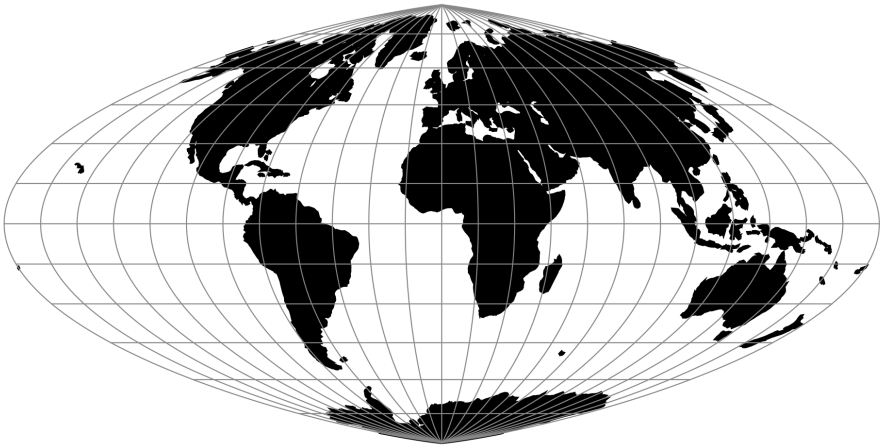


Fig. 15: proj-string: `+proj=boggs`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.16 Bonne (Werner lat_1=90)

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | bonne |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

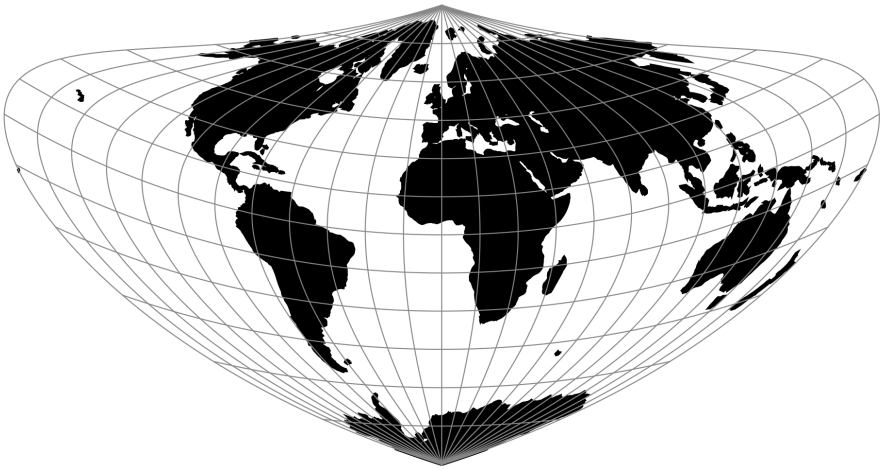


Fig. 16: proj-string: `+proj=bonne +lat_1=10`

#### 7.1.16.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

#### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

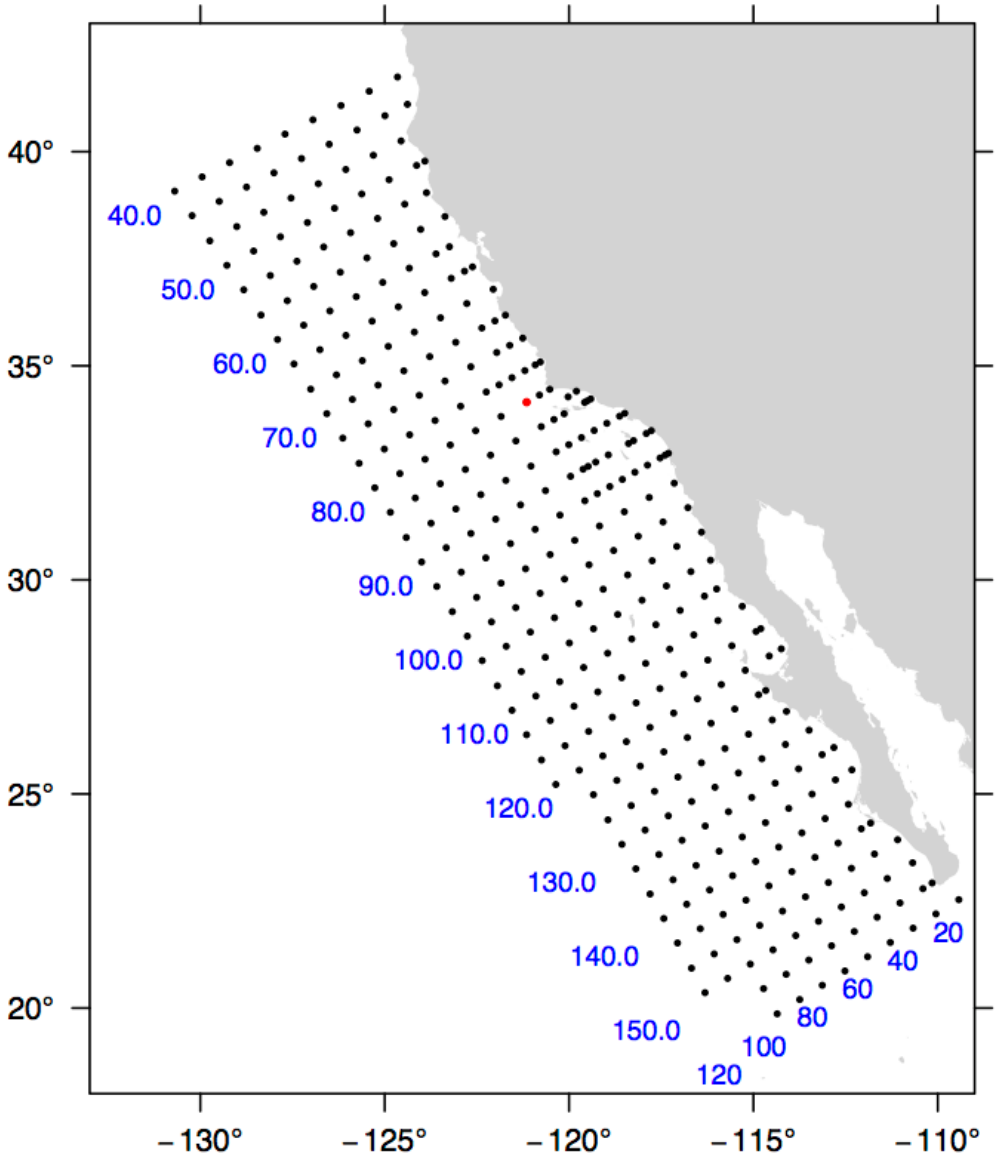
False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.17 Cal Coop Ocean Fish Invest Lines/Stations

The CalCOFI pseudo-projection is the line and station coordinate system of the California Cooperative Oceanic Fisheries Investigations program, known as CalCOFI, for sampling offshore of the west coast of the U.S. and Mexico.

| | |
|---|---|
| **Classification** | Conformal cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Only valid for the west coast of USA and Mexico |
| **Alias** | calcofi |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |




The coordinate system is based on the Mercator projection with units rotated -30 degrees from the meridian so that they are oriented with the coastline of the Southern California Bight and Baja California. Lines increase from Northwest to Southeast. A unit of line is 12 nautical miles. Stations increase from inshore to offshore. A unit of station is equal to 4 nautical miles. The rotation point is located at line 80, station 60, or 34.15 degrees N, -121.15 degrees W, and is depicted by the red dot in the figure. By convention, the ellipsoid of Clarke 1866 is used to calculate CalCOFI coordinates.

The CalCOFI program is a joint research effort by the U.S. National Oceanic and Atmospheric Administration, University of California Scripps Oceanographic Institute, and California Department of Fish and Game. Surveys have been conducted for the CalCOFI program since 1951, creating one of the oldest and most scientifically valuable joint oceanographic and fisheries data sets in the world. The CalCOFI line and station coordinate system is now used by several other programs including the Investigaciones Mexicanas de la Corriente de California (IMECOCAL) program offshore of Baja California. The figure depicts some commonly sampled locations from line 40 to line 156.7 and offshore to station 120. Blue numbers indicate line (bottom) or station (left) numbers along the grid. Note that lines spaced at approximate 3-1/3 intervals are commonly sampled, e.g., lines 43.3 and 46.7.

#### 7.1.17.1 Usage

A typical forward CalCOFI projection would be from long/lat coordinates on the Clark 1866 ellipsoid. For example:

```
proj +proj=calcofi +ellps=clrk66 -E <<EOF
-121.15 34.15
EOF
```

Output of the above command:

```
-121.15 34.15   80.00   60.00
```

The reverse projection from line/station coordinates to long/lat would be entered as:

```
proj +proj=calcofi +ellps=clrk66 -I -E -f "%.2f" <<EOF
80.0 60.0
EOF
```

Output of the above command:

```
80.0 60.0   -121.15 34.15
```

#### 7.1.17.2 Options

**Note**

All options are optional for the CalCOFI projection.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

#### 7.1.17.3 Mathematical definition

The algorithm used to make conversions is described in [EberHewitt1979] with a few corrections reported in [Weber-Moore2013].

#### 7.1.17.4 Further reading

1. General information about the CalCOFI program
2. The Investigaciones Mexicanas de la Corriente de California

### 7.1.18 Cassini (Cassini-Soldner)

Although the Cassini projection has been largely replaced by the Transverse Mercator, it is still in limited use outside the United States and was one of the major topographic mapping projections until the early 20th century.

| | |
|---|---|
| **Classification** | Transverse and oblique cylindrical |
| **Available forms** | Forward and inverse, Spherical and ellipsoidal |
| **Defined area** | Global, but best used near the central meridian with long, narrow areas |
| **Alias** | cass |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.18.1 Usage

There has been little usage of the spherical version of the Cassini, but the ellipsoidal Cassini-Soldner version was adopted by the Ordnance Survey for the official survey of Great Britain during the second half of the 19th century [Steers1970]. Many of these maps were prepared at a scale of 1:2,500. The Cassini-Soldner was also used for the detailed mapping of many German states during the same period.

Example using EPSG 30200 (Trinidad 1903, units in clarke's links):

```
$ echo 0.17453293 -1.08210414 | proj +proj=cass \
  +lat_0=10.44166666666667 +lon_0=-61.33333333333334 \
  +x_0=86501.46392051999 +y_0=65379.0134283 \
  +a=6378293.645208759 +b=6356617.987679838 \
  +to_meter=0.201166195164
66644.94    82536.22
```

Example using EPSG 3068 (Soldner Berlin):

```
$ echo 13.5 52.4 | proj +proj=cass \
  +lat_0=52.41864827777778 +lon_0=13.62720366666667 \
  +x_0=40000 +y_0=10000 +ellps=bessel
31343.05    7932.76
```

#### 7.1.18.2 Options

**Note**

All options are optional for the Cassini projection.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

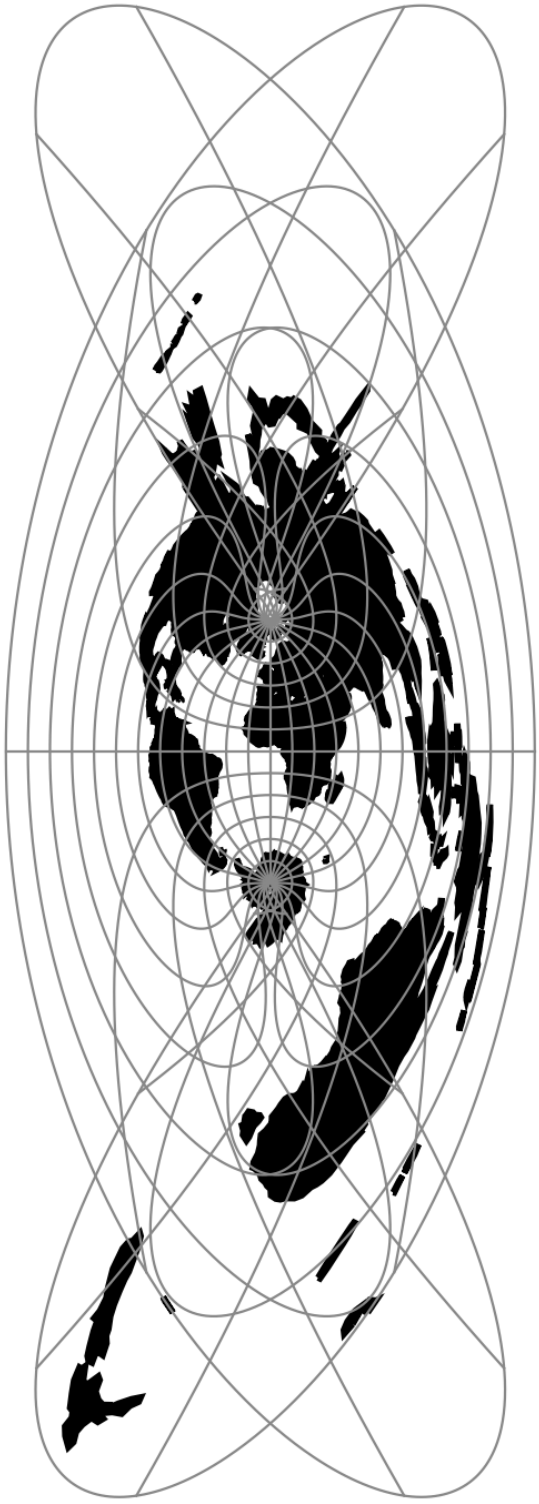


Fig. 17: proj-string: `+proj=cass`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+hyperbolic**

Use modified form of the standard Cassini-Soldner projection known as the Hyperbolic Cassini-Soldner. This is used in particular for the "Vanua Levu Grid" of the island of Vanua Levu, Fiji (EPSG:3139)

#### 7.1.18.3 Mathematical definition

The formulas describing the Cassini projection are taken from [Snyder1987].

$\phi_0$ is the latitude of origin that match the center of the map (default to 0). It can be set with +lat_0.

##### Spherical form

##### Forward projection

$$x = \arcsin(\cos(\phi)\sin(\lambda))$$

$$y = \arctan 2(\tan(\phi), \cos(\lambda)) - \phi_0$$

#### Inverse projection

$$\phi = \arcsin(\sin(y + \phi_0)\cos(x))$$

$$\lambda = \arctan 2(\tan(x), \cos(y + \phi_0))$$

#### Ellipsoidal form

#### Forward projection

$$N = (1 - e^2 \sin^2(\phi))^{-1/2}$$

$$T = \tan^2(\phi)$$

$$A = \lambda \cos(\phi)$$

$$C = \frac{e^2}{1 - e^2} cos^2(\phi)$$

$$x = N(A - T\frac{A^3}{6} - (8 - T + 8C)T\frac{A^5}{120})$$

$$y = M(\phi) - M(\phi_0) + N\tan(\phi)(\frac{A^2}{2} + (5 - T + 6C)\frac{A^4}{24})$$

and M() is the meridional distance function.

#### Inverse projection

$$\phi' = M^{-1}(M(\phi_0) + y)$$

if $\phi' = \frac{\pi}{2}$ then $\phi = \phi'$ and $\lambda = 0$

otherwise evaluate T and N above using $\phi'$ and

$$R = (1 - e^2)(1 - e^2 sin^2\phi')^{-3/2}$$

$$D = x/N$$

$$\phi = \phi' - \tan\phi'\frac{N}{R}(\frac{D^2}{2} - (1 + 3T)\frac{D^4}{24})$$

$$\lambda = \frac{(D - T\frac{D^3}{3} + (1 + 3T)T\frac{D^5}{15})}{\cos\phi'}$$

#### 7.1.18.4 Further reading

1. Wikipedia
2. EPSG, POSC literature pertaining to Coordinate Conversions and Transformations including Formulas

### 7.1.19 Central Cylindrical

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | cc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

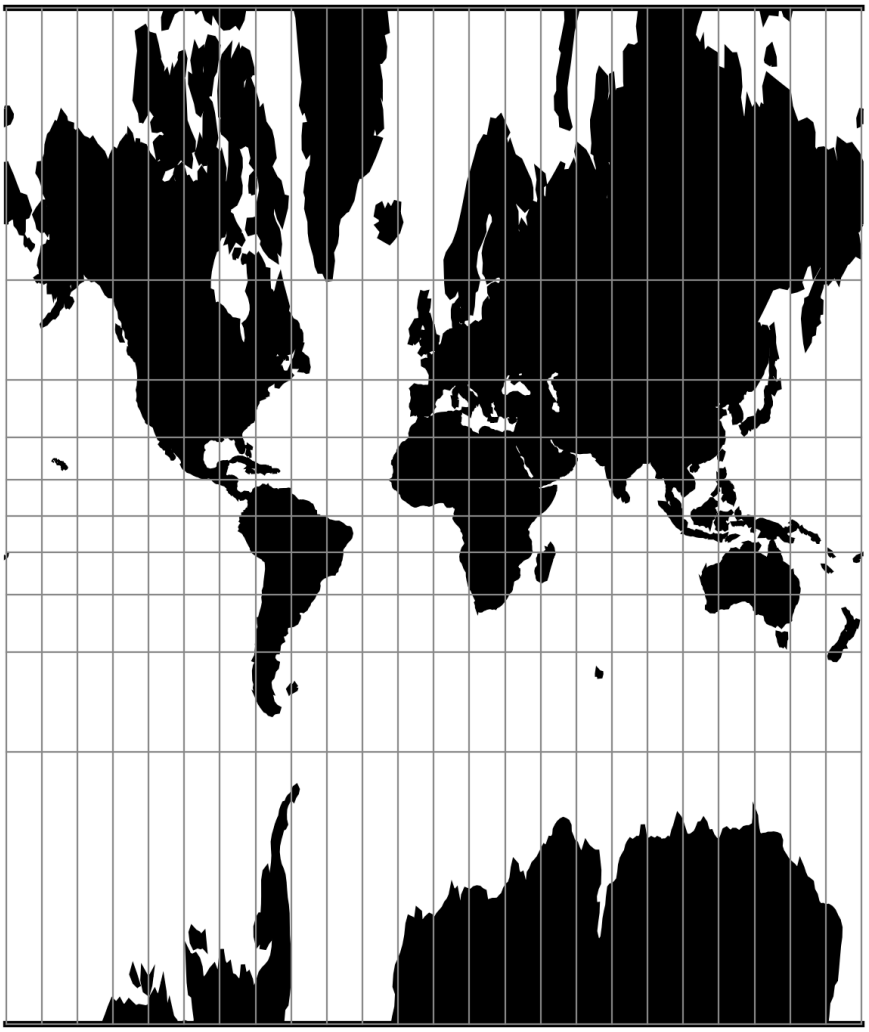


Fig. 18: proj-string: `+proj=cc`

#### 7.1.19.1 Parameters

> **Note**
>
> All parameters are optional for the Central Cylindrical projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.20 Central Conic

Added in version 5.0.0.

This is central (centrographic) projection on cone tangent at :option:`lat_1` latitude, identical with `conic()` projection from `mapproj` R package.

| | |
|---|---|
| **Classification** | Conic |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global, but best used near the standard parallel |
| **Alias** | ccon |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 19: proj-string: `+proj=ccon +lat_1=52 +lon_0=19`

#### 7.1.20.1 Usage

This simple projection is rarely used, as it is not equidistant, equal-area, nor conformal.

An example of usage (and the main reason to implement this projection in proj4) is the ATPOL geobotanical grid of Poland, developed in Institute of Botany, Jagiellonian University, Krakow, Poland in 1970s [Zajac1978]. The grid was originally handwritten on paper maps and further copied by hand. The projection (together with strange Earth radius) was chosen by its creators as the compromise fit to existing maps during first software development in DOS era. Many years later it is still de facto standard grid in Polish geobotanical research.

The ATPOL coordinates can be achieved with with the following parameters:

```
+proj=ccon +lat_1=52 +lon_0=19 +axis=esu +a=6390000 +x_0=330000 +y_0=-350000
```

For more information see [Komsta2016] and [Verey2017].

#### 7.1.20.2 Parameters

##### Required

**+lat_1**=<value>

Standard parallel of projection.

##### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.20.3 Mathematical definition

##### Forward projection

$$r = \cot \phi_0 - \tan(\phi - \phi_0)$$

$$x = r \sin(\lambda \sin \phi_0)$$

$$y = \cot \phi_0 - r \cos(\lambda \sin \phi_0)$$

##### Inverse projection

$$y = \cot \phi_0 - y$$

$$\phi = \phi_0 - \tan^{-1}(\sqrt{x^2 + y^2} - \cot \phi_0)$$

$$\lambda = \frac{\tan^{-1} \sqrt{x^2 + y^2}}{\sin \phi_0}$$

#### 7.1.20.4 Reference values

For ATPOL to WGS84 test, run the following script:

```
#!/bin/bash
cat << EOF | src/cs2cs -v -f "%E" +proj=ccon +lat_1=52 +lat_0=52 +lon_0=19 +axis=esu␣
↪+a=6390000 +x_0=330000 +y_0=-350000 +to +proj=longlat
0 0
0 700000
```

(continues on next page)

(continued from previous page)

```
700000 0
700000 700000
330000 350000
EOF
```

It should result with

```
1.384023E+01 5.503040E+01 0.000000E+00
1.451445E+01 4.877385E+01 0.000000E+00
2.478271E+01 5.500352E+01 0.000000E+00
2.402761E+01 4.875048E+01 0.000000E+00
1.900000E+01 5.200000E+01 0.000000E+00
```

Analogous script can be run for reverse test:

```
cat << EOF  | src/cs2cs -v -f "%E" +proj=longlat +to +proj=ccon +lat_1=52 +lat_0=52 +lon_
↪0=19 +axis=esu +a=6390000 +x_0=330000 +y_0=-350000
24 55
15 49
24 49
19 52
EOF
```

and it should give the following results:

```
6.500315E+05 4.106162E+03 0.000000E+00
3.707419E+04 6.768262E+05 0.000000E+00
6.960534E+05 6.722946E+05 0.000000E+00
3.300000E+05 3.500000E+05 0.000000E+00
```

### 7.1.21 Equal Area Cylindrical

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | cea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.21.1 Named specializations

The Equal Area Cylindrical projection is sometimes known under other names when it is instantiated with particular values of the `lat_ts` parameter:

| Name | lat_ts |
|---|---|
| Lambert cylindrical equal-area | 0 |
| Behrmann | 30 |
| Gall-Peters | 45 |

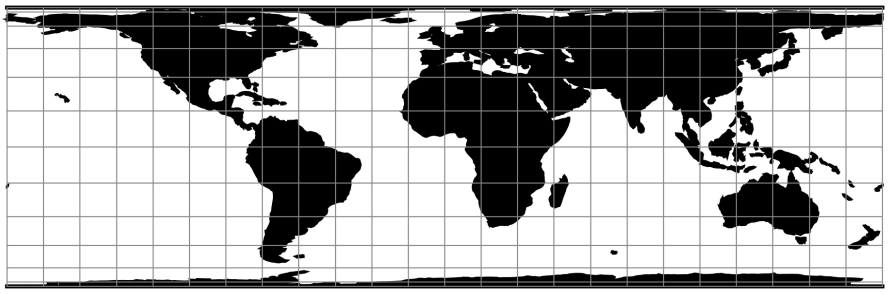


Fig. 20: proj-string: `+proj=cea`

#### 7.1.21.2 Parameters

> **Note**
>
> All parameters are optional for the Equal Area Cylindrical projection.

**+lat_ts**=<value>

Latitude of true scale. Defines the latitude where scale is not distorted. Takes precedence over **+k_0** if both options are used together.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_ts=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**Note**

`lat_ts` and `k_0` are mutually exclusive. If `lat_ts` is specified, it is equivalent to setting `k_0` to $\frac{\cos \phi_{ts}}{\sqrt{1-e^2 \sin^2 \phi_{ts}}}$

#### 7.1.21.3 Further reading

1. Wikipedia: Lambert cylindrical equal-area
2. Wikipedia: Gall-Peters
3. Wikipedia: Behrmann

### 7.1.22 Chamberlin Trimetric

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | chamb |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.22.1 Parameters

**Required**

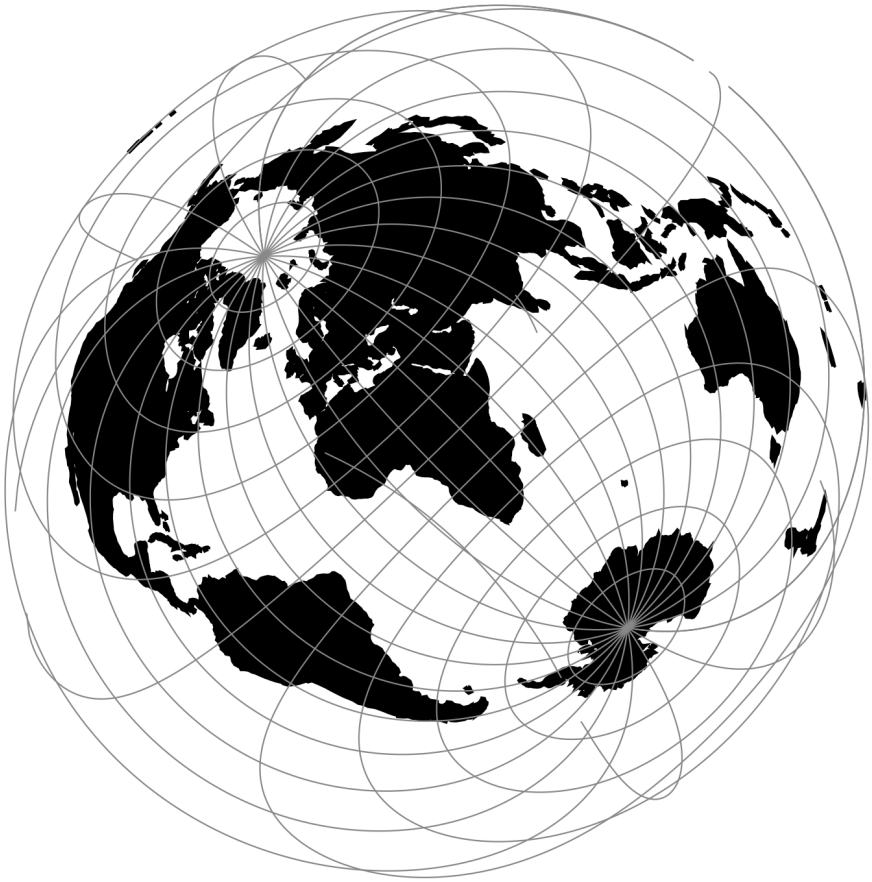


Fig. 21: proj-string: `+proj=chamb +lat_1=10 +lon_1=30 +lon_2=40`

> **Note**
>
> Control points should be oriented clockwise.

**+lat_1**=<value>
: Latitude of the first control point.

**+lon_1**=<value>
: Longitude of the first control point.

**+lat_2**=<value>
: Latitude of the second control point.

**+lon_2**=<value>
: Longitude of the second control point.

**+lat_3**=<value>
: Latitude of the third control point.

**+lon_3**=<value>
: Longitude of the third control point.

### Optional

**+lon_0**=<value>
: Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

  *Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>
: Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

  See *Ellipsoid size parameters* for more information.

**+x_0**=<value>
: False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

**+y_0**=<value>
: False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

### 7.1.23 Collignon

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | collg |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

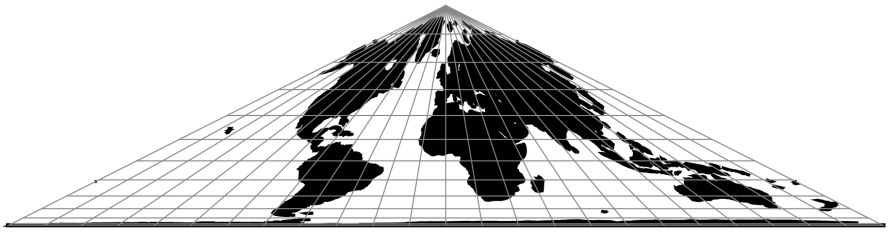


Fig. 22: proj-string: `+proj=collg`

#### 7.1.23.1 Parameters

> **Note**
>
> All parameters are optional for the Collignon projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.24 Colombia Urban

Added in version 7.2.

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse ellipsoidal projection |
| **Alias** | col_urban |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

From [IOGP2018]:

> The capital cites of each department in Colombia use an urban projection for large scale topographic mapping of the urban areas. It is based on a plane through the origin at an average height for the area, eliminating the need for corrections to engineering survey observations.

proj-string: +proj=col_urban

#### 7.1.24.1 Parameters

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+h_0**=<value>

Projection plane origin height (in metre)

*Defaults to 0.0.*

### 7.1.25 Compact Miller

The Compact Miller projection is a cylindrical map projection with a height-to-width ratio of 0.6:1.

See [Jenny2015]

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | comill |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.25.1 Parameters

**Note**

All parameters are optional for projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

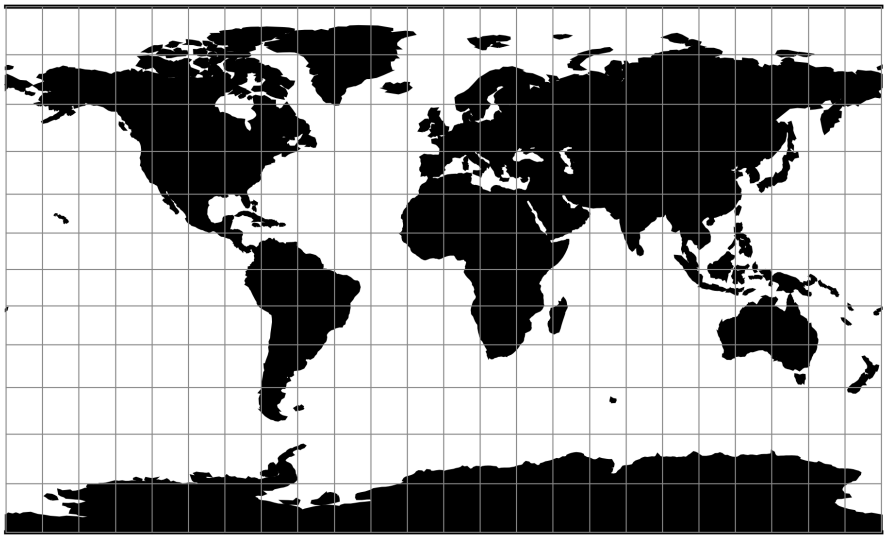


Fig. 23: proj-string: `+proj=comill`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.26 Craster Parabolic (Putnins P4)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | crast |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

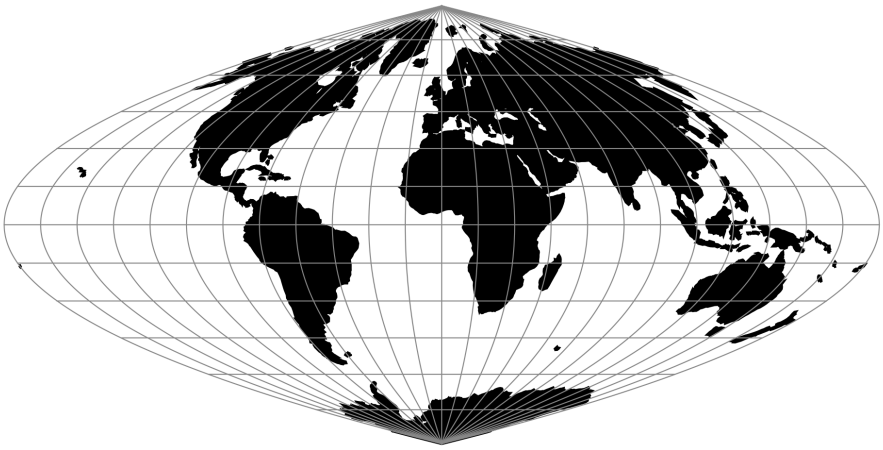


Fig. 24: proj-string: `+proj=crast`

#### 7.1.26.1 Parameters

> **Note**
>
> All parameters are optional for the Craster Parabolic projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.27 Denoyer Semi-Elliptical

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | denoy |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

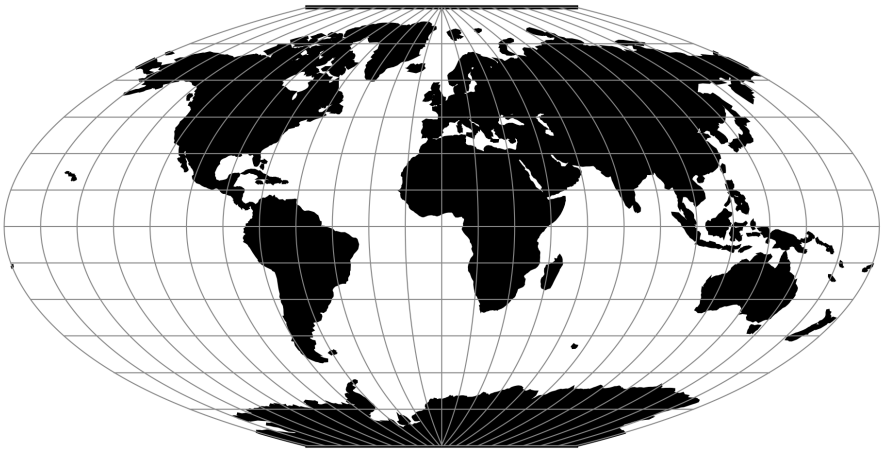


Fig. 25: proj-string: +proj=denoy

#### 7.1.27.1 Parameters

**Note**

All parameters are optional for the Denoyer Semi-Elliptical projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.28 Eckert I

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | eck1 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

$$x = 2\sqrt{2/3\pi}\lambda(1 - |\phi|/\pi)$$
$$y = 2\sqrt{2/3\pi}\phi$$

#### 7.1.28.1 Parameters

> **Note**
>
> All parameters are optional for the Eckert I projection.

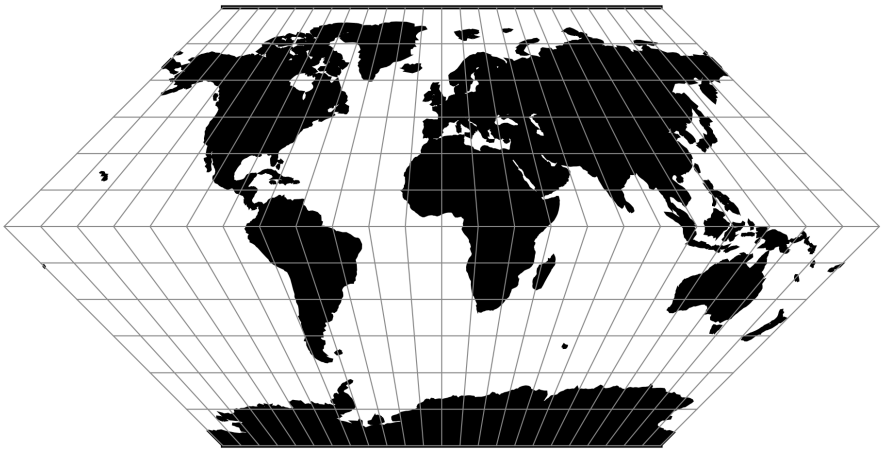


Fig. 26: proj-string: `+proj=eck1`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.29 Eckert II

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | eck2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

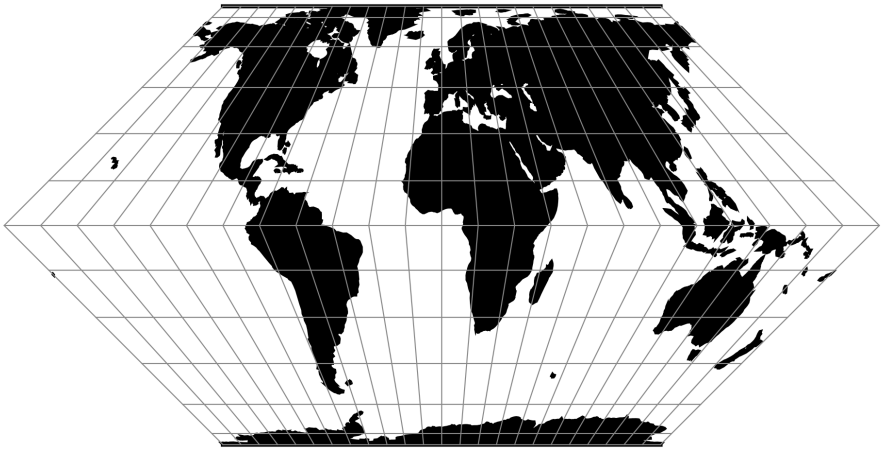


Fig. 27: proj-string: `+proj=eck2`

#### 7.1.29.1 Parameters

> **Note**
>
> All parameters are optional for the Eckert II projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.30 Eckert III

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | eck3 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

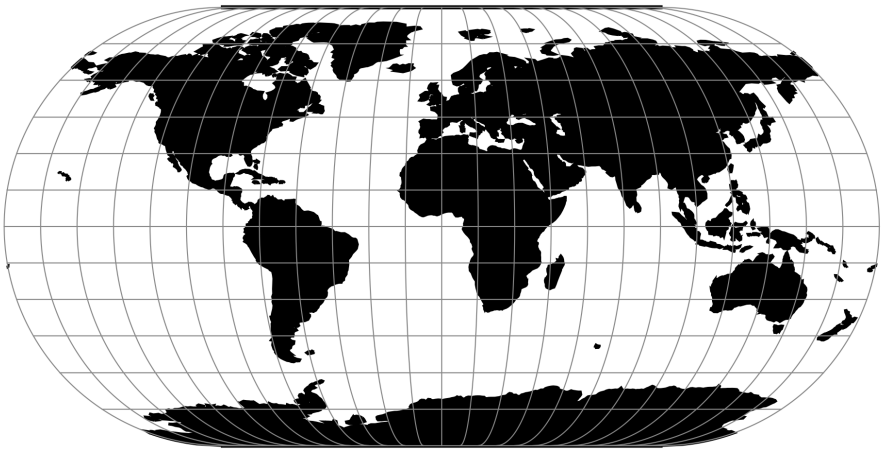


Fig. 28: proj-string: +proj=eck3

#### 7.1.30.1 Parameters

**Note**

All parameters are optional for the Eckert III projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.31 Eckert IV

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | eck4 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

$$x = \lambda(1 + cos\phi)/\sqrt{2 + \pi}$$

$$y = 2\phi/\sqrt{2 + \pi}$$

#### 7.1.31.1 Parameters

> **Note**
>
> All parameters are optional for the Eckert IV projection.

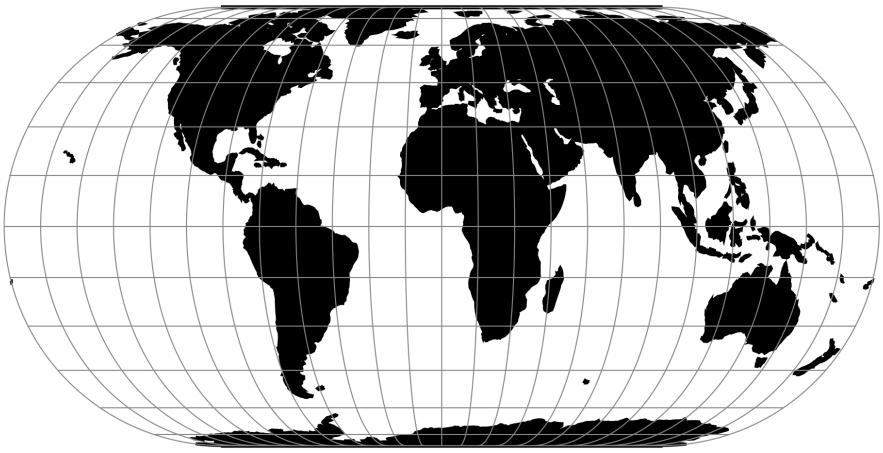


Fig. 29: proj-string: `+proj=eck4`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.32 Eckert V

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | eck5 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

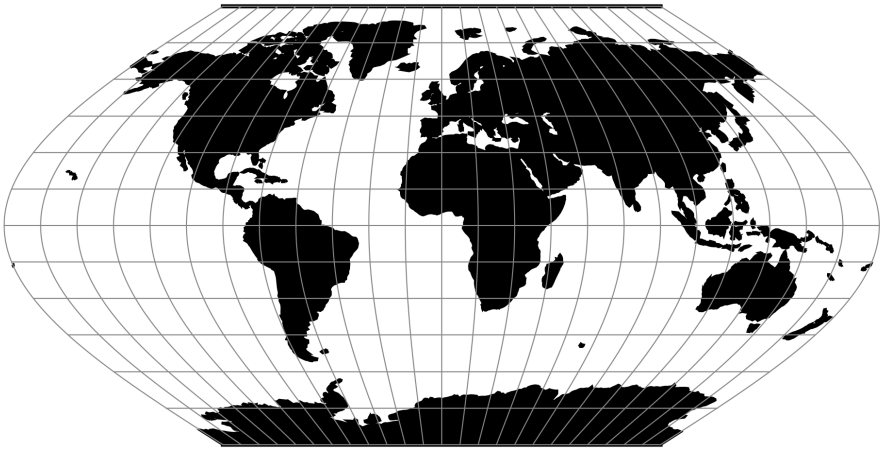


Fig. 30: proj-string: `+proj=eck5`

#### 7.1.32.1 Parameters

> **Note**
>
> All parameters are optional for the Eckert V projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.33 Eckert VI

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | eck6 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

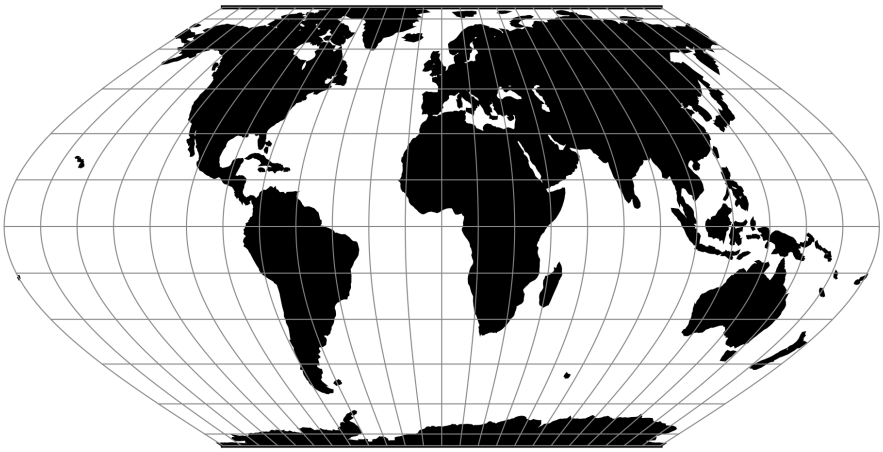


Fig. 31: proj-string: +proj=eck6

#### 7.1.33.1 Parameters

**Note**

All parameters are optional for the Eckert VI projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.34 Equidistant Cylindrical (Plate Carrée)

The simplest of all projections. Standard parallels (0° when omitted) may be specified that determine latitude of true scale (k=h=1).

| | |
|---|---|
| **Classification** | Conformal cylindrical |
| **Available forms** | Forward and inverse |
| **Defined area** | Global, but best used near the equator |
| **Alias** | eqc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.34.1 Usage

Because of the distortions introduced by this projection, it has little use in navigation or cadastral mapping and finds its main use in thematic mapping. In particular, the plate carrée has become a standard for global raster datasets, such as Celestia and NASA World Wind, because of the particularly simple relationship between the position of an image pixel on the map and its corresponding geographic location on Earth.

The following table gives special cases of the cylindrical equidistant projection.

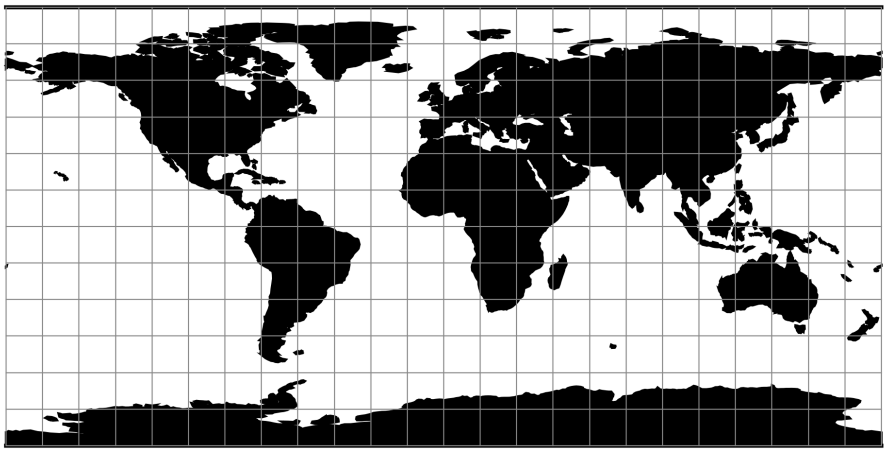


Fig. 32: proj-string: `+proj=eqc`

| Projection Name | (lat ts=) $\phi_0$ |
|---|---|
| Plain/Plane Chart | 0° |
| Simple Cylindrical | 0° |
| Plate Carrée | 0° |
| Ronald Miller—minimum overall scale distortion | 37°30′ |
| E.Grafarend and A.Niermann | 42° |
| Ronald Miller—minimum continental scale distortion | 43°30′ |
| Gall Isographic | 45° |
| Ronald Miller Equirectangular | 50°30′ |
| E.Grafarend and A.Niermann minimum linear distortion | 61°7′ |

Example using EPSG 32662 (WGS84 Plate Carrée):

```
$ echo 2 47 | proj +proj=eqc +ellps=WGS84
222638.98       5232016.07
```

Example using Plate Carrée projection with true scale at latitude 30° and central meridian 90°W:

```
$ echo -88 30 | proj +proj=eqc +lat_ts=30 +lon_0=90w
192811.01       3339584.72
```

#### 7.1.34.2 Parameters

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

**+lat_ts**=<value>

Latitude of true scale. Defines the latitude where scale is not distorted. Takes precedence over **+k_0** if both options are used together.

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_ts=1.570796r*

See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

#### 7.1.34.3 Mathematical definition

The formulas describing the Equidistant Cylindrical projection are all taken from [Snyder1987].

$\phi_{ts}$ is the latitude of true scale, i.e., the standard parallel where the scale of the projection is true. It can be set with +lat_ts.

$\phi_0$ is the latitude of origin that match the center of the map. It can be set with +lat_0.

##### Forward projection

$$x = \lambda \cos \phi_{ts}$$

$$y = \phi - \phi_0$$

##### Inverse projection

$$\lambda = x/cos\phi_{ts}$$

$$\phi = y + \phi_0$$

#### 7.1.34.4 Further reading

1. Wikipedia
2. Wolfram Mathworld

### 7.1.35 Equidistant Conic

| | |
|---|---|
| **Classification** | Conic |
| **Available forms** | Forward and inverse, ellipsoidal |
| **Defined area** | Global |
| **Alias** | eqdc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.35.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

Fig. 33: proj-string: `+proj=eqdc +lat_1=55 +lat_2=60`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_2=1.570796r*
>
> See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.36 Equal Earth

Added in version 5.2.0.

| | |
|---|---|
| **Classification** | Pseudo cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal projection |
| **Defined area** | Global |
| **Alias** | eqearth |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

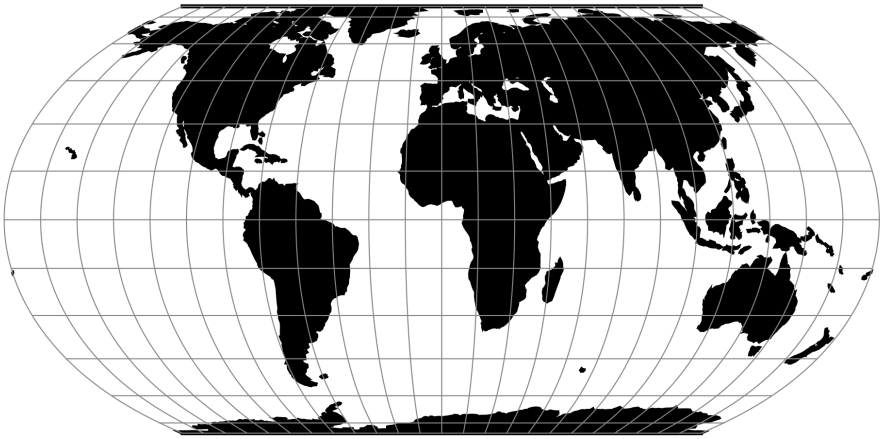


Fig. 34: proj-string: `+proj=eqearth`

The Equal Earth projection is intended for making world maps. Equal Earth is a projection inspired by the Robinson projection, but unlike the Robinson projection retains the relative size of areas. The projection was designed in 2018 by Bojan Savric, Tom Patterson and Bernhard Jenny [Savric2018].

#### 7.1.36.1 Usage

The Equal Earth projection has no special options. Here is an example of an forward projection on a sphere with a radius of 1 m:

```
$ echo 122 47 | proj +proj=eqearth +R=1
1.55    0.89
```

#### 7.1.36.2 Parameters

> **Note**
>
> All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.36.3 Further reading

1. Bojan Savric, Tom Patterson & Bernhard Jenny (2018). The Equal Earth map projection, International Journal of Geographical Information Science

### 7.1.37 Euler

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | euler |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 35: proj-string: `+proj=euler +lat_1=67 +lat_2=75`

#### 7.1.37.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_2=1.570796r*
>
> See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.38 Fahey

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | fahey |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

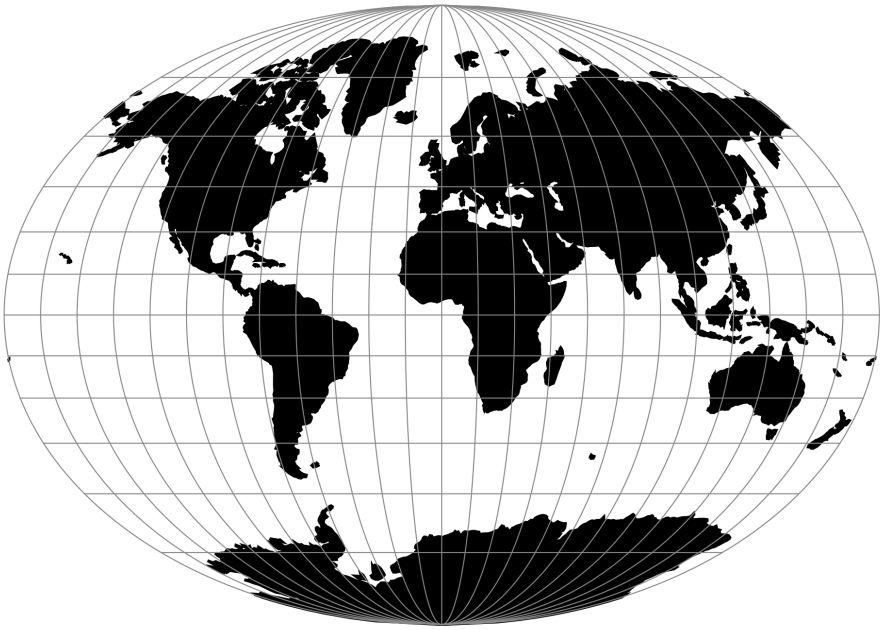


Fig. 36: proj-string: `+proj=fahey`

#### 7.1.38.1 Parameters

> **Note**
>
> All parameters are optional for the Fahey projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.39 Foucaut

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | fouc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.39.1 Parameters

**Note**

All parameters are optional for the Foucaut projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

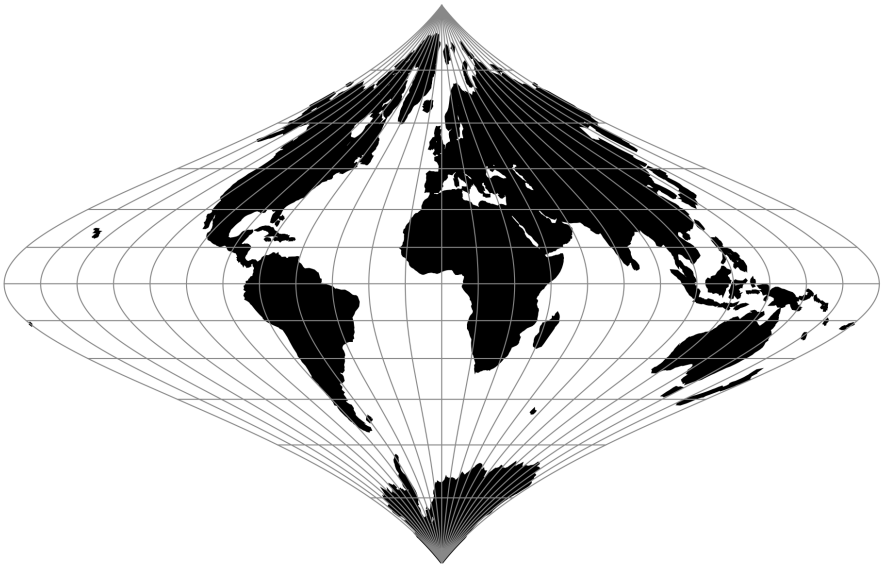


Fig. 37: proj-string: `+proj=fouc`

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.40 Foucaut Sinusoidal

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | fouc_s |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

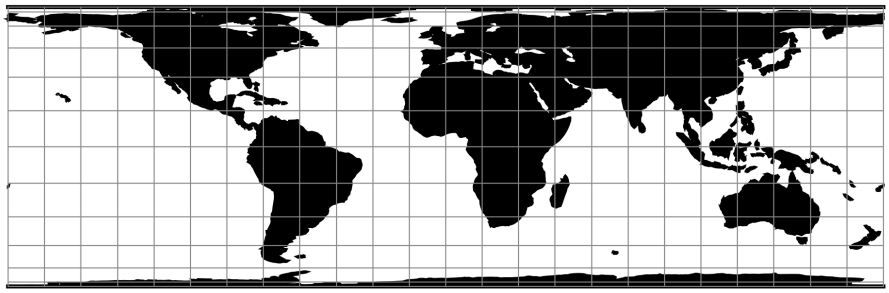


Fig. 38: proj-string: `+proj=fouc_s`

The $y$-axis is based upon a weighted mean of the cylindrical equal-area and the sinusoidal projections. Parameter $n = n$ is the weighting factor where $0 <= n <= 1$.

$$x = \lambda \cos \phi / (n + (1 - n)\ cos\phi)$$
$$y = n\phi + (1 - n) \sin \phi$$

For the inverse, the Newton-Raphson method can be used to determine $\phi$ from the equation for $y$ above. As $n \to 0$ and $\phi \to \pi/2$, convergence is slow but for $n = 0$, $\phi = \sin^1 y$

#### 7.1.40.1 Parameters

> **Note**
>
> All parameters are optional for the Foucaut Sinusoidal projection.

**+n**=<value>
: Weighting factor. Value should be in the interval 0-1.

**+lon_0**=<value>
: Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

  *Defaults to 0.0.*

  > **Note**
  >
  > The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
  >
  > Example: *+lon_0=1.570796r*
  >
  > See *Projection Units* for more information.

**+R**=<value>
: Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

  See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.41 Gall (Gall Stereographic)

The Gall stereographic projection, presented by James Gall in 1855, is a cylindrical projection. It is neither equal-area nor conformal but instead tries to balance the distortion inherent in any projection.

| | |
|---|---|
| **Classification** | Transverse and oblique cylindrical |
| **Available forms** | Forward and inverse, Spherical |
| **Defined area** | Global |
| **Alias** | gall |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

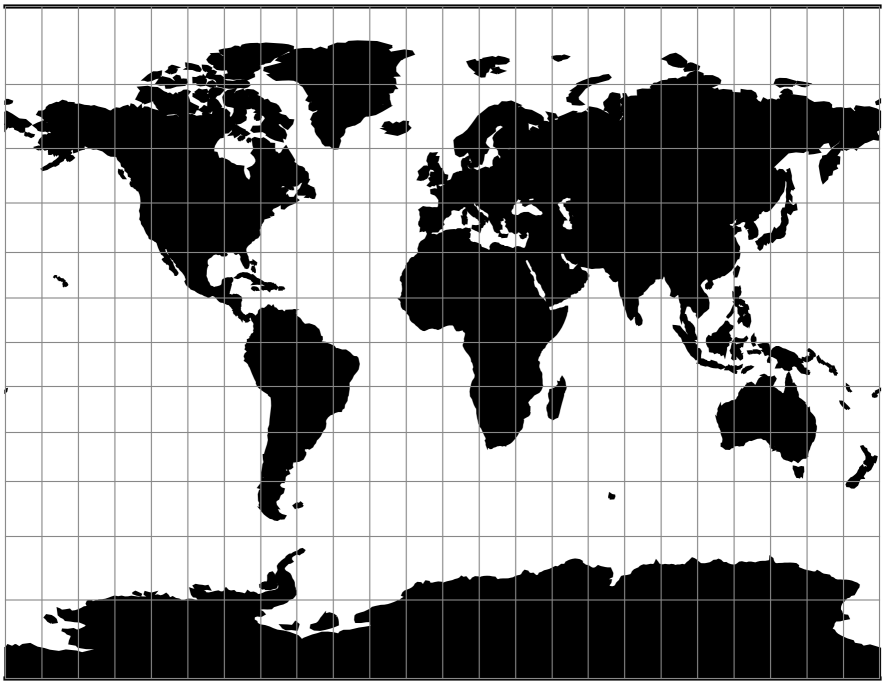


Fig. 39: proj-string: `+proj=gall`

#### 7.1.41.1 Usage

The need for a world map which avoids some of the scale exaggeration of the Mercator projection has led to some commonly used cylindrical modifications, as well as to other modifications which are not cylindrical. The earliest common cylindrical example was developed by James Gall of Edinburgh about 1855 (Gall, 1885, p. 119-123). His meridians are equally spaced, but the parallels are spaced at increasing intervals away from the Equator. The parallels of latitude are actually projected onto a cylinder wrapped about the sphere, but cutting it at lats. 45° N. and S., the point of perspective being a point on the Equator opposite the meridian being projected. It is used in several British atlases, but seldom in the United States. The Gall projection is neither conformal nor equal-area, but has a blend of various features. Unlike the Mercator, the Gall shows the poles as lines running across the top and bottom of the map.

**Note**

The Gall projection must not be confused with the Gall-Peters one, the later being a specialization of *Equal Area Cylindrical*.

Example using Gall Stereographic

```
$ echo 9 51 | proj +proj=gall
708432.90   5193386.36
```

Example using Gall Stereographic (Central meridian 90°W)

```
$ echo 9 51 | proj +proj=gall +lon_0=90w
7792761.91  5193386.36
```

#### 7.1.41.2 Parameters

**Note**

All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

#### 7.1.41.3 Mathematical definition

The formulas describing the Gall Stereographic are all taken from [Snyder1993].

##### Spherical form

##### Forward projection

$$x = \frac{\lambda}{\sqrt{2}}$$

$$y = (1 + \frac{\sqrt{2}}{2}) \tan(\phi/2)$$

##### Inverse projection

$$\phi = 2 \arctan(\frac{y}{1 + \frac{\sqrt{2}}{2}})$$

$$\lambda = \sqrt{2}x$$

#### 7.1.41.4 Further reading

1. Wikipedia
2. Cartographic Projection Procedures for the UNIX Environment-A User's Manual

### 7.1.42 Geostationary Satellite View

The geos projection pictures how a geostationary satellite scans the earth at regular scanning angle intervals.

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | geos |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 40: proj-string: `+proj=geos +h=35785831.0 +lon_0=-60 +sweep=y`

#### 7.1.42.1 Usage

In order to project using the geos projection you can do the following:

```
proj +proj=geos +h=35785831.0
```

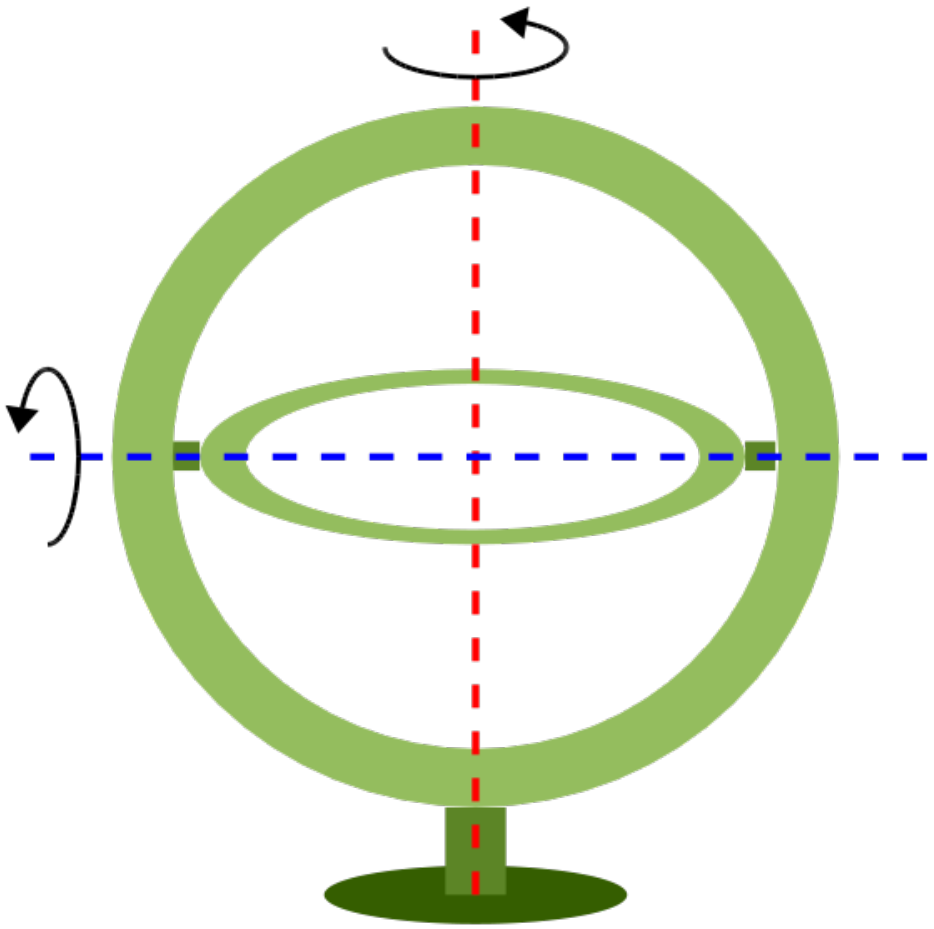
The required argument `h` is the viewing point (satellite position) height above the earth.

The projection coordinate relate to the scanning angle by the following simple relation:

```
scanning_angle (radians) = projection_coordinate / h
```

### Note on sweep angle

The viewing instrument on-board geostationary satellites described by this projection have a two-axis gimbal viewing geometry. This means that the different scanning positions are obtained by rotating the gimbal along a N/S axis (or `y`) and a E/W axis (or `x`).



In the image above, the outer-gimbal axis, or sweep-angle axis, is the N/S axis (`y`) while the inner-gimbal axis, or fixed-angle axis, is the E/W axis (`x`).

This example represents the scanning geometry of the Meteosat series satellite. However, the GOES satellite series use the opposite scanning geometry, with the E/W axis (`x`) as the sweep-angle axis, and the N/S (`y`) as the fixed-angle axis.

The sweep argument is used to tell PROJ which on which axis the outer-gimbal is rotating. The possible values are x or y, y being the default. Thus, the scanning geometry of the Meteosat series satellite should take sweep as y, and GOES should take sweep as x.

#### 7.1.42.2 Parameters

##### Required

**+h**=<value>

Height of the view point above the Earth and must be in the same units as the radius of the sphere or semimajor axis of the ellipsoid.

##### Optional

**+sweep**=<axis>

Sweep angle axis of the viewing instrument. Valid options are *"x"* and *"y"*.

*Defaults to "y".*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.43 Ginsburg VIII (TsNIIGAiK)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | gins8 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 41: proj-string: `+proj=gins8`

#### 7.1.43.1 Parameters

> **Note**
>
> All parameters are optional for the Ginsburg VIII projection.

`+lon_0=<value>`

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.44 General Sinusoidal Series

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | gn_sinu |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

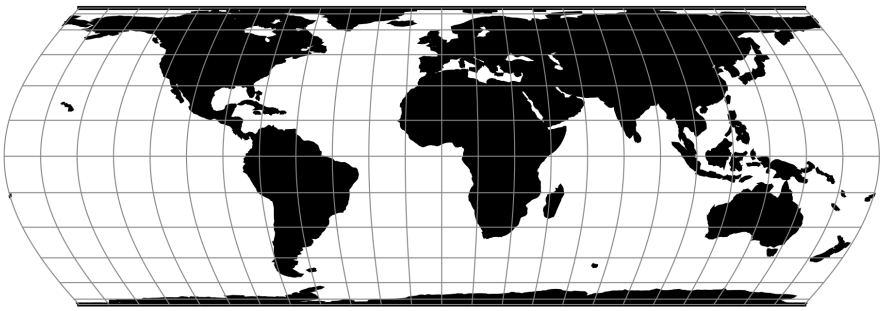


Fig. 42: proj-string: +proj=gn_sinu +m=2 +n=3

#### 7.1.44.1 Parameters

> **Note**
>
> All parameters are optional for the General Sinusoidal Series projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.45 Gnomonic

For a sphere, the gnomonic projection is a projection from the center of the sphere onto a plane tangent to the center point of the projection. This projects great circles to straight lines. For an ellipsoid, it is the limit of a doubly azimuthal projection, a projection where the azimuths from 2 points are preserved, as the two points merge into the center point. In this case, geodesics project to approximately straight lines (these are exactly straight if the geodesic includes the center point). For details, see Section 8 of [Karney2013].

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Within a quarter circumference of the center point |
| **Alias** | gnom |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

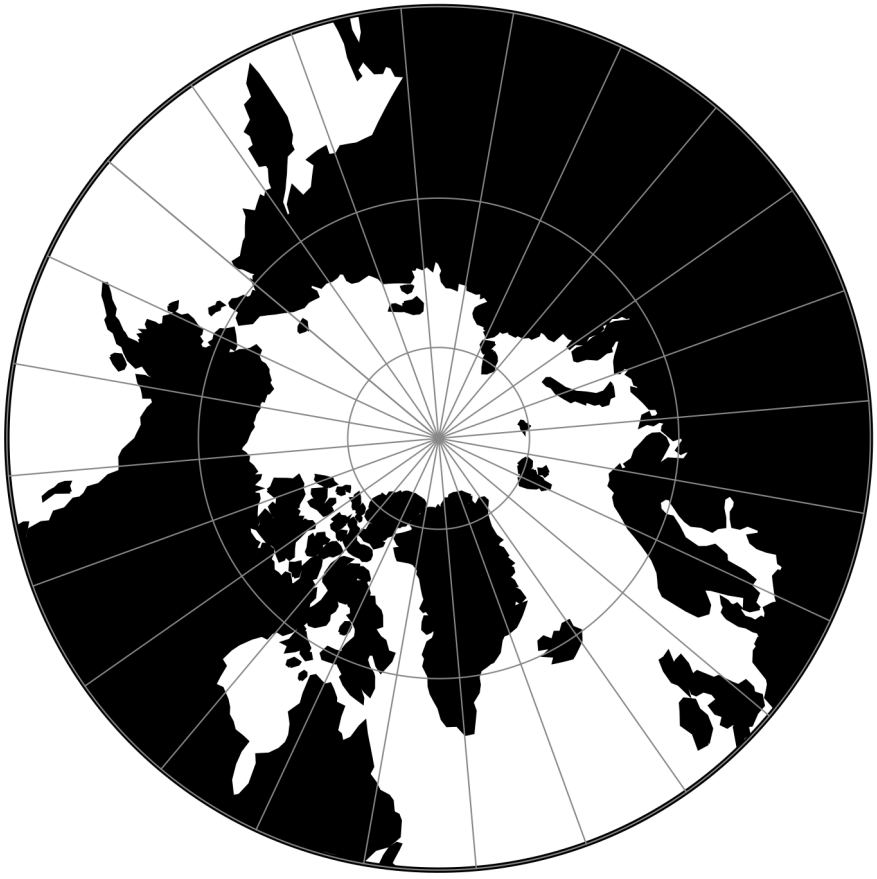


Fig. 43: proj-string: `+proj=gnom +lat_0=90 +lon_0=-50 +R=6.4e6`

#### 7.1.45.1 Parameters

> **Note**
>
> All parameters are optional for the Gnomonic projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

### 7.1.46 Goode Homolosine

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | goode |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

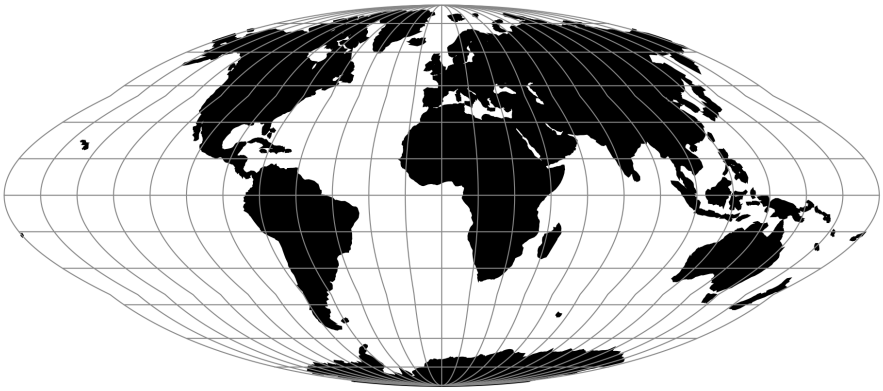


Fig. 44: proj-string: +proj=goode

#### 7.1.46.1 Parameters

**Note**

All parameters are optional for the Goode Homolosine projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.47 Modified Stereographic of 48 U.S.

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | The lower 48 states of the U.S. |
| **Alias** | gs48 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

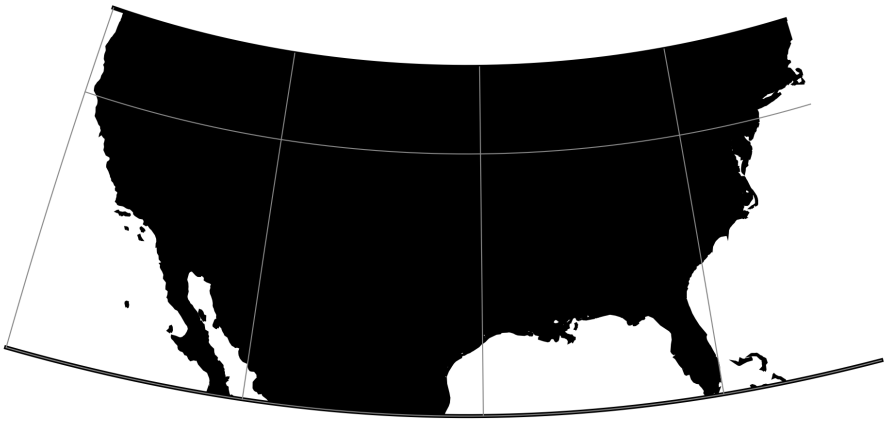


Fig. 45: proj-string: +proj=gs48

#### 7.1.47.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.48 Modified Stereographic of the 50 U.S. states

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | All 50 states of the U.S. |
| **Alias** | gs50 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.48.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

Fig. 46: proj-string: `+proj=gs50`

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.49 Guyou

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | guyou |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.49.1 Parameters

**Note**

All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

Fig. 47: proj-string: +proj=guyou

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.50 Hammer & Eckert-Greifendorff

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | hammer |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.50.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+W**=<value>

Set to 0.5 for the Hammer projection and 0.25 for the Eckert-Greifendorff projection. *+W* has to be larger than zero.

*Defaults to 0.5.*

**+M**=<value>

*+M* has to be larger than zero.

*Defaults to 1.0.*

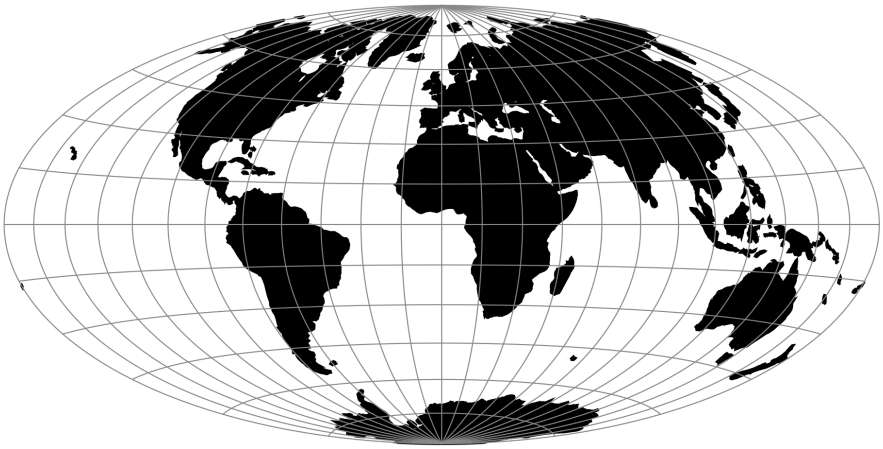


Fig. 48: proj-string: `+proj=hammer`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.51 Hatano Asymmetrical Equal Area

| | |
|---|---|
| **Classification** | *Pseudocylindrical Projection* |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | hatano |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

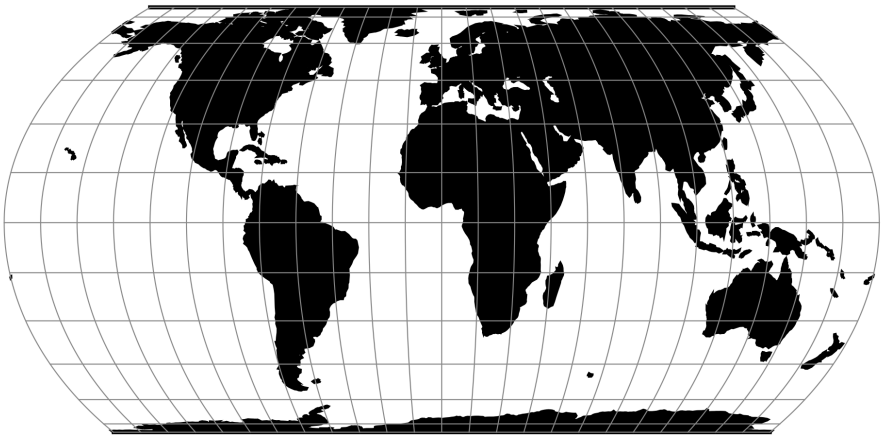


Fig. 49: proj-string: `+proj=hatano`

#### 7.1.51.1 Parameters

> **Note**
>
> All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### Mathematical Definition

#### Forward

$$
\begin{aligned}
x &= 0.85\lambda \cos\theta \\
y &= C_y \sin\theta \\
P(\theta) &= 2\theta + \sin 2\theta - C_p \sin\phi \\
P'(\theta) &= 2(1 + \cos 2\theta) \\
\theta_0 &= 2\phi
\end{aligned}
$$

| Condition | $C_y$ | $C_p$ |
|---|---|---|
| For $\phi > 0$ | 1.75859 | 2.67595 |
| For $\phi < 0$ | 1.93052 | 2.43763 |

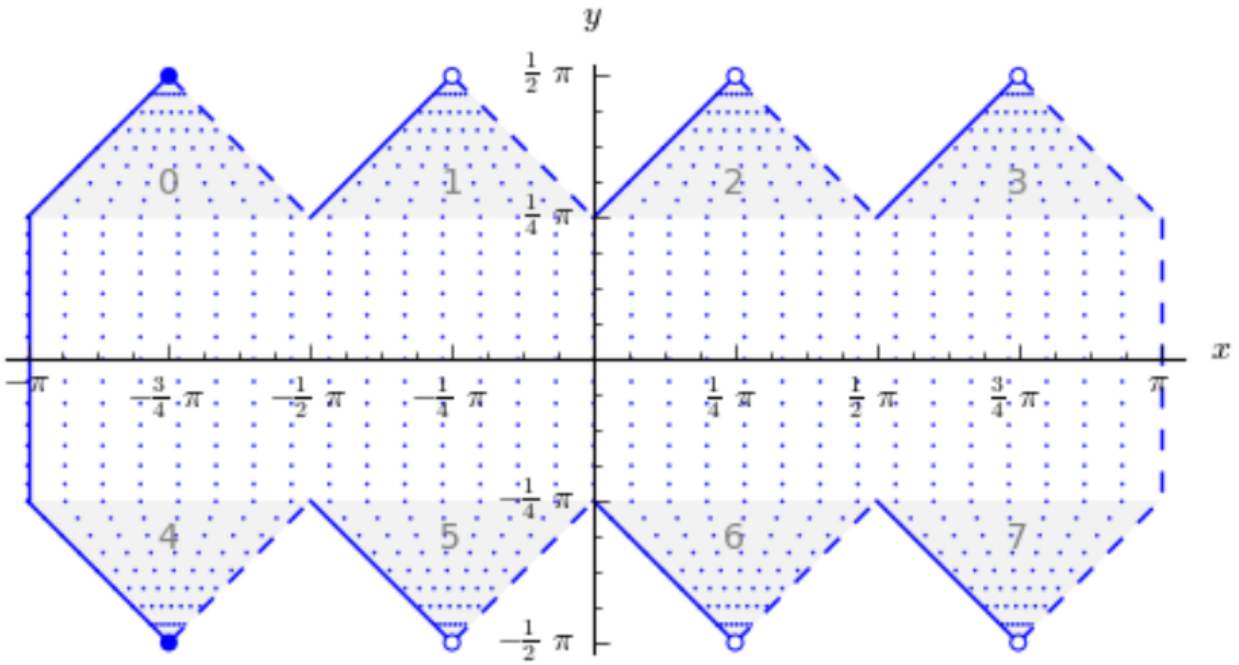
For $\phi = 0$, $y \leftarrow 0$, and $x \leftarrow 0.85\lambda$.

#### Further reading

1. Compare Map Projections
2. Mathworks

### 7.1.52 HEALPix

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | healpix |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |




The HEALPix projection is area preserving and can be used with a spherical and ellipsoidal model. It was initially developed for mapping cosmic background microwave radiation. The image below is the graphical representation of the mapping and consists of eight isomorphic triangular interrupted map graticules. The north and south contains four in which straight meridians converge polewards to a point and unequally spaced horizontal parallels. HEALPix provides a mapping in which points of equal latitude and equally spaced longitude are mapped to points of equal latitude and equally spaced longitude with the module of the polar interruptions.

#### 7.1.52.1 Usage

To run a forward HEALPix projection on a unit sphere model, use the following command:

```
proj +proj=healpix +a=1 -E <<EOF
0 0
EOF
# output
0 0 0.00 0.00
```

#### 7.1.52.2 Parameters

**Note**

All parameters for the projection are optional.

**+rot_xy**

Added in version 6.3.0.

Rotation of the HEALPix map in degrees. A positive value results in a clockwise rotation around (x_0, y_0) in the cartesian / projected coordinate space.

*Defaults to 0.0.*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

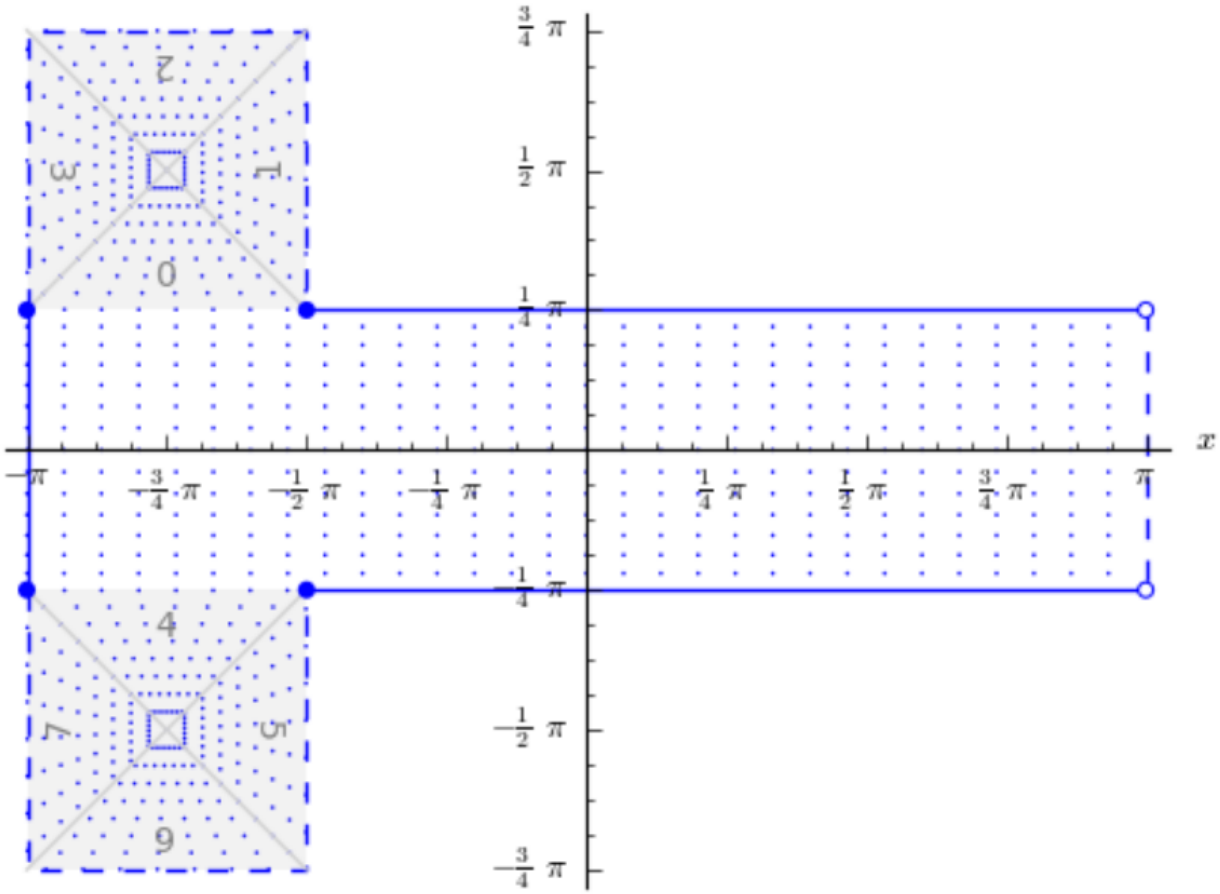
See *Ellipsoid size parameters* for more information.

#### 7.1.52.3 Further reading

1. NASA
2. Wikipedia

### 7.1.53 rHEALPix

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | rhealpix |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |




rHEALPix is a projection based on the HEALPix projection. The implementation of rHEALPix uses the HEALPix projection. The rHEALPix combines the peaks of the HEALPix into a square. The square's position can be translated and rotated across the x-axis which is a novel approach for the rHEALPix projection. The initial intention of using rHEALPix in the Spatial Computation Engine Science Collaboration Environment (SCENZGrid).

#### 7.1.53.1 Usage

To run a rHEALPix projection (on the default GRS80 ellipsoidal model,) use the following command:

```
proj +proj=rhealpix +north_square=2 -E << EOF
> 55 12
> EOF
55 12   6115727.86  1553840.13
```

#### 7.1.53.2 Parameters

**Note**

All parameters for the projection are optional.

**+north_square**

Position of the north polar square. Valid inputs are 0--3.

*Defaults to 0.0.*

**+south_square**

Position of the south polar square. Valid inputs are 0--3.

*Defaults to 0.0.*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.53.3 Further reading

1. NASA
2. Wikipedia

### 7.1.54 Interrupted Goode Homolosine

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | igh |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

The Interrupted Goode Homolosine projection is an equal-area composite projection intended for making world maps. Low latitudes are comprised of six separate Sinusoidal projection regions, and high latitudes are comprised of six

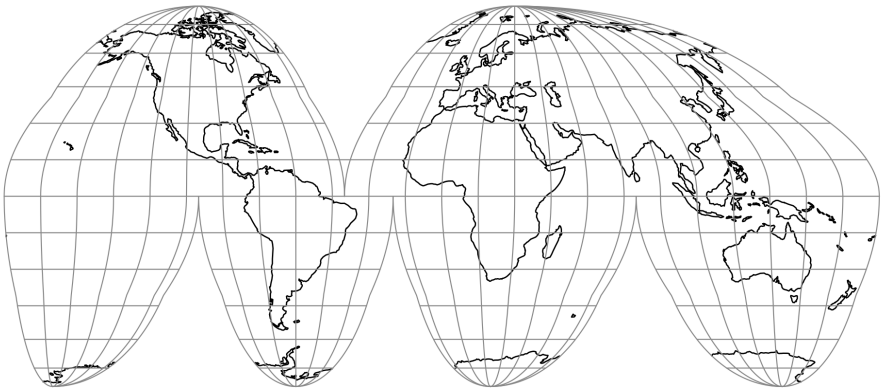


Fig. 50: proj-string: `+proj=igh`

separate Mollweide (`homolographic`) projections. The transition latitude is at 40d 44' 11.8", where the Sinusoidal and Mollweide scales are equal. The lobes in this projection are chosen to emphasize the land area of the Earth. This projection was first published in 1925 by J. P. Goode [Goode1925].

#### 7.1.54.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.55 Interrupted Goode Homolosine (Oceanic View)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | igh_o |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

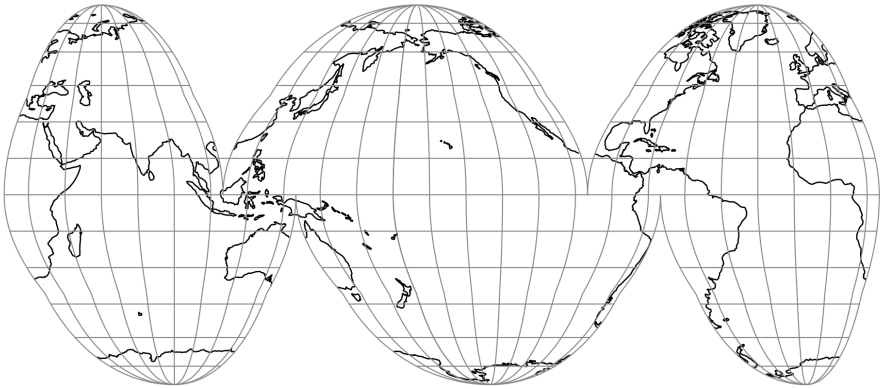


Fig. 51: proj-string: `+proj=igh_o +lon_0=-160`

The Interrupted Goode Homolosine (Oceanic View) projection is an equal-area composite projection intended for making maps of the Earth's oceans. Low latitudes are comprised of six separate Sinusoidal projection regions, and high latitudes are comprised of six separate Mollweide (`homolographic`) projections. The transition latitude is at 40d 44' 11.8", where the Sinusoidal and Mollweide scales are equal. The lobes in this projection are chosen to emphasize the ocean area of the Earth when used with a central longitude of -160 degrees. This projection was first published in 1925 by J. P. Goode [Goode1925].

#### 7.1.55.1 Parameters

**Note**

All parameters are optional for the projection. A value of +lon_0=-160 is recommended.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.56 Interrupted Mollweide

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | imoll |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

The Interrupted Mollweide projection is an equal-area projection intended for making world maps. The projection is comprised of six separate Mollweide (homolographic) projection regions. In contrast with the Interrupted Goode Homolosine projection, there is no transition latitude, which gives greater continuity at the cost of greater equatorial distortion. The lobes in this projection are chosen to emphasize the land area of the Earth. This projection was first published in 1919 by J. P. Goode [Goode1919].

#### 7.1.56.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

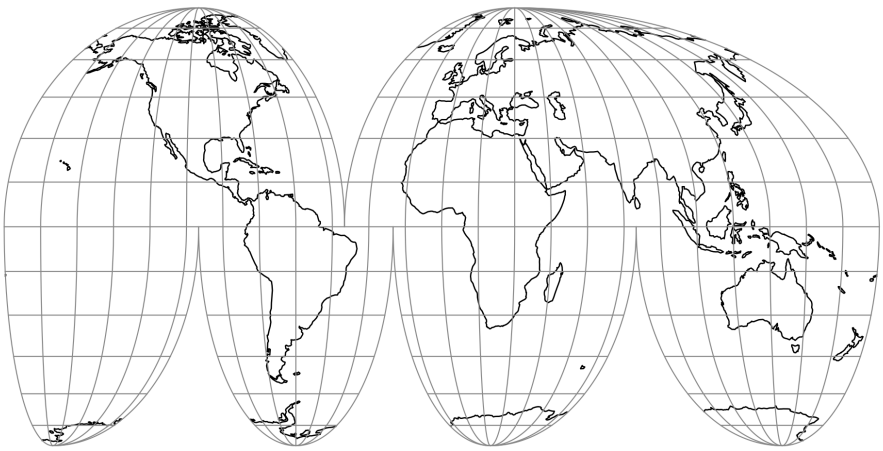


Fig. 52: proj-string: `+proj=imoll`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.57 Interrupted Mollweide (Oceanic View)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | imoll_o |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

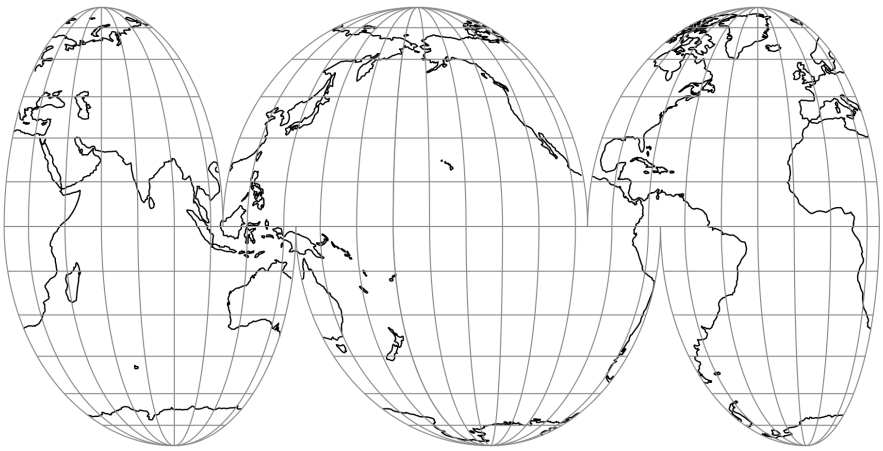


Fig. 53: proj-string: `+proj=imoll_o +lon_0=-160`

The Interrupted Mollweide (Oceanic View) projection is an equal-area projection intended for making maps of the Earth's oceans. The projection is comprised of six separate Mollweide (`homolographic`) projection regions. In contrast with the Interrupted Goode Homolosine (Oceanic View) projection, there is no transition latitude, which gives greater continuity at the cost of greater equatorial distortion. The lobes in this projection are chosen to emphasize the ocean area of the Earth when used with a central longitude of -160 degrees. This projection was first published in 1919 by J. P. Goode [Goode1919].

#### 7.1.57.1 Parameters

> **Note**
>
> All parameters are optional for the projection. A value of +lon_0=-160 is recommended.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.58 International Map of the World Polyconic

| | |
|---|---|
| **Classification** | Pseudoconical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | imw_p |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

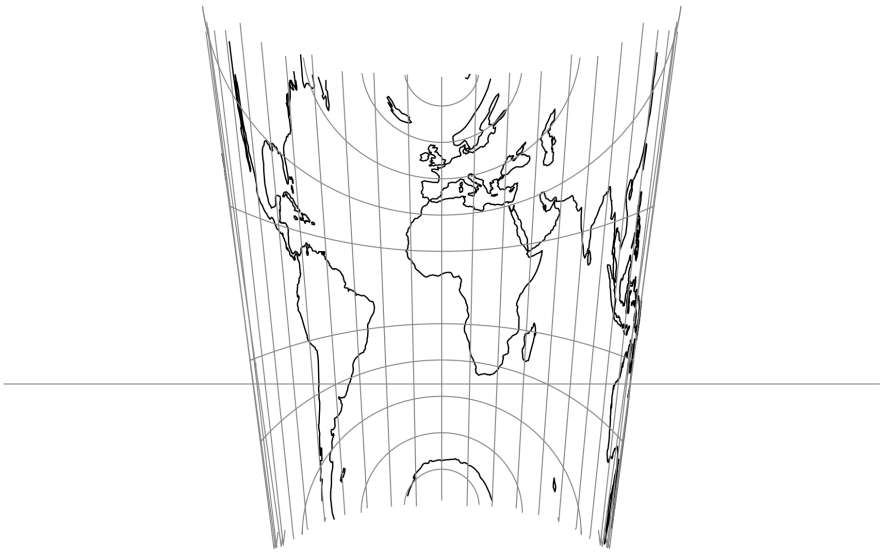


Fig. 54: proj-string: `+proj=imw_p +lat_1=30 +lat_2=-40`

#### 7.1.58.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_2=1.570796r*
>
> See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.59 Icosahedral Snyder Equal Area

Snyder's Icosahedral Equal Area map projection on an icosahedron polyhedral globe offers relatively low scale and angular distortion. The equations involved are relatively straight-forward. The interruptions remain a disadvantage, as with any low-error projection system applied to the entire globe [Snyder1992]. This projection is used as a basis for defining discrete global grid hierarchies.

| | |
|---|---|
| **Classification** | Polyhedral, equal area |
| **Available forms** | Forward and inverse, spherical |
| **Defined area** | Global |
| **Alias** | isea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

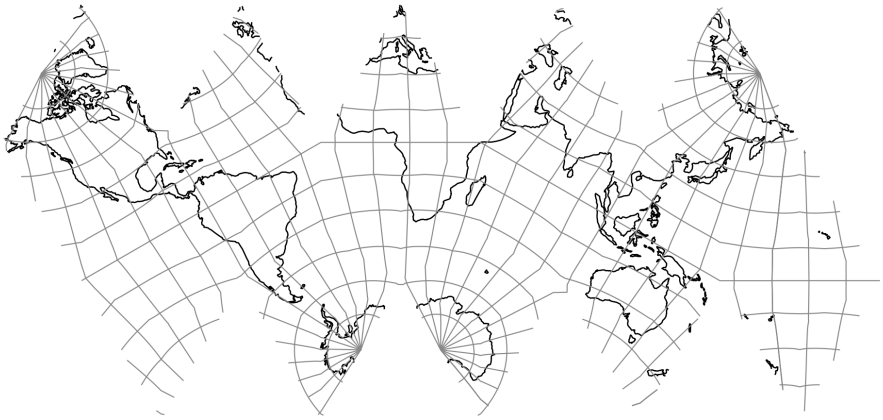


Fig. 55: proj-string: `+proj=isea`

**Note**

As the projection is only defined on a sphere, it should only be used with a spherical ellipsoid e.g., `+R=6371007.18091875` for a sphere with the authalic radius of the WGS84 ellipsoid. For mapping coordinates on the WGS84 ellipsoid to the authalic sphere, the input latitude should be converted from geodetic latitude to authalic latitude. A future version may automatically perform this conversion when using a non-spherical ellipsoid.

#### 7.1.59.1 Parameters

**Note**

All parameters are optional for the projection.

**+orient**=<string>

Can be set to either `isea` or `pole`. See Snyder's Figure 12 for pole orientation [Snyder1992].

*Defaults to isea*

**+azi**=<value>

Azimuth.

Not supported by the inverse.

*Defaults to 0.0*

**+aperture**=<value>

Not supported by the inverse.

*Defaults to 3.0*

**+resolution**=<value>

Not supported by the inverse.

*Defaults to 4.0*

**+mode**=<string>

Can be either `plane`, `di`, `dd` or `hex`.

Only `plane` supported by the inverse.

*Defaults to plane*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.60 Kavrayskiy V

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | kav5 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

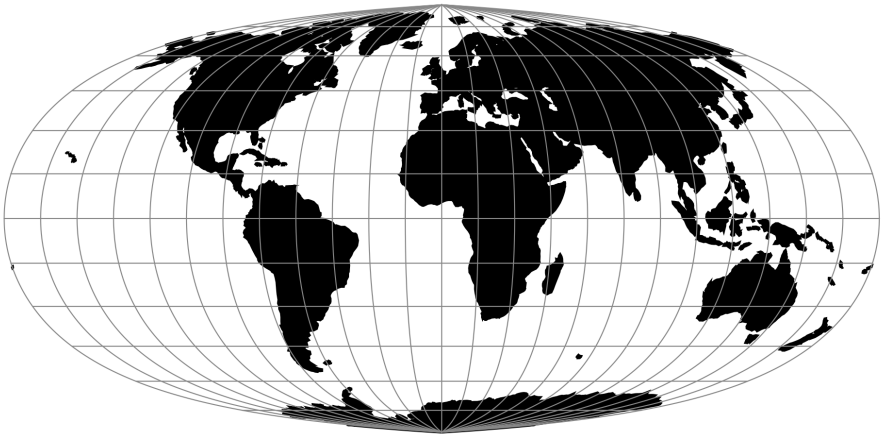


Fig. 56: proj-string: `+proj=kav5`

**Note**

This projection name may also be transliterated as Kavraisky V.

Created by Soviet cartographer Vladimir V. Kavrayskiy in 1933 [Snyder1993].

#### 7.1.60.1 Parameters

> **Note**
>
> All parameters are optional for the Kavrayskiy V projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.61 Kavrayskiy VII

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | kav7 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

> **Note**
>
> This projection name may also be transliterated as Kavraisky VII.

Created by Soviet cartographer Vladimir V. Kavrayskiy in 1939 [Snyder1993].

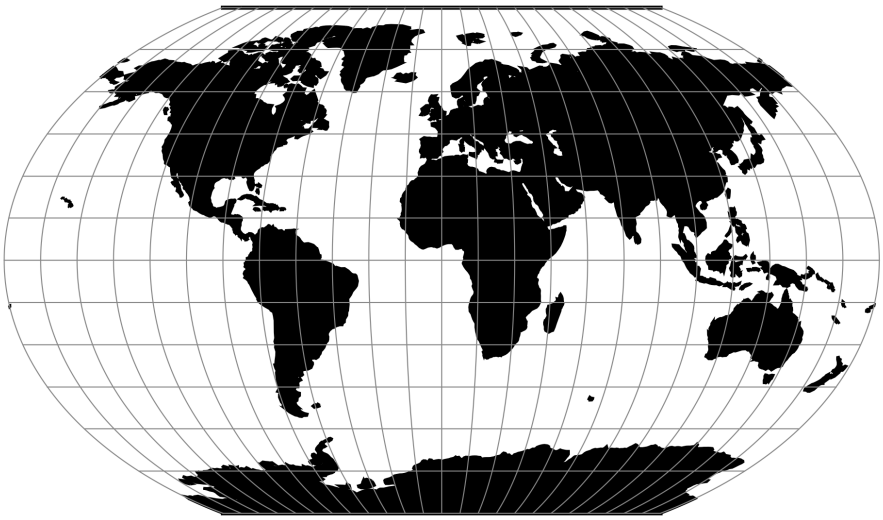


Fig. 57: proj-string: `+proj=kav7`

#### 7.1.61.1 Parameters

> **Note**
>
> All parameters are optional for the Kavrayskiy VII projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.62 Krovak

| | |
|---|---|
| **Classification** | Conformal Conical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global, but more accurate around Czech Republic and Slovakia |
| **Alias** | krovak |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

By default, coordinates in the forward direction are output in easting, northing, and negative in the Czech Republic and Slovakia, with absolute value of easting/westing being smaller than absolute value of northing/southing.

See also *Modified Krovak* for a variation of Krovak used with the S-JTSK/05 datum in the Czech Republic.

**Note**

Before PROJ 9.4, using other values for x_0 or y_0 than the default 0 would lead to incorrect results when not using the +czech switch.

#### 7.1.62.1 Parameters

**Note**

All parameters are optional for the Krovak projection.

The latitude of pseudo standard parallel is hardcoded to 78.5° and the ellipsoid to Bessel.

**+czech**

Reverse the sign of the output coordinates, as is tradition in the Czech Republic, to be westing, southing (positive values in Czech Republic and Slovakia).

**+lon_0**=<value>

Longitude of projection center.

*Defaults to 24°50' (24.8333333333333)*

**+lat_0**=<value>

Latitude of projection center.

*Defaults to 49.5*

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 0.9999*

Fig. 58: proj-string: +proj=krovak

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.63 Laborde

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global, but more accurate around Madagascar |
| **Alias** | labrd |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.63.1 Parameters

##### Required

**+lat_0**=<value>

Latitude of projection center. Must not be zero.

##### Optional

**+azi**=<value>

Azimuth of the central line.

*Defaults to 0.0*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

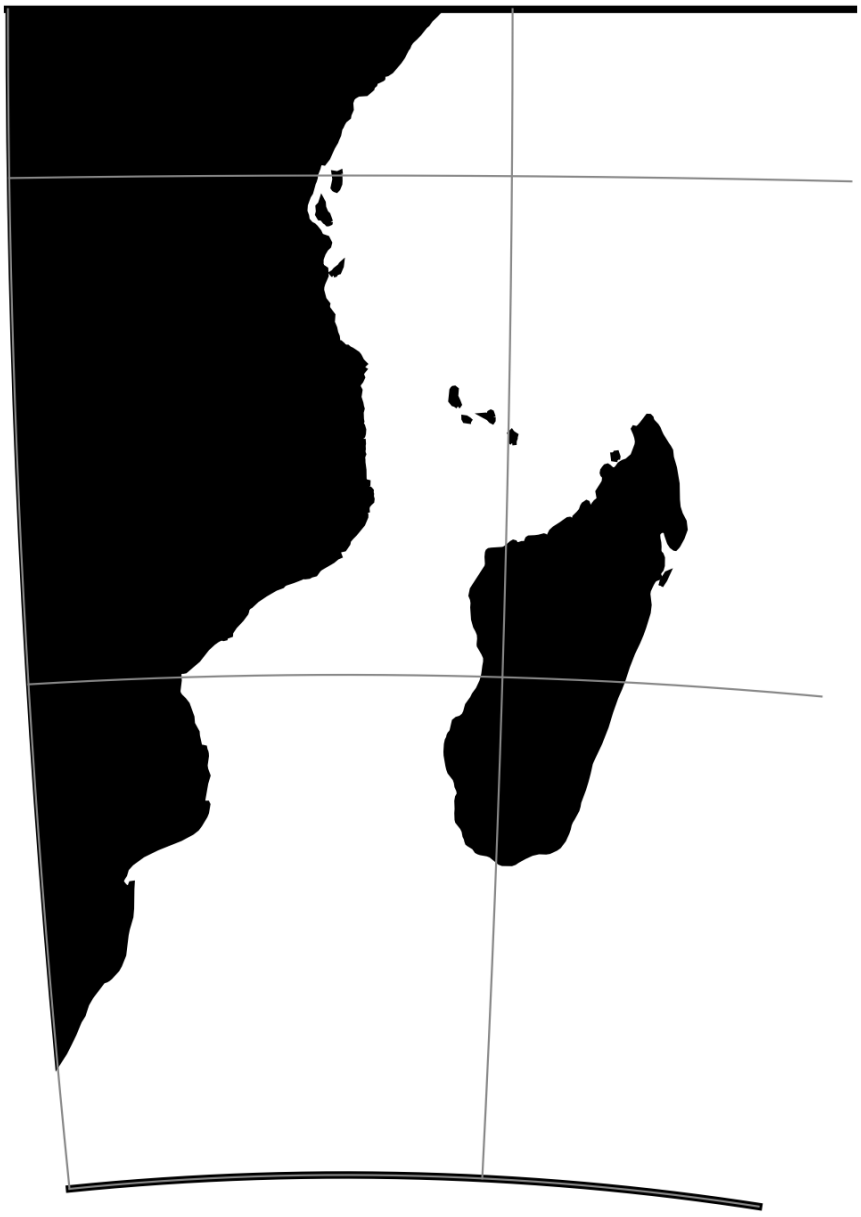


Fig. 59: proj-string: `+proj=labrd +lon_0=40 +lat_0=-10`

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.64 Lambert Azimuthal Equal Area

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | laea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.64.1 Parameters

**Note**

All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

Fig. 60: proj-string: `+proj=laea`

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.65 Lagrange

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | lagrng |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.65.1 Parameters

**Note**

All parameters are optional for the projection.

**+W**=<value>

The factor *+W* is the ratio of the difference in longitude from the central meridian to the a circular meridian to 90. *+W* must be a positive value.

*Defaults to 2.0*

Fig. 61: proj-string: `+proj=lagrng`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.66 Larrivee

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | larr |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 62: proj-string: `+proj=larr`

#### 7.1.66.1 Parameters

**Note**

All parameters are optional for the Larrivee projection.

`+lon_0=<value>`

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.67 Laskowski

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | lask |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.67.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

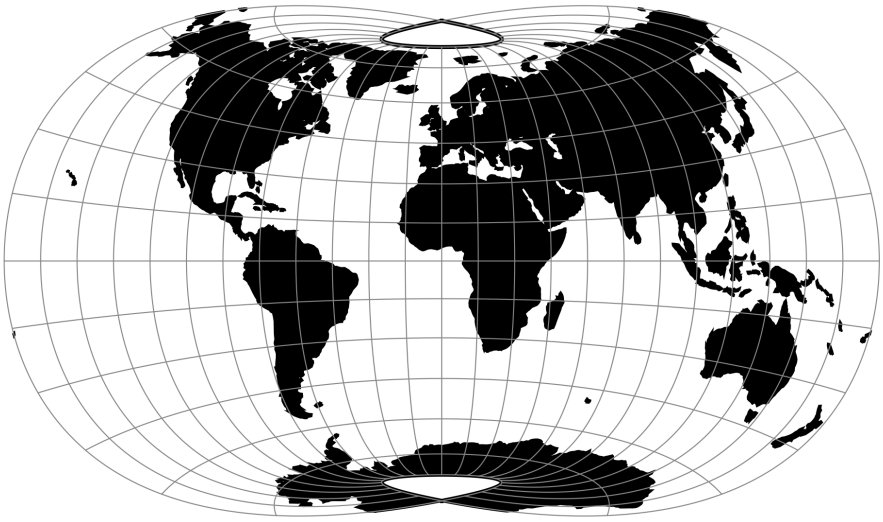


Fig. 63: proj-string: `+proj=lask`

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.68 Lambert Conformal Conic

A Lambert Conformal Conic projection (LCC) is a conic map projection used for aeronautical charts, portions of the State Plane Coordinate System, and many national and regional mapping systems. It is one of seven projections introduced by Johann Heinrich Lambert in 1772.

It has several different forms: with one and two standard parallels (referred to as 1SP and 2SP in EPSG guidance notes). Additionally we provide "2SP Michigan" form which is very similar to normal 2SP, but with a scaling factor on the ellipsoid (given as *k_0* parameter). It is implemented as per EPSG Guidance Note 7-2 (version 54, August 2018, page 25). It is used in a few systems in the EPSG database which justifies adding this otherwise non-standard projection.

| | |
|---|---|
| **Classification** | Conformal conic |
| **Available forms** | Forward and inverse, spherical and ellipsoidal . One or two standard parallels (1SP and 2SP). "LCC 2SP Michigan" form can be used by setting `+k_0` parameter to specify ellipsoid scale. |
| **Defined area** | Best for regions predominantly east–west in extent and located in the middle north or south latitudes. |
| **Alias** | lcc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 64: proj-string: `+proj=lcc +lon_0=-90 +lat_1=33 +lat_2=45`

#### 7.1.68.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_1=1.570796r*

See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_2=1.570796r*

See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+k_0**=<value>

This parameter can represent two different values depending on the form of the projection. In LCC 1SP it determines the scale factor at natural origin. In LCC 2SP Michigan it determines the ellipsoid scale factor.

*Defaults to 1.0.*

#### 7.1.68.2 Further reading

1. Wikipedia
2. Wolfram Mathworld
3. John P. Snyder "Map projections: A working manual" (pp. 104-110)
4. ArcGIS documentation on "Lambert Conformal Conic"
5. EPSG Guidance Note 7-2 (version 54, August 2018, page 25)

### 7.1.69 Lambert Conformal Conic Alternative

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | lcca |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.69.1 Parameters

**Note**

All parameters are optional for the projection.

Fig. 65: proj-string: `+proj=lcca +lat_0=35`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.70 Lambert Equal Area Conic

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | leac |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 66: proj-string: `+proj=leac`

#### 7.1.70.1 Parameters

> **Note**
>
> All parameters are optional for the Lambert Equal Area Conic projection.

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+south**

Sets the second standard parallel to 90°S. When the flag is off the second standard parallel is set to 90°N.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.71 Lee Oblated Stereographic

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | lee_os |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.71.1 Parameters

**Note**

All parameters are optional for the projection.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.72 Loximuthal

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | loxim |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 67: proj-string: `+proj=lee_os`

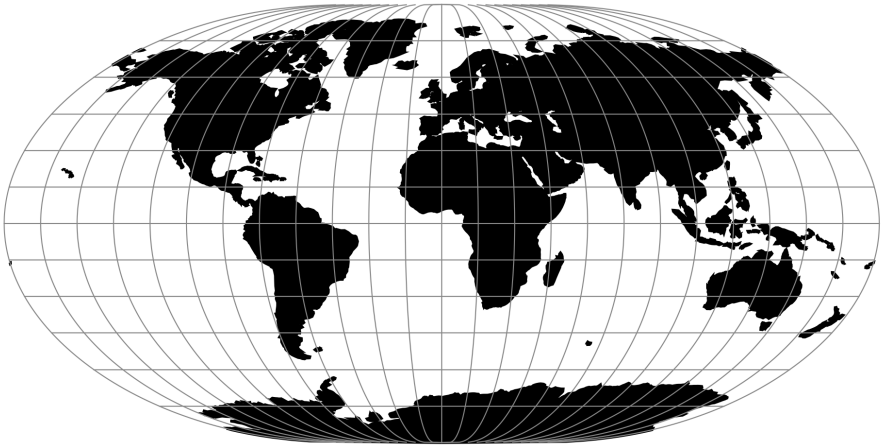


Fig. 68: proj-string: `+proj=loxim`

#### 7.1.72.1 Parameters

> **Note**
>
> All parameters are optional for the Loximuthal projection.

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.73 Space oblique for LANDSAT

Space oblique for LANDSAT is a specialization of *Space Oblique Mercator* which specifies the appropriate ascending longitude, inclination angle, and orbital period for Landsat satellites, based on their satellite designation / id and path number.

"Paths" are the repeating ground tracks that are traced by sun-synchronous satellites like those of the Landsat program.

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | lsat |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.73.1 Parameters

##### Required

**+lsat**=<value>

Landsat satellite used for the projection. Value between 1 and 5.

**+path**=<value>

Selected path of satellite. Value between 1 and 253 when *+lsat* is set to 1,2 or 3, otherwise valid input is between 1 and 233.

##### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

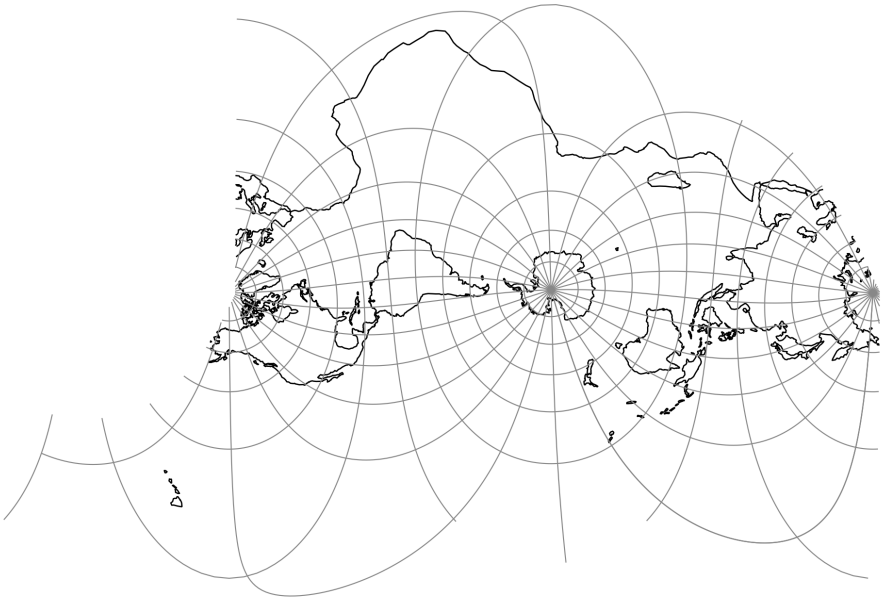


Fig. 69: proj-string: `+proj=lsat +lat_1=-60 +lat_2=60 +lsat=2 +path=2`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.73.2 Further reading

1. Landsat Wikipedia
2. *Space Oblique Mercator Projection*

### 7.1.74 McBryde-Thomas Flat-Polar Sine (No. 1)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | mbt_s |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

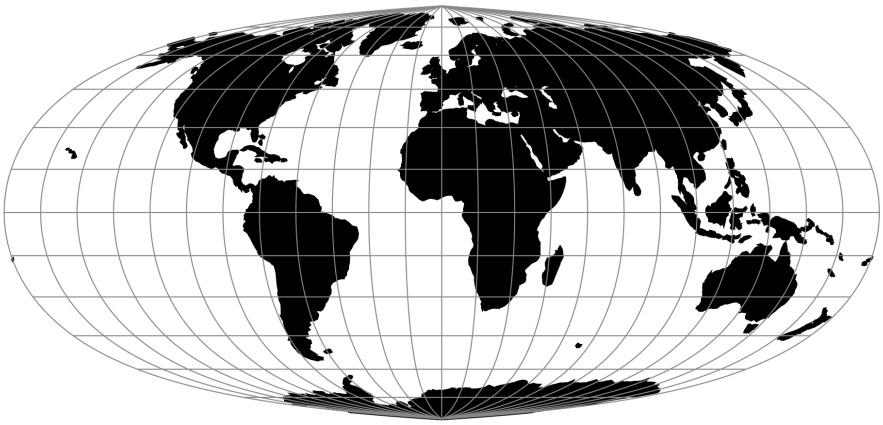


Fig. 70: proj-string: `+proj=mbt_s`

#### 7.1.74.1 Parameters

**Note**

All parameters are optional for the McBryde-Thomas Flat-Polar Sine projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.75 McBryde-Thomas Flat-Pole Sine (No. 2)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | mbt_fps |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

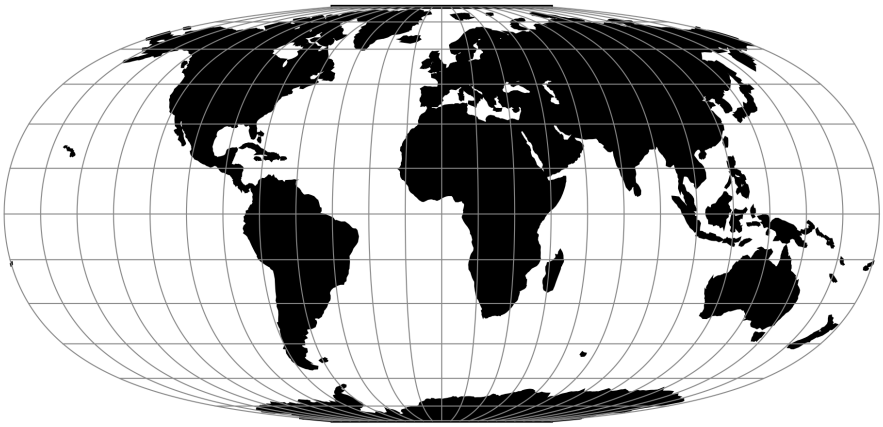


Fig. 71: proj-string: +proj=mbt_fps

#### 7.1.75.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.76 McBride-Thomas Flat-Polar Parabolic

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | mbtfpp |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.76.1 Parameters

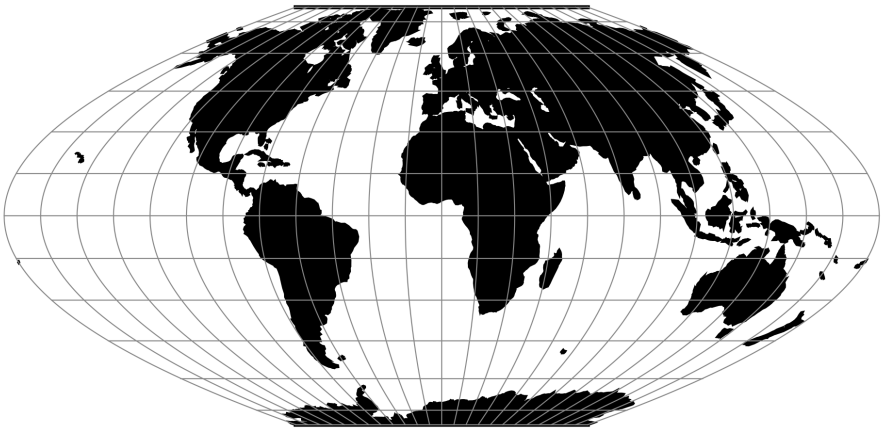


Fig. 72: proj-string: `+proj=mbtfpp`

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.77 McBryde-Thomas Flat-Polar Quartic

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | mbtfpq |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

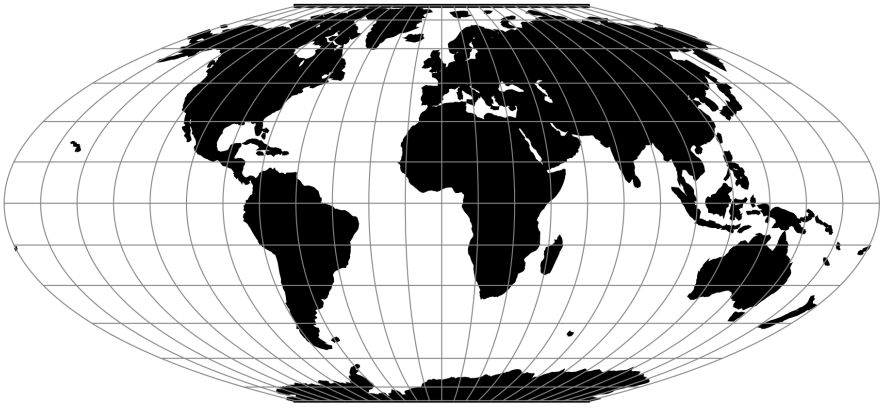


Fig. 73: proj-string: `+proj=mbtfpq`

#### 7.1.77.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.78 McBryde-Thomas Flat-Polar Sinusoidal

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | mbtfps |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

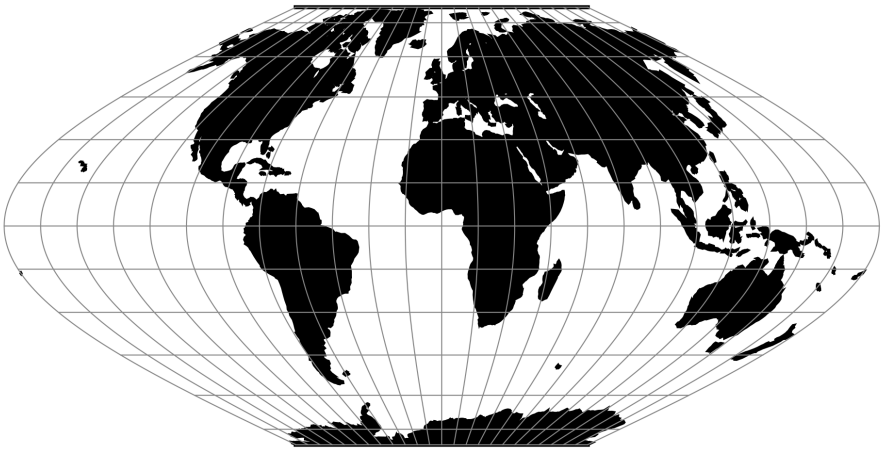


Fig. 74: proj-string: `+proj=mbtfps`

#### 7.1.78.1 Parameters

**Note**

All parameters are optional for the McBryde-Thomas Flat-Polar Sinusoidal projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.79 Mercator

The Mercator projection is a cylindrical map projection that origins from the 16th century. It is widely recognized as the first regularly used map projection. It is a conformal projection in which the equator projects to a straight line at constant scale. The projection has the property that a rhumb line, a course of constant heading, projects to a straight line. This makes it suitable for navigational purposes.

| | |
|---|---|
| **Classification** | Conformal cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global, but best used near the equator |
| **Alias** | merc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.79.1 Usage

Applications should be limited to equatorial regions, but is frequently used for navigational charts with latitude of true scale (*+lat_ts*) specified within or near chart's boundaries. It is considered to be inappropriate for world maps because of the gross distortions in area; for example the projected area of Greenland is larger than that of South America, despite the fact that Greenland's area is $\frac{1}{8}$ that of South America [Snyder1987].

Example using latitude of true scale:

```
$ echo 56.35 12.32 | proj +proj=merc +lat_ts=56.5
3470306.37    759599.90
```

Example using scaling factor:

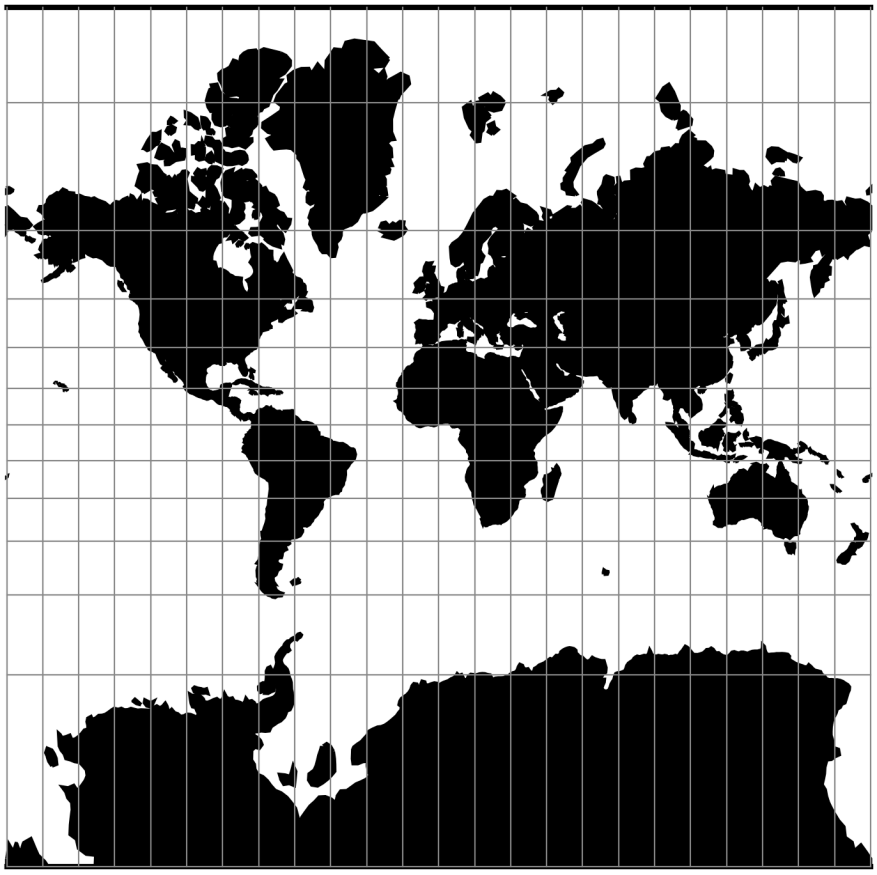


Fig. 75: proj-string: `+proj=merc`

```
echo 56.35 12.32 | proj +proj=merc +k_0=2
12545706.61     2746073.80
```

Note that *+lat_ts* and *+k_0* are mutually exclusive. If used together, *+lat_ts* takes precedence over *+k_0*.

#### 7.1.79.2 Parameters

> **Note**
>
> All parameters for the projection are optional.

**+lat_ts**=<value>

Latitude of true scale. Defines the latitude where scale is not distorted. Takes precedence over +k_0 if both options are used together.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_ts=1.570796r*
>
> See *Projection Units* for more information.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, `+R` takes precedence.

See *Ellipsoid size parameters* for more information.

#### 7.1.79.3 Mathematical definition

##### Spherical form

For the spherical form of the projection we introduce the scaling factor:

$$k_0 = \cos\phi_{ts}$$

##### Forward projection

$$x = k_0 R\lambda; \qquad y = k_0 R\psi$$

$$\begin{aligned}\psi &= \ln\tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right) \\ &= \sinh^{-1}\tan\phi\end{aligned}$$

The quantity $\psi$ is the isometric latitude.

##### Inverse projection

$$\lambda = \frac{x}{k_0 R}; \qquad \psi = \frac{y}{k_0 R}$$

$$\begin{aligned}\phi &= \frac{\pi}{2} - 2\tan^{-1}\exp(-\psi) \\ &= \tan^{-1}\sinh\psi\end{aligned}$$

##### Ellipsoidal form

For the ellipsoidal form of the projection we introduce the scaling factor:

$$k_0 = m(\phi_{ts})$$

where

$$m(\phi) = \frac{\cos\phi}{\sqrt{1 - e^2\sin^2\phi}}$$

$a\,m(\phi)$ is the radius of the circle of latitude $\phi$.

### Forward projection

$$x = k_0 a\lambda; \qquad y = k_0 a\psi$$

$$\begin{aligned}\psi &= \ln\tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right) - \frac{1}{2}e\ln\left(\frac{1 + e\sin\phi}{1 - e\sin\phi}\right)\\ &= \sinh^{-1}\tan\phi - e\tanh^{-1}(e\sin\phi)\end{aligned}$$

### Inverse projection

$$\lambda = \frac{x}{k_0 a}; \quad \psi = \frac{y}{k_0 a}$$

The latitude $\phi$ is found by inverting the equation for $\psi$. This follows the method given by [Karney2011tm]. Start by introducing the conformal latitude

$$\chi = \tan^{-1}\sinh\psi$$

The tangents of the latitudes $\tau = \tan\phi$ and $\tau' = \tan\chi = \sinh\psi$ are related by

$$\tau' = \tau\sqrt{1 + \sigma^2} - \sigma\sqrt{1 + \tau^2}$$

where

$$\sigma = \sinh\bigl(e\tanh^{-1}(e\tau/\sqrt{1 + \tau^2})\bigr)$$

This is obtained by taking the $\sinh$ of the equation for $\psi$ and using the multiple argument formula. The equation for $\tau'$ can be solved to give $\tau$ using Newton's method using $\tau = \tau'/(1 - e^2)$ as an initial guess and with the needed derivative given by

$$\frac{d\tau'}{d\tau} = \frac{1 - e^2}{1 + (1 - e^2)\tau^2}\sqrt{1 + \tau'^2}\sqrt{1 + \tau^2}$$

This converges after no more than 2 iterations. Finally set $\phi = \tan^{-1}\tau$.

#### 7.1.79.4 Further reading

1. Wikipedia
2. Wolfram Mathworld

## 7.1.80 Miller Oblated Stereographic

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | mil_os |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

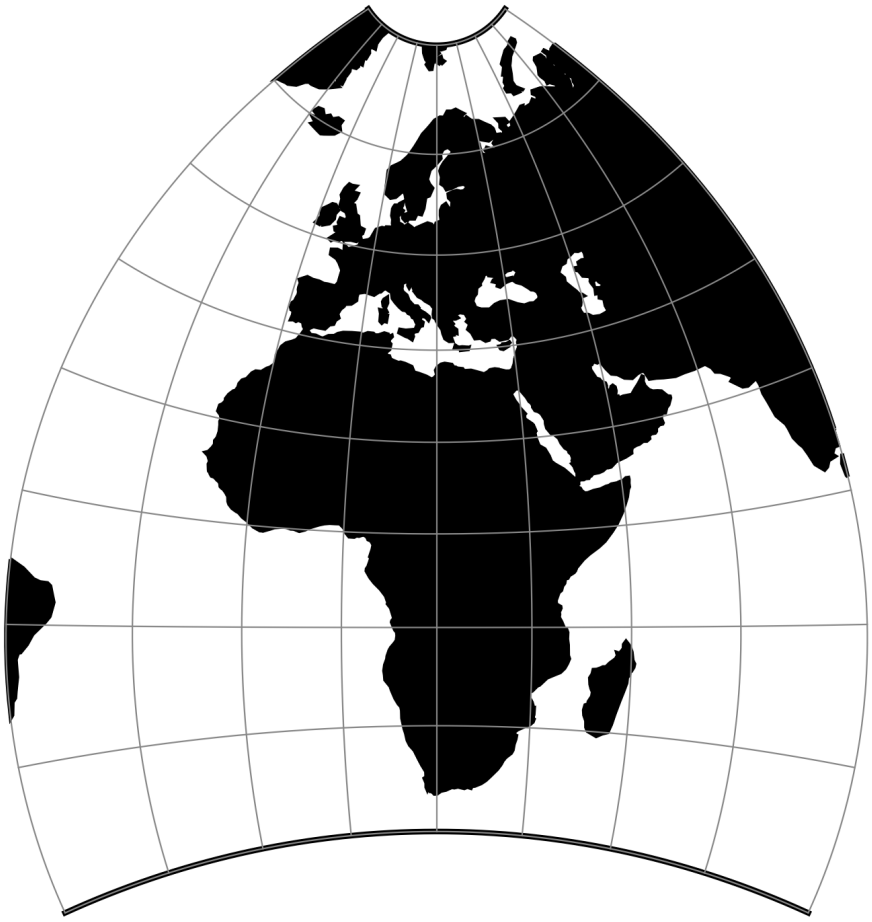


Fig. 76: proj-string: `+proj=mil_os`

#### 7.1.80.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+ellps**=<value>
: The name of a built-in ellipsoid definition.

  See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

  *Defaults to "GRS80".*

**+x_0**=<value>
: False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

**+y_0**=<value>
: False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

### 7.1.81 Miller Cylindrical

The Miller cylindrical projection is a modified Mercator projection, proposed by Osborn Maitland Miller in 1942. The latitude is scaled by a factor of $\frac{4}{5}$, projected according to Mercator, and then the result is multiplied by $\frac{5}{4}$ to retain scale along the equator.

| | |
|---|---|
| **Classification** | Neither conformal nor equal area cylindrical |
| **Available forms** | Forward and inverse spherical |
| **Defined area** | Global, but best used near the equator |
| **Alias** | mill |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.81.1 Usage

The Miller Cylindrical projection is used for world maps and in several atlases, including the National Atlas of the United States (USGS, 1970, p. 330-331) [Snyder1987].

Example using Central meridian 90°W:

```
$ echo -100 35 | proj +proj=mill +lon_0=90w
-1113194.91      4061217.24
```

#### 7.1.81.2 Parameters

> **Note**
>
> All parameters for the projection are optional.

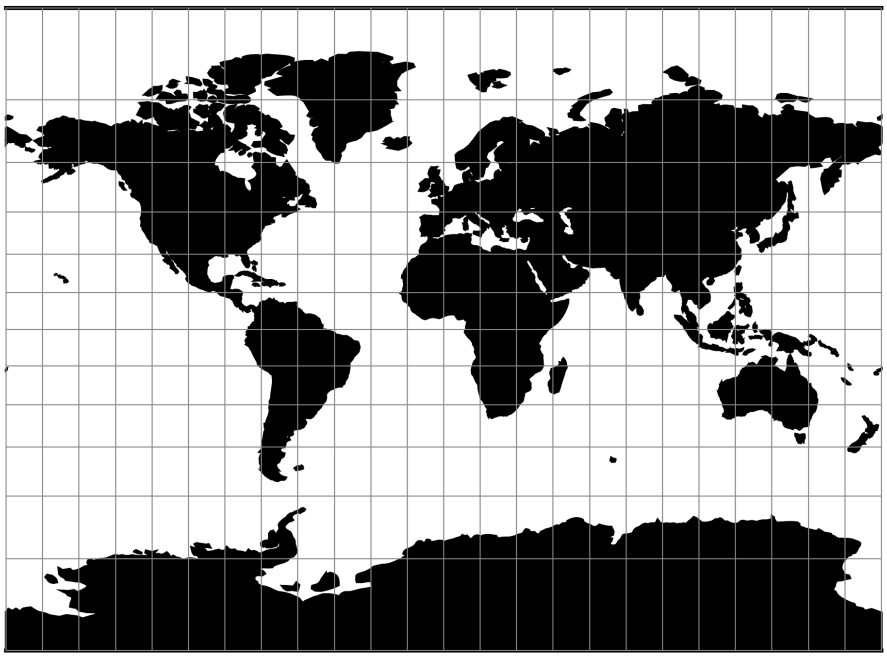


Fig. 77: proj-string: `+proj=mill`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.81.3 Mathematical definition

The formulas describing the Miller projection are all taken from [Snyder1987].

##### Forward projection

$$x = \lambda$$

$$y = 1.25 * \ln\left[\tan\left(\frac{\pi}{4} + 0.4 * \phi\right)\right]$$

##### Inverse projection

$$\lambda = x$$

$$\phi = 2.5 * (\arctan\left[e^{0.8*y}\right] - \frac{\pi}{4})$$

#### 7.1.81.4 Further reading

1. Wikipedia

### 7.1.82 Space oblique for MISR

Space oblique for MISR is a specialization of *Space Oblique Mercator* which specifies the appropriate ascending longitude, inclination angle, and orbital period for Multi-angle Imaging Spectroradiometer (MISR) orbits, based on the path number for the orbit.

"Paths" are the repeating ground tracks that are traced by sun-synchronous satellites like Terra on which the MISR instrument is aboard.

| | |
|---|---|
| **Classification** | Conformal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | misrsom |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.82.1 Parameters

##### Required

**+path**=<value>

Selected path of satellite. Value between 1 and 233.

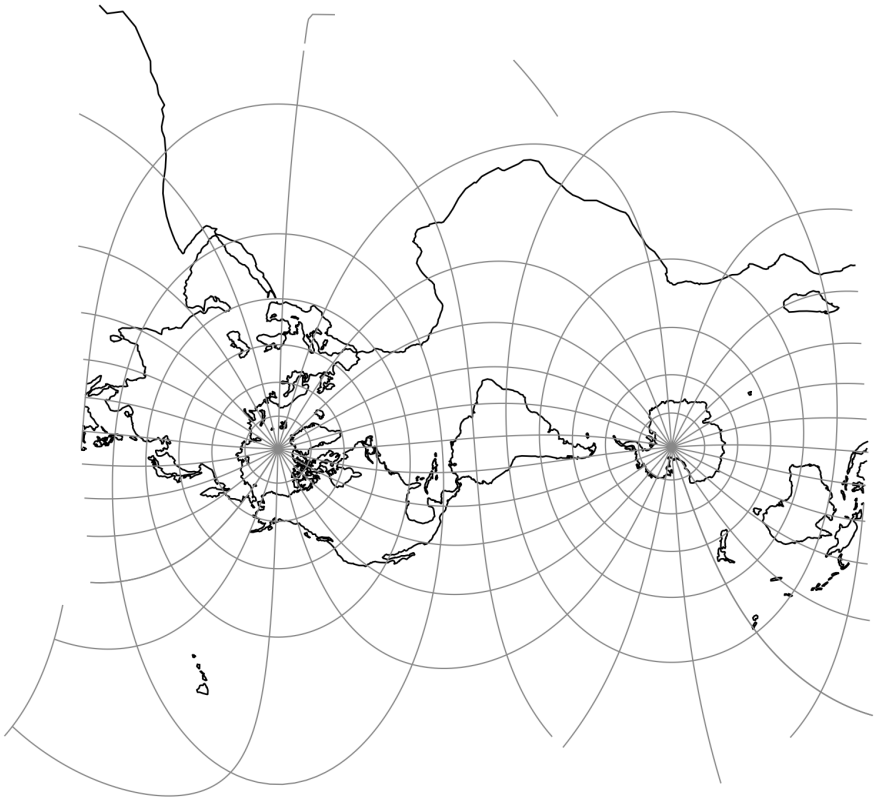


Fig. 78: proj-string: `+proj=misrsom +path=1`

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

1. MISR Wikipedia
2. Terra Wikipedia
3. *Space Oblique Mercator Projection*

## 7.1.83 Modified Krovak

Added in version 9.4.0.

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Czech Republic |
| **Alias** | mod_krovak |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 79: proj-string: `+proj=mod_krovak`

Modified Krovak builds upon traditional *Krovak*, with corrective terms that are better suited when using it with the S-JTSK/05 datum. This method is specific to the Czech Republic. Due to the corrective terms, this projection method is no longer strictly conformal.

By default, coordinates in the forward direction are output in easting, northing, and negative in the Czech Republic, with absolute value of easting/westing being smaller than absolute value of northing/southing. To distinguish it from regular Krovak, the usual value for `+x_0` and `+y_0` in Modified Krovak is typically 5,000,000.

#### 7.1.83.1 Parameters

**Note**

All parameters are optional for the Modified Krovak projection.

The latitude of pseudo standard parallel is hardcoded to 78.5° and the ellipsoid to Bessel.

**+czech**

Reverse the sign of the output coordinates, as is tradition in the Czech Republic, to be westing, southing (positive values in Czech Republic)

**+lon_0**=<value>

Longitude of projection center.

*Defaults to 24°50' (24.8333333333333)*

**+lat_0**=<value>

Latitude of projection center.

*Defaults to 49.5*

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 0.9999*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.84 Mollweide

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | moll |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

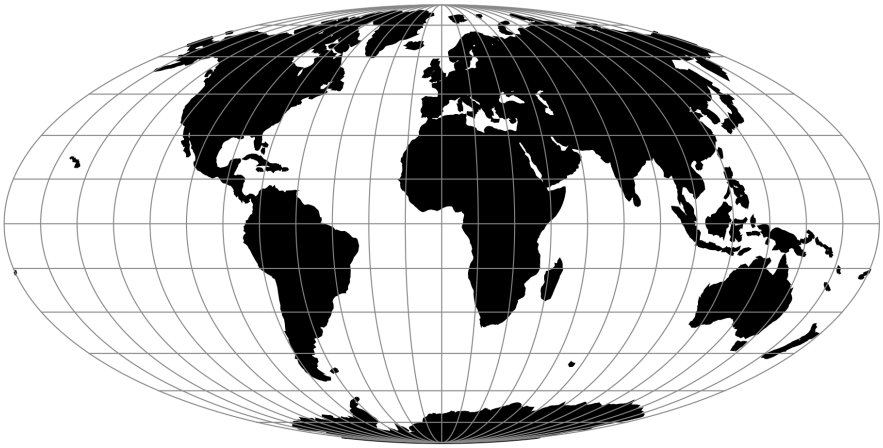


Fig. 80: proj-string: `+proj=moll`

#### 7.1.84.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.85 Murdoch I

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | murd1 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 81: proj-string: `+proj=murd1 +lat_1=30 +lat_2=50`

#### 7.1.85.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_1=1.570796r*

See *Projection Units* for more information.

`+lat_2`=<value>

Second standard parallel.

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_2=1.570796r*

See *Projection Units* for more information.

### Optional

`+lon_0`=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

`+R`=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

`+x_0`=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

`+y_0`=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.86 Murdoch II

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | murd2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

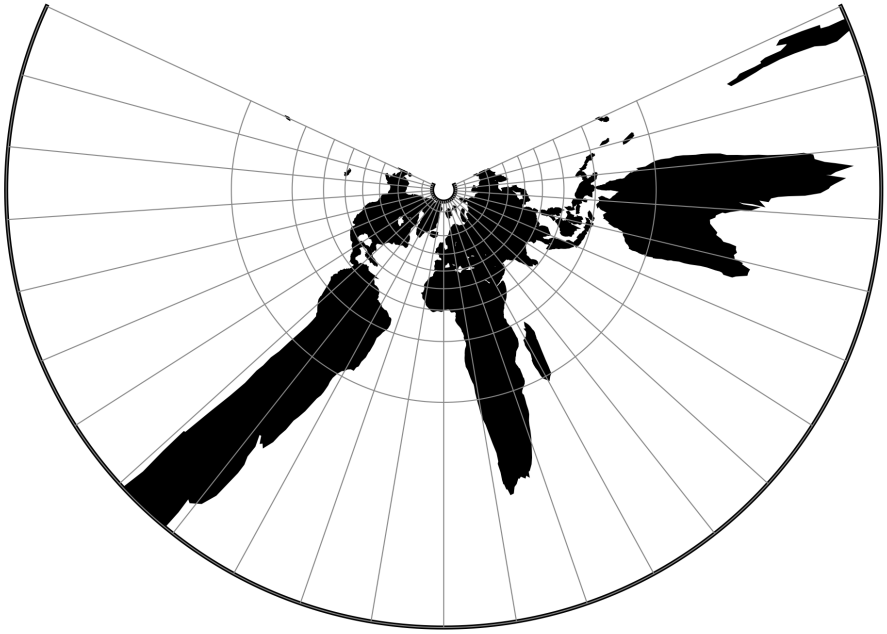


Fig. 82: proj-string: `+proj=murd2 +lat_1=30 +lat_2=50`

#### 7.1.86.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_2=1.570796r*
>
> See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.87 Murdoch III

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | murd3 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 83: proj-string: `+proj=murd3 +lat_1=30 +lat_2=50`

#### 7.1.87.1 Parameters

#### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_2=1.570796r*
>
> See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.88 Natural Earth

| | |
|---|---|
| **Classification** | Pseudo cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | natearth |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

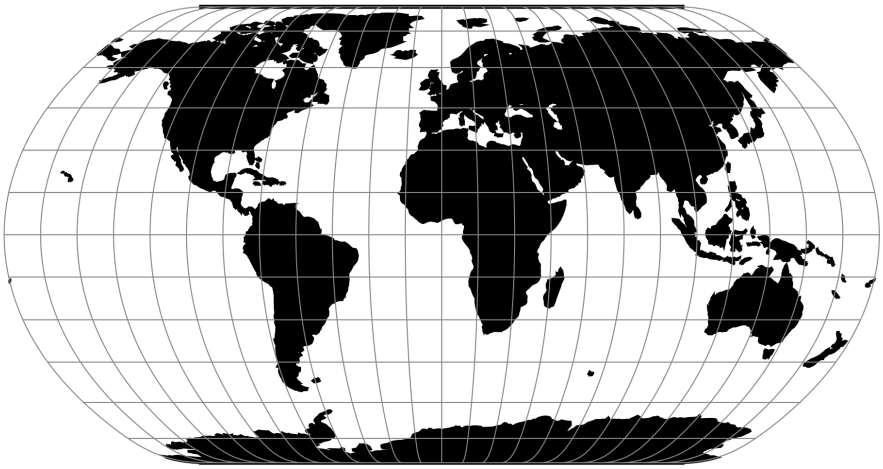


Fig. 84: proj-string: `+proj=natearth`

The Natural Earth projection is intended for making world maps. A distinguishing trait is its slightly rounded corners fashioned to emulate the spherical shape of Earth. The meridians (except for the central meridian) bend acutely inward as they approach the pole lines, giving the projection a hint of three-dimensionality. This bending also suggests that the meridians converge at the poles instead of truncating at the top and bottom edges. The distortion characteristics of the Natural Earth projection compare favorably to other world map projections.

#### 7.1.88.1 Usage

The Natural Earth projection has no special options so usage is simple. Here is an example of an inverse projection on a sphere with a radius of 7500 m:

```
$ echo 3500 -8000 | proj -I +proj=natearth +a=7500
37d54'6.091"E  61d23'4.582"S
```

#### 7.1.88.2 Parameters

**Note**

All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.88.3 Further reading

1. Wikipedia

### 7.1.89 Natural Earth II

| | |
|---|---|
| **Classification** | Pseudo cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | natearth2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

The Natural Earth II projection is intended for making world maps. At high latitudes, meridians bend steeply toward short pole lines resulting in a map with highly rounded corners that resembles an elongated globe.

See [Savric2015]

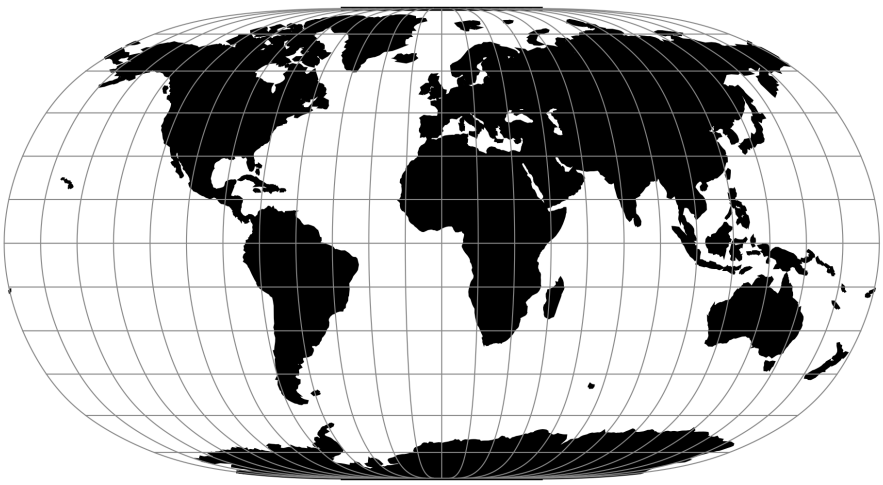


Fig. 85: proj-string: `+proj=natearth2`

#### 7.1.89.1 Parameters

> **Note**
>
> All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

`+y_0=<value>`

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.90 Nell

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | nell |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

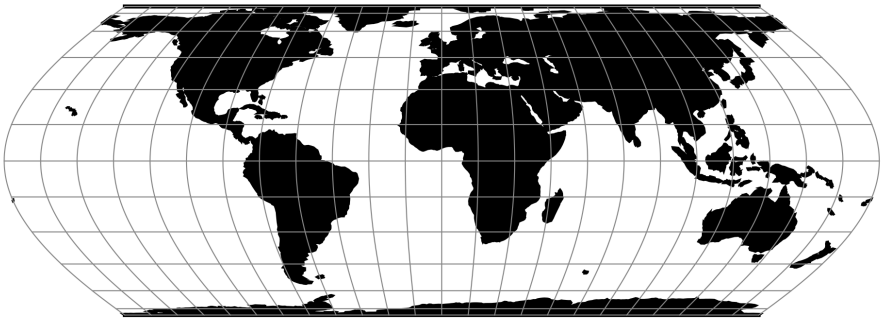


Fig. 86: proj-string: `+proj=nell`

#### 7.1.90.1 Parameters

> **Note**
>
> All parameters are optional.

`+lon_0=<value>`

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.91 Nell-Hammer

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | nell_h |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

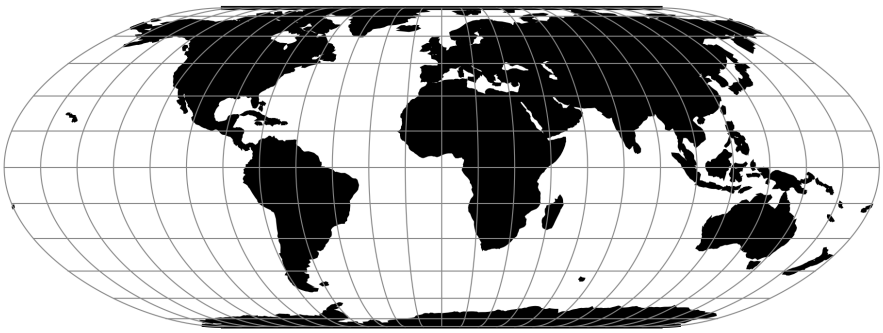


Fig. 87: proj-string: +proj=nell_h

#### 7.1.91.1 Parameters

**Note**

All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.92 Nicolosi Globular

| | |
|---|---|
| **Classification** | Pseudoconical |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | nicol |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.92.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Fig. 88: proj-string: `+proj=nicol`

> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.93 Near-sided perspective

The near-sided perspective projection simulates a view from a height $h$ similar to how a satellite in orbit would see it.

| | |
|---|---|
| **Classification** | Azimuthal. Neither conformal nor equal area. |
| **Available forms** | Forward and inverse spherical projection |
| **Defined area** | Global, although for one hemisphere at a time. |
| **Alias** | nsper |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 89: proj-string: `+proj=nsper +h=3000000 +lat_0=-20 +lon_0=145`

#### 7.1.93.1 Parameters

##### Required

**+h**=<value>

Height of the view point above the Earth and must be in the same units as the radius of the sphere or semimajor axis of the ellipsoid.

##### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.94 New Zealand Map Grid (EPSG:27200)

Fig. 90: proj-string: `+proj=nzmg`

#### 7.1.94.1 Parameters

**Note**

All standard projection parameters are hard-coded for this projection

### 7.1.95 General Oblique Transformation

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | ob_tran |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

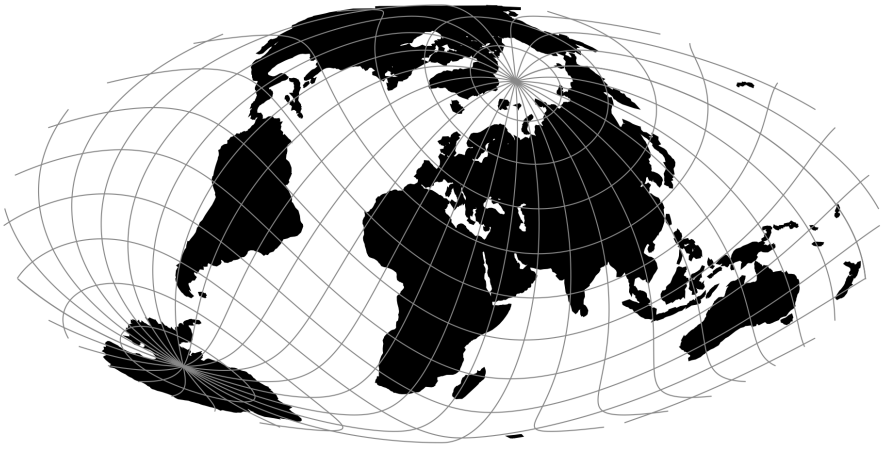


Fig. 91: proj-string: `+proj=ob_tran +o_proj=moll +o_lon_p=40 +o_lat_p=50 +lon_0=60`

#### 7.1.95.1 Usage

All of the projections of spherical library can be used as an oblique projection by means of the General Oblique Transformation. The user performs the oblique transformation by selecting the oblique projection `+proj=ob_tran`, specifying the translation factors, *+o_lat_p*, and *+o_lon_p*, and the projection to be used, *+o_proj*. In the example of the Fairgrieve projection, the latitude and longitude of the North pole of the unrotated geographic CRS, $\alpha$ and $\beta$ respectively, expressed in the rotated geographic CRS, are to be placed at 45°N and 90°W and the *Mollweide* projection is used. Because the central meridian of the translated coordinates will follow the $\beta$ meridian it is necessary to translate the translated system so that the Greenwich meridian will pass through the center of the projection by offsetting the central meridian.

The final control for this projection is:

```
+proj=ob_tran +o_proj=moll +o_lat_p=45 +o_lon_p=-90 +lon_0=-90
```

#### 7.1.95.2 Parameters

##### Required

**+o_proj**=<projection>
: Oblique projection.

In addition to specifying an oblique projection, *how* to rotate the projection should be specified. This is done in one of three ways: Define a new pole, rotate the projection about a given point or define a new "equator" spanned by two points on the sphere. See the details below.

### New pole

**+o_lat_p**=<latitude>

Latitude of the North pole of the unrotated source CRS, expressed in the rotated geographic CRS.

**+o_lon_p**=<longitude>

Longitude of the North pole of the unrotated source CRS, expressed in the rotated geographic CRS.

### Rotate about point

**+o_alpha**=<value>

Angle to rotate the projection with.

**+o_lon_c**=<value>

Longitude of the point the projection will be rotated about.

**+o_lat_c**=<value>

Latitude of the point the projection will be rotated about.

### New "equator" points

**+o_lon_1**=<value>

Longitude of first point.

**+o_lat_1**=<value>

Latitude of first point.

**+o_lon_2**=<value>

Longitude of second point.

**+o_lat_2**=<value>

Latitude of second point.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.96 Oblique Cylindrical Equal Area

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | ocea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

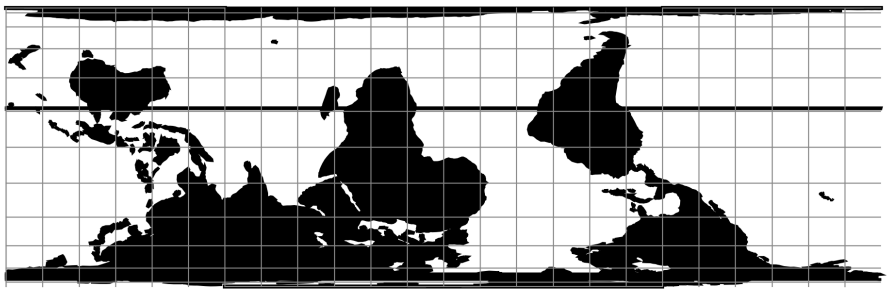


Fig. 92: proj-string: `+proj=ocea`

#### 7.1.96.1 Parameters

##### Required

For the Oblique Cylindrical Equal Area projection a pole of rotation is needed. The pole can be defined in two ways: By a point and azimuth or by providing two points that make up the pole.

##### Point & azimuth

**+lonc**=<value>

Longitude of rotational pole point.

**+alpha**=<value>

Angle of rotational pole.

### Two points

**+lon_1**=<value>

Longitude of first point.

**+lat_1**=<value>

Latitude of first point.

**+lon_2**=<value>

Longitude of second point.

**+lat_2**=<value>

Latitude of second point.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.97 Oblated Equal Area

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | oea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Described in [Snyder1988].

#### 7.1.97.1 Parameters

##### Required

**+m**=<value>

**+n**=<value>

##### Optional

**+theta**=<value>

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

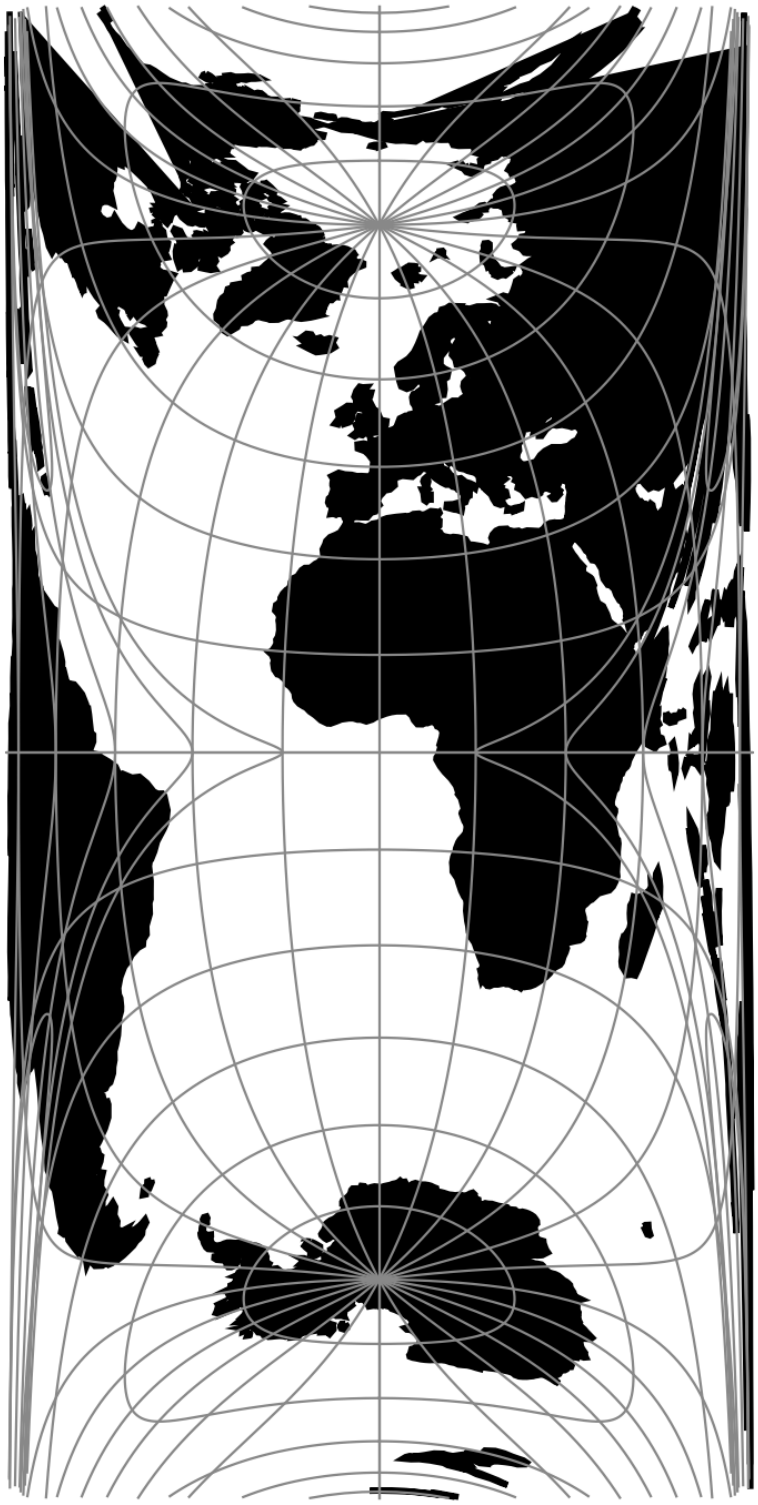


Fig. 93: proj-string: `+proj=oea +m=1 +n=2`

### 7.1.98 Oblique Mercator

The Oblique Mercator projection is a cylindrical map projection that closes the gap between the Mercator and the Transverse Mercator projections.

| | |
|---|---|
| **Classification** | Conformal cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global, but reasonably accurate only within 15 degrees of the oblique central line |
| **Alias** | omerc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

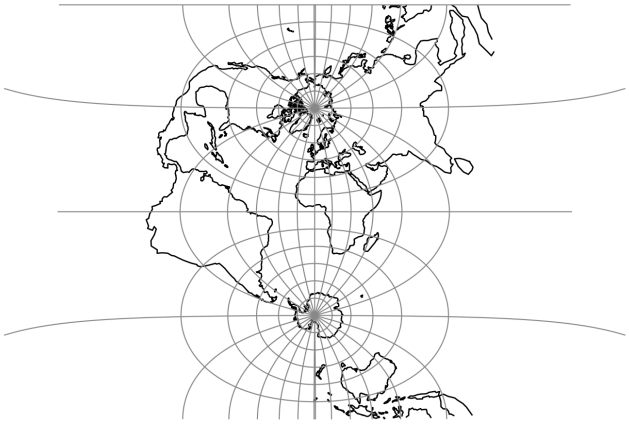


Fig. 94: proj-string: `+proj=omerc +lat_1=45 +lat_2=55`

Figuratively, the cylinder used for developing the Mercator projection touches the planet along the Equator, while that of the Transverse Mercator touches the planet along a meridian, i.e. along a line perpendicular to the Equator.

The cylinder for the Oblique Mercator, however, touches the planet along a line at an arbitrary angle with the Equator. Hence, the Oblique Mercator projection is useful for mapping areas having their greatest extent along a direction that is neither north-south, nor east-west.

The Mercator and the Transverse Mercator projections are both limiting forms of the Oblique Mercator: The Mercator projection is equivalent to an Oblique Mercator with central line along the Equator, while the Transverse Mercator is equivalent to an Oblique Mercator with central line along a meridian.

For the sphere, the construction of the Oblique Mercator projection can be imagined as "tilting the cylinder of a plain Mercator projection", so the cylinder, instead of touching the equator, touches an arbitrary great circle on the sphere. The great circle is defined by the tilt angle of the central line, hence putting land masses along that great circle near the centre of the map, where the Equator would go in the plain Mercator case.

The ellipsoidal case, developed by Hotine, and refined by Snyder [Snyder1987] is more complex, involving initial steps projecting from the ellipsoid to another curved surface, the "aposphere", then projection from the aposphere to the skew uv-plane, before finally rectifying the skew uv-plane onto the map XY plane.

**Note**

In certain special circumstances of the ellipsoidal form of the Oblique Mercator it is possible for two different input coordinates to project to the same output coordinate. This can happen when the input coordinates are far away from the projection center. A work around is to use the spherical version of the projection instead. See an example in the Usage section below.

#### 7.1.98.1 Usage

The tilt angle (azimuth) of the central line can be given in two different ways. In the first case, the azimuth is given directly, using the option *+alpha* and defining the centre of projection using the options *+lonc* and *+lat_0*. In the second case, the azimuth is given indirectly by specifying two points on the central line, using the options *+lat_1*, *+lon_1*, *+lat_2*, and *+lon_2*.

Example: Verify that the Mercator, and Transverse Mercator (on a sphere), projections are limiting forms of the Oblique Mercator

```
$ echo 12 55 | proj +proj=merc +ellps=GRS80
1335833.89 7326837.71

$ echo 12 55 | proj +proj=omerc +alpha=90 +ellps=GRS80
1335833.89 7326837.71

$ echo 12 55 | proj +proj=omerc +alpha=0 +R=6400000
766869.97 6209742.96

# Same, with azimuth given indirectly via two points:
$ echo 12 55 | proj +proj=omerc +lon_1=0 +lat_1=-1 +lon_2=0 +lat_2=0 +R=6400000
766869.97 6209742.96

$ echo 12 55 | proj +proj=tmerc +R=6400000
766869.97 6209742.96
```

Example: Second case - indirectly given azimuth

```
$ echo 12 55 | proj +proj=omerc +lon_1=-1 +lat_1=1 +lon_2=0 +lat_2=0 +ellps=GRS80
349567.57 6839490.50

# Same, with directly given azimuth, (via: echo 0 0 1 -1|geod -I -f %.7f +ellps=GRS80):
$ echo 12 55 | proj +proj=omerc +alpha=-45.1880402 +ellps=GRS80
349567.57 6839490.50
```

Example: An approximation of the Danish "System 34" from [Rittri2012]

```
$ echo 10.536498003 56.229892362 | cs2cs +proj=longlat +ellps=GRS80 +to +proj=omerc␣
↪+axis=wnu +lonc=9.46 +lat_0=56.13333333 +x_0=-266906.229 +y_0=189617.957 +k=0.9999537␣
↪+alpha=-0.76324 +gamma=0 +ellps=GRS80
200000.13   199999.89
```

The input coordinate represents the System 34 datum point "Agri Bavnehoj", with coordinates (200000, 200000) by definition. So at the datum point, the approximation is off by about 17 cm. This use case represents a datum shift from a cylinder projection on an old, slightly misaligned datum, to a similar projection on a modern datum.

Example: Non-unique results of the ellipsoidal form

```
proj +proj=omerc +lat_0=0.377041113875403 +lonc=33.8250934444081 +alpha=8.16321575614333␣
↪+gamma=0 +k=1 +x_0=0 +y_0=0 +ellps=WGS84 << EOF
-146.23 -78.19112222222222
-145.02309217 -78.19112222
EOF
875464.92       -11348616.86
875464.92       -11348616.86
```

Here we see that two coordinates with roughly a difference in latitude of about a degree results in the same projected coordinate. As mention above, a work around is to use the spherical form instead:

```
proj +proj=omerc +lat_0=0.377041113875403 +lonc=33.8250934444081 +alpha=8.16321575614333␣
↪+gamma=0 +k=1 +x_0=0 +y_0=0 +ellps=sphere << EOF
-146.23 -78.19112222222222
-145.02309217 -78.19112222
EOF
891266.38       -11376037.53
863563.62       -11374939.17
```

#### 7.1.98.2 Parameters

##### Central point and azimuth method

**+alpha**=<value>

Azimuth of centerline clockwise from north at the center point of the line. If *+gamma* is not given then *+alpha* determines the value of *+gamma*.

**+gamma**=<value>

Azimuth of centerline clockwise from north of the rectified bearing of centre line. If *+alpha* is not given, then *+gamma* is used to determine *+alpha*.

If specifying only *+gamma* without *+alpha*, the maximum value of the absolute value of *+gamma* is a function of the absolute value of *+lat_0*, equal to $90 - |\phi_0|$ on a sphere and slightly above on a non-spherical ellipsoid.

**+lonc**=<value>

Longitude of the projection centre. Note that this value is used to override the *+lon_0* parameter, so the latter should not be specified as it would get ignored.

**+lat_0**=<value>

Latitude of the projection centre.

##### Two point method

**+lon_1**=<value>

Longitude of first point.

**+lat_1**=<value>

Latitude of first point.

**+lon_2**=<value>

Longitude of second point.

**+lat_2**=<value>

Latitude of second point.

#### Optional

**+no_rot**

No rectification (not "no rotation" as one may well assume). Do not take the last step from the skew uv-plane to the map XY plane.

> **Note**
>
> This option is probably only marginally useful, but remains for (mostly) historical reasons.

**+no_off**

Do not offset origin to center of projection.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.98.3 Caveats

Note for the two-point method no rectification is done,

```
echo 0 0|proj -I +proj=omerc +R=6400000 +lonc=-87 +lat_0=42 +alpha=0
87dW    42dN
echo 0 0|proj -I +proj=omerc +R=6400000 +lonc=-87 +lat_0=42 +alpha=0 +no_rot
87dW    0dS
echo 0 0|proj -I +proj=omerc +R=6400000 +lon_1=-87 +lat_1=42 +lon_2=-87 +lat_2=43
87dW    0dN
```

Thus, just as was noted above regarding +no_rot, the two-point method itself is also probably only marginally useful.

### 7.1.99 Ortelius Oval

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | ortel |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

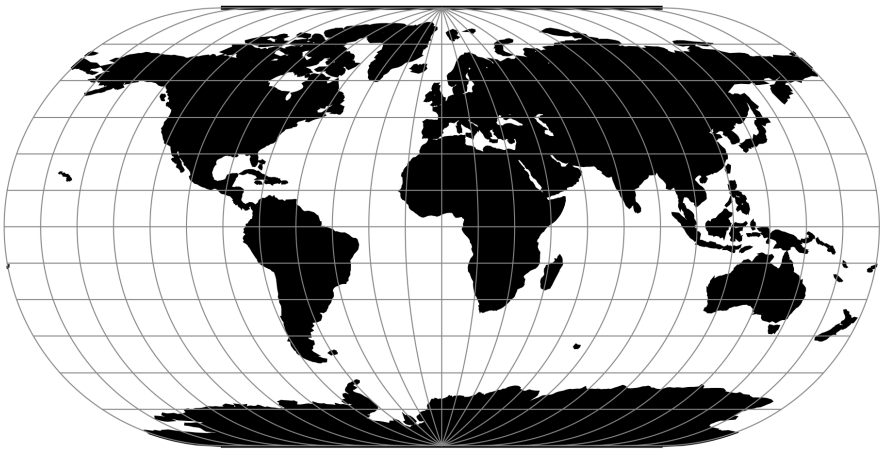


Fig. 95: proj-string: `+proj=ortel`

#### 7.1.99.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.100 Orthographic

The orthographic projection is a perspective azimuthal projection centered around a given latitude and longitude.

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global, although only one hemisphere can be seen at a time |
| **Alias** | ortho |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 96: proj-string: `+proj=ortho`

**Note**

Before PROJ 7.2, only the spherical formulation was implemented. If wanting to replicate PROJ < 7.2 results with newer versions, the ellipsoid must be forced to a sphere, for example by adding a `+f=0` parameter.

**Note**

Parameters `k_0` and `alpha` are added in PROJ 9.5.0

This projection method corresponds to `EPSG:9840` (or `EPSG:1130` with `k_0` or `alpha`).

#### 7.1.100.1 Parameters

**Note**

All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

**+alpha**=<value>

Added in version 9.5.0.

Azimuth clockwise from north at the center of projection.

*Defaults to 0.0.*

**+k_0**=<value>

Added in version 9.5.0: Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.101 Patterson

The Patterson projection is a cylindrical map projection designed for general-purpose mapmaking.

See [Patterson2014]

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | patterson |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.101.1 Parameters

**Note**

All parameters are optional for projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

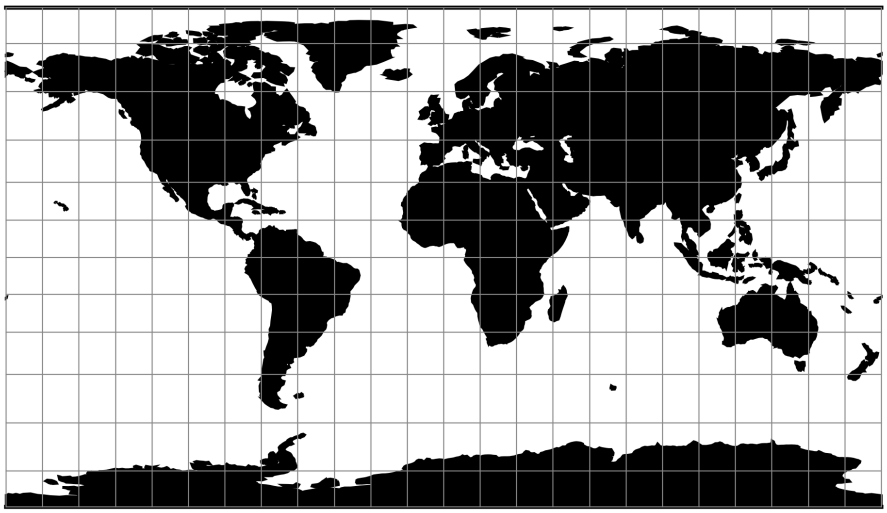


Fig. 97: proj-string: `+proj=patterson`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.102 Perspective Conic

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | pconic |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 98: proj-string: `+proj=pconic +lat_1=25 +lat_2=75`

#### 7.1.102.1 Parameters

**Required**

`+lat_1=<value>`

First standard parallel.

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_1=1.570796r*

See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_2=1.570796r*

See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.103 Peirce Quincuncial

The Peirce Quincuncial projection is a conformal map projection that transforms the circle of the northern hemisphere into a square, and the southern hemisphere split into four triangles arranged around the square to form a quincunx. The resulting projection is a regular diamond shape or can be rotated to form a square. The resulting tile can be infinitely tessellated. Though this implementation defaults to a central meridian of 0, it is more common to use a central meridian of around 25 to optimise the distortions. Peirce's original published map from 1879 used a central meridian of approx -70. The diamond and square versions can be produced using the `+shape=diamond` and `+shape=square` options respectively. This implementation includes an alternative lateral projection which places hemispheres side-by-side (`+shape=horizontal` or `+shape=vertical`). Combined with a general oblique transformation, this can be used to produced a Grieger Triptychial projection (see example below).

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | peirce_q |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.103.1 Parameters

**Note**

All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+shape**=square/diamond/horizontal/vertical/nhemisphere/shemisphere

Added in version 9.0.

*Defaults to diamond.*

**Warning**

This option was wrongly introduced in 8.2.1 with the `type` name, which was inappropriate as it conflicted with the `+type=crs` general hint.

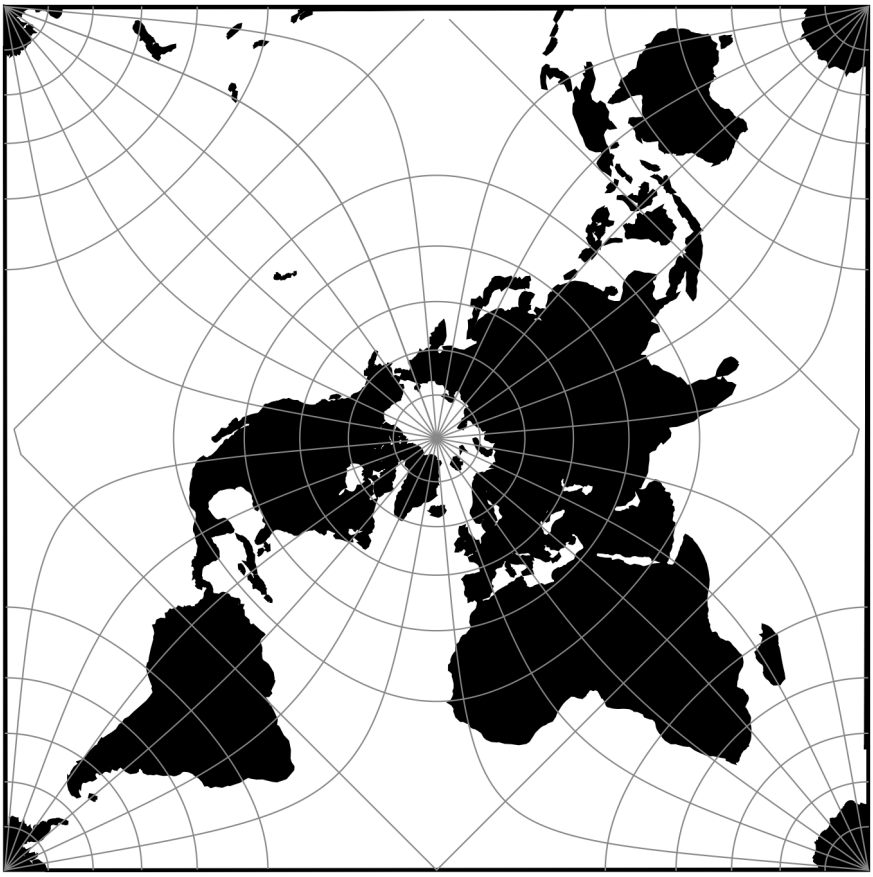


Fig. 99: proj-string: `+proj=peirce_q +lon_0=25 +shape=square`

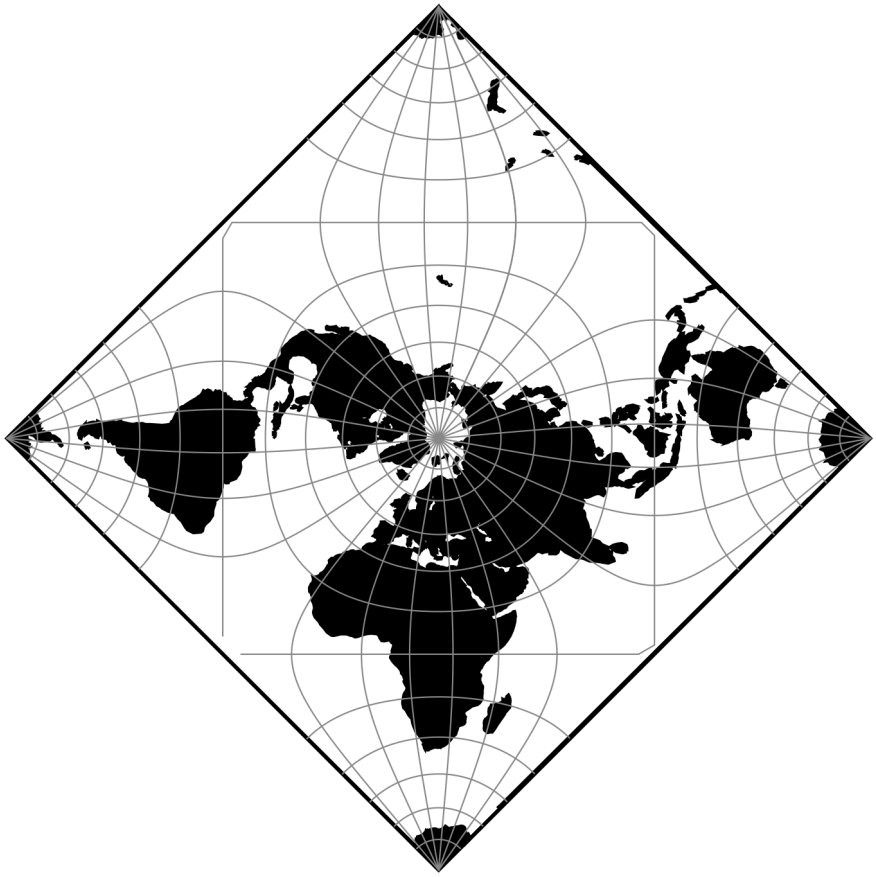


Fig. 100: proj-string: `+proj=peirce_q +lon_0=25 +shape=diamond`

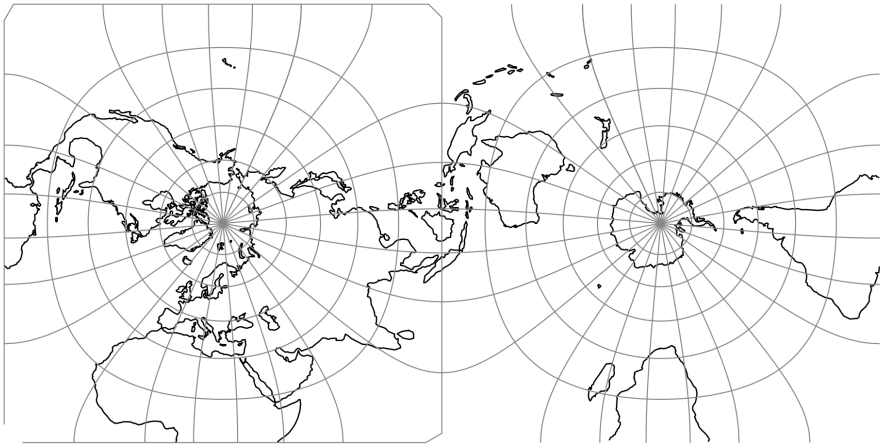


Fig. 101: proj-string: `+proj=peirce_q +lon_0=25 +shape=horizontal`

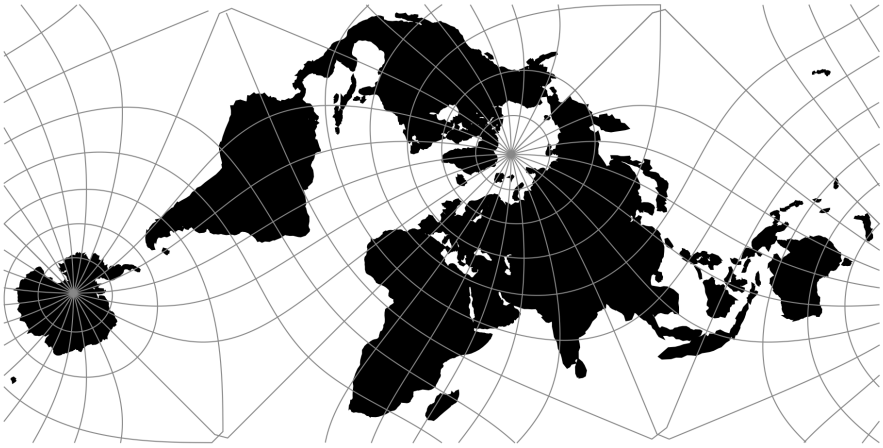


Fig. 102: proj-string: `+proj=pipeline +step +proj=ob_tran +o_proj=peirce_q +o_lat_p=-45 +o_lon_p=45 +o_type=horizontal +o_scrollx=-0.25 +step +proj=affine +s11=-1 +s12=0 +s21=0 +s22=-1`

Indicates the shape of transformation applied to the southern hemisphere: `square` and `diamond` represent the traditional quincuncial form suggested by Peirce with the southern hemisphere divided into 4 triangles and reflected outward from the northern hemisphere. The `square` shape is rotated by 45 degrees to produce the conventional square presentation. The origin lies at the centre of the square or diamond.

By contrast, the `horizontal` and `vertical` forms reflect the southern hemisphere laterally across the x or y axis respectively to produce a rectangular form. The origin lies at the centre of the rectangle.

The other two types, `nhemisphere` and `shemisphere`, discard latitudes of less than 0 or more than 0, respectively, to allow single hemispheres to be selected. The origin lies at the centre of the square or diamond.

**+scrollx**=<value>

For `horizontal` shape allows a scalar circular scroll of resulting x coordinates to shift sections of the projection to the other horizontal side of the map.

*Defaults to 0.0. Must be a scale between -1.0 and 1.0.*

**+scrolly**=<value>

For `vertical` shape allows a scalar circular scroll of resulting y coordinates to shift sections of the projection to the other vertical side of the map.

*Defaults to 0.0. Must be a scale between -1.0 and 1.0.*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.104 Polyconic (American)

| | |
|---|---|
| **Classification** | Pseudoconical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | poly |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 103: proj-string: `+proj=poly`

#### 7.1.104.1 Parameters

> **Note**
>
> All parameters are optional for projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.105 Putnins P1

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp1 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.105.1 Parameters

> **Note**
>
> All parameters are optional for the Putnins P1 projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

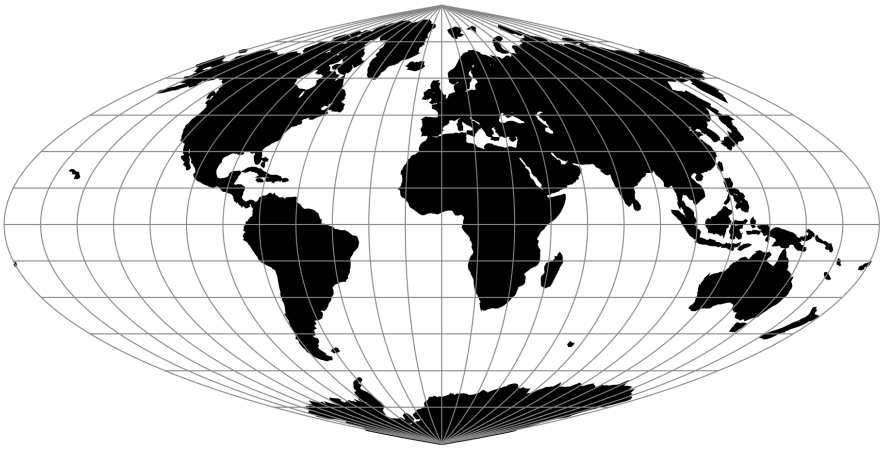


Fig. 104: proj-string: `+proj=putp1`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.106 Putnins P2

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

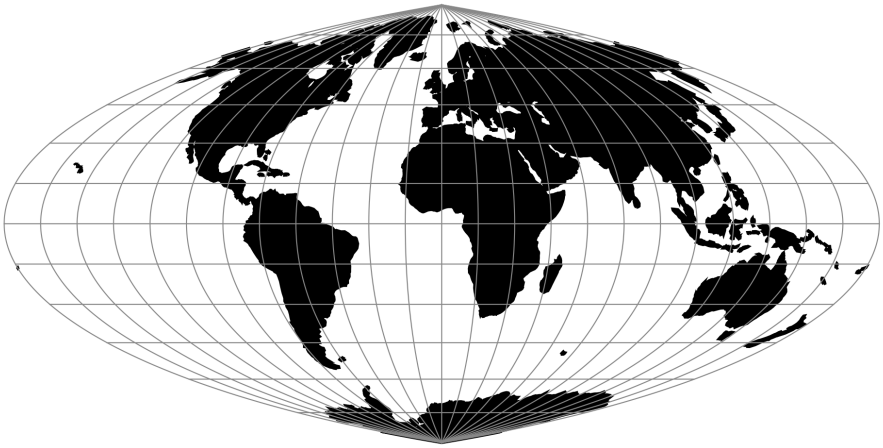


Fig. 105: proj-string: `+proj=putp2`

#### 7.1.106.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.107 Putnins P3

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp3 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

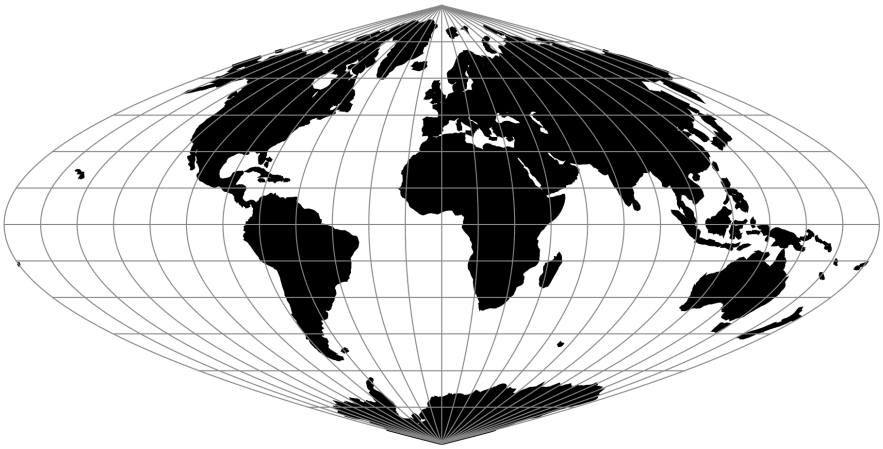


Fig. 106: proj-string: `+proj=putp3`

#### 7.1.107.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.108 Putnins P3'

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp3p |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

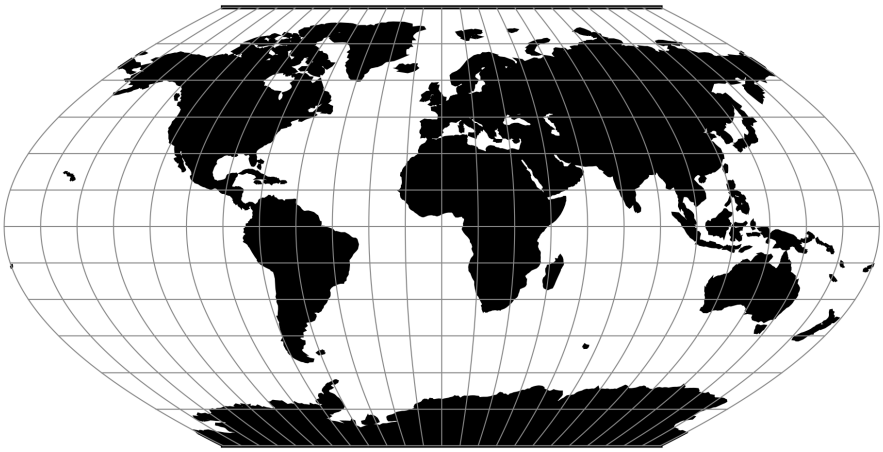


Fig. 107: proj-string: +proj=putp3p

#### 7.1.108.1 Parameters

**Note**

All parameters are optional for the projection.

`+lon_0`=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

`+R`=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

`+x_0`=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

`+y_0`=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.109 Putnins P4'

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp4p |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.109.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

`+lon_0`=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

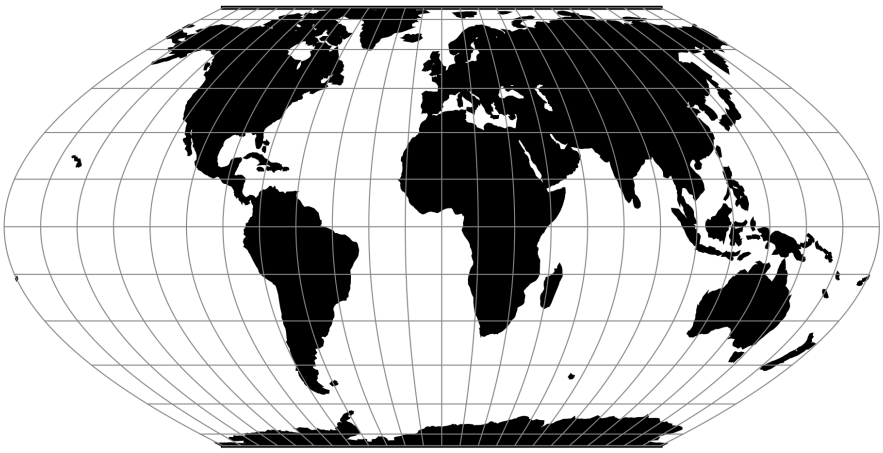


Fig. 108: proj-string: `+proj=putp4p`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.110 Putnins P5

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp5 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

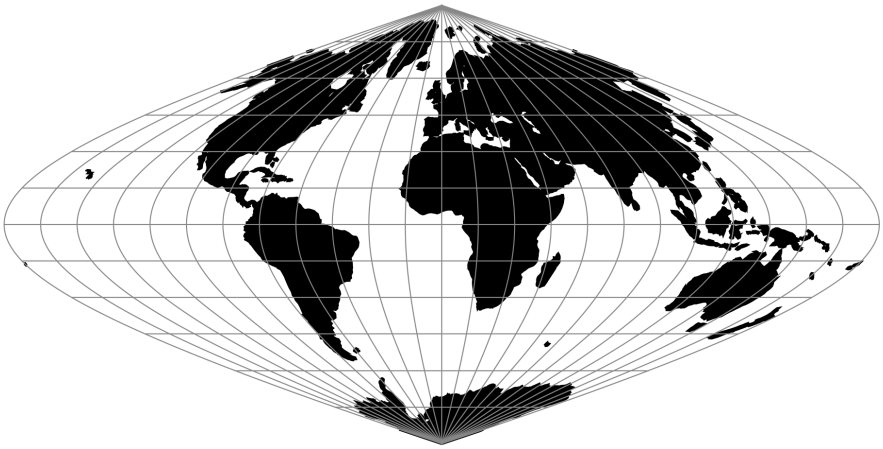


Fig. 109: proj-string: `+proj=putp5`

#### 7.1.110.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.111 Putnins P5'

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp5p |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

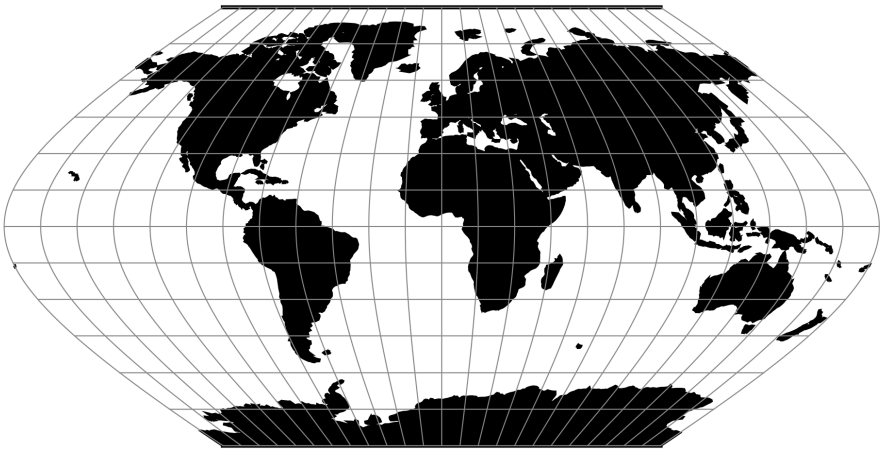


Fig. 110: proj-string: `+proj=putp5p`

#### 7.1.111.1 Parameters

**Note**

All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.112 Putnins P6

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp6 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

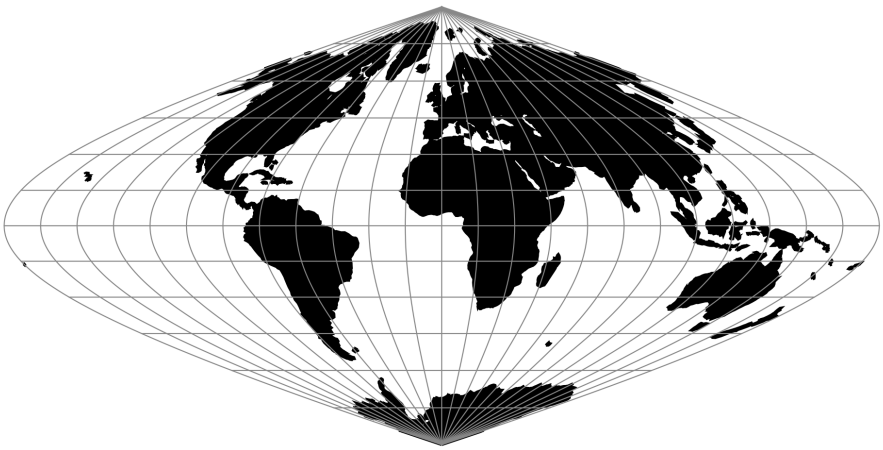


Fig. 111: proj-string: +proj=putp6

#### 7.1.112.1 Parameters

**Note**

All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.113 Putnins P6'

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | putp6p |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.113.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

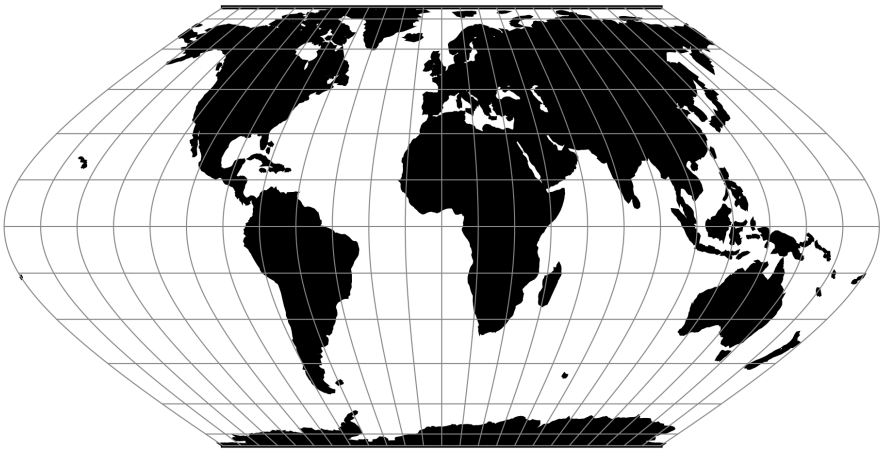


Fig. 112: proj-string: `+proj=putp6p`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.114 Quartic Authalic

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | qua_aut |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

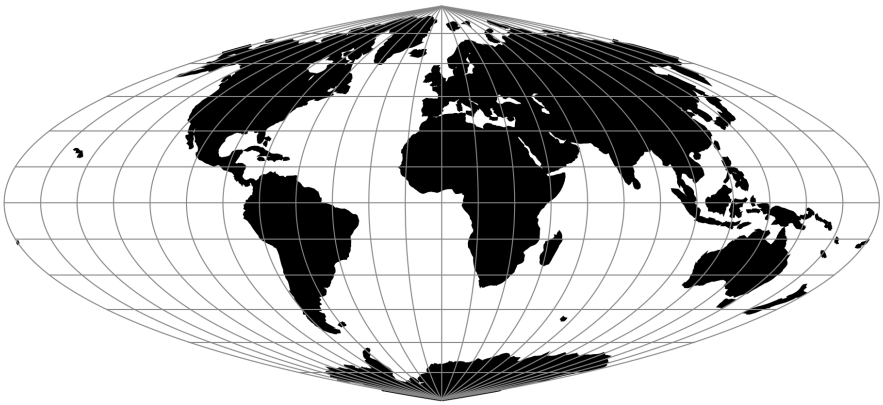


Fig. 113: proj-string: `+proj=qua_aut`

#### 7.1.114.1 Parameters

> **Note**
>
> All parameters are optional for the Quartic Authalic projection.

**+lon_0**=<value>
: Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

  *Defaults to 0.0.*

  > **Note**
  >
  > The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
  >
  > Example: *+lon_0=1.570796r*
  >
  > See *Projection Units* for more information.

**+R**=<value>
: Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

  See *Ellipsoid size parameters* for more information.

**+x_0**=<value>
: False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

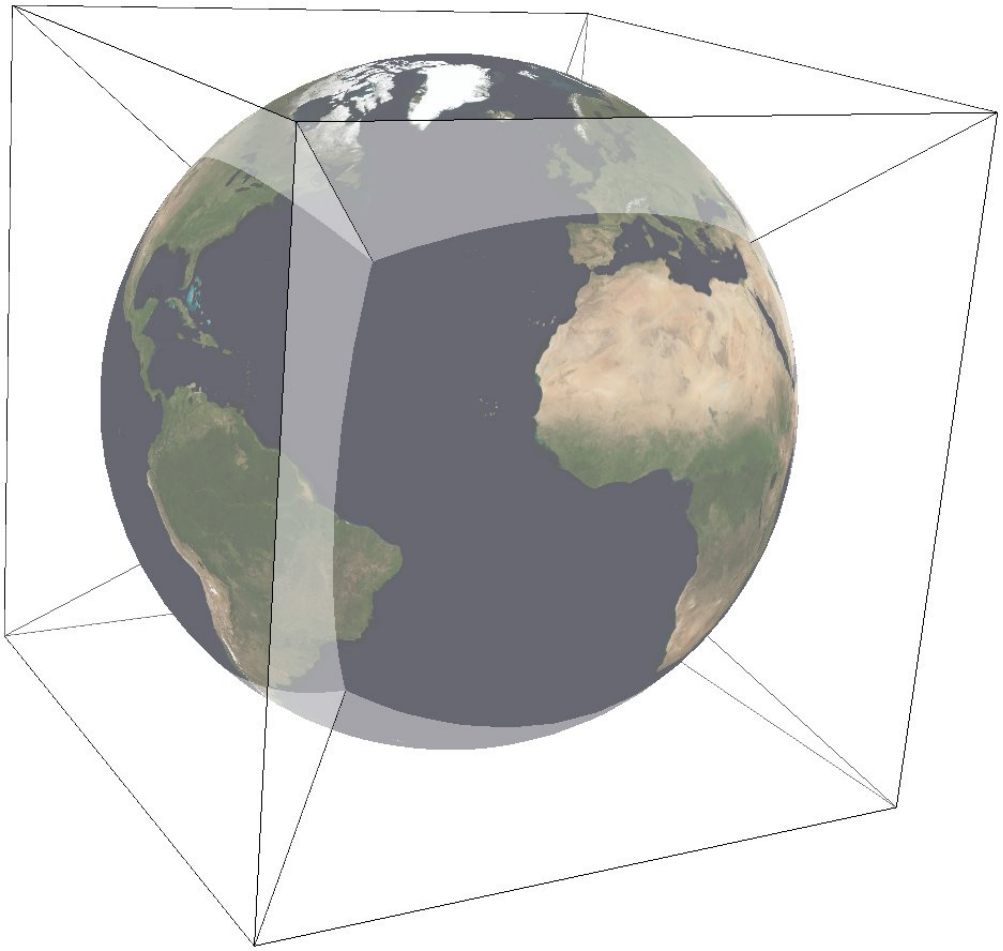
**+y_0**=<value>
: False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

### 7.1.115 Quadrilateralized Spherical Cube

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, ellipsoidal |
| **Defined area** | Global |
| **Alias** | qsc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

The purpose of the Quadrilateralized Spherical Cube (QSC) projection is to project a sphere surface onto the six sides of a cube:



For this purpose, other alternatives can be used, notably *Gnomonic* or *HEALPix*. However, QSC projection has the following favorable properties:

It is an equal-area projection, and at the same time introduces only limited angular distortions. It treats all cube sides equally, i.e. it does not use different projections for polar areas and equatorial areas. These properties make QSC projection a good choice for planetary-scale terrain rendering. Map data can be organized in quadtree structures for each cube side. See [LambersKolb2012] for an example.

The QSC projection was introduced by [ONeilLaubscher1976], building on previous work by [ChanONeil1975]. For clarity: The earlier QSC variant described in [ChanONeil1975] became known as the COBE QSC since it was used by the NASA Cosmic Background Explorer (COBE) project; it is an approximately equal-area projection and is not the

same as the QSC projection.

See also [CalabrettaGreisen2002] Sec. 5.6.2 and 5.6.3 for a description of both and some analysis.

In this implementation, the QSC projection projects onto one side of a circumscribed cube. The cube side is selected by choosing one of the following six projection centers:

| | |
|---|---|
| `+lat_0=0 +lon_0=0` | front cube side |
| `+lat_0=0 +lon_0=90` | right cube side |
| `+lat_0=0 +lon_0=180` | back cube side |
| `+lat_0=0 +lon_0=-90` | left cube side |
| `+lat_0=90` | top cube side |
| `+lat_0=-90` | bottom cube side |

Furthermore, this implementation allows the projection to be applied to ellipsoids. A preceding shift to a sphere is performed automatically; see [LambersKolb2012] for details.

#### 7.1.115.1 Usage

The following example uses QSC projection via GDAL to create the six cube side maps from a world map for the WGS84 ellipsoid:

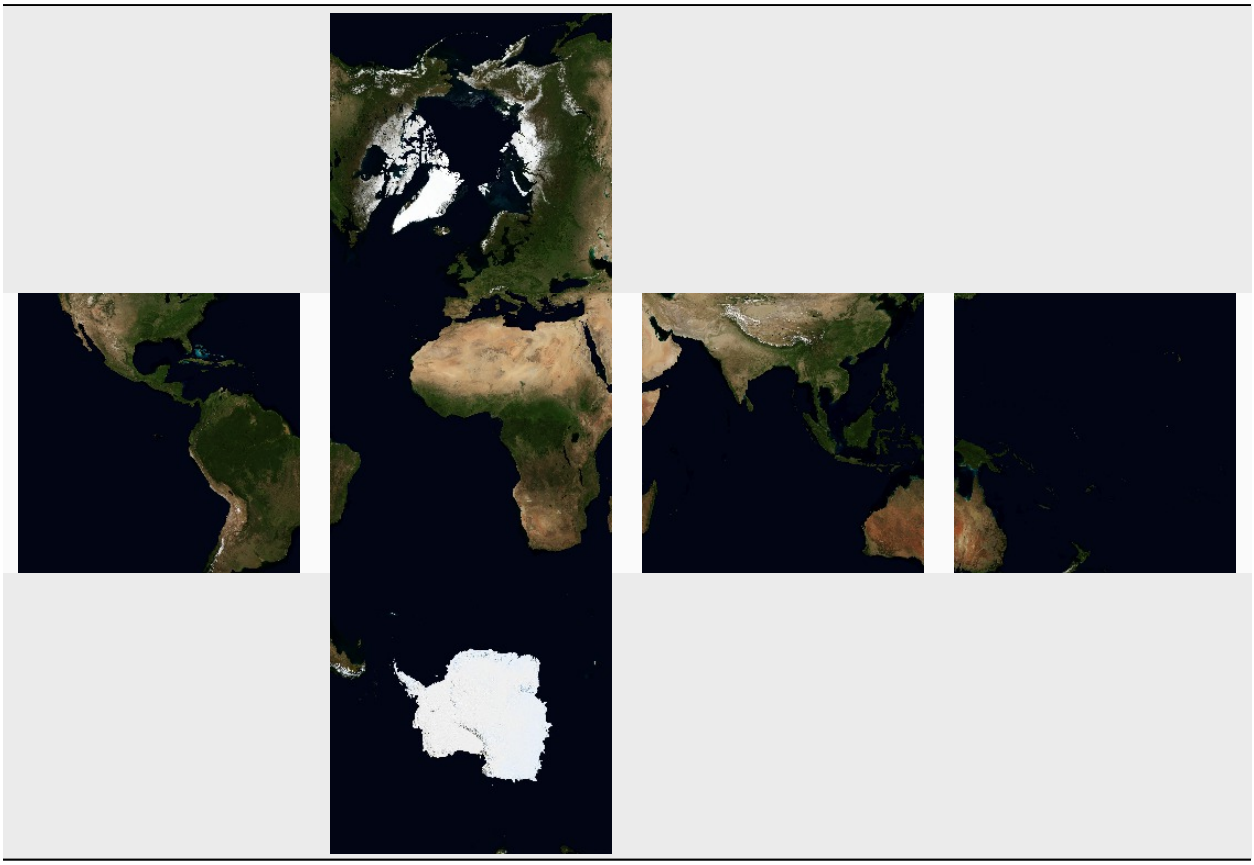
```
gdalwarp -t_srs "+wktext +proj=qsc +units=m +ellps=WGS84  +lat_0=0 +lon_0=0"        \
    -wo SOURCE_EXTRA=100 -wo SAMPLE_GRID=YES -te -6378137 -6378137 6378137 6378137  \
    worldmap.tiff frontside.tiff

gdalwarp -t_srs "+wktext +proj=qsc +units=m +ellps=WGS84 +lat_0=0 +lon_0=90"        \
    -wo SOURCE_EXTRA=100 -wo SAMPLE_GRID=YES -te -6378137 -6378137 6378137 6378137  \
    worldmap.tiff rightside.tiff

gdalwarp -t_srs "+wktext +proj=qsc +units=m +ellps=WGS84 +lat_0=0 +lon_0=180"       \
    -wo SOURCE_EXTRA=100 -wo SAMPLE_GRID=YES -te -6378137 -6378137 6378137 6378137  \
    worldmap.tiff backside.tiff

gdalwarp -t_srs "+wktext +proj=qsc +units=m +ellps=WGS84 +lat_0=0 +lon_0=-90"       \
    -wo SOURCE_EXTRA=100 -wo SAMPLE_GRID=YES -te -6378137 -6378137 6378137 6378137  \
    worldmap.tiff leftside.tiff

gdalwarp -t_srs "+wktext +proj=qsc +units=m +ellps=WGS84 +lat_0=90 +lon_0=0"        \
    -wo SOURCE_EXTRA=100 -wo SAMPLE_GRID=YES -te -6378137 -6378137 6378137 6378137  \
    worldmap.tiff topside.tiff

gdalwarp -t_srs "+wktext +proj=qsc +units=m +ellps=WGS84 +lat_0=-90 +lon_0=0"       \
    -wo SOURCE_EXTRA=100 -wo SAMPLE_GRID=YES -te -6378137 -6378137 6378137 6378137  \
    worldmap.tiff bottomside.tiff
```

Explanation:

- QSC projection is selected with `+wktext +proj=qsc`.
- The WGS84 ellipsoid is specified with `+ellps=WGS84`.
- The cube side is selected with `+lat_0=... +lon_0=...`.
- The `-wo` options are necessary for GDAL to avoid holes in the output maps.

- The `-te` option limits the extends of the output map to the major axis diameter (from -radius to +radius in both x and y direction). These are the dimensions of one side of the circumscribing cube.

The resulting images can be laid out in a grid like below.



### 7.1.115.2 Parameters

> **Note**
>
> All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.115.3 Further reading

1. Wikipedia
2. NASA

### 7.1.116 Robinson

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | robin |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.116.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

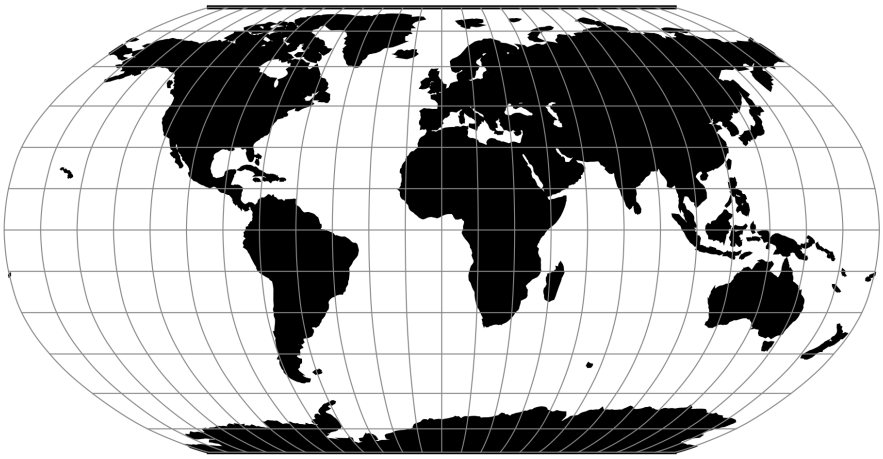


Fig. 114: proj-string: `+proj=robin`

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.117 Roussilhe Stereographic

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | rouss |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

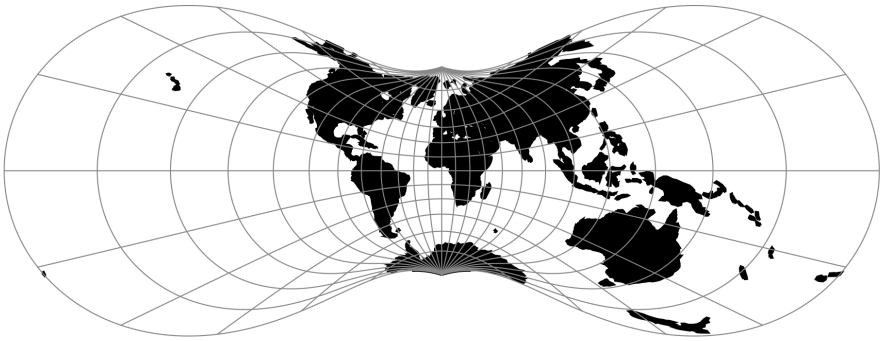


Fig. 115: proj-string: `+proj=rouss`

#### 7.1.117.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>
: Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

  *Defaults to 0.0.*

  > **Note**
  >
  > The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
  >
  > Example: *+lon_0=1.570796r*
  >
  > See *Projection Units* for more information.

**+ellps**=<value>
: The name of a built-in ellipsoid definition.

  See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

  *Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.118 Rectangular Polyconic

| | |
|---|---|
| **Classification** | Pseudoconical |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | rpoly |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

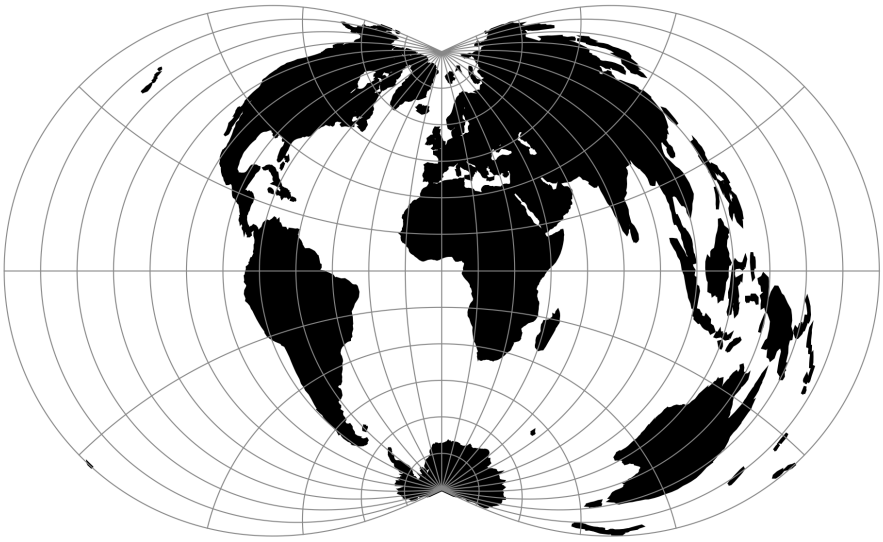


Fig. 116: proj-string: +proj=rpoly

#### 7.1.118.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lat_ts**=<value>

Latitude of true scale. Defines the latitude where scale is not distorted. Takes precedence over +k_0 if both options are used together.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_ts=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

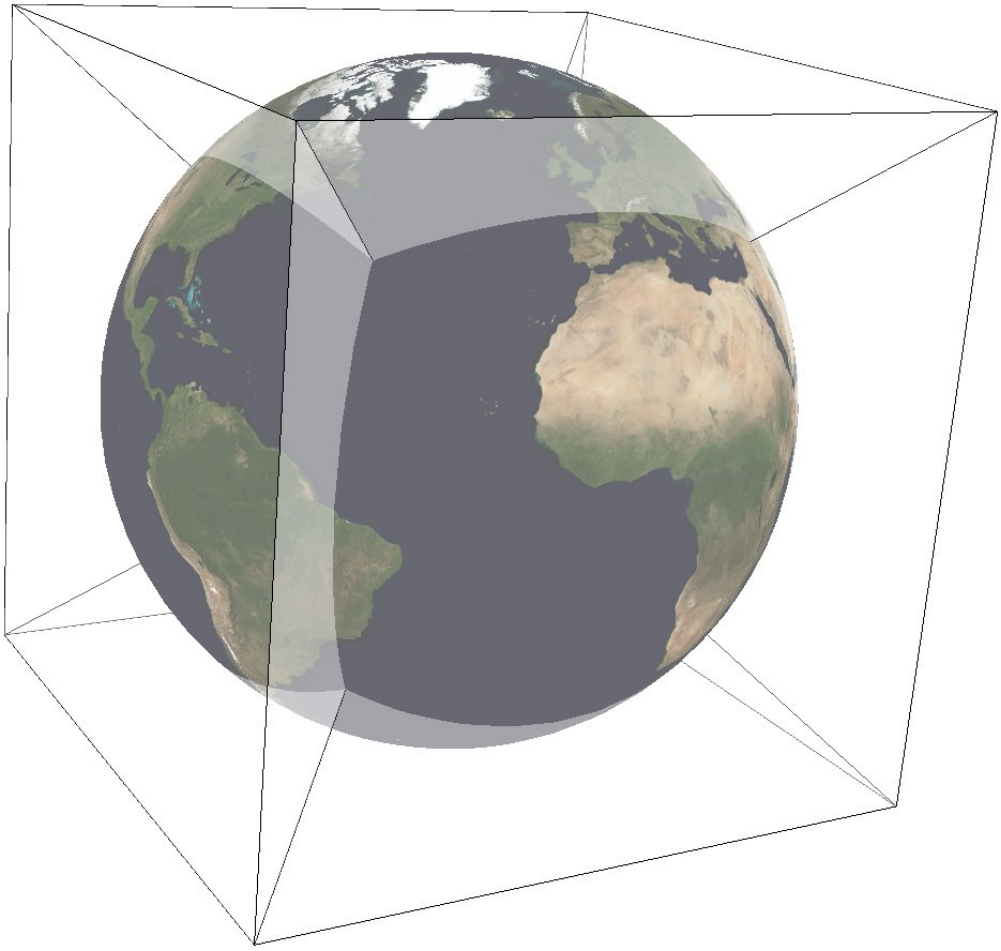
*Defaults to 0.0.*

### 7.1.119 S2

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, ellipsoidal |
| **Defined area** | Global |
| **Alias** | s2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Added in version 8.2.

The S2 projection, like the Quadrilateralized Spherical Cube (QSC) projection, projects a sphere surface onto the six sides of a cube:



S2 was created by Google to represent geographic data on the whole earth. The documentation can be found at S2 Geometry It works by first projecting a point on the sphere to a face of the cube. These are called u,v-coordinates, and they are in [-1,1] x [-1,1]. This step is followed by a non-linear transformation to normalize the area of rectangles on the sphere. There are three different choices available for this transformation, meaning that S2 is a family of projections. The final output is in s,t-coordinates, which are in [0,1] x [0,1]. See the comments in S2 Code for an explanation of the tradeoff between speed and area-preservation. Note that the projection is azimuthal when "none" or "linear" is selected for the area-normalization, but it is not azimuthal when "quadratic" or "tangent" is chosen. See S2's Earthcube page to visualize the unfolded cube and the orientation of each face.

In this implementation, the cube side is selected by choosing one of the following six projection centers:

| | |
|---|---|
| `+lat_0=0 +lon_0=0` | front cube side |
| `+lat_0=0 +lon_0=90` | right cube side |
| `+lat_0=0 +lon_0=180` | back cube side |
| `+lat_0=0 +lon_0=-90` | left cube side |
| `+lat_0=90` | top cube side |
| `+lat_0=-90` | bottom cube side |

The specific transformation can be chosen with the UVtoST parameter:

| | |
|---|---|
| `+UVtoST=linear` | fastest, no normalization |
| `+UVtoST=quadratic` | fast, good normalization |
| `+UVtoST=tangent` | slowest, best normalization |
| `+UVtoST=none` | returns u,v-coordinates |

Furthermore, this implementation allows the projection to be applied to ellipsoids. A preceding shift to a sphere is performed automatically; see [LambersKolb2012] for details. The output of the projection is in s,t-coordinates ([0,1] x [0,1]), so only the eccentricity of the ellipse is taken into account: the absolute value of the axes does not affect the output.

#### 7.1.119.1 Usage

The following example uses S2 on the right face:

```
echo 90 0 | proj +proj=s2 +lat_0=0 +lon_0=90 +ellps=WGS84 +UVtoST=linear

0.5 0.5
```

Explanation:

- S2 projection is selected with `+proj=s2`.
- The WGS84 ellipsoid is specified with `+ellps=WGS84`.
- The cube side is selected with `+lat_0=... +lon_0=...`.
- The normalization transformation is selected with `+UVtoST=...`.

#### 7.1.119.2 Parameters

**Note**

All parameters for the projection are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+UVtoST**=<value>

The area-normalization transformation. Choose from {linear, quadratic, tangent, none}

*Defaults to "quadratic".*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.119.3 Further reading

1. S2's Website

### 7.1.120 Spherical Cross-track Height

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse. |
| **Defined area** | Global |
| **Alias** | sch |
| **Domain** | 3D |
| **Input type** | 3D coordinates |
| **Output type** | Projected coordinates |

proj-string: `+proj=sch +plat_0=XX +plon_0=XX +phdg_0=XX`

The SCH coordinate system is a sensor aligned coordinate system developed at JPL (Jet Propulsion Laboratory) for radar mapping missions.

See [Hensley2002]

#### 7.1.120.1 Parameters

##### Required

**+plat_0**=<value>
: Peg latitude (in degree)

**+plon_0**=<value>
: Peg longitude (in degree)

**+phdg_0**=<value>
: Peg heading (in degree)

##### Optional

**+h_0**=<value>
: Average height (in metre)

*Defaults to 0.0.*

**+lon_0**=<value>
: Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+x_0**=<value>
: False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

### 7.1.121 Sinusoidal (Sanson-Flamsteed)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | sinu |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

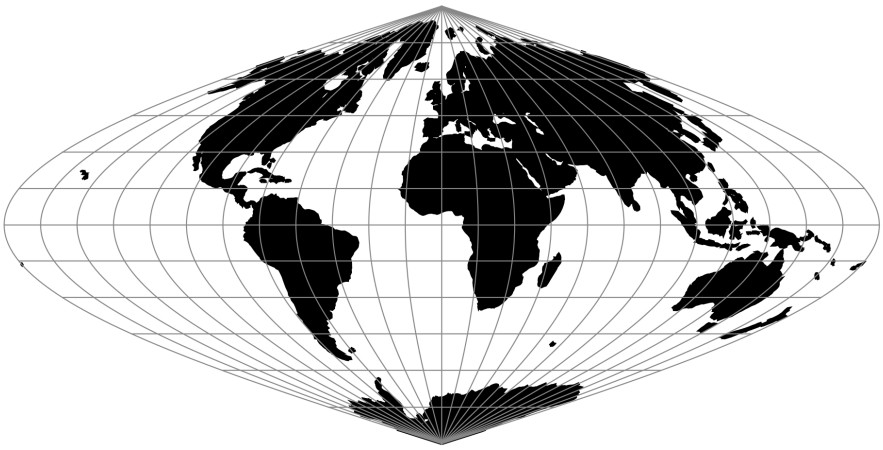


Fig. 117: proj-string: `+proj=sinu`

MacBryde and Thomas developed generalized formulas for several of the pseudocylindricals with sinusoidal meridians:

$$x = C\lambda(m + cos\theta)/(m + 1)$$

$$y = C\theta$$

$$C = \sqrt{(m + 1)/n}$$

#### 7.1.121.1 Parameters

> **Note**
>
> All parameters are optional for the Sinusoidal projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.122 Space Oblique Mercator (SOM)

The Space Oblique Mercator (SOM) projection is a generalization of the Oblique Mercator projection, itself an (oblique) aspect of the Mercator projection with origins in the 16th century. In the previously available map projections, ground tracks of satellites traced curved lines which complicated their use in representing satellite remote sensing data. In response, SOM was specifically designed to represent satellite ground tracks as straight lines. This had the effect of minimizing distortions of data along a ground track and allowing satellite data to be represented accurately as a rectangular array.

| | |
|---|---|
| **Classification** | Conformal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | som |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

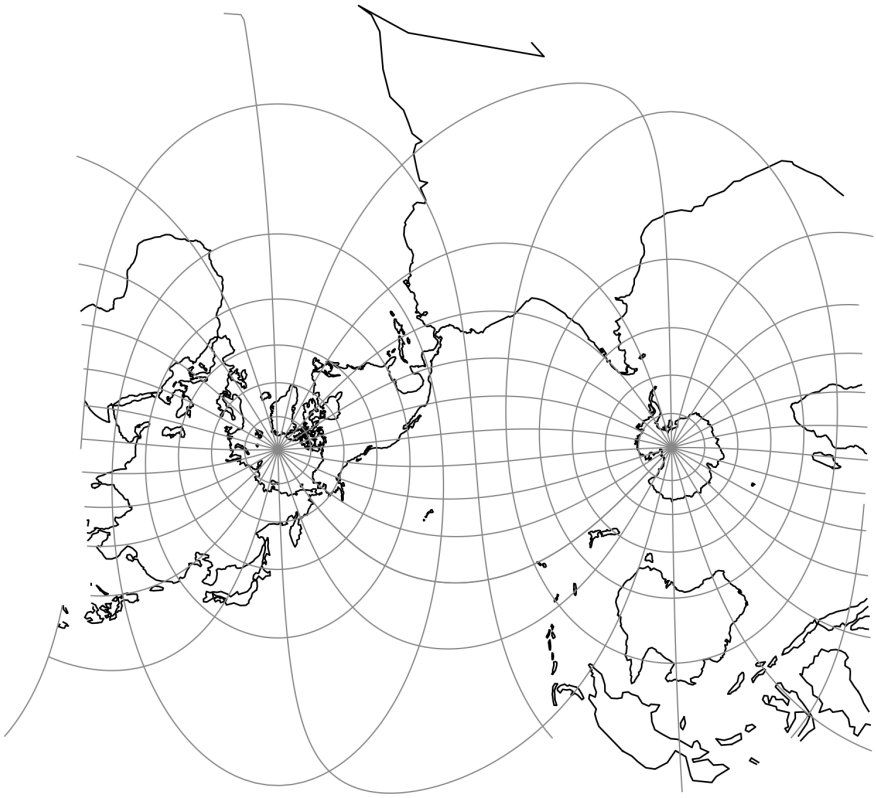


Fig. 118: proj-string: `+proj=som +inc_angle=98.303820000243860022 +ps_rev=0.06866666666666667 +asc_lon=64.412896137498847793`
(gives same results as: `+proj=som +inc_angle=1.7157253262878522r +ps_rev=0.06866666666666667 +asc_lon=1.1242171183417042r`)

#### 7.1.122.1 Parameters

##### Required

**+asc_lon**=<value>

Longitude of the ascending node for the orbit (decimal degrees).

**+inc_angle**=<value>

Inclination (vertical tilt) of the orbit with respect to the reference plane, measured at the ascending node (decimal degrees).

**+ps_rev**=<value>

Time required for a single orbit (days).

> **Note**
>
> To specify asc_lon or inc_angle in radians (instead of degrees). Follow the value with the "r" character.
>
> As an example, to specify the ascending longitude as 1.1242171183417042 radians (~64.4129 decimal degrees): *+asc_lon=1.1242171183417042r*.
>
> For more information see *projections usage*.

##### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.122.2 Further reading

1. Space Oblique Mercator Wikipedia
2. USGS Report on Space Oblique Mercator Mathematical Development

### 7.1.123 Swiss Oblique Mercator

Fig. 119: proj-string: `+proj=somerc`

#### 7.1.123.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.124 Spilhaus

Added in version 9.6.0.

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical, ellipsoidal |
| **Defined area** | Global |
| **Alias** | spilhaus |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

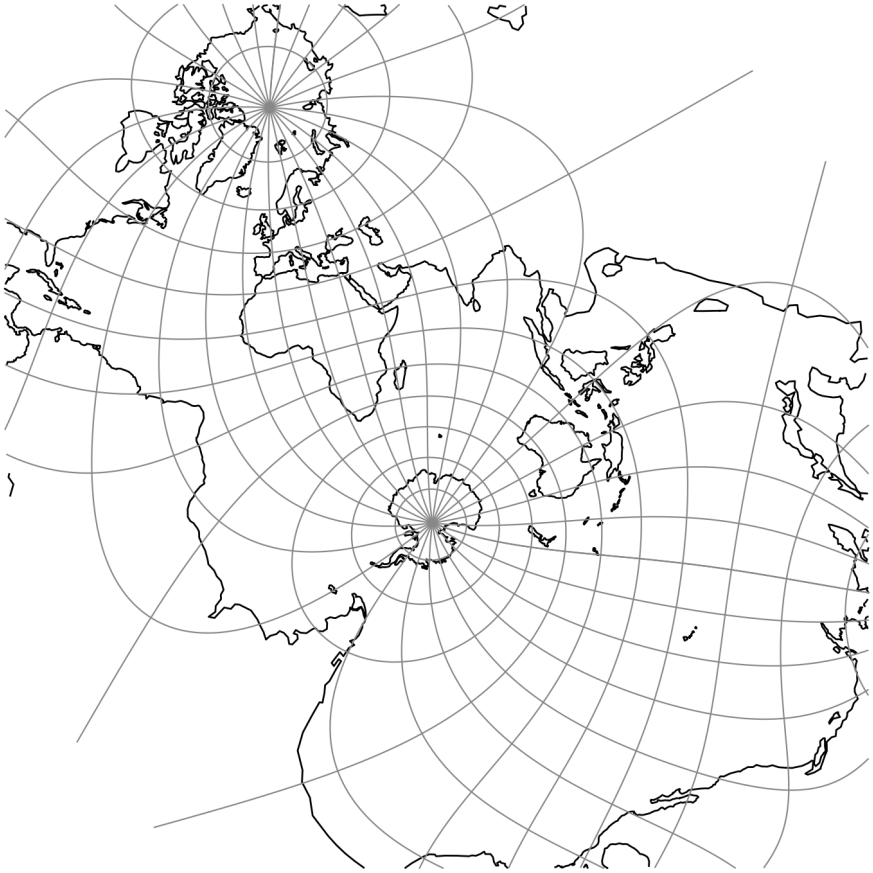


Fig. 120: proj-string: +proj=spilhaus

#### 7.1.124.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Longitude of projection centre.

*Defaults to 66.94970198.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of projection centre.

*Defaults to -49.56371678.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+azi**=<value>

Azimuth from north at the center of projection.

*Defaults to 40.17823482.*

**+rot**=<value>

Rotation of the projection.

*Defaults to 45.*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection. To reproduce the output of ESRI:54099, use +k_0=1.41421356237 (sqrt(2))

*Defaults to 1.0.*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.125 Stereographic

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | stere |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

**Note**

This projection method gives different results than the *Oblique Stereographic Alternative* method in the non-polar cases (i.e. the oblique and equatorial case). The later projection method is the one referenced by EPSG as "Oblique Stereographic".

#### 7.1.125.1 Parameters

**Note**

All parameters are optional for the projection.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_0=1.570796r*

See *Projection Units* for more information.

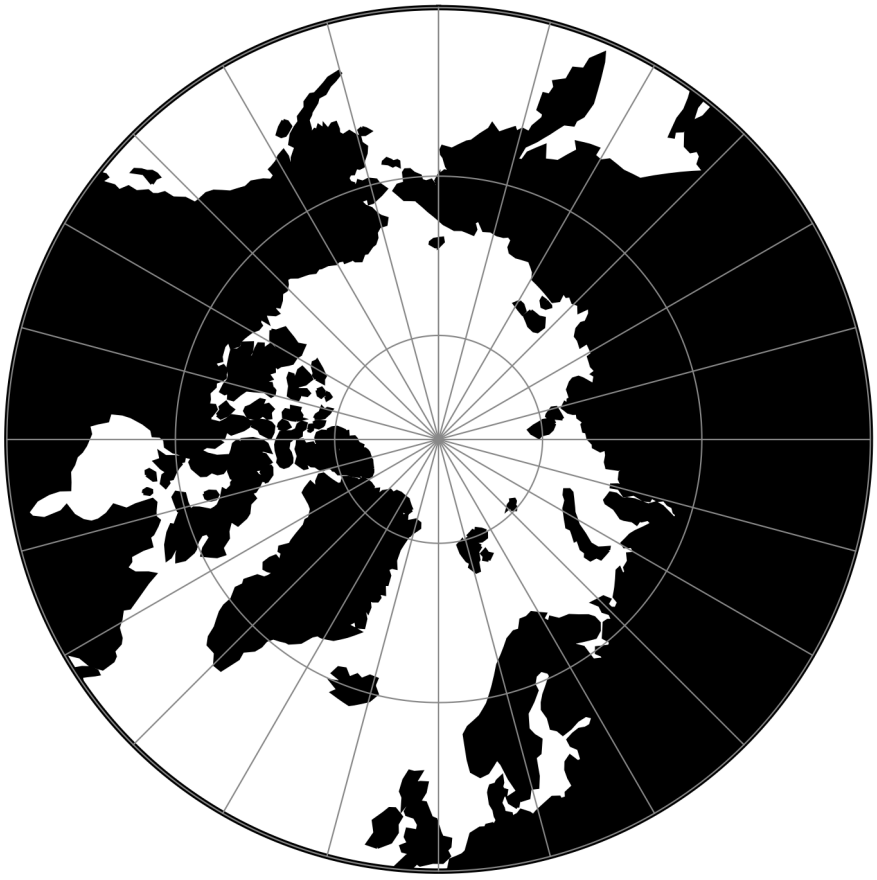


Fig. 121: proj-string: +proj=stere +lat_0=90 +lat_ts=75

**+lat_ts**=<value>

Defines the latitude where scale is not distorted. It is only taken into account for Polar Stereographic formulations (+lat_0 = +/- 90 ), and then defaults to the +lat_0 value. If set to a value different from +/- 90, it takes precedence over +k_0 if both options are used together.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.126 Oblique Stereographic Alternative

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global |
| **Alias** | sterea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

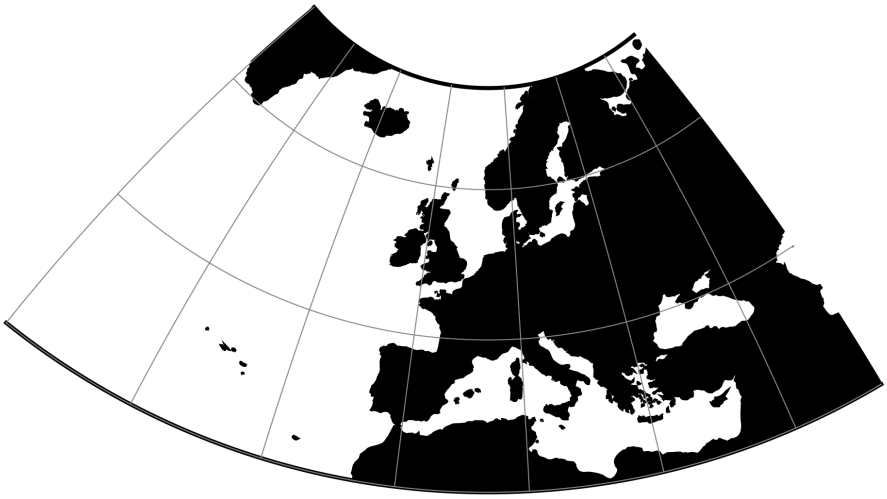


Fig. 122: proj-string: `+proj=sterea +lat_0=52.1561605555556 +lon_0=5.38763888888889 +k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel`

**Note**

This projection method, referenced by EPSG as "Oblique Stereographic", is for example used for the Netherlands "Amersfoort / RD New" projected CRS. It gives different results than the *Stereographic* method in the non-polar cases (i.e. the oblique and equatorial case).

#### 7.1.126.1 Parameters

**Note**

All parameters are optional for the projection.

`+lon_0=<value>`

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

`+lat_0=<value>`

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend

on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

### 7.1.127 Gauss-Schreiber Transverse Mercator (aka Gauss-Laborde Reunion)

| | |
|---|---|
| **Classification** | Conformal |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | gstmerc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

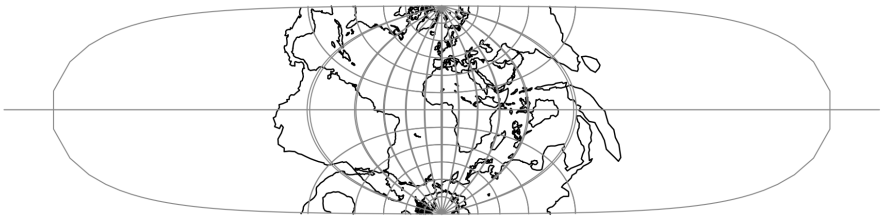


Fig. 123: proj-string: `+proj=gstmerc`

#### 7.1.127.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.128 Transverse Central Cylindrical

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | tcc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.128.1 Parameters

**Note**

All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

Fig. 124: proj-string: `+proj=tcc`

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.129 Transverse Cylindrical Equal Area

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | tcea |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

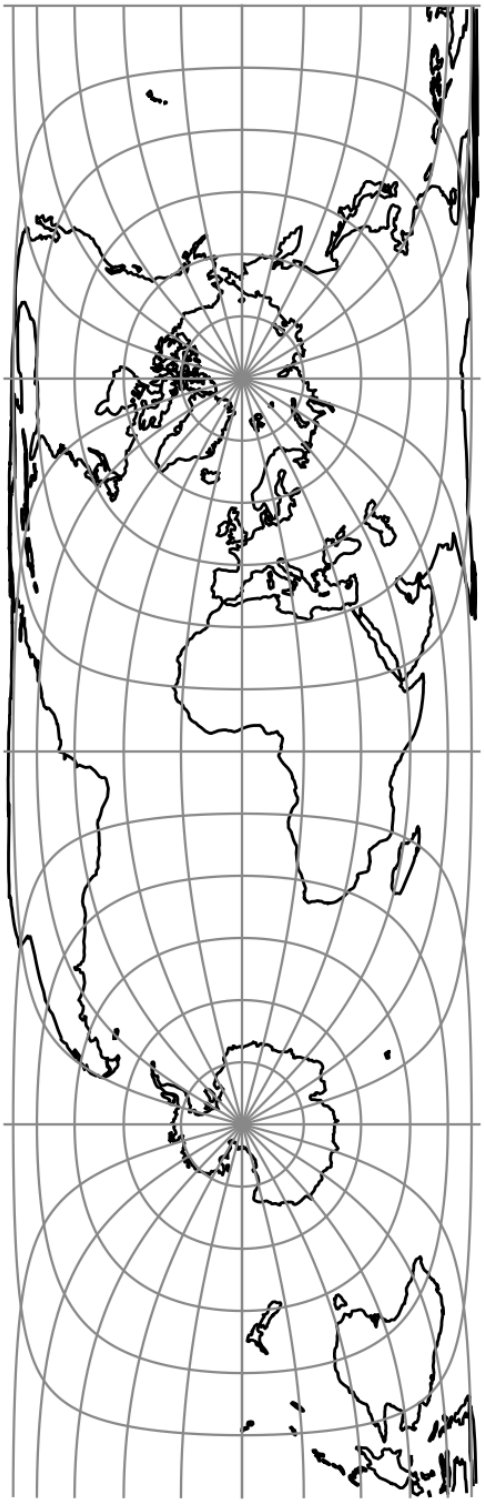


Fig. 125: proj-string: `+proj=tcea`

#### 7.1.129.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.130 Times

See [Snyder1993], p.213-214.

| | |
|---|---|
| **Classification** | Cylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | times |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

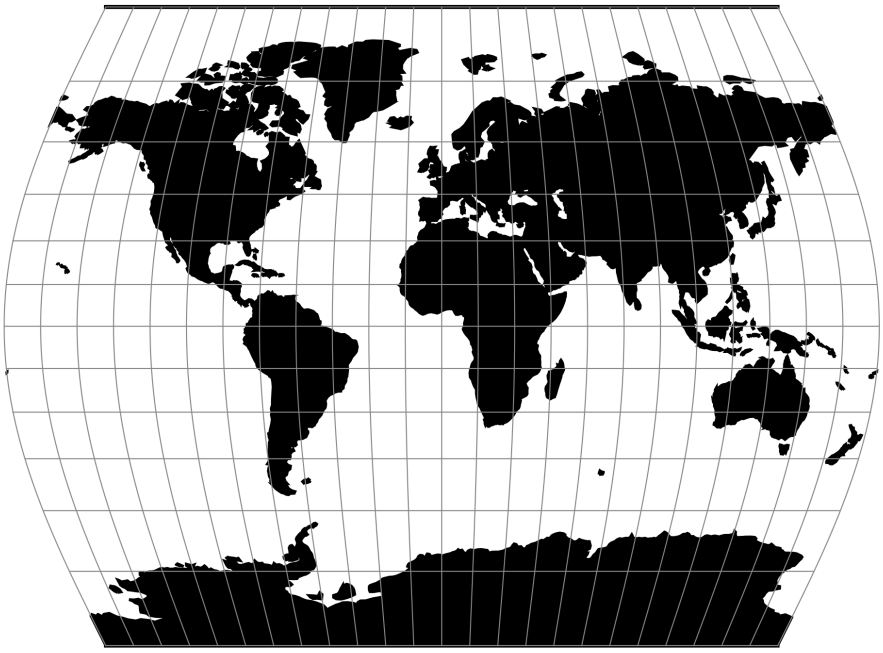


Fig. 126: proj-string: `+proj=times`

#### 7.1.130.1 Parameters

> **Note**
>
> All parameters are optional for projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.131 Tissot

Fig. 127: proj-string: `+proj=tissot +lat_1=60 +lat_2=65`

#### 7.1.131.1 Parameters

##### Required

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_2=1.570796r*
>
> See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.132 Transverse Mercator

The transverse Mercator projection in its various forms is the most widely used projected coordinate system for world topographical and offshore mapping. It is a conformal projection in which a chosen meridian projects to a straight line at constant scale.

| | |
|---|---|
| **Classification** | Transverse and oblique cylindrical |
| **Available forms** | Forward and inverse, spherical and ellipsoidal |
| **Defined area** | Global, with full accuracy within 3900 km of the central meridian |
| **Alias** | tmerc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

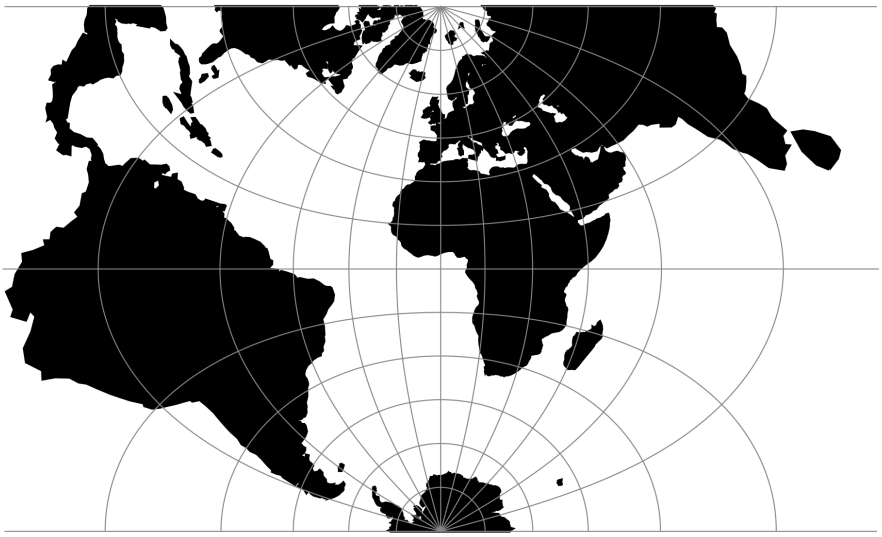


Fig. 128: proj-string: `+proj=tmerc`

#### 7.1.132.1 Usage

Prior to the development of the Universal Transverse Mercator coordinate system, several European nations demonstrated the utility of grid-based conformal maps by mapping their territory during the interwar period. Calculating the distance between two points on these maps could be performed more easily in the field (using the Pythagorean theorem) than was possible using the trigonometric formulas required under the graticule-based system of latitude and longitude. In the post-war years, these concepts were extended into the Universal Transverse Mercator/Universal Polar Stereographic (UTM/UPS) coordinate system, which is a global (or universal) system of grid-based maps.

The following table gives special cases of the Transverse Mercator projection.

| Projection Name | Areas | Central meridian | Zone width | Scale Factor |
|---|---|---|---|---|
| Transverse Mercator | World wide | Various | less than 1000 km | Various |
| Transverse Mercator south oriented | Southern Africa | 2° intervals E of 11°E | 2° | 1.000 |
| UTM North hemisphere | World wide equator to 84°N | 6° intervals E & W of 3° E & W | Usually 6°, wider for Norway and Svalbard | 0.9996 |
| UTM South hemisphere | World wide north of 80°S to equator | 6° intervals E & W of 3° E & W | Always 6° | 0.9996 |
| Gauss-Kruger | Former USSR, Yugoslavia, Germany, S. America, China | Various, according to area | Usually less than 6°, often less than 4° | 1.0000 |
| Gauss Boaga | Italy | Various, according to area | 6° | 0.9996 |

Example using Gauss-Kruger on Germany area (aka EPSG:31467)

```
$ echo 9 51 | proj +proj=tmerc +lon_0=9 +x_0=3500000 +ellps=bessel
3500000.00  5651505.56
```

Example using Gauss Boaga on Italy area (EPSG:3004)

```
$ echo 15 42 | proj +proj=tmerc +lon_0=15 +k_0=0.9996 +x_0=2520000 +ellps=intl
2520000.00  4649858.60
```

#### 7.1.132.2 Parameters

**Note**

All parameters for the projection are optional.

**+approx**

Added in version 6.0.0.

Use the Evenden-Snyder algorithm described below under "Legacy ellipsoidal form". It is faster than the default algorithm, but is less accurate and diverges beyond 3° from the central meridian.

**+algo**=auto/evenden_snyder/poder_engsager

Added in version 7.1.

Selects the algorithm to use. The hardcoded value and the one defined in *proj.ini* default to `poder_engsager`; that is the most precise one.

When using auto, a heuristics based on the input coordinate to transform is used to determine if the faster Evenden-Snyder method can be used, for faster computation, without causing an error greater than 0.1 mm (for an ellipsoid of the size of Earth)

Note that *+approx* and *+algo* are mutually exclusive.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.132.3 Mathematical definition

The formulation given here for the Transverse Mercator projection is due to Krüger [Krueger1912] who gave the series expansions accurate to $n^4$, where $n = (a-b)/(a+b)$ is the third flattening. These series were extended to sixth order by Engsager and Poder in [Poder1998] and [Engsager2007]. This gives full double-precision accuracy within 3900 km

of the central meridian (about 57% of the surface of the earth) [Karney2011tm]. The error is less than 0.1 mm within 7000 km of the central meridian (about 89% of the surface of the earth).

This formulation consists of three steps: a conformal projection from the ellipsoid to a sphere, the spherical transverse Mercator projection, rectifying this projection to give constant scale on the central meridian.

The scale on the central meridian is $k_0$ and is set by `+k_0`.

Option `+lon_0` sets the central meridian; in the formulation below $\lambda$ is the longitude relative to the central meridian.

Options `+lat_0`, `+x_0`, and `+y_0` serve to translate the projected coordinates so that at $(\phi, \lambda) = (\phi_0, \lambda_0)$, the projected coordinates are $(x, y) = (x_0, y_0)$. To simplify the formulas below, these options are set to zero (their default values).

Because the projection is conformal, the formulation is most conveniently given in terms of complex numbers. In particular, the unscaled projected coordinates $\eta$ (proportional to the easting, $x$) and $\xi$ (proportional to the northing, $y$) are combined into the single complex quantity $\zeta = \xi + i\eta$, where $i = \sqrt{-1}$. Then any analytic function $f(\zeta)$ defines a conformal mapping (this follows from the Cauchy-Riemann conditions).

### Spherical form

Because the full (ellipsoidal) projection includes the spherical projection as one of the components, we present the spherical form first with the coordinates tagged with primes, $\phi'$, $\lambda'$, $\zeta' = \xi' + i\eta'$, $x'$, $y'$, so that they can be distinguished from the corresponding ellipsoidal coordinates (without the primes). The projected coordinates for the sphere are given by

$$x' = k_0 R\eta'; \qquad y' = k_0 R\xi'$$

### Forward projection

$$\xi' = \tan^{-1}\left(\frac{\tan\phi'}{\cos\lambda'}\right)$$

$$\eta' = \sinh^{-1}\left(\frac{\sin\lambda'}{\sqrt{\tan^2\phi' + \cos^2\lambda'}}\right)$$

### Inverse projection

$$\phi' = \tan^{-1}\left(\frac{\sin\xi'}{\sqrt{\sinh^2\eta' + \cos^2\xi'}}\right)$$

$$\lambda' = \tan^{-1}\left(\frac{\sinh\eta'}{\cos\xi'}\right)$$

### Ellipsoidal form

The projected coordinates are given by

$$\zeta = \xi + i\eta; \qquad x = k_0 A\eta; \qquad y = k_0 A\xi$$

$$A = \frac{a}{1+n}\left(1 + \frac{1}{4}n^2 + \frac{1}{64}n^4 + \frac{1}{256}n^6\right)$$

The series for conversion between ellipsoidal and spherical geographic coordinates and ellipsoidal and spherical projected coordinates are given in matrix notation where $\mathbf{S}(\theta)$ and $\mathbf{N}$ are the row and column vectors of length 6

$$\mathbf{S}(\theta) = \begin{pmatrix}\sin 2\theta & \sin 4\theta & \sin 6\theta & \sin 8\theta & \sin 10\theta & \sin 12\theta\end{pmatrix}$$

$$\mathbf{N} = \begin{pmatrix} n \\ n^2 \\ n^3 \\ n^4 \\ n^5 \\ n^6 \end{pmatrix}$$

and $\mathsf{C}_{\alpha,\beta}$ are upper triangular $6 \times 6$ matrices.

### Relation between geographic coordinates

$$\lambda' = \lambda$$

$$\phi' = \tan^{-1}\sinh\left(\sinh^{-1}\tan\phi - e\tanh^{-1}(e\sin\phi)\right)$$

Instead of using this analytical formula for $\phi'$, the conversions between $\phi$ and $\phi'$ use the series approximations:

$$\phi' = \phi + \mathbf{S}(\phi) \cdot \mathsf{C}_{\chi,\phi} \cdot \mathbf{N}$$

$$\phi = \phi' + \mathbf{S}(\phi') \cdot \mathsf{C}_{\phi,\chi} \cdot \mathbf{N}$$

$$\mathsf{C}_{\chi,\phi} = \begin{pmatrix} -2 & \frac{2}{3} & \frac{4}{3} & -\frac{82}{45} & \frac{32}{45} & \frac{4642}{4725} \\ & \frac{5}{3} & -\frac{16}{15} & -\frac{13}{9} & \frac{904}{315} & -\frac{1522}{945} \\ & & -\frac{26}{15} & \frac{34}{21} & \frac{8}{5} & -\frac{12686}{2835} \\ & & & \frac{1237}{630} & -\frac{12}{5} & -\frac{24832}{14175} \\ & & & & -\frac{734}{315} & \frac{109598}{31185} \\ & & & & & \frac{444337}{155925} \end{pmatrix}$$

$$\mathsf{C}_{\phi,\chi} = \begin{pmatrix} 2 & -\frac{2}{3} & -2 & \frac{116}{45} & \frac{26}{45} & -\frac{2854}{675} \\ & \frac{7}{3} & -\frac{8}{5} & -\frac{227}{45} & \frac{2704}{315} & \frac{2323}{945} \\ & & \frac{56}{15} & -\frac{136}{35} & -\frac{1262}{105} & \frac{73814}{2835} \\ & & & \frac{4279}{630} & -\frac{332}{35} & -\frac{399572}{14175} \\ & & & & \frac{4174}{315} & -\frac{144838}{6237} \\ & & & & & \frac{601676}{22275} \end{pmatrix}$$

Here $\phi'$ is the conformal latitude (sometimes denoted by $\chi$) and $\mathsf{C}_{\chi,\phi}$ and $\mathsf{C}_{\phi,\chi}$ are the coefficients in the trigonometric series for converting between $\phi$ and $\chi$.

### Relation between projected coordinates

$$\zeta = \zeta' + \mathbf{S}(\zeta') \cdot \mathsf{C}_{\mu,\chi} \cdot \mathbf{N}$$

$$\zeta' = \zeta + \mathbf{S}(\zeta) \cdot \mathsf{C}_{\chi,\mu} \cdot \mathbf{N}$$

$$\mathsf{C}_{\mu,\chi} = \begin{pmatrix} \frac{1}{2} & -\frac{2}{3} & \frac{5}{16} & \frac{41}{180} & -\frac{127}{288} & \frac{7891}{37800} \\ & \frac{13}{48} & -\frac{3}{5} & \frac{557}{1440} & \frac{281}{630} & -\frac{1983433}{1935360} \\ & & \frac{61}{240} & -\frac{103}{140} & \frac{15061}{26880} & \frac{167603}{181440} \\ & & & \frac{49561}{161280} & -\frac{179}{168} & \frac{6601661}{7257600} \\ & & & & \frac{34729}{80640} & -\frac{3418889}{1995840} \\ & & & & & \frac{212378941}{319334400} \end{pmatrix}$$

$$\mathsf{C}_{\chi,\mu} = \begin{pmatrix} -\frac{1}{2} & \frac{2}{3} & -\frac{37}{96} & \frac{1}{360} & \frac{81}{512} & -\frac{96199}{604800} \\ & -\frac{1}{48} & -\frac{1}{15} & \frac{437}{1440} & -\frac{46}{105} & \frac{1118711}{3870720} \\ & & -\frac{17}{480} & \frac{37}{840} & \frac{209}{4480} & -\frac{5569}{90720} \\ & & & -\frac{4397}{161280} & \frac{11}{504} & \frac{830251}{7257600} \\ & & & & -\frac{4583}{161280} & \frac{108847}{3991680} \\ & & & & & -\frac{20648693}{638668800} \end{pmatrix}$$

On the central meridian ($\lambda = \lambda' = 0$), $\zeta' = \phi'$ is the conformal latitude $\chi$ and $\zeta$ plays the role of the rectifying latitude (sometimes denoted by $\mu$). $\mathsf{C}_{\mu,\chi}$ and $\mathsf{C}_{\chi,\mu}$ are the coefficients in the trigonometric series for converting between $\chi$ and $\mu$.

### Legacy ellipsoidal form

The formulas below describe the algorithm used when giving the `+approx` option. They are originally from [Snyder1987], but here quoted from [Evenden1995] and [Evenden2005]. These are less accurate that the formulation above and are only valid within about 5 degrees of the central meridian. Here $M(\phi)$ is the meridional distance.

### Forward projection

$$N = \frac{k_0}{(1 - e^2 \sin^2 \phi)^{1/2}}$$

$$R = \frac{k_0(1 - e^2)}{(1 - e^2 \sin^2 \phi)^{3/2}}$$

$$t = \tan \phi$$

$$\eta = \frac{e^2}{1 - e^2} \cos^2 \phi$$

$$\begin{aligned} x = {} & k_0 \lambda \cos \phi \\ & + \frac{k_0 \lambda^3 \cos^3 \phi}{3!}(1 - t^2 + \eta^2) \\ & + \frac{k_0 \lambda^5 \cos^5 \phi}{5!}(5 - 18t^2 + t^4 + 14\eta^2 - 58t^2\eta^2) \\ & + \frac{k_0 \lambda^7 \cos^7 \phi}{7!}(61 - 479t^2 + 179t^4 - t^6) \end{aligned}$$

$$\begin{aligned} y = {} & M(\phi) \\ & + \frac{k_0 \lambda^2 \sin \phi \cos \phi}{2!} \\ & + \frac{k_0 \lambda^4 \sin \phi \cos^3 \phi}{4!}(5 - t^2 + 9\eta^2 + 4\eta^4) \\ & + \frac{k_0 \lambda^6 \sin \phi \cos^5 \phi}{6!}(61 - 58t^2 + t^4 + 270\eta^2 - 330t^2\eta^2) \\ & + \frac{k_0 \lambda^8 \sin \phi \cos^7 \phi}{8!}(1385 - 3111t^2 + 543t^4 - t^6) \end{aligned}$$

### Inverse projection

$$\phi_1 = M^{-1}(y)$$

$$N_1 = \frac{k_0}{1 - e^2 \sin^2 \phi_1)^{1/2}}$$

$$R_1 = \frac{k_0(1 - e^2)}{(1 - e^2 \sin^2 \phi_1)^{3/2}}$$

$$t_1 = \tan(\phi_1)$$

$$\eta_1 = \frac{e^2}{1 - e^2} \cos^2 \phi_1$$

$$\begin{aligned}
\phi = \phi_1 & \\
& - \frac{t_1 x^2}{2! R_1 N_1} \\
& + \frac{t_1 x^4}{4! R_1 N_1^3}(5 + 3t_1^2 + \eta_1^2 - 4\eta_1^4 - 9\eta_1^2 t_1^2) \\
& - \frac{t_1 x^6}{6! R_1 N_1^5}(61 + 90t_1^2 + 46\eta_1^2 + 45t_1^4 - 252t_1^2\eta_1^2) \\
& + \frac{t_1 x^8}{8! R_1 N_1^7}(1385 + 3633t_1^2 + 4095t_1^4 + 1575t_1^6)
\end{aligned}$$

$$\begin{aligned}
\lambda = & \frac{x}{\cos \phi N_1} \\
& - \frac{x^3}{3! \cos \phi N_1^3}(1 + 2t_1^2 + \eta_1^2) \\
& + \frac{x^5}{5! \cos \phi N_1^5}(5 + 6\eta_1^2 + 28t_1^2 - 3\eta_1^2 + 8t_1^2\eta_1^2) \\
& - \frac{x^7}{7! \cos \phi N_1^7}(61 + 662t_1^2 + 1320t_1^4 + 720t_1^6)
\end{aligned}$$

#### 7.1.132.4 Further reading

1. Wikipedia

### 7.1.133 Tobler-Mercator

Added in version 6.0.0.

Equal area cylindrical projection with the same latitudinal spacing as Mercator projection.

| | |
|---|---|
| **Classification** | Cylindrical equal area |
| **Available forms** | Forward and inverse, spherical only |
| **Defined area** | Global, conventionally truncated at about 80 degrees north and south |
| **Alias** | tobmerc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.133.1 Usage

The inappropriate use of the Mercator projection has declined but still occasionally occurs. One method of contrasting the Mercator projection is to present an alternative in the form of an equal area projection. The map projection derived here is thus not simply a pretty Christmas tree ornament: it is instead a complement to Mercator's conformal navigation anamorphosis and can be displayed as an alternative. The equations for the new map projection preserve the latitudinal

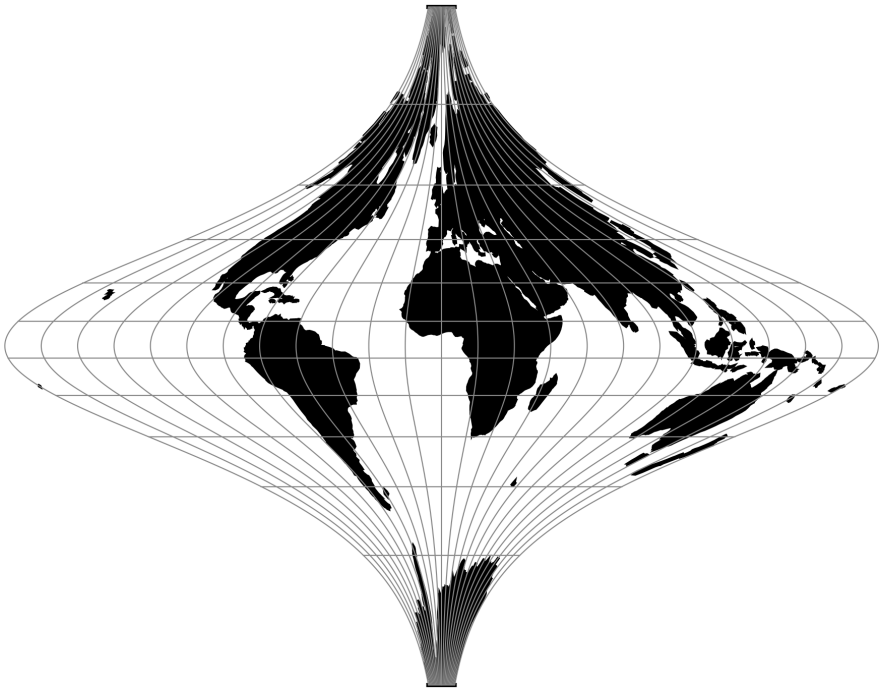


Fig. 129: proj-string: `+proj=tobmerc`

stretching of the Mercator while adjusting the longitudinal spacing. This allows placement of the new map adjacent to that of Mercator. The surface area, while drastically warped, maintains the correct magnitude.

#### 7.1.133.2 Parameters

> **Note**
>
> All parameters for the projection are optional.

**+k_0**=<value>

Scale factor. Determines scale factor used in the projection.

*Defaults to 1.0.*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

#### 7.1.133.3 Mathematical definition

The formulas describing the Tobler-Mercator are taken from Waldo Tobler's article [Tobler2018]

##### Spherical form

For the spherical form of the projection we introduce the scaling factor:

$$k_0 = \cos^2 \phi_{ts}$$

##### Forward projection

$$x = k_0 \lambda$$

$$y = k_0 \ln \left[ \tan \left( \frac{\pi}{4} + \frac{\phi}{2} \right) \right]$$

##### Inverse projection

$$\lambda = \frac{x}{k_0}$$

$$\phi = \frac{\pi}{2} - 2 \arctan \left[ e^{-y/k_0} \right]$$

### 7.1.134 Two Point Equidistant

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | tpeqd |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

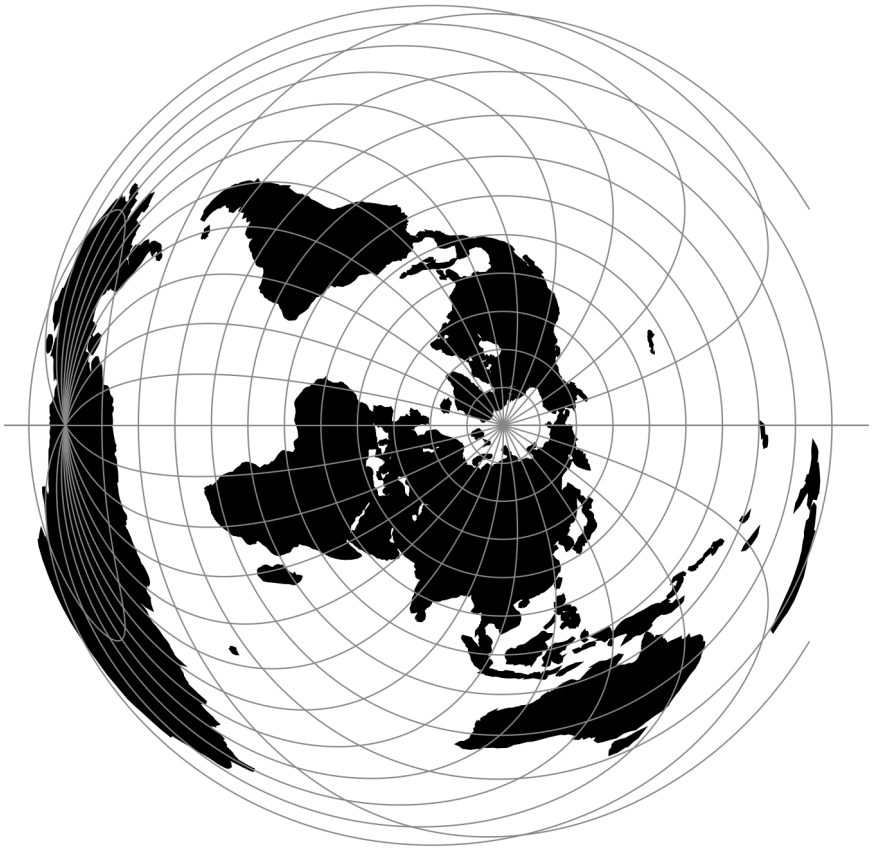


Fig. 130: proj-string: `+proj=tpeqd +lat_1=60 +lat_2=65`

#### 7.1.134.1 Parameters

**Note**

All parameters are optional for the projection.

**+lon_1**=<value>

Longitude of first point.

**+lat_1**=<value>

Latitude of first point.

**+lon_2**=<value>

Longitude of second point.

**+lat_2**=<value>

Latitude of second point.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.135 Tilted perspective

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | tpers |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Tilted Perspective is similar to *Near-sided perspective* (nsper) in that it simulates a perspective view from a height. Where nsper projects onto a plane tangent to the surface, Tilted Perspective orients the plane towards the direction of the view. Thus, extra parameters specifying azimuth and tilt are required beyond *nsper* 's h. As with nsper, lat_0 & lon_0 are also required for satellite position.

#### 7.1.135.1 Parameters

##### Required

**+h**=<value>

Height of the view point above the Earth and must be in the same units as the radius of the sphere or semimajor axis of the ellipsoid.

Fig. 131: proj-string: `+proj=tpers +h=5500000 +lat_0=40`

### Optional

**+azi**=<value>

Bearing in degrees away from north.

*Defaults to 0.0.*

**+tilt**=<value>

Angle in degrees away from nadir.

*Defaults to 0.0.*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+lat_0**=<value>

Latitude of natural origin, latitude of false origin or latitude of projection centre (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.136 Universal Polar Stereographic

Fig. 132: proj-string: `+proj=ups`

#### 7.1.136.1 Parameters

**Note**

All parameters are optional for the projection.

**+south**

South polar aspect.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.137 Urmaev V

#### 7.1.137.1 Parameters

#### Required parameters

**+n**=<value>

Set the $n$ constant. Value between 0 and 1.

#### Optional parameters

**+q**=<value>

Set the $q$ constant.

**+alpha**=<value>

Set the $\alpha$ constant.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

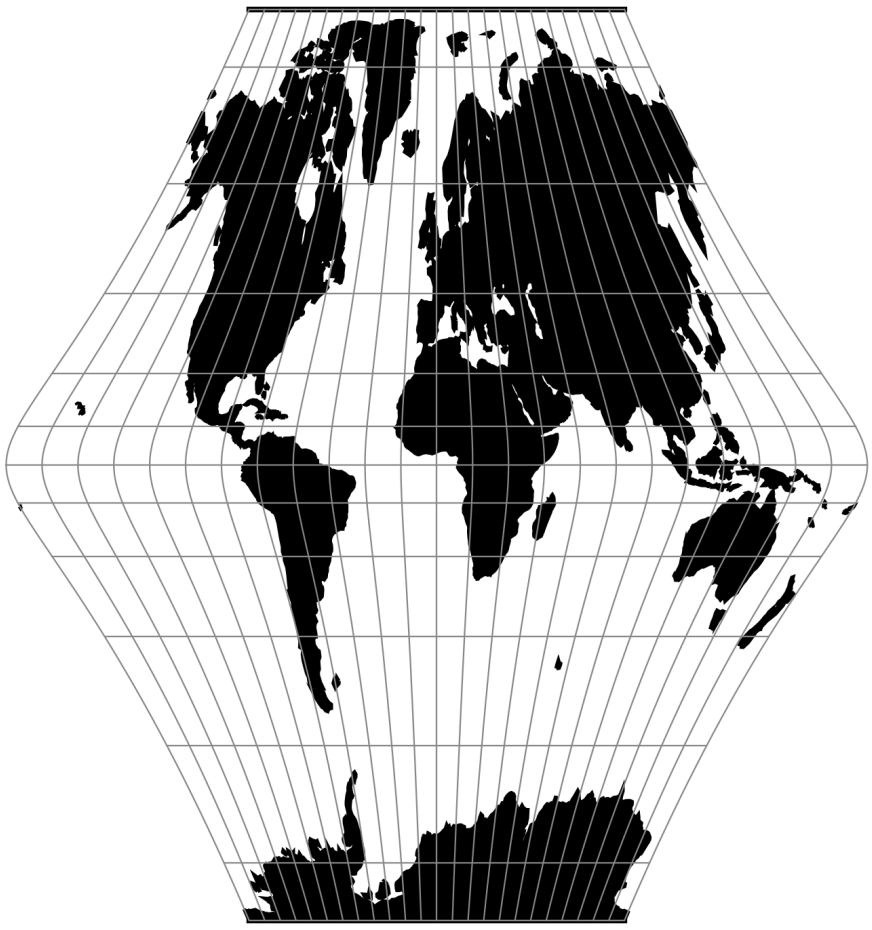


Fig. 133: proj-string: `+proj=urm5 +n=0.9 +alpha=2 +q=4`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.138 Urmaev Flat-Polar Sinusoidal

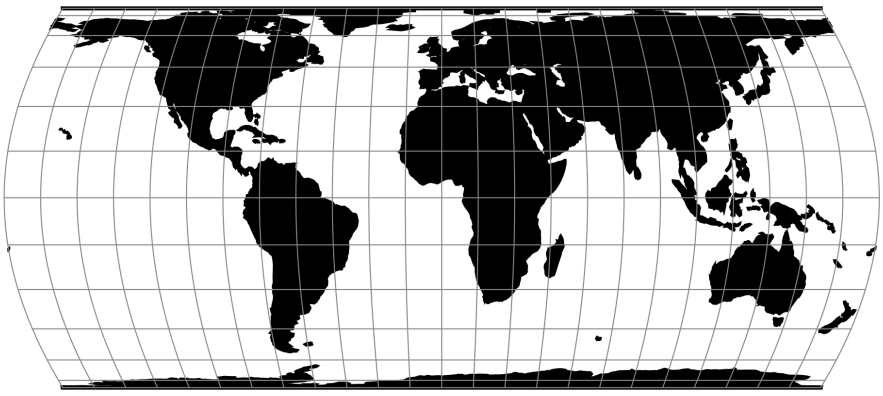


Fig. 134: proj-string: `+proj=urmfps +n=0.5`

#### 7.1.138.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+n**=<value>

Set the $n$ constant. Value between 0 and 1.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.139 Universal Transverse Mercator (UTM)

The Universal Transverse Mercator is a system of map projections divided into sixty zones across the globe, with each zone corresponding to 6 degrees of longitude.

| | |
|---|---|
| **Classification** | Transverse cylindrical, conformal |
| **Available forms** | Forward and inverse, ellipsoidal only |
| **Defined area** | Within the used zone, but transformations of coordinates in adjacent zones can be expected to be accurate as well |
| **Alias** | utm |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

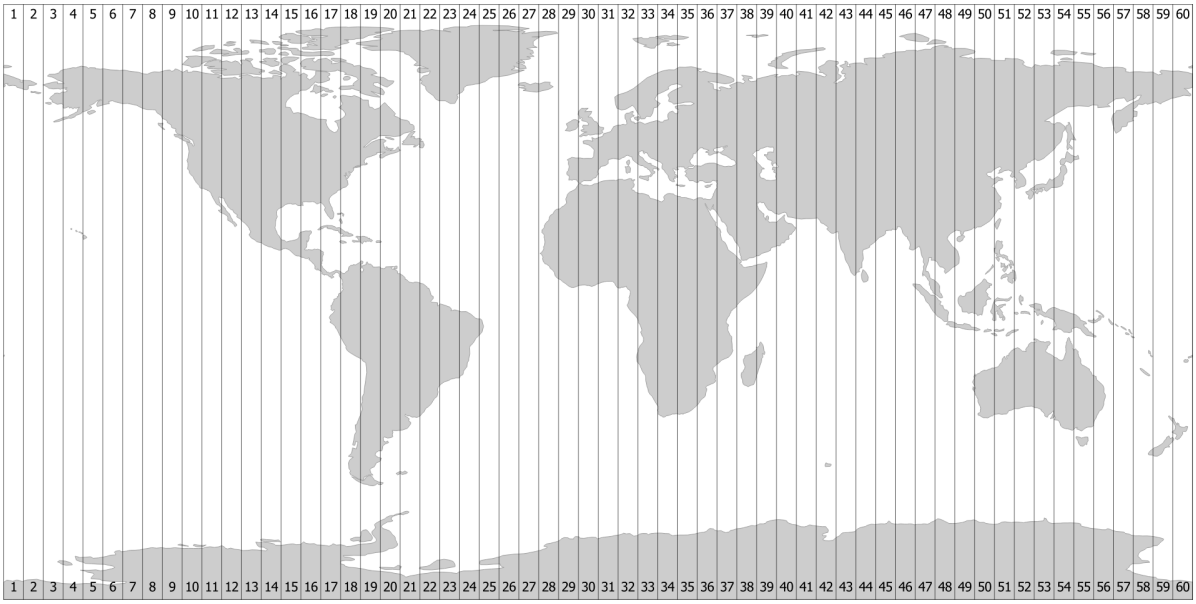



Fig. 135: UTM zones.

UTM projections are really the *Transverse Mercator* to which specific parameters, such as central meridians, have been applied. The Earth is divided into 60 zones each generally 6° wide in longitude. Bounding meridians are evenly divisible by 6°, and zones are numbered from 1 to 60 proceeding east from the 180th meridian from Greenwich with minor exceptions [Snyder1987].

#### 7.1.139.1 Usage

Convert geodetic coordinate to UTM Zone 32 on the northern hemisphere:

```
$ echo 12 56 | proj +proj=utm +zone=32
687071.44       6210141.33
```

Convert geodetic coordinate to UTM Zone 59 on the southern hemisphere:

```
$ echo 174 -44 | proj +proj=utm +zone=59 +south
740526.32       5123750.87
```

Show the relationship of UTM to TM:

```
$ echo 121 24 | proj +proj=utm +lon_0=123 | proj -I +proj=tmerc +lon_0=123 +x_0=500000␣
↪+k=0.9996
121dE 24dN
```

#### 7.1.139.2 Parameters

##### Required

**+zone**=<value>

Select which UTM zone to use. Can be a value between 1-60.

##### Optional

**+south**

Add this flag when using the UTM on the southern hemisphere.

**+approx**

Added in version 6.0.0.

Use faster, less accurate algorithm for the Transverse Mercator.

**+algo**=auto/evenden_snyder/poder_engsager

Added in version 7.1.

Selects the algorithm to use. The hardcoded value and the one defined in *proj.ini* default to poder_engsager, that is the most precise one.

When using auto, a heuristics based on the input coordinate to transform is used to determine if the faster Evenden-Snyder method can be used, for faster computation, without causing an error greater than 0.1 mm (for an ellipsoid of the size of Earth)

Note that *+approx* and *+algo* are mutually exclusive.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute *proj -le* for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

#### 7.1.139.3 Further reading

1. Wikipedia

### 7.1.140 van der Grinten (I)

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | vandg |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 136: proj-string: +proj=vandg

#### 7.1.140.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.141 van der Grinten II

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | vandg2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.141.1 Parameters

Fig. 137: proj-string: +proj=vandg2

> **Note**
>
> All parameters are optional for the projection.

`+lon_0`=<value>
: Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

  *Defaults to 0.0.*

  > **Note**
  >
  > The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
  >
  > Example: *+lon_0=1.570796r*
  >
  > See *Projection Units* for more information.

`+R`=<value>
: Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

  See *Ellipsoid size parameters* for more information.

`+x_0`=<value>
: False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

`+y_0`=<value>
: False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

  *Defaults to 0.0.*

### 7.1.142 van der Grinten III

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | vandg3 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.142.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

Fig. 138: proj-string: +proj=vandg3

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.143 van der Grinten IV

| | |
|---|---|
| **Classification** | Miscellaneous |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | vandg4 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.143.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

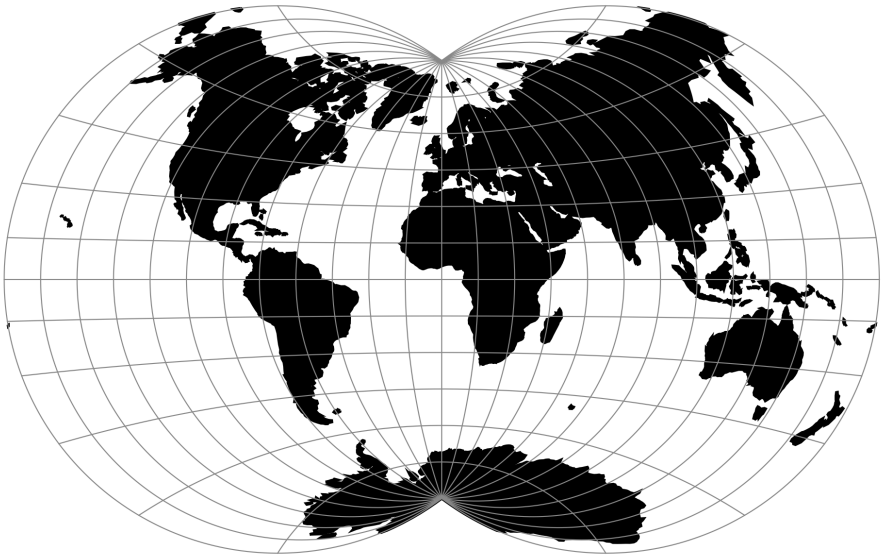


Fig. 139: proj-string: `+proj=vandg4`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.144 Vitkovsky I

| | |
|---|---|
| **Classification** | Conical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | vitk1 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

Fig. 140: proj-string: `+proj=vitk1 +lat_1=45 +lat_2=55`

#### 7.1.144.1 Parameters

#### Required

`+lat_1=<value>`

First standard parallel.

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_1=1.570796r*

See *Projection Units* for more information.

**+lat_2**=<value>

Second standard parallel.

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lat_2=1.570796r*

See *Projection Units* for more information.

### Optional

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.145 Wagner I (Kavrayskiy VI)

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wag1 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

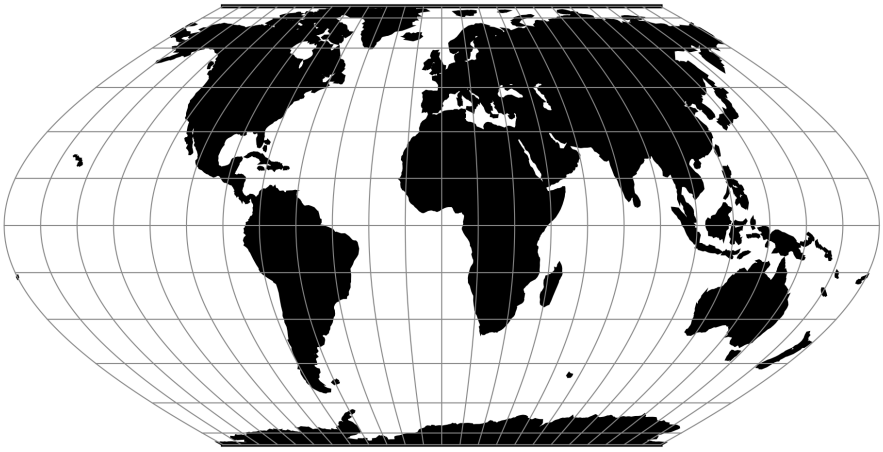


Fig. 141: proj-string: `+proj=wag1`

> **Note**
>
> This projection name may also be transliterated as Kavraisky VI.

#### 7.1.145.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.146 Wagner II

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wag2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

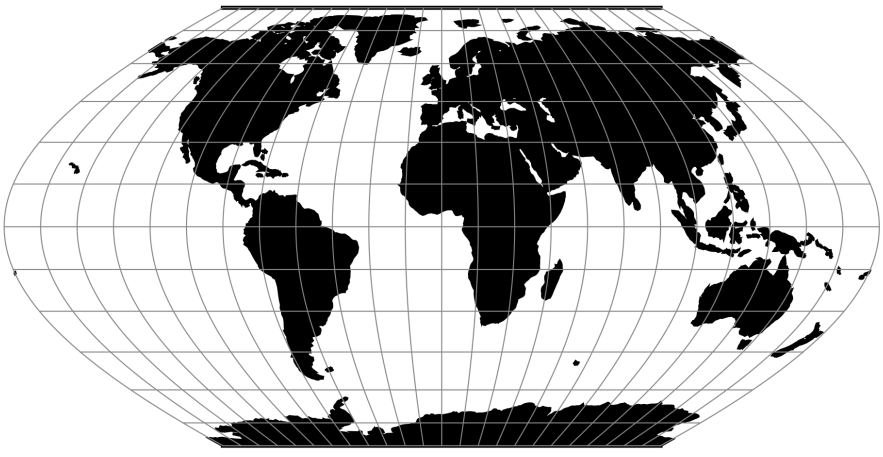


Fig. 142: proj-string: +proj=wag2

$$x = 0.92483\lambda\cos\theta$$
$$y = 1.38725\theta$$
$$\sin\theta = 0.88022\sin(0.8855\phi)$$

#### 7.1.146.1 Parameters

**Note**

All parameters are optional for the projection.

`+lon_0`=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

`+R`=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

`+x_0`=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

`+y_0`=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.147 Wagner III

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wag3 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

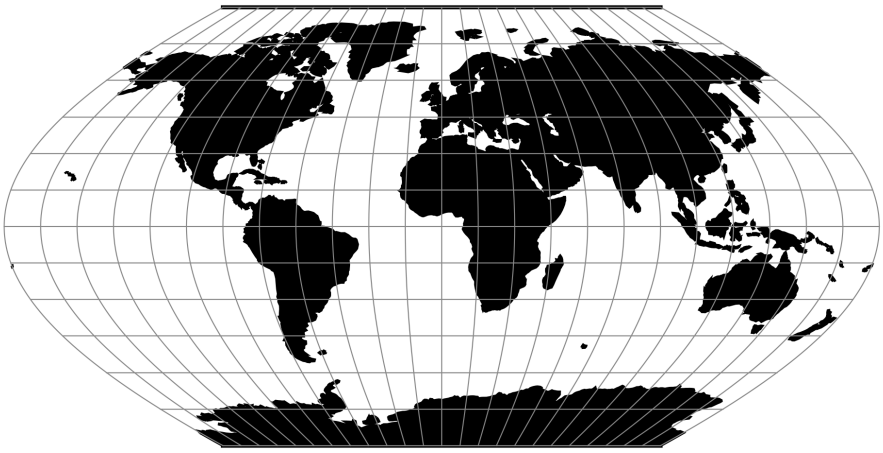


Fig. 143: proj-string: `+proj=wag3`

$$x = [\cos \phi_{ts} / \cos(2\phi_{ts}/3)]\lambda \cos(2\phi/3)$$
$$y = \phi$$

#### 7.1.147.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lat_ts**=<value>

Latitude of true scale. Defines the latitude where scale is not distorted. Takes precedence over **+k_0** if both options are used together.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_ts=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.148 Wagner IV

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wag4 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.148.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

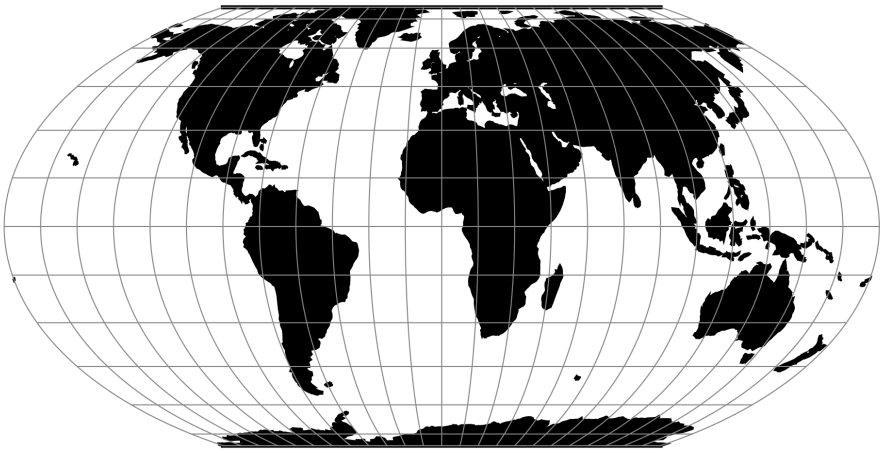


Fig. 144: proj-string: `+proj=wag4`

> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.149 Wagner V

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wag5 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

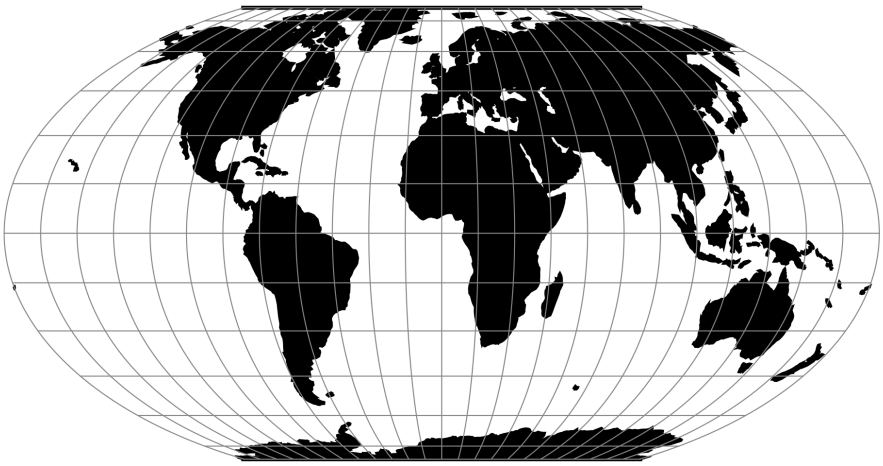


Fig. 145: proj-string: `+proj=wag5`

#### 7.1.149.1 Parameters

> **Note**
>
> All parameters are optional.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the

projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.150 Wagner VI

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wag6 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

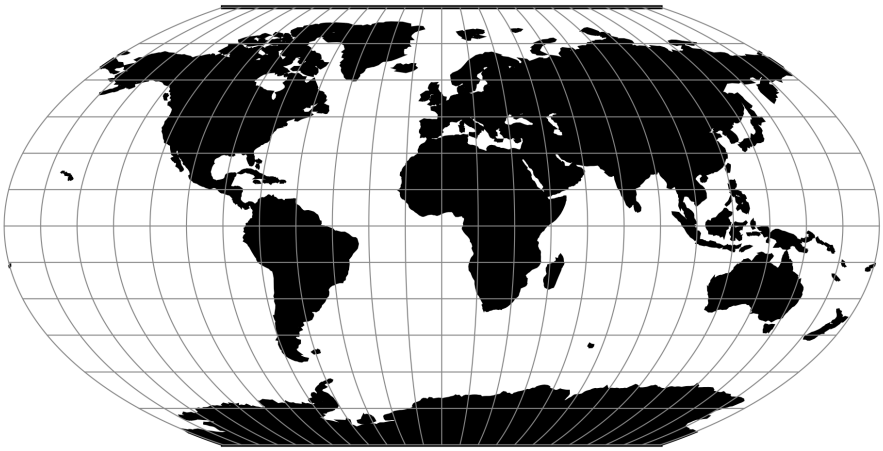


Fig. 146: proj-string: `+proj=wag6`

#### 7.1.150.1 Parameters

**Note**

All parameters are optional for the Wagner VI projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.151 Wagner VII

| | |
|---|---|
| **Classification** | Azimuthal |
| **Available forms** | Forward spherical projection |
| **Defined area** | Global |
| **Alias** | wag7 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

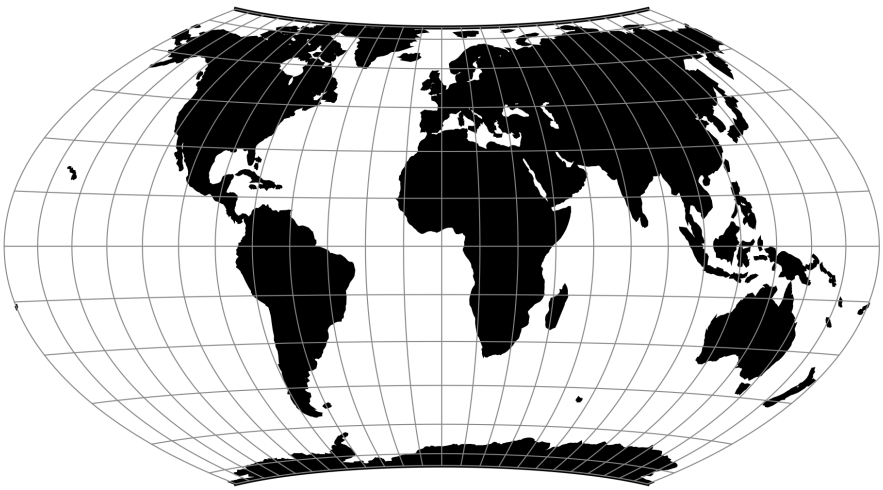


Fig. 147: proj-string: +proj=wag7

### 7.1.152 Web Mercator / Pseudo Mercator

Added in version 5.1.0.

The Web Mercator / Pseudo Mercator projection is a cylindrical map projection. This is a variant of the regular *Mercator* projection, except that the computation is done on a sphere, using the semi-major axis of the ellipsoid.

From Wikipedia:

> This projection is widely used by the Web Mercator, Google Web Mercator, Spherical Mercator, WGS 84 Web Mercator[1] or WGS 84/Pseudo-Mercator is a variant of the Mercator projection and is the de facto standard for Web mapping applications. [...] It is used by virtually all major online map providers [...] Its official EPSG identifier is EPSG:3857, although others have been used historically.

| | |
|---|---|
| **Classification** | Cylindrical (non conformant if used with ellipsoid) |
| **Available forms** | Forward and inverse |
| **Defined area** | Global |
| **Alias** | webmerc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.152.1 Usage

Example:

```
$ echo 2 49 | proj +proj=webmerc +datum=WGS84
222638.98       6274861.39
```

#### 7.1.152.2 Parameters

**Note**

All parameters for the projection are optional, except the ellipsoid definition, which is WGS84 for the typical use case of EPSG:3857. In which case, the other parameters are set to their default 0 value.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

**Note**

The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.

Example: *+lon_0=1.570796r*

See *Projection Units* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

#### 7.1.152.3 Mathematical definition

The formulas describing the Mercator projection are all taken from G. Evenden's libproj manuals [Evenden2005].

##### Forward projection

$$x = \lambda$$

$$y = \ln\left[\tan\left(\frac{\pi}{4} + \frac{\phi}{2}\right)\right]$$

##### Inverse projection

$$\lambda = x$$

$$\phi = \frac{\pi}{2} - 2\arctan\left[e^{-y}\right]$$

#### 7.1.152.4 Further reading

1. Wikipedia

### 7.1.153 Werenskiold I

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | weren |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.153.1 Parameters

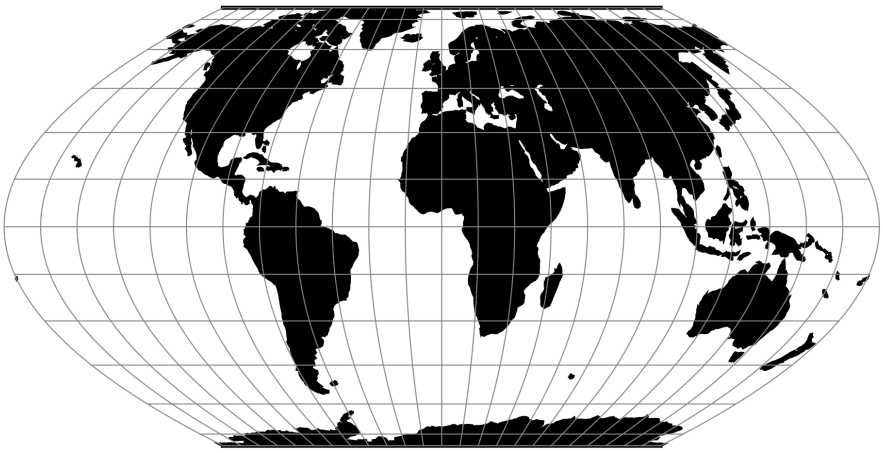


Fig. 148: proj-string: `+proj=weren`

> **Note**
>
> All parameters are optional for the projection.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.154 Winkel I

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wink1 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

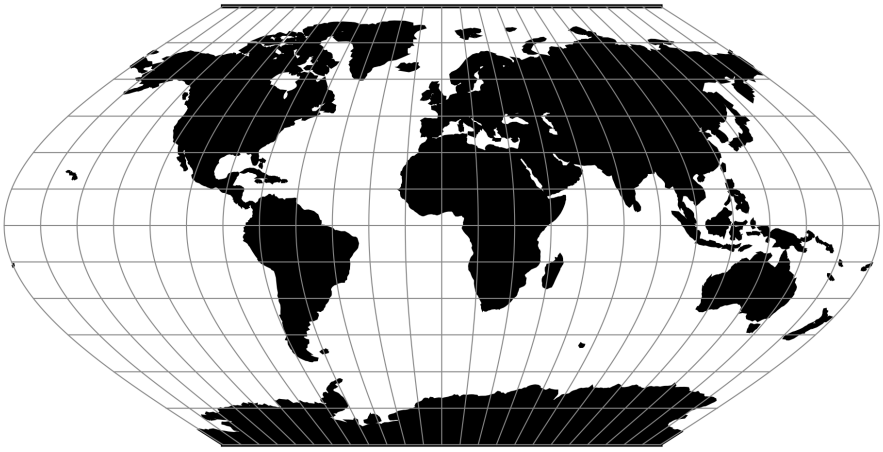


Fig. 149: proj-string: `+proj=wink1`

#### 7.1.154.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lat_ts**=<value>

Latitude of true scale. Defines the latitude where scale is not distorted. Takes precedence over `+k_0` if both options are used together.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_ts=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with +ellps, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.155 Winkel II

| | |
|---|---|
| **Classification** | Pseudocylindrical |
| **Available forms** | Forward and inverse, spherical projection |
| **Defined area** | Global |
| **Alias** | wink2 |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

#### 7.1.155.1 Parameters

> **Note**
>
> All parameters are optional for the projection.

**+lat_ts**=<value>

Latitude of true scale. Defines the latitude where scale is not distorted. Takes precedence over +k_0 if both options are used together.

*Defaults to 0.0.*

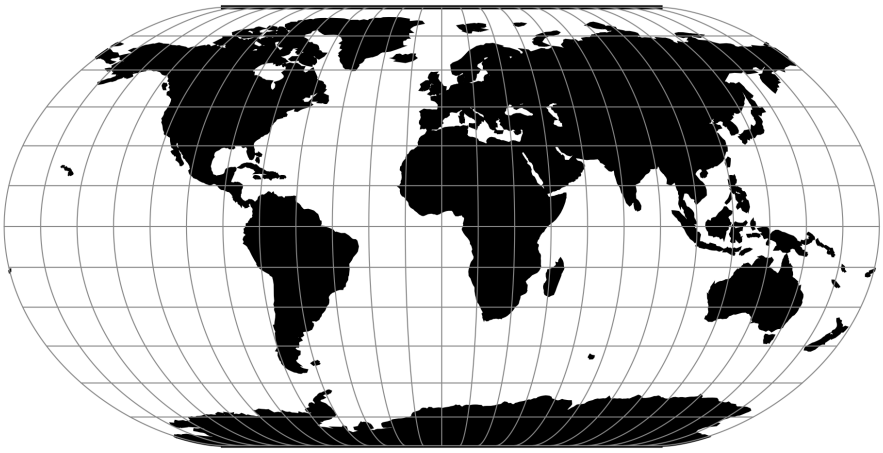


Fig. 150: proj-string: `+proj=wink2`

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_ts=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

### 7.1.156 Winkel Tripel

| | |
|---|---|
| **Classification** | Pseudoazimuthal |
| **Available forms** | Forward and inverse spherical projection |
| **Defined area** | Global |
| **Alias** | wintri |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Projected coordinates |

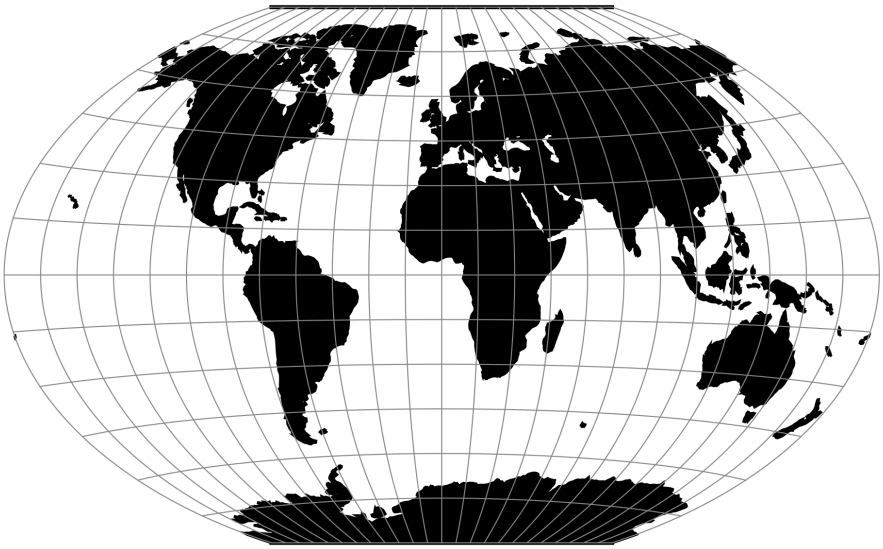


Fig. 151: proj-string: `+proj=wintri`

#### 7.1.156.1 Parameters

**Note**

All parameters are optional for the projection.

**+lat_1**=<value>

First standard parallel.

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lat_1=1.570796r*
>
> See *Projection Units* for more information.

**+lon_0**=<value>

Central meridian/longitude of natural origin, longitude of origin or longitude of false origin (naming and meaning depend on the projection method).

*Defaults to 0.0.*

> **Note**
>
> The default convention is to interpret this value as decimal degrees. To specify radians instead, follow the value with the "r" character.
>
> Example: *+lon_0=1.570796r*
>
> See *Projection Units* for more information.

**+R**=<value>

Radius of the sphere, given in meters. If used in conjunction with `+ellps`, *+R* takes precedence.

See *Ellipsoid size parameters* for more information.

**+x_0**=<value>

False easting, easting at false origin or easting at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

**+y_0**=<value>

False northing, northing at false origin or northing at projection centre (naming and meaning depend on the projection method). Always in meters.

*Defaults to 0.0.*

## 7.2 Conversions

Conversions are coordinate operations in which both coordinate reference systems are based on the same datum. In PROJ projections are differentiated from conversions.

### 7.2.1 Axis swap

Added in version 5.0.0.

Change the order and sign of 2,3 or 4 axes.

| | |
|---|---|
| **Alias** | axisswap |
| **Domain** | 2D, 3D or 4D |
| **Input type** | Any |
| **Output type** | Any |

Each of the possible four axes are numbered with 1--4, such that the first input axis is 1, the second is 2 and so on. The output ordering is controlled by a list of the input axes re-ordered to the new mapping.

#### 7.2.1.1 Usage

Reversing the order of the axes:

```
+proj=axisswap +order=4,3,2,1
```

Swapping the first two axes (x and y):

```
+proj=axisswap +order=2,1,3,4
```

The direction, or sign, of an axis can be changed by adding a minus in front of the axis-number:

```
+proj=axisswap +order=1,-2,3,4
```

It is only necessary to specify the axes that are affected by the swap operation:

```
+proj=axisswap +order=2,1
```

#### 7.2.1.2 Parameters

**+order**=<list>

Ordered comma-separated list of axis, e.g. *+order=2,1,3,4*. Adding a minus in front of an axis number results in a change of direction for that axis, e.g. southward instead of northward.

*Required.*

### 7.2.2 Geodetic to cartesian conversion

Added in version 5.0.0.

Convert geodetic coordinates to cartesian coordinates (in the forward path).

| | |
|---|---|
| **Alias** | cart |
| **Domain** | 3D |
| **Input type** | Geodetic coordinates |
| **Output type** | Geocentric cartesian coordinates |

This conversion converts geodetic coordinate values (longitude, latitude, elevation above ellipsoid) to their geocentric (X, Y, Z) representation, where the first axis (X) points from the Earth centre to the point of longitude=0, latitude=0, the second axis (Y) points from the Earth centre to the point of longitude=90, latitude=0 and the third axis (Z) points to the North pole.

#### 7.2.2.1 Usage

Convert geodetic coordinates to cartesian (uses default +ellps=GRS80):

```
echo 17.7562015132 45.3935192042 133.12 2017.8 | cct +proj=cart
4272922.1553   1368283.0597  4518261.3501     2017.8000
```

#### 7.2.2.2 Parameters

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

### 7.2.3 Geocentric Latitude

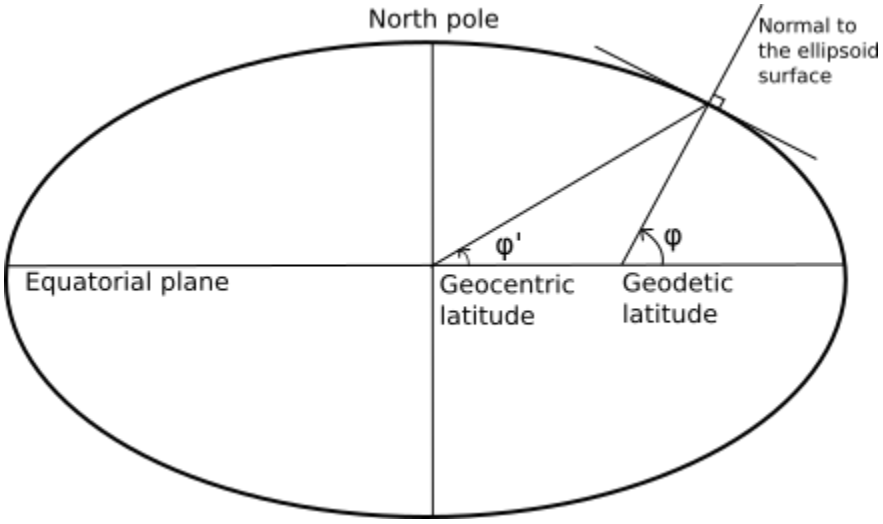
Added in version 5.0.0.

Convert from Geodetic Latitude to Geocentric Latitude (in the forward path).

| | |
|---|---|
| **Alias** | geoc |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Geocentric angular coordinates |

The geodetic (or geographic) latitude (also called planetographic latitude in the context of non-Earth bodies) is the angle between the equatorial plane and the normal (vertical) to the ellipsoid surface at the considered point. The geodetic latitude is what is normally used everywhere in PROJ when angular coordinates are expected or produced.

The geocentric latitude (also called planetocentric latitude in the context of non-Earth bodies) is the angle between the equatorial plane and a line joining the body centre to the considered point.




**Note**

This conversion must be distinguished from the *Geodetic to cartesian conversion* which converts geodetic coordinates to geocentric coordinates in the cartesian domain.

#### 7.2.3.1 Mathematical definition

The formulas describing the conversion are taken from [Snyder1987] (equation 3-28)

Let $\phi'$ to be the geocentric latitude and $\phi$ the geodetic latitude, then

$$\phi' = \arctan\left[(1 - e^2)\tan(\phi)\right]$$

The geocentric latitude is consequently lesser (in absolute value) than the geodetic latitude, except at the equator and the poles where they are equal.

On a sphere, they are always equal.

#### 7.2.3.2 Usage

Converting from geodetic latitude to geocentric latitude:

```
+proj=geoc
```

Converting from geocentric latitude to geodetic latitude:

```
+proj=pipeline +step +proj=geoc +inv
```

#### 7.2.3.3 Parameters

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

### 7.2.4 Lat/long (Geodetic alias)

Passes geodetic coordinates through unchanged.

| | |
|---|---|
| **Aliases** | latlon, latlong, lonlat, longlat |
| **Domain** | 2D |
| **Input type** | Geodetic coordinates |
| **Output type** | Geodetic coordinates |

**Note**

Can not be used with the *proj* application.

#### 7.2.4.1 Parameters

No parameters will affect the output of the operation if used on it's own. However, the parameters below can be used in a declarative manner when used with *cs2cs* or in a *transformation pipeline* .

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+datum**=<value>

Declare the datum used with the coordinates. Available options are: `WGS84`, `GGRS87`, `NAD38`, `NAD27`, `potsdam`, `carthage`, `hermannskogel`, `ire65`, `nzgd49`, `OSGB336`.

> **Note**
>
> The `+datum` option is primarily available to support the legacy use of PROJ.4 strings as CRS descriptors and should in most cases be avoided.

**+towgs84**=<list>

A list of three or seven comma separated *Helmert* parameters that maps the input coordinates to the WGS84 datum.

### 7.2.5 No operation

Added in version 6.1.0.

Pass a coordinate through unchanged.

| | |
|---|---|
| **Alias** | noop |
| **Domain** | 4D |
| **Input type** | Any |
| **Output type** | Any |

The no operation is a dummy operation that returns whatever is passed to it as seen in this example:

```
$ echo 12 34 56 78 | cct +proj=noop
  12.0000        34.0000       56.0000       78.0000
```

The operation has no options and default options will not affect the output.

### 7.2.6 Pop coordinate value to pipeline stack

Added in version 6.0.0.

Retrieve components of a coordinate that was saved in a previous pipeline step.

| | |
|---|---|
| **Alias** | pop |
| **Domain** | 4D |
| **Input type** | Any |
| **Output type** | Any |

This operations makes it possible to retrieve coordinate components that was saved in previous pipeline steps. A retrieved coordinate component is loaded, or *popped*, from a memory stack that is part of a *pipeline*. The pipeline coordinate stack is inspired by the stack data structure that is commonly used in computer science. There's four stacks available: One four each coordinate dimension. The dimensions, or coordinate components, are numbered 1--4. It is only possible to move data to and from the stack within the same coordinate component number. Values can be saved to the stack by using the *push operation*.

If the pop operation is used by itself, e.g. not in a pipeline, it will function as a no-operation that passes the coordinate through unchanged. Similarly, if no coordinate component is available on the stack to be popped the operation does nothing.

#### 7.2.6.1 Examples

A common use of the *push* and pop operations is in 3D *Helmert* transformations where only the horizontal components are needed. This is often the case when combining heights from a legacy vertical reference with a modern geocentric reference. Below is a an example of such a transformation, where the horizontal part is transformed with a Helmert operation but the vertical part is kept exactly as the input was.

```
$ echo 12 56 12.3 2020 | cct +proj=pipeline  \
+step +proj=push +v_3                        \
+step +proj=cart +ellps=GRS80                \
+step +proj=helmert +x=3000 +y=1000 +z=2000  \
+step +proj=cart +ellps=GRS80 +inv           \
+step +proj=pop +v_3                         \

 12.0056753463   55.9866540552       12.3000     2000.0000
```

Note that the third coordinate component in the output is the same as the input.

The same transformation without the push and pop operations would look like this:

```
$ echo 12 56 12.3 2020 | cct +proj=pipeline  \
+step +proj=cart +ellps=GRS80                \
+step +proj=helmert +x=3000 +y=1000 +z=2000  \
+step +proj=cart +ellps=GRS80 +inv           \

      12.0057        55.9867     3427.7404     2000.0000
```

Here the vertical component is adjusted significantly.

#### 7.2.6.2 Parameters

**+v_1**

Retrieves the first coordinate component from the pipeline stack

**+v_2**

Retrieves the second coordinate component from the pipeline stack

**+v_3**

Retrieves the third coordinate component from the pipeline stack

**+v_4**

Retrieves the fourth coordinate component from the pipeline stack

#### 7.2.6.3 Further reading

1. Stack data structure on Wikipedia

### 7.2.7 Push coordinate value to pipeline stack

Added in version 6.0.0.

Save components of a coordinate from one step of a pipeline and make it available for retrieving in another pipeline step.

| | |
|---|---|
| **Alias** | push |
| **Domain** | 4D |
| **Input type** | Any |
| **Output type** | Any |

This operations allows for components of coordinates to be saved for application in a later step. A saved coordinate component is moved, or *pushed*, to a memory stack that is part of a *pipeline*. The pipeline coordinate stack is inspired by the stack data structure that is commonly used in computer science. There's four stacks available: One four each coordinate dimension. The dimensions, or coordinate components, are numbered 1--4. It is only possible to move data to and from the stack within the same coordinate component number. Values can be moved off the stack again by using the *pop operation*.

If the push operation is used by itself, e.g. not in a pipeline, it will function as a no-operation that passes the coordinate through unchanged.

#### 7.2.7.1 Examples

A common use of the push and *pop* operations is in 3D *Helmert* transformations where only the horizontal components are needed. This is often the case when combining heights from a legacy vertical reference with a modern geocentric reference. Below is a an example of such a transformation, where the horizontal part is transformed with a Helmert operation but the vertical part is kept exactly as the input was.

```
$ echo 12 56 12.3 2020 | cct +proj=pipeline  \
+step +proj=push +v_3                        \
+step +proj=cart                             \
+step +proj=helmert +x=3000 +y=1000 +z=2000  \
+step +proj=cart +inv                        \
+step +proj=pop +v_3

 12.0056753463   55.9866540552       12.3000     2000.0000
```

Note that the third coordinate component in the output is the same as the input.

The same transformation without the push and pop operations would look like this:

```
$ echo 12 56 12.3 2020 | cct +proj=pipeline  \
+step +proj=cart                             \
+step +proj=helmert +x=3000 +y=1000 +z=2000  \
+step +proj=cart +inv

      12.0057        55.9867     3427.7404     2000.0000
```

Here the vertical component is adjusted significantly.

#### 7.2.7.2 Parameters

**+v_1**

Stores the first coordinate component on the pipeline stack

**+v_2**

Stores the second coordinate component on the pipeline stack

**+v_3**

Stores the third coordinate component on the pipeline stack

**+v_4**

Stores the fourth coordinate component on the pipeline stack

#### 7.2.7.3 Further reading

1. Stack data structure on Wikipedia

### 7.2.8 Set coordinate value

Added in version 7.0.0.

Set component(s) of a coordinate to a fixed value.

| | |
|---|---|
| **Alias** | set |
| **Domain** | 4D |
| **Input type** | Any |
| **Output type** | Any |

This operations allows for components of coordinates to be set to a fixed value. This may be useful in *pipeline* when a step requires some component, typically an elevation or a date, to be set to a fixed value.

#### 7.2.8.1 Example

In the ETRS89 to Dutch RD with NAP height transformation, the used ellipsoidal height for the Helmert transformation is not the NAP height, but the height is set to 0 m. This is an unconventional trick to get the same results as when the effect of the Helmert transformation is included in the horizontal NTv2 grid. For the forward transformation from ETRS89 to RD with NAP height, we need to set the ellipsoidal ETRS89 height for the Helmert transformation to the equivalent of 0 m NAP. This is 43 m for the centre of the Netherlands and this value can be used as an approximation elsewhere (the effect of this approximation is below 1 mm for the horizontal coordinates, in an area up to hundreds of km outside the Netherlands).

The `+proj=set +v_3=0` close to the end of the pipeline is to make it usable in the reverse direction.

```
$ cct -t 0 -d 4 +proj=pipeline \
    +step +proj=unitconvert +xy_in=deg +xy_out=rad \
    +step +proj=axisswap +order=2,1 \
    +step +proj=vgridshift +grids=nlgeo2018.gtx \
    +step +proj=push +v_3 \
    +step +proj=set +v_3=43 \
    +step +proj=cart \
    +step +proj=helmert +x=-565.7346 +y=-50.4058 +z=-465.2895 +rx=-0.395023 +ry=0.
↪330776 +rz=-1.876073 +s=-4.07242 +convention=coordinate_frame +exact \
    +step +proj=cart +inv +ellps=bessel \
    +step +proj=hgridshift +inv +grids=rdcorr2018.gsb,null \
    +step +proj=sterea +lat_0=52.156160556 +lon_0=5.387638889 +k=0.9999079 +x_0=155000␣
↪+y_0=463000 +ellps=bessel \
    +step +proj=set +v_3=0 \
    +step +proj=pop +v_3
```

#### 7.2.8.2 Parameters

**+v_1**=value

Set the first coordinate component to the specified value

**+v_2**=value
: Set the second coordinate component to the specified value

**+v_3**=value
: Set the third coordinate component to the specified value

**+v_4**=value
: Set the fourth coordinate component to the specified value

### 7.2.9 Geocentric to topocentric conversion

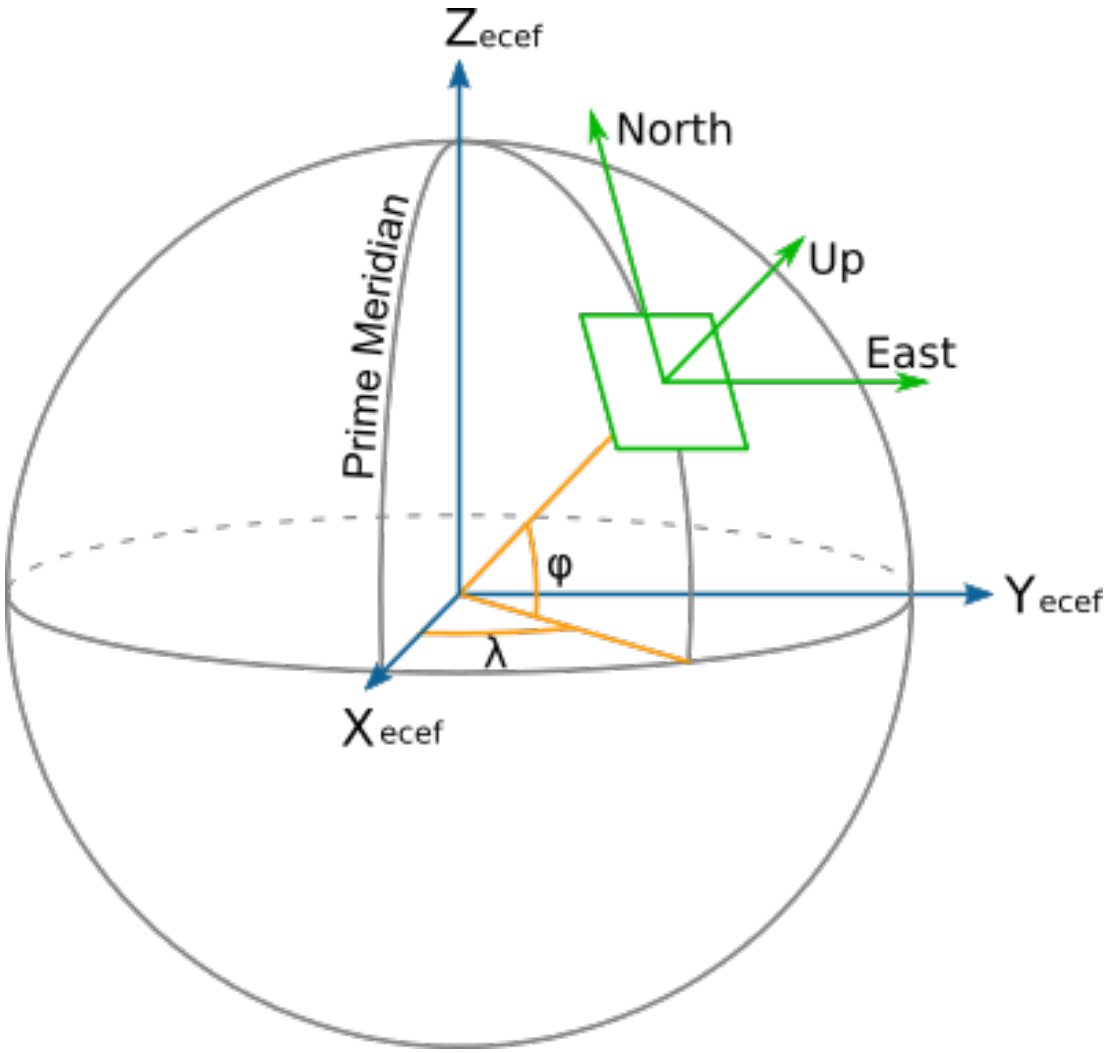
Added in version 8.0.0.

Convert geocentric coordinates to topocentric coordinates (in the forward path).

| | |
|---|---|
| **Alias** | topocentric |
| **Domain** | 3D |
| **Input type** | Geocentric cartesian coordinates |
| **Output type** | Topocentric cartesian coordinates |

This operation converts geocentric coordinate values (X, Y, Z) to topocentric (E/East, N/North, U/Up) values. This is also sometimes known as the ECEF (Earth Centered Earth Fixed) to ENU conversion.

Topocentric coordinates are expressed in a frame whose East and North axis form a local tangent plane to the Earth's ellipsoidal surface fixed to a specific location (the topocentric origin), and the Up axis points upwards along the normal to that plane.




The topocentric origin is a required parameter of the conversion, and can be expressed either as geocentric coordinates (`X_0`, `Y_0` and `Z_0`) or as geographic coordinates (`lat_0`, `lon_0`, `h_0`).

When conversion between geographic and topocentric coordinates is desired, the topocentric conversion must be preceded by the *Geodetic to cartesian conversion* conversion to perform the initial geographic to geocentric coordinates conversion.

The formulas used come from the "Geocentric/topocentric conversions" paragraph of [IOGP2018]. `+proj=topocentric` alone corresponds to the `EPSG:9836` conversion method, `+proj=cart` followed by `+proj=topocentric` corresponds to `EPSG:9837`.

#### 7.2.9.1 Usage

Convert geocentric coordinates to topocentric coordinates, with the topocentric origin specified in geocentric coordinates:

```
echo 3771793.968 140253.342 5124304.349 2020 | \
    cct -d 3 +proj=topocentric +X_0=3652755.3058 +Y_0=319574.6799 +Z_0=5201547.3536

-189013.869    -128642.040     -4220.171     2020.0000
```

Convert geographic coordinates to topocentric coordinates, with the topocentric origin specified in geographic coordi-

nates:

```
echo 2.12955 53.80939444 73 2020 | cct -d 3 +proj=pipeline \
    +step +proj=cart \
    +step +proj=topocentric +lon_0=5 +lat_0=55 +h_0=200

-189013.869    -128642.040     -4220.171     2020.0000
```

#### 7.2.9.2 Parameters

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

##### Topocentric origin described as geocentric coordinates

> **Note**
>
> The below options are mutually exclusive with the ones to express the origin as geographic coordinates.

**+X_0**=<value>

Geocentric X value of the topocentric origin (in metre)

**+Y_0**=<value>

Geocentric Y value of the topocentric origin (in metre)

**+Z_0**=<value>

Geocentric Z value of the topocentric origin (in metre)

##### Topocentric origin described as geographic coordinates

> **Note**
>
> The below options are mutually exclusive with the ones to express the origin as geocentric coordinates.

**+lat_0**=<value>

Latitude of topocentric origin (in degree)

**+lon_0**=<value>

Longitude of topocentric origin (in degree)

**+h_0**=<value>

Ellipsoidal height of topocentric origin (in metre)

*Defaults to 0.0.*

### 7.2.10 Unit conversion

Added in version 5.0.0.

Convert between various distance, angular and time units.

| | |
|---|---|
| **Alias** | unitconvert |
| **Domain** | 2D, 3D or 4D |
| **Input type** | Any |
| **Output type** | Any |

There are many examples of coordinate reference systems that are expressed in other units than the meter. There are also many cases where temporal data has to be translated to different units. The *unitconvert* operation takes care of that.

Many North American systems are defined with coordinates in feet. For example in Vermont:

```
+proj=pipeline \
+step +proj=tmerc +lat_0=42.5 +lon_0=-72.5 +k_0=0.999964286 +x_0=500000.00001016 \
+step +proj=unitconvert +xy_out=us-ft
```

Often when working with GNSS data the timestamps are presented in GPS-weeks, but when the data transformed with the *helmert* operation timestamps are expected to be in units of decimalyears. This can be fixed with *unitconvert*:

```
+proj=pipeline \
+step +proj=unitconvert +t_in=gps_week +t_out=decimalyear \
+step +proj=helmert +epoch=2000.0 +t_obs=2017.5 ...
```

#### 7.2.10.1 Parameters

**+xy_in**=<unit> or <conversion_factor>

Horizontal input units. See *Distance units* and *Angular units* for a list of available units. Default: meter. <*conversion_factor*> is the conversion factor from the input unit to metre for linear units, or to radian for angular units. Default if not given: meters.

**+xy_out**=<unit> or <conversion_factor>

Horizontal output units. See *Distance units* and *Angular units* for a list of available units. Default: meter. <*conversion_factor*> is the conversion factor from the output unit to metre for linear units, or to radian for angular units. Default if not given: meters.

**+z_in**=<unit> or <conversion_factor>

Vertical output units. See *Distance units* and *Angular units* for a list of available units. <*conversion_factor*> is the conversion factor from the input unit to metre for linear units, or to radian for angular units.

**+z_out**=<unit> or <conversion_factor>

Vertical output units. See *Distance units* and *Angular units* for a list of available units. <*conversion_factor*> is the conversion factor from the output unit to metre for linear units, or to radian for angular units.

**+t_in**=<unit>

Temporal input units. See *Time units* for a list of available units.

**+t_out**=<unit>

Temporal output units. See *Time units* for a list of available units.

#### 7.2.10.2 Distance units

In the table below all distance units supported by PROJ are listed. The same list can also be produced on the command line with *proj* or *cs2cs*, by adding the *-lu* flag when calling the utility.

| Label | Name |
|---|---|
| km | Kilometer |
| m | Meter |
| dm | Decimeter |
| cm | Centimeter |
| mm | Millimeter |
| kmi | International Nautical Mile |
| in | International Inch |
| ft | International Foot |
| yd | International Yard |
| mi | International Statute Mile |
| fath | International Fathom |
| ch | International Chain |
| link | International Link |
| us-in | U.S. Surveyor's Inch |
| us-ft | U.S. Surveyor's Foot |
| us-yd | U.S. Surveyor's Yard |
| us-ch | U.S. Surveyor's Chain |
| us-mi | U.S. Surveyor's Statute Mile |
| ind-yd | Indian Yard |
| ind-ft | Indian Foot |
| ind-ch | Indian Chain |

#### 7.2.10.3 Angular units

Added in version 5.2.0.

In the table below all angular units supported by PROJ *unitconvert* are listed.

| Label | Name |
|---|---|
| deg | Degree |
| grad | Grad |
| rad | Radian |

#### 7.2.10.4 Time units

In the table below all time units supported by PROJ are listed.

**Note**

When converting time units from a date-only format (*yyyymmdd*), PROJ assumes a time value of 00:00 midnight. When converting time units to a date-only format, PROJ rounds to the *nearest* date at 00:00 midnight. That is, any time values less than 12:00 noon will round to 00:00 on the same day. Time values greater than or equal to 12:00 noon will round to 00:00 on the following day.

| Label | Name |
|---|---|
| mjd | Modified Julian date |
| decimalyear | Decimal year |
| gps_week | GPS Week |
| yyyymmdd | Date in yyyymmdd format |

## 7.3 Transformations

Transformations coordinate operation in which the two coordinate reference systems are based on different datums.

### 7.3.1 Affine transformation

Added in version 6.0.0.

The affine transformation applies translation and scaling/rotation terms on the x,y,z coordinates, and translation and scaling on the temporal coordinate.

| | |
|---|---|
| **Alias** | affine |
| **Domain** | 4D |
| **Input type** | XYZT |
| **output type** | XYZT |

By default, the parameters are set for an identity transforms. The transformation is reversible unless the determinant of the sji matrix is 0, or *tscale* is 0

This can be used to implement:

- EPSG "Affine parametric transformation" of code 9624 by setting *A0*, *A1*, *A2*, *B0*, *B1*, *B2* EPSG parameters to respectively *xoff*, *s11*, *s12*, *yoff*, *s21*, *s22* PROJ parameters.
- EPSG "Similarity transformation" of code 9621 by setting:
  - xoff to $X_{T0}$
  - yoff to $Y_{T0}$
  - s11 to $M \cos \theta$
  - s12 to $M \sin \theta$
  - s21 to $-M \sin \theta$
  - s22 to $M \cos \theta$

  where:
  - $X_{T0}$ is the first ordinate of the origin point of the source CRS expressed in the target CRS.
  - $Y_{T0}$ is the second ordinate of the origin point of the source CRS expressed in the target CRS.
  - $M$ is the multiplication factor applied to coordinates in the source CRS to obtain the correct scale of the target CRS.
  - $\theta$ is the angle about which the axes of the source CRS need to be rotated to coincide with the axes of the target CRS, counter-clockwise being positive

#### 7.3.1.1 Parameters

### Optional

**+xoff**=<value>
: Offset in X. Default value: 0

**+yoff**=<value>
: Offset in Y. Default value: 0

**+zoff**=<value>
: Offset in Z. Default value: 0

**+toff**=<value>
: Offset in T. Default value: 0

**+s11**=<value>
: Rotation/scaling term. Default value: 1

**+s12**=<value>
: Rotation/scaling term. Default value: 0

**+s13**=<value>
: Rotation/scaling term. Default value: 0

**+s21**=<value>
: Rotation/scaling term. Default value: 0

**+s22**=<value>
: Rotation/scaling term. Default value: 1

**+s23**=<value>
: Rotation/scaling term. Default value: 0

**+s31**=<value>
: Rotation/scaling term. Default value: 0

**+s32**=<value>
: Rotation/scaling term. Default value: 0

**+s33**=<value>
: Rotation/scaling term. Default value: 1

**+tscale**=<value>
: Time scaling term. Default value: 1

### Mathematical description

$$\begin{bmatrix} X \\ Y \\ Z \\ T \end{bmatrix}^{dest} = \begin{bmatrix} xoff \\ yoff \\ zoff \\ toff \end{bmatrix} + \begin{bmatrix} s11 & s12 & s13 & 0 \\ s21 & s22 & s23 & 0 \\ s31 & s32 & s33 & 0 \\ 0 & 0 & 0 & tscale \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ T \end{bmatrix}^{source} \tag{7.1}$$

### 7.3.2 Multi-component time-based deformation model

Added in version 7.1.0.

| | |
|---|---|
| **Alias** | defmodel |
| **Input type** | Geodetic or projected coordinates (horizontal), meters (vertical), decimalyear (temporal) |
| **Output type** | Geodetic or projected coordinates (horizontal), meters (vertical), decimalyear (temporal) |
| **Domain** | 4D |
| **Available forms** | Forward and inverse |

The defmodel transformation can be used to represent most deformation models currently in use, in particular for areas subject to complex deformation, including large scale secular crustal deformation near plate boundaries and vertical deformation due to Glacial Isostatic Adjustment (GIA). These can often be represented by a constant velocity model. Additionally, many areas suffer episodic deformation events such as earthquakes and aseismic slow slip event.

The transformation relies on a "master" JSON file, describing general metadata on the deformation model, its spatial and temporal extent, and listing spatial components whose values are stored in *Geodetic TIFF grids (GTG)*. The valuation of each component is modulated by a time function (constant, step, reverse step, velocity, piecewise, exponential).

All details on the content of this JSON file are given in the Proposal for encoding of a Deformation Model

If input coordinates are given in the geographic domain (resp. projected domain), the output will also be in the geographic domain (resp. projected domain). The domain should be the corresponding to the source_crs metadata of the model.

This transformation is a generalization of the *Kinematic datum shifting utilizing a deformation model* transformation.

#### 7.3.2.1 Parameters

##### Required

**+model**=<filename>

Filename to the JSON master file for the deformation model. This file must validate against the deformation_model.schema.json JSON schema.

#### 7.3.2.2 Example

Transforming a point with the LINZ NZGD2000 deformation model:

```
echo 166.7133850980 -44.5105886020 293.3700 2007.689 |
     cct +proj=defmodel +model=nzgd2000-20180701.json
```

### 7.3.3 Kinematic datum shifting utilizing a deformation model

Added in version 5.0.0.

Perform datum shifts means of a deformation/velocity model.

| | |
|---|---|
| **Alias** | deformation |
| **Input type** | Cartesian coordinates (spatial), decimalyears (temporal). |
| **Output type** | Cartesian coordinates (spatial), decimalyears (temporal). |
| **Domain** | 4D |
| **Input type** | Geodetic coordinates |
| **Output type** | Geodetic coordinates |

The deformation operation is used to adjust coordinates for intraplate deformations. Usually the transformation parameters for regional plate-fixed reference frames such as the ETRS89 does not take intraplate deformation into account. It is assumed that tectonic plate of the region is rigid. Often times this is true, but near the plate boundary and in areas with post-glacial uplift the assumption breaks. Intraplate deformations can be modelled and then applied to the coordinates so that they represent the physical world better. In PROJ this is done with the deformation operation.

Starting with PROJ 7.0, it is recommended to store both the horizontal and vertical velocities in a single GeoTIFF file (*Geodetic TIFF grids (GTG)*). Grid values are expected to be in units of mm/year. Using GDAL for construction of new grids is recommended. Prior to PROJ 7.0, the horizontal grid used to be stored in CTable2 format and the vertical grid in the GTX format. Both grids GDAL both reads and writes both file formats.

More complex deformations can be done with the *Multi-component time-based deformation model* transformation.

#### 7.3.3.1 Example

In [Hakli2016] coordinate transformation including a deformation model is described. The paper describes how coordinates from the global ITRFxx frames are transformed to the local Nordic realisations of ETRS89. Scandinavia is an area with significant post-glacial rebound. The deformations from the post-glacial uplift is not accounted for in the official ETRS89 transformations so in order to get accurate transformations in the Nordic countries it is necessary to apply the deformation model. The transformation from ITRF2008 to the Danish realisation of ETRS89 is in PROJ described as:

```
proj =  pipeline ellps = GRS80
        # ITRF2008@t_obs -> ITRF2000@t_obs
step    init = ITRF2008:ITRF2000
        # ITRF2000@t_obs -> ETRF2000@t_obs
step    proj=helmert t_epoch = 2000.0 convention=position_vector
        x =  0.054  rx =  0.000891 drx =  8.1e-05
        y =  0.051  ry =  0.00539  dry =  0.00049
        z = -0.048  rz = -0.008712 drz = -0.000792
        # ETRF2000@t_obs -> NKG_ETRF00@2000.0
step    proj = deformation t_epoch = 2000.0
        grids = ./eur_nkg_nkgrf03vel_realigned.tif
        inv
        # NKG_ETRF@2000.0 -> ETRF92@2000.0
step    proj=helmert convention=position_vector s = -0.009420e
        x = 0.03863 rx = 0.00617753
        y = 0.147   ry = 5.064e-05
        z = 0.02776 rz = 4.729e-05
        # ETRF92@2000.0 -> ETRF92@1994.704
step    proj = deformation dt = -5.296
        grids = ./eur_nkg_nkgrf03vel_realigned.tif
```

From this we can see that the transformation from ITRF2008 to the Danish realisation of ETRS89 is a combination of Helmert transformations and adjustments with a deformation model. The first use of the deformation operation is:

```
proj = deformation t_epoch = 2000.0 grids = ./eur_nkg_nkgrf03vel_realigned.tif
```

Here we set the central epoch of the transformation, 2000.0. The observation epoch is expected as part of the input coordinate tuple. The deformation model is described by a single grid, specified with *+grids*.

#### 7.3.3.2 Parameters

**+grids**=<list>

Added in version 7.0.0.

Comma-separated list of grids to load. If a grid is prefixed by an @ the grid is considered optional and PROJ will the not complain if the grid is not available.

Grids should be in GeoTIFF format (*Geodetic TIFF grids (GTG)*) with the first 3 components being respectively the easting, northing and up velocities in mm/year. Setting the Description and Unit Type GDAL band metadata items is strongly recommended, so that gdalinfo reports:

```
Band 1 Block=... Type=Float32, ColorInterp=Gray
    Description = east_velocity
    Unit Type: mm/year
Band 2 Block=... Type=Float32, ColorInterp=Undefined
    Description = north_velocity
    Unit Type: mm/year
Band 3 Block=... Type=Float32, ColorInterp=Undefined
    Description = up_velocity
    Unit Type: mm/year
```

**Note**

*+grids* is mutually exclusive with *+xy_grids* and *+z_grids*

**+t_epoch**=<value>

Central epoch of transformation given in decimalyears. Will be used in conjunction with the observation time from the input coordinate to determine $dt$ as used in eq. (7.1) below.

**Note**

*+t_epoch* is mutually exclusive with *+dt*

**+dt**=<value>

Added in version 6.0.0.

$dt$ as used in eq. (7.1) below. Is useful when no observation time is available in the input coordinate or when a deformation for a specific timespan needs to be applied in a transformation. $dt$ is given in units of decimalyears.

**Note**

*+dt* is mutually exclusive with *+t_epoch*

**+xy_grids**=<list>

Deprecated since version 7.0.

Comma-separated list of grids to load. If a grid is prefixed by an @ the grid is considered optional and PROJ will the not complain if the grid is not available.

Grids for the horizontal component of a deformation model is expected to be in CTable2 format.

> **Note**
>
> *+xy_grids* is mutually exclusive with *+grids*

**+z_grids**=<list>

Deprecated since version 7.0.

Comma-separated list of grids to load. If a grid is prefixed by an @ the grid is considered optional and PROJ will the not complain if the grid is not available.

Grids for the vertical component of a deformation model is expected to be in either GTX format.

> **Note**
>
> *+z_grids* is mutually exclusive with *+grids*

#### 7.3.3.3 Mathematical description

Mathematically speaking, application of a deformation model is simple. The deformation model is represented as a grid of velocities in three dimensions. Coordinate corrections are applied in cartesian space. For a given coordinate, $(X, Y, Z)$, velocities $(V_X, V_Y, V_Z)$ can be interpolated from the gridded model. The time span between $t_{obs}$ and $t_c$ determine the magnitude of the coordinate correction as seen in eq. (7.1) below.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_B = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_A + (t_{obs} - t_c) \begin{pmatrix} V_X \\ V_Y \\ V_Z \end{pmatrix} \tag{7.1}$$

Corrections are done in cartesian space.

Coordinates of the gridded model are in ENU (east, north, up) space because it would otherwise require an enormous 3 dimensional grid to handle the corrections in cartesian space. Keeping the correction in lat/long space reduces the complexity of the grid significantly. Consequently though, the input coordinates needs to be converted to lat/long space when searching for corrections in the grid. This is done with the *cart* operation. The converted grid corrections can then be applied to the input coordinates in cartesian space. The conversion from ENU space to cartesian space is done in the following way:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} -\sin\phi\cos\lambda N - \sin\lambda E + \cos\phi\cos\lambda U \\ -\sin\phi\sin\lambda N + \sin\lambda E + \cos\phi\sin\lambda U \\ \cos\phi N + \sin\phi U \end{pmatrix} \tag{7.1}$$

where $\phi$ and $\lambda$ are the latitude and longitude of the coordinate that is searched for in the grid. $(E, N, U)$ are the grid values in ENU-space and $(X, Y, Z)$ are the corrections converted to cartesian space.

#### 7.3.3.4 See also

1. *Behavioural changes from version 5 to 6*

### 7.3.4 Geographic offsets

Added in version 6.0.0.

The Geographic offsets transformation adds an offset to the geographic longitude, latitude coordinates, and an offset to the ellipsoidal height.

| | |
|---|---|
| **Alias** | geogoffset |
| **Domain** | 3D |
| **Input type** | Geodetic coordinates (horizontal), meters (vertical) |
| **output type** | Geodetic coordinates (horizontal), meters (vertical) |

This method is normally only used when low accuracy is tolerated. It is documented as coordinate operation method code 9619 (for geographic 2D) and 9660 (for geographic 3D) in the EPSG dataset ([IOGP2018])

It can also be used to implement the method Geographic2D with Height Offsets (code 9618) by noting that the input vertical component is a gravity-related height and the output vertical component is the ellipsoid height (dh being the geoid undulation).

It can also be used to implement the method Vertical offset (code 9616)

The reverse transformation simply consists in subtracting the offsets.

This method is a conveniency wrapper for the more general *Affine transformation*.

#### 7.3.4.1 Examples

Geographic offset from the old Greek geographic 2D CRS to the newer GGRS87 CRS:

```
+proj=geogoffset +dlon=0.28 +dlat=-5.86
```

Conversion from Tokyo + JSLD69 height to WGS 84:

```
+proj=geogoffset +dlon=-13.97 +dlat=7.94 +dh=26.9
```

Conversion from Baltic 1977 height to Black Sea height:

```
+proj=geogoffset +dh=0.4
```

#### 7.3.4.2 Parameters

##### Optional

**+dlon**=<value>
: Offset in longitude, expressed in arc-second, to add.

**+dlat**=<value>
: Offset in latitude, expressed in arc-second, to add.

**+dh**=<value>
: Offset in height, expressed in meter, to add.

### 7.3.5 General grid shift

Added in version 9.2.0.

Translation of geodetic coordinates using a grid shift.

| | |
|---|---|
| **Alias** | gridshift |
| **Domain** | 2D and 3D |
| **Input type** | Geodetic or projected coordinates (horizontal), meters (vertical) |
| **Output type** | Geodetic or projected coordinates (horizontal), meters (vertical) |

The transformation may apply horizontal geodetic or projected offsetting and/or vertical (ellipsoidal or orthometric height) offsetting, depending on the type of the grid(s).

This is a generalization of the *Horizontal grid shift* and *Vertical grid shift* methods, that may be used in particular for US NADCON5 grids that contain both horizontal geodetic and ellipsoidal height offsets.

> **Note**
>
> Support for grids referenced in a projected CRS has been added in PROJ 9.4.0

#### 7.3.5.1 Example

Transformation of a point from NAD83(NSRS2007) to NAD83(2011) on conterminous USA using a NADCON5 grid that contains horizontal and ellipsoidal height offsets.

```
+proj=gridshift +grids=us_noaa_nadcon5_nad83_2007_nad83_2011_conus.tif
```

##### Parameters

##### Required

**+grids**=<list>

Comma-separated list of grids to load. If a grid is prefixed by an `@` the grid is considered optional and PROJ will the not complain if the grid is not available.

Grids must be in GeoTIFF format (*Geodetic TIFF grids (GTG)*) and have an explicit TYPE metadata item whose value is `HORIZONTAL_OFFSET`, `GEOGRAPHIC_3D_OFFSET`, `ELLIPSOIDAL_HEIGHT_OFFSET` `VERTICAL_OFFSET_GEOGRAPHIC_TO_VERTICAL` or `VERTICAL_OFFSET_VERTICAL_TO_VERTICAL`.

##### Optional

**+interpolation**=bilinear/biquadratic

Default is bilinear, unless the grid contains a `interpolation_method` metadata item specifying the method. Biquadratic is typically used for NADCON5 grids, and is defined in NOAA Technical Memorandum NOS NGS 84 - Biquadratic Interpolation

**+no_z_transform**

If specified, vertical coordinate transformation will be skipped.

### 7.3.6 Helmert transform

The Helmert transformation changes coordinates from one reference frame to another by means of 3-, 4-and 7-parameter shifts, or one of their 6-, 8- and 14-parameter kinematic counterparts.

| | |
|---|---|
| **Alias** | helmert |
| **Domain** | 2D, 3D and 4D |
| **Input type** | Cartesian coordinates (spatial), decimalyears (temporal). |
| **Output type** | Cartesian coordinates (spatial), decimalyears (temporal). |
| **Input type** | Cartesian coordinates |
| **Output type** | Cartesian coordinates |

The Helmert transform, in all its various incarnations, is used to perform reference frame shifts. The transformation operates in cartesian space. It can be used to transform planar coordinates from one datum to another, transform 3D

cartesian coordinates from one static reference frame to another or it can be used to do fully kinematic transformations from global reference frames to local static frames.

All of the parameters described in the table above are marked as optional. This is true as long as at least one parameter is defined in the setup of the transformation. The behavior of the transformation depends on which parameters are used in the setup. For instance, if a rate of change parameter is specified a kinematic version of the transformation is used.

The kinematic transformations require an observation time of the coordinate, as well as a central epoch for the transformation. The latter is usually documented alongside the rest of the transformation parameters for a given transformation. The central epoch is controlled with the parameter *t_epoch*. The observation time is given as part of the coordinate when using PROJ's 4D-functionality.

#### 7.3.6.1 Examples

Transforming coordinates from NAD72 to NAD83 using the 4 parameter 2D Helmert:

```
proj=helmert convention=coordinate_frame x=-9597.3572 y=.6112 \
    s=0.304794780637 theta=-1.244048
```

Simplified transformations from ITRF2008/IGS08 to ETRS89 using 7 parameters:

```
proj=helmert convention=coordinate_frame x=0.67678    y=0.65495   z=-0.52827
            rx=-0.022742 ry=0.012667 rz=0.022704  s=-0.01070
```

Transformation from *ITRF2000* to *ITRF93* using 15 parameters:

```
proj=helmert convention=position_vector
     x=0.0127     y=0.0065     z=-0.0209  s=0.00195
     dx=-0.0029   dy=-0.0002   dz=-0.0006 ds=0.00001
     rx=-0.00039  ry=0.00080   rz=-0.00114
     drx=-0.00011 dry=-0.00019 drz=0.00007
     t_epoch=1988.0
```

#### 7.3.6.2 Parameters

**Note**

All parameters are optional but at least one should be used, otherwise the operation will return the coordinates unchanged.

**+convention**=coordinate_frame/position_vector

Indicates the convention to express the rotational terms when a 3D-Helmert / 7-parameter more transform is involved. As soon as a rotational parameter is specified (one of `rx`, `ry`, `rz`, `drx`, `dry`, `drz`), `convention` is required.

The two conventions are equally popular and a frequent source of confusion. The coordinate frame convention is also described as an clockwise rotation of the coordinate frame. It corresponds to EPSG method code 1032 (in the geocentric domain) or 9607 (in the geographic domain) The position vector convention is also described as an anticlockwise (counter-clockwise) rotation of the coordinate frame. It corresponds to as EPSG method code 1033 (in the geocentric domain) or 9606 (in the geographic domain).

This parameter is ignored when only a 3-parameter (translation terms only: `x`, `y`, `z`) , 4-parameter (3-parameter and `theta`) or 6-parameter (3-parameter and their derivative terms) is used.

The result obtained with parameters specified in a given convention can be obtained in the other convention by negating the rotational parameters (`rx`, `ry`, `rz`, `drx`, `dry`, `drz`)

> **Note**
>
> This parameter obsoletes `transpose` which was present in PROJ 5.0 and 5.1, and is forbidden starting with PROJ 5.2

**+x**=<value>
: Translation of the x-axis given in meters.

**+y**=<value>
: Translation of the y-axis given in meters.

**+z**=<value>
: Translation of the z-axis given in meters.

**+s**=<value>
: Scale factor given in ppm.

**+rx**=<value>
: X-axis rotation in the 3D Helmert given arc seconds.

**+ry**=<value>
: Y-axis rotation in the 3D Helmert given in arc seconds.

**+rz**=<value>
: Z-axis rotation in the 3D Helmert given in arc seconds.

**+theta**=<value>
: Rotation angle in the 2D Helmert given in arc seconds.

**+dx**=<value>
: Translation rate of the x-axis given in m/year.

**+dy**=<value>
: Translation rate of the y-axis given in m/year.

**+dz**=<value>
: Translation rate of the z-axis given in m/year.

**+ds**=<value>
: Scale rate factor given in ppm/year.

**+drx**=<value>
: Rotation rate of the x-axis given in arc seconds/year.

**+dry**=<value>
: Rotation rate of the y-axis given in arc seconds/year.

**+drz**=<value>
: Rotation rate of the y-axis given in arc seconds/year.

**+t_epoch**=<value>
: Central epoch of transformation given in decimalyear. Only used spatiotemporal transformations.

**+exact**
: Use exact transformation equations.

  See (7.6)

#### 7.3.6.3 Mathematical description

In the notation used below, $\hat{P}$ is the rate of change of a given transformation parameter $P$. $\dot{P}$ is the kinematically adjusted version of $P$, described by

$$\dot{P} = P + \hat{P}\,(t - t_{central}) \tag{7.1}$$

where $t$ is the observation time of the coordinate and $t_{central}$ is the central epoch of the transformation. Equation (7.1) can be used to propagate all transformation parameters in time.

Superscripts of vectors denote the reference frame the coordinates in the vector belong to.

### 2D Helmert

The simplest version of the Helmert transform is the 2D case. In the 2-dimensional case only the horizontal coordinates are changed. The coordinates can be translated, rotated and scaled. Translation is controlled with the *x* and *y* parameters. The rotation is determined by *theta* and the scale is controlled with the *s* parameters.

**Note**

The scaling parameter *s* is unitless for the 2D Helmert, as opposed to the 3D version where the scaling parameter is given in units of ppm.

Mathematically the 2D Helmert is described as:

$$\begin{bmatrix} X \\ Y \end{bmatrix}^B = \begin{bmatrix} T_x \\ T_y \end{bmatrix} + s \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}^A \tag{7.2}$$

(7.2) can be extended to a time-varying kinematic version by adjusting the parameters with (7.1) to (7.2), which yields the kinematic 2D Helmert transform:

$$\begin{bmatrix} X \\ Y \end{bmatrix}^B = \begin{bmatrix} \dot{T}_x \\ \dot{T}_y \end{bmatrix} + s(t) \begin{bmatrix} \cos\dot{\theta} & \sin\dot{\theta} \\ -\sin\ \dot{\theta} & \cos\dot{\theta} \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}^A \tag{7.2}$$

All parameters in (7.2) are determined by the use of (7.1), which applies the rate of change to each individual parameter for a given timespan between $t$ and $t_{central}$.

### 3D Helmert

The general form of the 3D Helmert is

$$V^B = T + \left(1 + s \times 10^{-6}\right) \mathbf{R} V^A \tag{7.2}$$

Where $T$ is a vector consisting of the three translation parameters, $s$ is the scaling factor and $\mathbf{R}$ is a rotation matrix. $V^A$ and $V^B$ are coordinate vectors, with $V^A$ being the input coordinate and $V^B$ is the output coordinate.

In the *Position Vector* convention, we define $R_x = radians\,(rx)$, $R_z = radians\,(ry)$ and $R_z = radians\,(rz)$

In the *Coordinate Frame* convention, $R_x = -radians\,(rx)$, $R_z = -radians\,(ry)$ and $R_z = -radians\,(rz)$

The rotation matrix is composed of three rotation matrices, one for each axis.

$$\mathbf{R}_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos R_x & -\sin R_x \\ 0 & \sin R_x & \cos R_x \end{bmatrix} \tag{7.2}$$

$$\mathbf{R}_Y = \begin{bmatrix} \cos R_y & 0 & \sin R_y \\ 0 & 1 & 0 \\ -\sin R_y & 0 & \cos R_y \end{bmatrix} \tag{7.3}$$

$$\mathbf{R}_Z = \begin{bmatrix} \cos R_z & -\sin R_z & 0 \\ \sin R_z & \cos R_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7.4}$$

The three rotation matrices can be combined in one:

$$\mathbf{R} = \mathbf{R_Z R_Y R_X} \tag{7.5}$$

For $\mathbf{R}$, this yields:

$$\begin{bmatrix} \cos R_y \cos R_z & -\cos R_x \sin R_z + & \sin R_x \sin R_z + \\ & \sin R_x \sin R_y \cos R_z & \cos R_x \sin R_y \cos R_z \\ \cos R_y \sin R_z & \cos R_x \cos R_z + & -\sin R_x \cos R_z + \\ & \sin R_x \sin R_y \sin R_z & \cos R_x \sin R_y \sin R_z \\ -\sin R_y & \sin R_x \cos R_y & \cos R_x \cos R_y \end{bmatrix} \tag{7.6}$$

Using the small angle approximation the rotation matrix can be simplified to

$$\mathbf{R} = \begin{bmatrix} 1 & -R_z & R_y \\ Rz & 1 & -R_x \\ -Ry & R_x & 1 \end{bmatrix} \tag{7.7}$$

Which allow us to express the most common version of the Helmert transform, using the approximated rotation matrix:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} + \left(1 + s \times 10^{-6}\right) \begin{bmatrix} 1 & -R_z & R_y \\ Rz & 1 & -R_x \\ -Ry & R_x & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{7.7}$$

If the rotation matrix is transposed, or the sign of the rotation terms negated, the rotational part of the transformation is effectively reversed. This is what happens when switching between the 2 conventions `position_vector` and `coordinate_frame`

Applying (7.1) we get the kinematic version of the approximated 3D Helmert:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} \dot{T_x} \\ \dot{T_y} \\ \dot{T_z} \end{bmatrix} + \left(1 + \dot{s} \times 10^{-6}\right) \begin{bmatrix} 1 & -\dot{R_z} & \dot{R_y} \\ \dot{R_z} & 1 & -\dot{R_x} \\ -\dot{R_y} & \dot{R_x} & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{7.7}$$

The Helmert transformation can be applied without using the rotation parameters, in which case it becomes a simple translation of the origin of the coordinate system. When using the Helmert in this version equation (7.2) simplifies to:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} + \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{7.7}$$

That after application of (7.1) has the following kinematic counterpart:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} \dot{T_x} \\ \dot{T_y} \\ \dot{T_z} \end{bmatrix} + \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{7.7}$$

### 7.3.7 Horner polynomial evaluation

Added in version 5.0.0.

Changed in version 9.1.0: Iterative polynormal inversion

| | |
|---|---|
| **Alias** | horner |
| **Domain** | 2D and 3D |
| **Input type** | Geodetic and projected coordinates |
| **Output type** | Geodetic and projected coordinates |

The Horner polynomial evaluation scheme is used for transformations between reference frames where one or both are inhomogeneous or internally distorted. This will typically be reference frames created before the introduction of space geodetic techniques such as GPS.

Horner polynomials, or Multiple Regression Equations as they are also known as, have their strength in being able to create complicated mappings between coordinate reference frames while still being lightweight in both computational cost and disk space used.

PROJ implements two versions of the Horner evaluation scheme: Real and complex polynomial evaluation. Below both are briefly described. For more details consult [Ruffhead2016] and [IOGP2018].

The polynomial evaluation in real number space is defined by the following equations:

$$\begin{aligned} \Delta X &= \sum_{i,j} u_{i,j} U^i V^j \\ \Delta Y &= \sum_{i,j} v_{i,j} U^i V^j \end{aligned} \tag{7.7}$$

where

$$\begin{aligned} U &= X_{in} - X_{origin} \\ V &= Y_{in} - Y_{origin} \end{aligned} \tag{7.8}$$

and $u_{i,j}$ and $v_{i,j}$ are coefficients that make up the polynomial.

The order of coefficients $u_{i,j}$ is (example for degree 3):

| | $\mathbf{V^0}$ | $\mathbf{V^1}$ | $\mathbf{V^2}$ | $\mathbf{V^3}$ |
|---|---|---|---|---|
| $\mathbf{U^0}$ | 1st | 5th | 8rd | 10th |
| $\mathbf{U^1}$ | 2nd | 6th | 9th | – |
| $\mathbf{U^2}$ | 3rd | 7th | – | – |
| $\mathbf{U^3}$ | 4th | – | – | – |

The order of coefficients $v_{i,j}$ is (example for degree 3):

| | $\mathbf{V^0}$ | $\mathbf{V^1}$ | $\mathbf{V^2}$ | $\mathbf{V^3}$ |
|---|---|---|---|---|
| $\mathbf{U^0}$ | 1st | 2nd | 3rd | 4th |
| $\mathbf{U^1}$ | 5th | 6th | 7th | – |
| $\mathbf{U^2}$ | 8th | 9th | – | – |
| $\mathbf{U^3}$ | 10th | – | – | – |

The final coordinates are determined as

$$\begin{aligned} X_{out} &= X_{in} + \Delta X \\ Y_{out} &= Y_{in} + \Delta Y \end{aligned} \tag{7.9}$$

The inverse transform is the same as the above but requires a different set of coefficients. If only the forward set of coefficients and origin is known the inverse transform can be done by iteratively solving a system of equations. By

writing (7.7) as:

$$\begin{bmatrix}\Delta X\\ \Delta Y\end{bmatrix} = \begin{bmatrix}u_{0,0}\\ v_{0,0}\end{bmatrix} + \begin{bmatrix}u_{1,0}+u_{2,0}U+... & u_{0,1}+u_{1,1}U+u_{0,2}V+...\\ v_{0,1}+v_{1,1}V+v_{0,2}U+... & v_{1,0}+v_{2,0}V\end{bmatrix}\begin{bmatrix}U\\ V\end{bmatrix}$$

$$\begin{bmatrix}\Delta X\\ \Delta Y\end{bmatrix} = \begin{bmatrix}u_{0,0}\\ v_{0,0}\end{bmatrix} + \begin{bmatrix}MA & MB\\ MC & MD\end{bmatrix}\begin{bmatrix}U\\ V\end{bmatrix}$$

$$\begin{bmatrix}U\\ V\end{bmatrix} = \begin{bmatrix}MA & MB\\ MC & MD\end{bmatrix}^{-1}\begin{bmatrix}\Delta X-u_{0,0}\\ \Delta Y-v_{0,0}\end{bmatrix}$$

We can iteratively solve with initial values of $U=0$ and $V=0$ and find $U$ and $V$.

**Note**

This iterative inverse transformation is a more general solution to *reversible polynormials of degree n* as presented in [IOGP2019]. These can provide a satisfactory solution in a single step when certain conditions are met.

Evaluation of the complex polynomials are defined by the following equations:

$$\Delta X + i\Delta Y = \sum_{j=1}^{n}(c_{2j-1}+ic_{2j})(U+iV)^j \tag{7.10}$$

Where $n$ is the degree of the polynomial. $U$ and $V$ are defined as in (7.8) and the resulting coordinates are again determined by (7.9). Complex polynomials can be solved iteratively similar to real polynomials.

#### 7.3.7.1 Examples

Mapping between Danish TC32 and ETRS89/UTM zone 32 using polynomials in real number space:

```
+proj=horner
+ellps=intl
+range=500000
+fwd_origin=877605.269066,6125810.306769
+inv_origin=877605.760036,6125811.281773
+deg=4
+fwd_v=6.1258112678e+06,9.9999971567e-01,1.5372750011e-10,5.9300860915e-15,2.2609497633e-
↪19,4.3188227445e-05,2.8225130416e-10,7.8740007114e-16,-1.7453997279e-19,1.6877465415e-
↪10,-1.1234649773e-14,-1.7042333358e-18,-7.9303467953e-15,-5.2906832535e-19,3.
↪9984284847e-19
+fwd_u=8.7760574982e+05,9.9999752475e-01,2.8817299305e-10,5.5641310680e-15,-1.
↪5544700949e-18,-4.1357045890e-05,4.2106213519e-11,2.8525551629e-14,-1.9107771273e-18,3.
↪3615590093e-10,2.4380247154e-14,-2.0241230315e-18,1.2429019719e-15,5.3886155968e-19,-1.
↪0167505000e-18
+inv_v=6.1258103208e+06,1.0000002826e+00,-1.5372762184e-10,-5.9304261011e-15,-2.
↪2612705361e-19,-4.3188331419e-05,-2.8225549995e-10,-7.8529116371e-16,1.7476576773e-19,-
↪1.6875687989e-10,1.1236475299e-14,1.7042518057e-18,7.9300735257e-15,5.2881862699e-19,-
↪3.9990736798e-19
+inv_u=8.7760527928e+05,1.0000024735e+00,-2.8817540032e-10,-5.5627059451e-15,1.
↪5543637570e-18,4.1357152105e-05,-4.2114813612e-11,-2.8523713454e-14,1.9109017837e-18,-
↪3.3616407783e-10,-2.4382678126e-14,2.0245020199e-18,-1.2441377565e-15,-5.3885232238e-
↪19,1.0167203661e-18
```

Mapping between Danish System Storebælt and ETRS89/UTM zone 32 using complex polynomials:

```
+proj=horner
+ellps=intl
+range=500000
+fwd_origin=4.94690026817276e+05,6.13342113183056e+06
+inv_origin=6.19480258923588e+05,6.13258568148837e+06
+deg=3
+fwd_c=6.13258562111350e+06,6.19480105709997e+05,9.99378966275206e-01,-2.82153291753490e-
↪02,-2.27089979140026e-10,-1.77019590701470e-09,1.08522286274070e-14,2.11430298751604e-
↪15
+inv_c=6.13342118787027e+06,4.94690181709311e+05,9.99824464710368e-01,2.82279070814774e-
↪02,7.66123542220864e-11,1.78425334628927e-09,-1.05584823306400e-14,-3.32554258683744e-
↪15
```

#### 7.3.7.2 Parameters

Setting up Horner polynomials requires many coefficients being explicitly written, even for polynomials of low degree. For this reason it is recommended to store the polynomial definitions in an *init file* for easier writing and reuse.

##### Required

Below is a list of required parameters that can be set for the Horner polynomial transformation. As stated above, the transformation takes to forms, either using real or complex polynomials. These are divided into separate sections below. Parameters from the two sections are mutually exclusive, that is parameters describing real and complex polynomials can't be mixed.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+deg**=<value>

Degree of polynomial

**+fwd_origin**=<northing,easting>

Coordinate of origin for the forward mapping

##### Real polynomials

The following parameters has to be set if the transformation consists of polynomials in real space. Each parameter takes a comma-separated list of coefficients. The number of coefficients is governed by the degree, $d$, of the polynomial:

$$N = \frac{(d+1)(d+2)}{2}$$

**+fwd_u**=<u_00,u_10,...,u_ij,..,u_nn>

Coefficients for the forward transformation i.e. latitude to northing as described in (7.7).

**+fwd_v**=<v_00,v_01,...,v_ij,..,v_nn>

Coefficients for the forward transformation i.e. longitude to easting as described in (7.7).

### Complex polynomials

The following parameters has to be set if the transformation consists of polynomials in complex space. Each parameter takes a comma-separated list of coefficients. The number of coefficients is governed by the degree, $d$, of the polynomial:

$$N = 2d + 2$$

**+fwd_c**=<c_1,c_2,...,c_N>

Coefficients for the complex forward transformation as described in (7.10).

### Optional

**+inv_origin**=<northing,easting>

Changed in version 9.1.0.

Coordinate of origin for the inverse mapping. Without this option iterative polynomial evaluation is used for the inverse transformation.

**+inv_u**=<u_00,u_10,...,u_ij,..,u_nn>

Changed in version 9.1.0.

Coefficients for the inverse transformation i.e. latitude to northing as described in (7.7). Only applies for real polynomials. Without this option iterative polynomial evaluation is used for the inverse transformation.

**+inv_v**=<v_00,v_01,...,v_ij,..,v_nn>

Changed in version 9.1.0.

Coefficients for the inverse transformation i.e. longitude to easting as described in (7.7). Only applies for real polynomials. Without this option iterative polynomial evaluation is used for the inverse transformation.

**+inv_c**=<c_1,c_2,...,c_N>

Changed in version 9.1.0.

Coefficients for the complex inverse transformation as described in (7.10). Only applies for complex polynomials. Without this option iterative polynomial evaluation is used for the inverse transformation.

**+range**=<value>

Radius of the region of validity.

**+uneg**

Express latitude as southing. Only applies for complex polynomials.

**+vneg**

Express longitude as westing. Only applies for complex polynomials.

**+inv_tolerance**=<value>

Added in version 9.1.0.

Only applies to cases of iterative inversion. The procedure converges to the correct results with each step. Iteration stops when the result differs from the previous calculated result by less than <value>. <value> should be the same units as $U$ and $V$ of (7.8)

*Defaults to 0.001.*

#### 7.3.7.3 Further reading

1. Wikipedia

### 7.3.8 Molodensky transform

Added in version 5.0.0.

The Molodensky transformation resembles a *Helmert transform* with zero rotations and a scale of unity, but converts directly from geodetic coordinates to geodetic coordinates, without the intermediate shifts to and from cartesian geocentric coordinates, associated with the Helmert transformation. The Molodensky transformation is simple to implement and to parametrize, requiring only the 3 shifts between the input and output frame, and the corresponding differences between the semimajor axes and flattening parameters of the reference ellipsoids. Due to its algorithmic simplicity, it was popular prior to the ubiquity of digital computers. Today, it is mostly interesting for historical reasons, but nevertheless indispensable due to the large amount of data that has already been transformed that way [EversKnudsen2017].

| | |
|---|---|
| **Alias** | molodensky |
| **Domain** | 3D |
| **Input type** | Geodetic coordinates (horizontal), meters (vertical) |
| **output type** | Geodetic coordinates (horizontal), meters (vertical) |

The Molodensky transform can be used to perform a datum shift from coordinate $(\phi_1, \lambda_1, h_1)$ to $(\phi_2, \lambda_2, h_2)$ where the two coordinates are referenced to different ellipsoids. This is based on three assumptions:

1. The cartesian axes, $X, Y, Z$, of the two ellipsoids are parallel.
2. The offset, $\delta X, \delta Y, \delta Z$, between the two ellipsoid are known.
3. The characteristics of the two ellipsoids, expressed as the difference in semimajor axis ($\delta a$) and flattening ($\delta f$), are known.

The Molodensky transform is mostly used for transforming between old systems dating back to the time before computers. The advantage of the Molodensky transform is that it is fairly simple to compute by hand. The ease of computation come at the cost of limited accuracy.

A derivation of the mathematical formulas for the Molodensky transform can be found in [Deakin2004].

#### 7.3.8.1 Examples

The abridged Molodensky:

```
proj=molodensky a=6378160 rf=298.25 da=-23 df=-8.120449e-8  dx=-134 dy=-48 dz=149␣
↪abridged
```

The same transformation using the standard Molodensky:

```
proj=molodensky a=6378160 rf=298.25 da=-23 df=-8.120449e-8  dx=-134 dy=-48 dz=149
```

#### 7.3.8.2 Parameters

##### Required

**+da**=<value>
: Difference in semimajor axis of the defining ellipsoids.

**+df**=<value>
: Difference in flattening of the defining ellipsoids.

**+dx**=<value>
: Offset of the X-axes of the defining ellipsoids.

**+dy**=<value>
: Offset of the Y-axes of the defining ellipsoids.

**+dz**=<value>
: Offset of the Z-axes of the defining ellipsoids.

**+ellps**=<value>
: The name of a built-in ellipsoid definition.

  See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

  *Defaults to "GRS80".*

#### Optional

**+abridged**
: Use the abridged version of the Molodensky transform.

### 7.3.9 Molodensky-Badekas transform

Added in version 6.0.0.

The Molodensky-Badekas transformation changes coordinates from one reference frame to another by means of a 10-parameter shift.

> **Note**
>
> It should not be confused with the *Molodensky transform* transform which operates directly in the geodetic coordinates. Molodensky-Badekas can rather be seen as a variation of *Helmert transform*

| | |
|---|---|
| **Alias** | molobadekas |
| **Domain** | 3D |
| **Input type** | Cartesian coordinates |
| **Output type** | Cartesian coordinates |

The Molodensky-Badekas transformation is a variation of the *Helmert transform* where the rotational terms are not directly applied to the ECEF coordinates, but on cartesian coordinates relative to a reference point (usually close to Earth surface, and to the area of use of the transformation). When `px = py = pz = 0`, this is equivalent to a 7-parameter Helmert transformation.

#### 7.3.9.1 Example

Transforming coordinates from La Canoa to REGVEN:

```
proj=molobadekas convention=coordinate_frame
       x=-270.933 y=115.599 z=-360.226 rx=-5.266 ry=-1.238 rz=2.381
       s=-5.109 px=2464351.59 py=-5783466.61 pz=974809.81
```

#### 7.3.9.2 Parameters

> **Note**
>
> All parameters (except convention) are optional but at least one should be used, otherwise the operation will return the coordinates unchanged.

**+convention**=coordinate_frame/position_vector

Indicates the convention to express the rotational terms when a 3D-Helmert / 7-parameter more transform is involved.

The two conventions are equally popular and a frequent source of confusion. The coordinate frame convention is also described as an clockwise rotation of the coordinate frame. It corresponds to EPSG method code 1034 (in the geocentric domain) or 9636 (in the geographic domain) The position vector convention is also described as an anticlockwise (counter-clockwise) rotation of the coordinate frame. It corresponds to as EPSG method code 1061 (in the geocentric domain) or 1063 (in the geographic domain).

The result obtained with parameters specified in a given convention can be obtained in the other convention by negating the rotational parameters (**rx**, **ry**, **rz**)

**+x**=<value>

Translation of the x-axis given in meters.

**+y**=<value>

Translation of the y-axis given in meters.

**+z**=<value>

Translation of the z-axis given in meters.

**+s**=<value>

Scale factor given in ppm.

**+rx**=<value>

X-axis rotation given arc seconds.

**+ry**=<value>

Y-axis rotation given in arc seconds.

**+rz**=<value>

Z-axis rotation given in arc seconds.

**+px**=<value>

Coordinate along the x-axis of the reference point given in meters.

**+py**=<value>

Coordinate along the y-axis of the reference point given in meters.

**+pz**=<value>

Coordinate along the z-axis of the reference point given in meters.

#### 7.3.9.3 Mathematical description

In the *Position Vector* convention, we define $R_x = radians\,(rx)$, $R_z = radians\,(ry)$ and $R_z = radians\,(rz)$

In the *Coordinate Frame* convention, $R_x = -radians\,(rx)$, $R_z = -radians\,(ry)$ and $R_z = -radians\,(rz)$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^{output} = \begin{bmatrix} T_x + P_x \\ T_y + P_y \\ T_z + P_z \end{bmatrix} + \left(1 + s \times 10^{-6}\right) \begin{bmatrix} 1 & -R_z & R_y \\ Rz & 1 & -R_x \\ -Ry & R_x & 1 \end{bmatrix} \begin{bmatrix} X^{input} - P_x \\ Y^{input} - P_y \\ Z^{input} - P_z \end{bmatrix} \tag{7.11}$$

### 7.3.10 Horizontal grid shift

Added in version 5.0.0.

Change of horizontal datum by grid shift.

| | |
|---|---|
| **Alias** | hgridshift |
| **Domain** | 2D, 3D and 4D |
| **Input type** | Geodetic coordinates (horizontal), meters (vertical), decimalyear (temporal) |
| **Output type** | Geodetic coordinates (horizontal), meters (vertical), decimalyear (temporal) |

The horizontal grid shift is done by offsetting the planar input coordinates by a specific amount determined by the loaded grids. The simplest use case of the horizontal grid shift is applying a single grid:

```
+proj=hgridshift +grids=nzgr2kgrid0005.gsb
```

More than one grid can be loaded at the same time, for instance in case the dataset needs to be transformed spans several countries. In this example grids of the continental US, Alaska and Canada is loaded at the same time:

```
+proj=hgridshift +grids=@conus,@alaska,@ntv2_0.gsb,@ntv_can.dat
```

The `@` in the above example states that the grid is optional, in case the grid is not found in the PROJ search path. The list of grids is prioritized so that grids in the start of the list takes precedence over the grids in the back of the list.

PROJ supports GeoTIFF (*Geodetic TIFF grids (GTG)*), CTable2 and NTv2 files for horizontal grid corrections. Details about all three formats can be found in the GDAL documentation and/or driver source code. GDAL reads and writes all three formats. Using GDAL for construction of new grids is recommended.

To apply as well ellipsoidal height differences sometimes present in some grids (such as US NADCON5 grids), use the *General grid shift* method.

#### 7.3.10.1 Temporal gridshifting

Added in version 5.1.0.

By initializing the horizontal gridshift operation with a central epoch, it can be used as a step function applying the grid offsets only if a coordinate is transformed from an epoch before grids central epoch to an epoch after. This is handy in transformations where it is necessary to handle deformations caused by seismic activity.

The central epoch of the grid is controlled with `+t_epoch` and the final epoch of the coordinate is set with `+t_final`. The observation epoch of the coordinate is part of the coordinate tuple.

Suppose we want to model the deformation of the 2008 earthquake in Iceland in a transformation of data from 2005 to 2009:

```
echo 63.992 -21.014 10.0 2005.0 |
    cct +proj=hgridshift +grids=iceland2008.gsb \
    +t_epoch=2008.4071 +t_final=2009.0
63.9920021 -21.0140013 10.0 2005.0
```

**Note**

The timestamp of the resulting coordinate is still 2005.0. The observation time is always kept unchanged as it would otherwise be impossible to do the inverse transformation.

Temporal gridshifting is especially powerful in transformation pipelines where several gridshifts can be chained together, effectively acting as a series of step functions that can be applied to a coordinate that is propagated through time. In the following example we establish a pipeline that allows transformation of coordinates from any given epoch up until the current date, applying only those gridshifts that have central epochs between the observation epoch and the final epoch:

```
+proj=pipeline +t_final=now
+step +proj=hgridshift +grids=earthquake_1.gsb +t_epoch=2010.421
+step +proj=hgridshift +grids=earthquake_2.gsb +t_epoch=2013.853
+step +proj=hgridshift +grids=earthquake_3.gsb +t_epoch=2017.713
```

**Note**

The special epoch *now* is used when specifying the final epoch with `+t_final`. This results in coordinates being transformed to the current date. Additionally, `+t_final` is used as a *global pipeline parameter*, which means that it is applied to all the steps in the pipeline.

In the above transformation, a coordinate with observation epoch 2009.32 would be subject to all three gridshift steps in the pipeline. A coordinate with observation epoch 2014.12 would only by offset by the last step in the pipeline.

#### 7.3.10.2 Parameters

##### Required

**+grids**=<list>

Comma-separated list of grids to load. If a grid is prefixed by an @ the grid is considered optional and PROJ will the not complain if the grid is not available.

Grids are expected to be in GeoTIFF (*Geodetic TIFF grids (GTG)*) or NTv2 format (or CTable2 for legacy grids)

##### Optional

**+t_epoch**=<time>

Central epoch of the transformation.

Added in version 5.1.0.

**+t_final**=<time>

Final epoch that the coordinate will be propagated to after transformation. The special epoch *now* can be used instead of writing a specific period in time. When *now* is used, it is replaced internally with the epoch of the transformation. This means that the resulting coordinate will be slightly different if carried out again at a later date.

Added in version 5.1.0.

### 7.3.11 Triangulated Irregular Network based transformation

Added in version 7.2.0.

| | |
|---|---|
| **Alias** | tinshift |
| **Input type** | Projected or geographic coordinates (horizontal), meters (vertical) |
| **Output type** | Projected or geographic coordinates (horizontal), meters (vertical) |
| **Domain** | 2D or 3D |
| **Available forms** | Forward and inverse |

The `tinshift` transformation takes one mandatory argument, `file`, that points to a JSON file, which contains the triangulation and associated metadata. Input and output coordinates must be geographic or projected coordinates. Depending on the content of the JSON file, horizontal, vertical or both components of the coordinates may be transformed.

The transformation is invertible, with the same computational complexity than the forward transformation.

#### 7.3.11.1 Parameters

##### Required

**+file**=<filename>

Filename to the JSON file for the TIN.

#### 7.3.11.2 Example

Transforming a point with the Finland EPSG:2393 ("KKJ / Finland Uniform Coordinate System") projected CRS to EPSG:3067 ("ETRS89 / TM35FIN(E,N)")

```
$ echo 3210000.0000 6700000.0000 0 2020 | cct +proj=tinshift +file=./triangulation_kkj.
↪json

209948.3217     6697187.0009    0.0000     2020
```

##### Algorithm

Internally, `tinshift` ingest the whole file into memory. It is considered that triangulation should be small enough for that.

When a point is transformed, one must find the triangle into which it falls into. Instead of iterating over all triangles, we build a in-memory quadtree to speed-up the identification of candidates triangles.

To determine if a point falls into a triangle, one computes its 3 barycentric coordinates from its projected coordinates, $\lambda_i$ for $i = 1, 2, 3$. They are real values (in the [0,1] range for a point inside the triangle), giving the weight of each of the 3 vertices of the triangles.

Once those weights are known, interpolating the target horizontal coordinate is a matter of doing the linear combination of those weights with the target horizontal coordinates at the 3 vertices of the triangle ($Xt_i$ and $Yt_i$):

$$X_{target} = Xt_1 * \lambda_1 + Xt_2 * \lambda_2 + Xt_3 * \lambda_3$$
$$Y_{target} = Yt_1 * \lambda_1 + Yt_2 * \lambda_2 + Yt_3 * \lambda_3$$

This interpolation is exact at the vertices of the triangulation, and has linear properties inside each triangle. It is completely equivalent to other formulations of triangular interpolation, such as

$$X_{target} = A + X_{source} * B + Y_{source} * C$$
$$Y_{target} = D + X_{source} * E + Y_{source} * F$$

where the A, B, C, D, E, F constants (for a given triangle) are found by solving the 2 systems of 3 linear equations, constraint by the source and target coordinate pairs of the 3 vertices of the triangle:

$$Xt_i = A + Xs_i * B + Ys_i * C$$
$$Yt_i = D + Xs_i * E + Ys_i * F$$

Similarly for a vertical coordinate transformation, where $Zoff_i$ is the vertical offset at each vertex of the triangle:

$$Z_{target} = Z_{source} + Zoff_1 * \lambda_1 + Zoff_2 * \lambda_2 + Zoff_3 * \lambda_3$$

### Constraints on the triangulation

No check is done on the consistence of the triangulation. It is highly recommended that triangles do not overlap each other (when considering the source coordinates or the forward transformation, or the target coordinates for the inverse transformation), otherwise which triangle will be selected is unspecified. Besides that, the triangulation does not need to have particular properties (like being a Delaunay triangulation)

### File format

The triangulation is stored in a text-based format, using JSON as a serialization.

Below a minimal example, from the KKJ to ETRS89 transformation, with just a single triangle:

```
{
  "file_type": "triangulation_file",
  "format_version": "1.0",
  "name": "Name",
  "version": "Version",
  "publication_date": "2018-07-01T00:00:00Z",
  "license": "Creative Commons Attribution 4.0 International",
  "description": "Test triangulation",
  "authority": {
    "name": "Authority name",
    "url": "http://example.com",
    "address": "Address",
    "email": "test@example.com"
  },
  "links": [
    {
      "href": "https://example.com/about.html",
      "rel": "about",
      "type": "text/html",
      "title": "About"
    },
    {
      "href": "https://example.com/download",
      "rel": "source",
      "type": "application/zip",
      "title": "Authoritative source"
    },
    {
      "href": "https://creativecommons.org/licenses/by/4.0/",
      "rel": "license",
      "type": "text/html",
```

(continues on next page)

(continued from previous page)

```
      "title": "Creative Commons Attribution 4.0 International license"
    },
    {
      "href": "https://example.com/metadata.xml",
      "rel": "metadata",
      "type": "application/xml",
      "title": " ISO 19115 XML encoded metadata regarding the deformation model"
    }
  ],
  "transformed_components": [ "horizontal" ],
  "vertices_columns": [ "source_x", "source_y", "target_x", "target_y" ],
  "triangles_columns": [ "idx_vertex1", "idx_vertex2", "idx_vertex3" ],
  "vertices": [ [2,49,2.1,49.1], [3,50,3.1,50.1], [2, 50, 2.1,50.1] ],
  "triangles": [ [0, 1, 2] ]
}
```

So after the generic metadata, we define the input and output CRS (informative only), and that the transformation affects horizontal components of coordinates. We name the columns of the `vertices` and `triangles` arrays. We defined the source and target coordinates of each vertex, and define a triangle by referring to the index of its vertices in the `vertices` array.

More formally, the specific items for the triangulation file are:

**input_crs**
: String identifying the CRS of source coordinates in the vertices. Typically `EPSG:XXXX`. If the transformation is for vertical component, this should be the code for a compound CRS (can be EPSG:XXXX+YYYY where XXXX is the code of the horizontal CRS and YYYY the code of the vertical CRS). For example, for the KKJ->ETRS89 transformation, this is EPSG:2393 (`KKJ / Finland Uniform Coordinate System`). The input coordinates are assumed to be passed in the "normalized for visualisation" / "GIS friendly" order, that is longitude, latitude for geographic coordinates and easting, northing for projected coordinates.

**output_crs**
: String identifying the CRS of target coordinates in the vertices. Typically `EPSG:XXXX`. If the transformation is for vertical component, this should be the code for a compound CRS (can be EPSG:XXXX+YYYY where XXXX is the code of the horizontal CRS and YYYY the code of the vertical CRS). For example, for the KKJ->ETRS89 transformation, this is EPSG:3067 ("ETRS89 / TM35FIN(E,N)"). The output coordinates will be returned in the "normalized for visualisation" / "GIS friendly" order, that is longitude, latitude for geographic coordinates and easting, northing for projected coordinates.

**transformed_components**
: Array which may contain one or two strings: "horizontal" when horizontal components of the coordinates are transformed and/or "vertical" when the vertical component is transformed.

**fallback_strategy**
: String identifying how to treat points that do not fall into any of the specified triangles. This item is available for `format_version` >= 1.1. Possible values are `none`, `nearest_side` and `nearest_centroid`. The default is `none` and signifies, that points that fall outside the specified triangles are not transformed. This is also the behavior for `format_version` before 1.1. If `fallback_strategy` is set to `nearest_side`, then points that do not fall into any triangle are transformed according to the triangle closest to them by euclidean distance. If `fallback_strategy` is set to `nearest_centroid`, then points that do not fall into any triangle are transformed according to the triangle with the closest centroid (intersection of its medians).

**vertices_columns**
: Specify the name of the columns of the rows in the `vertices` array. There must be exactly as many elements in `vertices_columns` as in a row of `vertices`. The following names have a special meaning: `source_x`, `source_y`, `target_x`, `target_y`, `source_z`, `target_z` and `offset_z`. `source_x` and `source_y`

are compulsory. `source_x` is for the source longitude (in degree) or easting. `source_y` is for the source latitude (in degree) or northing. `target_x` and `target_y` are compulsory when `horizontal` is specified in `transformed_components`. (`source_z` and `target_z`) or `offset_z` are compulsory when `vertical` is specified in `transformed_components`

**triangles_columns**
Specify the name of the columns of the rows in the `triangles` array. There must be exactly as many elements in `triangles_columns` as in a row of `triangles`. The following names have a special meaning: `idx_vertex1`, `idx_vertex2`, `idx_vertex3`. They are compulsory.

**vertices**
An array whose items are themselves arrays with as many columns as described in `vertices_columns`.

**triangles**
An array whose items are themselves arrays with as many columns as described in `triangles_columns`. The value of the `idx_vertexN` columns must be indices (between 0 and len(`vertices`-1) of items of the `vertices` array.

A JSON schema is available for this file format.

### 7.3.12 Vertical Offset And Slope

Added in version 9.1.0.

The Vertical Offset And Slope offsets transformation adds an offset to an orthometric height, taking account a constant offset and a inclinated plane, defined by its slope along latitude and longitude axis.

| | |
|---|---|
| **Alias** | vertoffset |
| **Domain** | 3D |
| **Input type** | Geodetic coordinates (horizontal), meters (vertical) |
| **output type** | Geodetic coordinates (horizontal), meters (vertical) |

It is documented as coordinate operation method code 1046 in the EPSG dataset ([IOGP2018]). It is typically used in Europe, to relate national vertical systems to pan-European vertical systems (EVRF200, EVRF2007).

#### 7.3.12.1 Examples

Vertical offset from LN02 height to EVRF2000 height with horizontal coordinates in ETRS89:

```
+proj=vertoffset +lat_0=46.9166666666666666 +lon_0=8.183333333333334 \
    +dh=-0.245 +slope_lat=-0.210 +slope_lon=-0.032
```

#### 7.3.12.2 Parameters

##### Required

**+lat_0**=<value>
Latitude of origin of the inclinated plane.

**+lon_0**=<value>
Longitude of origin of the inclinated plane

**+dh**=<value>
Offset in height, expressed in meter, to add.

**+slope_lat**=<value>

Slope parameter in the latitude domain, expressed in arc-seconds.

**+slope_lon**=<value>

Slope parameter in the longitude domain, expressed in arc-seconds.

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

#### 7.3.12.3 Formula

The $Z_{dest}$ destination elevation is obtained from the $Z_{src}$ source elevation with:

$$Z_{dest} = Z_{src} + (dh + slope_{lat} * \rho_0 * (\phi - \phi_0) + slope_{lon} * \nu_0 * (\lambda - \lambda_0) * cos(\phi)) \tag{7.11}$$

where:

- $dh$, $slope_{lat}$ and $slope_{lon}$ are the above mentioned parameters
- $\lambda_0$, $\phi_0$ is the longitude, latitude of the point of origin of the inclinate plane (**+lon_0**, **+lat_0**)
- $\lambda$, $\phi$ is the longitude, latitude of the point to evaluate
- $\rho_0$ is the radius of curvature of the meridian at latitude $\phi_0$
- $\nu_0$ is the radius of curvature on the prime vertical (i.e. perpendicular to the meridian) at latitude $\phi_0$

The reverse formula is:

$$Z_{src} = Z_{dest} - (dh + slope_{lat} * \rho_0 * (\phi - \phi_0) + slope_{lon} * \nu_0 * (\lambda - \lambda_0) * cos(\phi)) \tag{7.12}$$

### 7.3.13 Vertical grid shift

Added in version 5.0.0.

Change Vertical datum change by grid shift

| | |
|---|---|
| **Alias** | vgridshift |
| **Domain** | 3D and 4D |
| **Input type** | Geodetic coordinates (horizontal), meters (vertical), decimalyear (temporal) |
| **Output type** | Geodetic coordinates (horizontal), meters (vertical), decimalyear (temporal) |

The vertical grid shift is done by offsetting the vertical input coordinates by a specific amount determined by the loaded grids. The simplest use case of the horizontal grid shift is applying a single grid. Here we change the vertical reference from the ellipsoid to the global geoid model, EGM96:

```
+proj=vgridshift +grids=egm96_15.gtx
```

More than one grid can be loaded at the same time, for instance in the case where a better geoid model than the global is available for a certain area. Here the gridshift is set up so that the local DVR90 geoid model takes precedence over the global model:

```
+proj=vgridshift +grids=@dvr90.gtx,egm96_15.gtx
```

The `@` in the above example states that the grid is optional, in case the grid is not found in the PROJ search path. The list of grids is prioritized so that grids in the start of the list takes precedence over the grids in the back of the list.

PROJ supports the GeoTIFF (*Geodetic TIFF grids (GTG)*) and GTX file formats for vertical grid corrections. Details about all the formats can be found in the GDAL documentation. GDAL both reads and writes the format. Using GDAL for construction of new grids is recommended.

#### 7.3.13.1 Temporal gridshifting

Added in version 5.1.0.

By initializing the vertical gridshift operation with a central epoch, it can be used as a step function applying the grid offsets only if a coordinate is transformed from an epoch before grids central epoch to an epoch after. This is handy in transformations where it is necessary to handle deformations caused by seismic activity.

The central epoch of the grid is controlled with `+t_epoch` and the final epoch of the coordinate is set with `+t_final`. The observation epoch of the coordinate is part of the coordinate tuple.

Suppose we want to model the deformation of the 2008 earthquake in Iceland in a transformation of data from 2005 to 2009:

```
echo 63.992 -21.014 10.0 2005.0 | cct +proj=vgridshift +grids=iceland2008.gtx +t_
↪epoch=2008.4071 +t_final=2009.0
63.992 -21.014 10.11 2005.0
```

**Note**

The timestamp of the resulting coordinate is still 2005.0. The observation time is always kept unchanged as it would otherwise be impossible to do the inverse transformation.

Temporal gridshifting is especially powerful in transformation pipelines where several gridshifts can be chained together, effectively acting as a series of step functions that can be applied to a coordinate that is propagated through time. In the following example we establish a pipeline that allows transformation of coordinates from any given epoch up until the current date, applying only those gridshifts that have central epochs between the observation epoch and the final epoch:

```
+proj=pipeline +t_final=now
+step +proj=vgridshift +grids=earthquake_1.gtx +t_epoch=2010.421
+step +proj=vgridshift +grids=earthquake_2.gtx +t_epoch=2013.853
+step +proj=vgridshift +grids=earthquake_3.gtx +t_epoch=2017.713
```

**Note**

The special epoch *now* is used when specifying the final epoch with `+t_final`. This results in coordinates being transformed to the current date. Additionally, `+t_final` is used as a *global pipeline parameter*, which means that it is applied to all the steps in the pipeline.

In the above transformation, a coordinate with observation epoch 2009.32 would be subject to all three gridshift steps in the pipeline. A coordinate with observation epoch 2014.12 would only by offset by the last step in the pipeline.

#### 7.3.13.2 Parameters

##### Required

**+grids**=<list>

Comma-separated list of grids to load. If a grid is prefixed by an @ the grid is considered optional and PROJ will the not complain if the grid is not available.

Grids are expected to be in GeoTIFF (*Geodetic TIFF grids (GTG)*) or GTX formats.

##### Optional

**+t_epoch**=<time>

Central epoch of the transformation.

Added in version 5.1.0.

**+t_final**=<time>

Final epoch that the coordinate will be propagated to after transformation. The special epoch *now* can be used instead of writing a specific period in time. When *now* is used, it is replaced internally with the epoch of the transformation. This means that the resulting coordinate will be slightly different if carried out again at a later date.

Added in version 5.1.0.

**+multiplier**=<value>

Specify the multiplier to apply to the grid value in the forward transformation direction, such that:

$$Z_{target} = Z_{source} + multiplier \times gridvalue \tag{7.13}$$

The multiplier can be used to control whether the gridvalue should be added or subtracted, and if unit conversion must be done (the multiplied gridvalue must be expressed in metre).

Note that the default is *-1.0* for historical reasons.

Added in version 5.2.0.

### 7.3.14 Geocentric grid shift

Added in version 7.0.0.

Geocentric translation using a grid shift

| | |
|---|---|
| **Alias** | xyzgridshift |
| **Domain** | 3D |
| **Input type** | Cartesian coordinates |
| **Output type** | Cartesian coordinates |

Perform a geocentric translation by bilinear interpolation of dx, dy, dz translation values from a grid. The grid is referenced against either the 2D geographic CRS corresponding to the input (or sometimes output) CRS.

This method is described (in French) in [NTF_88] and as EPSG operation method code 9655 in [IOGP2018] (§2.4.4.1.1 France geocentric interpolation).

The translation in the grids are added to the input coordinates in the forward direction, and subtracted in the reverse direction. By default (if `grid_ref=input_crs`), in the forward direction, the input coordinates are converted to their geographic equivalent to directly read and interpolate from the grid. In the reverse direction, an iterative method is used to be able to find the grid locations to read. If `grid_ref=output_crs` is used, then the reverse strategy is applied: iterative method in the forward direction, and direct read in the reverse direction.

#### 7.3.14.1 Example

NTF to RGF93 transformation using `gr3df97a.tif` grid

```
+proj=pipeline
    +step +proj=push +v_3
    +step +proj=cart +ellps=clrk80ign
    +step +proj=xyzgridshift +grids=gr3df97a.tif +grid_ref=output_crs
    +step +proj=cart +inv
    +step +proj=pop +v_3
```

#### Parameters

The ellipsoid parameters should be the ones consistent with `grid_ref`. They are used to perform a geocentric to geographic conversion to find the translation parameters.

#### Required

**+ellps**=<value>

The name of a built-in ellipsoid definition.

See *Ellipsoids* for more information, or execute `proj -le` for a list of built-in ellipsoid names.

*Defaults to "GRS80".*

**+grids**=<list>

Comma-separated list of grids to load. If a grid is prefixed by an `@` the grid is considered optional and PROJ will the not complain if the grid is not available.

Grids are expected to be in GeoTIFF format (*Geodetic TIFF grids (GTG)*). If no metadata is provided, the first, second and third samples are assumed to be the geocentric translation along X, Y and Z axis respectively, in metres.

#### Optional

**+grid_ref**=input_crs/output_crs

Specify in which CRS the grid is referenced to. The default value is input_crs. That is the grid is referenced in the geographic CRS corresponding to the input geocentric CRS.

If output_crs is specified, the grid is referenced in the geographic CRS corresponding to the output geocentric CRS. This is for example the case for the French `gr3df97a.tif` grid converting from NTF to RGF93, but referenced against RGF93. Thus in the forward direction (NTF->RGF93), an iterative method is used to find the appropriate shift.

**+multiplier**=<value>

Specify the multiplier to apply to the grid values. Defaults to 1.0

## 7.4 The pipeline operator

Added in version 5.0.0.

Construct complex operations by daisy-chaining operations in a sequential pipeline.

| | |
|---|---|
| **Alias** | pipeline |
| **Domain** | 2D, 3D and 4D |
| **Input type** | Any |
| **Output type** | Any |

**Note**

See the section on *Geodetic transformation* for a more thorough introduction to the concept of transformation pipelines in PROJ.

With the pipeline operation it is possible to perform several operations after each other on the same input data. This feature makes it possible to create transformations that are made up of more than one operation, e.g. performing a datum shift and then applying a suitable map projection. Theoretically any transformation between two coordinate reference systems is possible to perform using the pipeline operation, provided that the necessary coordinate operations in each step is available in PROJ.

A pipeline is made up of a number of steps, with each step being a coordinate operation in itself. By connecting these individual steps sequentially we end up with a concatenated coordinate operation. An example of this is a transformation from geodetic coordinates on the GRS80 ellipsoid to a projected system where the east-west and north-east axes has been swapped:

```
+proj=pipeline +ellps=GRS80 +step +proj=merc +step +proj=axisswap +order=2,1
```

Here the first step is applying the *Mercator* projection and the second step is applying the *Axis swap* conversion. Note that the *+ellps=GRS80* is specified before the first occurrence of *+step*. This means that the GRS80 ellipsoid is used in both steps, since any parameter stated before the first occurrence of *+step* is treated as a global parameter and is transferred to each individual steps.

### 7.4.1 Rules for pipelines

**1. Pipelines must consist of at least one step.**

```
+proj=pipeline
```

Will result in an error.

**2. Pipelines can only be nested if the nested pipeline is defined in an init file.**

```
+proj=pipeline
+step +proj=pipeline +step +proj=merc +step +proj=axisswap +order=2,1
+step +proj=unitconvert +xy_in=m +xy_out=us-ft
```

Results in an error, while

```
+proj=pipeline
+step +init=predefined_pipelines:projectandswap
+step +proj=unitconvert +xy_in=m +xy_out=us-ft
```

does not.

**3. Pipelines without a forward path can't be constructed.**

```
+proj=pipeline +step +inv +proj=urm5
```

Will result in an error since *Urmaev V* does not have an inverse operation defined.

**4. Parameters added before the first `+step` are global and will be applied to all steps.**

In the following the GRS80 ellipsoid will be applied to all steps.

```
+proj=pipeline +ellps=GRS80
+step +proj=cart
+step +proj=helmert +x=10 +y=3 +z=1
+step +proj=cart +inv
+step +proj=merc
```

**5. Units of operations must match between steps.**

Added in version 5.1.0.

The output units of step *n* must match the expected input unit of step *n+1*. E.g., you can't pass an operation that outputs projected coordinates to an operation that expects angular units (degrees). An example of such a unit mismatch is displayed below.

```
+proj=pipeline
+step +proj=merc  # Mercator outputs projected coordinates
+step +proj=robin # The Robinson projection expects angular input
```

### 7.4.2 Parameters

#### 7.4.2.1 Required

**+step**

Separate each step in a pipeline.

#### 7.4.2.2 Optional

**+inv**

Invert a step in a pipeline.

```
echo -88 42 0 0|cct +zone=16 +proj=pipeline +step +proj=utm +step +proj=utm +inv
```

**+omit_fwd**

Added in version 6.3.0.

Skip a step of the pipeline when it is followed in the forward path.

The following example shows a combined use of *push* and *pop* operators, with `omit_fwd` and `omit_inv` options, to implement a vertical adjustment that must be done in a interpolation CRS that is different from the horizontal CRS used in input and output. +omit_fwd in the forward path avoid a useless inverse horizontal transformation and relies on the pop operator to restore initial horizontal coordinates. +omit_inv serves the similar purpose when the pipeline is executed in the reverse direction

```
+proj=pipeline
+step +proj=unitconvert +xy_in=deg +xy_out=rad
+step +proj=push +v_1 +v_2
+step +proj=hgridshift +grids=nvhpgn.gsb +omit_inv
+step +proj=vgridshift +grids=g1999u05.gtx +multiplier=1
```

(continues on next page)

(continued from previous page)

```
+step +inv +proj=hgridshift +grids=nvhpgn.gsb +omit_fwd
+step +proj=pop +v_1 +v_2
+step +proj=unitconvert +xy_in=rad +xy_out=deg
```

**+omit_inv**

Added in version 6.3.0.

Skip a step of the pipeline when it is followed in the reverse path.

## 7.5 Computation of coordinate operations between two CRS

**Author**
Even Rouault

**Last Updated**
2023-08-26

### 7.5.1 Introduction

When using **projinfo -s {crs_def} -t {crs_def}**, **cs2cs {crs_def} {crs_def}** or the underlying *proj_create_crs_to_crs()* or proj_create_operations() functions, PROJ applies an algorithm to compute one or several candidate coordinate operations, that can be expressed as a PROJ *pipeline* to transform between the source and the target CRS. This document is about the description of this algorithm that finds the actual operations to apply to be able later to perform transform coordinates. So this is mostly about metadata management around coordinate operation methods, and not about the actual mathematics used to implement those methods. As a matter of fact with PROJ 6, there are about 60 000 lines of code dealing with "metadata" management (including conversions between PROJ strings, all CRS WKT variants), to be compared to 30 000 for the purely computation part.

This document is meant as a plain text explanation of the code for developers, but also as a in-depth examination of what happens under the hood for curious PROJ users. It is important to keep in mind that it is not meant to be the ultimate source of truth of how coordinate operations should be computed. There are clearly implementation choices and compromises that can be questioned.

Let us start with an example to research operations between the NAD27 and NAD83 geographic CRS:

```
$ projinfo -s NAD27 -t NAD83 --summary --spatial-test intersects --grid-check none

Candidate operations found: 10
DERIVED_FROM(EPSG):1312, NAD27 to NAD83 (3), 1.0 m, Canada
DERIVED_FROM(EPSG):1313, NAD27 to NAD83 (4), 1.5 m, Canada - NAD27, at least one grid␣
↪missing
DERIVED_FROM(EPSG):1241, NAD27 to NAD83 (1), 0.15 m, USA - CONUS including EEZ
DERIVED_FROM(EPSG):1243, NAD27 to NAD83 (2), 0.5 m, USA - Alaska including EEZ
DERIVED_FROM(EPSG):1573, NAD27 to NAD83 (6), 1.5 m, Canada - Quebec, at least one grid␣
↪missing
EPSG:1462, NAD27 to NAD83 (5), 1.0 m, Canada - Quebec, at least one grid missing
EPSG:9111, NAD27 to NAD83 (9), 1.5 m, Canada - Saskatchewan, at least one grid missing
unknown id, Ballpark geographic offset from NAD27 to NAD83, unknown accuracy, World, has␣
↪ballpark transformation
EPSG:8555, NAD27 to NAD83 (7), 0.15 m, USA - CONUS and GoM, at least one grid missing
EPSG:8549, NAD27 to NAD83 (8), 0.5 m, USA - Alaska, at least one grid missing
```

The algorithm involves many cases, so we will progress in the explanation from the most simple case to more complex ones. We document here the working of this algorithm as implemented in PROJ 8.0.0. The results of some examples

might also be quite sensitive to the content of the PROJ database and the PROJ version used.

From a code point of view, the entry point of the algorithm is the C++ `osgeo::proj::operation::CoordinateOperation::createOperations()` method.

It combines several strategies:

- look up in the PROJ database for available operations
- consider the pair (source CRS, target CRS) to synthesize operations depending on the nature of the source and target CRS.

### 7.5.2 Geographic CRS to Geographic CRS, with known identifiers

With the above example of two geographic CRS, that have an identified identifier, (*projinfo* internally resolves NAD27 to EPSG:4267 and NAD83 to EPSG:4269) the algorithm will first search in the coordinate operation related tables of the `proj.db` if there are records that list direct transformations between the source and the target CRS. The transformations typically involve *Helmert*-style operations or datum shift based on grids (more esoteric operations are possible).

A request similar to the following will be emitted:

```
$ sqlite3 proj.db "SELECT auth_name, code, name, method_name, accuracy FROM \
                   coordinate_operation_view WHERE \
                   source_crs_auth_name = 'EPSG' AND \
                   source_crs_code = '4267' AND \
                   target_crs_auth_name = 'EPSG' AND \
                   target_crs_code = '4269'"

EPSG|1241|NAD27 to NAD83 (1)|NADCON|0.15
EPSG|1243|NAD27 to NAD83 (2)|NADCON|0.5
EPSG|1312|NAD27 to NAD83 (3)|NTv1|1.0
EPSG|1313|NAD27 to NAD83 (4)|NTv2|1.5
EPSG|1462|NAD27 to NAD83 (5)|NTv1|1.0
EPSG|1573|NAD27 to NAD83 (6)|NTv2|1.5
EPSG|8549|NAD27 to NAD83 (8)|NADCON5 (2D)|0.5
EPSG|8555|NAD27 to NAD83 (7)|NADCON5 (2D)|0.15
EPSG|9111|NAD27 to NAD83 (9)|NTv2|1.5
ESRI|108003|NAD_1927_To_NAD_1983_PR_VI|NTv2|0.05
```

As we have found direct transformations, we will not attempt any more complicated research. One can note in the above result set that a ESRI:108003 operation was found, but as the source and target CRS are in the EPSG registry, and there are operations between those CRS in the EPSG registry itself, transformations from other authorities will be ignored (except if they are in the PROJ authority, which can be used as an override).

As those results all involve operations that does not have a perfect accuracy and that does not cover the area of use of the 2 CRSs, a 'Ballpark geographic offset from NAD27 to NAD83' operation is synthesized by PROJ (see *Ballpark transformation*)

### 7.5.3 Filtering and sorting of coordinate operations

The last step is to filter and sort results in order of relevance.

The filtering takes into account the following criteria to decide which operations must be retained or discarded:

- a minimum accuracy that the user might have expressed,
- an area of use on which the coordinate operation(s) must apply
- if the absence of grids needed by an operation must result in discarding it.

The sorting algorithm determines the order of relevance of the operations we got. A comparison function compares pair of operations to determine which of the two is the most relevant. This is implemented by the `operator ()` method of the SortFunction structure. When comparing two operations, the following criteria are used. The tests are performed in the order they are listed below:

1. consider as more relevant an operation that can be expressed as a PROJ operation string (the database might list operations whose method is not (yet) implemented by PROJ)
2. if both operations evaluate identically with respect to the above criterion, consider as more relevant an operation that does not include a synthetic ballpark vertical transformation (occurs when there is a geoid model).
3. if both operations evaluate identically with respect to the above criterion, consider as more relevant an operation that does not include a synthetic ballpark horizontal transformation.
4. consider as more relevant an operation that refers to shift grids that are locally available.
5. consider as more relevant an operation that refers to grids that are available in one of the proj-datumgrid packages, but not necessarily locally available
6. consider as more relevant an operation that has a known accuracy.
7. if two operations have unknown accuracy, consider as more relevant an operation that uses grid(s) if the other one does not (grid based operations are assumed to be more precise than operations relying on a few parameters)
8. consider as more relevant an operation whose area of use is larger (note: the computation of the are of use is approximate, based on a bounding box)
9. consider as more relevant an operation that has a better accuracy.
10. in case of same accuracy, consider as more relevant an operation that does not use grids (operations that use only parameters will be faster)
11. consider as more relevant an operation that involves less transformation steps (transformation steps considered are the ones listed in the WKT output, not PROJ pipeline steps)
12. and for completeness, if two operations are comparable given all the above criteria, consider as more relevant the one which has the shorter name, and if they have the same length, consider as more relevant the one whose name comes last in lexicographic order (e.g. "FOO to BAR (3)" will have higher precedence than "FOO to BAR (2)")

**Note**

*proj_trans()*, on the results returned by *proj_create_crs_to_crs()*, will not necessarily use the operation that is listed in first position due to the above algorithm. *proj_trans()* has more context, since it has the coordinate to transform, so it can compare this coordinate to the area of use of operations. Typically, the above criteria will favor an operation that has a larger area of use over another one with a smaller area, due to it being more generally applicable. But once coordinates are known, *proj_trans()* can select an operation with a smaller area of use that applies to the coordinate to transform.

### 7.5.4 Geodetic/geographic CRS to Geodetic/geographic CRS, without known identifiers

In a number of situations, the source and/or target CRS do not have an identifier (WKT without identifier, PROJ string, ..) The first step is to try to find in the `proj.db` a CRS of the same nature of the CRS to identify and whose name exactly matches the one provided to the `createOperations()` method. If there is exactly one match and that the CRS are "computationally" equivalent, then use the code of the CRS for further computations.

If this search did not succeed, or if the previous case with known CRS identifiers did not result in matches in the database, the search will be based on the datums. That is, a list of geographic CRS whose datum matches the datum of

the source and target CRS is searched for in the database (by querying the *geodetic_crs* database table). If the datum has a known identifier, we will use it, otherwise we will look for an equivalent datum in the database based on the datum name.

Let's consider the case where the datum of the source CRS is EPSG:6171 "Reseau Geodesique Francais 1993" and the datum of the target CRS is EPSG:6258 "European Terrestrial Reference System 1989". For EPSG:6171, there are 10 matching (non-deprecated) geodetic CRSs:

- EPSG:4171, RGF93, geographic 2D
- EPSG:4964, RGF93, geocentric
- EPSG:4965, RGF93, geographic 3D
- EPSG:7042, RGF93 (lon-lat), geographic 3D
- EPSG:7084, RGF93 (lon-lat), geographic 2D
- IGNF:RGF93, RGF93 cartesiennes geocentriques, geocentric
- IGNF:RGF93GDD, RGF93 geographiques (dd),geographic 2D
- IGNF:RGF93GEODD, RGF93 geographiques (dd), geographic 3D
- IGNF:RGF93G, RGF93 geographiques (dms), geographic 2D
- IGNF:RGF93GEO, RGF93 geographiques (dms), geographic 3D

The first three entries from the EPSG dataset are typical: for each datum, one can define a geographic 2D CRS (latitude, longitude), a geographic 3D crs (latitude, longitude, ellipsoidal height) and a geocentric one. For that particular case, the EPSG dataset has also included two extra definitions corresponding to a longitude, latitude, [ellipsoidal height] coordinate system, as found in the official French IGNF registry. This IGNF registry has also definitions for a geographic 2D CRS (with an extra subtlety with an entry using decimal degree as unit and another one degree-minute-second), geographic 3D and geocentric.

For EPSG:6258, there are 7 matching (non-deprecated) geodetic CRSs:

- EPSG:4258, ETRS89, geographic 2D
- EPSG:4936, ETRS89, geocentric
- EPSG:4937, ETRS89, geographic 3D
- IGNF:ETRS89, ETRS89 cartesiennes geocentriques, geocentric
- IGNF:ETRS89G, ETRS89 geographiques (dms), geographic 2D
- IGNF:ETRS89GEO, ETRS89 geographiques (dms), geographic 3D
- ESRI:104129, GCS_EUREF_FIN, geographic 2D

So the 3 typical EPSG entries, 3 equivalent (with long, lat ordering for the geographic CRS) and one from the ESRI registry;

PROJ can now test 10 x 7 different combinations of source x target CRSs, using the database searching method explained in the previous section. As soon as one of this combination returns at least one non-ballpark combination, the result set coming from that combination is used. PROJ will then add before that transformation a conversion between the source CRS and the first intermediate CRS, and will add at the end a conversion between the second intermediate CRS and the target CRS. Those conversions are conversion between geographic 2D and geographic 3D CRS or geographic 2D/3D and geocentric CRS.

This is done by the `createOperationsWithDatumPivot()` method.

So if transforming between EPSG:7042, RGF93 (lon-lat), geographic 3D and EPSG:4936, ETRS89, geocentric, one get the following concatenated operation, chaining an axis order change, the null geocentric translation between RGF93

and ETRS89 (EPSG:1591), and a conversion between geographic and geocentric coordinates. This concatenated operation is assumed to have a perfect accuracy as both the initial and final operations are conversions, and the middle transformation accounts for the fact that the RGF93 datum is one realization of ETRS89, so they are equivalent for most purposes.

```
$ projinfo -s EPSG:7042 -t EPSG:4936

Candidate operations found: 1
-------------------------------------
Operation No. 1:

unknown id, axis order change (geographic3D horizontal) + RGF93 to ETRS89 (1) +␣
↪Conversion from ETRS89 (geog2D) to ETRS89 (geocentric), 0 m, France

PROJ string:
+proj=pipeline +step +proj=unitconvert +xy_in=deg +xy_out=rad +step +proj=cart␣
↪+ellps=GRS80

WKT2:2019 string:
CONCATENATEDOPERATION["axis order change (geographic3D horizontal) + RGF93 to ETRS89 (1)␣
↪+ Conversion from ETRS89 (geog2D) to ETRS89 (geocentric)",
    SOURCECRS[
        GEOGCRS["RGF93 (lon-lat)",
            [...]
            ID["EPSG",7042]]],
    TARGETCRS[
        GEODCRS["ETRS89",
            [...]
            ID["EPSG",4936]]],
    STEP[
        CONVERSION["axis order change (geographic3D horizontal)",
            METHOD["Axis Order Reversal (Geographic3D horizontal)",
                ID["EPSG",9844]],
            ID["EPSG",15499]]],
    STEP[
        COORDINATEOPERATION["RGF93 to ETRS89 (1)",
            [...]
            METHOD["Geocentric translations (geog2D domain)",
                ID["EPSG",9603]],
            PARAMETER["X-axis translation",0,
                LENGTHUNIT["metre",1],
                ID["EPSG",8605]],
            PARAMETER["Y-axis translation",0,
                LENGTHUNIT["metre",1],
                ID["EPSG",8606]],
            PARAMETER["Z-axis translation",0,
                LENGTHUNIT["metre",1],
                ID["EPSG",8607]],
            OPERATIONACCURACY[0.0],
            ID["EPSG",1591],
            REMARK["May be taken as approximate transformation RGF93 to WGS 84 - see␣
↪code 1671."]]],
    STEP[
```

(continues on next page)

(continued from previous page)

```
        CONVERSION["Conversion from ETRS89 (geog2D) to ETRS89 (geocentric)",
            METHOD["Geographic/geocentric conversions",
                ID["EPSG",9602]]],
    USAGE[
        SCOPE["unknown"],
        AREA["France"],
        BBOX[41.15,-9.86,51.56,10.38]]]
```

### 7.5.5 Geodetic/geographic CRS to Geodetic/geographic CRS, without direct transformation

Still considering transformations between geodetic/geographic CRS, but let's consider that the lookup in the database for a transformation between the source and target CRS (possibly going through the "equivalent" CRS based on the same datum as detailed in the previous section) leads to an empty set.

Of course, as most operations are invertible, one first tries to do a lookup switching the source and target CRS, and inverting the resulting operation(s):

```
$ projinfo -s NAD83 -t NAD27 --spatial-test intersects --summary

Candidate operations found: 10
INVERSE(DERIVED_FROM(EPSG)):1312, Inverse of NAD27 to NAD83 (3), 2.0 m, Canada
INVERSE(DERIVED_FROM(EPSG)):1313, Inverse of NAD27 to NAD83 (4), 1.5 m, Canada - NAD27
INVERSE(DERIVED_FROM(EPSG)):1241, Inverse of NAD27 to NAD83 (1), 0.15 m, USA - CONUS␣
↪including EEZ
INVERSE(DERIVED_FROM(EPSG)):1243, Inverse of NAD27 to NAD83 (2), 0.5 m, USA - Alaska␣
↪including EEZ
INVERSE(DERIVED_FROM(EPSG)):1573, Inverse of NAD27 to NAD83 (6), 1.5 m, Canada - Quebec,␣
↪at least one grid missing
INVERSE(EPSG):1462, Inverse of NAD27 to NAD83 (5), 2.0 m, Canada - Quebec, at least one␣
↪grid missing
INVERSE(EPSG):9111, Inverse of NAD27 to NAD83 (9), 1.5 m, Canada - Saskatchewan, at␣
↪least one grid missing
unknown id, Ballpark geographic offset from NAD83 to NAD27, unknown accuracy, World, has␣
↪ballpark transformation
INVERSE(EPSG):8555, Inverse of NAD27 to NAD83 (7), 0.15 m, USA - CONUS and GoM, at least␣
↪one grid missing
INVERSE(EPSG):8549, Inverse of NAD27 to NAD83 (8), 0.5 m, USA - Alaska, at least one␣
↪grid missing
```

That was an easy case. Now let's consider the transformation between the Australian CRS AGD84 and GDA2020. There is no direct transformation from AGD84 to GDA2020, or in the reverse direction, even when considering alternative geodetic CRS based on the underlying datums. PROJ will then do a cross-join in the coordinate_operation_view table to find the tuples (op1, op2) of coordinate operations such that:

- SOURCE_CRS = op1.source_crs AND op1.target_crs = op2.source_crs AND op2.target_crs = TARGET_CRS or
- SOURCE_CRS = op1.source_crs AND op1.target_crs = op2.target_crs AND op2.source_crs = TARGET_CRS or
- SOURCE_CRS = op1.target_crs AND op1.source_crs = op2.source_crs AND op2.target_crs = TARGET_CRS or
- SOURCE_CRS = op1.target_crs AND op1.source_crs = op2.target_crs AND op2.source_crs = TARGET_CRS

Depending on which case is selected, op1 and op2 should be reversed, before being concatenated.

This logic is implement by the `findsOpsInRegistryWithIntermediate()` method.

Assuming that the proj-datumgrid-oceania package is installed, we get the following results for the AGD84 to GDA2020 coordinate operations lookup:

```
$ projinfo -s AGD84 -t GDA2020 --spatial-test intersects -o PROJ

Candidate operations found: 4
-------------------------------------
Operation No. 1:

unknown id, AGD84 to GDA94 (5) + GDA94 to GDA2020 (1), 0.11 m, Australia - AGD84

PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 \
               +step +proj=unitconvert +xy_in=deg +xy_out=rad \
               +step +proj=hgridshift +grids=National_84_02_07_01.gsb \
               +step +proj=push +v_3 \
               +step +proj=cart +ellps=GRS80 \
               +step +proj=helmert +x=0.06155 +y=-0.01087 +z=-0.04019 \
                                   +rx=-0.0394924 +ry=-0.0327221 +rz=-0.0328979 \
                                   +s=-0.009994 +convention=coordinate_frame \
               +step +inv +proj=cart +ellps=GRS80 \
               +step +proj=pop +v_3 \
               +step +proj=unitconvert +xy_in=rad +xy_out=deg \
               +step +proj=axisswap +order=2,1

-------------------------------------
Operation No. 2:

unknown id, AGD84 to GDA94 (2) + GDA94 to GDA2020 (1), 1.01 m, Australia - AGD84

PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 \
               +step +proj=unitconvert +xy_in=deg +xy_out=rad \
               +step +proj=push +v_3 \
               +step +proj=cart +ellps=aust_SA \
               +step +proj=helmert +x=-117.763 +y=-51.51 +z=139.061 \
                                   +rx=-0.292 +ry=-0.443 +rz=-0.277 +s=-0.191 \
                                   +convention=coordinate_frame \
               +step +proj=helmert +x=0.06155 +y=-0.01087 +z=-0.04019 \
                                   +rx=-0.0394924 +ry=-0.0327221 +rz=-0.0328979 \
                                   +s=-0.009994 +convention=coordinate_frame \
               +step +inv +proj=cart +ellps=GRS80 \
               +step +proj=pop +v_3 \
               +step +proj=unitconvert +xy_in=rad +xy_out=deg \
               +step +proj=axisswap +order=2,1

-------------------------------------
Operation No. 3:

unknown id, AGD84 to GDA94 (5) + GDA94 to GDA2020 (2), 0.15 m, unknown domain of validity
```

(continues on next page)

(continued from previous page)

```
PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 \
               +step +proj=unitconvert +xy_in=deg +xy_out=rad \
               +step +proj=hgridshift +grids=National_84_02_07_01.gsb \
               +step +proj=hgridshift +grids=GDA94_GDA2020_conformal_and_distortion.gsb \
               +step +proj=unitconvert +xy_in=rad +xy_out=deg \
               +step +proj=axisswap +order=2,1

-------------------------------------
Operation No. 4:

unknown id, AGD84 to GDA94 (5) + GDA94 to GDA2020 (3), 0.15 m, unknown domain of validity

PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 \
               +step +proj=unitconvert +xy_in=deg +xy_out=rad \
               +step +proj=hgridshift +grids=National_84_02_07_01.gsb \
               +step +proj=hgridshift +grids=GDA94_GDA2020_conformal.gsb \
               +step +proj=unitconvert +xy_in=rad +xy_out=deg \
               +step +proj=axisswap +order=2,1
```

One can see that the selected intermediate CRS that has been used is GDA94. This is a completely novel behavior of PROJ 6 as opposed to the logic of PROJ.4 where datum transformations implied using EPSG:4326 / WGS 84 has the mandatory datum hub. PROJ 6 no longer hardcodes it as the mandatory datum hub, and relies on the database to find the appropriate hub(s). Actually, WGS 84 has been considered during the above lookup, because there are transformations between AGD84 and WGS 84 and WGS 84 and GDA2020. However those have been discarded in a step which we did not mention previously: just after the initial filtering of results and their sorting, there is a final filtering that is done. In the list of sorted results, given two operations A and B that have the same area of use, if B has an accuracy lower than A, and A does not use grids, or all the needed grids are available, then B is discarded.

If one forces the datum hub to be considered to be EPSG:4326, ones gets:

```
$ projinfo -s AGD84 -t GDA2020 --spatial-test intersects --pivot-crs EPSG:4326 -o PROJ

Candidate operations found: 2
-------------------------------------
Operation No. 1:

unknown id, AGD84 to WGS 84 (7) + Inverse of GDA2020 to WGS 84 (2), 4 m, Australia -␣
↪AGD84

PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 \
               +step +proj=unitconvert +xy_in=deg +xy_out=rad \
               +step +proj=push +v_3 \
               +step +proj=cart +ellps=aust_SA \
               +step +proj=helmert +x=-117.763 +y=-51.51 +z=139.061 \
                                   +rx=-0.292 +ry=-0.443 +rz=-0.277 \
                                   +s=-0.191 +convention=coordinate_frame \
               +step +inv +proj=cart +ellps=GRS80 \
               +step +proj=pop +v_3 \
               +step +proj=unitconvert +xy_in=rad +xy_out=deg \
```

(continues on next page)

(continued from previous page)

```
               +step +proj=axisswap +order=2,1

------------------------------------
Operation No. 2:

unknown id, AGD84 to WGS 84 (9) + Inverse of GDA2020 to WGS 84 (2), 4 m, Australia -␣
↪AGD84

PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 \
               +step +proj=unitconvert +xy_in=deg +xy_out=rad \
               +step +proj=hgridshift +grids=National_84_02_07_01.gsb \
               +step +proj=unitconvert +xy_in=rad +xy_out=deg \
               +step +proj=axisswap +order=2,1
```

Those operations are less accurate, since WGS 84 is assumed to be equivalent to GDA2020 with an accuracy of 4 metre. This is an instance demonstrating that using WGS 84 as a hub systematically can be sub-optimal.

There are still situations where the attempt to find a hub CRS does not work, because there is no such hub. This can occur for example when transforming from GDA94 to the latest realization at time of writing of WGS 84, WGS 84 (G1762). There are transformations between WGS 84 (G1762). Using the above described techniques, we would only find one non-ballpark operation taking the route: 1. Conversion from GDA94 (geog2D) to GDA94 (geocentric): synthesized by PROJ 2. Inverse of ITRF2008 to GDA94 (1): from EPSG 3. Inverse of WGS 84 (G1762) to ITRF2008 (1): from EPSG 4. Conversion from WGS 84 (G1762) (geocentric) to WGS 84 (G1762): synthesized by PROJ

This is not bad, but the global validity area of use is "Australia - onshore and EEZ", whereas GDA94 has a larger area of use. There is another road that can be taken by going through GDA2020 instead of ITRF2008. The GDA94 to GDA2020 transformations operate on the respective geographic CRS, whereas GDA2020 to WGS 84 (G1762) operate on the geocentric CRS. Consequently, GDA2020 cannot be identifier as a hub by a "simple" self-join SQL request on the coordinate operation table. This requires to do the join based on the datum referenced by the source and target CRS of each operation rather than the source and target CRS themselves. When there is a match, PROJ inserts the required conversions between geographic and geocentric CRS to have a consistent concatenated operation, like the following: 1. GDA94 to GDA2020 (1): from EPSG 2. Conversion from GDA2020 (geog2D) to GDA2020 (geocentric): synthesized by PROJ 3. GDA2020 to WGS 84 (G1762) (1): from EPSG 4. Conversion from WGS 84 (G1762) (geocentric) to WGS 84 (G1762) (geog2D): synthesized by PROJ

### 7.5.6 Projected CRS to any target CRS

This actually extends to any Derived CRS, whose Projected CRS is a well-known particular case. Such transformations are done in 2 steps:

1. Use the inverse conversion of the derived CRS to its base CRS, typically an inverse map projection.
2. Find transformations from this base CRS to the target CRS. If the base CRS is the target CRS, this step can be skipped.

```
$ projinfo -s EPSG:32631 -t RGF93

Candidate operations found: 1
------------------------------------
Operation No. 1:

unknown id, Inverse of UTM zone 31N + Inverse of RGF93 to WGS 84 (1), 1 m, France
```

(continues on next page)

(continued from previous page)

```
PROJ string:
+proj=pipeline +step +inv +proj=utm +zone=31 +ellps=WGS84 +step +proj=unitconvert +xy_
↪in=rad +xy_out=deg +step +proj=axisswap +order=2,1
```

This is implemented by the `createOperationsDerivedTo` method

For the symmetric case, source CRS to a derived CRS, the above algorithm is applied by switching the source and target CRS, and then inverting the resulting operation(s). This is mostly a matter of avoiding to write very similar code twice. This logic is also applied to all below cases when considering the transformation between 2 different types of objects.

### 7.5.7 Vertical CRS to a Geographic CRS

Such transformation is normally not meant as being used as standalone by PROJ users, but as an internal computation step of a Compound CRS to a target CRS.

In cases where we are lucky, the PROJ database will have a transformation registered between those:

```
$ projinfo -s "NAVD88 height" -t "NAD83(2011)" -o PROJ --spatial-test intersects
Candidate operations found: 11
-------------------------------------
Operation No. 1:

INVERSE(DERIVED_FROM(EPSG)):9229, Inverse of NAD83(2011) to NAVD88 height (3), 0.015 m,␣
↪USA - CONUS - onshore

PROJ string:
+proj=vgridshift +grids=g2018u0.gtx +multiplier=1
```

But in cases where there is no match, the `createOperationsVertToGeog` method will be used to synthesize a ballpark vertical transformation, just taking care of unit changes, and axis reversal in case the vertical CRS was a depth rather than a height. Of course the results of such an operation are questionable, hence the ballpark qualifier and a unknown accuracy advertised for such an operation.

### 7.5.8 Vertical CRS to a Vertical CRS

Overall logic is similar to the above case. There might be direct operations in the PROJ database, involving grid transformations or simple offsets. The fallback case is to synthesize a ballpark transformation.

This is implemented by the `createOperationsVertToVert` method

```
$ projinfo -s "NGVD29 depth (ftUS)" -t "NAVD88 height" --spatial-test intersects -o PROJ

Candidate operations found: 3
-------------------------------------
Operation No. 1:

unknown id, Inverse of NGVD29 height (ftUS) to NGVD29 depth (ftUS) + NGVD29 height␣
↪(ftUS) to NGVD29 height (m) + NGVD29 height (m) to NAVD88 height (3), 0.02 m, USA -␣
↪CONUS east of 89°W - onshore

PROJ string:
+proj=pipeline +step +proj=axisswap +order=1,2,-3 +step +proj=unitconvert +z_in=us-ft +z_
↪out=m +step +proj=vgridshift +grids=vertcone.gtx +multiplier=0.001
```

(continues on next page)

(continued from previous page)

```
------------------------------------
Operation No. 2:

unknown id, Inverse of NGVD29 height (ftUS) to NGVD29 depth (ftUS) + NGVD29 height␣
↪(ftUS) to NGVD29 height (m) + NGVD29 height (m) to NAVD88 height (2), 0.02 m, USA -␣
↪CONUS 89°W-107°W - onshore

PROJ string:
+proj=pipeline +step +proj=axisswap +order=1,2,-3 +step +proj=unitconvert +z_in=us-ft +z_
↪out=m +step +proj=vgridshift +grids=vertconc.gtx +multiplier=0.001

------------------------------------
Operation No. 3:

unknown id, Inverse of NGVD29 height (ftUS) to NGVD29 depth (ftUS) + NGVD29 height␣
↪(ftUS) to NGVD29 height (m) + NGVD29 height (m) to NAVD88 height (1), 0.02 m, USA -␣
↪CONUS west of 107°W - onshore

PROJ string:
+proj=pipeline +step +proj=axisswap +order=1,2,-3 +step +proj=unitconvert +z_in=us-ft +z_
↪out=m +step +proj=vgridshift +grids=vertconw.gtx +multiplier=0.001
```

### 7.5.9 Compound CRS to a Geographic CRS

A typical example of a Compound CRS is a CRS made of a geographic or projected CRS as the horizontal component, and a vertical CRS. E.g. "NAD83 + NAVD88 height"

When the horizontal component of the compound source CRS is a projected CRS, we first look for the operation from this source CRS to another compound CRS made of the geographic CRS base of the projected CRS, like "NAD83 / California zone 1 (ftUS) + NAVD88 height" to "NAD83 + NAVD88 height", which ultimately goes to one of the above described case. Then we can consider the transformation from a compound CRS made of a geographic CRS to another geographic CRS.

It first starts by the vertical transformations from the vertical CRS of the source compound CRS to the target geographic CRS, using the strategy detailed in *Vertical CRS to a Geographic CRS*

What we did not mention is that when there is not a transformation registered between the vertical CRS and the target geographic CRS, PROJ attempts to find transformations between that vertical CRS and any other geographic CRS. This is clearly an approximation. If the research of the vertical CRS to the target geographic CRS resulted in operations that use grids that are not available, as another approximation, we research operations from the vertical CRS to the source geographic CRS for the vertical component.

Once we got those more or less accurate vertical transformations, we must consider the horizontal transformation(s). The algorithm iterates over all found vertical transformations and look for their target geographic CRS. This will be used as the interpolation CRS for horizontal transformations. PROJ will then look for available transformations from the source geographic CRS to the interpolation CRS and from the interpolation CRS to the target geographic CRS. There is then a 3-level loop to create the final set of operations chaining together:

- the horizontal transformation from the source geographic CRS to the interpolation CRS
- the vertical transformation from the source vertical CRS to the interpolation CRS
- the horizontal transformation from the interpolation CRS to the target geographic CRS.

This is implemented by the `createOperationsCompoundToGeog` method

Example:

```
$ projinfo -s "NAD83(NSRS2007) + NAVD88 height" -t "WGS 84 (G1762)" --spatial-test␣
↪intersects --summary

Candidate operations found: 21
unknown id, Inverse of NAD83(NSRS2007) to NAVD88 height (1) + NAD83(NSRS2007) to WGS 84␣
↪(1) + WGS 84 to WGS 84 (G1762), 3.05 m, USA - CONUS - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (7) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, 95°W to 78°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (7) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, 95°W to 78°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (6) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, 112°W to 95°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (6) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, 112°W to 95°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (2) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, 112°W to 95°W
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (2) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, 112°W to 95°W
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (3) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, 95°W to 78°W
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (3) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, 95°W to 78°W
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (5) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, west of 112°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (5) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, west of 112°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (1) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, west of 112°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (1) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, west of 112°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (4) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, east of 78°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (4) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS north of 41°N, east of 78°W - onshore
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (8) + NAD83(HARN) to WGS 84 (1) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, east of 78°W - onshore
```

(continues on next page)

(continued from previous page)

```
unknown id, Inverse of NAD83(HARN) to NAD83(NSRS2007) (1) + Inverse of NAD83(HARN) to␣
↪NAVD88 height (8) + NAD83(HARN) to WGS 84 (3) + WGS 84 to WGS 84 (G1762), 3.15 m, USA -
↪ CONUS south of 41°N, east of 78°W - onshore
unknown id, Ballpark geographic offset from NAD83(NSRS2007) to NAD83(FBN) + Inverse of␣
↪NAD83(FBN) to NAVD88 height (1) + Ballpark geographic offset from NAD83(FBN) to WGS 84␣
↪(G1762), unknown accuracy, USA - CONUS - onshore, has ballpark transformation
unknown id, Ballpark geographic offset from NAD83(NSRS2007) to NAD83(2011) + Inverse of␣
↪NAD83(2011) to NAVD88 height (3) + Ballpark geographic offset from NAD83(2011) to WGS␣
↪84 (G1762), unknown accuracy, USA - CONUS - onshore, has ballpark transformation
unknown id, Ballpark geographic offset from NAD83(NSRS2007) to NAD83(2011) + Inverse of␣
↪NAD83(2011) to NAVD88 height (3) + Conversion from NAD83(2011) (geog2D) to NAD83(2011)␣
↪(geocentric) + Inverse of ITRF2008 to NAD83(2011) (1) + Inverse of WGS 84 (G1762) to␣
↪ITRF2008 (1) + Conversion from WGS 84 (G1762) (geocentric) to WGS 84 (G1762) (geog2D),␣
↪unknown accuracy, USA - CONUS - onshore, has ballpark transformation
unknown id, NAD83(NSRS2007) to WGS 84 (1) + WGS 84 to WGS 84 (G1762) + Transformation␣
↪from NAVD88 height to WGS 84 (G1762) (ballpark vertical transformation, without␣
↪ellipsoid height to vertical height correction), unknown accuracy, USA - CONUS and␣
↪Alaska; PRVI, has ballpark transformation
```

### 7.5.10 CompoundCRS to CompoundCRS

There is some similarity with the previous paragraph. We first research the vertical transformations between the two vertical CRS.

1. If there is such a transformation, be it direct, or if both vertical CRS relate to a common intermediate CRS. If it has a registered interpolation geographic CRS, then it is used. Otherwise we fallback to the geographic CRS of the source CRS.

   Finally, a 3-level loop to create the final set of operations chaining together:

   - the horizontal transformation from the source CRS to the interpolation CRS
   - the vertical transformation
   - the horizontal transformation from the interpolation CRS to the target CRS.

   Example:

```
$ projinfo -s "NAD27 + NGVD29 height (ftUS)" -t "NAD83 + NAVD88 height" --
↪spatial-test intersects --summary

Candidate operations found: 20
unknown id, NGVD29 height (ftUS) to NAVD88 height (3) + NAD27 to NAD83 (1),␣
↪0.17 m, USA - CONUS east of 89°W - onshore
unknown id, NGVD29 height (ftUS) to NAVD88 height (2) + NAD27 to NAD83 (1),␣
↪0.17 m, USA - CONUS 89°W-107°W - onshore
unknown id, NGVD29 height (ftUS) to NAVD88 height (1) + NAD27 to NAD83 (1),␣
↪0.17 m, USA - CONUS west of 107°W - onshore
unknown id, NGVD29 height (ftUS) to NAVD88 height (3) + NAD27 to NAD83 (3),␣
↪1.02 m, unknown domain of validity
unknown id, NGVD29 height (ftUS) to NAVD88 height (2) + NAD27 to NAD83 (3),␣
↪1.02 m, unknown domain of validity
unknown id, NGVD29 height (ftUS) to NAVD88 height (1) + NAD27 to NAD83 (3),␣
↪1.02 m, unknown domain of validity
unknown id, NGVD29 height (ftUS) to NAVD88 height (3) + NAD27 to NAD83 (5),␣
```

(continues on next page)

(continued from previous page)

```
↪1.02 m, unknown domain of validity, at least one grid missing
unknown id, NGVD29 height (ftUS) to NAVD88 height (3) + NAD27 to NAD83 (6),␣
↪1.52 m, unknown domain of validity, at least one grid missing
unknown id, NGVD29 height (ftUS) to NAVD88 height (2) + NAD27 to NAD83 (9),␣
↪1.52 m, unknown domain of validity, at least one grid missing
unknown id, NGVD29 height (ftUS) to NAVD88 height (1) + NAD27 to NAD83 (9),␣
↪1.52 m, unknown domain of validity, at least one grid missing
unknown id, NGVD29 height (ftUS) to NAVD88 height (3) + Ballpark geographic␣
↪offset from NAD27 to NAD83, unknown accuracy, USA - CONUS east of 89°W -␣
↪onshore, has ballpark transformation
unknown id, NGVD29 height (ftUS) to NAVD88 height (2) + Ballpark geographic␣
↪offset from NAD27 to NAD83, unknown accuracy, USA - CONUS 89°W-107°W -␣
↪onshore, has ballpark transformation
unknown id, NGVD29 height (ftUS) to NAVD88 height (1) + Ballpark geographic␣
↪offset from NAD27 to NAD83, unknown accuracy, USA - CONUS west of 107°W -␣
↪onshore, has ballpark transformation
unknown id, Transformation from NGVD29 height (ftUS) to NAVD88 height␣
↪(ballpark vertical transformation) + NAD27 to NAD83 (1), unknown accuracy,
↪ USA - CONUS including EEZ, has ballpark transformation
unknown id, Transformation from NGVD29 height (ftUS) to NAVD88 height␣
↪(ballpark vertical transformation) + NAD27 to NAD83 (3), unknown accuracy,
↪ Canada, has ballpark transformation
unknown id, Transformation from NGVD29 height (ftUS) to NAVD88 height␣
↪(ballpark vertical transformation) + NAD27 to NAD83 (4), unknown accuracy,
↪ Canada - NAD27, has ballpark transformation
unknown id, Transformation from NGVD29 height (ftUS) to NAVD88 height␣
↪(ballpark vertical transformation) + NAD27 to NAD83 (5), unknown accuracy,
↪ Canada - Quebec, has ballpark transformation, at least one grid missing
unknown id, Transformation from NGVD29 height (ftUS) to NAVD88 height␣
↪(ballpark vertical transformation) + NAD27 to NAD83 (6), unknown accuracy,
↪ Canada - Quebec, has ballpark transformation, at least one grid missing
unknown id, Transformation from NGVD29 height (ftUS) to NAVD88 height␣
↪(ballpark vertical transformation) + NAD27 to NAD83 (9), unknown accuracy,
↪ Canada - Saskatchewan, has ballpark transformation, at least one grid␣
↪missing
unknown id, Transformation from NGVD29 height (ftUS) to NAVD88 height␣
↪(ballpark vertical transformation) + Ballpark geographic offset from␣
↪NAD27 to NAD83, unknown accuracy, World, has ballpark transformation
```

2. Otherwise, when there is no such transformation, we decompose into 3 steps:

   - transform from the source CRS to the geographic 3D CRS corresponding to it
   - transform from the geographic 3D CRS corresponding to the source CRS to the geographic 3D CRS corresponding to the target CRS
   - transform from the geographic 3D CRS corresponding to the target CRS to the target CRS.

   Example:

```
$  projinfo -s "WGS 84 + EGM96 height" -t "ETRS89 + Belfast height" --
↪spatial-test intersects --summary

Candidate operations found: 7
```

(continues on next page)

(continued from previous page)

```
unknown id, Inverse of WGS 84 to EGM96 height (1) + Inverse of ETRS89 to␣
↪WGS 84 (1) + ETRS89 to Belfast height (2), 2.014 m, UK - Northern Ireland␣
↪- onshore
unknown id, Inverse of WGS 84 to EGM96 height (1) + Inverse of ETRS89 to␣
↪WGS 84 (1) + ETRS89 to Belfast height (1), 2.03 m, UK - Northern Ireland -
↪ onshore, at least one grid missing
unknown id, Inverse of WGS 84 to EGM96 height (1) + Null geographic offset␣
↪from WGS 84 (geog3D) to WGS 84 (geog2D) + Inverse of OSGB 1936 to WGS 84␣
↪(4) + OSGB 1936 to ETRS89 (2) + Null geographic offset from ETRS89␣
↪(geog2D) to ETRS89 (geog3D) + ETRS89 to Belfast height (2), 19.044 m,␣
↪unknown domain of validity
unknown id, Inverse of WGS 84 to EGM96 height (1) + Null geographic offset␣
↪from WGS 84 (geog3D) to WGS 84 (geog2D) + Inverse of OSGB 1936 to WGS 84␣
↪(2) + OSGB 1936 to ETRS89 (2) + Null geographic offset from ETRS89␣
↪(geog2D) to ETRS89 (geog3D) + ETRS89 to Belfast height (2), 11.044 m,␣
↪unknown domain of validity
unknown id, Inverse of WGS 84 to EGM96 height (1) + Null geographic offset␣
↪from WGS 84 (geog3D) to WGS 84 (geog2D) + Inverse of TM75 to WGS 84 (2) +␣
↪TM75 to ETRS89 (3) + Null geographic offset from ETRS89 (geog2D) to␣
↪ETRS89 (geog3D) + ETRS89 to Belfast height (2), 2.424 m, UK - Northern␣
↪Ireland - onshore, at least one grid missing
unknown id, Inverse of WGS 84 to EGM96 height (1) + Null geographic offset␣
↪from WGS 84 (geog3D) to WGS 84 (geog2D) + Inverse of TM75 to WGS 84 (2) +␣
↪TM75 to ETRS89 (3) + Null geographic offset from ETRS89 (geog2D) to␣
↪ETRS89 (geog3D) + ETRS89 to Belfast height (1), 2.44 m, UK - Northern␣
↪Ireland - onshore, at least one grid missing
unknown id, Inverse of WGS 84 to EGM96 height (1) + Null geographic offset␣
↪from WGS 84 (geog3D) to WGS 84 (geog2D) + Inverse of OSGB 1936 to WGS 84␣
↪(4) + OSGB 1936 to ETRS89 (2) + Null geographic offset from ETRS89␣
↪(geog2D) to ETRS89 (geog3D) + ETRS89 to Belfast height (1), 19.06 m,␣
↪unknown domain of validity, at least one grid missing
```

This is implemented by the `createOperationsCompoundToCompound` method

### 7.5.11 When the source or target CRS is a BoundCRS

The BoundCRS concept is an hybrid concept where a CRS is linked to a transformation from it to a hub CRS, typically WGS 84. This is a long-time practice in PROJ.4 strings with the `+towgs84`, `+nadgrids` and `+geoidgrids` keywords, or the `TOWGS84[]` node of WKT 1. When encountering those attributes when parsing a CRS string, PROJ will create a BoundCRS object capturing this transformation. A BoundCRS object can also be provided with a WKT2 string, and in that case with a hub CRS being potentially different from WGS 84.

Let's consider the case of a transformation between a BoundCRS ("+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100000 +ellps=airy +towgs84=446.448,-125.157,542.06,0.15,0.247,0.842,-20.489 +units=m" which used to be the PROJ.4 definition of "OSGB 1936 / British National Grid") and a target Geographic CRS, ETRS89.

We apply the following steps:

- transform from the base of the source CRS (that is the CRS wrapped by BoundCRS, here a ProjectedCRS) to the geographic CRS of this base CRS
- apply the transformation of the BoundCRS to go from the geographic CRS of this base CRS to the hub CRS of the BoundCRS, in that instance WGS 84.

- apply a transformation from the hub CRS to the target CRS.

This is implemented by the `createOperationsBoundToGeog` method

Example:

```
$ projinfo -s "+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100000␣
↪+ellps=airy +towgs84=446.448,-125.157,542.06,0.15,0.247,0.842,-20.489 +units=m␣
↪+type=crs" -t ETRS89 -o PROJ

Candidate operations found: 1
-------------------------------------
Operation No. 1:

unknown id, Inverse of unknown + Transformation from unknown to WGS84 + Inverse of␣
↪ETRS89 to WGS 84 (1), unknown accuracy, Europe - ETRS89

PROJ string:
+proj=pipeline +step +inv +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_
↪0=-100000 +ellps=airy +step +proj=push +v_3 +step +proj=cart +ellps=airy +step␣
↪+proj=helmert +x=446.448 +y=-125.157 +z=542.06 +rx=0.15 +ry=0.247 +rz=0.842 +s=-20.489␣
↪+convention=position_vector +step +inv +proj=cart +ellps=GRS80 +step +proj=pop +v_3␣
↪+step +proj=unitconvert +xy_in=rad +xy_out=deg +step +proj=axisswap +order=2,1
```

There are other situations with BoundCRS, involving vertical transformations, or transforming to other objects than a geographic CRS, but the curious reader will have to inspect the code for the actual gory details.

### 7.5.12 Using Derived Projected for source or target CRS

The *WKT2* tag `DERIVEDPROJCRS` can be useful to define a customized CRS adding a `DERIVINGCONVERSION` to apply a conversion on top of the projection. It can be also done inside a Compound CRS. One use case is to describe a local CRS produced in a site calibration, as explained in [JimenezShaw2023].

Using the WKT2 from that paper (stored in wkt2.txt file for readability) we would get this:

```
projinfo -s EPSG:6319 -t "`cat wkt2.txt`" -o proj
Candidate operations found: 1
-------------------------------------
Operation No. 1:

unknown id, Inverse of Transformation from Ellipsoid (metre) to NAD83(2011) (ballpark␣
↪vertical transformation, without ellipsoid height to vertical height correction) +␣
↪Conv Vertical Offset and Slope + Transverse Mercator + Affine transformation as PROJ-
↪based, unknown accuracy, World, has ballpark transformation

PROJ string:
+proj=pipeline
+step +proj=axisswap +order=2,1
+step +proj=unitconvert +xy_in=deg +xy_out=rad
+step +proj=vertoffset +lat_0=41.2305352787143 +lon_0=-73.1815861874286
        +dh=31.0121985701957 +slope_lat=-6.12572852418232
        +slope_lon=-2.67487863214139
+step +proj=tmerc +lat_0=41.2305352787143 +lon_0=-73.1815861874286 +k=1 +x_0=0
        +y_0=0 +ellps=GRS80
+step +proj=affine +xoff=265262.95287 +yoff=196619.27389 +s11=1.00003994119
        +s12=0.00548156923529 +s21=-0.00548156923529 +s22=1.00003994119
```

CHAPTER

EIGHT

# RESOURCE FILES

A number of files containing preconfigured transformations and default parameters for certain projections are bundled with the PROJ distribution. Init files contain preconfigured proj-strings for various coordinate reference systems and the *defaults* file contains default values for parameters of select projections.

In addition to the bundled init files the PROJ project also distributes a number of packages containing transformation grids and additional init files not included in the main PROJ package.

## 8.1 Where are PROJ resource files looked for ?

PROJ will attempt to locate its resource files - database, transformation grids or init files - from several directories. The following paths are checked in order:

- For resource files that have an explicit relative or absolute path, the directory specified in the filename.
- Path resolved by the callback function set with the *proj_context_set_file_finder()*. If it is set, the next tests will not be run.
- Path(s) set with the *proj_context_set_search_paths()*. If set, the next tests will not be run.
- Added in version 7.0.

  The PROJ user writable directory, which is :

  - on Windows, `%LOCALAPPDATA%\proj`
  - on macOS, `${HOME}/Library/Application Support/proj`
  - on other platforms (Linux), `${XDG_DATA_HOME}/proj` if `XDG_DATA_HOME` is defined. Else `${HOME}/.local/share/proj`

  Applications may also customize the user writable directory using `proj_context_set_user_writable_directory()` (since 9.5)
- Path(s) set with by the environment variable *PROJ_DATA*. Prior to PROJ 9.1, this environment variable was called `PROJ_LIB`. This older name is still supported in PROJ 9.1 as a fallback, but support for it may be removed in future release. On Linux/macOS/Unix, use `:` to separate paths. On Windows, `;`
- Added in version 7.0.

  The `../share/proj/` and its contents are found automatically at run-time if the installation respects the build structure. That is, the binaries and `proj.dll/libproj.so` are installed under `../bin/` or `../lib/`, and resource files are in `../share/proj/`.
- A path built into PROJ as its resource installation directory (whose value is `$(pkgdatadir)` for builds using the Makefile build system or `${CMAKE_INSTALL_PREFIX}/${DATADIR}` for CMake builds). Note, however, that since this is a hard-wired path setting, it only works if the whole PROJ installation is not moved somewhere else.

> **Note**
>
> If PROJ is built with the PROJ_DATA_ENV_VAR_TRIED_LAST CMake option (called PROJ_LIB_ENV_VAR_TRIED_LAST before PROJ 9.1), then this hard-wired path will be tried before looking at the environment variable *PROJ_DATA*.

- The current directory

When networking capabilities are enabled, either by API with the *proj_context_set_enable_network()* function or when the *PROJ_NETWORK* environment variable is set to ON, PROJ will attempt to use remote grids stored on CDN (Content Delivery Network) storage.

## 8.2 proj.db

A proj installation includes a SQLite database of transformation information that must be accessible for the library to work properly. The library will print an error if the database can't be found.

The database may be customized/reduced by deleting entries not relevant for a certain use-case. An example for a simple SQL script removing all entries not related to 'WGS 84' ellipsoid:

```
PRAGMA FOREIGN_KEYS=1;
DELETE FROM ellipsoid WHERE name != 'WGS 84';
-- clean up table usage
DELETE FROM usage WHERE (object_table_name, object_auth_name, object_code) IN (
  SELECT object_table_name, object_auth_name, object_code FROM usage WHERE NOT EXISTS (
    SELECT 1 FROM object_view o WHERE
        o.table_name = object_table_name AND
        o.auth_name = object_auth_name AND
        o.code = object_code));
VACUUM;
PRAGMA foreign_key_check;
```

## 8.3 proj.ini

Added in version 7.0.

proj.ini is a text configuration file, mostly dedicated at setting up network related parameters.

Its default content is:

```
[general]
; Lines starting by ; are commented lines.
;

; Network capabilities disabled by default.
; Can be overridden with the PROJ_NETWORK=ON environment variable.
; Cf https://proj.org/en/latest/usage/network.html
; Valid values = on, off
network = off

; Endpoint of the Content Delivery Network where remote resources might
; be accessed. Only used if network access is allowed (cf above "network"
; option)
```

(continues on next page)

(continued from previous page)

```
; Can be overridden with the PROJ_NETWORK_ENDPOINT environment variable.
cdn_endpoint = https://cdn.proj.org

; Whether to enable a cache of remote resources that are accessed, on the
; local file system
; Valid values = on, off
cache_enabled = on

; Size of the cache in megabytes
cache_size_MB = 300

; Time-to-live delay in seconds before already accessed remote resources are
; accessed again to check if they have been updated.
cache_ttl_sec = 86400

; Can be set to on so that by default the lack of a known resource files needed
; for the best transformation PROJ would normally use causes an error, or off
; to accept missing resource files without errors or warnings.
; This default value itself is overridden by the PROJ_ONLY_BEST_DEFAULT environment
; variable if set, and then by the ONLY_BEST setting that can be
; passed to the proj_create_crs_to_crs() method, or with the --only-best
; option of the cs2cs program.
; (added in PROJ 9.2)
; Valid values = on, off
only_best_default = off

; Filename of the Certificate Authority (CA) bundle.
; Can be overridden with the PROJ_CURL_CA_BUNDLE / CURL_CA_BUNDLE environment variable.
; (added in PROJ 9.0)
; ca_bundle_path = /path/to/cabundle.pem

; When this is set to on, the operating systems native CA store will be used for␣
↪certificate verification
; If you set this option to on and also set ca_bundle_path then during verification␣
↪those certificates are
; searched in addition to the native CA store.
; (added in PROJ 9.6)
; Valid values = on, off
;native_ca = on


; Transverse Mercator (and UTM)  default algorithm: auto, evenden_snyder or poder_
↪engsager
; * evenden_snyder is the fastest, but less accurate far from central meridian
; * poder_engsager is slower, but more accurate far from central meridian
; * default will auto-select between the two above depending on the coordinate
;   to transform and will use evenden_snyder if the error in doing so is below
;   0.1 mm (for an ellipsoid of the size of Earth)
tmerc_default_algo = poder_engsager
```

## 8.4 Transformation grids

Grid files are important for shifting and transforming between datums.

PROJ supports CTable2, NTv1 and NTv2 files for horizontal grid corrections and the GTX file format for vertical corrections. Details about the formats can be found in the GDAL documentation. GDAL reads and writes all formats. Using GDAL for construction of new grids is recommended.

## 8.5 External resources and packaged grids

### 8.5.1 proj-data

The `proj-data` package is a collection of all the resource files that are freely available for use with PROJ. The package is maintained on GitHub and the contents of the package are show-cased on the PROJ CDN. The contents of the package can be installed using the *projsync* package or by downloading the zip archive of the package and unpacking in the *PROJ_DATA* directory.

### 8.5.2 proj-datumgrid

**Note**

The packages described below can be used with PROJ 7 and later but are primarily meant to be used with PROJ 6 and earlier versions. The `proj-datumgrid` series of packages are not maintained anymore and are only kept available for legacy purposes.

For a functioning build of PROJ prior to version 7, installation of the proj-datumgrid is needed. If you have installed PROJ from a package system chances are that this will already be done for you. The *proj-datumgrid* package provides transformation grids that are essential for many of the predefined transformations in PROJ. Which grids are included in the package can be seen on the proj-datumgrid repository as well as descriptions of those grids. This is the main grid package and the only one that is required. It includes various older grids that is mostly needed for legacy reasons. Without this package, the test suite fails miserably.

### 8.5.3 Regional packages

In addition to the default `proj-datumgrid` package regional packages are also distributed. These include grids and init files that are valid within the given region. The packages are divided into geographical regions in order to keep the needed disk space by PROJ at a minimum. Some users may have a use for resource files covering several regions in which case they can download more than one.

At the moment three regional resource file packages are distributed:

- Europe
- Oceania
- North America

If someone supplies grids relevant for Africa, South-America, Asia or Antarctica we will create new regional packages.

Click the links to jump to the relevant README files for each package. Details on the content of the packages maintained there.

**Tip**

To download the various datumgrid packages head to the *download section*.

### 8.5.4 World package

The world package includes grids that have global extent, e.g. the global geoid model EGM08.

### 8.5.5 -latest packages

All packages above come in different versions, e.g., `proj-datumgrid-1.8` or `proj-datumgrid-europe-1.4`. The `-latest` packages are symbolic links to the latest version of a given package. That means that the link https://download.osgeo.org/proj/proj-datumgrid-north-america-latest.zip is equivalent to https://download.osgeo.org/proj/proj-datumgrid-north-america-1.2.zip (as of the time of writing this).

## 8.6 Other transformation grids

Below is a list of grid resources for various countries which are not included in the grid distributions mentioned above.

### 8.6.1 Free grids

The following is a list of grids distributed under a free and open license.

#### 8.6.1.1 Hungary

Hungarian grid ETRS89 - HD72/EOV (epsg:23700), both horizontal and elevation grids

### 8.6.2 Non-Free Grids

Not all grid shift files have licensing that allows them to be freely distributed, but can be obtained by users through free and legal methods.

#### 8.6.2.1 Austria

Overview of Austrian grids and other resources related to the local geodetic reference.

#### 8.6.2.2 Brazil

Brazilian grids for datums Corrego Alegre 1961, Corrego Alegre 1970-72, SAD69 and SAD69(96)

#### 8.6.2.3 Netherlands

Dutch grid (Registration required before download)

#### 8.6.2.4 Portugal

Portuguese grids for ED50, Lisbon 1890, Lisbon 1937 and Datum 73

### 8.6.3 HTDP

This section describes the use of the `crs2crs2grid.py` script and the HTDP (Horizontal Time Dependent Positioning) grid shift modelling program from NGS/NOAA to produce PROJ compatible grid shift files for fine grade conversions between various NAD83 epochs and WGS84. Traditionally PROJ has treated NAD83 and WGS84 as equivalent and failed to distinguish between different epochs or realizations of those datums. At the scales of much mapping this is adequate but as interest grows in high resolution imagery and other high resolution mapping this is inadequate. Also, as the North American crust drifts over time the displacement between NAD83 and WGS84 grows (more than one foot over the last two decades).

#### 8.6.3.1 Getting and building HTDP

The HTDP modelling program is written in FORTRAN. The source and documentation can be found on the HTDP page at http://www.ngs.noaa.gov/TOOLS/Htdp/Htdp.shtml

On Linux systems it will be necessary to install GFortran or some Fortran compiler. For Ubuntu something like the following should work.

```
apt-get install gfortran
```

To compile the program do something like the following to produce the binary `htdp` from the source code.

```
gfortran htdp.for -o htdp
```

#### 8.6.3.2 Getting `crs2crs2grid.py`

The `crs2crs2grid.py` script can be found at https://github.com/OSGeo/gdal/blob/master/swig/python/gdal-utils/osgeo_utils/samples/crs2crs2grid.py

The script depends on having the GDAL Python bindings operational; if they are not you will get an error such as:

```
Traceback (most recent call last):
  File "./crs2crs2grid.py", line 37, in <module>
    from osgeo import gdal, gdal_array, osr
ImportError: No module named osgeo
```

#### 8.6.3.3 Usage

```
crs2crs2grid.py
        <src_crs_id> <src_crs_date> <dst_crs_id> <dst_crs_year>
        [-griddef <ul_lon> <ul_lat> <ll_lon> <ll_lat> <lon_count> <lat_count>]
        [-htdp <path_to_exe>] [-wrkdir <dirpath>] [-kwf]
        -o <output_grid_name>

 -griddef: by default the following values for roughly the continental USA
           at a six minute step size are used:
           -127 50 -66 25 251 611
 -kwf: keep working files in the working directory for review.
```

```
crs2crs2grid.py 29 2002.0 8 2002.0 -o nad83_2002.ct2
```

The goal of `crs2crs2grid.py` is to produce a grid shift file for a designated region. The region is defined using the `-griddef` switch. When missing a continental US region is used. The script creates a set of sample points for the grid definition, runs `htdp` against it and then parses the resulting points and computes a point by point shift to encode into the final grid shift file. By default it is assumed that `htdp` is in the executable path. If not, please provide the path to the executable using the `-htdp` switch.

The `htdp` program supports transformations between many CRSes and for each (or most?) of them you need to provide a date at which the CRS is fixed. The full set of CRS Ids available in the HTDP program are:

```
 1...NAD_83(2011) (North America tectonic plate fixed)
 29...NAD_83(CORS96)  (NAD_83(2011) will be used)
 30...NAD_83(2007)    (NAD_83(2011) will be used)
 2...NAD_83(PA11) (Pacific tectonic plate fixed)
 31...NAD_83(PACP00)  (NAD 83(PA11) will be used)
```

(continues on next page)

(continued from previous page)

```
 3...NAD_83(MA11) (Mariana tectonic plate fixed)
 32...NAD_83(MARP00)  (NAD_83(MA11) will be used)

 4...WGS_72                              16...ITRF92
 5...WGS_84(transit) = NAD_83(2011)      17...ITRF93
 6...WGS_84(G730) = ITRF92               18...ITRF94 = ITRF96
 7...WGS_84(G873) = ITRF96               19...ITRF96
 8...WGS_84(G1150) = ITRF2000            20...ITRF97
 9...PNEOS_90 = ITRF90                   21...IGS97 = ITRF97
10...NEOS_90 = ITRF90                    22...ITRF2000
11...SIO/MIT_92 = ITRF91                 23...IGS00 = ITRF2000
12...ITRF88                              24...IGb00 = ITRF2000
13...ITRF89                              25...ITRF2005
14...ITRF90                              26...IGS05 = ITRF2005
15...ITRF91                              27...ITRF2008
                                         28...IGS08 = ITRF2008
```

The typical use case is mapping from NAD83 on a particular date to WGS84 on some date. In this case the source CRS Id "29" (NAD_83(CORS96)) and the destination CRS Id is "8 (WGS_84(G1150)). It is also necessary to select the source and destination date (epoch). For example:

```
crs2crs2grid.py 29 2002.0 8 2002.0 -o nad83_2002.ct2
```

The output is a CTable2 format grid shift file suitable for use with PROJ (4.8.0 or newer). It might be utilized something like:

```
cs2cs +proj=latlong +ellps=GRS80 +nadgrids=./nad83_2002.ct2 +to +proj=latlong␣
↪+datum=WGS84
```

#### 8.6.3.4 See Also

- http://www.ngs.noaa.gov/TOOLS/Htdp/Htdp.shtml - NGS/NOAA page about the HTDP model and program. Source for the HTDP program can be downloaded from here.

## 8.7 Init files

Init files are used for preconfiguring proj-strings for often used transformations, such as those found in the EPSG database. Most init files contain transformations from a given coordinate reference system to WGS84. This makes it easy to transform between any two coordinate reference systems with *cs2cs*. Init files can however contain any proj-string and don't necessarily have to follow the *cs2cs* paradigm where WGS84 is used as a pivot datum. The ITRF init file is a good example of that.

A number of init files come pre-bundled with PROJ but it is also possible to add your own custom init files. PROJ looks for the init files in the directory listed in the *PROJ_DATA* environment variable.

The format of init files is an identifier in angled brackets and a proj-string:

```
<3819> +proj=longlat +ellps=bessel
       +towgs84=595.48,121.69,515.35,4.115,-2.9383,0.853,-3.408 +no_defs <>
```

The above example is the first entry from the `epsg` init file. So, this is the coordinate reference system with ID 3819 in the EPSG database. Comments can be inserted by prefixing them with a "#". With version 4.10.0 a new special metadata entry is now accepted in init files. It can be parsed with a function from the public API. The metadata entry in the epsg init file looks like this at the time of writing:

```
<metadata> +version=9.0.0 +origin=EPSG +lastupdate=2017-01-10
```

Pre-configured proj-strings from init files are used in the following way:

```
$ cs2cs -v +proj=latlong +to +init=epsg:3819
# ---- From Coordinate System ----
#Lat/long (Geodetic alias)
#
# +proj=latlong +ellps=WGS84
# ---- To Coordinate System ----
#Lat/long (Geodetic alias)
#
# +init=epsg:3819 +proj=longlat +ellps=bessel
# +towgs84=595.48,121.69,515.35,4.115,-2.9383,0.853,-3.408 +no_defs
```

It is possible to override parameters when using `+init`. Just add the parameter to the proj-string alongside the `+init` parameter. For instance by overriding the ellipsoid as in the following example

```
+init=epsg:25832 +ellps=intl
```

where the Hayford ellipsoid is used instead of the predefined GRS80 ellipsoid.

It is also possible to add additional parameters not specified in the init file, for instance by adding a central epoch when applying the ITRF2014:NOAM plate motion model:

```
+init=ITRF2014:NOAM +t_epoch=2010.0
```

which then expands to

```
+proj=helmert +drx=0.000024 +dry=-0.000694 +drz=-0.000063 +convention=position_vector +t_
↪epoch=2010.0
```

Below is a list of the init files that are packaged with PROJ.

| Name | Description |
|---|---|
| GL27 | Great Lakes Grids |
| ITRF2000 | Full set of transformation parameters between ITRF2000 and other ITRF's |
| ITRF2008 | Full set of transformation parameters between ITRF2008 and other ITRF's |
| ITRF2014 | Full set of transformation parameters between ITRF2014 and other ITRF's |
| nad27 | State plane coordinate systems, North American Datum 1927 |
| nad83 | State plane coordinate systems, North American Datum 1983 |

CHAPTER

NINE

# GEODESIC CALCULATIONS

## 9.1 Introduction

Consider an ellipsoid of revolution with equatorial radius $a$, polar semi-axis $b$, and flattening $f = (a - b)/a$. Points on the surface of the ellipsoid are characterized by their latitude $\phi$ and longitude $\lambda$. (Note that latitude here means the *geographical latitude*, the angle between the normal to the ellipsoid and the equatorial plane).

The shortest path between two points on the ellipsoid at $(\phi_1, \lambda_1)$ and $(\phi_2, \lambda_2)$ is called the geodesic. Its length is $s_{12}$ and the geodesic from point 1 to point 2 has forward azimuths $\alpha_1$ and $\alpha_2$ at the two end points. In this figure, we have $\lambda_{12} = \lambda_2 - \lambda_1$.

A geodesic can be extended indefinitely by requiring that any sufficiently small segment is a shortest path; geodesics are also the straightest curves on the surface.

## 9.2 Solution of geodesic problems

Traditionally two geodesic problems are considered:

- the direct problem — given $\phi_1$, $\lambda_1$, $\alpha_1$, $s_{12}$, determine $\phi_2$, $\lambda_2$, $\alpha_2$.
- the inverse problem — given $\phi_1$, $\lambda_1$, $\phi_2$, $\lambda_2$, determine $s_{12}$, $\alpha_1$, $\alpha_2$.

PROJ incorporates C library for Geodesics. This library provides routines to solve the direct and inverse geodesic problems. Full double precision accuracy is maintained provided that $|f| < \frac{1}{50}$.

The interface to the geodesic routines differ in two respects from the rest of PROJ:

- angles (latitudes, longitudes, and azimuths) are in degrees (instead of in radians);
- the shape of ellipsoid is specified by the flattening $f$; this can be negative to denote a prolate ellipsoid; setting $f = 0$ corresponds to a sphere, in which case the geodesic becomes a great circle.

PROJ also includes a command line tool, *geod*(1), for performing simple geodesic calculations.

## 9.3 Additional properties

The routines also calculate several other quantities of interest

- $S_{12}$ is the area between the geodesic from point 1 to point 2 and the equator; i.e., it is the area, measured counter-clockwise, of the quadrilateral with corners $(\phi_1, \lambda_1)$, $(0, \lambda_1)$, $(0, \lambda_2)$, and $(\phi_2, \lambda_2)$. It is given in meters$^2$.
- $m_{12}$, the reduced length of the geodesic is defined such that if the initial azimuth is perturbed by $d\alpha_1$ (radians) then the second point is displaced by $m_{12}\, d\alpha_1$ in the direction perpendicular to the geodesic. $m_{12}$ is given in

meters. On a curved surface the reduced length obeys a symmetry relation, $m_{12} + m_{21} = 0$. On a flat surface, we have $m_{12} = s_{12}$.

- $M_{12}$ and $M_{21}$ are geodesic scales. If two geodesics are parallel at point 1 and separated by a small distance $dt$, then they are separated by a distance $M_{12}\,dt$ at point 2. $M_{21}$ is defined similarly (with the geodesics being parallel to one another at point 2). $M_{12}$ and $M_{21}$ are dimensionless quantities. On a flat surface, we have $M_{12} = M_{21} = 1$.
- $\sigma_{12}$ is the arc length on the auxiliary sphere. This is a construct for converting the problem to one in spherical trigonometry. The spherical arc length from one equator crossing to the next is always $180^\circ$.

If points 1, 2, and 3 lie on a single geodesic, then the following addition rules hold:

- $s_{13} = s_{12} + s_{23}$,
- $\sigma_{13} = \sigma_{12} + \sigma_{23}$,
- $S_{13} = S_{12} + S_{23}$,
- $m_{13} = m_{12}M_{23} + m_{23}M_{21}$,
- $M_{13} = M_{12}M_{23} - (1 - M_{12}M_{21})m_{23}/m_{12}$,
- $M_{31} = M_{32}M_{21} - (1 - M_{23}M_{32})m_{12}/m_{23}$.

## 9.4 Multiple shortest geodesics

The shortest distance found by solving the inverse problem is (obviously) uniquely defined. However, in a few special cases there are multiple azimuths which yield the same shortest distance. Here is a catalog of those cases:

- $\phi_1 = -\phi_2$ (with neither point at a pole). If $\alpha_1 = \alpha_2$, the geodesic is unique. Otherwise there are two geodesics and the second one is obtained by setting $[\alpha_1, \alpha_2] \leftarrow [\alpha_2, \alpha_1]$, $[M_{12}, M_{21}] \leftarrow [M_{21}, M_{12}]$, $S_{12} \leftarrow -S_{12}$. (This occurs when the longitude difference is near $\pm 180^\circ$ for oblate ellipsoids.)
- $\lambda_2 = \lambda_1 \pm 180^\circ$ (with neither point at a pole). If $\alpha_1 = 0^\circ$ or $\pm 180^\circ$, the geodesic is unique. Otherwise there are two geodesics and the second one is obtained by setting $[\alpha_1, \alpha_2] \leftarrow [-\alpha_1, -\alpha_2]$, $S_{12} \leftarrow -S_{12}$. (This occurs when $\phi_2$ is near $-\phi_1$ for prolate ellipsoids.)
- Points 1 and 2 at opposite poles. There are infinitely many geodesics which can be generated by setting $[\alpha_1, \alpha_2] \leftarrow [\alpha_1, \alpha_2] + [\delta, -\delta]$, for arbitrary $\delta$. (For spheres, this prescription applies when points 1 and 2 are antipodal.)
- $s_{12} = 0$ (coincident points). There are infinitely many geodesics which can be generated by setting $[\alpha_1, \alpha_2] \leftarrow [\alpha_1, \alpha_2] + [\delta, \delta]$, for arbitrary $\delta$.

## 9.5 Area of a polygon

The area of a geodesic polygon can be determined by summing $-S_{12}$ for successive edges of the polygon ($S_{12}$ is negated so that clockwise traversal of a polygon gives a positive area). However, if the polygon encircles a pole, the sum must be adjusted by $\pm A/2$, where $A$ is the area of the full ellipsoid, with the sign chosen to place the result in $(-A/2, A/2]$.

## 9.6 Background

The algorithms implemented by this package are given in [Karney2013] (addenda) and are based on [Bessel1825] and [Helmert1880]; the algorithm for areas is based on [Danielsen1989]. These improve on the work of [Vincenty1975] in the following respects:

- The results are accurate to round-off for terrestrial ellipsoids (the error in the distance is less than 15 nanometers, compared to 0.1 mm for Vincenty).

- The solution of the inverse problem is always found. (Vincenty's method fails to converge for nearly antipodal points.)
- The routines calculate differential and integral properties of a geodesic. This allows, for example, the area of a geodesic polygon to be computed.

Additional background material is provided in GeographicLib's geodesic bibliography, Wikipedia's article "Geodesics on an ellipsoid", and [Karney2011] (errata).

CHAPTER

# TEN

# DEVELOPMENT

These pages are primarily focused towards developers either contributing to the PROJ project or using the library in their own software.

## 10.1 Development practices

### 10.1.1 Commit messages

Commit messages should indicate a component name (eg a driver name), a short description, and when relevant, a reference to a issue (with 'fixes #' if it actually fixes it)

```
COMPONENT_NAME: fix bla bla (fixes #1234)

Details here...
```

### 10.1.2 Commit hooks

PROJ provides pre-commit hooks to run code linters before a commit is made. The hooks are cloned with the repository and can be installed using pre-commit:

```
python -m pip install pre-commit
pre-commit install
```

Once installed, the hooks can be run manually via `pre-commit run --all-files`.

### 10.1.3 Blame ignore file

Due to whole-tree code reformatting done during PROJ 9.2 development, `git blame` information might be misleading. To avoid that, you need to modify your git configuration as following to ignore the revision of the whole-tree reformatting:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

## 10.2 Quick start for C API usage

This is a short introduction to the PROJ C API. In the following section we create two simple programs that illustrate how to transform points between two different coordinate systems, and how to convert between projected and geodetic (geographic) coordinates for a single coordinate system. Explanations for individual code sniplets and the full programs are provided.

See the following sections for more in-depth descriptions of different parts of the PROJ API or consult the *API reference* for specifics.

Before the PROJ API can be used it is necessary to include the `proj.h` header file. Here `stdio.h` is also included so we can print some text to the screen:

```
#include <proj.h>
#include <stdio.h>
```

Let's declare a few variables that'll be used later in the program. Each variable will be discussed below. See the *reference for more info on data types*.

```
PJ_CONTEXT *C;
PJ *P;
PJ *norm;
PJ_COORD a, b;
```

For use in multi-threaded programs the *PJ_CONTEXT* threading-context is used. In this particular example it is not needed, but for the sake of completeness we demonstrate its use here.

```
C = proj_context_create();
```

Next we create the *PJ* transformation object `P` with the function *proj_create_crs_to_crs()*.

```
P = proj_create_crs_to_crs(
    C, "EPSG:4326", "+proj=utm +zone=32 +datum=WGS84", /* or EPSG:32632 */
    NULL);

if (0 == P) {
    fprintf(stderr, "Failed to create transformation object.\n");
    return 1;
}
```

Here we have set up a transformation from geographic coordinates to UTM zone 32N. In general, this is a transformation between two different coordinate reference systems (one of which is here in geographic coordinates). The related function *proj_create()* can be used to set up transformations that are not available through the PROJ database, for instance for converting geodetic coordinates to a custom definition of a map projection.

*proj_create_crs_to_crs()* takes as its arguments:

- the threading context `C` created above,
- a string that describes the source coordinate reference system (CRS),
- a string that describes the target CRS and
- an optional description of the area of use.

It is recommended to create one threading context per thread used by the program. This ensures that all *PJ* objects created in the same context will be sharing resources such as error-numbers and loaded grids.

If you are sure that `P` will only be used by a single program thread, you may pass `NULL` for the threading context. This will assign the default thread context to `P`.

The strings for the source and target CRS may be any of:

- PROJ strings, e.g. `+proj=longlat +datum=WGS84 +type=crs`,
- CRS identified by their code, e.g. `EPSG:4326` or `urn:ogc:def:crs:EPSG::4326`, or
- a well-known text (WKT) string, e.g.:

```
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    USAGE[
        SCOPE["unknown"],
        AREA["World"],
        BBOX[-90,-180,90,180]],
    ID["EPSG",4326]]
```

**Warning**

The use of PROJ strings to describe a CRS is not recommended. One of the main weaknesses of PROJ strings is their inability to describe a geodetic datum, other than the few ones hardcoded in the `+datum` parameter.

*proj_create_crs_to_crs()* will return a pointer to a *PJ* object, or a null pointer in the case of an error. The details of the error can be retrieved using *proj_context_errno()*. See *Error handling* for further details.

Now that we have a normalized transformation object in P, we can use it with *proj_trans()* to transform coordinates from the source CRS to the target CRS, but first we will discuss the interpretation of coordinates.

By default, a *PJ* transformation object accepts coordinates expressed in the units and axis order of the source CRS, and returns transformed coordinates in the units and axis order of the target CRS.

For most geographic CRS, the units will be in degrees. In rare cases, such as EPSG:4807 / NTF (Paris), this can be grads. For geographic CRS defined by the EPSG authority, the order of coordinates is latitude first, longitude second. When using a PROJ string, the order is the reverse; longitude first, latitude second.

For projected CRS, the units may vary (metre, us-foot, etc.). For projected CRS defined by the EPSG authority, and with EAST / NORTH directions, the order might be easting first, northing second, or the reverse. When using a PROJ string, the order will be easting first, northing second, except if the `+axis` parameter modifies it.

If you prefer to work with a uniform axis order, regardless of the axis orders mandated by the source and target CRS, you can use the *proj_normalize_for_visualization()* function.

*proj_normalize_for_visualization()* takes a threading context and an existing *PJ* object, and generates from it a new *PJ* that accepts as input and returns as output coordinates using the traditional GIS order. That is, longitude followed by latitude, optionally followed by elevation and time for geographic CRS, and easting followed by northing for most projected CRS.

```
if (0 == norm) {
    fprintf(stderr, "Failed to normalize transformation object.\n");
    return 1;
}
```

(continues on next page)

(continued from previous page)

```
proj_destroy(P);
P = norm;
```

Next we create a *PJ_COORD* coordinate object, using the function *proj_coord()*.

The following example creates a coordinate for 55°N 12°E (Copenhagen).

Because we have normalized the transformation object with *proj_normalize_for_visualization()*, the order of coordinates is longitude followed by latitude, and the units are degrees.

```
a = proj_coord(12, 55, 0, 0);
```

Now we are ready to transform the coordinate into UTM zone 32, using the function *proj_trans()*.

```
b = proj_trans(P, PJ_FWD, a);
printf("easting: %.3f, northing: %.3f\n", b.enu.e, b.enu.n);
```

*proj_trans()* takes as its arguments:

- a *PJ* transformation object,
- a *PJ_DIRECTION* direction, and
- the *PJ_COORD* coordinate to transform.

The direction argument can be one of:

- `PJ_FWD` -- "forward" transformation from source CRS to target CRS.
- `PJ_IDENT` -- "identity", return the source coordinate unchanged.
- `PJ_INV` -- "inverse" transformation from target CRS to source CRS.

It returns the new transformed *PJ_COORD* coordinate.

We can perform the transformation in reverse (from UTM zone 32 back to geographic) as follows:

```
b = proj_trans(P, PJ_INV, b);
printf("longitude: %g, latitude: %g\n", b.lp.lam, b.lp.phi);
```

Before ending the program, we need to release the memory allocated to our objects:

```
proj_destroy(P);
proj_context_destroy(C); /* may be omitted in the single threaded case */
```

A complete compilable version of the example code can be seen below:

```
#include <proj.h>
#include <stdio.h>

int main(void) {
    PJ_CONTEXT *C;
    PJ *P;
    PJ *norm;
    PJ_COORD a, b;

    /* or you may set C=PJ_DEFAULT_CTX if you are sure you will     */
    /* use PJ objects from only one thread                          */
```

(continues on next page)

(continued from previous page)

```
    C = proj_context_create();

    P = proj_create_crs_to_crs(
        C, "EPSG:4326", "+proj=utm +zone=32 +datum=WGS84", /* or EPSG:32632 */
        NULL);

    if (0 == P) {
        fprintf(stderr, "Failed to create transformation object.\n");
        return 1;
    }

    /* This will ensure that the order of coordinates for the input CRS */
    /* will be longitude, latitude, whereas EPSG:4326 mandates latitude, */
    /* longitude */
    norm = proj_normalize_for_visualization(C, P);
    if (0 == norm) {
        fprintf(stderr, "Failed to normalize transformation object.\n");
        return 1;
    }
    proj_destroy(P);
    P = norm;

    /* a coordinate union representing Copenhagen: 55d N, 12d E */
    /* Given that we have used proj_normalize_for_visualization(), the order */
    /* of coordinates is longitude, latitude, and values are expressed in */
    /* degrees. */
    a = proj_coord(12, 55, 0, 0);

    /* transform to UTM zone 32, then back to geographical */
    b = proj_trans(P, PJ_FWD, a);
    printf("easting: %.3f, northing: %.3f\n", b.enu.e, b.enu.n);

    b = proj_trans(P, PJ_INV, b);
    printf("longitude: %g, latitude: %g\n", b.lp.lam, b.lp.phi);

    /* Clean up */
    proj_destroy(P);
    proj_context_destroy(C); /* may be omitted in the single threaded case */
    return 0;
}
```

The following example illustrates how to convert between a CRS and geodetic coordinates for that CRS.

```
#include <math.h>
#include <proj.h>
#include <stdio.h>

int main(void) {

    /* Create the context. */
    /* You may set C=PJ_DEFAULT_CTX if you are sure you will     */
    /* use PJ objects from only one thread                       */
```

(continues on next page)

(continued from previous page)

```
    PJ_CONTEXT *C = proj_context_create();

    /* Create a projection. */
    PJ *P = proj_create(C, "+proj=utm +zone=32 +datum=WGS84 +type=crs");

    if (0 == P) {
        fprintf(stderr, "Failed to create transformation object.\n");
        return 1;
    }

    /* Get the geodetic CRS for that projection. */
    PJ *G = proj_crs_get_geodetic_crs(C, P);

    /* Create the transform from geodetic to projected coordinates.*/
    PJ_AREA *A = NULL;
    const char *const *options = NULL;
    PJ *G2P = proj_create_crs_to_crs_from_pj(C, G, P, A, options);

    /* Longitude and latitude of Copenhagen, in degrees. */
    double lon = 12.0, lat = 55.0;

    /* Prepare the input */
    PJ_COORD c_in;
    c_in.lpzt.z = 0.0;
    c_in.lpzt.t = HUGE_VAL; // important only for time-dependent projections
    c_in.lp.lam = lon;
    c_in.lp.phi = lat;
    printf("Input longitude: %g, latitude: %g (degrees)\n", c_in.lp.lam,
           c_in.lp.phi);

    /* Compute easting and northing */
    PJ_COORD c_out = proj_trans(G2P, PJ_FWD, c_in);
    printf("Output easting: %g, northing: %g (meters)\n", c_out.enu.e,
           c_out.enu.n);

    /* Apply the inverse transform */
    PJ_COORD c_inv = proj_trans(G2P, PJ_INV, c_out);
    printf("Inverse applied. Longitude: %g, latitude: %g (degrees)\n",
           c_inv.lp.lam, c_inv.lp.phi);

    /* Clean up */
    proj_destroy(P);
    proj_destroy(G);
    proj_destroy(G2P);
    proj_context_destroy(C); /* may be omitted in the single threaded case */
    return 0;
}
```

## 10.3 Quick start for C++ API usage

This is a short introduction to the PROJ *C++ API*. In the following section we create a simple program that illustrate how to transform points between two different coordinate systems.

Note: the essential osgeo::proj::operation::CoordinateOperation::coordinateTransformer() method was only added in PROJ 9.3. For earlier versions, coordinate operations must be exported as a PROJ string with the osgeo::proj::operation::CoordinateOperation::exportToPROJ() method, and passed to *proj_create()* to instance a PJ* object to use with *proj_trans()* (cf the *Quick start for C API usage*). The use of the C API *proj_create_crs_to_crs()* might still be easier to let PROJ automatically select the "best" coordinate operation when several ones are possible, as the corresponding automation is not currently available in the C++ API.

Before the PROJ API can be used it is necessary to include the various header files:

```
#include <cassert>
#include <cmath>   // for HUGE_VAL
#include <iomanip> // for std::setprecision()

#include <iostream>

#include "proj/coordinateoperation.hpp"
#include "proj/crs.hpp"
#include "proj/io.hpp"
#include "proj/util.hpp" // for nn_dynamic_pointer_cast
```

For convenience, we also declare using a few namespaces:

```
using namespace NS_PROJ::crs;
using namespace NS_PROJ::io;
using namespace NS_PROJ::operation;
using namespace NS_PROJ::util;
```

We start by creating a database context (*osgeo::proj::io::DatabaseContext*) with the default settings to find the PROJ database.

```
auto dbContext = DatabaseContext::create();
```

We then instantiate a generic authority factory (*osgeo::proj::io::AuthorityFactory*), that is not tied to a particular authority, to be able to get transformations registered by different authorities. This can only be used for a *osgeo::proj::operation::CoordinateOperationContext*, and not to instantiate objects of the database which are all tied to a non-generic authority.

```
    auto authFactory = AuthorityFactory::create(dbContext, std::string());
```

We create a coordinate operation context, that can be customized to amend the way coordinate operations are computed. Here we ask for default settings, as we have a coordinate operation that just involves a "simple" map projection in the same datum.

```
auto coord_op_ctxt =
    CoordinateOperationContext::create(authFactory, nullptr, 0.0);
```

We instantiate a authority factory for EPSG related objects.

```
auto authFactoryEPSG = AuthorityFactory::create(dbContext, "EPSG");
```

We instantiate the source CRS from its code: 4326, for WGS 84 latitude/longitude.

```
auto sourceCRS = authFactoryEPSG->createCoordinateReferenceSystem("4326");
```

We instantiate the source CRS from its PROJ.4 string (it would be possible to instantiate it from its 32631 code, similarly to above), and cast the generic *osgeo::proj::util::BaseObject* to the *osgeo::proj::crs::CRS* class required later.

```
auto targetCRS =
    NN_CHECK_THROW(nn_dynamic_pointer_cast<CRS>(createFromUserInput(
        "+proj=utm +zone=31 +datum=WGS84 +type=crs", dbContext)));
```

**Warning**

The use of PROJ strings to describe a CRS is not recommended. One of the main weaknesses of PROJ strings is their inability to describe a geodetic datum, other than the few ones hardcoded in the `+datum` parameter.

We ask for the list of operations available to transform from the source to the target CRS with the *osgeo::proj::operation::CoordinateOperationFactory::createOperations()* method.

```
auto list = CoordinateOperationFactory::create()->createOperations(
    sourceCRS, targetCRS, coord_op_ctxt);
```

We check that we got a non-empty list of operations. The list is sorted from the most relevant to the less relevant one. Cf *Filtering and sorting of coordinate operations* for more details on the sorting of those operations. For a transformation between a projected CRS and its base CRS, like we do here, there will be only one operation.

```
assert(!list.empty());
```

We create an execution context (must only be used by one thread at a time) with the *proj_context_create()* function.

```
PJ_CONTEXT *ctx = proj_context_create();
```

We create a coordinate transformer (*osgeo::proj::operation::CoordinateTransformer*) from the first operation of the list:

```
auto transformer = list[0]->coordinateTransformer(ctx);
```

We can now transform a point with the *osgeo::proj::operation::CoordinateTransformer::transform()* method. Note that the the expected input values should be passed in the order and the unit of the successive axis of the input CRS. Similarly the values returned in the v[] array of the output PJ_COORD are in the order and the unit of the successive axis of the output CRS. For coordinate operations involving a time-dependent operation, coord.v[3] is the decimal year of the coordinate epoch of the input (or HUGE_VAL to indicate none).

```
PJ_COORD c = {{
    49.0,    // latitude in degree
    2.0,     // longitude in degree
    0.0,     // z ordinate. unused
    HUGE_VAL // time ordinate. unused
}};
c = transformer->transform(c);
```

and output the result:

```
std::cout << std::fixed << std::setprecision(3);
std::cout << "Easting: " << c.v[0] << std::endl;  // should be 426857.988
std::cout << "Northing: " << c.v[1] << std::endl; // should be 5427937.523
```

We need to clean up the PJ_CONTEXT handle before exiting with the *proj_context_destroy()* function.

```
proj_context_destroy(ctx);
```

A complete compilable version of the example code can be seen below:

```
#include <cassert>
#include <cmath>   // for HUGE_VAL
#include <iomanip> // for std::setprecision()

#include <iostream>

#include "proj/coordinateoperation.hpp"
#include "proj/crs.hpp"
#include "proj/io.hpp"
#include "proj/util.hpp" // for nn_dynamic_pointer_cast

using namespace NS_PROJ::crs;
using namespace NS_PROJ::io;
using namespace NS_PROJ::operation;
using namespace NS_PROJ::util;

int main(void) {
    auto dbContext = DatabaseContext::create();

    // Instantiate a generic authority factory, that is not tied to a particular
    // authority, to be able to get transformations registered by different
    // authorities. This can only be used for CoordinateOperationContext.
    auto authFactory = AuthorityFactory::create(dbContext, std::string());

    // Create a coordinate operation context, that can be customized to amend
    // the way coordinate operations are computed. Here we ask for default
    // settings.
    auto coord_op_ctxt =
        CoordinateOperationContext::create(authFactory, nullptr, 0.0);

    // Instantiate a authority factory for EPSG related objects.
    auto authFactoryEPSG = AuthorityFactory::create(dbContext, "EPSG");

    // Instantiate source CRS from EPSG code
    auto sourceCRS = authFactoryEPSG->createCoordinateReferenceSystem("4326");

    // Instantiate target CRS from PROJ.4 string (commented out, the equivalent
    // from the EPSG code)
    // auto targetCRS =
    // authFactoryEPSG->createCoordinateReferenceSystem("32631");
    auto targetCRS =
        NN_CHECK_THROW(nn_dynamic_pointer_cast<CRS>(createFromUserInput(
            "+proj=utm +zone=31 +datum=WGS84 +type=crs", dbContext)));
```

(continues on next page)

(continued from previous page)

```
    // List operations available to transform from EPSG:4326
    // (WGS 84 latitude/longitude) to EPSG:32631 (WGS 84 / UTM zone 31N).
    auto list = CoordinateOperationFactory::create()->createOperations(
        sourceCRS, targetCRS, coord_op_ctxt);

    // Check that we got a non-empty list of operations
    // The list is sorted from the most relevant to the less relevant one.
    // Cf
    // https://proj.org/operations/operations_computation.html#filtering-and-sorting-of-
↪coordinate-operations
    // for more details on the sorting of those operations.
    // For a transformation between a projected CRS and its base CRS, like
    // we do here, there will be only one operation.
    assert(!list.empty());

    // Create an execution context (must only be used by one thread at a time)
    PJ_CONTEXT *ctx = proj_context_create();

    // Create a coordinate transformer from the first operation of the list
    auto transformer = list[0]->coordinateTransformer(ctx);

    // Perform the coordinate transformation.
    PJ_COORD c = {{
        49.0,    // latitude in degree
        2.0,     // longitude in degree
        0.0,     // z ordinate. unused
        HUGE_VAL // time ordinate. unused
    }};
    c = transformer->transform(c);

    // Display result
    std::cout << std::fixed << std::setprecision(3);
    std::cout << "Easting: " << c.v[0] << std::endl;  // should be 426857.988
    std::cout << "Northing: " << c.v[1] << std::endl; // should be 5427937.523

    // Destroy execution context
    proj_context_destroy(ctx);

    return 0;
}
```

## 10.4 Error handling

PROJ maintains an internal error state, which is local to a *PJ_CONTEXT* thread context.

See *Quick start for C API usage* for more information about how to create and use a thread context object.

If you receive an abnormal return from a PROJ API function (e.g. a NULL pointer) you may wish to discover more information about the error.

In this case you can make a call to *proj_context_errno()*, passing in your thread context. This will return an integer error code.

If the error code is zero, the last PROJ operation was deemed successful and no error has been detected.

If the error code is non-zero, an error has been detected. You can pass your thread context together with this error code to *proj_context_errno_string()* to retrieve a string describing the error condition.

A basic example showing how a C program might catch and report errors follows:

Listing 1: errorhandling.c

```c
#include <stdio.h>
#include <proj.h>

int main (void) {
    PJ_CONTEXT *c;
    PJ *p;
    int errno;
    const char *errstr;

    c = proj_context_create();
    p = proj_create_crs_to_crs(c, "EPSG:4326", "EPSG:3857", NULL);

    if (p == 0) {
        /* Something is wrong, let's try to get details ... */
        errno = proj_context_errno(c);
        if (errno == 0) {
            /* This should be impossible. */
            fprintf(stderr, "Failed to create transformation, reason unknown.\n");
        } else {
            errstr = proj_context_errno_string(c, errno);
            fprintf(stderr, "Failed to create transformation: %s.\n", errstr);
        }
        proj_context_destroy(c);
        return 1;
    }

    /* transformation object is valid, do work ... */

    proj_destroy(p);
    proj_context_destroy(c);

    return 0;
}
```

## 10.5 Reference

### 10.5.1 Macros

**PROJ_VERSION_MAJOR**

Major version number, e.g. 8 for PROJ 8.0.1

**PROJ_VERSION_MINOR**

Minor version number, e.g. 0 for PROJ 8.0.1

**PROJ_VERSION_PATCH**

Patch version number, e.g. 1 for PROJ 8.0.1

**PROJ_COMPUTE_VERSION**(maj, min, patch)

Added in version 8.0.1.

Compute the version number from the major, minor and patch numbers.

**PROJ_VERSION_NUMBER**

Added in version 8.0.1.

Total version number, equal to `PROJ_COMPUTE_VERSION(PROJ_VERSION_MAJOR, PROJ_VERSION_MINOR, PROJ_VERSION_PATCH)`

**PROJ_AT_LEAST_VERSION**(maj, min, patch)

Added in version 8.0.1.

Macro that returns true if the current PROJ version is at least the version specified by (maj,min,patch)

Equivalent to `PROJ_VERSION_NUMBER >= PROJ_COMPUTE_VERSION(maj,min,patch)`

## 10.5.2 Data types

This section describes the numerous data types in use in PROJ. As a rule of thumb PROJ data types are prefixed with `PJ_`, or in one particular case, is simply called *PJ*. A few notable exceptions can be traced back to the very early days of PROJ when the `PJ_` prefix was not consistently used.

### 10.5.2.1 Transformation objects

type **PJ**

Object containing everything related to a given projection or transformation. As a user of the PROJ library you are only exposed to pointers to this object and the contents is hidden behind the public API. *PJ* objects are created with *proj_create()* and destroyed with *proj_destroy()*.

type **PJ_DIRECTION**

Enumeration that is used to convey in which direction a given transformation should be performed. Used in transformation function call as described in the section on *transformation functions*.

Forward transformations are defined with the :c:

```
typedef enum proj_direction {
    PJ_FWD   =  1,   /* Forward    */
    PJ_IDENT =  0,   /* Do nothing */
    PJ_INV   = -1    /* Inverse    */
} PJ_DIRECTION;
```

enumerator **PJ_FWD**

Perform transformation in the forward direction.

enumerator **PJ_IDENT**

Identity. Do nothing.

enumerator **PJ_INV**

Perform transformation in the inverse direction.

type **PJ_CONTEXT**

Context objects enable safe multi-threaded usage of PROJ. Each *PJ* object is connected to a context (if not specified, the default context is used). All operations within a context should be performed in the same thread. *PJ_CONTEXT* objects are created with *proj_context_create()* and destroyed with *proj_context_destroy()*.

type **PJ_AREA**

Added in version 6.0.0.

Opaque object describing an area in which a transformation is performed.

It is used with *proj_create_crs_to_crs()* to select the best transformation between the two input coordinate reference systems.

#### 10.5.2.2 2 dimensional coordinates

Various 2-dimensional coordinate data types.

type **PJ_LP**

Geodetic coordinate, latitude and longitude. Usually in radians.

```
typedef struct { double lam, phi; } PJ_LP;
```

double PJ_LP.**lam**

Longitude. Lambda.

double PJ_LP.**phi**

Latitude. Phi.

type **PJ_XY**

2-dimensional cartesian coordinate.

```
typedef struct { double x, y; } PJ_XY;
```

double PJ_XY.**x**

Easting.

double PJ_XY.**y**

Northing.

type **PJ_UV**

2-dimensional generic coordinate. Usually used when contents can be either a *PJ_XY* or *PJ_LP*.

```
typedef struct {double u, v; } PJ_UV;
```

double PJ_UV.**u**

Longitude or easting, depending on use.

double PJ_UV.**v**

Latitude or northing, depending on use.

#### 10.5.2.3 3 dimensional coordinates

The following data types are the 3-dimensional equivalents to the data types above.

type **PJ_LPZ**

3-dimensional version of *PJ_LP*. Holds longitude, latitude and a vertical component.

```
typedef struct { double lam, phi, z; } PJ_LPZ;
```

double PJ_LPZ.**lam**

Longitude. Lambda.

double PJ_LPZ.**phi**

Latitude. Phi.

double PJ_LPZ.**z**

Vertical component.

type **PJ_XYZ**

Cartesian coordinate in 3 dimensions. Extension of *PJ_XY*.

```
typedef struct { double x, y, z; } PJ_XYZ;
```

double PJ_XYZ.**x**

Easting or the X component of a 3D cartesian system.

double PJ_XYZ.**y**

Northing or the Y component of a 3D cartesian system.

double PJ_XYZ.**z**

Vertical component or the Z component of a 3D cartesian system.

type **PJ_UVW**

3-dimensional extension of *PJ_UV*.

```
typedef struct {double u, v, w; } PJ_UVW;
```

double PJ_UVW.**u**

Longitude or easting, depending on use.

double PJ_UVW.**v**

Latitude or northing, depending on use.

double PJ_UVW.**w**

Vertical component.

#### 10.5.2.4 Spatiotemporal coordinate types

The following data types are extensions of the triplets above into the time domain.

type **PJ_LPZT**

Spatiotemporal version of *PJ_LPZ*.

```
typedef struct {
    double lam;
    double phi;
    double z;
    double t;
} PJ_LPZT;
```

double PJ_LPZT.**lam**

Longitude.

double PJ_LPZT.**phi**

Latitude

double PJ_LPZT.**z**

Vertical component.

double PJ_LPZT.**t**

Time component.

type **PJ_XYZT**

Generic spatiotemporal coordinate. Useful for e.g. cartesian coordinates with an attached time-stamp.

```
typedef struct {
    double x;
    double y;
    double z;
    double t;
} PJ_XYZT;
```

double PJ_XYZT.**x**

Easting or the X component of a 3D cartesian system.

double PJ_XYZT.**y**

Northing or the Y component of a 3D cartesian system.

double PJ_XYZT.**z**

Vertical or the Z component of a 3D cartesian system.

double PJ_XYZT.**t**

Time component.

type **PJ_UVWT**

Spatiotemporal version of *PJ_UVW*.

```
typedef struct { double u, v, w, t; } PJ_UVWT;
```

double PJ_UVWT.**e**

First horizontal component.

double PJ_UVWT.**n**

Second horizontal component.

double PJ_UVWT.**w**

Vertical component.

double PJ_UVWT.**t**

Temporal component.

#### 10.5.2.5 Ancillary types for geodetic computations

type **PJ_OPK**

Rotations, for instance three euler angles.

```
typedef struct { double o, p, k; } PJ_OPK;
```

double PJ_OPK.**o**

First rotation angle, omega.

double PJ_OPK.**p**

Second rotation angle, phi.

double PJ_OPK.**k**

Third rotation angle, kappa.

type **PJ_ENU**

East, north and up components.

```
typedef struct { double e, n, u; }          PJ_ENU;
```

double PJ_ENU.**e**

East component.

double PJ_ENU.**n**

North component.

double PJ_ENU.**u**

Up component.

type **PJ_GEOD**

Geodesic length, forward and reverse azimuths.

```
typedef struct { double s, a1, a2; }        PJ_GEOD;
```

double PJ_GEOD.**s**

Geodesic length.

double PJ_GEOD.**a1**

Forward azimuth.

double PJ_GEOD.**a2**

Reverse azimuth.

#### 10.5.2.6 Complex coordinate types

type **PJ_COORD**

General purpose coordinate union type, applicable in two, three and four dimensions. This is the default coordinate datatype used in PROJ.

```
typedef union {
    double v[4];
    PJ_XYZT xyzt;
    PJ_UVWT uvwt;
    PJ_LPZT lpzt;
    PJ_GEOD geod;
    PJ_OPK opk;
    PJ_ENU enu;
    PJ_XYZ  xyz;
    PJ_UVW  uvw;
    PJ_LPZ  lpz;
    PJ_XY   xy;
    PJ_UV   uv;
    PJ_LP   lp;
} PJ_COORD ;
```

double **v**[4]

Generic four-dimensional vector.

*PJ_XYZT* PJ_COORD.**xyzt**

Spatiotemporal cartesian coordinate.

*PJ_UVWT* PJ_COORD.**uvwt**

Spatiotemporal generic coordinate.

*PJ_LPZT* PJ_COORD.**lpzt**

Longitude, latitude, vertical and time components.

*PJ_GEOD* PJ_COORD.**geod**

Geodesic length, forward and reverse azimuths.

*PJ_OPK* PJ_COORD.**opk**

Rotations, for instance three euler angles.

*PJ_ENU* PJ_COORD.**enu**

East, north and up components.

*PJ_XYZ* PJ_COORD.**xyz**

3-dimensional cartesian coordinate.

*PJ_UVW* PJ_COORD.**uvw**

3-dimensional generic coordinate.

*PJ_LPZ* PJ_COORD.**lpz**

Longitude, latitude and vertical component.

*PJ_XY* PJ_COORD.**xy**

2-dimensional cartesian coordinate.

*PJ_UV* PJ_COORD.**uv**

2-dimensional generic coordinate.

*PJ_LP* PJ_COORD.**lp**

Longitude and latitude.

#### 10.5.2.7 Projection derivatives

type **PJ_FACTORS**

Various cartographic properties, such as scale factors, angular distortion and meridian convergence. Calculated with *proj_factors()*.

```
typedef struct {
    double meridional_scale;
    double parallel_scale;
    double areal_scale;

    double angular_distortion;
    double meridian_parallel_angle;
    double meridian_convergence;

    double tissot_semimajor;
    double tissot_semiminor;

    double dx_dlam;
    double dx_dphi;
```

(continues on next page)

(continued from previous page)

```
    double dy_dlam;
    double dy_dphi;
} PJ_FACTORS;
```

double PJ_FACTORS.**meridional_scale**

Meridional scale at coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**parallel_scale**

Parallel scale at coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**areal_scale**

Areal scale factor at coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**angular_distortion**

Angular distortion at coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**meridian_parallel_angle**

Meridian/parallel angle, $\theta'$, at coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**meridian_convergence**

Meridian convergence at coordinate $(\lambda, \phi)$. Sometimes also described as *grid declination*.

double PJ_FACTORS.**tissot_semimajor**

Maximum scale factor.

double PJ_FACTORS.**tissot_semiminor**

Minimum scale factor.

double PJ_FACTORS.**dx_dlam**

Partial derivative $\frac{\partial x}{\partial \lambda}$ of coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**dy_dlam**

Partial derivative $\frac{\partial y}{\partial \lambda}$ of coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**dx_dphi**

Partial derivative $\frac{\partial x}{\partial \phi}$ of coordinate $(\lambda, \phi)$.

double PJ_FACTORS.**dy_dphi**

Partial derivative $\frac{\partial y}{\partial \phi}$ of coordinate $(\lambda, \phi)$.

#### 10.5.2.8 List structures

type **PJ_OPERATIONS**

Description a PROJ operation

```
struct PJ_OPERATIONS {
    const char  *id;            /* operation keyword */
    PJ *(*proj)(PJ *);          /* operation  entry point */
    char    * const *descr;     /* description text */
};
```

const char ***id**

Operation keyword.

*PJ* *(***op**)(*PJ**)

Operation entry point.

char *const ***descr**

Description of operation.

type **PJ_ELLPS**

Description of ellipsoids defined in PROJ

```
struct PJ_ELLPS {
    const char  *id;
    const char  *major;
    const char  *ell;
    const char  *name;
};
```

const char ***id**

Keyword name of the ellipsoid.

const char ***major**

Semi-major axis of the ellipsoid, or radius in case of a sphere.

const char ***ell**

Elliptical parameter, e.g. `rf=298.257` or `b=6356772.2`.

const char ***name**

Name of the ellipsoid

type **PJ_UNITS**

Distance units defined in PROJ.

```
struct PJ_UNITS {
    const char  *id;         /* units keyword */
    const char  *to_meter;   /* multiply by value to get meters */
    const char  *name;       /* comments */
    double      factor;      /* to_meter factor in actual numbers */
};
```

const char ***id**

Keyword for the unit.

const char ***to_meter**

Text representation of the factor that converts a given unit to meters

const char ***name**

Name of the unit.

double **factor**

Conversion factor that converts the unit to meters.

type **PJ_PRIME_MERIDIANS**

Hard-coded prime meridians defined in PROJ. Note that the structure is no longer updated, and some values may conflict with other sources.

```
struct PJ_PRIME_MERIDIANS {
    const char  *id;
    const char  *defn;
};
```

const char ***id**

Keyword for the prime meridian

const char ***def**

Offset from Greenwich in DMS format.

#### 10.5.2.9 Info structures

type **PJ_INFO**

Struct holding information about the current instance of PROJ. Struct is populated by *proj_info()*.

```
typedef struct {
    int           major;
    int           minor;
    int           patch;
    const char   *release;
    const char   *version;
    const char   *searchpath;
} PJ_INFO;
```

const char *PJ_INFO.**release**

Release info. Version number and release date, e.g. "Rel. 4.9.3, 15 August 2016".

const char *PJ_INFO.**version**

Text representation of the full version number, e.g. "4.9.3".

int PJ_INFO.**major**

Major version number.

int PJ_INFO.**minor**

Minor version number.

int PJ_INFO.**patch**

Patch level of release.

const char PJ_INFO.**searchpath**

Search path for PROJ. List of directories separated by semicolons (Windows) or colons (non-Windows), e.g. `C:\\Users\\doctorwho;C:\\OSGeo4W64\\share\\proj`. Grids and *init files* are looked for in directories in the search path.

type **PJ_PROJ_INFO**

Struct holding information about a *PJ* object. Populated by *proj_pj_info()*. The PJ_PROJ_INFO object provides a view into the internals of a *PJ*, so once the *PJ* is destroyed or otherwise becomes invalid, so does the PJ_PROJ_INFO

```
typedef struct {
    const char  *id;
    const char  *description;
    const char  *definition;
```

(continues on next page)

(continued from previous page)

```
    int          has_inverse;
    double       accuracy;
} PJ_PROJ_INFO;
```

const char *PJ_PROJ_INFO.**id**

Short ID of the operation the *PJ* object is based on, that is, what comes after the +proj= in a proj-string, e.g. "*merc*".

const char *PJ_PROJ_INFO.**description**

Long describes of the operation the *PJ* object is based on, e.g. "*Mercator Cyl, Sph&Ell lat_ts=*".

const char *PJ_PROJ_INFO.**definition**

The proj-string that was used to create the *PJ* object with, e.g. "*+proj=merc +lat_0=24 +lon_0=53 +ellps=WGS84*".

int PJ_PROJ_INFO.**has_inverse**

1 if an inverse mapping of the defined operation exists, otherwise 0.

double PJ_PROJ_INFO.**accuracy**

Expected accuracy of the transformation. -1 if unknown.

type **PJ_GRID_INFO**

Struct holding information about a specific grid in the search path of PROJ. Populated with the function *proj_grid_info()*.

```
typedef struct {
    char        gridname[32];
    char        filename[260];
    char        format[8];
    LP          lowerleft;
    LP          upperright;
    int         n_lon, n_lat;
    double      cs_lon, cs_lat;
} PJ_GRID_INFO;
```

char PJ_GRID_INFO.**gridname**[32]

Name of grid, e.g. "*BETA2007.gsb*".

char PJ_GRID_INFO.**filename**[260]

Full path of grid file, e.g. *"C:\OSGeo4W64\share\proj\BETA2007.gsb"*

char PJ_GRID_INFO.**format**[8]

File format of grid file, e.g. "*ntv2*"

LP PJ_GRID_INFO.**lowerleft**

Geodetic coordinate of lower left corner of grid.

LP PJ_GRID_INFO.**upperright**

Geodetic coordinate of upper right corner of grid.

int PJ_GRID_INFO.**n_lon**

Number of grid cells in the longitudinal direction.

int PJ_GRID_INFO.**n_lat**

Number of grid cells in the latitudinal direction.

double PJ_GRID_INFO.**cs_lon**

Cell size in the longitudinal direction. In radians.

double PJ_GRID_INFO.**cs_lat**

Cell size in the latitudinal direction. In radians.

type **PJ_INIT_INFO**

Struct holding information about a specific init file in the search path of PROJ. Populated with the function *proj_init_info()*.

```
typedef struct {
    char        name[32];
    char        filename[260];
    char        version[32];
    char        origin[32];
    char        lastupdate[16];
} PJ_INIT_INFO;
```

char PJ_INIT_INFO.**name**[32]

Name of init file, e.g. "*epsg*".

char PJ_INIT_INFO.**filename**[260]

Full path of init file, e.g. "*C:\OSGeo4W64\share\proj\epsg*"

char PJ_INIT_INFO.**version**[32]

Version number of init file, e.g. "*9.0.0*"

char PJ_INIT_INFO.**origin**[32]

Originating entity of the init file, e.g. "*EPSG*"

char PJ_INIT_INFO.**lastupdate**

Date of last update of the init file.

#### 10.5.2.10 Error codes

Added in version 8.0.0.

Three classes of errors are defined below. The belonging of a given error code to a class can bit tested with a binary and test. The error class itself can be used as an error value in some rare cases where the error does not fit into a more precise error value.

Those error codes are still quite generic for a number of them. Details on the actual errors will be typically logged with the PJ_LOG_ERROR level.

##### Errors in class PROJ_ERR_INVALID_OP

**PROJ_ERR_INVALID_OP**

Class of error codes typically related to coordinate operation initialization, typically when creating a PJ* object from a PROJ string.

> **Note**
>
> some of them can also be emitted during coordinate transformation, like PROJ_ERR_INVALID_OP_FILE_NOT_FOUND_OR_INVALID in case the resource loading is deferred until it is really needed.

**PROJ_ERR_INVALID_OP_WRONG_SYNTAX**

Invalid pipeline structure, missing +proj argument, etc.

**PROJ_ERR_INVALID_OP_MISSING_ARG**

Missing required operation parameter

**PROJ_ERR_INVALID_OP_ILLEGAL_ARG_VALUE**

One of the operation parameter has an illegal value.

**PROJ_ERR_INVALID_OP_MUTUALLY_EXCLUSIVE_ARGS**

Mutually exclusive arguments

**PROJ_ERR_INVALID_OP_FILE_NOT_FOUND_OR_INVALID**

File not found or with invalid content (particular case of PROJ_ERR_INVALID_OP_ILLEGAL_ARG_VALUE)

#### Errors in class PROJ_ERR_COORD_TRANSFM

**PROJ_ERR_COORD_TRANSFM**

Class of error codes related to transformation on a specific coordinate.

**PROJ_ERR_COORD_TRANSFM_INVALID_COORD**

Invalid input coordinate. e.g. a latitude > 90°.

**PROJ_ERR_COORD_TRANSFM_OUTSIDE_PROJECTION_DOMAIN**

Coordinate is outside of the projection domain. e.g. approximate mercator with |longitude - lon_0| > 90°, or iterative convergence method failed.

**PROJ_ERR_COORD_TRANSFM_NO_OPERATION**

No operation found, e.g. if no match the required accuracy, or if ballpark transformations were asked to not be used and they would be only such candidate.

**PROJ_ERR_COORD_TRANSFM_OUTSIDE_GRID**

Point to transform falls outside grid/subgrid/TIN.

**PROJ_ERR_COORD_TRANSFM_GRID_AT_NODATA**

Point to transform falls in a grid cell that evaluates to nodata.

#### Errors in class PROJ_ERR_OTHER

**PROJ_ERR_OTHER**

Class of error codes that do not fit into one of the above class.

**PROJ_ERR_OTHER_API_MISUSE**

Error related to a misuse of PROJ API.

**PROJ_ERR_OTHER_NO_INVERSE_OP**

No inverse method available

**PROJ_ERR_OTHER_NETWORK_ERROR**

Failure when accessing a network resource.

### 10.5.2.11 Logging

type **PJ_LOG_LEVEL**

Enum of logging levels in PROJ. Used to set the logging level in PROJ. Usually using *proj_log_level()*.

enumerator **PJ_LOG_NONE**

Don't log anything.

enumerator **PJ_LOG_ERROR**

Log only errors.

enumerator **PJ_LOG_DEBUG**

Log errors and additional debug information.

enumerator **PJ_LOG_TRACE**

Highest logging level. Log everything including very detailed debug information.

enumerator **PJ_LOG_TELL**

Special logging level that when used in *proj_log_level()* will return the current logging level set in PROJ.

Added in version 5.1.0.

type **PJ_LOG_FUNC**

Function prototype for the logging function used by PROJ. Defined as

```
typedef void (*PJ_LOG_FUNCTION)(void *, int, const char *);
```

where the first argument (void pointer) references a data structure used by the calling application, the second argument (int type) is used to set the logging level and the third argument (const char pointer) is the string that will be logged by the function.

Added in version 5.1.0.

#### 10.5.2.12 Setting custom I/O functions

Added in version 7.0.0.

struct **PROJ_FILE_API**

File API callbacks

##### Public Members

int **version**

Version of this structure. Should be set to 1 currently.

*PROJ_FILE_HANDLE* *(***open_cbk**)(*PJ_CONTEXT* *ctx, const char *filename, *PROJ_OPEN_ACCESS* access, void *user_data)

Open file. Return NULL if error

size_t (***read_cbk**)(*PJ_CONTEXT* *ctx, *PROJ_FILE_HANDLE**, void *buffer, size_t sizeBytes, void *user_data)

Read sizeBytes into buffer from current position and return number of bytes read

size_t (***write_cbk**)(*PJ_CONTEXT* *ctx, *PROJ_FILE_HANDLE**, const void *buffer, size_t sizeBytes, void *user_data)

Write sizeBytes into buffer from current position and return number of bytes written

int (***seek_cbk**)(*PJ_CONTEXT* *ctx, *PROJ_FILE_HANDLE**, long long offset, int whence, void *user_data)

Seek to offset using whence=SEEK_SET/SEEK_CUR/SEEK_END. Return TRUE in case of success

unsigned long long (***tell_cbk**)(*PJ_CONTEXT* *ctx, *PROJ_FILE_HANDLE**, void *user_data)

Return current file position

void (***close_cbk**)(*PJ_CONTEXT* *ctx, *PROJ_FILE_HANDLE**, void *user_data)

Close file

int (***exists_cbk**)(*PJ_CONTEXT* *ctx, const char *filename, void *user_data)

Return TRUE if a file exists

int (***mkdir_cbk**)(*PJ_CONTEXT* *ctx, const char *filename, void *user_data)

Return TRUE if directory exists or could be created

int (***unlink_cbk**)(*PJ_CONTEXT* *ctx, const char *filename, void *user_data)

Return TRUE if file could be removed

int (***rename_cbk**)(*PJ_CONTEXT* *ctx, const char *oldPath, const char *newPath, void *user_data)

Return TRUE if file could be renamed

typedef struct *PROJ_FILE_HANDLE* **PROJ_FILE_HANDLE**

Opaque structure for PROJ for a file handle. Implementations might cast it to their structure/class of choice.

enum **PROJ_OPEN_ACCESS**

Open access / mode

*Values:*

enumerator **PROJ_OPEN_ACCESS_READ_ONLY**

Read-only access. Equivalent to "rb"

enumerator **PROJ_OPEN_ACCESS_READ_UPDATE**

Read-update access. File should be created if not existing. Equivalent to "r+b"

enumerator **PROJ_OPEN_ACCESS_CREATE**

Create access. File should be truncated to 0-byte if already existing. Equivalent to "w+b"

#### 10.5.2.13 Network related functionality

Added in version 7.0.0.

typedef struct *PROJ_NETWORK_HANDLE* **PROJ_NETWORK_HANDLE**

Opaque structure for PROJ for a network handle. Implementations might cast it to their structure/class of choice.

typedef *PROJ_NETWORK_HANDLE* *(***proj_network_open_cbk_type**)(*PJ_CONTEXT* *ctx, const char *url, unsigned long long offset, size_t size_to_read, void *buffer, size_t *out_size_read, size_t error_string_max_size, char *out_error_string, void *user_data)

Network access: open callback

Should try to read the size_to_read first bytes at the specified offset of the file given by URL url, and write them to buffer. *out_size_read should be updated with the actual amount of bytes read (== size_to_read if the file is larger than size_to_read). During this read, the implementation should make sure to store the HTTP headers from the server response to be able to respond to proj_network_get_header_value_cbk_type callback.

error_string_max_size should be the maximum size that can be written into the out_error_string buffer (including terminating nul character).

**Return**
a non-NULL opaque handle in case of success.

typedef void (***proj_network_close_cbk_type**)(*PJ_CONTEXT* *ctx, *PROJ_NETWORK_HANDLE* *handle, void *user_data)

Network access: close callback

typedef const char *(***proj_network_get_header_value_cbk_type**)(*PJ_CONTEXT* *ctx, *PROJ_NETWORK_HANDLE* *handle, const char *header_name, void *user_data)

Network access: get HTTP headers

typedef size_t (***proj_network_read_range_type**)(*PJ_CONTEXT* *ctx, *PROJ_NETWORK_HANDLE* *handle, unsigned long long offset, size_t size_to_read, void *buffer, size_t error_string_max_size, char *out_error_string, void *user_data)

Network access: read range

Read size_to_read bytes from handle, starting at offset, into buffer. During this read, the implementation should make sure to store the HTTP headers from the server response to be able to respond to proj_network_get_header_value_cbk_type callback.

error_string_max_size should be the maximum size that can be written into the out_error_string buffer (including terminating nul character).

**Return**
the number of bytes actually read (0 in case of error)

#### 10.5.2.14 C API for ISO-19111 functionality

enum **PJ_GUESSED_WKT_DIALECT**

Guessed WKT "dialect".

*Values:*

enumerator **PJ_GUESSED_WKT2_2019**

*WKT2:2019*

enumerator **PJ_GUESSED_WKT2_2018**

Deprecated alias for PJ_GUESSED_WKT2_2019

enumerator **PJ_GUESSED_WKT2_2015**

*WKT2:2015*

enumerator **PJ_GUESSED_WKT1_GDAL**

*WKT1 specification*

enumerator **PJ_GUESSED_WKT1_ESRI**

ESRI variant of WKT1

enumerator **PJ_GUESSED_NOT_WKT**

Not WKT / unrecognized

enum **PJ_CATEGORY**

Object category.

*Values:*

enumerator **PJ_CATEGORY_ELLIPSOID**

enumerator **PJ_CATEGORY_PRIME_MERIDIAN**

enumerator **PJ_CATEGORY_DATUM**

enumerator **PJ_CATEGORY_CRS**

enumerator **PJ_CATEGORY_COORDINATE_OPERATION**

enumerator **PJ_CATEGORY_DATUM_ENSEMBLE**

enum **PJ_TYPE**

Object type.

*Values:*

enumerator **PJ_TYPE_UNKNOWN**

enumerator **PJ_TYPE_ELLIPSOID**

enumerator **PJ_TYPE_PRIME_MERIDIAN**

enumerator **PJ_TYPE_GEODETIC_REFERENCE_FRAME**

enumerator **PJ_TYPE_DYNAMIC_GEODETIC_REFERENCE_FRAME**

enumerator **PJ_TYPE_VERTICAL_REFERENCE_FRAME**

enumerator **PJ_TYPE_DYNAMIC_VERTICAL_REFERENCE_FRAME**

enumerator **PJ_TYPE_DATUM_ENSEMBLE**

enumerator **PJ_TYPE_CRS**

Abstract type, not returned by *proj_get_type()*

enumerator **PJ_TYPE_GEODETIC_CRS**

enumerator **PJ_TYPE_GEOCENTRIC_CRS**

enumerator **PJ_TYPE_GEOGRAPHIC_CRS**

*proj_get_type()* will never return that type, but PJ_TYPE_GEOGRAPHIC_2D_CRS or PJ_TYPE_GEOGRAPHIC_3D_CRS.

enumerator **PJ_TYPE_GEOGRAPHIC_2D_CRS**

enumerator **PJ_TYPE_GEOGRAPHIC_3D_CRS**

enumerator **PJ_TYPE_VERTICAL_CRS**

enumerator **PJ_TYPE_PROJECTED_CRS**

enumerator **PJ_TYPE_COMPOUND_CRS**

enumerator **PJ_TYPE_TEMPORAL_CRS**

enumerator **PJ_TYPE_ENGINEERING_CRS**

enumerator **PJ_TYPE_BOUND_CRS**

enumerator **PJ_TYPE_OTHER_CRS**

enumerator **PJ_TYPE_CONVERSION**

enumerator **PJ_TYPE_TRANSFORMATION**

enumerator **PJ_TYPE_CONCATENATED_OPERATION**

enumerator **PJ_TYPE_OTHER_COORDINATE_OPERATION**

enumerator **PJ_TYPE_TEMPORAL_DATUM**

enumerator **PJ_TYPE_ENGINEERING_DATUM**

enumerator **PJ_TYPE_PARAMETRIC_DATUM**

enumerator **PJ_TYPE_DERIVED_PROJECTED_CRS**

enumerator **PJ_TYPE_COORDINATE_METADATA**

enum **PJ_COMPARISON_CRITERION**

Comparison criterion.

*Values:*

enumerator **PJ_COMP_STRICT**

All properties are identical.

enumerator **PJ_COMP_EQUIVALENT**

The objects are equivalent for the purpose of coordinate operations. They can differ by the name of their objects, identifiers, other metadata. Parameters may be expressed in different units, provided that the value is (with some tolerance) the same once expressed in a common unit.

enumerator **PJ_COMP_EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS**

Same as EQUIVALENT, relaxed with an exception that the axis order of the base CRS of a DerivedCRS/ProjectedCRS or the axis order of a GeographicCRS is ignored. Only to be used with DerivedCRS/ProjectedCRS/GeographicCRS

enum **PJ_WKT_TYPE**

WKT version.

*Values:*

enumerator **PJ_WKT2_2015**

cf *osgeo::proj::io::WKTFormatter::Convention::WKT2*

enumerator **PJ_WKT2_2015_SIMPLIFIED**

cf *osgeo::proj::io::WKTFormatter::Convention::WKT2_SIMPLIFIED*

enumerator **PJ_WKT2_2019**

cf *osgeo::proj::io::WKTFormatter::Convention::WKT2_2019*

enumerator **PJ_WKT2_2018**

Deprecated alias for PJ_WKT2_2019

enumerator **PJ_WKT2_2019_SIMPLIFIED**

cf *osgeo::proj::io::WKTFormatter::Convention::WKT2_2019_SIMPLIFIED*

enumerator **PJ_WKT2_2018_SIMPLIFIED**

Deprecated alias for PJ_WKT2_2019

enumerator **PJ_WKT1_GDAL**

cf *osgeo::proj::io::WKTFormatter::Convention::WKT1_GDAL*

enumerator **PJ_WKT1_ESRI**

cf *osgeo::proj::io::WKTFormatter::Convention::WKT1_ESRI*

enum **PROJ_CRS_EXTENT_USE**

Specify how source and target CRS extent should be used to restrict candidate operations (only taken into account if no explicit area of interest is specified.

*Values:*

enumerator **PJ_CRS_EXTENT_NONE**

Ignore CRS extent

enumerator **PJ_CRS_EXTENT_BOTH**

Test coordinate operation extent against both CRS extent.

enumerator **PJ_CRS_EXTENT_INTERSECTION**

Test coordinate operation extent against the intersection of both CRS extent.

enumerator **PJ_CRS_EXTENT_SMALLEST**

Test coordinate operation against the smallest of both CRS extent.

enum **PROJ_GRID_AVAILABILITY_USE**

Describe how grid availability is used.

*Values:*

enumerator **PROJ_GRID_AVAILABILITY_USED_FOR_SORTING**

Grid availability is only used for sorting results. Operations where some grids are missing will be sorted last.

enumerator **PROJ_GRID_AVAILABILITY_DISCARD_OPERATION_IF_MISSING_GRID**

Completely discard an operation if a required grid is missing.

enumerator **PROJ_GRID_AVAILABILITY_IGNORED**

Ignore grid availability at all. Results will be presented as if all grids were available.

enumerator **PROJ_GRID_AVAILABILITY_KNOWN_AVAILABLE**

Results will be presented as if grids known to PROJ (that is registered in the grid_alternatives table of its database) were available. Used typically when networking is enabled.

enum **PJ_PROJ_STRING_TYPE**

PROJ string version.

*Values:*

enumerator **PJ_PROJ_5**

cf *osgeo::proj::io::PROJStringFormatter::Convention::PROJ_5*

enumerator **PJ_PROJ_4**

cf *osgeo::proj::io::PROJStringFormatter::Convention::PROJ_4*

enum **PROJ_SPATIAL_CRITERION**

Spatial criterion to restrict candidate operations.

*Values:*

enumerator **PROJ_SPATIAL_CRITERION_STRICT_CONTAINMENT**

The area of validity of transforms should strictly contain the are of interest.

enumerator **PROJ_SPATIAL_CRITERION_PARTIAL_INTERSECTION**

The area of validity of transforms should at least intersect the area of interest.

enum **PROJ_INTERMEDIATE_CRS_USE**

Describe if and how intermediate CRS should be used

*Values:*

enumerator **PROJ_INTERMEDIATE_CRS_USE_ALWAYS**

Always search for intermediate CRS.

enumerator **PROJ_INTERMEDIATE_CRS_USE_IF_NO_DIRECT_TRANSFORMATION**

Only attempt looking for intermediate CRS if there is no direct transformation available.

enumerator **PROJ_INTERMEDIATE_CRS_USE_NEVER**

enum **PJ_COORDINATE_SYSTEM_TYPE**

Type of coordinate system.

*Values:*

enumerator **PJ_CS_TYPE_UNKNOWN**

enumerator **PJ_CS_TYPE_CARTESIAN**

enumerator **PJ_CS_TYPE_ELLIPSOIDAL**

enumerator **PJ_CS_TYPE_VERTICAL**

enumerator **PJ_CS_TYPE_SPHERICAL**

enumerator **PJ_CS_TYPE_ORDINAL**

enumerator **PJ_CS_TYPE_PARAMETRIC**

enumerator **PJ_CS_TYPE_DATETIMETEMPORAL**

enumerator **PJ_CS_TYPE_TEMPORALCOUNT**

enumerator **PJ_CS_TYPE_TEMPORALMEASURE**

typedef char ***PROJ_STRING_LIST**

Type representing a NULL terminated list of NULL-terminate strings.

struct **PROJ_CRS_INFO**

*#include <proj.h>* Structure given overall description of a CRS.

This structure may grow over time, and should not be directly allocated by client code.

### Public Members

char ***auth_name**

Authority name.

char ***code**

Object code.

char ***name**

Object name.

*PJ_TYPE* **type**

Object type.

int **deprecated**

Whether the object is deprecated

int **bbox_valid**

Whereas the west_lon_degree, south_lat_degree, east_lon_degree and north_lat_degree fields are valid.

double **west_lon_degree**

Western-most longitude of the area of use, in degrees.

double **south_lat_degree**

Southern-most latitude of the area of use, in degrees.

double **east_lon_degree**

Eastern-most longitude of the area of use, in degrees.

double **north_lat_degree**

Northern-most latitude of the area of use, in degrees.

char ***area_name**

Name of the area of use.

char ***projection_method_name**

Name of the projection method for a projected CRS. Might be NULL even for projected CRS in some cases.

char ***celestial_body_name**

Name of the celestial body of the CRS (e.g. "Earth").

**Since**
8.1

struct **PROJ_CRS_LIST_PARAMETERS**

*#include <proj.h>* Structure describing optional parameters for proj_get_crs_list();.

This structure may grow over time, and should not be directly allocated by client code.

### Public Members

const *PJ_TYPE* ***types**

Array of allowed object types. Should be NULL if all types are allowed

size_t **typesCount**

Size of types. Should be 0 if all types are allowed

int **crs_area_of_use_contains_bbox**

If TRUE and bbox_valid == TRUE, then only CRS whose area of use entirely contains the specified bounding box will be returned. If FALSE and bbox_valid == TRUE, then only CRS whose area of use intersects the specified bounding box will be returned.

int **bbox_valid**

To set to TRUE so that west_lon_degree, south_lat_degree, east_lon_degree and north_lat_degree fields are taken into account.

double **west_lon_degree**

Western-most longitude of the area of use, in degrees.

double **south_lat_degree**

Southern-most latitude of the area of use, in degrees.

double **east_lon_degree**

Eastern-most longitude of the area of use, in degrees.

double **north_lat_degree**

Northern-most latitude of the area of use, in degrees.

int **allow_deprecated**

Whether deprecated objects are allowed. Default to FALSE.

const char ***celestial_body_name**

Celestial body of the CRS (e.g. "Earth"). The default value, NULL, means no restriction

**Since**
8.1

struct **PROJ_UNIT_INFO**

*#include <proj.h>* Structure given description of a unit.

This structure may grow over time, and should not be directly allocated by client code.

**Since**
7.1

### Public Members

char ***auth_name**

Authority name.

char ***code**

Object code.

char ***name**

Object name. For example "metre", "US survey foot", etc.

char ***category**

Category of the unit: one of "linear", "linear_per_time", "angular", "angular_per_time", "scale", "scale_per_time" or "time"

double **conv_factor**

Conversion factor to apply to transform from that unit to the corresponding SI unit (metre for "linear", radian for "angular", etc.). It might be 0 in some cases to indicate no known conversion factor.

char ***proj_short_name**

PROJ short name, like "m", "ft", "us-ft", etc... Might be NULL

int **deprecated**

Whether the object is deprecated

struct **PROJ_CELESTIAL_BODY_INFO**

*#include <proj.h>* Structure given description of a celestial body.

This structure may grow over time, and should not be directly allocated by client code.

**Since**
8.1

### Public Members

char ***auth_name**
Authority name.

char ***name**
Object name. For example "Earth"

enum **PJ_UNIT_TYPE**
Type of unit of measure.
*Values:*

enumerator **PJ_UT_ANGULAR**
Angular unit of measure

enumerator **PJ_UT_LINEAR**
Linear unit of measure

enumerator **PJ_UT_SCALE**
Scale unit of measure

enumerator **PJ_UT_TIME**
Time unit of measure

enumerator **PJ_UT_PARAMETRIC**
Parametric unit of measure

enum **PJ_CARTESIAN_CS_2D_TYPE**
Type of Cartesian 2D coordinate system.
*Values:*

enumerator **PJ_CART2D_EASTING_NORTHING**
Easting-Norting

enumerator **PJ_CART2D_NORTHING_EASTING**
Northing-Easting

enumerator **PJ_CART2D_NORTH_POLE_EASTING_SOUTH_NORTHING_SOUTH**
North Pole Easting/SOUTH-Norting/SOUTH

enumerator **PJ_CART2D_SOUTH_POLE_EASTING_NORTH_NORTHING_NORTH**
South Pole Easting/NORTH-Norting/NORTH

enumerator **PJ_CART2D_WESTING_SOUTHING**

Westing-southing

enum **PJ_ELLIPSOIDAL_CS_2D_TYPE**

Type of Ellipsoidal 2D coordinate system.

*Values:*

enumerator **PJ_ELLPS2D_LONGITUDE_LATITUDE**

Longitude-Latitude

enumerator **PJ_ELLPS2D_LATITUDE_LONGITUDE**

Latitude-Longitude

enum **PJ_ELLIPSOIDAL_CS_3D_TYPE**

Type of Ellipsoidal 3D coordinate system.

*Values:*

enumerator **PJ_ELLPS3D_LONGITUDE_LATITUDE_HEIGHT**

Longitude-Latitude-Height(up)

enumerator **PJ_ELLPS3D_LATITUDE_LONGITUDE_HEIGHT**

Latitude-Longitude-Height(up)

struct **PJ_AXIS_DESCRIPTION**

*#include <proj.h>* Axis description.

struct **PJ_PARAM_DESCRIPTION**

*#include <proj.h>* Description of a parameter value for a Conversion.

## 10.5.3 Functions

### 10.5.3.1 Threading contexts

*PJ_CONTEXT* ***proj_context_create**(void)

Create a new threading-context.

**Returns**
a new context

*PJ_CONTEXT* ***proj_context_clone**(*PJ_CONTEXT* *ctx)

Added in version 7.2.

Create a new threading-context based on an existing context.

**Returns**
a new context

void **proj_context_destroy**(*PJ_CONTEXT* *ctx)

Deallocate a threading-context.

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- Threading context.

#### 10.5.3.2 Transformation setup

The objects returned by the functions defined in this section have minimal interaction with the functions of the *C API for ISO-19111 functionality*, and vice versa. See its introduction paragraph for more details.

*PJ* ***proj_create**(*PJ_CONTEXT* *ctx, const char *definition)

Create a transformation object, or a CRS object, from:

- a proj-string,
- a WKT string,
- an object code (like "EPSG:4326", "urn:ogc:def:crs:EPSG::4326", "urn:ogc:def:coordinateOperation:EPSG::1671"),
- an Object name. e.g "WGS 84", "WGS 84 / UTM zone 31N". In that case as uniqueness is not guaranteed, heuristics are applied to determine the appropriate best match.
- a OGC URN combining references for compound coordinate reference systems (e.g "urn:ogc:def:crs,crs:EPSG::2393,crs:EPSG::5717" or custom abbreviated syntax "EPSG:2393+5717"),
- a OGC URN combining references for concatenated operations (e.g. "urn:ogc:def:coordinateOperation,coordinateOperation:EPSG::3895,coordinateOperation:EPSG::1618")
- a PROJJSON string. The jsonschema is at https://proj.org/schemas/v0.4/projjson.schema.json (*added in 6.2*)
- a compound CRS made from two object names separated with " + ". e.g. "WGS 84 + EGM96 height" (*added in 7.1*)

Example call:

```
PJ *P = proj_create(0, "+proj=etmerc +lat_0=38 +lon_0=125 +ellps=bessel");
```

If a proj-string contains a +type=crs option, then it is interpreted as a CRS definition. In particular geographic CRS are assumed to have axis in the longitude, latitude order and with degree angular unit. The use of proj-string to describe a CRS is discouraged. It is a legacy means of conveying CRS descriptions: use of object codes (EPSG:XXXX typically) or WKT description is recommended for better expressivity.

If a proj-string does not contain +type=crs, then it is interpreted as a coordination operation / transformation.

If creation of the transformation object fails, the function returns *0* and the PROJ error number is updated. The error number can be read with *proj_errno()* or *proj_context_errno()*.

The returned *PJ*-pointer should be deallocated with *proj_destroy()*.

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- Threading context.
- **definition** (*const char**) -- Proj-string of the desired transformation.

*PJ* ***proj_create_argv**(*PJ_CONTEXT* *ctx, int argc, char **argv)

Create a transformation object, or a CRS object, with argc/argv-style initialization. For this application each parameter in the defining proj-string is an entry in *argv*.

Example call:

```
char *args[3] = {"proj=utm", "zone=32", "ellps=GRS80"};
PJ* P = proj_create_argv(0, 3, args);
```

If there is a type=crs argument, then the arguments are interpreted as a CRS definition. In particular geographic CRS are assumed to have axis in the longitude, latitude order and with degree angular unit.

If there is no type=crs argument, then it is interpreted as a coordination operation / transformation.

If creation of the transformation object fails, the function returns *0* and the PROJ error number is updated. The error number can be read with *proj_errno()* or *proj_context_errno()*.

The returned *PJ*-pointer should be deallocated with *proj_destroy()*.

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- Threading context.
- **argc** (*int*) -- Count of arguments in *argv*
- **argv** (*char* **) -- Array of strings with proj-string parameters, e.g. +proj=merc

**Returns**

*PJ* *

*PJ* ***proj_create_crs_to_crs**(*PJ_CONTEXT* *ctx, const char *source_crs, const char *target_crs, *PJ_AREA* *area)

Create a transformation object that is a pipeline between two known coordinate reference systems.

source_crs and target_crs can be :

- a "AUTHORITY:CODE", like EPSG:25832. When using that syntax for a source CRS, the created pipeline will expect that the values passed to *proj_trans()* respect the axis order and axis unit of the official definition ( so for example, for EPSG:4326, with latitude first and longitude next, in degrees). Similarly, when using that syntax for a target CRS, output values will be emitted according to the official definition of this CRS.
- a PROJ string, like "+proj=longlat +datum=WGS84". When using that syntax, the axis order and unit for geographic CRS will be longitude, latitude, and the unit degrees.
- the name of a CRS as found in the PROJ database, e.g "WGS84", "NAD27", etc.
- more generally any string accepted by *proj_create()* representing a CRS

Starting with PROJ 9.2, source_crs (exclusively) or target_crs can be a CoordinateMetadata with an associated coordinate epoch.

Starting with PROJ 9.4, both source_crs and target_crs can be a CoordinateMetadata with an associated coordinate epoch, to perform changes of coordinate epochs. Note however than this is in practice limited to use of velocity grids inside the same dynamic CRS.

An "area of use" can be specified in area. When it is supplied, the more accurate transformation between two given systems can be chosen.

When no area of use is specific and several coordinate operations are possible depending on the area of use, this function will internally store those candidate coordinate operations in the return PJ object. Each subsequent coordinate transformation done with *proj_trans()* will then select the appropriate coordinate operation by comparing the input coordinates with the area of use of the candidate coordinate operations.

Example call:

```
PJ *P = proj_create_crs_to_crs(0, "EPSG:25832", "EPSG:25833", 0);
```

If creation of the transformation object fails, the function returns *0* and the PROJ error number is updated. The error number can be read with *proj_errno()* or *proj_context_errno()*.

The returned *PJ*-pointer should be deallocated with *proj_destroy()*.

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- Threading context.
- **source_crs** (*const char**) -- Source CRS or CoordinateMetadata.
- **target_crs** (*const char**) -- Destination SRS or CoordinateMetadata
- **area** (*PJ_AREA* *) -- Descriptor of the desired area for the transformation.

**Returns**

*PJ* *

*PJ* ***proj_create_crs_to_crs_from_pj**(*PJ_CONTEXT* *ctx, *PJ* *source_crs, *PJ* *target_crs, *PJ_AREA* *area, const char *const *options)

Added in version 6.2.0.

Create a transformation object that is a pipeline between two known coordinate reference systems.

This is the same as *proj_create_crs_to_crs()* except that the source and target CRS are passed as PJ* objects which must be of the CRS variety.

Starting with PROJ 9.2, source_crs (exclusively) or target_crs can be a CoordinateMetadata with an associated coordinate epoch.

Starting with PROJ 9.4, both source_crs and target_crs can be a CoordinateMetadata with an associated coordinate epoch, to perform changes of coordinate epochs. Note however than this is in practice limited to use of velocity grids inside the same dynamic CRS.

**Parameters**

- **options** -- a list of NUL terminated options, or NULL.

The list of supported options is:

- AUTHORITY=name: to restrict the authority of coordinate operations looked up in the database. When not specified, coordinate operations from any authority will be searched, with the restrictions set in the authority_to_authority_preference database table related to the authority of the source/target CRS themselves. If authority is set to `any`, then coordinate operations from any authority will be searched. If authority is a non-empty string different of `any`, then coordinate operations will be searched only in that authority namespace (e.g `EPSG`).
- ACCURACY=value: to set the minimum desired accuracy (in metres) of the candidate coordinate operations.
- ALLOW_BALLPARK=YES/NO: can be set to NO to disallow the use of *Ballpark transformation* in the candidate coordinate operations.
- ONLY_BEST=YES/NO: (PROJ >= 9.2) Can be set to YES to cause PROJ to error out if the best transformation, known of PROJ, and usable by PROJ if all grids known and usable by PROJ were accessible, cannot be used. Best transformation should be understood as the transformation returned by `proj_get_suggested_operation()` if all known grids were accessible (either locally or through network). Note that the default value for this option can be also set with the `PROJ_ONLY_BEST_DEFAULT` environment variable, or with the `only_best_default` setting of *proj.ini* (the ONLY_BEST option when specified overrides such default value).
- FORCE_OVER=YES/NO: can be set to YES to force the `+over` flag on the transformation returned by this function. See *Longitude Wrapping*

*PJ* ***proj_normalize_for_visualization**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Returns a PJ* object whose axis order is the one expected for visualization purposes.

The input object must be either:

- a coordinate operation, that has been created with proj_create_crs_to_crs(). If the axis order of its source or target CRS is northing,easting, then an axis swap operation will be inserted.
- or a CRS. The axis order of geographic CRS will be longitude, latitude [,height], and the one of projected CRS will be easting, northing [, height]

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS, or CoordinateOperation created with proj_create_crs_to_crs() (must not be NULL)

**Returns**

a new PJ* object to free with proj_destroy() in case of success, or nullptr in case of error

*PJ* ***proj_destroy**(*PJ* *P)

Deallocate a *PJ* transformation object.

**Parameters**

- **P** (const *PJ* *) -- Transformation object

**Returns**

*PJ* *

#### 10.5.3.3 Area of interest

Added in version 6.0.0.

*PJ_AREA* ***proj_area_create**(void)

Create an area of use.

Such an area of use is to be passed to *proj_create_crs_to_crs()* to specify the area of use for the choice of relevant coordinate operations.

**Returns**

*PJ_AREA* * to be deallocated with *proj_area_destroy()*

void **proj_area_set_bbox**(*PJ_AREA* *area, double west_lon_degree, double south_lat_degree, double east_lon_degree, double north_lat_degree)

Set the bounding box of the area of use

Such an area of use is to be passed to *proj_create_crs_to_crs()* to specify the area of use for the choice of relevant coordinate operations.

In the case of an area of use crossing the antimeridian (longitude +/- 180 degrees), *west_lon_degree* will be greater than *east_lon_degree*.

**Parameters**

- **area** -- Pointer to an object returned by *proj_area_create()*.
- **west_lon_degree** -- West longitude, in degrees. In [-180,180] range.
- **south_lat_degree** -- South latitude, in degrees. In [-90,90] range.
- **east_lon_degree** -- East longitude, in degrees. In [-180,180] range.
- **north_lat_degree** -- North latitude, in degrees. In [-90,90] range.

void **proj_area_destroy**(*PJ_AREA* *area)

Deallocate a *PJ_AREA* object.

:param PJ_AREA* area

#### 10.5.3.4 Coordinate transformation

*PJ_COORD* **proj_trans**(*PJ* *P, *PJ_DIRECTION* direction, *PJ_COORD* coord)

Transform a single *PJ_COORD* coordinate.

If the input coordinate contains any NaNs you are guaranteed to get a coordinate with all NaNs as a result.

Changed in version 9.2.0: Define NaN handling. Prior NaN handling behavior was undefined.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Transformation direction.
- **coord** (*PJ_COORD*) -- Coordinate that will be transformed.

**Returns**

*PJ_COORD*

*PJ* ***proj_trans_get_last_used_operation**(*PJ* *P)

Added in version 9.1.0.

Return the operation used during the last invocation of proj_trans(). This is especially useful when P has been created with proj_create_crs_to_crs() and has several alternative operations. The returned object must be freed with proj_destroy().

**Parameters**

- **P** (*PJ* *) -- Transformation object

**Returns**

*PJ* *

size_t **proj_trans_generic**(*PJ* *P, *PJ_DIRECTION* direction, double *x, size_t sx, size_t nx, double *y, size_t sy, size_t ny, double *z, size_t sz, size_t nz, double *t, size_t st, size_t nt)

Transform a series of coordinates, where the individual coordinate dimension may be represented by an array that is either

1. fully populated
2. a null pointer and/or a length of zero, which will be treated as a fully populated array of zeroes
3. of length one, i.e. a constant, which will be treated as a fully populated array of that constant value

> **Note**
>
> Even though the coordinate components are named *x*, *y*, *z* and *t*, axis ordering of the to and from CRS is respected. Transformations exhibit the same behavior as if they were gathered in a *PJ_COORD* struct.

The strides, *sx*, *sy*, *sz*, *st*, represent the step length, in bytes, between consecutive elements of the corresponding array. This makes it possible for *proj_trans_generic()* to handle transformation of a large class of application specific data structures, without necessarily understanding the data structure format, as in:

```
typedef struct {
    double x, y;
    int quality_level;
    char surveyor_name[134];
} XYQS;

XYQS survey[345];
double height = 23.45;
size_t stride = sizeof (XYQS);

...

proj_trans_generic (
    P, PJ_INV,
    &(survey[0].x), stride, 345,  /*  We have 345 eastings  */
    &(survey[0].y), stride, 345,  /*  ...and 345 northings. */
    &height, sizeof(double), 1,   /*  The height is the constant  23.45 m */
    0, 0, 0                       /*  and the time is the constant 0.00 s */
);
```

This is similar to the inner workings of the deprecated `pj_transform()` function, but the stride functionality has been generalized to work for any size of basic unit, not just a fixed number of doubles.

In most cases, the stride will be identical for x, y, z, and t, since they will typically be either individual arrays (`stride = sizeof(double)`), or strided views into an array of application specific data structures (`stride = sizeof (...)`).

But in order to support cases where *x*, *y*, *z*, and *t* come from heterogeneous sources, individual strides, *sx*, *sy*, *sz*, *st*, are used.

**Note**

Since *proj_trans_generic()* does its work *in place*, this means that even the supposedly constants (i.e. length 1 arrays) will return from the call in altered state. Hence, remember to reinitialize between repeated calls.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Transformation direction.
- **x** (*double* *) -- Array of x-coordinates
- **sx** (*size_t*) -- Step length, in bytes, between consecutive elements of the corresponding array
- **nx** (*size_t*) -- Number of elements in the corresponding array
- **y** (*double* *) -- Array of y-coordinates
- **sy** (*size_t*) -- Step length, in bytes, between consecutive elements of the corresponding array
- **ny** (*size_t*) -- Number of elements in the corresponding array
- **z** (*double* *) -- Array of z-coordinates
- **sz** (*size_t*) -- Step length, in bytes, between consecutive elements of the corresponding array
- **nz** (*size_t*) -- Number of elements in the corresponding array

- **t** (*double* *) -- Array of t-coordinates
- **st** (*size_t*) -- Step length, in bytes, between consecutive elements of the corresponding array
- **nt** (*size_t*) -- Number of elements in the corresponding array

**Returns**
Number of transformations successfully completed

int **proj_trans_array**(*PJ* *P, *PJ_DIRECTION* direction, size_t n, *PJ_COORD* *coord)

Batch transform an array of *PJ_COORD*.

Performs transformation on all points, even if errors occur on some points (new to 8.0. Previous versions would exit early in case of failure on a given point)

Individual points that fail to transform will have their components set to HUGE_VAL

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Transformation direction.
- **n** (*size_t*) -- Number of coordinates in *coord*

**Returns**
*int* 0 if all observations are transformed without error, otherwise returns error number. This error number will be a precise error number if all coordinates that fail to transform for the same reason, or a generic error code if they fail for different reasons.

int **proj_trans_bounds**(*PJ_CONTEXT* *context, *PJ* *P, *PJ_DIRECTION* direction, double xmin, double ymin, double xmax, double ymax, double *out_xmin, double *out_ymin, double *out_xmax, double *out_ymax, int densify_pts)

Transform boundary.

Transform boundary densifying the edges to account for nonlinear transformations along these edges and extracting the outermost bounds.

If the destination CRS is geographic, the first axis is longitude, and *out_xmax < *out_xmin then the bounds crossed the antimeridian. In this scenario there are two polygons, one on each side of the antimeridian. The first polygon should be constructed with (*out_xmin, *out_ymin, 180, ymax) and the second with (-180, *out_ymin, *out_xmax, *out_ymax).

If the destination CRS is geographic, the first axis is latitude, and *out_ymax < *out_ymin then the bounds crossed the antimeridian. In this scenario there are two polygons, one on each side of the antimeridian. The first polygon should be constructed with (*out_ymin, *out_xmin, *out_ymax, 180) and the second with (*out_ymin, -180, *out_ymax, *out_xmax).

For transformations involving a 3D CRS, consult *proj_trans_bounds_3D()*.

**See also**

*proj_trans_bounds_3D()*

**Since**
8.2

**Parameters**

- **context** -- The PJ_CONTEXT object.
- **P** -- The PJ object representing the transformation.
- **direction** -- The direction of the transformation.
- **xmin** -- Minimum bounding coordinate of the first axis in source CRS (target CRS if direction is inverse).
- **ymin** -- Minimum bounding coordinate of the second axis in source CRS. (target CRS if direction is inverse).
- **xmax** -- Maximum bounding coordinate of the first axis in source CRS. (target CRS if direction is inverse).
- **ymax** -- Maximum bounding coordinate of the second axis in source CRS. (target CRS if direction is inverse).
- **out_xmin** -- Minimum bounding coordinate of the first axis in target CRS (source CRS if direction is inverse).
- **out_ymin** -- Minimum bounding coordinate of the second axis in target CRS. (source CRS if direction is inverse).
- **out_xmax** -- Maximum bounding coordinate of the first axis in target CRS. (source CRS if direction is inverse).
- **out_ymax** -- Maximum bounding coordinate of the second axis in target CRS. (source CRS if direction is inverse).
- **densify_pts** -- Recommended to use 21. This is the number of points to use to densify the bounding polygon in the transformation.

**Returns**
an integer. 1 if successful. 0 if failures encountered.

int **proj_trans_bounds_3D**(*PJ_CONTEXT* *context, *PJ* *P, *PJ_DIRECTION* direction, const double xmin, const double ymin, const double zmin, const double xmax, const double ymax, const double zmax, double *out_xmin, double *out_ymin, double *out_zmin, double *out_xmax, double *out_ymax, double *out_zmax, const int densify_pts)

Transform boundary, taking into account 3D coordinates.

Transform boundary densifying the edges to account for nonlinear transformations along these edges and extracting the outermost bounds.

Note that the current implementation is not "perfect" when the source CRS is geocentric, the target CRS is geographic, and the input bounding box includes the center of the Earth, a pole or the antimeridian. In those circumstances, exact values of the latitude of longitude of discontinuity will not be returned.

If one of the source or target CRS of the transformation is not 3D, the values of *out_zmin / *out_zmax may not be significant.

For 2D or "2.5D" transformation (that is planar component is geographic/coordinates and 3D axis is elevation), the documentation of *proj_trans_bounds()* applies.

**Since**
9.6

**Parameters**

- **context** -- The PJ_CONTEXT object.

- **P** -- The PJ object representing the transformation.
- **direction** -- The direction of the transformation.
- **xmin** -- Minimum bounding coordinate of the first axis in source CRS (target CRS if direction is inverse).
- **ymin** -- Minimum bounding coordinate of the second axis in source CRS. (target CRS if direction is inverse).
- **zmin** -- Minimum bounding coordinate of the third axis in source CRS. (target CRS if direction is inverse).
- **xmax** -- Maximum bounding coordinate of the first axis in source CRS. (target CRS if direction is inverse).
- **ymax** -- Maximum bounding coordinate of the second axis in source CRS. (target CRS if direction is inverse).
- **zmax** -- Maximum bounding coordinate of the third axis in source CRS. (target CRS if direction is inverse).
- **out_xmin** -- Minimum bounding coordinate of the first axis in target CRS (source CRS if direction is inverse).
- **out_ymin** -- Minimum bounding coordinate of the second axis in target CRS. (source CRS if direction is inverse).
- **out_zmin** -- Minimum bounding coordinate of the third axis in target CRS. (source CRS if direction is inverse).
- **out_xmax** -- Maximum bounding coordinate of the first axis in target CRS. (source CRS if direction is inverse).
- **out_ymax** -- Maximum bounding coordinate of the second axis in target CRS. (source CRS if direction is inverse).
- **out_zmax** -- Maximum bounding coordinate of the third axis in target CRS. (source CRS if direction is inverse).
- **densify_pts** -- Recommended to use 21. This is the number of points to use to densify the bounding polygon in the transformation.

**Returns**

an integer. 1 if successful. 0 if failures encountered.

#### 10.5.3.5 Error reporting

int **proj_errno**(*PJ* *P)

Get a reading of the current error-state of *P*. An non-zero error codes indicates an error either with the transformation setup or during a transformation. In cases *P* is *0* the error number of the default context is read. A text representation of the error number can be retrieved with *proj_errno_string()*.

Consult *Error codes* for the list of error codes (PROJ >= 8.0)

**Parameters**

- **P** (*PJ* *) -- Transformation object

**Returns**

*int*

int **proj_context_errno**(*PJ_CONTEXT* *ctx)

Get a reading of the current error-state of *ctx*. An non-zero error codes indicates an error either with the transformation setup or during a transformation. A text representation of the error number can be retrieved with *proj_errno_string()*.

Consult *Error codes* for the list of error codes (PROJ >= 8.0)

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- threading context.

**Returns**

*int*

void **proj_errno_set**(*PJ* *P, int err)

Change the error-state of *P* to *err*.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **err** (*int*) -- Error number.

int **proj_errno_reset**(*PJ* *P)

Clears the error number in *P*, and bubbles it up to the context.

Example:

```
void foo (PJ *P) {
    int last_errno = proj_errno_reset (P);

    do_something_with_P (P);

    /* failure - keep latest error status */
    if (proj_errno(P))
        return;
    /* success - restore previous error status */
    proj_errno_restore (P, last_errno);
    return;
}
```

**Parameters**

- **P** (*PJ* *) -- Transformation object

**Returns**

*int* Returns the previous value of the errno, for convenient reset/restore operations.

void **proj_errno_restore**(*PJ* *P, int err)

Reduce some mental impedance in the canonical reset/restore use case: Basically, *proj_errno_restore()* is a synonym for *proj_errno_set()*, but the use cases are very different: *set* indicate an error to higher level user code, *restore* passes previously set error indicators in case of no errors at this level.

Hence, although the inner working is identical, we provide both options, to avoid some rather confusing real world code.

See usage example under *proj_errno_reset()*

**Parameters**

- **P** (*PJ* *) -- Transformation object

- **err** (*int*) -- Error number.

const char ***proj_errno_string**(int err)

Added in version 5.1.0.

Get a text representation of an error number.

Deprecated since version This: function is potentially thread-unsafe, replaced by *proj_context_errno_string()*.

**Parameters**

- **err** (*int*) -- Error number.

**Returns**
*const char** String with description of error.

const char ***proj_context_errno_string**(*PJ_CONTEXT* *ctx, int err)

Added in version 8.0.0.

Get a text representation of an error number.

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- threading context.
- **err** (*int*) -- Error number.

**Returns**
*const char** String with description of error.

#### 10.5.3.6 Logging

*PJ_LOG_LEVEL* **proj_log_level**(*PJ_CONTEXT* *ctx, *PJ_LOG_LEVEL* level)

Get and set logging level for a given context. Changes the log level to *level* and returns the previous logging level. If called with *level* set to PJ_LOG_TELL the function returns the current logging level without changing it.

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- Threading context.
- **level** (*PJ_LOG_LEVEL*) -- New logging level.

**Returns**
*PJ_LOG_LEVEL*

Added in version 5.1.0.

void **proj_log_func**(*PJ_CONTEXT* *ctx, void *app_data, PJ_LOG_FUNCTION logf)

Override the internal log function of PROJ.

**Parameters**

- **ctx** (*PJ_CONTEXT* *) -- Threading context.
- **app_data** (*void* *) -- Pointer to data structure used by the calling application.
- **logf** (PJ_LOG_FUNCTION) -- Log function that overrides the PROJ log function.

Added in version 5.1.0.

#### 10.5.3.7 Info functions

*PJ_INFO* **proj_info**(void)

Get information about the current instance of the PROJ library.

**Returns**
*PJ_INFO*

*PJ_PROJ_INFO* **proj_pj_info**(const *PJ* *P)

Get information about a specific transformation object, *P*.

**Parameters**

- **P** (const *PJ* *) -- Transformation object

**Returns**
*PJ_PROJ_INFO*

*PJ_GRID_INFO* **proj_grid_info**(const char *gridname)

Get information about a specific grid.

**Parameters**

- **gridname** (*const char**) -- Gridname in the PROJ searchpath

**Returns**
*PJ_GRID_INFO*

*PJ_INIT_INFO* **proj_init_info**(const char *initname)

Get information about a specific init file.

**Parameters**

- **initname** (*const char**) -- Init file in the PROJ searchpath

**Returns**
*PJ_INIT_INFO*

#### 10.5.3.8 Lists

const *PJ_OPERATIONS* ***proj_list_operations**(void)

Get a pointer to an array of all operations in PROJ. The last entry of the returned array is a NULL-entry. The array is statically allocated and does not need to be freed after use.

Print a list of all operations in PROJ:

```
PJ_OPERATIONS *ops;
for (ops = proj_list_operations(); ops->id; ++ops)
    printf("%s\n", ops->id);
```

**Returns**
const *PJ_OPERATIONS* *

const *PJ_ELLPS* ***proj_list_ellps**(void)

Get a pointer to an array of ellipsoids defined in PROJ. The last entry of the returned array is a NULL-entry. The array is statically allocated and does not need to be freed after use.

**Returns**
const *PJ_ELLPS* *

const *PJ_UNITS* ***proj_list_units**(void)*

Get a pointer to an array of distance units defined in PROJ. The last entry of the returned array is a NULL-entry. The array is statically allocated and does not need to be freed after use.

Note: starting with PROJ 7.1, this function is deprecated by `proj_get_units_from_database()`

**Returns**
const *PJ_UNITS* *

const *PJ_PRIME_MERIDIANS* ***proj_list_prime_meridians**(void)*

Get a pointer to an array of hard-coded prime meridians defined in PROJ. Note that this list is no longer updated. The last entry of the returned array is a NULL-entry. The array is statically allocated and does not need to be freed after use.

**Returns**
const *PJ_PRIME_MERIDIANS* *

#### 10.5.3.9 Distances

double **proj_lp_dist**(const *PJ* *P, *PJ_COORD* a, *PJ_COORD* b)

Calculate geodesic distance between two points in geodetic coordinates. The calculated distance is between the two points located on the ellipsoid.

The coordinates in *a* and *b* needs to be given as longitude and latitude in radians. Note that the axis order of the *P* object is not taken into account in this function, so even though a CRS object comes with axis ordering latitude/longitude coordinates used in this function should be reordered as longitude/latitude.

**Parameters**

- **P** (const *PJ* *) -- Transformation or CRS object
- **a** (*PJ_COORD*) -- Coordinate of first point
- **b** (*PJ_COORD*) -- Coordinate of second point

**Returns**
*double* Distance between *a* and *b* in meters.

double **proj_lpz_dist**(const *PJ* *P, *PJ_COORD* a, *PJ_COORD* b)

Calculate geodesic distance between two points in geodetic coordinates. Similar to *proj_lp_dist()* but also takes the height above the ellipsoid into account.

The coordinates in *a* and *b* needs to be given as longitude and latitude in radians. Note that the axis order of the *P* object is not taken into account in this function, so even though a CRS object comes with axis ordering latitude/longitude coordinates used in this function should be reordered as longitude/latitude.

**Parameters**

- **P** (const *PJ* *) -- Transformation or CRS object
- **a** (*PJ_COORD*) -- Coordinate of first point
- **b** (*PJ_COORD*) -- Coordinate of second point

**Returns**
*double* Distance between *a* and *b* in meters.

double **proj_xy_dist**(*PJ_COORD* a, *PJ_COORD* b)

Calculate 2-dimensional euclidean between two projected coordinates.

**Parameters**

- **a** (*PJ_COORD*) -- First coordinate

- **b** (*PJ_COORD*) -- Second coordinate

**Returns**
*double* Distance between *a* and *b* in meters.

double **proj_xyz_dist**(*PJ_COORD* a, *PJ_COORD* b)

Calculate 3-dimensional euclidean between two projected coordinates.

**Parameters**

- **a** (*PJ_COORD*) -- First coordinate
- **b** (*PJ_COORD*) -- Second coordinate

**Returns**
*double* Distance between *a* and *b* in meters.

*PJ_COORD* **proj_geod**(const *PJ* *P, *PJ_COORD* a, *PJ_COORD* b)

Calculate the geodesic distance as well as forward and reverse azimuth between two points on the ellipsoid.

The coordinates in *a* and *b* needs to be given as longitude and latitude in radians. Note that the axis order of the *P* object is not taken into account in this function, so even though a CRS object comes with axis ordering latitude/longitude coordinates used in this function should be reordered as longitude/latitude.

**Parameters**

- **P** (const *PJ* *) -- Transformation or CRS object
- **a** (*PJ_COORD*) -- Coordinate of first point
- **b** (*PJ_COORD*) -- Coordinate of second point

**Returns**
*PJ_COORD* where the first value is the distance between *a* and *b* in meters, the second value is the forward azimuth and the third value is the reverse azimuth. The fourth coordinate value is unused.

#### 10.5.3.10 Various

*PJ_COORD* **proj_coord**(double x, double y, double z, double t)

Initializer for the *PJ_COORD* union. The function is shorthand for the otherwise convoluted assignment. Equivalent to

```
PJ_COORD c = {{10.0, 20.0, 30.0, 40.0}};
```

or

```
PJ_COORD c;
// Assign using the PJ_XYZT struct in the union
c.xyzt.x = 10.0;
c.xyzt.y = 20.0;
c.xyzt.z = 30.0;
c.xyzt.t = 40.0;
```

Since *PJ_COORD* is a union of structs, the above assignment can also be expressed in terms of the other types in the union, e.g. *PJ_UVWT* or *PJ_LPZT*.

**Parameters**

- **x** (*double*) -- 1st component in a *PJ_COORD*
- **y** (*double*) -- 2nd component in a *PJ_COORD*

- **z** (*double*) -- 3rd component in a *PJ_COORD*
- **t** (*double*) -- 4th component in a *PJ_COORD*

**Returns**
*PJ_COORD*

double **proj_roundtrip**(*PJ* *P, *PJ_DIRECTION* direction, int n, *PJ_COORD* *coord)

Measure internal consistency of a given transformation. The function performs *n* round trip transformations starting in either the forward or reverse *direction*. Returns the euclidean distance of the starting point `coo` and the resulting coordinate after *n* iterations back and forth.

If the input coordinate has any NaNs and the expected output of all NaNs is returned then the final distance will be 0.

Changed in version 9.2.0: Define expected NaN distance of 0.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Starting direction of transformation
- **n** (*int*) -- Number of roundtrip transformations
- **coord** (*PJ_COORD* *) -- Input coordinate

**Returns**
*double* Distance between original coordinate and the resulting coordinate after *n* transformation iterations.

*PJ_FACTORS* **proj_factors**(*PJ* *P, *PJ_COORD* lp)

Calculate various cartographic properties, such as scale factors, angular distortion and meridian convergence. Depending on the underlying projection values will be calculated either numerically (default) or analytically.

Starting with PROJ 8.2, the P object can be a projected CRS, for example instantiated from a EPSG CRS code. The factors computed will be those of the map projection implied by the transformation from the base geographic CRS of the projected CRS to the projected CRS. Starting with PROJ 9.6, to improve performance on repeated calls on a projected CRS object, the above steps will modify the internal state of the provided P object, and thus calling this function concurrently from multiple threads on the same P object will no longer be supported.

The input geodetic coordinate lp should be such that lp.lam is the longitude in radian, and lp.phi the latitude in radian (thus independently of the definition of the base CRS, if P is a projected CRS).

The function also calculates the partial derivatives of the given coordinate.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **lp** (*PJ_COORD*) -- Geodetic coordinate

**Returns**
*PJ_FACTORS*

double **proj_torad**(double angle_in_degrees)

Convert degrees to radians.

**Parameters**

- **angle_in_degrees** (*double*) -- Degrees

**Returns**
*double* Radians

double **proj_todeg**(double angle_in_radians)

Convert radians to degrees

**Parameters**

- **angle_in_radians** (*double*) -- Radians

**Returns**

*double* Degrees

double **proj_dmstor**(const char *is, char **rs)

Convert string of degrees, minutes and seconds to radians. Works similarly to the C standard library function `strtod()`.

**Parameters**

- **is** (*const char**) -- Value to be converted to radians
- **rs** -- Reference to an already allocated char*, whose value is set by the function to the next character in *is* after the numerical value.

char ***proj_rtodms**(char *s, double r, int pos, int neg)

Convert radians to string representation of degrees, minutes and seconds.

Deprecated since version 9.2: Use `proj_rtodms2()` instead.

**Parameters**

- **s** (*char **) -- Buffer that holds the output string
- **r** (*double*) -- Value to convert to dms-representation
- **pos** (*int*) -- Character denoting positive direction, typically *'N'* or *'E'*.
- **neg** (*int*) -- Character denoting negative direction, typically *'S'* or *'W'*.

**Returns**

*char** Pointer to output buffer (same as *s*)

char ***proj_rtodms2**(char *s, size_t sizeof_s, double r, int pos, int neg)

Added in version 9.2.0.

Convert radians to string representation of degrees, minutes and seconds.

**Parameters**

- **s** (*char **) -- Buffer that holds the output string
- **sizeof_s** (*size_t*) -- Size of s buffer
- **r** (*double*) -- Value to convert to dms-representation
- **pos** (*int*) -- Character denoting positive direction, typically *'N'* or *'E'*.
- **neg** (*int*) -- Character denoting negative direction, typically *'S'* or *'W'*.

**Returns**

*char** Pointer to output buffer (same as *s*)

int **proj_angular_input**(*PJ* *P, enum *PJ_DIRECTION* dir)

Check if an operation expects input in radians or not.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Starting direction of transformation

**Returns**
*int* 1 if input units is expected in radians, otherwise 0

int **proj_angular_output**(*PJ* *P, enum *PJ_DIRECTION* dir)

Check if an operation returns output in radians or not.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Starting direction of transformation

**Returns**
*int* 1 if output units is expected in radians, otherwise 0

int **proj_degree_input**(*PJ* *P, enum *PJ_DIRECTION* dir)

Added in version 7.1.0.

Check if an operation expects input in degrees or not.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Starting direction of transformation

**Returns**
*int* 1 if input units is expected in degrees, otherwise 0

int **proj_degree_output**(*PJ* *P, enum *PJ_DIRECTION* dir)

Added in version 7.1.0.

Check if an operation returns output in degrees or not.

**Parameters**

- **P** (*PJ* *) -- Transformation object
- **direction** (*PJ_DIRECTION*) -- Starting direction of transformation

**Returns**
*int* 1 if output units is expected in degrees, otherwise 0

#### 10.5.3.11 Setting custom I/O functions

Added in version 7.0.0.

int **proj_context_set_fileapi**(*PJ_CONTEXT* *ctx, const *PROJ_FILE_API* *fileapi, void *user_data)

Set a file API

All callbacks should be provided (non NULL pointers). If read-only usage is intended, then the callbacks might have a dummy implementation.

**Since**
7.0

> **Note**
>
> Those callbacks will not be used for SQLite3 database access. If custom I/O is desired for that, then *proj_context_set_sqlite3_vfs_name()* should be used.

**Parameters**

- **ctx** -- PROJ context, or NULL
- **fileapi** -- Pointer to file API structure (content will be copied).
- **user_data** -- Arbitrary pointer provided by the user, and passed to the above callbacks. May be NULL.

**Returns**

TRUE in case of success.

void **proj_context_set_sqlite3_vfs_name**(*PJ_CONTEXT* *ctx, const char *name)

Set the name of a custom SQLite3 VFS.

This should be a valid SQLite3 VFS name, such as the one passed to the sqlite3_vfs_register(). See https://www.sqlite.org/vfs.html

It will be used to read proj.db or create&access the cache.db file in the PROJ user writable directory.

**Since**

7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **name** -- SQLite3 VFS name. If NULL is passed, default implementation by SQLite will be used.

void **proj_context_set_file_finder**(*PJ_CONTEXT* *ctx, proj_file_finder finder, void *user_data)

Assign a file finder callback to a context.

This callback will be used whenever PROJ must open one of its resource files (proj.db database, grids, etc...)

The callback will be called with the context currently in use at the moment where it is used (not necessarily the one provided during this call), and with the provided user_data (which may be NULL). The user_data must remain valid during the whole lifetime of the context.

A finder set on the default context will be inherited by contexts created later.

**Since**

PROJ 6.0

**Parameters**

- **ctx** -- PROJ context, or NULL for the default context.
- **finder** -- Finder callback. May be NULL
- **user_data** -- User data provided to the finder callback. May be NULL.

void **proj_context_set_search_paths**(*PJ_CONTEXT* *ctx, int count_paths, const char *const *paths)

Sets search paths.

Those search paths will be used whenever PROJ must open one of its resource files (proj.db database, grids, etc...)

If set on the default context, they will be inherited by contexts created later.

Starting with PROJ 7.0, the path(s) should be encoded in UTF-8.

**Since**
PROJ 6.0

**Parameters**

- **ctx** -- PROJ context, or NULL for the default context.
- **count_paths** -- Number of paths. 0 if paths == NULL.
- **paths** -- Paths. May be NULL.

void **proj_context_set_ca_bundle_path**(*PJ_CONTEXT* *ctx, const char *path)

Sets CA Bundle path.

Those CA Bundle path will be used by PROJ when curl and PROJ_NETWORK are enabled.

If set on the default context, they will be inherited by contexts created later.

The path should be encoded in UTF-8.

**Since**
PROJ 7.2

**Parameters**

- **ctx** -- PROJ context, or NULL for the default context.
- **path** -- Path. May be NULL.

#### 10.5.3.12 Network related functionality

Added in version 7.0.0.

int **proj_context_set_network_callbacks**(*PJ_CONTEXT* *ctx, *proj_network_open_cbk_type* open_cbk, *proj_network_close_cbk_type* close_cbk, *proj_network_get_header_value_cbk_type* get_header_value_cbk, *proj_network_read_range_type* read_range_cbk, void *user_data)

Define a custom set of callbacks for network access.

All callbacks should be provided (non NULL pointers).

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **open_cbk** -- Callback to open a remote file given its URL
- **close_cbk** -- Callback to close a remote file.
- **get_header_value_cbk** -- Callback to get HTTP headers
- **read_range_cbk** -- Callback to read a range of bytes inside a remote file.

- **user_data** -- Arbitrary pointer provided by the user, and passed to the above callbacks. May be NULL.

**Returns**
TRUE in case of success.

int **proj_context_set_enable_network**(*PJ_CONTEXT* *ctx, int enabled)

Enable or disable network access.

This overrides the default endpoint in the PROJ configuration file or with the PROJ_NETWORK environment variable.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **enable** -- TRUE if network access is allowed.

**Returns**
TRUE if network access is possible. That is either libcurl is available, or an alternate interface has been set.

int **proj_context_is_network_enabled**(*PJ_CONTEXT* *ctx)

Return if network access is enabled.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL

**Returns**
TRUE if network access has been enabled

void **proj_context_set_url_endpoint**(*PJ_CONTEXT* *ctx, const char *url)

Define the URL endpoint to query for remote grids.

This overrides the default endpoint in the PROJ configuration file or with the PROJ_NETWORK_ENDPOINT environment variable.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **url** -- Endpoint URL. Must NOT be NULL.

const char *proj_context_get_url_endpoint(*PJ_CONTEXT* *ctx)

Get the URL endpoint to query for remote grids.

**Since**

7.1

**Parameters**

- **ctx** -- PROJ context, or NULL

**Returns**

Endpoint URL. The returned pointer would be invalidated by a later call to *proj_context_set_url_endpoint()*

const char *proj_context_get_user_writable_directory(*PJ_CONTEXT* *ctx, int create)

Get the PROJ user writable directory for downloadable resource files, such as datum shift grids.

**See also**

proj_context_set_user_writable_directory()

**Since**

7.1

**Parameters**

- **ctx** -- PROJ context, or NULL
- **create** -- If set to TRUE, create the directory if it does not exist already.

**Returns**

The path to the PROJ user writable directory.

void proj_grid_cache_set_enable(*PJ_CONTEXT* *ctx, int enabled)

Enable or disable the local cache of grid chunks

This overrides the setting in the PROJ configuration file.

**Since**

7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **enabled** -- TRUE if the cache is enabled.

void proj_grid_cache_set_filename(*PJ_CONTEXT* *ctx, const char *fullname)

Override, for the considered context, the path and file of the local cache of grid chunks.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **fullname** -- Full name to the cache (encoded in UTF-8). If set to NULL, caching will be disabled.

void **proj_grid_cache_set_max_size**(*PJ_CONTEXT* *ctx, int max_size_MB)

Override, for the considered context, the maximum size of the local cache of grid chunks.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **max_size_MB** -- Maximum size, in mega-bytes (1024*1024 bytes), or negative value to set unlimited size.

void **proj_grid_cache_set_ttl**(*PJ_CONTEXT* *ctx, int ttl_seconds)

Override, for the considered context, the time-to-live delay for re-checking if the cached properties of files are still up-to-date.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **ttl_seconds** -- Delay in seconds. Use negative value for no expiration.

void **proj_grid_cache_clear**(*PJ_CONTEXT* *ctx)

Clear the local cache of grid chunks.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL

int **proj_is_download_needed**(*PJ_CONTEXT* *ctx, const char *url_or_filename, int ignore_ttl_setting)

Return if a file must be downloaded or is already available in the PROJ user-writable directory.

The file will be determinted to have to be downloaded if it does not exist yet in the user-writable directory, or if it is determined that a more recent version exists. To determine if a more recent version exists, PROJ will use the "downloaded_file_properties" table of its grid cache database. Consequently files manually placed in the user-writable directory without using this function would be considered as non-existing/obsolete and would be unconditionally downloaded again.

This function can only be used if networking is enabled, and either the default curl network API or a custom one have been installed.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **url_or_filename** -- URL or filename (without directory component)
- **ignore_ttl_setting** -- If set to FALSE, PROJ will only check the recentness of an already downloaded file, if the delay between the last time it has been verified and the current time exceeds the TTL setting. This can save network accesses. If set to TRUE, PROJ will unconditionally check from the server the recentness of the file.

**Returns**
TRUE if the file must be downloaded with *proj_download_file()*

int **proj_download_file**(*PJ_CONTEXT* *ctx, const char *url_or_filename, int ignore_ttl_setting, int (*progress_cbk)(*PJ_CONTEXT**, double pct, void *user_data), void *user_data)

Download a file in the PROJ user-writable directory.

The file will only be downloaded if it does not exist yet in the user-writable directory, or if it is determined that a more recent version exists. To determine if a more recent version exists, PROJ will use the "downloaded_file_properties" table of its grid cache database. Consequently files manually placed in the user-writable directory without using this function would be considered as non-existing/obsolete and would be unconditionally downloaded again.

This function can only be used if networking is enabled, and either the default curl network API or a custom one have been installed.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL
- **url_or_filename** -- URL or filename (without directory component)
- **ignore_ttl_setting** -- If set to FALSE, PROJ will only check the recentness of an already downloaded file, if the delay between the last time it has been verified and the current time exceeds the TTL setting. This can save network accesses. If set to TRUE, PROJ will unconditionally check from the server the recentness of the file.
- **progress_cbk** -- Progress callback, or NULL. The passed percentage is in the [0, 1] range. The progress callback must return TRUE if download must be continued.
- **user_data** -- User data to provide to the progress callback, or NULL

**Returns**
TRUE if the download was successful (or not needed)

#### 10.5.3.13 Cleanup

void **proj_cleanup**()

Added in version 6.2.0.

This function frees global resources (grids, cache of +init files). It should be called typically before process termination, and *after* having freed PJ and PJ_CONTEXT objects.

#### 10.5.3.14 C API for ISO-19111 functionality

Added in version 6.0.0.

The PJ* objects returned by *proj_create_from_wkt()*, *proj_create_from_database()* and other functions in that section will have generally minimal interaction with the functions declared in the previous sections (calling those functions on those objects will either return an error or default/nonsensical values). The exception is for ISO19111 objects of type CoordinateOperation that can be exported as a valid PROJ pipeline. In this case, objects will work for example with *proj_trans_generic()*. Conversely, objects returned by *proj_create()* and *proj_create_argv()*, which are not of type CRS (can be tested with *proj_is_crs()*), will return an error when used with functions of this section.

##### Base functions

void **proj_string_list_destroy**(*PROJ_STRING_LIST* list)

Free a list of NULL terminated strings.

void **proj_context_set_autoclose_database**(*PJ_CONTEXT* *ctx, int autoclose)

Starting with PROJ 8.1, this function does nothing.

If you want to take into account changes to the PROJ database, you need to re-create a new context.

**Since**

6.2 deprecated Since 8.1

**Parameters**

- **ctx** -- Ignored
- **autoclose** -- Ignored

int **proj_context_set_database_path**(*PJ_CONTEXT* *ctx, const char *dbPath, const char *const *auxDbPaths, const char *const *options)

Explicitly point to the main PROJ CRS and coordinate operation definition database ("proj.db"), and potentially auxiliary databases with same structure.

Starting with PROJ 8.1, if the auxDbPaths parameter is an empty array, the PROJ_AUX_DB environment variable will be used, if set. It must contain one or several paths. If several paths are provided, they must be separated by the colon (:) character on Unix, and on Windows, by the semi-colon (;) character.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **dbPath** -- Path to main database, or NULL for default.
- **auxDbPaths** -- NULL-terminated list of auxiliary database filenames, or NULL.
- **options** -- should be set to NULL for now

**Returns**
TRUE in case of success

const char *__proj_context_get_database_path__(*PJ_CONTEXT* *ctx)

Returns the path to the database.

The returned pointer remains valid while ctx is valid, and until *proj_context_set_database_path()* is called.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context

**Returns**
path, or nullptr

const char *__proj_context_get_database_metadata__(*PJ_CONTEXT* *ctx, const char *key)

Return a metadata from the database.

The returned pointer remains valid while ctx is valid, and until *proj_context_get_database_metadata()* is called.

Available keys:

- DATABASE.LAYOUT.VERSION.MAJOR
- DATABASE.LAYOUT.VERSION.MINOR
- EPSG.VERSION
- EPSG.DATE
- ESRI.VERSION
- ESRI.DATE
- IGNF.SOURCE
- IGNF.VERSION
- IGNF.DATE
- NKG.SOURCE
- NKG.VERSION
- NKG.DATE
- PROJ.VERSION
- PROJ_DATA.VERSION : PROJ-data version most compatible with this database.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **key** -- Metadata key. Must not be NULL

**Returns**
value, or nullptr

*PROJ_STRING_LIST* __proj_context_get_database_structure__(*PJ_CONTEXT* *ctx, const char *const *options)

Return the database structure.

Return SQL statements to run to initiate a new valid auxiliary empty database. It contains definitions of tables, views and triggers, as well as metadata for the version of the layout of the database.

**Since**
8.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **options** -- null-terminated list of options, or NULL. None currently.

**Returns**
list of SQL statements (to be freed with *proj_string_list_destroy()*), or NULL in case of error.

*PJ_GUESSED_WKT_DIALECT* **proj_context_guess_wkt_dialect**(*PJ_CONTEXT* *ctx, const char *wkt)

Guess the "dialect" of the WKT string.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **wkt** -- String (must not be NULL)

*PJ* ***proj_create_from_wkt**(*PJ_CONTEXT* *ctx, const char *wkt, const char *const *options, *PROJ_STRING_LIST* *out_warnings, *PROJ_STRING_LIST* *out_grammar_errors)

Instantiate an object from a WKT string.

This function calls *osgeo::proj::io::WKTParser::createFromWKT()*

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

The distinction between warnings and grammar errors is somewhat artificial and does not tell much about the real criticity of the non-compliance. Some warnings may be more concerning than some grammar errors. Human expertise (or, by the time this comment will be read, specialized AI) is generally needed to perform that assessment.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **wkt** -- WKT string (must not be NULL)
- **options** -- null-terminated list of options, or NULL. Currently supported options are:
  - STRICT=YES/NO. Defaults to NO. When set to YES, strict validation will be enabled.
  - UNSET_IDENTIFIERS_IF_INCOMPATIBLE_DEF=YES/NO. Defaults to YES. When set to YES, object identifiers are unset when there is a contradiction between the definition from WKT and the one from the database./
  -
- **out_warnings** -- Pointer to a PROJ_STRING_LIST object, or NULL. If provided, *out_warnings will contain a list of warnings, typically for non recognized projection method or parameters, or other issues found during WKT analys. It must be freed with *proj_string_list_destroy()*.
- **out_grammar_errors** -- Pointer to a PROJ_STRING_LIST object, or NULL. If provided, *out_grammar_errors will contain a list of errors regarding the WKT grammar. It must be freed with *proj_string_list_destroy()*.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_from_database***(*PJ_CONTEXT* *ctx, const char *auth_name, const char *code, *PJ_CATEGORY* category, int usePROJAlternativeGridNames, const char *const *options)

Instantiate an object from a database lookup.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **auth_name** -- Authority name (must not be NULL)
- **code** -- Object code (must not be NULL)
- **category** -- Object category
- **usePROJAlternativeGridNames** -- Whether PROJ alternative grid names should be substituted to the official grid names. Only used on transformations
- **options** -- should be set to NULL for now

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

int ***proj_uom_get_info_from_database***(*PJ_CONTEXT* *ctx, const char *auth_name, const char *code, const char **out_name, double *out_conv_factor, const char **out_category)

Get information for a unit of measure from a database lookup.

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **auth_name** -- Authority name (must not be NULL)
- **code** -- Unit of measure code (must not be NULL)
- **out_name** -- Pointer to a string value to store the parameter name. or NULL. This value remains valid until the next call to *proj_uom_get_info_from_database()* or the context destruction.
- **out_conv_factor** -- Pointer to a value to store the conversion factor of the prime meridian longitude unit to radian. or NULL
- **out_category** -- Pointer to a string value to store the parameter name. or NULL. This value might be "unknown", "none", "linear", "linear_per_time", "angular", "angular_per_time", "scale", "scale_per_time", "time", "parametric" or "parametric_per_time"

**Returns**

TRUE in case of success

int ***proj_grid_get_info_from_database***(*PJ_CONTEXT* *ctx, const char *grid_name, const char **out_full_name, const char **out_package_name, const char **out_url, int *out_direct_download, int *out_open_license, int *out_available)

Get information for a grid from a database lookup.

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **grid_name** -- Grid name (must not be NULL)

- **out_full_name** -- Pointer to a string value to store the grid full filename. or NULL
- **out_package_name** -- Pointer to a string value to store the package name where the grid might be found. or NULL
- **out_url** -- Pointer to a string value to store the grid URL or the package URL where the grid might be found. or NULL
- **out_direct_download** -- Pointer to a int (boolean) value to store whether *out_url can be downloaded directly. or NULL
- **out_open_license** -- Pointer to a int (boolean) value to store whether the grid is released with an open license. or NULL
- **out_available** -- Pointer to a int (boolean) value to store whether the grid is available at runtime. or NULL

**Returns**
TRUE in case of success.

*PJ* ***proj_clone**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

"Clone" an object.

The object might be used independently of the original object, provided that the use of context is compatible. In particular if you intend to use a clone in a different thread than the original object, you should pass a context that is different from the one of the original object (or later assign a different context with proj_assign_context()).

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object to clone. Must not be NULL.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

PJ_OBJ_LIST ***proj_create_from_name**(*PJ_CONTEXT* *ctx, const char *auth_name, const char *searchedName, const *PJ_TYPE* *types, size_t typesCount, int approximateMatch, size_t limitResultCount, const char *const *options)

Return a list of objects by their name.

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **auth_name** -- Authority name, used to restrict the search. Or NULL for all authorities.
- **searchedName** -- Searched name. Must be at least 2 character long.
- **types** -- List of object types into which to search. If NULL, all object types will be searched.
- **typesCount** -- Number of elements in types, or 0 if types is NULL
- **approximateMatch** -- Whether approximate name identification is allowed.
- **limitResultCount** -- Maximum number of results to return. Or 0 for unlimited.
- **options** -- should be set to NULL for now

**Returns**
a result set that must be unreferenced with *proj_list_destroy()*, or NULL in case of error.

*PJ_TYPE* **proj_get_type**(const *PJ* *obj)

Return the type of an object.

**Parameters**

- **obj** -- Object (must not be NULL)

**Returns**

its type.

int **proj_is_deprecated**(const *PJ* *obj)

Return whether an object is deprecated.

**Parameters**

- **obj** -- Object (must not be NULL)

**Returns**

TRUE if it is deprecated, FALSE otherwise

PJ_OBJ_LIST ***proj_get_non_deprecated**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Return a list of non-deprecated objects related to the passed one.

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **obj** -- Object (of type CRS for now) for which non-deprecated objects must be searched. Must not be NULL

**Returns**

a result set that must be unreferenced with *proj_list_destroy()*, or NULL in case of error.

int **proj_is_equivalent_to**(const *PJ* *obj, const *PJ* *other, *PJ_COMPARISON_CRITERION* criterion)

Return whether two objects are equivalent.

Use *proj_is_equivalent_to_with_ctx()* to be able to use database information.

**Parameters**

- **obj** -- Object (must not be NULL)
- **other** -- Other object (must not be NULL)
- **criterion** -- Comparison criterion

**Returns**

TRUE if they are equivalent

int **proj_is_equivalent_to_with_ctx**(*PJ_CONTEXT* *ctx, const *PJ* *obj, const *PJ* *other, *PJ_COMPARISON_CRITERION* criterion)

Return whether two objects are equivalent.

Possibly using database to check for name aliases.

**Since**

6.3

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object (must not be NULL)

- **other** -- Other object (must not be NULL)
- **criterion** -- Comparison criterion

**Returns**
TRUE if they are equivalent

int **proj_is_crs**(const *PJ* *obj)

Return whether an object is a CRS.

**Parameters**

- **obj** -- Object (must not be NULL)

const char ***proj_get_name**(const *PJ* *obj)

Get the name of an object.

The lifetime of the returned string is the same as the input obj parameter.

**Parameters**

- **obj** -- Object (must not be NULL)

**Returns**
a string, or NULL in case of error or missing name.

const char ***proj_get_id_auth_name**(const *PJ* *obj, int index)

Get the authority name / codespace of an identifier of an object.

The lifetime of the returned string is the same as the input obj parameter.

**Parameters**

- **obj** -- Object (must not be NULL)
- **index** -- Index of the identifier. 0 = first identifier

**Returns**
a string, or NULL in case of error or missing name.

const char ***proj_get_id_code**(const *PJ* *obj, int index)

Get the code of an identifier of an object.

The lifetime of the returned string is the same as the input obj parameter.

**Parameters**

- **obj** -- Object (must not be NULL)
- **index** -- Index of the identifier. 0 = first identifier

**Returns**
a string, or NULL in case of error or missing name.

const char ***proj_get_remarks**(const *PJ* *obj)

Get the remarks of an object.

The lifetime of the returned string is the same as the input obj parameter.

**Parameters**

- **obj** -- Object (must not be NULL)

**Returns**
a string, or NULL in case of error.

int **proj_get_domain_count**(const *PJ* *obj)

Get the number of domains/usages for a given object.

Most objects have a single domain/usage, but for some of them, there might be multiple.

**Since**
9.2

**Parameters**

- **obj** -- Object (must not be NULL)

**Returns**
the number of domains, or 0 in case of error.

const char ***proj_get_scope**(const *PJ* *obj)

Get the scope of an object.

In case of multiple usages, this will be the one of first usage.

The lifetime of the returned string is the same as the input obj parameter.

**Parameters**

- **obj** -- Object (must not be NULL)

**Returns**
a string, or NULL in case of error or missing scope.

const char ***proj_get_scope_ex**(const *PJ* *obj, int domainIdx)

Get the scope of an object.

The lifetime of the returned string is the same as the input obj parameter.

**Since**
9.2

**Parameters**

- **obj** -- Object (must not be NULL)
- **domainIdx** -- Index of the domain/usage. In [0,proj_get_domain_count(obj)[

**Returns**
a string, or NULL in case of error or missing scope.

int **proj_get_area_of_use**(*PJ_CONTEXT* *ctx, const *PJ* *obj, double *out_west_lon_degree, double *out_south_lat_degree, double *out_east_lon_degree, double *out_north_lat_degree, const char **out_area_name)

Return the area of use of an object.

In case of multiple usages, this will be the one of first usage.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object (must not be NULL)

- **out_west_lon_degree** -- Pointer to a double to receive the west longitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_south_lat_degree** -- Pointer to a double to receive the south latitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_east_lon_degree** -- Pointer to a double to receive the east longitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_north_lat_degree** -- Pointer to a double to receive the north latitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_area_name** -- Pointer to a string to receive the name of the area of use. Or NULL. *p_area_name is valid while obj is valid itself.

**Returns**
TRUE in case of success, FALSE in case of error or if the area of use is unknown.

int **proj_get_area_of_use_ex**(*PJ_CONTEXT* *ctx, const *PJ* *obj, int domainIdx, double *out_west_lon_degree, double *out_south_lat_degree, double *out_east_lon_degree, double *out_north_lat_degree, const char **out_area_name)

Return the area of use of an object.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object (must not be NULL)
- **domainIdx** -- Index of the domain/usage. In [0,proj_get_domain_count(obj)[
- **out_west_lon_degree** -- Pointer to a double to receive the west longitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_south_lat_degree** -- Pointer to a double to receive the south latitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_east_lon_degree** -- Pointer to a double to receive the east longitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_north_lat_degree** -- Pointer to a double to receive the north latitude (in degrees). Or NULL. If the returned value is -1000, the bounding box is unknown.
- **out_area_name** -- Pointer to a string to receive the name of the area of use. Or NULL. *p_area_name is valid while obj is valid itself.

**Returns**
TRUE in case of success, FALSE in case of error or if the area of use is unknown.

const char *__proj_as_wkt__(*PJ_CONTEXT* *ctx, const *PJ* *obj, *PJ_WKT_TYPE* type, const char *const *options)

Get a WKT representation of an object.

The returned string is valid while the input obj parameter is valid, and until a next call to *proj_as_wkt()* with the same input object.

This function calls *osgeo::proj::io::IWKTExportable::exportToWKT()*.

This function may return NULL if the object is not compatible with an export to the requested type.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object (must not be NULL)

- **type** -- WKT version.
- **options** -- null-terminated list of options, or NULL. Currently supported options are:
  - MULTILINE=YES/NO. Defaults to YES, except for WKT1_ESRI
  - INDENTATION_WIDTH=number. Defaults to 4 (when multiline output is on).
  - OUTPUT_AXIS=AUTO/YES/NO. In AUTO mode, axis will be output for WKT2 variants, for WKT1_GDAL for ProjectedCRS with easting/northing ordering (otherwise stripped), but not for WKT1_ESRI. Setting to YES will output them unconditionally, and to NO will omit them unconditionally.
  - STRICT=YES/NO. Default is YES. If NO, a Geographic 3D CRS can be for example exported as WKT1_GDAL with 3 axes, whereas this is normally not allowed.
  - ALLOW_ELLIPSOIDAL_HEIGHT_AS_VERTICAL_CRS=YES/NO. Default is NO. If set to YES and type == PJ_WKT1_GDAL, a Geographic 3D CRS or a Projected 3D CRS will be exported as a compound CRS whose vertical part represents an ellipsoidal height (for example for use with LAS 1.4 WKT1).
  - ALLOW_LINUNIT_NODE=YES/NO. Default is YES starting with PROJ 9.1. Only taken into account with type == PJ_WKT1_ESRI on a Geographic 3D CRS.

**Returns**

a string, or NULL in case of error.

const char *__proj_as_proj_string__(*PJ_CONTEXT* *ctx, const *PJ* *obj, *PJ_PROJ_STRING_TYPE* type, const char *const *options)

Get a PROJ string representation of an object.

The returned string is valid while the input obj parameter is valid, and until a next call to *proj_as_proj_string()* with the same input object.

This function calls *osgeo::proj::io::IPROJStringExportable::exportToPROJString()*.

This function may return NULL if the object is not compatible with an export to the requested type.

**Warning**

If a CRS object was not created from a PROJ string, exporting to a PROJ string will in most cases cause a loss of information. This can potentially lead to erroneous transformations.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object (must not be NULL)
- **type** -- PROJ String version.
- **options** -- NULL-terminated list of strings with "KEY=VALUE" format. or NULL. Currently supported options are:
  - USE_APPROX_TMERC=YES to add the +approx flag to +proj=tmerc or +proj=utm.
  - MULTILINE=YES/NO. Defaults to NO
  - INDENTATION_WIDTH=number. Defaults to 2 (when multiline output is on).
  - MAX_LINE_LENGTH=number. Defaults to 80 (when multiline output is on).

**Returns**
a string, or NULL in case of error.

const char ***proj_as_projjson**(*PJ_CONTEXT* *ctx, const *PJ* *obj, const char *const *options)

Get a PROJJSON string representation of an object.

The returned string is valid while the input obj parameter is valid, and until a next call to *proj_as_proj_string()* with the same input object.

This function calls *osgeo::proj::io::IJSONExportable::exportToJSON()*.

This function may return NULL if the object is not compatible with an export to the requested type.

**Since**
6.2

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object (must not be NULL)
- **options** -- NULL-terminated list of strings with "KEY=VALUE" format. or NULL. Currently supported options are:
  - MULTILINE=YES/NO. Defaults to YES
  - INDENTATION_WIDTH=number. Defaults to 2 (when multiline output is on).
  - SCHEMA=string. URL to PROJJSON schema. Can be set to empty string to disable it.

**Returns**
a string, or NULL in case of error.

*PJ* ***proj_get_source_crs**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Return the base CRS of a BoundCRS or a DerivedCRS/ProjectedCRS, or the source CRS of a CoordinateOperation, or the CRS of a CoordinateMetadata.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type BoundCRS or CoordinateOperation (must not be NULL)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error, or missing source CRS.

*PJ* ***proj_get_target_crs**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Return the hub CRS of a BoundCRS or the target CRS of a CoordinateOperation.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type BoundCRS or CoordinateOperation (must not be NULL)

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error, or missing target CRS.

PJ_OBJ_LIST ***proj_identify**(*PJ_CONTEXT* *ctx, const *PJ* *obj, const char *auth_name, const char *const *options, int **out_confidence)

Identify the CRS with reference CRSs.

The candidate CRSs are either hard-coded, or looked in the database when it is available.

Note that the implementation uses a set of heuristics to have a good compromise of successful identifications over execution time. It might miss legitimate matches in some circumstances.

The method returns a list of matching reference CRS, and the percentage (0-100) of confidence in the match. The list is sorted by decreasing confidence.

- 100% means that the name of the reference entry perfectly matches the CRS name, and both are equivalent. In which case a single result is returned. Note: in the case of a GeographicCRS whose axis order is implicit in the input definition (for example ESRI WKT), then axis order is ignored for the purpose of identification. That is the CRS built from GEOGCS["GCS_WGS_1984",DATUM["D_WGS_1984",SPHEROID["WGS_1984",6378137.0,298.257223563]], PRIMEM["Greenwich",0.0],UNIT["Degree",0.0174532925199433]] will be identified to EPSG:4326, but will not pass a isEquivalentTo(EPSG_4326, util::IComparable::Criterion::EQUIVALENT) test, but rather isEquivalentTo(EPSG_4326, util::IComparable::Criterion::EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS)
- 90% means that CRS are equivalent, but the names are not exactly the same.
- 70% means that CRS are equivalent, but the names are not equivalent.
- 25% means that the CRS are not equivalent, but there is some similarity in the names.

Other confidence values may be returned by some specialized implementations.

This is implemented for GeodeticCRS, ProjectedCRS, VerticalCRS and CompoundCRS.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS. Must not be NULL
- **auth_name** -- Authority name, or NULL for all authorities
- **options** -- Placeholder for future options. Should be set to NULL.
- **out_confidence** -- Output parameter. Pointer to an array of integers that will be allocated by the function and filled with the confidence values (0-100). There are as many elements in this array as *proj_list_get_count()* returns on the return value of this function. *confidence should be released with *proj_int_list_destroy()*.

**Returns**

a list of matching reference CRS, or nullptr in case of error.

*PROJ_STRING_LIST* **proj_get_geoid_models_from_database**(*PJ_CONTEXT* *ctx, const char *auth_name, const char *code, const char *const *options)

Returns a list of geoid models available for that crs.

The list includes the geoid models connected directly with the crs, or via "Height Depth Reversal" or "Change of Vertical Unit" transformations. The returned list is NULL terminated and must be freed with *proj_string_list_destroy()*.

**Since**
8.1

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **auth_name** -- Authority name (must not be NULL)
- **code** -- Object code (must not be NULL)
- **options** -- should be set to NULL for now

**Returns**
list of geoid models names (to be freed with *proj_string_list_destroy()*), or NULL in case of error.

void **proj_int_list_destroy**(int *list)

Free an array of integer.

*PROJ_STRING_LIST* **proj_get_authorities_from_database**(*PJ_CONTEXT* *ctx)

Return the list of authorities used in the database.

The returned list is NULL terminated and must be freed with *proj_string_list_destroy()*.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context

**Returns**
a NULL terminated list of NUL-terminated strings that must be freed with *proj_string_list_destroy()*, or NULL in case of error.

*PROJ_STRING_LIST* **proj_get_codes_from_database**(*PJ_CONTEXT* *ctx, const char *auth_name, *PJ_TYPE* type, int allow_deprecated)

Returns the set of authority codes of the given object type.

The returned list is NULL terminated and must be freed with *proj_string_list_destroy()*.

**See also**

*proj_get_crs_info_list_from_database()*

**Parameters**

- **ctx** -- PROJ context, or NULL for default context.
- **auth_name** -- Authority name (must not be NULL)
- **type** -- Object type.
- **allow_deprecated** -- whether we should return deprecated objects as well.

**Returns**
a NULL terminated list of NUL-terminated strings that must be freed with *proj_string_list_destroy()*, or NULL in case of error.

*PROJ_CELESTIAL_BODY_INFO* **proj_get_celestial_body_list_from_database(*PJ_CONTEXT* *ctx, const char *auth_name, int *out_result_count)

Enumerate celestial bodies from the database.

The returned object is an array of PROJ_CELESTIAL_BODY_INFO* pointers, whose last entry is NULL. This array should be freed with *proj_celestial_body_list_destroy()*

**Since**
8.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **auth_name** -- Authority name, used to restrict the search. Or NULL for all authorities.
- **out_result_count** -- Output parameter pointing to an integer to receive the size of the result list. Might be NULL

**Returns**
an array of PROJ_CELESTIAL_BODY_INFO* pointers to be freed with *proj_celestial_body_list_destroy()*, or NULL in case of error.

void **proj_celestial_body_list_destroy**(*PROJ_CELESTIAL_BODY_INFO* **list)

Destroy the result returned by *proj_get_celestial_body_list_from_database()*.

**Since**
8.1

*PROJ_CRS_LIST_PARAMETERS* ***proj_get_crs_list_parameters_create**(void)

Instantiate a default set of parameters to be used by proj_get_crs_list().

**Returns**
a new object to free with *proj_get_crs_list_parameters_destroy()*

void **proj_get_crs_list_parameters_destroy**(*PROJ_CRS_LIST_PARAMETERS* *params)

Destroy an object returned by *proj_get_crs_list_parameters_create()*

*PROJ_CRS_INFO* **proj_get_crs_info_list_from_database(*PJ_CONTEXT* *ctx, const char *auth_name, const *PROJ_CRS_LIST_PARAMETERS* *params, int *out_result_count)

Enumerate CRS objects from the database, taking into account various criteria.

The returned object is an array of PROJ_CRS_INFO* pointers, whose last entry is NULL. This array should be freed with *proj_crs_info_list_destroy()*

When no filter parameters are set, this is functionally equivalent to *proj_get_codes_from_database()*, instantiating a PJ* object for each of the codes with *proj_create_from_database()* and retrieving information with the various getters. However this function will be much faster.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **auth_name** -- Authority name, used to restrict the search. Or NULL for all authorities.

- **params** -- Additional criteria, or NULL. If not-NULL, params SHOULD have been allocated by *proj_get_crs_list_parameters_create()*, as the *PROJ_CRS_LIST_PARAMETERS* structure might grow over time.
- **out_result_count** -- Output parameter pointing to an integer to receive the size of the result list. Might be NULL

**Returns**
an array of PROJ_CRS_INFO* pointers to be freed with *proj_crs_info_list_destroy()*, or NULL in case of error.

void **proj_crs_info_list_destroy**(*PROJ_CRS_INFO* **list)

Destroy the result returned by *proj_get_crs_info_list_from_database()*.

*PROJ_UNIT_INFO* ***proj_get_units_from_database**(*PJ_CONTEXT* *ctx, const char *auth_name, const char *category, int allow_deprecated, int *out_result_count)

Enumerate units from the database, taking into account various criteria.

The returned object is an array of PROJ_UNIT_INFO* pointers, whose last entry is NULL. This array should be freed with *proj_unit_list_destroy()*

**Since**
7.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **auth_name** -- Authority name, used to restrict the search. Or NULL for all authorities.
- **category** -- Filter by category, if this parameter is not NULL. Category is one of "linear", "linear_per_time", "angular", "angular_per_time", "scale", "scale_per_time" or "time"
- **allow_deprecated** -- whether we should return deprecated objects as well.
- **out_result_count** -- Output parameter pointing to an integer to receive the size of the result list. Might be NULL

**Returns**
an array of PROJ_UNIT_INFO* pointers to be freed with *proj_unit_list_destroy()*, or NULL in case of error.

void **proj_unit_list_destroy**(*PROJ_UNIT_INFO* **list)

Destroy the result returned by *proj_get_units_from_database()*.

**Since**
7.1

PJ_INSERT_SESSION ***proj_insert_object_session_create**(*PJ_CONTEXT* *ctx)

Starts a session for *proj_get_insert_statements()*

Starts a new session for one or several calls to *proj_get_insert_statements()*.

An insertion session guarantees that the inserted objects will not create conflicting intermediate objects.

The session must be stopped with *proj_insert_object_session_destroy()*.

Only one session may be active at a time for a given context.

**Since**
8.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context

**Returns**
the session, or NULL in case of error.

void **proj_insert_object_session_destroy**(*PJ_CONTEXT* *ctx, PJ_INSERT_SESSION *session)

Stops an insertion session started with *proj_insert_object_session_create()*

**Since**
8.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **session** -- The insertion session.

*PROJ_STRING_LIST* **proj_get_insert_statements**(*PJ_CONTEXT* *ctx, PJ_INSERT_SESSION *session, const *PJ* *object, const char *authority, const char *code, int numeric_codes, const char *const *allowed_authorities, const char *const *options)

Returns SQL statements needed to insert the passed object into the database.

*proj_insert_object_session_create()* may have been called previously.

It is strongly recommended that new objects should not be added in common registries, such as "EPSG", "ESRI", "IAU", etc. Users should use a custom authority name instead. If a new object should be added to the official EPSG registry, users are invited to follow the procedure explained at https://epsg.org/dataset-change-requests.html.

Combined with *proj_context_get_database_structure()*, users can create auxiliary databases, instead of directly modifying the main proj.db database. Those auxiliary databases can be specified through *proj_context_set_database_path()* or the PROJ_AUX_DB environment variable.

**Since**
8.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **session** -- The insertion session. May be NULL if a single object must be inserted.
- **object** -- The object to insert into the database. Currently only PrimeMeridian, Ellipsoid, Datum, GeodeticCRS, ProjectedCRS, VerticalCRS, CompoundCRS or BoundCRS are supported.
- **authority** -- Authority name into which the object will be inserted. Must not be NULL.
- **code** -- Code with which the object will be inserted.Must not be NULL.
- **numeric_codes** -- Whether intermediate objects that can be created should use numeric codes (true), or may be alphanumeric (false)

- **allowed_authorities** -- NULL terminated list of authority names, or NULL. Authorities to which intermediate objects are allowed to refer to. "authority" will be implicitly added to it. Note that unit, coordinate systems, projection methods and parameters will in any case be allowed to refer to EPSG. If NULL, allowed_authorities defaults to {"EPSG", "PROJ", nullptr}
- **options** -- NULL terminated list of options, or NULL. No options are supported currently.

**Returns**
a list of insert statements (to be freed with *proj_string_list_destroy()*), or NULL in case of error.

char ***proj_suggests_code_for**(*PJ_CONTEXT* *ctx, const *PJ* *object, const char *authority, int numeric_code, const char *const *options)

Suggests a database code for the passed object.

Supported type of objects are PrimeMeridian, Ellipsoid, Datum, DatumEnsemble, GeodeticCRS, ProjectedCRS, VerticalCRS, CompoundCRS, BoundCRS, Conversion.

**Since**
8.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **object** -- Object for which to suggest a code.
- **authority** -- Authority name into which the object will be inserted.
- **numeric_code** -- Whether the code should be numeric, or derived from the object name.
- **options** -- NULL terminated list of options, or NULL. No options are supported currently.

**Returns**
the suggested code, that is guaranteed to not conflict with an existing one (to be freed with proj_string_destroy), or nullptr in case of error.

void **proj_string_destroy**(char *str)

Free a string.

Only to be used with functions that document using this function.

**Since**
8.1

**Parameters**

- **str** -- String to free.

PJ_OPERATION_FACTORY_CONTEXT ***proj_create_operation_factory_context**(*PJ_CONTEXT* *ctx, const char *authority)

Instantiate a context for building coordinate operations between two CRS.

The returned object must be unreferenced with *proj_operation_factory_context_destroy()* after use.

If authority is NULL or the empty string, then coordinate operations from any authority will be searched, with the restrictions set in the authority_to_authority_preference database table. If authority is set to "any", then

coordinate operations from any authority will be searched If authority is a non-empty string different of "any", then coordinate operations will be searched only in that authority namespace.

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **authority** -- Name of authority to which to restrict the search of candidate operations.

**Returns**

Object that must be unreferenced with *proj_operation_factory_context_destroy()*, or NULL in case of error.

void **proj_operation_factory_context_destroy**(PJ_OPERATION_FACTORY_CONTEXT *ctx)

Drops a reference on an object.

This method should be called one and exactly one for each function returning a PJ_OPERATION_FACTORY_CONTEXT*

**Parameters**

- **ctx** -- Object, or NULL.

void **proj_operation_factory_context_set_desired_accuracy**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, double accuracy)

Set the desired accuracy of the resulting coordinate transformations.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **accuracy** -- Accuracy in meter (or 0 to disable the filter).

void **proj_operation_factory_context_set_area_of_interest**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, double west_lon_degree, double south_lat_degree, double east_lon_degree, double north_lat_degree)

Set the desired area of interest for the resulting coordinate transformations.

For an area of interest crossing the anti-meridian, west_lon_degree will be greater than east_lon_degree.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **west_lon_degree** -- West longitude (in degrees).
- **south_lat_degree** -- South latitude (in degrees).
- **east_lon_degree** -- East longitude (in degrees).
- **north_lat_degree** -- North latitude (in degrees).

void **proj_operation_factory_context_set_area_of_interest_name**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, const char *area_name)

Set the name of the desired area of interest for the resulting coordinate transformations.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **area_name** -- Area name. Must be known of the database.

void **proj_operation_factory_context_set_crs_extent_use**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, *PROJ_CRS_EXTENT_USE* use)

Set how source and target CRS extent should be used when considering if a transformation can be used (only takes effect if no area of interest is explicitly defined).

The default is PJ_CRS_EXTENT_SMALLEST.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **use** -- How source and target CRS extent should be used.

void **proj_operation_factory_context_set_spatial_criterion**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, *PROJ_SPATIAL_CRITERION* criterion)

Set the spatial criterion to use when comparing the area of validity of coordinate operations with the area of interest / area of validity of source and target CRS.

The default is PROJ_SPATIAL_CRITERION_STRICT_CONTAINMENT.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **criterion** -- spatial criterion to use

void **proj_operation_factory_context_set_grid_availability_use**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, *PROJ_GRID_AVAILABILITY_USE* use)

Set how grid availability is used.

The default is USE_FOR_SORTING.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **use** -- how grid availability is used.

void **proj_operation_factory_context_set_use_proj_alternative_grid_names**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, int usePROJNames)

Set whether PROJ alternative grid names should be substituted to the official authority names.

The default is true.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **usePROJNames** -- whether PROJ alternative grid names should be used

void **proj_operation_factory_context_set_allow_use_intermediate_crs**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, *PROJ_INTERMEDIATE_CRS_USE* use)

Set whether an intermediate pivot CRS can be used for researching coordinate operations between a source and target CRS.

Concretely if in the database there is an operation from A to C (or C to A), and another one from C to B (or B to C), but no direct operation between A and B, setting this parameter to true, allow chaining both operations.

The current implementation is limited to researching one intermediate step.

By default, with the IF_NO_DIRECT_TRANSFORMATION strategy, all potential C candidates will be used if there is no direct transformation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **use** -- whether and how intermediate CRS may be used.

void **proj_operation_factory_context_set_allowed_intermediate_crs**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, const char *const *list_of_auth_name_codes)

Restrict the potential pivot CRSs that can be used when trying to build a coordinate operation between two CRS that have no direct operation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **list_of_auth_name_codes** -- an array of strings NLL terminated, with the format { "auth_name1", "code1", "auth_name2", "code2", ... NULL }

void **proj_operation_factory_context_set_discard_superseded**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, int discard)

Set whether transformations that are superseded (but not deprecated) should be discarded.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL

- **discard** -- superseded crs or not

void **proj_operation_factory_context_set_allow_ballpark_transformations**(*PJ_CONTEXT* *ctx, PJ_OPERATION_FACTORY_CONTEXT *factory_ctx, int allow)

Set whether ballpark transformations are allowed.

**Since**
7.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **factory_ctx** -- Operation factory context. must not be NULL
- **allow** -- set to TRUE to allow ballpark transformations.

PJ_OBJ_LIST ***proj_create_operations**(*PJ_CONTEXT* *ctx, const *PJ* *source_crs, const *PJ* *target_crs, const PJ_OPERATION_FACTORY_CONTEXT *operationContext)

Find a list of CoordinateOperation from source_crs to target_crs.

The operations are sorted with the most relevant ones first: by descending area (intersection of the transformation area with the area of interest, or intersection of the transformation with the area of use of the CRS), and by increasing accuracy. Operations with unknown accuracy are sorted last, whatever their area.

Starting with PROJ 9.1, vertical transformations are only done if both source CRS and target CRS are 3D CRS or Compound CRS with a vertical component. You may need to use *proj_crs_promote_to_3D()*.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **source_crs** -- source CRS. Must not be NULL.
- **target_crs** -- source CRS. Must not be NULL.
- **operationContext** -- Search context. Must not be NULL.

**Returns**
a result set that must be unreferenced with *proj_list_destroy()*, or NULL in case of error.

int **proj_list_get_count**(const PJ_OBJ_LIST *result)

Return the number of objects in the result set.

**Parameters**

- **result** -- Object of type PJ_OBJ_LIST (must not be NULL)

*PJ* ***proj_list_get**(*PJ_CONTEXT* *ctx, const PJ_OBJ_LIST *result, int index)

Return an object from the result set.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **result** -- Object of type PJ_OBJ_LIST (must not be NULL)
- **index** -- Index

**Returns**
a new object that must be unreferenced with proj_destroy(), or nullptr in case of error.

void **proj_list_destroy**(PJ_OBJ_LIST *result)

Drops a reference on the result set.

This method should be called one and exactly one for each function returning a PJ_OBJ_LIST*

**Parameters**

- **result** -- Object, or NULL.

int **proj_get_suggested_operation**(*PJ_CONTEXT* *ctx, PJ_OBJ_LIST *operations, *PJ_DIRECTION* direction, *PJ_COORD* coord)

Return the index of the operation that would be the most appropriate to transform the specified coordinates.

This operation may use resources that are not locally available, depending on the search criteria used by *proj_create_operations()*.

This could be done by using *proj_create_operations()* with a punctual bounding box, but this function is faster when one needs to evaluate on many points with the same (source_crs, target_crs) tuple.

**Since**
7.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **operations** -- List of operations returned by *proj_create_operations()*
- **direction** -- Direction into which to transform the point.
- **coord** -- Coordinate to transform

**Returns**
the index in operations that would be used to transform coord. Or -1 in case of error, or no match.

int **proj_crs_is_derived**(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Returns whether a CRS is a derived CRS.

**Since**
8.0

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type CRS (must not be NULL)

**Returns**
TRUE if the CRS is a derived CRS.

*PJ* ***proj_crs_get_geodetic_crs**(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Get the geodeticCRS / geographicCRS from a CRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type CRS (must not be NULL)

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_get_horizontal_datum**(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Get the horizontal datum from a CRS.

This function may return a Datum or DatumEnsemble object.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type CRS (must not be NULL)

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_get_sub_crs**(*PJ_CONTEXT* *ctx, const *PJ* *crs, int index)

Get a CRS component from a CompoundCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type CRS (must not be NULL)
- **index** -- Index of the CRS component (typically 0 = horizontal, 1 = vertical)

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_get_datum**(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Returns the datum of a SingleCRS.

If that function returns NULL,

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**See also**

*proj_crs_get_datum_ensemble()* to potentially get a DatumEnsemble instead.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type SingleCRS (must not be NULL)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error (or if there is no datum)

*PJ* ***proj_crs_get_datum_ensemble**(*PJ_CONTEXT* *ctx, const *PJ* *crs)*

Returns the datum ensemble of a SingleCRS.

If that function returns NULL,

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**See also**

*proj_crs_get_datum()* to potentially get a Datum instead.

**Since**
7.2

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type SingleCRS (must not be NULL)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error (or if there is no datum ensemble)

*PJ* ***proj_crs_get_datum_forced**(*PJ_CONTEXT* *ctx, const *PJ* *crs)*

Returns a datum for a SingleCRS.

If the SingleCRS has a datum, then this datum is returned. Otherwise, the SingleCRS has a datum ensemble, and this datum ensemble is returned as a regular datum instead of a datum ensemble.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
7.2

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type SingleCRS (must not be NULL)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error (or if there is no datum)

int **proj_crs_has_point_motion_operation**(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Return whether a CRS has an associated PointMotionOperation.

**Since**
9.4

int **proj_datum_ensemble_get_member_count**(*PJ_CONTEXT* *ctx, const *PJ* *datum_ensemble)

Returns the number of members of a datum ensemble.

**Since**
7.2

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **datum_ensemble** -- Object of type DatumEnsemble (must not be NULL)

double **proj_datum_ensemble_get_accuracy**(*PJ_CONTEXT* *ctx, const *PJ* *datum_ensemble)

Returns the positional accuracy of the datum ensemble.

**Since**
7.2

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **datum_ensemble** -- Object of type DatumEnsemble (must not be NULL)

**Returns**
the accuracy, or -1 in case of error.

*PJ* ***proj_datum_ensemble_get_member**(*PJ_CONTEXT* *ctx, const *PJ* *datum_ensemble, int member_index)

Returns a member from a datum ensemble.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
7.2

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **datum_ensemble** -- Object of type DatumEnsemble (must not be NULL)
- **member_index** -- Index of the datum member to extract (between 0 and *proj_datum_ensemble_get_member_count()*-1)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error (or if there is no datum ensemble)

double **proj_dynamic_datum_get_frame_reference_epoch**(*PJ_CONTEXT* *ctx, const *PJ* *datum)

Returns the frame reference epoch of a dynamic geodetic or vertical reference frame.

**Since**

7.2

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **datum** -- Object of type DynamicGeodeticReferenceFrame or DynamicVerticalReferenceFrame (must not be NULL)

**Returns**

the frame reference epoch as decimal year, or -1 in case of error.

*PJ* ***proj_crs_get_coordinate_system**(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Returns the coordinate system of a SingleCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type SingleCRS (must not be NULL)

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ_COORDINATE_SYSTEM_TYPE* **proj_cs_get_type**(*PJ_CONTEXT* *ctx, const *PJ* *cs)

Returns the type of the coordinate system.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **cs** -- Object of type CoordinateSystem (must not be NULL)

**Returns**

type, or PJ_CS_TYPE_UNKNOWN in case of error.

int **proj_cs_get_axis_count**(*PJ_CONTEXT* *ctx, const *PJ* *cs)

Returns the number of axis of the coordinate system.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **cs** -- Object of type CoordinateSystem (must not be NULL)

**Returns**

number of axis, or -1 in case of error.

int **proj_cs_get_axis_info**(*PJ_CONTEXT* *ctx, const *PJ* *cs, int index, const char **out_name, const char **out_abbrev, const char **out_direction, double *out_unit_conv_factor, const char **out_unit_name, const char **out_unit_auth_name, const char **out_unit_code)

Returns information on an axis.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **cs** -- Object of type CoordinateSystem (must not be NULL)
- **index** -- Index of the coordinate system (between 0 and *proj_cs_get_axis_count()* - 1)
- **out_name** -- Pointer to a string value to store the axis name. or NULL
- **out_abbrev** -- Pointer to a string value to store the axis abbreviation. or NULL
- **out_direction** -- Pointer to a string value to store the axis direction. or NULL
- **out_unit_conv_factor** -- Pointer to a double value to store the axis unit conversion factor. or NULL
- **out_unit_name** -- Pointer to a string value to store the axis unit name. or NULL
- **out_unit_auth_name** -- Pointer to a string value to store the axis unit authority name. or NULL
- **out_unit_code** -- Pointer to a string value to store the axis unit code. or NULL

**Returns**
TRUE in case of success

*PJ* ***proj_get_ellipsoid**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Get the ellipsoid from a CRS or a GeodeticReferenceFrame.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS or GeodeticReferenceFrame (must not be NULL)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

int **proj_ellipsoid_get_parameters**(*PJ_CONTEXT* *ctx, const *PJ* *ellipsoid, double *out_semi_major_metre, double *out_semi_minor_metre, int *out_is_semi_minor_computed, double *out_inv_flattening)

Return ellipsoid parameters.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **ellipsoid** -- Object of type Ellipsoid (must not be NULL)
- **out_semi_major_metre** -- Pointer to a value to store the semi-major axis in metre. or NULL
- **out_semi_minor_metre** -- Pointer to a value to store the semi-minor axis in metre. or NULL
- **out_is_semi_minor_computed** -- Pointer to a boolean value to indicate if the semi-minor value was computed. If FALSE, its value comes from the definition. or NULL
- **out_inv_flattening** -- Pointer to a value to store the inverse flattening. or NULL

**Returns**
TRUE in case of success.

const char *proj_get_celestial_body_name(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Get the name of the celestial body of this object.

Object should be a CRS, Datum or Ellipsoid.

**Since**

8.1

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS, Datum or Ellipsoid.(must not be NULL)

**Returns**

the name of the celestial body, or NULL.

*PJ* *proj_get_prime_meridian(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Get the prime meridian of a CRS or a GeodeticReferenceFrame.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS or GeodeticReferenceFrame (must not be NULL)

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

int proj_prime_meridian_get_parameters(*PJ_CONTEXT* *ctx, const *PJ* *prime_meridian, double *out_longitude, double *out_unit_conv_factor, const char **out_unit_name)

Return prime meridian parameters.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **prime_meridian** -- Object of type PrimeMeridian (must not be NULL)
- **out_longitude** -- Pointer to a value to store the longitude of the prime meridian, in its native unit. or NULL
- **out_unit_conv_factor** -- Pointer to a value to store the conversion factor of the prime meridian longitude unit to radian. or NULL
- **out_unit_name** -- Pointer to a string value to store the unit name. or NULL

**Returns**

TRUE in case of success.

*PJ* *proj_crs_get_coordoperation(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Return the Conversion of a DerivedCRS (such as a ProjectedCRS), or the Transformation from the baseCRS to the hubCRS of a BoundCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- Object of type DerivedCRS or BoundCRSs (must not be NULL)

**Returns**
Object of type SingleOperation that must be unreferenced with proj_destroy(), or NULL in case of error.

int **proj_coordoperation_get_method_info**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation, const char **out_method_name, const char **out_method_auth_name, const char **out_method_code)

Return information on the operation method of the SingleOperation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type SingleOperation (typically a Conversion or Transformation) (must not be NULL)
- **out_method_name** -- Pointer to a string value to store the method (projection) name. or NULL
- **out_method_auth_name** -- Pointer to a string value to store the method authority name. or NULL
- **out_method_code** -- Pointer to a string value to store the method code. or NULL

**Returns**
TRUE in case of success.

int **proj_coordoperation_is_instantiable**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation)

Return whether a coordinate operation can be instantiated as a PROJ pipeline, checking in particular that referenced grids are available.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type CoordinateOperation or derived classes (must not be NULL)

**Returns**
TRUE or FALSE.

int **proj_coordoperation_has_ballpark_transformation**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation)

Return whether a coordinate operation has a "ballpark" transformation, that is a very approximate one, due to lack of more accurate transformations.

Typically a null geographic offset between two horizontal datum, or a null vertical offset (or limited to unit changes) between two vertical datum. Errors of several tens to one hundred meters might be expected, compared to more accurate transformations.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type CoordinateOperation or derived classes (must not be NULL)

**Returns**
TRUE or FALSE.

int **proj_coordoperation_requires_per_coordinate_input_time**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation)

Return whether a coordinate operation requires coordinate tuples to have a valid input time for the coordinate transformation to succeed. (this applies for the forward direction)

Note: in the case of a time-dependent Helmert transformation, this function will return true, but when executing proj_trans(), execution will still succeed if the time information is missing, due to the transformation central epoch being used as a fallback.

**Since**
9.5

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type CoordinateOperation or derived classes (must not be NULL)

**Returns**
TRUE or FALSE.

int **proj_coordoperation_get_param_count**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation)

Return the number of parameters of a SingleOperation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type SingleOperation or derived classes (must not be NULL)

int **proj_coordoperation_get_param_index**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation, const char *name)

Return the index of a parameter of a SingleOperation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type SingleOperation or derived classes (must not be NULL)
- **name** -- Parameter name. Must not be NULL

**Returns**
index (>=0), or -1 in case of error.

int **proj_coordoperation_get_param**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation, int index, const char **out_name, const char **out_auth_name, const char **out_code, double *out_value, const char **out_value_string, double *out_unit_conv_factor, const char **out_unit_name, const char **out_unit_auth_name, const char **out_unit_code, const char **out_unit_category)

Return a parameter of a SingleOperation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type SingleOperation or derived classes (must not be NULL)
- **index** -- Parameter index.
- **out_name** -- Pointer to a string value to store the parameter name. or NULL

- **out_auth_name** -- Pointer to a string value to store the parameter authority name. or NULL
- **out_code** -- Pointer to a string value to store the parameter code. or NULL
- **out_value** -- Pointer to a double value to store the parameter value (if numeric). or NULL
- **out_value_string** -- Pointer to a string value to store the parameter value (if of type string). or NULL
- **out_unit_conv_factor** -- Pointer to a double value to store the parameter unit conversion factor. or NULL
- **out_unit_name** -- Pointer to a string value to store the parameter unit name. or NULL
- **out_unit_auth_name** -- Pointer to a string value to store the unit authority name. or NULL
- **out_unit_code** -- Pointer to a string value to store the unit code. or NULL
- **out_unit_category** -- Pointer to a string value to store the parameter name. or NULL. This value might be "unknown", "none", "linear", "linear_per_time", "angular", "angular_per_time", "scale", "scale_per_time", "time", "parametric" or "parametric_per_time"

**Returns**

TRUE in case of success.

int **proj_coordoperation_get_grid_used_count**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation)

Return the number of grids used by a CoordinateOperation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type CoordinateOperation or derived classes (must not be NULL)

int **proj_coordoperation_get_grid_used**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation, int index, const char **out_short_name, const char **out_full_name, const char **out_package_name, const char **out_url, int *out_direct_download, int *out_open_license, int *out_available)

Return a parameter of a SingleOperation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type SingleOperation or derived classes (must not be NULL)
- **index** -- Parameter index.
- **out_short_name** -- Pointer to a string value to store the grid short name. or NULL
- **out_full_name** -- Pointer to a string value to store the grid full filename. or NULL
- **out_package_name** -- Pointer to a string value to store the package name where the grid might be found. or NULL
- **out_url** -- Pointer to a string value to store the grid URL or the package URL where the grid might be found. or NULL
- **out_direct_download** -- Pointer to a int (boolean) value to store whether *out_url can be downloaded directly. or NULL
- **out_open_license** -- Pointer to a int (boolean) value to store whether the grid is released with an open license. or NULL

- **out_available** -- Pointer to a int (boolean) value to store whether the grid is available at runtime. or NULL

**Returns**
TRUE in case of success.

double **proj_coordoperation_get_accuracy**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Return the accuracy (in metre) of a coordinate operation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Coordinate operation. Must not be NULL.

**Returns**
the accuracy, or a negative value if unknown or in case of error.

int **proj_coordoperation_get_towgs84_values**(*PJ_CONTEXT* *ctx, const *PJ* *coordoperation, double *out_values, int value_count, int emit_error_if_incompatible)

Return the parameters of a Helmert transformation as WKT1 TOWGS84 values.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **coordoperation** -- Object of type Transformation, that can be represented as a WKT1 TOWGS84 node (must not be NULL)
- **out_values** -- Pointer to an array of value_count double values.
- **value_count** -- Size of out_values array. The suggested size is 7 to get translation, rotation and scale difference parameters. Rotation and scale difference terms might be zero if the transformation only includes translation parameters. In that case, value_count could be set to 3.
- **emit_error_if_incompatible** -- Boolean to indicate if an error must be logged if coordoperation is not compatible with a WKT1 TOWGS84 representation.

**Returns**
TRUE in case of success, or FALSE if coordoperation is not compatible with a WKT1 TOWGS84 representation.

*PJ* ***proj_coordoperation_create_inverse**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Returns a PJ* coordinate operation object which represents the inverse operation of the specified coordinate operation.

**Since**
6.3

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CoordinateOperation (must not be NULL)

**Returns**
a new PJ* object to free with proj_destroy() in case of success, or nullptr in case of error

int **proj_concatoperation_get_step_count**(*PJ_CONTEXT* *ctx, const *PJ* *concatoperation)

Returns the number of steps of a concatenated operation.

The input object must be a concatenated operation.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **concatoperation** -- Concatenated operation (must not be NULL)

**Returns**
the number of steps, or 0 in case of error.

*PJ* *__proj_concatoperation_get_step__(*PJ_CONTEXT* *ctx, const *PJ* *concatoperation, int i_step)

Returns a step of a concatenated operation.

The input object must be a concatenated operation.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **concatoperation** -- Concatenated operation (must not be NULL)
- **i_step** -- Index of the step to extract. Between 0 and *proj_concatoperation_get_step_count()*-1

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* *__proj_coordinate_metadata_create__(*PJ_CONTEXT* *ctx, const *PJ* *crs, double epoch)

Instantiate a CoordinateMetadata object.

**Since**
9.4

double **proj_coordinate_metadata_get_epoch**(*PJ_CONTEXT* *ctx, const *PJ* *obj)

Return the coordinate epoch associated with a CoordinateMetadata.

It may return a NaN value if there is no associated coordinate epoch.

**Since**
9.2

### Advanced functions

Available in `proj.h` since PROJ 9.4. Previously were available in `proj_experimental.h`.

*PJ* *__proj_create_cs__(*PJ_CONTEXT* *ctx, *PJ_COORDINATE_SYSTEM_TYPE* type, int axis_count, const *PJ_AXIS_DESCRIPTION* *axis)

Instantiate a CoordinateSystem.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **type** -- Coordinate system type.
- **axis_count** -- Number of axis
- **axis** -- Axis description (array of size axis_count)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_cartesian_2D_cs**(*PJ_CONTEXT* *ctx, *PJ_CARTESIAN_CS_2D_TYPE* type, const char *unit_name, double unit_conv_factor)

Instantiate a CartesiansCS 2D.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **type** -- Coordinate system type.
- **unit_name** -- Unit name.
- **unit_conv_factor** -- Unit conversion factor to SI.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_ellipsoidal_2D_cs**(*PJ_CONTEXT* *ctx, *PJ_ELLIPSOIDAL_CS_2D_TYPE* type, const char *unit_name, double unit_conv_factor)

Instantiate a Ellipsoidal 2D.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **type** -- Coordinate system type.
- **unit_name** -- Name of the angular units. Or NULL for Degree
- **unit_conv_factor** -- Conversion factor from the angular unit to radian. Or 0 for Degree if unit_name == NULL. Otherwise should be not NULL

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_ellipsoidal_3D_cs**(*PJ_CONTEXT* *ctx, *PJ_ELLIPSOIDAL_CS_3D_TYPE* type, const char *horizontal_angular_unit_name, double horizontal_angular_unit_conv_factor, const char *vertical_linear_unit_name, double vertical_linear_unit_conv_factor)

Instantiate a Ellipsoidal 3D.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
6.3

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **type** -- Coordinate system type.
- **horizontal_angular_unit_name** -- Name of the angular units. Or NULL for Degree.
- **horizontal_angular_unit_conv_factor** -- Conversion factor from the angular unit to radian. Or 0 for Degree if horizontal_angular_unit_name == NULL. Otherwise should be not NULL
- **vertical_linear_unit_name** -- Vertical linear unit name. Or NULL for Metre.
- **vertical_linear_unit_conv_factor** -- Vertical linear unit conversion factor to metre. Or 0 for Metre if vertical_linear_unit_name == NULL. Otherwise should be not NULL

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

PJ_OBJ_LIST ***proj_query_geodetic_crs_from_datum**(*PJ_CONTEXT* *ctx, const char *crs_auth_name, const char *datum_auth_name, const char *datum_code, const char *crs_type)

Return GeodeticCRS that use the specified datum.

**Parameters**

- **ctx** -- Context, or NULL for default context.
- **crs_auth_name** -- CRS authority name, or NULL.
- **datum_auth_name** -- Datum authority name (must not be NULL)
- **datum_code** -- Datum code (must not be NULL)
- **crs_type** -- "geographic 2D", "geographic 3D", "geocentric" or NULL

**Returns**

a result set that must be unreferenced with *proj_list_destroy()*, or NULL in case of error.

*PJ* ***proj_create_geographic_crs**(*PJ_CONTEXT* *ctx, const char *crs_name, const char *datum_name, const char *ellps_name, double semi_major_metre, double inv_flattening, const char *prime_meridian_name, double prime_meridian_offset, const char *pm_angular_units, double pm_units_conv, const *PJ* *ellipsoidal_cs)

Create a GeographicCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **datum_name** -- Name of the GeodeticReferenceFrame. Or NULL
- **ellps_name** -- Name of the Ellipsoid. Or NULL
- **semi_major_metre** -- Ellipsoid semi-major axis, in metres.
- **inv_flattening** -- Ellipsoid inverse flattening. Or 0 for a sphere.
- **prime_meridian_name** -- Name of the PrimeMeridian. Or NULL

- **prime_meridian_offset** -- Offset of the prime meridian, expressed in the specified angular units.
- **pm_angular_units** -- Name of the angular units. Or NULL for Degree
- **pm_angular_units_conv** -- Conversion factor from the angular unit to radian. Or 0 for Degree if pm_angular_units == NULL. Otherwise should be not NULL
- **ellipsoidal_cs** -- Coordinate system. Must not be NULL.

**Returns**

Object of type GeographicCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_geographic_crs_from_datum**(*PJ_CONTEXT* *ctx, const char *crs_name, const *PJ* *datum_or_datum_ensemble, const *PJ* *ellipsoidal_cs)

Create a GeographicCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **datum_or_datum_ensemble** -- Datum or DatumEnsemble (DatumEnsemble possible since 7.2). Must not be NULL.
- **ellipsoidal_cs** -- Coordinate system. Must not be NULL.

**Returns**

Object of type GeographicCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_geocentric_crs**(*PJ_CONTEXT* *ctx, const char *crs_name, const char *datum_name, const char *ellps_name, double semi_major_metre, double inv_flattening, const char *prime_meridian_name, double prime_meridian_offset, const char *angular_units, double angular_units_conv, const char *linear_units, double linear_units_conv)

Create a GeodeticCRS of geocentric type.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **datum_name** -- Name of the GeodeticReferenceFrame. Or NULL
- **ellps_name** -- Name of the Ellipsoid. Or NULL
- **semi_major_metre** -- Ellipsoid semi-major axis, in metres.
- **inv_flattening** -- Ellipsoid inverse flattening. Or 0 for a sphere.
- **prime_meridian_name** -- Name of the PrimeMeridian. Or NULL
- **prime_meridian_offset** -- Offset of the prime meridian, expressed in the specified angular units.

- **angular_units** -- Name of the angular units. Or NULL for Degree
- **angular_units_conv** -- Conversion factor from the angular unit to radian. Or 0 for Degree if angular_units == NULL. Otherwise should be not NULL
- **linear_units** -- Name of the linear units. Or NULL for Metre
- **linear_units_conv** -- Conversion factor from the linear unit to metre. Or 0 for Metre if linear_units == NULL. Otherwise should be not NULL

**Returns**
Object of type GeodeticCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_geocentric_crs_from_datum**(*PJ_CONTEXT* *ctx, const char *crs_name, const *PJ* *datum_or_datum_ensemble, const char *linear_units, double linear_units_conv)

Create a GeodeticCRS of geocentric type.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **datum_or_datum_ensemble** -- Datum or DatumEnsemble (DatumEnsemble possible since 7.2). Must not be NULL.
- **linear_units** -- Name of the linear units. Or NULL for Metre
- **linear_units_conv** -- Conversion factor from the linear unit to metre. Or 0 for Metre if linear_units == NULL. Otherwise should be not NULL

**Returns**
Object of type GeodeticCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_derived_geographic_crs**(*PJ_CONTEXT* *ctx, const char *crs_name, const *PJ* *base_geographic_crs, const *PJ* *conversion, const *PJ* *ellipsoidal_cs)

Create a DerivedGeograhicCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **base_geographic_crs** -- Base Geographic CRS. Must not be NULL.
- **conversion** -- Conversion from the base Geographic to the DerivedGeograhicCRS. Must not be NULL.

- **ellipsoidal_cs** -- Coordinate system. Must not be NULL.

**Returns**
Object of type GeodeticCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

int **proj_is_derived_crs**(*PJ_CONTEXT* *ctx, const *PJ* *crs)

Return whether a CRS is a Derived CRS.

**Since**
7.0

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs** -- CRS. Must not be NULL.

**Returns**
whether a CRS is a Derived CRS.

*PJ* ***proj_alter_name**(*PJ_CONTEXT* *ctx, const *PJ* *obj, const char *name)

Return a copy of the object with its name changed.

Currently, only implemented on CRS objects.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS. Must not be NULL
- **name** -- New name. Must not be NULL

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_alter_id**(*PJ_CONTEXT* *ctx, const *PJ* *obj, const char *auth_name, const char *code)

Return a copy of the object with its identifier changed/set.

Currently, only implemented on CRS objects.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS. Must not be NULL
- **auth_name** -- Authority name. Must not be NULL
- **code** -- Code. Must not be NULL

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_alter_geodetic_crs***(*PJ_CONTEXT* *ctx, const *PJ* *obj, const *PJ* *new_geod_crs)

Return a copy of the CRS with its geodetic CRS changed.

Currently, when obj is a GeodeticCRS, it returns a clone of new_geod_crs When obj is a ProjectedCRS, it replaces its base CRS with new_geod_crs. When obj is a CompoundCRS, it replaces the GeodeticCRS part of the horizontal CRS with new_geod_crs. In other cases, it returns a clone of obj.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS. Must not be NULL
- **new_geod_crs** -- Object of type GeodeticCRS. Must not be NULL

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_alter_cs_angular_unit***(*PJ_CONTEXT* *ctx, const *PJ* *obj, const char *angular_units, double angular_units_conv, const char *unit_auth_name, const char *unit_code)

Return a copy of the CRS with its angular units changed.

The CRS must be or contain a GeographicCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS. Must not be NULL
- **angular_units** -- Name of the angular units. Or NULL for Degree
- **angular_units_conv** -- Conversion factor from the angular unit to radian. Or 0 for Degree if angular_units == NULL. Otherwise should be not NULL
- **unit_auth_name** -- Unit authority name. Or NULL.
- **unit_code** -- Unit code. Or NULL.

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_alter_cs_linear_unit***(*PJ_CONTEXT* *ctx, const *PJ* *obj, const char *linear_units, double linear_units_conv, const char *unit_auth_name, const char *unit_code)

Return a copy of the CRS with the linear units of its coordinate system changed.

The CRS must be or contain a ProjectedCRS, VerticalCRS or a GeocentricCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type CRS. Must not be NULL
- **linear_units** -- Name of the linear units. Or NULL for Metre

- **linear_units_conv** -- Conversion factor from the linear unit to metre. Or 0 for Metre if linear_units == NULL. Otherwise should be not NULL
- **unit_auth_name** -- Unit authority name. Or NULL.
- **unit_code** -- Unit code. Or NULL.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_alter_parameters_linear_unit**(*PJ_CONTEXT* *ctx, const *PJ* *obj, const char *linear_units, double linear_units_conv, const char *unit_auth_name, const char *unit_code, int convert_to_new_unit)*

Return a copy of the CRS with the linear units of the parameters of its conversion modified.

The CRS must be or contain a ProjectedCRS, VerticalCRS or a GeocentricCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **obj** -- Object of type ProjectedCRS. Must not be NULL
- **linear_units** -- Name of the linear units. Or NULL for Metre
- **linear_units_conv** -- Conversion factor from the linear unit to metre. Or 0 for Metre if linear_units == NULL. Otherwise should be not NULL
- **unit_auth_name** -- Unit authority name. Or NULL.
- **unit_code** -- Unit code. Or NULL.
- **convert_to_new_unit** -- TRUE if existing values should be converted from their current unit to the new unit. If FALSE, their value will be left unchanged and the unit overridden (so the resulting CRS will not be equivalent to the original one for reprojection purposes).

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_promote_to_3D**(*PJ_CONTEXT* *ctx, const char *crs_3D_name, const *PJ* *crs_2D)*

Create a 3D CRS from an existing 2D CRS.

The new axis will be ellipsoidal height, oriented upwards, and with metre units.

See *osgeo::proj::crs::CRS::promoteTo3D()*.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
6.3

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_3D_name** -- CRS name. Or NULL (in which case the name of crs_2D will be used)
- **crs_2D** -- 2D CRS to be "promoted" to 3D. Must not be NULL.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_create_projected_3D_crs_from_2D**(*PJ_CONTEXT* *ctx, const char *crs_name, const *PJ* *projected_2D_crs, const *PJ* *geog_3D_crs)

Create a projected 3D CRS from an existing projected 2D CRS.

The passed projected_2D_crs is used so that its name is replaced by crs_name and its base geographic CRS is replaced by geog_3D_crs. The vertical axis of geog_3D_crs (ellipsoidal height) will be added as the 3rd axis of the resulting projected 3D CRS. Normally, the passed geog_3D_crs should be the 3D counterpart of the original 2D base geographic CRS of projected_2D_crs, but such no check is done.

It is also possible to invoke this function with a NULL geog_3D_crs. In which case, the existing base geographic 2D CRS of projected_2D_crs will be automatically promoted to 3D by assuming a 3rd axis being an ellipsoidal height, oriented upwards, and with metre units. This is equivalent to using *proj_crs_promote_to_3D()*.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
6.3

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- CRS name. Or NULL (in which case the name of projected_2D_crs will be used)
- **projected_2D_crs** -- Projected 2D CRS to be "promoted" to 3D. Must not be NULL.
- **geog_3D_crs** -- Base geographic 3D CRS for the new CRS. May be NULL.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_demote_to_2D**(*PJ_CONTEXT* *ctx, const char *crs_2D_name, const *PJ* *crs_3D)

Create a 2D CRS from an existing 3D CRS.

See *osgeo::proj::crs::CRS::demoteTo2D()*.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
6.3

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_2D_name** -- CRS name. Or NULL (in which case the name of crs_3D will be used)
- **crs_3D** -- 3D CRS to be "demoted" to 2D. Must not be NULL.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_engineering_crs***(*PJ_CONTEXT* *ctx, const char *crsName)

Instantiate a EngineeringCRS with just a name.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- CRS name. Or NULL.

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_vertical_crs***(*PJ_CONTEXT* *ctx, const char *crs_name, const char *datum_name, const char *linear_units, double linear_units_conv)

Create a VerticalCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **datum_name** -- Name of the VerticalReferenceFrame. Or NULL
- **linear_units** -- Name of the linear units. Or NULL for Metre
- **linear_units_conv** -- Conversion factor from the linear unit to metre. Or 0 for Metre if linear_units == NULL. Otherwise should be not NULL

**Returns**

Object of type VerticalCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_vertical_crs_ex***(*PJ_CONTEXT* *ctx, const char *crs_name, const char *datum_name, const char *datum_auth_name, const char *datum_code, const char *linear_units, double linear_units_conv, const char *geoid_model_name, const char *geoid_model_auth_name, const char *geoid_model_code, const *PJ* *geoid_geog_crs, const char *const *options)

Create a VerticalCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

This is an extended (_ex) version of *proj_create_vertical_crs()* that adds the capability of defining a geoid model.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **datum_name** -- Name of the VerticalReferenceFrame. Or NULL
- **datum_auth_name** -- Authority name of the VerticalReferenceFrame. Or NULL
- **datum_code** -- Code of the VerticalReferenceFrame. Or NULL
- **linear_units** -- Name of the linear units. Or NULL for Metre

- **linear_units_conv** -- Conversion factor from the linear unit to metre. Or 0 for Metre if linear_units == NULL. Otherwise should be not NULL
- **geoid_model_name** -- Geoid model name, or NULL. Can be a name from the geoid_model name or a string "PROJ foo.gtx"
- **geoid_model_auth_name** -- Authority name of the transformation for the geoid model. or NULL
- **geoid_model_code** -- Code of the transformation for the geoid model. or NULL
- **geoid_geog_crs** -- Geographic CRS for the geoid transformation, or NULL.
- **options** -- NULL-terminated list of strings with "KEY=VALUE" format. or NULL. The currently recognized option is ACCURACY=value, where value is in metre.

**Returns**

Object of type VerticalCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_compound_crs**(*PJ_CONTEXT* *ctx, const char *crs_name, const *PJ* *horiz_crs, const *PJ* *vert_crs)

Create a CompoundCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- Name of the GeographicCRS. Or NULL
- **horiz_crs** -- Horizontal CRS. must not be NULL.
- **vert_crs** -- Vertical CRS. must not be NULL.

**Returns**

Object of type CompoundCRS that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_conversion**(*PJ_CONTEXT* *ctx, const char *name, const char *auth_name, const char *code, const char *method_name, const char *method_auth_name, const char *method_code, int param_count, const *PJ_PARAM_DESCRIPTION* *params)

Instantiate a Conversion.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **name** -- Conversion name. Or NULL.
- **auth_name** -- Conversion authority name. Or NULL.
- **code** -- Conversion code. Or NULL.
- **method_name** -- Method name. Or NULL.
- **method_auth_name** -- Method authority name. Or NULL.
- **method_code** -- Method code. Or NULL.
- **param_count** -- Number of parameters (size of params argument)

- **params** -- Parameter descriptions (array of size param_count)

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_transformation**(*PJ_CONTEXT* *ctx, const char *name, const char *auth_name, const char *code, const *PJ* *source_crs, const *PJ* *target_crs, const *PJ* *interpolation_crs, const char *method_name, const char *method_auth_name, const char *method_code, int param_count, const *PJ_PARAM_DESCRIPTION* *params, double accuracy)

Instantiate a Transformation.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **name** -- Transformation name. Or NULL.
- **auth_name** -- Transformation authority name. Or NULL.
- **code** -- Transformation code. Or NULL.
- **source_crs** -- Object of type CRS representing the source CRS. Must not be NULL.
- **target_crs** -- Object of type CRS representing the target CRS. Must not be NULL.
- **interpolation_crs** -- Object of type CRS representing the interpolation CRS. Or NULL.
- **method_name** -- Method name. Or NULL.
- **method_auth_name** -- Method authority name. Or NULL.
- **method_code** -- Method code. Or NULL.
- **param_count** -- Number of parameters (size of params argument)
- **params** -- Parameter descriptions (array of size param_count)
- **accuracy** -- Accuracy of the transformation in meters. A negative values means unknown.

**Returns**

Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_convert_conversion_to_other_method**(*PJ_CONTEXT* *ctx, const *PJ* *conversion, int new_method_epsg_code, const char *new_method_name)

Return an equivalent projection.

Currently implemented:

- EPSG_CODE_METHOD_MERCATOR_VARIANT_A (1SP) to EPSG_CODE_METHOD_MERCATOR_VARIANT_B (2SP)
- EPSG_CODE_METHOD_MERCATOR_VARIANT_B (2SP) to EPSG_CODE_METHOD_MERCATOR_VARIANT_A (1SP)
- EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_1SP to EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_2SP
- EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_2SP to EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_1SP

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **conversion** -- Object of type Conversion. Must not be NULL.
- **new_method_epsg_code** -- EPSG code of the target method. Or 0 (in which case new_method_name must be specified).
- **new_method_name** -- EPSG or PROJ target method name. Or nullptr (in which case new_method_epsg_code must be specified).

**Returns**
new conversion that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_projected_crs**(*PJ_CONTEXT* *ctx, const char *crs_name, const *PJ* *geodetic_crs, const *PJ* *conversion, const *PJ* *coordinate_system)

Instantiate a ProjectedCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **crs_name** -- CRS name. Or NULL
- **geodetic_crs** -- Base GeodeticCRS. Must not be NULL.
- **conversion** -- Conversion. Must not be NULL.
- **coordinate_system** -- Cartesian coordinate system. Must not be NULL.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_create_bound_crs**(*PJ_CONTEXT* *ctx, const *PJ* *base_crs, const *PJ* *hub_crs, const *PJ* *transformation)

Returns a BoundCRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **base_crs** -- Base CRS (must not be NULL)
- **hub_crs** -- Hub CRS (must not be NULL)
- **transformation** -- Transformation (must not be NULL)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_create_bound_crs_to_WGS84**(*PJ_CONTEXT* *ctx, const *PJ* *crs, const char *const *options)

Returns potentially a BoundCRS, with a transformation to EPSG:4326, wrapping this CRS.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

This is the same as method *osgeo::proj::crs::CRS::createBoundCRSToWGS84IfPossible()*

**Parameters**

- **ctx** -- PROJ context, or NULL for default context

- **crs** -- Object of type CRS (must not be NULL)
- **options** -- null-terminated list of options, or NULL. Currently supported options are:
  - ALLOW_INTERMEDIATE_CRS=ALWAYS/IF_NO_DIRECT_TRANSFORMATION/NEVER. Defaults to NEVER. When set to ALWAYS/IF_NO_DIRECT_TRANSFORMATION, intermediate CRS may be considered when computing the possible transformations. Slower.

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_crs_create_bound_vertical_crs**(*PJ_CONTEXT* *ctx, const *PJ* *vert_crs, const *PJ* *hub_geographic_3D_crs, const char *grid_name)

Returns a BoundCRS, with a transformation to a hub geographic 3D crs (use EPSG:4979 for WGS84 for example), using a grid.

The returned object must be unreferenced with proj_destroy() after use. It should be used by at most one thread at a time.

**Since**
6.3

**Parameters**

- **ctx** -- PROJ context, or NULL for default context
- **vert_crs** -- Object of type VerticalCRS (must not be NULL)
- **hub_geographic_3D_crs** -- Object of type Geographic 3D CRS (must not be NULL)
- **grid_name** -- Grid name (typically a .gtx file)

**Returns**
Object that must be unreferenced with proj_destroy(), or NULL in case of error.

*PJ* ***proj_create_conversion_utm**(*PJ_CONTEXT* *ctx, int zone, int north)

Instantiate a ProjectedCRS with a Universal Transverse Mercator conversion.

See *osgeo::proj::operation::Conversion::createUTM()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor).

*PJ* ***proj_create_conversion_transverse_mercator**(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Transverse Mercator projection method.

See *osgeo::proj::operation::Conversion::createTransverseMercator()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_gauss_schreiber_transverse_mercator***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Gauss Schreiber Transverse Mercator projection method.

See *osgeo::proj::operation::Conversion::createGaussSchreiberTransverseMercator()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_transverse_mercator_south_oriented***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Transverse Mercator South Orientated projection method.

See *osgeo::proj::operation::Conversion::createTransverseMercatorSouthOriented()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_two_point_equidistant***(*PJ_CONTEXT* *ctx, double latitude_first_point, double longitude_first_point, double latitude_second_point, double longitude_secon_point, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Two Point Equidistant projection method.

See *osgeo::proj::operation::Conversion::createTwoPointEquidistant()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_tunisia_mapping_grid***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Tunisia Mining Grid projection method.

See *osgeo::proj::operation::Conversion::createTunisiaMiningGrid()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*Deprecated:*

Replaced by proj_create_conversion_tunisia_mining_grid

*PJ* ***proj_create_conversion_tunisia_mining_grid***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Tunisia Mining Grid projection method.

See *osgeo::proj::operation::Conversion::createTunisiaMiningGrid()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

**Since**

9.2

*PJ* ***proj_create_conversion_albers_equal_area***(*PJ_CONTEXT* *ctx, double latitude_false_origin, double longitude_false_origin, double latitude_first_parallel, double latitude_second_parallel, double easting_false_origin, double northing_false_origin, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Albers Conic Equal Area projection method.

See *osgeo::proj::operation::Conversion::createAlbersEqualArea()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_lambert_conic_conformal_1sp***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Conic Conformal 1SP projection method.

See *osgeo::proj::operation::Conversion::createLambertConicConformal_1SP()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_lambert_conic_conformal_1sp_variant_b***(*PJ_CONTEXT* *ctx, double latitude_nat_origin, double scale, double latitude_false_origin, double longitude_false_origin, double easting_false_origin, double northing_false_origin, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Conic Conformal (1SP Variant B) projection method.

See *osgeo::proj::operation::Conversion::createLambertConicConformal_1SP_VariantB()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

**Since**
9.2.1

*PJ* ***proj_create_conversion_lambert_conic_conformal_2sp***(*PJ_CONTEXT* *ctx, double latitude_false_origin, double longitude_false_origin, double latitude_first_parallel, double latitude_second_parallel, double easting_false_origin, double northing_false_origin, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Conic Conformal (2SP) projection method.

See *osgeo::proj::operation::Conversion::createLambertConicConformal_2SP()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_lambert_conic_conformal_2sp_michigan***(*PJ_CONTEXT* *ctx, double latitude_false_origin, double longitude_false_origin, double latitude_first_parallel, double latitude_second_parallel, double easting_false_origin, double northing_false_origin, double ellipsoid_scaling_factor, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Conic Conformal (2SP Michigan) projection method.

See *osgeo::proj::operation::Conversion::createLambertConicConformal_2SP_Michigan()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_lambert_conic_conformal_2sp_belgium***(*PJ_CONTEXT* *ctx, double latitude_false_origin, double longitude_false_origin, double latitude_first_parallel, double latitude_second_parallel, double easting_false_origin, double northing_false_origin, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Conic Conformal (2SP Belgium) projection method.

See *osgeo::proj::operation::Conversion::createLambertConicConformal_2SP_Belgium()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_azimuthal_equidistant***(*PJ_CONTEXT* *ctx, double latitude_nat_origin, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Modified Azimuthal Equidistant projection method.

See *osgeo::proj::operation::Conversion::createAzimuthalEquidistant()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_guam_projection***(*PJ_CONTEXT* *ctx, double latitude_nat_origin, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Guam Projection projection method.

See *osgeo::proj::operation::Conversion::createGuamProjection()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_bonne***(*PJ_CONTEXT* *ctx, double latitude_nat_origin, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Bonne projection method.

See *osgeo::proj::operation::Conversion::createBonne()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_lambert_cylindrical_equal_area_spherical***(*PJ_CONTEXT* *ctx, double latitude_first_parallel, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Cylindrical Equal Area (Spherical) projection method.

See *osgeo::proj::operation::Conversion::createLambertCylindricalEqualAreaSpherical()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_lambert_cylindrical_equal_area***(*PJ_CONTEXT* *ctx, double latitude_first_parallel, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Cylindrical Equal Area (ellipsoidal form) projection method.

See *osgeo::proj::operation::Conversion::createLambertCylindricalEqualArea()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_cassini_soldner***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Cassini-Soldner projection method.

See *osgeo::proj::operation::Conversion::createCassiniSoldner()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_equidistant_conic***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double latitude_first_parallel, double latitude_second_parallel, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Equidistant Conic projection method.

See *osgeo::proj::operation::Conversion::createEquidistantConic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_eckert_i***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Eckert I projection method.

See *osgeo::proj::operation::Conversion::createEckertI()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_eckert_ii***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Eckert II projection method.

See *osgeo::proj::operation::Conversion::createEckertII()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_eckert_iii***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Eckert III projection method.

See *osgeo::proj::operation::Conversion::createEckertIII()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_eckert_iv***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Eckert IV projection method.

See *osgeo::proj::operation::Conversion::createEckertIV()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_eckert_v***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Eckert V projection method.

See *osgeo::proj::operation::Conversion::createEckertV()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_eckert_vi***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Eckert VI projection method.

See *osgeo::proj::operation::Conversion::createEckertVI()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_equidistant_cylindrical***(*PJ_CONTEXT* *ctx, double latitude_first_parallel, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Equidistant Cylindrical projection method.

See *osgeo::proj::operation::Conversion::createEquidistantCylindrical()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_equidistant_cylindrical_spherical***(*PJ_CONTEXT* *ctx, double latitude_first_parallel, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Equidistant Cylindrical (Spherical) projection method.

See *osgeo::proj::operation::Conversion::createEquidistantCylindricalSpherical()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_gall***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Gall (Stereographic) projection method.

See *osgeo::proj::operation::Conversion::createGall()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_goode_homolosine***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Goode Homolosine projection method.

See *osgeo::proj::operation::Conversion::createGoodeHomolosine()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_interrupted_goode_homolosine***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Interrupted Goode Homolosine projection method.

See *osgeo::proj::operation::Conversion::createInterruptedGoodeHomolosine()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_geostationary_satellite_sweep_x***(*PJ_CONTEXT* *ctx, double center_long, double height, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Geostationary Satellite View projection method, with the sweep angle axis of the viewing instrument being x.

See *osgeo::proj::operation::Conversion::createGeostationarySatelliteSweepX()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_geostationary_satellite_sweep_y***(*PJ_CONTEXT* *ctx, double center_long, double height, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Geostationary Satellite View projection method, with the sweep angle axis of the viewing instrument being y.

See *osgeo::proj::operation::Conversion::createGeostationarySatelliteSweepY()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_gnomonic***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Gnomonic projection method.

See *osgeo::proj::operation::Conversion::createGnomonic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_hotine_oblique_mercator_variant_a***(*PJ_CONTEXT* *ctx, double latitude_projection_centre, double longitude_projection_centre, double azimuth_initial_line, double angle_from_rectified_to_skrew_grid, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Hotine Oblique Mercator (Variant A) projection method.

See *osgeo::proj::operation::Conversion::createHotineObliqueMercatorVariantA()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_hotine_oblique_mercator_variant_b***(*PJ_CONTEXT* *ctx, double latitude_projection_centre, double longitude_projection_centre, double azimuth_initial_line, double angle_from_rectified_to_skrew_grid, double scale, double easting_projection_centre, double northing_projection_centre, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Hotine Oblique Mercator (Variant B) projection method.

See *osgeo::proj::operation::Conversion::createHotineObliqueMercatorVariantB()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_hotine_oblique_mercator_two_point_natural_origin***(*PJ_CONTEXT* *ctx, double latitude_projection_centre, double latitude_point1, double longitude_point1, double latitude_point2, double longitude_point2, double scale, double easting_projection_centre, double northing_projection_centre, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Hotine Oblique Mercator Two Point Natural Origin projection method.

See *osgeo::proj::operation::Conversion::createHotineObliqueMercatorTwoPointNaturalOrigin()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_laborde_oblique_mercator***(*PJ_CONTEXT* *ctx, double latitude_projection_centre, double longitude_projection_centre, double azimuth_initial_line, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Laborde Oblique Mercator projection method.

See *osgeo::proj::operation::Conversion::createLabordeObliqueMercator()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_international_map_world_polyconic***(*PJ_CONTEXT* *ctx, double center_long, double latitude_first_parallel, double latitude_second_parallel, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the International Map of the World Polyconic projection method.

See *osgeo::proj::operation::Conversion::createInternationalMapWorldPolyconic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_krovak_north_oriented***(*PJ_CONTEXT* *ctx, double latitude_projection_centre, double longitude_of_origin, double colatitude_cone_axis, double latitude_pseudo_standard_parallel, double scale_factor_pseudo_standard_parallel, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Krovak (north oriented) projection method.

See *osgeo::proj::operation::Conversion::createKrovakNorthOriented()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_krovak***(*PJ_CONTEXT* *ctx, double latitude_projection_centre, double longitude_of_origin, double colatitude_cone_axis, double latitude_pseudo_standard_parallel, double scale_factor_pseudo_standard_parallel, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Krovak projection method.

See *osgeo::proj::operation::Conversion::createKrovak()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_lambert_azimuthal_equal_area***(*PJ_CONTEXT* *ctx, double latitude_nat_origin, double longitude_nat_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Lambert Azimuthal Equal Area projection method.

See *osgeo::proj::operation::Conversion::createLambertAzimuthalEqualArea()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_miller_cylindrical***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Miller Cylindrical projection method.

See *osgeo::proj::operation::Conversion::createMillerCylindrical()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_mercator_variant_a***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Mercator projection method.

See *osgeo::proj::operation::Conversion::createMercatorVariantA()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_mercator_variant_b***(*PJ_CONTEXT* *ctx, double latitude_first_parallel, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Mercator projection method.

See *osgeo::proj::operation::Conversion::createMercatorVariantB()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_popular_visualisation_pseudo_mercator***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Popular Visualisation Pseudo Mercator projection method.

See *osgeo::proj::operation::Conversion::createPopularVisualisationPseudoMercator()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_mollweide***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Mollweide projection method.

See *osgeo::proj::operation::Conversion::createMollweide()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_new_zealand_mapping_grid***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the New Zealand Map Grid projection method.

See *osgeo::proj::operation::Conversion::createNewZealandMappingGrid()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_oblique_stereographic***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Oblique Stereographic (Alternative) projection method.

See *osgeo::proj::operation::Conversion::createObliqueStereographic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_orthographic***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Orthographic projection method.

See *osgeo::proj::operation::Conversion::createOrthographic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_local_orthographic***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double azimuth, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Local Orthographic projection method.

See *osgeo::proj::operation::Conversion::createLocalOrthographic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_american_polyconic***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the American Polyconic projection method.

See *osgeo::proj::operation::Conversion::createAmericanPolyconic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_polar_stereographic_variant_a***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Polar Stereographic (Variant A) projection method.

See *osgeo::proj::operation::Conversion::createPolarStereographicVariantA()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_polar_stereographic_variant_b***(*PJ_CONTEXT* *ctx, double latitude_standard_parallel, double longitude_of_origin, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Polar Stereographic (Variant B) projection method.

See *osgeo::proj::operation::Conversion::createPolarStereographicVariantB()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_robinson***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Robinson projection method.

See *osgeo::proj::operation::Conversion::createRobinson()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_sinusoidal***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Sinusoidal projection method.

See *osgeo::proj::operation::Conversion::createSinusoidal()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_stereographic***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double scale, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Stereographic projection method.

See *osgeo::proj::operation::Conversion::createStereographic()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_van_der_grinten***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Van der Grinten projection method.

See *osgeo::proj::operation::Conversion::createVanDerGrinten()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_wagner_i***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Wagner I projection method.

See *osgeo::proj::operation::Conversion::createWagnerI()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_wagner_ii***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Wagner II projection method.

See *osgeo::proj::operation::Conversion::createWagnerII()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_wagner_iii***(*PJ_CONTEXT* *ctx, double latitude_true_scale, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Wagner III projection method.

See *osgeo::proj::operation::Conversion::createWagnerIII()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_wagner_iv***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Wagner IV projection method.

See *osgeo::proj::operation::Conversion::createWagnerIV()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_wagner_v***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Wagner V projection method.

See *osgeo::proj::operation::Conversion::createWagnerV()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_wagner_vi***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Wagner VI projection method.

See *osgeo::proj::operation::Conversion::createWagnerVI()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_wagner_vii***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Wagner VII projection method.

See *osgeo::proj::operation::Conversion::createWagnerVII()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_quadrilateralized_spherical_cube***(*PJ_CONTEXT* *ctx, double center_lat, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Quadrilateralized Spherical Cube projection method.

See *osgeo::proj::operation::Conversion::createQuadrilateralizedSphericalCube()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_spherical_cross_track_height***(*PJ_CONTEXT* *ctx, double peg_point_lat, double peg_point_long, double peg_point_heading, double peg_point_height, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Spherical Cross-Track Height projection method.

See *osgeo::proj::operation::Conversion::createSphericalCrossTrackHeight()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_equal_earth***(*PJ_CONTEXT* *ctx, double center_long, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a ProjectedCRS with a conversion based on the Equal Earth projection method.

See *osgeo::proj::operation::Conversion::createEqualEarth()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_vertical_perspective***(*PJ_CONTEXT* *ctx, double topo_origin_lat, double topo_origin_long, double topo_origin_height, double view_point_height, double false_easting, double false_northing, const char *ang_unit_name, double ang_unit_conv_factor, const char *linear_unit_name, double linear_unit_conv_factor)

Instantiate a conversion based on the Vertical Perspective projection method.

See *osgeo::proj::operation::Conversion::createVerticalPerspective()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

**Since**
6.3

*PJ* ***proj_create_conversion_pole_rotation_grib_convention***(*PJ_CONTEXT* *ctx, double south_pole_lat_in_unrotated_crs, double south_pole_long_in_unrotated_crs, double axis_rotation, const char *ang_unit_name, double ang_unit_conv_factor)

Instantiate a conversion based on the Pole Rotation method, using the conventions of the GRIB 1 and GRIB 2 data formats.

See *osgeo::proj::operation::Conversion::createPoleRotationGRIBConvention()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

*PJ* ***proj_create_conversion_pole_rotation_netcdf_cf_convention***(*PJ_CONTEXT* *ctx, double grid_north_pole_latitude, double grid_north_pole_longitude, double north_pole_grid_longitude, const char *ang_unit_name, double ang_unit_conv_factor)

Instantiate a conversion based on the Pole Rotation method, using the conventions of the netCDF CF convention for the netCDF format.

See *osgeo::proj::operation::Conversion::createPoleRotationNetCDFCFConvention()*.

Linear parameters are expressed in (linear_unit_name, linear_unit_conv_factor). Angular parameters are expressed in (ang_unit_name, ang_unit_conv_factor).

## 10.5.4 C++ API

### 10.5.4.1 General documentation

*page* `General documentation`

#### General API design

The design of the class hierarchy is strongly derived from *ISO 19111:2019*.

Classes for which the constructors are not directly accessible have their instances constructed with create() methods. The returned object is a non-null shared pointer. Such objects are immutable, and thread-safe.

TODO

#### General properties

All classes deriving from IdentifiedObject have general properties that can be defined at creation time. Those properties are:

- *osgeo::proj::metadata::Identifier::DESCRIPTION_KEY* ("description"): the natural language description of the meaning of the code value, provided a a string.
- *osgeo::proj::metadata::Identifier::CODE_KEY* ("code"): a numeric or alphanumeric code, provided as a integer or a string. For example 4326, for the EPSG:4326 "WGS84" GeographicalCRS
- *osgeo::proj::metadata::Identifier::CODESPACE_KEY* ("codespace"): the organization responsible for definition and maintenance of the code., provided a a string. For example "EPSG".
- *osgeo::proj::metadata::Identifier::VERSION_KEY* ("version"): the version identifier for the namespace, provided a a string.
- *osgeo::proj::metadata::Identifier::AUTHORITY_KEY* ("authority"): a citation for the authority, provided as a string or a *osgeo::proj::metadata::Citation* object. Often unused
- *osgeo::proj::metadata::Identifier::URI_KEY* ("uri"): the URI of the identifier, provided as a string. Often unused

- *osgeo::proj::common::IdentifiedObject::NAME_KEY* ("name"): the name of a *osgeo::proj::common::IdentifiedObject*, provided as a string or *osgeo::proj::metadata::IdentifierNNPtr*.
- *osgeo::proj::common::IdentifiedObject::IDENTIFIERS_KEY* ("identifiers"): the identifier(s) of a *osgeo::proj::common::IdentifiedObject*, provided as a osgeo::proj::common::IdentifierNNPtr or a *osgeo::proj::util::ArrayOfBaseObjectNNPtr* of *osgeo::proj::metadata::IdentifierNNPtr*.
- *osgeo::proj::common::IdentifiedObject::ALIAS_KEY* ("alias"): the alias(es) of a *osgeo::proj::common::IdentifiedObject*, provided as string, a *osgeo::proj::util::GenericNameNNPtr* or a *osgeo::proj::util::ArrayOfBaseObjectNNPtr* of *osgeo::proj::util::GenericNameNNPtr*.
- *osgeo::proj::common::IdentifiedObject::REMARKS_KEY* ("remarks"): the remarks of a *osgeo::proj::common::IdentifiedObject*, provided as a string.
- *osgeo::proj::common::IdentifiedObject::DEPRECATED_KEY* ("deprecated"): the deprecation flag of a *osgeo::proj::common::IdentifiedObject*, provided as a boolean.

- *osgeo::proj::common::ObjectUsage::SCOPE_KEY* ("scope"): the scope of a *osgeo::proj::common::ObjectUsage*, provided as a string.
- *osgeo::proj::common::ObjectUsage::DOMAIN_OF_VALIDITY_KEY* ("domainOfValidity"): the domain of validity of a *osgeo::proj::common::ObjectUsage*, provided as a *osgeo::proj::metadata::ExtentNNPtr*.
- *osgeo::proj::common::ObjectUsage::OBJECT_DOMAIN_KEY* ("objectDomain"): the object domain(s) of a *osgeo::proj::common::ObjectUsage*, provided as a *osgeo::proj::common::ObjectDomainNNPtr* or a *osgeo::proj::util::ArrayOfBaseObjectNNPtr* of *osgeo::proj::common::ObjectDomainNNPtr*.

### Applicable standards

#### ISO:19111 / OGC Topic 2 standard

Topic 2 - Spatial referencing by coordinates.

This is an Abstract Specification describes the data elements, relationships and associated metadata required for spatial referencing by coordinates. It describes Coordinate Reference Systems (CRS), coordinate systems (CS) and coordinate transformation or coordinate conversion between two different coordinate reference systems.

#### ISO 19111:2019

This is the revision mostly used for PROJ C++ modelling.

OGC 18-005r5, 2021-07-02, ISO 19111:2019

#### ISO 19111:2007

The precedent version of the specification was: OGC 08-015r2, 2010-04-27, ISO 19111:2007

#### WKT2 standard

Well-known text representation of coordinate reference systems.

Well-known Text (WKT) offers a compact machine- and human-readable representation of the critical elements of coordinate reference system (CRS) definitions, and coordinate operations. This is an implementation of *ISO:19111 / OGC Topic 2 standard*

PROJ implements the two following revisions of the standard:

#### WKT2:2019

OGC 18-010r7, 2019-06-24, WKT2-2019

#### WKT2:2015

OGC 12-063r5, 2015-05-01, ISO 19162:2015(E), WKT2-2015

#### WKT1 specification

Older specifications of well-known text representation of coordinate reference systems are also supported by PROJ, mostly for compatibility with legacy systems, or older versions of GDAL.

GDAL v2.4 and earlier mostly implements:

OGC 01-009, 2001-01-12, OpenGIS Coordinate Transformation Service Implementation Specification

The GDAL documentation, OGC WKT Coordinate System Issues discusses issues, and GDAL implementation choices.

An older specification of WKT1 is/was used by some software packages:

OGC 99-049, 1999-05-05, OpenGIS Simple Features Specification For SQL v1.1

#### ISO 19115 (Metadata)

Defines the schema required for describing geographic information and services. It provides information about the identification, the extent, the quality, the spatial and temporal schema, spatial reference, and distribution of digital geographic data.

PROJ implements a simplified subset of ISO 19115.

#### GeoAPI

A set of Java and Python language programming interfaces for geospatial applications.

GeoAPI main page

GeoAPI Javadoc

OGC GeoAPI Implementation Specification

### 10.5.4.2 common namespace

namespace **common**

Common classes.

*osgeo.proj.common* namespace

#### Typedefs

typedef std::shared_ptr<*UnitOfMeasure*> **UnitOfMeasurePtr**

Shared pointer of *UnitOfMeasure*.

typedef *util*::nn<*UnitOfMeasurePtr*> **UnitOfMeasureNNPtr**

Non-null shared pointer of *UnitOfMeasure*.

typedef std::shared_ptr<*IdentifiedObject*> **IdentifiedObjectPtr**

Shared pointer of *IdentifiedObject*.

typedef *util*::nn<*IdentifiedObjectPtr*> **IdentifiedObjectNNPtr**

Non-null shared pointer of *IdentifiedObject*.

using **ObjectDomainPtr** = std::shared_ptr<*ObjectDomain*>

Shared pointer of *ObjectDomain*.

using **ObjectDomainNNPtr** = *util*::nn<*ObjectDomainPtr*>

Non-null shared pointer of *ObjectDomain*.

using **ObjectUsagePtr** = std::shared_ptr<*ObjectUsage*>

Shared pointer of *ObjectUsage*.

using **ObjectUsageNNPtr** = *util*::nn<*ObjectUsagePtr*>

Non-null shared pointer of *ObjectUsage*.

class **UnitOfMeasure** : public osgeo::proj::*util*::*BaseObject*

*#include <common.hpp>* Unit of measure.

This is a mutable object.

### Public Types

enum class **Type**

Type of unit of measure.

*Values:*

enumerator **UNKNOWN**

Unknown unit of measure

enumerator **NONE**

No unit of measure

enumerator **ANGULAR**

Angular unit of measure

enumerator **LINEAR**

Linear unit of measure

enumerator **SCALE**

*Scale* unit of measure

enumerator **TIME**

Time unit of measure

enumerator **PARAMETRIC**

Parametric unit of measure

### Public Functions

explicit **UnitOfMeasure**(const std::string &nameIn = std::string(), double toSIIn = 1.0, *Type* typeIn = *Type*::*UNKNOWN*, const std::string &codeSpaceIn = std::string(), const std::string &codeIn = std::string())

Creates a *UnitOfMeasure*.

const std::string &**name**()

Return the name of the unit of measure.

double **conversionToSI**()

Return the conversion factor to the unit of the International System of Units of the same Type.

For example, for foot, this would be 0.3048 (metre)

**Returns**
the conversion factor, or 0 if no conversion exists.

*Type* **type**()

Return the type of the unit of measure.

const std::string &**codeSpace**()

Return the code space of the unit of measure.

For example "EPSG"

**Returns**
the code space, or empty string.

const std::string &**code**()

Return the code of the unit of measure.

**Returns**
the code, or empty string.

bool **operator==**(const *UnitOfMeasure* &other)

Returns whether two units of measures are equal.

The comparison is based on the name.

bool **operator!=**(const *UnitOfMeasure* &other)

Returns whether two units of measures are different.

The comparison is based on the name.

### Public Static Attributes

static const *UnitOfMeasure* **NONE**

"Empty"/"None", unit of measure of type NONE.

static const *UnitOfMeasure* **SCALE_UNITY**

*Scale* unity, unit of measure of type SCALE.

static const *UnitOfMeasure* **PARTS_PER_MILLION**

Parts-per-million, unit of measure of type SCALE.

static const *UnitOfMeasure* **PPM_PER_YEAR**

Parts-per-million per year, unit of measure of type SCALE.

static const *UnitOfMeasure* **METRE**

Metre, unit of measure of type LINEAR (SI unit).

static const *UnitOfMeasure* **METRE_PER_YEAR**

Metre per year, unit of measure of type LINEAR.

static const *UnitOfMeasure* **FOOT**

Foot, unit of measure of type LINEAR.

static const *UnitOfMeasure* **US_FOOT**

US survey foot, unit of measure of type LINEAR.

static const *UnitOfMeasure* **RADIAN**

Radian, unit of measure of type ANGULAR (SI unit).

static const *UnitOfMeasure* **MICRORADIAN**

Microradian, unit of measure of type ANGULAR.

static const *UnitOfMeasure* **DEGREE**

Degree, unit of measure of type ANGULAR.

static const *UnitOfMeasure* **ARC_SECOND**

Arc-second, unit of measure of type ANGULAR.

static const *UnitOfMeasure* **GRAD**

Grad, unit of measure of type ANGULAR.

static const *UnitOfMeasure* **ARC_SECOND_PER_YEAR**

Arc-second per year, unit of measure of type ANGULAR.

static const *UnitOfMeasure* **SECOND**

Second, unit of measure of type TIME (SI unit).

static const *UnitOfMeasure* **YEAR**

Year, unit of measure of type TIME.

class **Measure** : public osgeo::proj::*util*::*BaseObject*

*#include <common.hpp>* Numeric value associated with a *UnitOfMeasure*.

Subclassed by *osgeo::proj::common::Angle*, *osgeo::proj::common::Length*, *osgeo::proj::common::Scale*

### Public Functions

**Measure**(double valueIn = 0.0, const *UnitOfMeasure* &unitIn = *UnitOfMeasure*())

Instantiate a *Measure*.

const *UnitOfMeasure* &**unit**()

Return the unit of the *Measure*.

double **getSIValue**()

Return the value of the *Measure*, after conversion to the corresponding unit of the International System.

double **value**()

Return the value of the measure, expressed in the *unit()*

double **convertToUnit**(const *UnitOfMeasure* &otherUnit)

Return the value of this measure expressed into the provided unit.

bool **operator==**(const *Measure* &other)

Return whether two measures are equal.

The comparison is done both on the value and the unit.

bool **_isEquivalentTo**(const *Measure* &other, *util*::*IComparable*::*Criterion* criterion = *util*::*IComparable*::*Criterion*::*STRICT*, double maxRelativeError = *DEFAULT_MAX_REL_ERROR*) const

Returns whether an object is equivalent to another one.

**Parameters**

- **other** -- other object to compare to
- **criterion** -- comparison criterion.
- **maxRelativeError** -- Maximum relative error allowed.

**Returns**

true if objects are equivalent.

### Public Static Attributes

static constexpr double **DEFAULT_MAX_REL_ERROR** = 1e-10

Default maximum resulative error.

class **Scale** : public osgeo::proj::*common*::*Measure*

*#include <common.hpp>* Numeric value, without a physical unit of measure.

### Public Functions

explicit **Scale**(double valueIn = 0.0)

Instantiate a *Scale*.

**Parameters**

**valueIn** -- value

explicit **Scale**(double valueIn, const *UnitOfMeasure* &unitIn)

Instantiate a *Scale*.

**Parameters**

- **valueIn** -- value
- **unitIn** -- unit. Constraint: unit.type() == *UnitOfMeasure::Type::SCALE*

class **Angle** : public osgeo::proj::*common*::*Measure*

*#include <common.hpp>* Numeric value, with a angular unit of measure.

### Public Functions

explicit **Angle**(double valueIn = 0.0)

Instantiate a *Angle*.

**Parameters**

**valueIn** -- value

**Angle**(double valueIn, const *UnitOfMeasure* &unitIn)

Instantiate a *Angle*.

**Parameters**

- **valueIn** -- value

- **unitIn** -- unit. Constraint: unit.type() == *UnitOfMeasure::Type::ANGULAR*

class **Length** : public osgeo::proj::*common*::*Measure*

*#include <common.hpp>* Numeric value, with a linear unit of measure.

### Public Functions

explicit **Length**(double valueIn = 0.0)

Instantiate a *Length*.

**Parameters**

**valueIn** -- value

**Length**(double valueIn, const *UnitOfMeasure* &unitIn)

Instantiate a *Length*.

**Parameters**

- **valueIn** -- value
- **unitIn** -- unit. Constraint: unit.type() == *UnitOfMeasure::Type::LINEAR*

class **DateTime**

*#include <common.hpp>* Date-time value, as a ISO:8601 encoded string, or other string encoding.

### Public Functions

bool **isISO_8601**() const

Return whether the *DateTime* is ISO:8601 compliant.

**Remark**

The current implementation is really simplistic, and aimed at detecting date-times that are not ISO:8601 compliant.

std::string **toString**() const

Return the *DateTime* as a string.

### Public Static Functions

static *DateTime* **create**(const std::string &str)

Instantiate a *DateTime*.

class **DataEpoch**

*#include <common.hpp>* Data epoch.

### Public Functions

const *Measure* &**coordinateEpoch**() const

Return the coordinate epoch, as a measure in decimal year.

class **IdentifiedObject** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*util*::*IComparable*, public osgeo::proj::*io*::*IWKTExportable*

*#include <common.hpp>* Abstract class representing a CRS-related object that has an identification.

**Remark**

Implements *IdentifiedObject* from *ISO 19111:2019*

Subclassed by *osgeo::proj::common::ObjectUsage*, *osgeo::proj::cs::CoordinateSystem*, *osgeo::proj::cs::CoordinateSystemAxis*, *osgeo::proj::cs::Meridian*, *osgeo::proj::datum::Ellipsoid*, *osgeo::proj::datum::PrimeMeridian*, *osgeo::proj::operation::GeneralOperationParameter*, *osgeo::proj::operation::OperationMethod*

### Public Functions

const *metadata*::*IdentifierNNPtr* &**name**()

Return the name of the object.

Generally, the only interesting field of the name will be *name()*->description().

const std::string &**nameStr**()

Return the name of the object.

Return *(*name()*->description())

const std::vector<*metadata*::*IdentifierNNPtr*> &**identifiers**()

Return the identifier(s) of the object.

Generally, those will have Identifier::code() and Identifier::codeSpace() filled.

const std::vector<*util*::*GenericNameNNPtr*> &**aliases**()

Return the alias(es) of the object.

const std::string &**remarks**()

Return the remarks.

bool **isDeprecated**()

Return whether the object is deprecated.

**Remark**

Extension of *ISO 19111:2019*

std::string **alias**()

Return the (first) alias of the object as a string.

Shortcut for *aliases()*[0]->toFullyQualifiedName()->toString()

int **getEPSGCode**()

Return the EPSG code.

**Returns**

code, or 0 if not found

### Public Static Attributes

static const std::string **NAME_KEY**

Key to set the name of a *common::IdentifiedObject*.

The value is to be provided as a string or *metadata::IdentifierNNPtr*.

static const std::string **IDENTIFIERS_KEY**

Key to set the identifier(s) of a *common::IdentifiedObject*.

The value is to be provided as a common::IdentifierNNPtr or a *util::ArrayOfBaseObjectNNPtr* of common::IdentifierNNPtr.

static const std::string **ALIAS_KEY**

Key to set the alias(es) of a *common::IdentifiedObject*.

The value is to be provided as string, a *util::GenericNameNNPtr* or a *util::ArrayOfBaseObjectNNPtr* of *util::GenericNameNNPtr*.

static const std::string **REMARKS_KEY**

Key to set the remarks of a *common::IdentifiedObject*.

The value is to be provided as a string.

static const std::string **DEPRECATED_KEY**

Key to set the deprecation flag of a *common::IdentifiedObject*.

The value is to be provided as a boolean.

class **ObjectDomain** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*util*::*IComparable*

*#include <common.hpp>* The scope and validity of a CRS-related object.

**Remark**

Implements *ObjectDomain* from *ISO 19111:2019*

### Public Functions

const *util*::*optional*<std::string> &**scope**()

Return the scope.

**Returns**

the scope, or empty.

const *metadata*::*ExtentPtr* &**domainOfValidity**()

Return the domain of validity.

**Returns**

the domain of validity, or nullptr.

#### Public Static Functions

static *ObjectDomainNNPtr* **create**(const *util*::*optional*<std::string> &scopeIn, const *metadata*::*ExtentPtr* &extent)

Instantiate a *ObjectDomain*.

class **ObjectUsage** : public osgeo::proj::*common*::*IdentifiedObject*

*#include <common.hpp>* Abstract class of a CRS-related object that has usages.

**Remark**

Implements *ObjectUsage* from *ISO 19111:2019*

Subclassed by *osgeo::proj::crs::CRS*, *osgeo::proj::datum::Datum*, *osgeo::proj::datum::DatumEnsemble*, *osgeo::proj::operation::CoordinateOperation*

#### Public Functions

const std::vector<*ObjectDomainNNPtr*> &**domains**()

Return the domains of the object.

#### Public Static Attributes

static const std::string **SCOPE_KEY**

Key to set the scope of a *common::ObjectUsage*.

The value is to be provided as a string.

static const std::string **DOMAIN_OF_VALIDITY_KEY**

Key to set the domain of validity of a *common::ObjectUsage*.

The value is to be provided as a common::ExtentNNPtr.

static const std::string **OBJECT_DOMAIN_KEY**

Key to set the object domain(s) of a *common::ObjectUsage*.

The value is to be provided as a *common::ObjectDomainNNPtr* or a *util::ArrayOfBaseObjectNNPtr* of *common::ObjectDomainNNPtr*.

### 10.5.4.3 util namespace

namespace **util**

A set of base types from ISO 19103, *GeoAPI* and other PROJ specific classes.

*osgeo.proj.util* namespace.

### Typedefs

using **BaseObjectPtr** = std::shared_ptr<*BaseObject*>

Shared pointer of *BaseObject*.

using **BoxedValuePtr** = std::shared_ptr<*BoxedValue*>

Shared pointer of *BoxedValue*.

using **BoxedValueNNPtr** = *util*::nn<*BoxedValuePtr*>

Non-null shared pointer of *BoxedValue*.

using **ArrayOfBaseObjectPtr** = std::shared_ptr<*ArrayOfBaseObject*>

Shared pointer of *ArrayOfBaseObject*.

using **ArrayOfBaseObjectNNPtr** = *util*::nn<*ArrayOfBaseObjectPtr*>

Non-null shared pointer of *ArrayOfBaseObject*.

using **LocalNamePtr** = std::shared_ptr<*LocalName*>

Shared pointer of *LocalName*.

using **LocalNameNNPtr** = *util*::nn<*LocalNamePtr*>

Non-null shared pointer of *LocalName*.

using **NameSpacePtr** = std::shared_ptr<*NameSpace*>

Shared pointer of *NameSpace*.

using **NameSpaceNNPtr** = *util*::nn<*NameSpacePtr*>

Non-null shared pointer of *NameSpace*.

using **GenericNamePtr** = std::shared_ptr<*GenericName*>

Shared pointer of *GenericName*.

using **GenericNameNNPtr** = *util*::nn<*GenericNamePtr*>

Non-null shared pointer of *GenericName*.

template<class **T**>

class **optional**

*#include <util.hpp>* Loose transposition of std::optional available from C++17.

### Public Functions

inline const *T* ***operator->**()* const

Returns a pointer to the contained value.

inline const *T* &**operator***()* const

Returns a reference to the contained value.

inline explicit **operator bool**() const noexcept

Return whether the optional has a value

inline bool **has_value**() const noexcept

Return whether the optional has a value

struct **BaseObjectNNPtr** : public *util*::nn<*BaseObjectPtr*>

*#include <util.hpp>* Non-null shared pointer of *BaseObject*.

class **BaseObject**

*#include <util.hpp>* Class that can be derived from, to emulate Java's Object behavior.

Subclassed by *osgeo::proj::common::IdentifiedObject*, *osgeo::proj::common::Measure*, *osgeo::proj::common::ObjectDomain*, *osgeo::proj::common::UnitOfMeasure*, osgeo::proj::coordinates::CoordinateMetadata, *osgeo::proj::metadata::Citation*, *osgeo::proj::metadata::Extent*, *osgeo::proj::metadata::GeographicExtent*, *osgeo::proj::metadata::Identifier*, *osgeo::proj::metadata::PositionalAccuracy*, *osgeo::proj::metadata::TemporalExtent*, *osgeo::proj::metadata::VerticalExtent*, *osgeo::proj::operation::GeneralParameterValue*, *osgeo::proj::operation::ParameterValue*, *osgeo::proj::util::ArrayOfBaseObject*, *osgeo::proj::util::BoxedValue*, *osgeo::proj::util::GenericName*

class **IComparable**

*#include <util.hpp>* Interface for an object that can be compared to another.

Subclassed by *osgeo::proj::common::IdentifiedObject*, *osgeo::proj::common::ObjectDomain*, *osgeo::proj::metadata::Extent*, *osgeo::proj::metadata::GeographicExtent*, *osgeo::proj::metadata::TemporalExtent*, *osgeo::proj::metadata::VerticalExtent*, *osgeo::proj::operation::GeneralParameterValue*, *osgeo::proj::operation::ParameterValue*

### Public Types

enum class **Criterion**

Comparison criterion.

*Values:*

enumerator **STRICT**

All properties are identical.

enumerator **EQUIVALENT**

The objects are equivalent for the purpose of coordinate operations. They can differ by the name of their objects, identifiers, other metadata. Parameters may be expressed in different units, provided that the value is (with some tolerance) the same once expressed in a common unit.

enumerator **EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS**

Same as EQUIVALENT, relaxed with an exception that the axis order of the base CRS of a DerivedCRS/ProjectedCRS or the axis order of a GeographicCRS is ignored. Only to be used with DerivedCRS/ProjectedCRS/GeographicCRS

### Public Functions

bool **isEquivalentTo**(const *IComparable* *other, *Criterion* criterion = *Criterion*::*STRICT*, const *io*::*DatabaseContextPtr* &dbContext = nullptr) const

Returns whether an object is equivalent to another one.

**Parameters**

- **other** -- other object to compare to
- **criterion** -- comparison criterion.
- **dbContext** -- Database context, or nullptr.

**Returns**

true if objects are equivalent.

class **BoxedValue** : public osgeo::proj::*util*::*BaseObject*

*#include <util.hpp>* Encapsulate standard datatypes in an object.

### Public Functions

**BoxedValue**(const char *stringValueIn)

Constructs a *BoxedValue* from a string.

**BoxedValue**(const std::string &stringValueIn)

Constructs a *BoxedValue* from a string.

**BoxedValue**(int integerValueIn)

Constructs a *BoxedValue* from an integer.

**BoxedValue**(bool booleanValueIn)

Constructs a *BoxedValue* from a boolean.

class **ArrayOfBaseObject** : public osgeo::proj::*util*::*BaseObject*

*#include <util.hpp>* Array of *BaseObject*.

### Public Functions

void **add**(const *BaseObjectNNPtr* &obj)

Adds an object to the array.

**Parameters**

**obj** -- the object to add.

### Public Static Functions

static *ArrayOfBaseObjectNNPtr* **create**()

Instantiate a *ArrayOfBaseObject*.

**Returns**

a new *ArrayOfBaseObject*.

class **PropertyMap**

*#include <util.hpp>* Wrapper of a std::map<std::string, BaseObjectNNPtr>

### Public Functions

*PropertyMap* &**set**(const std::string &key, const *BaseObjectNNPtr* &val)

Set a *BaseObjectNNPtr* as the value of a key.

*PropertyMap* &**set**(const std::string &key, const char *val)

Set a string as the value of a key.

*PropertyMap* &**set**(const std::string &key, const std::string &val)

Set a string as the value of a key.

*PropertyMap* &**set**(const std::string &key, int val)

Set a integer as the value of a key.

*PropertyMap* &**set**(const std::string &key, bool val)

Set a boolean as the value of a key.

*PropertyMap* &**set**(const std::string &key, const std::vector<std::string> &array)

Set a vector of strings as the value of a key.

class **GenericName** : public osgeo::proj::*util*::*BaseObject*

*#include <util.hpp>* A sequence of identifiers rooted within the context of a namespace.

**Remark**

Simplified version of GenericName from *GeoAPI*

Subclassed by *osgeo::proj::util::LocalName*

### Public Functions

virtual const *NameSpacePtr* **scope**() const = 0

Return the scope of the object, possibly a global one.

virtual std::string **toString**() const = 0

Return the *LocalName* as a string.

virtual *GenericNameNNPtr* **toFullyQualifiedName**() const = 0

Return a fully qualified name corresponding to the local name.

The namespace of the resulting name is a global one.

class **NameSpace**

*#include <util.hpp>* A domain in which names given by strings are defined.

**Remark**

Simplified version of NameSpace from *GeoAPI*

### Public Functions

bool **isGlobal**() const

Returns whether this is a global namespace.

const *GenericNamePtr* &**name**() const

Returns the name of this namespace.

class **LocalName** : public osgeo::proj::*util*::*GenericName*

*#include <util.hpp>* Identifier within a *NameSpace* for a local object.

Local names are names which are directly accessible to and maintained by a *NameSpace* within which they are local, indicated by the scope.

**Remark**

Simplified version of LocalName from *GeoAPI*

### Public Functions

virtual const *NameSpacePtr* **scope**() const override

Return the scope of the object, possibly a global one.

virtual std::string **toString**() const override

Return the *LocalName* as a string.

virtual *GenericNameNNPtr* **toFullyQualifiedName**() const override

Return a fully qualified name corresponding to the local name.

The namespace of the resulting name is a global one.

class **NameFactory**

*#include <util.hpp>* Factory for generic names.

**Remark**

Simplified version of NameFactory from *GeoAPI*

### Public Static Functions

static *NameSpaceNNPtr* **createNameSpace**(const *GenericNameNNPtr* &name, const *PropertyMap* &properties)

Instantiate a *NameSpace*.

**Parameters**

- **name** -- name of the namespace.
- **properties** -- Properties. Allowed keys are "separator" and "separator.head".

**Returns**

a new *NameFactory*.

static *LocalNameNNPtr* **createLocalName**(const *NameSpacePtr* &scope, const std::string &name)

Instantiate a *LocalName*.

**Parameters**

- **scope** -- scope.
- **name** -- string of the local name.

**Returns**
a new *LocalName*.

static *GenericNameNNPtr* **createGenericName**(const *NameSpacePtr* &scope, const std::vector<std::string> &parsedNames)

Instantiate a *GenericName*.

**Parameters**
- **scope** -- scope.
- **parsedNames** -- the components of the name.

**Returns**
a new *GenericName*.

class **CodeList**

*#include <util.hpp>* Abstract class to define an enumeration of values.

Subclassed by *osgeo::proj::cs::AxisDirection*, *osgeo::proj::cs::RangeMeaning*, *osgeo::proj::datum::RealizationMethod*

### Public Functions

inline const std::string &**toString**()

Return the *CodeList* item as a string.

inline **operator std::string**()

Return the *CodeList* item as a string.

class **Exception** : public std::exception

*#include <util.hpp>* Root exception class.

Subclassed by *osgeo::proj::crs::InvalidCompoundCRSException*, *osgeo::proj::io::FactoryException*, *osgeo::proj::io::FormattingException*, *osgeo::proj::io::ParsingException*, *osgeo::proj::operation::InvalidOperation*, *osgeo::proj::util::InvalidValueTypeException*, *osgeo::proj::util::UnsupportedOperationException*

### Public Functions

virtual const char *__what__() const noexcept override

Return the exception text.

class **InvalidValueTypeException** : public osgeo::proj::*util*::*Exception*

*#include <util.hpp>* *Exception* thrown when an invalid value type is set as the value of a key of a *PropertyMap*.

class **UnsupportedOperationException** : public osgeo::proj::*util*::*Exception*

*#include <util.hpp>* *Exception* Thrown to indicate that the requested operation is not supported.

#### 10.5.4.4 metadata namespace

namespace **metadata**

Common classes from *ISO 19115 (Metadata)* standard.

*osgeo.proj.metadata* namespace

### Typedefs

typedef std::shared_ptr<*Extent*> **ExtentPtr**

Shared pointer of *Extent*.

typedef *util*::nn<*ExtentPtr*> **ExtentNNPtr**

Non-null shared pointer of *Extent*.

using **GeographicExtentPtr** = std::shared_ptr<*GeographicExtent*>

Shared pointer of *GeographicExtent*.

using **GeographicExtentNNPtr** = *util*::nn<*GeographicExtentPtr*>

Non-null shared pointer of *GeographicExtent*.

using **GeographicBoundingBoxPtr** = std::shared_ptr<*GeographicBoundingBox*>

Shared pointer of *GeographicBoundingBox*.

using **GeographicBoundingBoxNNPtr** = *util*::nn<*GeographicBoundingBoxPtr*>

Non-null shared pointer of *GeographicBoundingBox*.

using **TemporalExtentPtr** = std::shared_ptr<*TemporalExtent*>

Shared pointer of *TemporalExtent*.

using **TemporalExtentNNPtr** = *util*::nn<*TemporalExtentPtr*>

Non-null shared pointer of *TemporalExtent*.

using **VerticalExtentPtr** = std::shared_ptr<*VerticalExtent*>

Shared pointer of *VerticalExtent*.

using **VerticalExtentNNPtr** = *util*::nn<*VerticalExtentPtr*>

Non-null shared pointer of *VerticalExtent*.

using **IdentifierPtr** = std::shared_ptr<*Identifier*>

Shared pointer of *Identifier*.

using **IdentifierNNPtr** = *util*::nn<*IdentifierPtr*>

Non-null shared pointer of *Identifier*.

using **PositionalAccuracyPtr** = std::shared_ptr<*PositionalAccuracy*>

Shared pointer of *PositionalAccuracy*.

using **PositionalAccuracyNNPtr** = *util*::nn<*PositionalAccuracyPtr*>

Non-null shared pointer of *PositionalAccuracy*.

class **Citation** : public osgeo::proj::*util*::*BaseObject*

*#include <metadata.hpp>* Standardized resource reference.

A citation contains a title.

> **Remark**
>
> Simplified version of Citation from *GeoAPI*

### Public Functions

explicit **Citation**(const std::string &titleIn)

Constructs a citation by its title.

const *util*::*optional*<std::string> &**title**()

Returns the name by which the cited resource is known.

class **GeographicExtent** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*util*::*IComparable*

*#include <metadata.hpp>* Base interface for geographic area of the dataset.

> **Remark**
>
> Simplified version of GeographicExtent from *GeoAPI*

Subclassed by *osgeo::proj::metadata::GeographicBoundingBox*

### Public Functions

virtual bool **contains**(const *GeographicExtentNNPtr* &other) const = 0

Returns whether this extent contains the other one.

virtual bool **intersects**(const *GeographicExtentNNPtr* &other) const = 0

Returns whether this extent intersects the other one.

virtual *GeographicExtentPtr* **intersection**(const *GeographicExtentNNPtr* &other) const = 0

Returns the intersection of this extent with another one.

class **GeographicBoundingBox** : public osgeo::proj::*metadata*::*GeographicExtent*

*#include <metadata.hpp>* Geographic position of the dataset.

This is only an approximate so specifying the coordinate reference system is unnecessary.

> **Remark**
>
> Implements GeographicBoundingBox from *GeoAPI*

### Public Functions

double **westBoundLongitude**()

Returns the western-most coordinate of the limit of the dataset extent.

The unit is degrees.

If eastBoundLongitude < *westBoundLongitude()*, then the bounding box crosses the anti-meridian.

double **southBoundLatitude**()

Returns the southern-most coordinate of the limit of the dataset extent.

The unit is degrees.

double **eastBoundLongitude**()

Returns the eastern-most coordinate of the limit of the dataset extent.

The unit is degrees.

If eastBoundLongitude < *westBoundLongitude()*, then the bounding box crosses the anti-meridian.

double **northBoundLatitude**()

Returns the northern-most coordinate of the limit of the dataset extent.

The unit is degrees.

virtual bool **contains**(const *GeographicExtentNNPtr* &other) const override

Returns whether this extent contains the other one.

virtual bool **intersects**(const *GeographicExtentNNPtr* &other) const override

Returns whether this extent intersects the other one.

virtual *GeographicExtentPtr* **intersection**(const *GeographicExtentNNPtr* &other) const override

Returns the intersection of this extent with another one.

### Public Static Functions

static *GeographicBoundingBoxNNPtr* **create**(double west, double south, double east, double north)

Instantiate a *GeographicBoundingBox*.

If east < west, then the bounding box crosses the anti-meridian.

**Parameters**

- **west** -- Western-most coordinate of the limit of the dataset extent (in degrees).
- **south** -- Southern-most coordinate of the limit of the dataset extent (in degrees).
- **east** -- Eastern-most coordinate of the limit of the dataset extent (in degrees).
- **north** -- Northern-most coordinate of the limit of the dataset extent (in degrees).

**Returns**

a new *GeographicBoundingBox*.

class **TemporalExtent** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*util*::*IComparable*

*#include <metadata.hpp>* Time period covered by the content of the dataset.

**Remark**

Simplified version of TemporalExtent from *GeoAPI*

### Public Functions

const std::string &**start**()

Returns the start of the temporal extent.

const std::string &**stop**()

Returns the end of the temporal extent.

bool **contains**(const *TemporalExtentNNPtr* &other) const

Returns whether this extent contains the other one.

bool **intersects**(const *TemporalExtentNNPtr* &other) const

Returns whether this extent intersects the other one.

### Public Static Functions

static *TemporalExtentNNPtr* **create**(const std::string &start, const std::string &stop)

Instantiate a *TemporalExtent*.

**Parameters**

- **start** -- start.
- **stop** -- stop.

**Returns**

a new *TemporalExtent*.

class **VerticalExtent** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*util*::*IComparable*

*#include <metadata.hpp>* Vertical domain of dataset.

**Remark**

Simplified version of VerticalExtent from *GeoAPI*

### Public Functions

double **minimumValue**()

Returns the minimum of the vertical extent.

double **maximumValue**()

Returns the maximum of the vertical extent.

*common*::*UnitOfMeasureNNPtr* &**unit**()

Returns the unit of the vertical extent.

bool **contains**(const *VerticalExtentNNPtr* &other) const

Returns whether this extent contains the other one.

bool **intersects**(const *VerticalExtentNNPtr* &other) const

Returns whether this extent intersects the other one.

### Public Static Functions

static *VerticalExtentNNPtr* **create**(double minimumValue, double maximumValue, const *common*::*UnitOfMeasureNNPtr* &unitIn)

Instantiate a *VerticalExtent*.

**Parameters**
- **minimumIn** -- minimum.
- **maximumIn** -- maximum.
- **unitIn** -- unit.

**Returns**
a new *VerticalExtent*.

class **Extent** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*util*::*IComparable*

*#include <metadata.hpp>* Information about spatial, vertical, and temporal extent.

**Remark**

Simplified version of Extent from *GeoAPI*

### Public Functions

const *util*::*optional*<std::string> &**description**()

Return a textual description of the extent.

**Returns**
the description, or empty.

const std::vector<*GeographicExtentNNPtr*> &**geographicElements**()

Return the geographic element(s) of the extent

**Returns**
the geographic element(s), or empty.

const std::vector<*TemporalExtentNNPtr*> &**temporalElements**()

Return the temporal element(s) of the extent

**Returns**
the temporal element(s), or empty.

const std::vector<*VerticalExtentNNPtr*> &**verticalElements**()

Return the vertical element(s) of the extent

**Returns**
the vertical element(s), or empty.

bool **contains**(const *ExtentNNPtr* &other) const

Returns whether this extent contains the other one.

Behavior only well specified if each sub-extent category as at most one element.

bool **intersects**(const *ExtentNNPtr* &other) const

Returns whether this extent intersects the other one.

Behavior only well specified if each sub-extent category as at most one element.

*ExtentPtr* **intersection**(const *ExtentNNPtr* &other) const

Returns the intersection of this extent with another one.

Behavior only well specified if there is one single *GeographicExtent* in each object. Returns nullptr otherwise.

### Public Static Functions

static *ExtentNNPtr* **create**(const *util*::*optional*<std::string> &descriptionIn, const std::vector<*GeographicExtentNNPtr*> &geographicElementsIn, const std::vector<*VerticalExtentNNPtr*> &verticalElementsIn, const std::vector<*TemporalExtentNNPtr*> &temporalElementsIn)

Instantiate a *Extent*.

**Parameters**

- **descriptionIn** -- Textual description, or empty.
- **geographicElementsIn** -- Geographic element(s), or empty.
- **verticalElementsIn** -- Vertical element(s), or empty.
- **temporalElementsIn** -- Temporal element(s), or empty.

**Returns**

a new *Extent*.

static *ExtentNNPtr* **createFromBBOX**(double west, double south, double east, double north, const *util*::*optional*<std::string> &descriptionIn = *util*::*optional*<std::string>())

Instantiate a *Extent* from a bounding box.

**Parameters**

- **west** -- Western-most coordinate of the limit of the dataset extent (in degrees).
- **south** -- Southern-most coordinate of the limit of the dataset extent (in degrees).
- **east** -- Eastern-most coordinate of the limit of the dataset extent (in degrees).
- **north** -- Northern-most coordinate of the limit of the dataset extent (in degrees).
- **descriptionIn** -- Textual description, or empty.

**Returns**

a new *Extent*.

### Public Static Attributes

static const *ExtentNNPtr* **WORLD**

World extent.

class **Identifier** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*io*::*IWKTExportable*, public osgeo::proj::*io*::*IJSONExportable*

*#include <metadata.hpp>* Value uniquely identifying an object within a namespace.

**Remark**

Implements *Identifier* as described in *ISO 19111:2019* but which originates from *ISO 19115 (Metadata)*

### Public Functions

const *util*::*optional*<*Citation*> &**authority**()

Return a citation for the organization responsible for definition and maintenance of the code.

**Returns**

the citation for the authority, or empty.

const std::string &**code**()

Return the alphanumeric value identifying an instance in the codespace.

e.g. "4326" (for EPSG:4326 WGS 84 GeographicCRS)

**Returns**
the code.

const *util*::*optional*<std::string> &**codeSpace**()

Return the organization responsible for definition and maintenance of the code.

e.g "EPSG"

**Returns**
the authority codespace, or empty.

const *util*::*optional*<std::string> &**version**()

Return the version identifier for the namespace.

When appropriate, the edition is identified by the effective date, coded using ISO 8601 date format.

**Returns**
the version or empty.

const *util*::*optional*<std::string> &**description**()

Return the natural language description of the meaning of the code value.

**Returns**
the description or empty.

const *util*::*optional*<std::string> &**uri**()

Return the URI of the identifier.

**Returns**
the URI or empty.

### Public Static Functions

static *IdentifierNNPtr* **create**(const std::string &codeIn = std::string(), const *util*::*PropertyMap* &properties = *util*::*PropertyMap*())

Instantiate a *Identifier*.

**Parameters**
- **codeIn** -- Alphanumeric value identifying an instance in the codespace
- **properties** -- See *General properties*. Generally, the *Identifier::CODESPACE_KEY* should be set.

**Returns**
a new *Identifier*.

static bool **isEquivalentName**(const char *a, const char *b) noexcept

Returns whether two names are considered equivalent.

Two names are equivalent by removing any space, underscore, dash, slash, { or } character from them, and comparing in a case insensitive way.

static bool **isEquivalentName**(const char *a, const char *b, bool biggerDifferencesAllowed) noexcept

Returns whether two names are considered equivalent.

Two names are equivalent by removing any space, underscore, dash, slash, { or } character from them, and comparing in a case insensitive way.

**Since**
9.6

**Parameters**
- **a** -- first string
- **b** -- second string

- **biggerDifferencesAllowed** -- if true, "height" and "zone" words are ignored, and "north" is shortened as "n" and "south" as "n".

### Public Static Attributes

static const std::string **AUTHORITY_KEY**

Key to set the authority citation of a *metadata::Identifier*.

The value is to be provided as a string or a *metadata::Citation*.

static const std::string **CODE_KEY**

Key to set the code of a *metadata::Identifier*.

The value is to be provided as a integer or a string.

static const std::string **CODESPACE_KEY**

Key to set the organization responsible for definition and maintenance of the code of a *metadata::Identifier*.

The value is to be provided as a string.

static const std::string **VERSION_KEY**

Key to set the version identifier for the namespace of a *metadata::Identifier*.

The value is to be provided as a string.

static const std::string **DESCRIPTION_KEY**

Key to set the natural language description of the meaning of the code value of a *metadata::Identifier*.

The value is to be provided as a string.

static const std::string **URI_KEY**

Key to set the URI of a *metadata::Identifier*.

The value is to be provided as a string.

static const std::string **EPSG**

EPSG codespace.

static const std::string **OGC**

OGC codespace.

class **PositionalAccuracy** : public osgeo::proj::*util*::*BaseObject*

*#include <metadata.hpp>* Accuracy of the position of features.

**Remark**

Simplified version of PositionalAccuracy from *GeoAPI*, which originates from *ISO 19115 (Metadata)*

#### Public Functions

const std::string &**value**()

Return the value of the positional accuracy.

#### Public Static Functions

static *PositionalAccuracyNNPtr* **create**(const std::string &valueIn)

Instantiate a *PositionalAccuracy*.

**Parameters**

**valueIn** -- positional accuracy value.

**Returns**

a new *PositionalAccuracy*.

### 10.5.4.5 cs namespace

namespace **cs**

Coordinate systems and their axis.

*osgeo.proj.cs* namespace

#### Typedefs

using **MeridianPtr** = std::shared_ptr<*Meridian*>

Shared pointer of *Meridian*.

using **MeridianNNPtr** = *util*::nn<*MeridianPtr*>

Non-null shared pointer of *Meridian*.

using **CoordinateSystemAxisPtr** = std::shared_ptr<*CoordinateSystemAxis*>

Shared pointer of *CoordinateSystemAxis*.

using **CoordinateSystemAxisNNPtr** = *util*::nn<*CoordinateSystemAxisPtr*>

Non-null shared pointer of *CoordinateSystemAxis*.

typedef std::shared_ptr<*CoordinateSystem*> **CoordinateSystemPtr**

Shared pointer of *CoordinateSystem*.

typedef *util*::nn<*CoordinateSystemPtr*> **CoordinateSystemNNPtr**

Non-null shared pointer of *CoordinateSystem*.

using **SphericalCSPtr** = std::shared_ptr<*SphericalCS*>

Shared pointer of *SphericalCS*.

using **SphericalCSNNPtr** = *util*::nn<*SphericalCSPtr*>

Non-null shared pointer of *SphericalCS*.

using **EllipsoidalCSPtr** = std::shared_ptr<*EllipsoidalCS*>

Shared pointer of *EllipsoidalCS*.

using **EllipsoidalCSNNPtr** = *util*::nn<*EllipsoidalCSPtr*>

Non-null shared pointer of *EllipsoidalCS*.

using **VerticalCSPtr** = std::shared_ptr<*VerticalCS*>

Shared pointer of *VerticalCS*.

using **VerticalCSNNPtr** = *util*::nn<*VerticalCSPtr*>

Non-null shared pointer of *VerticalCS*.

using **CartesianCSPtr** = std::shared_ptr<*CartesianCS*>

Shared pointer of *CartesianCS*.

using **CartesianCSNNPtr** = *util*::nn<*CartesianCSPtr*>

Non-null shared pointer of *CartesianCS*.

using **AffineCSPtr** = std::shared_ptr<*AffineCS*>

Shared pointer of *AffineCS*.

using **AffineCSNNPtr** = *util*::nn<*AffineCSPtr*>

Non-null shared pointer of *AffineCS*.

using **OrdinalCSPtr** = std::shared_ptr<*OrdinalCS*>

Shared pointer of *OrdinalCS*.

using **OrdinalCSNNPtr** = *util*::nn<*OrdinalCSPtr*>

Non-null shared pointer of *OrdinalCS*.

using **ParametricCSPtr** = std::shared_ptr<*ParametricCS*>

Shared pointer of *ParametricCS*.

using **ParametricCSNNPtr** = *util*::nn<*ParametricCSPtr*>

Non-null shared pointer of *ParametricCS*.

using **TemporalCSPtr** = std::shared_ptr<*TemporalCS*>

Shared pointer of *TemporalCS*.

using **TemporalCSNNPtr** = *util*::nn<*TemporalCSPtr*>

Non-null shared pointer of *TemporalCS*.

using **DateTimeTemporalCSPtr** = std::shared_ptr<*DateTimeTemporalCS*>

Shared pointer of *DateTimeTemporalCS*.

using **DateTimeTemporalCSNNPtr** = *util*::nn<*DateTimeTemporalCSPtr*>

Non-null shared pointer of *DateTimeTemporalCS*.

using **TemporalCountCSPtr** = std::shared_ptr<*TemporalCountCS*>

Shared pointer of *TemporalCountCS*.

using **TemporalCountCSNNPtr** = *util*::nn<*TemporalCountCSPtr*>

Non-null shared pointer of *TemporalCountCS*.

using **TemporalMeasureCSPtr** = std::shared_ptr<*TemporalMeasureCS*>

Shared pointer of *TemporalMeasureCS*.

using **TemporalMeasureCSNNPtr** = *util*::nn<*TemporalMeasureCSPtr*>

Non-null shared pointer of *TemporalMeasureCS*.

class **AxisDirection** : public osgeo::proj::*util*::*CodeList*

*#include <coordinatesystem.hpp>* The direction of positive increase in the coordinate value for a coordinate system axis.

**Remark**

Implements *AxisDirection* from *ISO 19111:2019*

### Public Static Attributes

static const *AxisDirection* **NORTH**

Axis positive direction is north. In a geodetic or projected CRS, north is defined through the geodetic reference frame. In an engineering CRS, north may be defined with respect to an engineering object rather than a geographical direction.

static const *AxisDirection* **NORTH_NORTH_EAST**

Axis positive direction is approximately north-north-east.

static const *AxisDirection* **NORTH_EAST**

Axis positive direction is approximately north-east.

static const *AxisDirection* **EAST_NORTH_EAST**

Axis positive direction is approximately east-north-east.

static const *AxisDirection* **EAST**

Axis positive direction is 90deg clockwise from north.

static const *AxisDirection* **EAST_SOUTH_EAST**

Axis positive direction is approximately east-south-east.

static const *AxisDirection* **SOUTH_EAST**

Axis positive direction is approximately south-east.

static const *AxisDirection* **SOUTH_SOUTH_EAST**

Axis positive direction is approximately south-south-east.

static const *AxisDirection* **SOUTH**

Axis positive direction is 180deg clockwise from north.

static const *AxisDirection* **SOUTH_SOUTH_WEST**

Axis positive direction is approximately south-south-west.

static const *AxisDirection* **SOUTH_WEST**

Axis positive direction is approximately south-west.

static const *AxisDirection* **WEST_SOUTH_WEST**

Axis positive direction is approximately west-south-west.

static const *AxisDirection* **WEST**

Axis positive direction is 270deg clockwise from north.

static const *AxisDirection* **WEST_NORTH_WEST**

Axis positive direction is approximately west-north-west.

static const *AxisDirection* **NORTH_WEST**

Axis positive direction is approximately north-west.

static const *AxisDirection* **NORTH_NORTH_WEST**

Axis positive direction is approximately north-north-west.

static const *AxisDirection* **UP**

Axis positive direction is up relative to gravity.

static const *AxisDirection* **DOWN**

Axis positive direction is down relative to gravity.

static const *AxisDirection* **GEOCENTRIC_X**

Axis positive direction is in the equatorial plane from the centre of the modelled Earth towards the intersection of the equator with the prime meridian.

static const *AxisDirection* **GEOCENTRIC_Y**

Axis positive direction is in the equatorial plane from the centre of the modelled Earth towards the intersection of the equator and the meridian 90deg eastwards from the prime meridian.

static const *AxisDirection* **GEOCENTRIC_Z**

Axis positive direction is from the centre of the modelled Earth parallel to its rotation axis and towards its north pole.

static const *AxisDirection* **COLUMN_POSITIVE**

Axis positive direction is towards higher pixel column.

static const *AxisDirection* **COLUMN_NEGATIVE**

Axis positive direction is towards lower pixel column.

static const *AxisDirection* **ROW_POSITIVE**

Axis positive direction is towards higher pixel row.

static const *AxisDirection* **ROW_NEGATIVE**

Axis positive direction is towards lower pixel row.

static const *AxisDirection* **DISPLAY_RIGHT**

Axis positive direction is right in display.

static const *AxisDirection* **DISPLAY_LEFT**

Axis positive direction is left in display.

static const *AxisDirection* **DISPLAY_UP**

Axis positive direction is towards top of approximately vertical display surface.

static const *AxisDirection* **DISPLAY_DOWN**

Axis positive direction is towards bottom of approximately vertical display surface.

static const *AxisDirection* **FORWARD**

Axis positive direction is forward; for an observer at the centre of the object this is will be towards its front, bow or nose.

static const *AxisDirection* **AFT**

Axis positive direction is aft; for an observer at the centre of the object this will be towards its back, stern or tail.

static const *AxisDirection* **PORT**

Axis positive direction is port; for an observer at the centre of the object this will be towards its left.

static const *AxisDirection* **STARBOARD**

Axis positive direction is starboard; for an observer at the centre of the object this will be towards its right.

static const *AxisDirection* **CLOCKWISE**

Axis positive direction is clockwise from a specified direction.

static const *AxisDirection* **COUNTER_CLOCKWISE**

Axis positive direction is counter clockwise from a specified direction.

static const *AxisDirection* **TOWARDS**

Axis positive direction is towards the object.

static const *AxisDirection* **AWAY_FROM**

Axis positive direction is away from the object.

static const *AxisDirection* **FUTURE**

Temporal axis positive direction is towards the future.

static const *AxisDirection* **PAST**

Temporal axis positive direction is towards the past.

static const *AxisDirection* **UNSPECIFIED**

Axis positive direction is unspecified.

class **RangeMeaning** : public osgeo::proj::*util*::*CodeList*

*#include <coordinatesystem.hpp>* Meaning of the axis value range specified through minimumValue and maximumValue.

**Remark**

Implements *RangeMeaning* from *ISO 19111:2019*

**Since**
9.2

### Public Static Attributes

static const *RangeMeaning* **EXACT**

any value between and including minimumValue and maximumValue is valid.

static const *RangeMeaning* **WRAPAROUND**

Axis is continuous with values wrapping around at the minimumValue and maximumValue

class **Meridian** : public osgeo::proj::*common*::*IdentifiedObject*, public osgeo::proj::*io*::*IJSONExportable*

*#include <coordinatesystem.hpp>* The meridian that the axis follows from the pole, for a coordinate reference system centered on a pole.

**Remark**

Implements MERIDIAN from *WKT2 standard*

**Note**

There is no modelling for this concept in *ISO 19111:2019*

### Public Functions

const *common*::*Angle* &**longitude**()

Return the longitude of the meridian that the axis follows from the pole.

**Returns**

the longitude.

### Public Static Functions

static *MeridianNNPtr* **create**(const *common*::*Angle* &longitudeIn)

Instantiate a *Meridian*.

**Parameters**

**longitudeIn** -- longitude of the meridian that the axis follows from the pole.

**Returns**

new *Meridian*.

class **CoordinateSystemAxis** : public osgeo::proj::*common*::*IdentifiedObject*, public osgeo::proj::*io*::*IJSONExportable*

*#include <coordinatesystem.hpp>* The definition of a coordinate system axis.

**Remark**

Implements *CoordinateSystemAxis* from *ISO 19111:2019*

### Public Functions

const std::string &**abbreviation**()

Return the axis abbreviation.

The abbreviation used for this coordinate system axis; this abbreviation is also used to identify the coordinates in the coordinate tuple. Examples are X and Y.

**Returns**

the abbreviation.

const *AxisDirection* &**direction**()

Return the axis direction.

The direction of this coordinate system axis (or in the case of Cartesian projected coordinates, the direction of this coordinate system axis locally) Examples: north or south, east or west, up or down. Within any set of coordinate system axes, only one of each pair of terms can be used. For Earth-fixed CRSs, this direction is often approximate and intended to provide a human interpretable meaning to the axis. When a geodetic reference frame is used, the precise directions of the axes may therefore vary slightly from this approximate direction. Note that an EngineeringCRS often requires specific descriptions of the directions of its coordinate system axes.

**Returns**

the direction.

const *common*::*UnitOfMeasure* &**unit**()

Return the axis unit.

This is the spatial unit or temporal quantity used for this coordinate system axis. The value of a coordinate in a coordinate tuple shall be recorded using this unit.

**Returns**
the axis unit.

const *util*::*optional*<double> &**minimumValue**()

Return the minimum value normally allowed for this axis, in the unit for the axis.

**Returns**
the minimum value, or empty.

const *util*::*optional*<double> &**maximumValue**()

Return the maximum value normally allowed for this axis, in the unit for the axis.

**Returns**
the maximum value, or empty.

const *util*::*optional*<*RangeMeaning*> &**rangeMeaning**()

Return the range meaning.

**Since**
9.2

**Returns**
the range meaning, or empty.

const *MeridianPtr* &**meridian**()

Return the meridian that the axis follows from the pole, for a coordinate reference system centered on a pole.

**Returns**
the meridian, or null.

### Public Static Functions

static *CoordinateSystemAxisNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::string &abbreviationIn, const *AxisDirection* &directionIn, const *common*::*UnitOfMeasure* &unitIn, const *MeridianPtr* &meridianIn = nullptr)

Instantiate a *CoordinateSystemAxis*.

**Parameters**
- **properties** -- See *General properties*. The name should generally be defined.
- **abbreviationIn** -- Axis abbreviation (might be empty)
- **directionIn** -- Axis direction
- **unitIn** -- Axis unit
- **meridianIn** -- The meridian that the axis follows from the pole, for a coordinate reference system centered on a pole, or nullptr

**Returns**
a new *CoordinateSystemAxis*.

static *CoordinateSystemAxisNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::string &abbreviationIn, const *AxisDirection* &directionIn, const *common*::*UnitOfMeasure* &unitIn, const *util*::*optional*<double> &minimumValueIn, const *util*::*optional*<double> &maximumValueIn, const *util*::*optional*<*RangeMeaning*> &rangeMeaningIn, const *MeridianPtr* &meridianIn = nullptr)

Instantiate a *CoordinateSystemAxis*.

**Since**
9.2

**Parameters**
- **properties** -- See *General properties*. The name should generally be defined.
- **abbreviationIn** -- Axis abbreviation (might be empty)
- **directionIn** -- Axis direction
- **unitIn** -- Axis unit
- **minimumValueIn** -- Minimum value along axis
- **maximumValueIn** -- Maximum value along axis
- **rangeMeaningIn** -- Range Meaning
- **meridianIn** -- The meridian that the axis follows from the pole, for a coordinate reference system centered on a pole, or nullptr

**Returns**
a new *CoordinateSystemAxis*.

class **CoordinateSystem** : public osgeo::proj::*common*::*IdentifiedObject*, public osgeo::proj::*io*::*IJSONExportable*

*#include <coordinatesystem.hpp>* Abstract class modelling a coordinate system (CS)

A CS is the non-repeating sequence of coordinate system axes that spans a given coordinate space. A CS is derived from a set of mathematical rules for specifying how coordinates in a given space are to be assigned to points. The coordinate values in a coordinate tuple shall be recorded in the order in which the coordinate system axes associations are recorded.

**Remark**

Implements *CoordinateSystem* from *ISO 19111:2019*

Subclassed by *osgeo::proj::cs::AffineCS*, *osgeo::proj::cs::CartesianCS*, *osgeo::proj::cs::EllipsoidalCS*, *osgeo::proj::cs::OrdinalCS*, *osgeo::proj::cs::ParametricCS*, *osgeo::proj::cs::SphericalCS*, *osgeo::proj::cs::TemporalCS*, *osgeo::proj::cs::VerticalCS*

### Public Functions

const std::vector<*CoordinateSystemAxisNNPtr*> &**axisList**()

Return the list of axes of this coordinate system.

**Returns**
the axes.

class **SphericalCS** : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* A three-dimensional coordinate system in Euclidean space with one distance measured from the origin and two angular coordinates.

Not to be confused with an ellipsoidal coordinate system based on an ellipsoid "degenerated" into a sphere. A *SphericalCS* shall have three axis associations.

**Remark**

Implements *SphericalCS* from *ISO 19111:2019*

### Public Static Functions

static *SphericalCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2, const *CoordinateSystemAxisNNPtr* &axis3)

Instantiate a *SphericalCS*.

**Parameters**

- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.
- **axis3** -- The third axis.

**Returns**

a new *SphericalCS*.

static *SphericalCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2)

Instantiate a *SphericalCS* with 2 axis.

This is an extension to ISO19111 to support (planet)-ocentric CS with geocentric latitude.

**Parameters**

- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.

**Returns**

a new *SphericalCS*.

### Public Static Attributes

static constexpr const char ***WKT2_TYPE** = "spherical"

Value of getWKT2Type()

class **EllipsoidalCS** : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* A two- or three-dimensional coordinate system in which position is specified by geodetic latitude, geodetic longitude, and (in the three-dimensional case) ellipsoidal height.

An *EllipsoidalCS* shall have two or three associations.

**Remark**

Implements *EllipsoidalCS* from *ISO 19111:2019*

### Public Static Functions

static *EllipsoidalCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2)

Instantiate a *EllipsoidalCS*.

**Parameters**

- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.

**Returns**

a new *EllipsoidalCS*.

static *EllipsoidalCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2, const *CoordinateSystemAxisNNPtr* &axis3)

Instantiate a *EllipsoidalCS*.

**Parameters**

- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.
- **axis3** -- The third axis.

**Returns**

a new *EllipsoidalCS*.

static *EllipsoidalCSNNPtr* **createLatitudeLongitude**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *EllipsoidalCS* with a Latitude (first) and Longitude (second) axis.

**Parameters**

**unit** -- Angular unit of the axes.

**Returns**

a new *EllipsoidalCS*.

static *EllipsoidalCSNNPtr* **createLatitudeLongitudeEllipsoidalHeight**(const *common*::*UnitOfMeasure* &angularUnit, const *common*::*UnitOfMeasure* &linearUnit)

Instantiate a *EllipsoidalCS* with a Latitude (first), Longitude (second) axis and ellipsoidal height (third) axis.

**Parameters**

- **angularUnit** -- Angular unit of the latitude and longitude axes.
- **linearUnit** -- Linear unit of the ellipsoidal height axis.

**Returns**

a new *EllipsoidalCS*.

static *EllipsoidalCSNNPtr* **createLongitudeLatitude**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *EllipsoidalCS* with a Longitude (first) and Latitude (second) axis.

**Parameters**

**unit** -- Angular unit of the axes.

**Returns**

a new *EllipsoidalCS*.

static *EllipsoidalCSNNPtr* **createLongitudeLatitudeEllipsoidalHeight**(const *common*::*UnitOfMeasure* &angularUnit, const *common*::*UnitOfMeasure* &linearUnit)

Instantiate a *EllipsoidalCS* with a Longitude (first), Latitude (second) axis and ellipsoidal height (third) axis.

**Since**
7.0

**Parameters**
- **angularUnit** -- Angular unit of the latitude and longitude axes.
- **linearUnit** -- Linear unit of the ellipsoidal height axis.

**Returns**
a new *EllipsoidalCS*.

### Public Static Attributes

static constexpr const char ***WKT2_TYPE** = "ellipsoidal"

Value of getWKT2Type()

class **VerticalCS** : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* A one-dimensional coordinate system used to record the heights or depths of points.

Such a coordinate system is usually dependent on the Earth's gravity field. A *VerticalCS* shall have one axis association.

**Remark**

Implements *VerticalCS* from *ISO 19111:2019*

### Public Static Functions

static *VerticalCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis)

Instantiate a *VerticalCS*.

**Parameters**
- **properties** -- See *General properties*.
- **axis** -- The axis.

**Returns**
a new *VerticalCS*.

static *VerticalCSNNPtr* **createGravityRelatedHeight**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *VerticalCS* with a Gravity-related height axis.

**Parameters**
**unit** -- linear unit.

**Returns**
a new *VerticalCS*.

### Public Static Attributes

static constexpr const char *__WKT2_TYPE__ = "vertical"

Value of getWKT2Type()

class **CartesianCS** : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* A two- or three-dimensional coordinate system in Euclidean space with orthogonal straight axes.

All axes shall have the same length unit. A *CartesianCS* shall have two or three axis associations; the number of associations shall equal the dimension of the CS.

**Remark**

Implements *CartesianCS* from *ISO 19111:2019*

### Public Static Functions

static *CartesianCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2)

Instantiate a *CartesianCS*.

**Parameters**

- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.

**Returns**

a new *CartesianCS*.

static *CartesianCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2, const *CoordinateSystemAxisNNPtr* &axis3)

Instantiate a *CartesianCS*.

**Parameters**

- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.
- **axis3** -- The third axis.

**Returns**

a new *CartesianCS*.

static *CartesianCSNNPtr* **createEastingNorthing**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *CartesianCS* with a Easting (first) and Northing (second) axis.

**Parameters**

**unit** -- Linear unit of the axes.

**Returns**

a new *CartesianCS*.

static *CartesianCSNNPtr* **createNorthingEasting**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *CartesianCS* with a Northing (first) and Easting (second) axis.

**Parameters**
**unit** -- Linear unit of the axes.
**Returns**
a new *CartesianCS*.

static *CartesianCSNNPtr* **createNorthPoleEastingSouthNorthingSouth**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *CartesianCS*, north-pole centered, with a Easting (first) South-Oriented and Northing (second) South-Oriented axis.
**Parameters**
**unit** -- Linear unit of the axes.
**Returns**
a new *CartesianCS*.

static *CartesianCSNNPtr* **createSouthPoleEastingNorthNorthingNorth**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *CartesianCS*, south-pole centered, with a Easting (first) North-Oriented and Northing (second) North-Oriented axis.
**Parameters**
**unit** -- Linear unit of the axes.
**Returns**
a new *CartesianCS*.

static *CartesianCSNNPtr* **createWestingSouthing**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *CartesianCS* with a Westing (first) and Southing (second) axis.
**Parameters**
**unit** -- Linear unit of the axes.
**Returns**
a new *CartesianCS*.

static *CartesianCSNNPtr* **createGeocentric**(const *common*::*UnitOfMeasure* &unit)

Instantiate a *CartesianCS* with the three geocentric axes.
**Parameters**
**unit** -- Linear unit of the axes.
**Returns**
a new *CartesianCS*.

### Public Static Attributes

static constexpr const char ***WKT2_TYPE** = "Cartesian"

Value of getWKT2Type()

class **AffineCS** : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* A two- or three-dimensional coordinate system in Euclidean space with straight axes that are not necessarily orthogonal.

**Remark**

Implements *AffineCS* from *ISO 19111:2019*

**Since**
9.2

### Public Static Functions

static *AffineCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2)

Instantiate a *AffineCS*.

**Parameters**
- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.

**Returns**
a new *AffineCS*.

static *AffineCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis1, const *CoordinateSystemAxisNNPtr* &axis2, const *CoordinateSystemAxisNNPtr* &axis3)

Instantiate a *AffineCS*.

**Parameters**
- **properties** -- See *General properties*.
- **axis1** -- The first axis.
- **axis2** -- The second axis.
- **axis3** -- The third axis.

**Returns**
a new *AffineCS*.

### Public Static Attributes

static constexpr const char ***WKT2_TYPE** = "affine"

Value of getWKT2Type()

class **OrdinalCS** : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* n-dimensional coordinate system in which every axis uses integers.

The number of associations shall equal the dimension of the CS.

**Remark**

Implements *OrdinalCS* from *ISO 19111:2019*

### Public Static Functions

static *OrdinalCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::vector<*CoordinateSystemAxisNNPtr*> &axisIn)

Instantiate a *OrdinalCS*.

**Parameters**
- **properties** -- See *General properties*.
- **axisIn** -- List of axis.

**Returns**
a new *OrdinalCS*.

### Public Static Attributes

static constexpr const char *__WKT2_TYPE__ = "ordinal"

Value of getWKT2Type()

class __ParametricCS__ : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* one-dimensional coordinate reference system which uses parameter values or functions that may vary monotonically with height.

**Remark**

Implements *ParametricCS* from *ISO 19111:2019*

### Public Static Functions

static *ParametricCSNNPtr* __create__(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axisIn)

Instantiate a *ParametricCS*.

**Parameters**

- __properties__ -- See *General properties*.
- __axisIn__ -- Axis.

**Returns**

a new *ParametricCS*.

### Public Static Attributes

static constexpr const char *__WKT2_TYPE__ = "parametric"

Value of getWKT2Type()

class __TemporalCS__ : public osgeo::proj::*cs*::*CoordinateSystem*

*#include <coordinatesystem.hpp>* (Abstract class) A one-dimensional coordinate system used to record time.

A *TemporalCS* shall have one axis association.

**Remark**

Implements *TemporalCS* from *ISO 19111:2019*

Subclassed by *osgeo::proj::cs::DateTimeTemporalCS*, *osgeo::proj::cs::TemporalCountCS*, *osgeo::proj::cs::TemporalMeasureCS*

### Public Static Attributes

static constexpr const char *__WKT2_2015_TYPE__ = "temporal"

WKT2:2015 type

class **DateTimeTemporalCS** : public osgeo::proj::*cs*::*TemporalCS*

*#include <coordinatesystem.hpp>* A one-dimensional coordinate system used to record time in dateTime representation as defined in ISO 8601.

A *DateTimeTemporalCS* shall have one axis association. It does not use axisUnitID; the temporal quantities are defined through the ISO 8601 representation.

**Remark**

Implements *DateTimeTemporalCS* from *ISO 19111:2019*

### Public Static Functions

static *DateTimeTemporalCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis)

Instantiate a *DateTimeTemporalCS*.

**Parameters**

- **properties** -- See *General properties*.
- **axisIn** -- The axis.

**Returns**

a new *DateTimeTemporalCS*.

### Public Static Attributes

static constexpr const char ***WKT2_2019_TYPE** = "TemporalDateTime"

WKT2:2019 type

class **TemporalCountCS** : public osgeo::proj::*cs*::*TemporalCS*

*#include <coordinatesystem.hpp>* A one-dimensional coordinate system used to record time as an integer count.

A *TemporalCountCS* shall have one axis association.

**Remark**

Implements *TemporalCountCS* from *ISO 19111:2019*

### Public Static Functions

static *TemporalCountCSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis)

Instantiate a *TemporalCountCS*.

**Parameters**

- **properties** -- See *General properties*.
- **axisIn** -- The axis.

**Returns**

a new *TemporalCountCS*.

#### Public Static Attributes

static constexpr const char *__WKT2_2019_TYPE__ = "TemporalCount"

WKT2:2019 type

class __TemporalMeasureCS__ : public osgeo::proj::*cs*::*TemporalCS*

*#include <coordinatesystem.hpp>* A one-dimensional coordinate system used to record a time as a real number.

A *TemporalMeasureCS* shall have one axis association.

**Remark**

Implements *TemporalMeasureCS* from *ISO 19111:2019*

#### Public Static Functions

static *TemporalMeasureCSNNPtr* __create__(const *util*::*PropertyMap* &properties, const *CoordinateSystemAxisNNPtr* &axis)

Instantiate a *TemporalMeasureCS*.

**Parameters**

- __properties__ -- See *General properties*.
- __axisIn__ -- The axis.

**Returns**

a new *TemporalMeasureCS*.

#### Public Static Attributes

static constexpr const char *__WKT2_2019_TYPE__ = "TemporalMeasure"

WKT2:2019 type

### 10.5.4.6 datum namespace

namespace __datum__

*Datum* (the relationship of a coordinate system to the body).

*osgeo.proj.datum* namespace

#### Typedefs

typedef std::shared_ptr<*Datum*> __DatumPtr__

Shared pointer of *Datum*

typedef *util*::nn<*DatumPtr*> __DatumNNPtr__

Non-null shared pointer of *Datum*

typedef std::shared_ptr<*DatumEnsemble*> __DatumEnsemblePtr__

Shared pointer of *DatumEnsemble*

typedef *util*::nn<*DatumEnsemblePtr*> **DatumEnsembleNNPtr**

Non-null shared pointer of *DatumEnsemble*

typedef std::shared_ptr<*PrimeMeridian*> **PrimeMeridianPtr**

Shared pointer of *PrimeMeridian*

typedef *util*::nn<*PrimeMeridianPtr*> **PrimeMeridianNNPtr**

Non-null shared pointer of *PrimeMeridian*

typedef std::shared_ptr<*Ellipsoid*> **EllipsoidPtr**

Shared pointer of *Ellipsoid*

typedef *util*::nn<*EllipsoidPtr*> **EllipsoidNNPtr**

Non-null shared pointer of *Ellipsoid*

typedef std::shared_ptr<*GeodeticReferenceFrame*> **GeodeticReferenceFramePtr**

Shared pointer of *GeodeticReferenceFrame*

typedef *util*::nn<*GeodeticReferenceFramePtr*> **GeodeticReferenceFrameNNPtr**

Non-null shared pointer of *GeodeticReferenceFrame*

using **DynamicGeodeticReferenceFramePtr** = std::shared_ptr<*DynamicGeodeticReferenceFrame*>

Shared pointer of *DynamicGeodeticReferenceFrame*

using **DynamicGeodeticReferenceFrameNNPtr** = *util*::nn<*DynamicGeodeticReferenceFramePtr*>

Non-null shared pointer of *DynamicGeodeticReferenceFrame*

typedef std::shared_ptr<*VerticalReferenceFrame*> **VerticalReferenceFramePtr**

Shared pointer of *VerticalReferenceFrame*

typedef *util*::nn<*VerticalReferenceFramePtr*> **VerticalReferenceFrameNNPtr**

Non-null shared pointer of *VerticalReferenceFrame*

using **DynamicVerticalReferenceFramePtr** = std::shared_ptr<*DynamicVerticalReferenceFrame*>

Shared pointer of *DynamicVerticalReferenceFrame*

using **DynamicVerticalReferenceFrameNNPtr** = *util*::nn<*DynamicVerticalReferenceFramePtr*>

Non-null shared pointer of *DynamicVerticalReferenceFrame*

using **TemporalDatumPtr** = std::shared_ptr<*TemporalDatum*>

Shared pointer of *TemporalDatum*

using **TemporalDatumNNPtr** = *util*::nn<*TemporalDatumPtr*>

Non-null shared pointer of *TemporalDatum*

typedef std::shared_ptr<*EngineeringDatum*> **EngineeringDatumPtr**

Shared pointer of *EngineeringDatum*

typedef *util*::nn<*EngineeringDatumPtr*> **EngineeringDatumNNPtr**

Non-null shared pointer of *EngineeringDatum*

using **ParametricDatumPtr** = std::shared_ptr<*ParametricDatum*>

Shared pointer of *ParametricDatum*

using **ParametricDatumNNPtr** = *util*::nn<*ParametricDatumPtr*>

Non-null shared pointer of *ParametricDatum*

class **Datum** : public osgeo::proj::*common*::*ObjectUsage*, public osgeo::proj::*io*::*IJSONExportable*

*#include <datum.hpp>* Abstract class of the relationship of a coordinate system to an object, thus creating a coordinate reference system.

For geodetic and vertical coordinate reference systems, it relates a coordinate system to the Earth (or the celestial body considered). With other types of coordinate reference systems, the datum may relate the coordinate system to another physical or virtual object. A datum uses a parameter or set of parameters that determine the location of the origin of the coordinate reference system. Each datum subtype can be associated with only specific types of coordinate reference systems.

**Remark**

Implements *Datum* from *ISO 19111:2019*

Subclassed by *osgeo::proj::datum::EngineeringDatum*, *osgeo::proj::datum::GeodeticReferenceFrame*, *osgeo::proj::datum::ParametricDatum*, *osgeo::proj::datum::TemporalDatum*, *osgeo::proj::datum::VerticalReferenceFrame*

### Public Functions

const *util*::*optional*<std::string> &**anchorDefinition**() const

Return the anchor definition.

A description - possibly including coordinates of an identified point or points - of the relationship used to anchor a coordinate system to the Earth or alternate object.

- For modern geodetic reference frames the anchor may be a set of station coordinates; if the reference frame is dynamic it will also include coordinate velocities. For a traditional geodetic datum, this anchor may be a point known as the fundamental point, which is traditionally the point where the relationship between geoid and ellipsoid is defined, together with a direction from that point.
- For a vertical reference frame the anchor may be the zero level at one or more defined locations or a conventionally defined surface.
- For an engineering datum, the anchor may be an identified physical point with the orientation defined relative to the object.

**Returns**
the anchor definition, or empty.

const *util*::*optional*<*common*::*Measure*> &**anchorEpoch**() const

Return the anchor epoch.

Epoch at which a static reference frame matches a dynamic reference frame from which it has been derived.

Note: Not to be confused with the frame reference epoch of dynamic geodetic and dynamic vertical reference frames. Nor with the epoch at which a reference frame is defined to be aligned with another reference frame; this information should be included in the datum anchor definition.

**Since**
9.2

**Returns**
the anchor epoch, or empty.

const *util*::*optional*<*common*::*DateTime*> &**publicationDate**() const

Return the date on which the datum definition was published.

**Note**

Departure from *ISO 19111:2019* : we return a DateTime instead of a Citation::Date.

**Returns**
the publication date, or empty.

const *common*::*IdentifiedObjectPtr* &**conventionalRS**() const

Return the conventional reference system.

This is the name, identifier, alias and remarks for the terrestrial reference system or vertical reference system realized by this reference frame, for example "ITRS" for ITRF88 through ITRF2008 and ITRF2014, or "EVRS" for EVRF2000 and EVRF2007.

**Returns**
the conventional reference system, or nullptr.

class **DatumEnsemble** : public osgeo::proj::*common*::*ObjectUsage*, public osgeo::proj::*io*::*IJSONExportable*

*#include <datum.hpp>* A collection of two or more geodetic or vertical reference frames (or if not geodetic or vertical reference frame, a collection of two or more datums) which for all but the highest accuracy requirements may be considered to be insignificantly different from each other.

Every frame within the datum ensemble must be a realizations of the same Terrestrial Reference System or Vertical Reference System.

**Remark**

Implements *DatumEnsemble* from *ISO 19111:2019*

### Public Functions

const std::vector<*DatumNNPtr*> &**datums**() const

Return the set of datums which may be considered to be insignificantly different from each other.

**Returns**
the set of datums of the *DatumEnsemble*.

const *metadata*::*PositionalAccuracyNNPtr* &**positionalAccuracy**() const

Return the inaccuracy introduced through use of this collection of datums.

It is an indication of the differences in coordinate values at all points between the various realizations that have been grouped into this datum ensemble.

**Returns**

the accuracy.

### Public Static Functions

static *DatumEnsembleNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::vector<*DatumNNPtr*> &datumsIn, const *metadata*::*PositionalAccuracyNNPtr* &accuracy)

Instantiate a *DatumEnsemble*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **datumsIn** -- Array of at least 2 datums.
- **accuracy** -- Accuracy of the datum ensemble

**Throws**

*util*::*Exception* -- in case of error.

**Returns**

new *DatumEnsemble*.

class **PrimeMeridian** : public osgeo::proj::*common*::*IdentifiedObject*, public osgeo::proj::*io*::*IPROJStringExportable*, public osgeo::proj::*io*::*IJSONExportable*

*#include <datum.hpp>* The origin meridian from which longitude values are determined.

**Remark**

Implements *PrimeMeridian* from *ISO 19111:2019*

**Note**

The default value for prime meridian name is "Greenwich". When the default applies, the value for the longitude shall be 0 (degrees).

### Public Functions

const *common*::*Angle* &**longitude**()

Return the longitude of the prime meridian.

It is measured from the internationally-recognised reference meridian ('Greenwich meridian'), positive eastward. The default value is 0 degrees.

**Returns**

the longitude of the prime meridian.

### Public Static Functions

static *PrimeMeridianNNPtr* **create**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &longitudeIn)

Instantiate a *PrimeMeridian*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **longitudeIn** -- the longitude of the prime meridian.

**Returns**

new *PrimeMeridian*.

### Public Static Attributes

static const *PrimeMeridianNNPtr* **GREENWICH**

The Greenwich *PrimeMeridian*.

static const *PrimeMeridianNNPtr* **REFERENCE_MERIDIAN**

The "Reference Meridian" *PrimeMeridian*.

This is a meridian of longitude 0 to be used with non-Earth bodies.

static const *PrimeMeridianNNPtr* **PARIS**

The Paris *PrimeMeridian*.

class **Ellipsoid** : public osgeo::proj::*common*::*IdentifiedObject*, public osgeo::proj::*io*::*IPROJStringExportable*, public osgeo::proj::*io*::*IJSONExportable*

*#include <datum.hpp>* A geometric figure that can be used to describe the approximate shape of an object.

For the Earth an oblate biaxial ellipsoid is used: in mathematical terms, it is a surface formed by the rotation of an ellipse about its minor axis.

**Remark**

Implements *Ellipsoid* from *ISO 19111:2019*

### Public Functions

const *common*::*Length* &**semiMajorAxis**()

Return the length of the semi-major axis of the ellipsoid.

**Returns**

the semi-major axis.

const *util*::*optional*<*common*::*Scale*> &**inverseFlattening**()

Return the inverse flattening value of the ellipsoid, if the ellipsoid has been defined with this value.

**See also**

computeInverseFlattening() that will always return a valid value of the inverse flattening, whether the ellipsoid has been defined through inverse flattening or semi-minor axis.

**Returns**
the inverse flattening value of the ellipsoid, or empty.

const *util*::*optional*<*common*::*Length*> &**semiMinorAxis**()

Return the length of the semi-minor axis of the ellipsoid, if the ellipsoid has been defined with this value.

**See also**

*computeSemiMinorAxis()* that will always return a valid value of the semi-minor axis, whether the ellipsoid has been defined through inverse flattening or semi-minor axis.

**Returns**
the semi-minor axis of the ellipsoid, or empty.

bool **isSphere**()

Return whether the ellipsoid is spherical.

That is to say is *semiMajorAxis()* == *computeSemiMinorAxis()*.

A sphere is completely defined by the semi-major axis, which is the radius of the sphere.

**Returns**
true if the ellipsoid is spherical.

const *util*::*optional*<*common*::*Length*> &**semiMedianAxis**()

Return the length of the semi-median axis of a triaxial ellipsoid.

This parameter is not required for a biaxial ellipsoid.

**Returns**
the semi-median axis of the ellipsoid, or empty.

double **computedInverseFlattening**()

Return or compute the inverse flattening value of the ellipsoid.

If computed, the inverse flattening is the result of a / (a - b), where a is the semi-major axis and b the semi-minor axis.

**Returns**
the inverse flattening value of the ellipsoid, or 0 for a sphere.

double **squaredEccentricity**()

Return the squared eccentricity of the ellipsoid.

**Returns**
the squared eccentricity, or a negative value if invalid.

*common*::*Length* **computeSemiMinorAxis**() const

Return or compute the length of the semi-minor axis of the ellipsoid.

If computed, the semi-minor axis is the result of a * (1 - 1 / rf) where a is the semi-major axis and rf the reverse/inverse flattening.

**Returns**
the semi-minor axis of the ellipsoid.

const std::string &**celestialBody**()

Return the name of the celestial body on which the ellipsoid refers to.

*EllipsoidNNPtr* **identify**() const

Return a *Ellipsoid* object where some parameters are better identified.

**Returns**

a new *Ellipsoid*.

### Public Static Functions

static *EllipsoidNNPtr* **createSphere**(const *util*::*PropertyMap* &properties, const *common*::*Length* &radius, const std::string &celestialBody = *EARTH*)

Instantiate a *Ellipsoid* as a sphere.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **radius** -- the sphere radius (semi-major axis).
- **celestialBody** -- Name of the celestial body on which the ellipsoid refers to.

**Returns**

new *Ellipsoid*.

static *EllipsoidNNPtr* **createFlattenedSphere**(const *util*::*PropertyMap* &properties, const *common*::*Length* &semiMajorAxisIn, const *common*::*Scale* &invFlattening, const std::string &celestialBody = *EARTH*)

Instantiate a *Ellipsoid* from its inverse/reverse flattening.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **semiMajorAxisIn** -- the semi-major axis.
- **invFlattening** -- the inverse/reverse flattening. If set to 0, this will be considered as a sphere.
- **celestialBody** -- Name of the celestial body on which the ellipsoid refers to.

**Returns**

new *Ellipsoid*.

static *EllipsoidNNPtr* **createTwoAxis**(const *util*::*PropertyMap* &properties, const *common*::*Length* &semiMajorAxisIn, const *common*::*Length* &semiMinorAxisIn, const std::string &celestialBody = *EARTH*)

Instantiate a *Ellipsoid* from the value of its two semi axis.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **semiMajorAxisIn** -- the semi-major axis.
- **semiMinorAxisIn** -- the semi-minor axis.
- **celestialBody** -- Name of the celestial body on which the ellipsoid refers to.

**Returns**

new *Ellipsoid*.

### Public Static Attributes

static const std::string **EARTH**

Earth celestial body.

static const *EllipsoidNNPtr* **CLARKE_1866**

The EPSG:7008 / "Clarke 1866" *Ellipsoid*.

static const *EllipsoidNNPtr* **WGS84**

The EPSG:7030 / "WGS 84" *Ellipsoid*.

static const *EllipsoidNNPtr* **GRS1980**

The EPSG:7019 / "GRS 1980" *Ellipsoid*.

class **GeodeticReferenceFrame** : public osgeo::proj::*datum*::*Datum*

*#include <datum.hpp>* The definition of the position, scale and orientation of a geocentric Cartesian 3D coordinate system relative to the Earth.

It may also identify a defined ellipsoid (or sphere) that approximates the shape of the Earth and which is centred on and aligned to this geocentric coordinate system. Older geodetic datums define the location and orientation of a defined ellipsoid (or sphere) that approximates the shape of the earth.

**Remark**

Implements *GeodeticReferenceFrame* from *ISO 19111:2019*

**Note**

The terminology "Datum" is often used to mean a *GeodeticReferenceFrame*.

**Note**

In *ISO 19111:2007*, this class was called GeodeticDatum.

Subclassed by *osgeo::proj::datum::DynamicGeodeticReferenceFrame*

### Public Functions

const *PrimeMeridianNNPtr* &**primeMeridian**()

Return the *PrimeMeridian* associated with a *GeodeticReferenceFrame*.

**Returns**

the *PrimeMeridian*.

const *EllipsoidNNPtr* &**ellipsoid**()

Return the *Ellipsoid* associated with a *GeodeticReferenceFrame*.

**Note**

The *ISO 19111:2019* modelling allows (but discourages) a *GeodeticReferenceFrame* to not be associated with a *Ellipsoid* in the case where it is used by a geocentric *crs::GeodeticCRS*. We have made the choice of making the ellipsoid specification compulsory.

**Returns**

the *Ellipsoid*.

### Public Static Functions

static *GeodeticReferenceFrameNNPtr* **create**(const *util*::*PropertyMap* &properties, const *EllipsoidNNPtr* &ellipsoid, const *util*::*optional*<std::string> &anchor, const *PrimeMeridianNNPtr* &primeMeridian)

Instantiate a *GeodeticReferenceFrame*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **ellipsoid** -- the *Ellipsoid*.
- **anchor** -- the anchor definition, or empty.
- **primeMeridian** -- the *PrimeMeridian*.

**Returns**

new *GeodeticReferenceFrame*.

static *GeodeticReferenceFrameNNPtr* **create**(const *util*::*PropertyMap* &properties, const *EllipsoidNNPtr* &ellipsoid, const *util*::*optional*<std::string> &anchor, const *util*::*optional*<*common*::*Measure*> &anchorEpoch, const *PrimeMeridianNNPtr* &primeMeridian)

Instantiate a *GeodeticReferenceFrame*.

**Since**

9.2

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **ellipsoid** -- the *Ellipsoid*.
- **anchor** -- the anchor definition, or empty.
- **anchorEpoch** -- the anchor epoch, or empty.
- **primeMeridian** -- the *PrimeMeridian*.

**Returns**

new *GeodeticReferenceFrame*.

### Public Static Attributes

static const *GeodeticReferenceFrameNNPtr* **EPSG_6267**

The EPSG:6267 / "North_American_Datum_1927" *GeodeticReferenceFrame*.

static const *GeodeticReferenceFrameNNPtr* **EPSG_6269**

The EPSG:6269 / "North_American_Datum_1983" *GeodeticReferenceFrame*.

static const *GeodeticReferenceFrameNNPtr* **EPSG_6326**

The EPSG:6326 / "WGS_1984" *GeodeticReferenceFrame*.

class **DynamicGeodeticReferenceFrame** : public osgeo::proj::*datum*::*GeodeticReferenceFrame*

*#include <datum.hpp>* A geodetic reference frame in which some of the parameters describe time evolution of defining station coordinates.

For example defining station coordinates having linear velocities to account for crustal motion.

**Remark**

Implements *DynamicGeodeticReferenceFrame* from *ISO 19111:2019*

### Public Functions

const *common*::*Measure* &**frameReferenceEpoch**() const

Return the epoch to which the coordinates of stations defining the dynamic geodetic reference frame are referenced.

Usually given as a decimal year e.g. 2016.47.

**Returns**

the frame reference epoch.

const *util*::*optional*<std::string> &**deformationModelName**() const

Return the name of the deformation model.

**Note**

This is an extension to the *ISO 19111:2019* modeling, to hold the content of the DYNAMIC.MODEL WKT2 node.

**Returns**

the name of the deformation model.

### Public Static Functions

static *DynamicGeodeticReferenceFrameNNPtr* **create**(const *util*::*PropertyMap* &properties, const *EllipsoidNNPtr* &ellipsoid, const *util*::*optional*<std::string> &anchor, const *PrimeMeridianNNPtr* &primeMeridian, const *common*::*Measure* &frameReferenceEpochIn, const *util*::*optional*<std::string> &deformationModelNameIn)

Instantiate a *DynamicGeodeticReferenceFrame*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **ellipsoid** -- the *Ellipsoid*.
- **anchor** -- the anchor definition, or empty.
- **primeMeridian** -- the *PrimeMeridian*.
- **frameReferenceEpochIn** -- the frame reference epoch.
- **deformationModelNameIn** -- deformation model name, or empty

**Returns**

new *DynamicGeodeticReferenceFrame*.

class **RealizationMethod** : public osgeo::proj::*util*::*CodeList*

*#include <datum.hpp>* The specification of the method by which the vertical reference frame is realized.

**Remark**

Implements *RealizationMethod* from *ISO 19111:2019*

### Public Static Attributes

static const *RealizationMethod* **LEVELLING**

The realization is by adjustment of a levelling network fixed to one or more tide gauges.

static const *RealizationMethod* **GEOID**

The realization is through a geoid height model or a height correction model. This is applied to a specified geodetic CRS.

static const *RealizationMethod* **TIDAL**

The realization is through a tidal model or by tidal predictions.

class **VerticalReferenceFrame** : public osgeo::proj::*datum*::*Datum*

*#include <datum.hpp>* A textual description and/or a set of parameters identifying a particular reference level surface used as a zero-height or zero-depth surface, including its position with respect to the Earth.

**Remark**

Implements *VerticalReferenceFrame* from *ISO 19111:2019*

**Note**

In *ISO 19111:2007*, this class was called VerticalDatum.

Subclassed by *osgeo::proj::datum::DynamicVerticalReferenceFrame*

### Public Functions

const *util*::*optional*<*RealizationMethod*> &**realizationMethod**() const

Return the method through which this vertical reference frame is realized.

**Returns**

the realization method.

### Public Static Functions

static *VerticalReferenceFrameNNPtr* **create**(const *util*::*PropertyMap* &properties, const *util*::*optional*<std::string> &anchor = *util*::*optional*<std::string>(), const *util*::*optional*<*RealizationMethod*> &realizationMethodIn = *util*::*optional*<*RealizationMethod*>())

Instantiate a *VerticalReferenceFrame*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **anchor** -- the anchor definition, or empty.
- **realizationMethodIn** -- the realization method, or empty.

**Returns**
new *VerticalReferenceFrame*.

static *VerticalReferenceFrameNNPtr* **create**(const *util*::*PropertyMap* &properties, const *util*::*optional*<std::string> &anchor, const *util*::*optional*<*common*::*Measure*> &anchorEpoch, const *util*::*optional*<*RealizationMethod*> &realizationMethodIn = *util*::*optional*<*RealizationMethod*>())

Instantiate a *VerticalReferenceFrame*.

**Since**
9.2

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **anchor** -- the anchor definition, or empty.
- **anchorEpoch** -- the anchor epoch, or empty.
- **realizationMethodIn** -- the realization method, or empty.

**Returns**
new *VerticalReferenceFrame*.

class **DynamicVerticalReferenceFrame** : public osgeo::proj::*datum*::*VerticalReferenceFrame*

*#include <datum.hpp>* A vertical reference frame in which some of the defining parameters have time dependency.

For example defining station heights have velocity to account for post-glacial isostatic rebound motion.

**Remark**

Implements *DynamicVerticalReferenceFrame* from *ISO 19111:2019*

### Public Functions

const *common*::*Measure* &**frameReferenceEpoch**() const

Return the epoch to which the coordinates of stations defining the dynamic geodetic reference frame are referenced.

Usually given as a decimal year e.g. 2016.47.

**Returns**
the frame reference epoch.

const *util*::*optional*<std::string> &**deformationModelName**() const

Return the name of the deformation model.

**Note**

This is an extension to the *ISO 19111:2019* modeling, to hold the content of the DYNAMIC.MODEL WKT2 node.

**Returns**
the name of the deformation model.

### Public Static Functions

static *DynamicVerticalReferenceFrameNNPtr* **create**(const *util*::*PropertyMap* &properties, const *util*::*optional*<std::string> &anchor, const *util*::*optional*<*RealizationMethod*> &realizationMethodIn, const *common*::*Measure* &frameReferenceEpochIn, const *util*::*optional*<std::string> &deformationModelNameIn)

Instantiate a *DynamicVerticalReferenceFrame*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **anchor** -- the anchor definition, or empty.
- **realizationMethodIn** -- the realization method, or empty.
- **frameReferenceEpochIn** -- the frame reference epoch.
- **deformationModelNameIn** -- deformation model name, or empty

**Returns**

new *DynamicVerticalReferenceFrame*.

class **TemporalDatum** : public osgeo::proj::*datum*::*Datum*

*#include <datum.hpp>* The definition of the relationship of a temporal coordinate system to an object. The object is normally time on the Earth.

**Remark**

Implements *TemporalDatum* from *ISO 19111:2019*

### Public Functions

const *common*::*DateTime* &**temporalOrigin**() const

Return the date and time to which temporal coordinates are referenced, expressed in conformance with ISO 8601.

**Returns**

the temporal origin.

const std::string &**calendar**() const

Return the calendar to which the temporal origin is referenced.

Default value: *TemporalDatum::CALENDAR_PROLEPTIC_GREGORIAN*.

**Returns**

the calendar.

### Public Static Functions

static *TemporalDatumNNPtr* **create**(const *util*::*PropertyMap* &properties, const *common*::*DateTime* &temporalOriginIn, const std::string &calendarIn)

Instantiate a *TemporalDatum*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **temporalOriginIn** -- the temporal origin into which temporal coordinates are referenced.

- **calendarIn** -- the calendar (generally *TemporalDatum::CALENDAR_PROLEPTIC_GREGORIAN*)

**Returns**

new *TemporalDatum*.

### Public Static Attributes

static const std::string **CALENDAR_PROLEPTIC_GREGORIAN**

The proleptic Gregorian calendar.

class **EngineeringDatum** : public osgeo::proj::*datum*::*Datum*

*#include <datum.hpp>* The definition of the origin and orientation of an engineering coordinate reference system.

**Remark**

Implements *EngineeringDatum* from *ISO 19111:2019*

**Note**

The origin can be fixed with respect to the Earth (such as a defined point at a construction site), or be a defined point on a moving vehicle (such as on a ship or satellite), or a defined point of an image.

### Public Static Functions

static *EngineeringDatumNNPtr* **create**(const *util*::*PropertyMap* &properties, const *util*::*optional*<std::string> &anchor = *util*::*optional*<std::string>())

Instantiate a *EngineeringDatum*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **anchor** -- the anchor definition, or empty.

**Returns**

new *EngineeringDatum*.

class **ParametricDatum** : public osgeo::proj::*datum*::*Datum*

*#include <datum.hpp>* Textual description and/or a set of parameters identifying a particular reference surface used as the origin of a parametric coordinate system, including its position with respect to the Earth.

**Remark**

Implements *ParametricDatum* from *ISO 19111:2019*

#### Public Static Functions

static *ParametricDatumNNPtr* **create**(const *util*::*PropertyMap* &properties, const *util*::*optional*<std::string> &anchor = *util*::*optional*<std::string>())

Instantiate a *ParametricDatum*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **anchor** -- the anchor definition, or empty.

**Returns**

new *ParametricDatum*.

### 10.5.4.7 crs namespace

namespace **crs**

*CRS* (coordinate reference system = coordinate system with a datum).

*osgeo.proj.crs* namespace

#### Typedefs

typedef std::shared_ptr<*CRS*> **CRSPtr**

Shared pointer of *CRS*

typedef *util*::nn<*CRSPtr*> **CRSNNPtr**

Non-null shared pointer of *CRS*

typedef std::shared_ptr<*GeographicCRS*> **GeographicCRSPtr**

Shared pointer of *GeographicCRS*

typedef *util*::nn<*GeographicCRSPtr*> **GeographicCRSNNPtr**

Non-null shared pointer of *GeographicCRS*

typedef std::shared_ptr<*VerticalCRS*> **VerticalCRSPtr**

Shared pointer of *VerticalCRS*

typedef *util*::nn<*VerticalCRSPtr*> **VerticalCRSNNPtr**

Non-null shared pointer of *VerticalCRS*

using **BoundCRSPtr** = std::shared_ptr<*BoundCRS*>

Shared pointer of *BoundCRS*

using **BoundCRSNNPtr** = *util*::nn<*BoundCRSPtr*>

Non-null shared pointer of *BoundCRS*

typedef std::shared_ptr<*CompoundCRS*> **CompoundCRSPtr**

Shared pointer of *CompoundCRS*

typedef *util*::nn<*CompoundCRSPtr*> **CompoundCRSNNPtr**

Non-null shared pointer of *CompoundCRS*

using **SingleCRSPtr** = std::shared_ptr<*SingleCRS*>

Shared pointer of *SingleCRS*

using **SingleCRSNNPtr** = *util*::nn<*SingleCRSPtr*>

Non-null shared pointer of *SingleCRS*

typedef std::shared_ptr<*GeodeticCRS*> **GeodeticCRSPtr**

Shared pointer of *GeodeticCRS*

typedef *util*::nn<*GeodeticCRSPtr*> **GeodeticCRSNNPtr**

Non-null shared pointer of *GeodeticCRS*

using **DerivedCRSPtr** = std::shared_ptr<*DerivedCRS*>

Shared pointer of *DerivedCRS*

using **DerivedCRSNNPtr** = *util*::nn<*DerivedCRSPtr*>

Non-null shared pointer of *DerivedCRS*

typedef std::shared_ptr<*ProjectedCRS*> **ProjectedCRSPtr**

Shared pointer of *ProjectedCRS*

typedef *util*::nn<*ProjectedCRSPtr*> **ProjectedCRSNNPtr**

Non-null shared pointer of *ProjectedCRS*

using **TemporalCRSPtr** = std::shared_ptr<*TemporalCRS*>

Shared pointer of *TemporalCRS*

using **TemporalCRSNNPtr** = *util*::nn<*TemporalCRSPtr*>

Non-null shared pointer of *TemporalCRS*

typedef std::shared_ptr<*EngineeringCRS*> **EngineeringCRSPtr**

Shared pointer of *EngineeringCRS*

typedef *util*::nn<*EngineeringCRSPtr*> **EngineeringCRSNNPtr**

Non-null shared pointer of *EngineeringCRS*

using **ParametricCRSPtr** = std::shared_ptr<*ParametricCRS*>

Shared pointer of *ParametricCRS*

using **ParametricCRSNNPtr** = *util*::nn<*ParametricCRSPtr*>

Non-null shared pointer of *ParametricCRS*

using **DerivedGeodeticCRSPtr** = std::shared_ptr<*DerivedGeodeticCRS*>

Shared pointer of *DerivedGeodeticCRS*

using **DerivedGeodeticCRSNNPtr** = *util*::nn<*DerivedGeodeticCRSPtr*>

Non-null shared pointer of *DerivedGeodeticCRS*

using **DerivedGeographicCRSPtr** = std::shared_ptr<*DerivedGeographicCRS*>

Shared pointer of *DerivedGeographicCRS*

using **DerivedGeographicCRSNNPtr** = *util*::nn<*DerivedGeographicCRSPtr*>

Non-null shared pointer of *DerivedGeographicCRS*

using **DerivedProjectedCRSPtr** = std::shared_ptr<*DerivedProjectedCRS*>

Shared pointer of *DerivedProjectedCRS*

using **DerivedProjectedCRSNNPtr** = *util*::nn<*DerivedProjectedCRSPtr*>

Non-null shared pointer of *DerivedProjectedCRS*

using **DerivedVerticalCRSPtr** = std::shared_ptr<*DerivedVerticalCRS*>

Shared pointer of *DerivedVerticalCRS*

using **DerivedVerticalCRSNNPtr** = *util*::nn<*DerivedVerticalCRSPtr*>

Non-null shared pointer of *DerivedVerticalCRS*

using **DerivedEngineeringCRSPtr** = std::shared_ptr<*DerivedEngineeringCRS*>

Shared pointer of *DerivedEngineeringCRS*

using **DerivedEngineeringCRSNNPtr** = *util*::nn<*DerivedEngineeringCRSPtr*>

Non-null shared pointer of *DerivedEngineeringCRS*

using **DerivedParametricCRSPtr** = std::shared_ptr<*DerivedParametricCRS*>

Shared pointer of *DerivedParametricCRS*

using **DerivedParametricCRSNNPtr** = *util*::nn<*DerivedParametricCRSPtr*>

Non-null shared pointer of *DerivedParametricCRS*

using **DerivedTemporalCRSPtr** = std::shared_ptr<*DerivedTemporalCRS*>

Shared pointer of *DerivedTemporalCRS*

using **DerivedTemporalCRSNNPtr** = *util*::nn<*DerivedTemporalCRSPtr*>

Non-null shared pointer of *DerivedTemporalCRS*

class **CRS** : public osgeo::proj::*common*::*ObjectUsage*, public osgeo::proj::*io*::*IJSONExportable*

*#include <crs.hpp>* Abstract class modelling a coordinate reference system which is usually single but may be compound.

**Remark**

Implements *CRS* from *ISO 19111:2019*

Subclassed by *osgeo::proj::crs::BoundCRS*, *osgeo::proj::crs::CompoundCRS*, *osgeo::proj::crs::SingleCRS*

### Public Functions

bool **isDynamic**(bool considerWGS84AsDynamic = false) const

Return whether a *CRS* is a dynamic *CRS*.

A dynamic *CRS* is a *CRS* that contains a geodetic *CRS* whose geodetic reference frame is dynamic, or a vertical *CRS* whose vertical reference frame is dynamic.

**Since**

9.2

**Parameters**

**considerWGS84AsDynamic** -- set to true to consider the WGS 84 / EPSG:6326 datum ensemble as dynamic.

*GeodeticCRSPtr* **extractGeodeticCRS**() const

Return the *GeodeticCRS* of the *CRS*.

Returns the *GeodeticCRS* contained in a *CRS*. This works currently with input parameters of type *GeodeticCRS* or derived, *ProjectedCRS*, *CompoundCRS* or *BoundCRS*.

**Returns**

a GeodeticCRSPtr, that might be null.

*GeographicCRSPtr* **extractGeographicCRS**() const

Return the *GeographicCRS* of the *CRS*.

Returns the *GeographicCRS* contained in a *CRS*. This works currently with input parameters of type *GeographicCRS* or derived, *ProjectedCRS*, *CompoundCRS* or *BoundCRS*.

**Returns**

a GeographicCRSPtr, that might be null.

*VerticalCRSPtr* **extractVerticalCRS**() const

Return the *VerticalCRS* of the *CRS*.

Returns the *VerticalCRS* contained in a *CRS*. This works currently with input parameters of type *VerticalCRS* or derived, *CompoundCRS* or *BoundCRS*.

**Returns**

a VerticalCRSPtr, that might be null.

*CRSNNPtr* **createBoundCRSToWGS84IfPossible**(const *io*::*DatabaseContextPtr* &dbContext, *operation*::*CoordinateOperationContext*::*IntermediateCRSUse* allowIntermediateCRSUse) const

Returns potentially a *BoundCRS*, with a transformation to EPSG:4326, wrapping this *CRS*.

If no such *BoundCRS* is possible, the object will be returned.

The purpose of this method is to be able to format a PROJ.4 string with a +towgs84 parameter or a WKT1:GDAL string with a TOWGS node.

This method will fetch the *GeographicCRS* of this *CRS* and find a transformation to EPSG:4326 using the domain of the validity of the main *CRS*, and there's only one Helmert transformation.

**Returns**

a *CRS*.

*CRSNNPtr* **stripVerticalComponent**() const

Returns a *CRS* whose coordinate system does not contain a vertical component.

As of PROJ 9.5, this method is an alias of demoteTo2D(std::string(), nullptr), which deals with all potential *CRS* types.

*demoteTo2D()* is a preferred alternative, especially when invoked with a non-null database context, to perform a look-up in the database for already registered 2D *CRS*.

**Returns**

a *CRS*.

const *BoundCRSPtr* &**canonicalBoundCRS**()

Return the *BoundCRS* potentially attached to this *CRS*.

In the case this method is called on a object returned by *BoundCRS::baseCRSWithCanonicalBoundCRS()*, this method will return this *BoundCRS*

**Returns**

a BoundCRSPtr, that might be null.

std::list<std::pair<*CRSNNPtr*, int>> **identify**(const *io*::*AuthorityFactoryPtr* &authorityFactory) const

Identify the *CRS* with reference CRSs.

The candidate CRSs are either hard-coded, or looked in the database when authorityFactory is not null.

Note that the implementation uses a set of heuristics to have a good compromise of successful identifications over execution time. It might miss legitimate matches in some circumstances.

The method returns a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match. The list is sorted by decreasing confidence.

- 100% means that the name of the reference entry perfectly matches the *CRS* name, and both are equivalent. In which case a single result is returned. Note: in the case of a *GeographicCRS* whose axis order is implicit in the input definition (for example ESRI WKT), then axis order is ignored for the purpose of identification. That is the *CRS* built from GEOGCS["GCS_WGS_1984",DATUM["D_WGS_1984",SPHEROID["WGS_1984",6378137.0,298.257223563]], PRIMEM["Greenwich",0.0],UNIT["Degree",0.0174532925199433]] will be identified to EPSG:4326, but will not pass a isEquivalentTo(EPSG_4326, util::IComparable::Criterion::EQUIVALENT) test, but rather isEquivalentTo(EPSG_4326, *util::IComparable::Criterion::EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS*)
- 90% means that *CRS* are equivalent, but the names are not exactly the same.
- 70% means that *CRS* are equivalent), but the names do not match at all.
- 25% means that the *CRS* are not equivalent, but there is some similarity in the names.

Other confidence values may be returned by some specialized implementations.

This is implemented for *GeodeticCRS*, *ProjectedCRS*, *VerticalCRS* and *CompoundCRS*.

**Parameters**

**authorityFactory** -- Authority factory (or null, but degraded functionality)

**Returns**

a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match.

std::list<*CRSNNPtr*> **getNonDeprecated**(const *io*::*DatabaseContextNNPtr* &dbContext) const

Return CRSs that are non-deprecated substitutes for the current *CRS*.

*CRSNNPtr* **promoteTo3D**(const std::string &newName, const *io*::*DatabaseContextPtr* &dbContext) const

Return a variant of this *CRS* "promoted" to a 3D one, if not already the case.

The new axis will be ellipsoidal height, oriented upwards, and with metre units.

**Since**
6.3

**Parameters**
- **newName** -- Name of the new *CRS*. If empty, *nameStr()* will be used.
- **dbContext** -- Database context to look for potentially already registered 3D *CRS*. May be nullptr.

**Returns**
a new *CRS* promoted to 3D, or the current one if already 3D or not applicable.

*CRSNNPtr* **demoteTo2D**(const std::string &newName, const *io*::*DatabaseContextPtr* &dbContext) const

Return a variant of this *CRS* "demoted" to a 2D one, if not already the case.

**Since**
6.3

**Parameters**
- **newName** -- Name of the new *CRS*. If empty, *nameStr()* will be used.
- **dbContext** -- Database context to look for potentially already registered 2D *CRS*. May be nullptr.

**Returns**
a new *CRS* demoted to 2D, or the current one if already 2D or not applicable.

class **SingleCRS** : public osgeo::proj::*crs*::*CRS*

*#include <crs.hpp>* Abstract class modelling a coordinate reference system consisting of one Coordinate System and either one *datum::Datum* or one *datum::DatumEnsemble*.

**Remark**

Implements *SingleCRS* from *ISO 19111:2019*

Subclassed by *osgeo::proj::crs::DerivedCRS*, *osgeo::proj::crs::EngineeringCRS*, *osgeo::proj::crs::GeodeticCRS*, *osgeo::proj::crs::ParametricCRS*, *osgeo::proj::crs::TemporalCRS*, *osgeo::proj::crs::VerticalCRS*

### Public Functions

const *datum*::DatumPtr &**datum**()

Return the *datum::Datum* associated with the *CRS*.

This might be null, in which case *datumEnsemble()* return will not be null.

**Returns**
a Datum that might be null.

const *datum*::DatumEnsemblePtr &**datumEnsemble**()

Return the *datum::DatumEnsemble* associated with the *CRS*.

This might be null, in which case *datum()* return will not be null.

**Returns**
a DatumEnsemble that might be null.

const *cs*::*CoordinateSystemNNPtr* &**coordinateSystem**()

Return the *cs::CoordinateSystem* associated with the *CRS*.

**Returns**
a CoordinateSystem.

class **GeodeticCRS** : public virtual osgeo::proj::*crs*::*SingleCRS*, public osgeo::proj::*io*::*IPROJStringExportable*

*#include <crs.hpp>* A coordinate reference system associated with a geodetic reference frame and a three-dimensional Cartesian or spherical coordinate system.

If the geodetic reference frame is dynamic or if the geodetic *CRS* has an association to a velocity model then the geodetic *CRS* is dynamic, else it is static.

**Remark**

Implements *GeodeticCRS* from *ISO 19111:2019*

Subclassed by *osgeo::proj::crs::DerivedGeodeticCRS*, *osgeo::proj::crs::GeographicCRS*

## Public Functions

const *datum*::GeodeticReferenceFramePtr &**datum**()

Return the *datum::GeodeticReferenceFrame* associated with the *CRS*.

**Returns**
a GeodeticReferenceFrame or null (in which case *datumEnsemble()* should return a non-null pointer.)

const *datum*::PrimeMeridianNNPtr &**primeMeridian**()

Return the PrimeMeridian associated with the GeodeticReferenceFrame or with one of the GeodeticReferenceFrame of the *datumEnsemble()*.

**Returns**
the PrimeMeridian.

const *datum*::EllipsoidNNPtr &**ellipsoid**()

Return the ellipsoid associated with the GeodeticReferenceFrame or with one of the GeodeticReferenceFrame of the *datumEnsemble()*.

**Returns**
the PrimeMeridian.

const std::vector<*operation*::*PointMotionOperationNNPtr*> &**velocityModel**()

Return the velocity model associated with the *CRS*.

**Returns**
a velocity model. might be null.

bool **isGeocentric**()

Return whether the *CRS* is a Cartesian geocentric one.

A geocentric *CRS* is a geodetic *CRS* that has a Cartesian coordinate system with three axis, whose direction is respectively *cs::AxisDirection::GEOCENTRIC_X*, *cs::AxisDirection::GEOCENTRIC_Y* and *cs::AxisDirection::GEOCENTRIC_Z*.

**Returns**
true if the *CRS* is a geocentric *CRS*.

bool **isSphericalPlanetocentric**()

Return whether the *CRS* is a Spherical planetocentric one.

A Spherical planetocentric *CRS* is a geodetic *CRS* that has a spherical (angular) coordinate system with 2 axis, which represent geocentric latitude/ longitude or longitude/geocentric latitude.

Such *CRS* are typically used in use case that apply to non-Earth bodies.

**Since**
8.2

**Returns**
true if the *CRS* is a Spherical planetocentric *CRS*.

std::list<std::pair<*GeodeticCRSNNPtr*, int>> **identify**(const *io*::*AuthorityFactoryPtr* &authorityFactory) const

Identify the *CRS* with reference CRSs.

The candidate CRSs are either hard-coded, or looked in the database when authorityFactory is not null.

Note that the implementation uses a set of heuristics to have a good compromise of successful identifications over execution time. It might miss legitimate matches in some circumstances.

The method returns a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match:

- 100% means that the name of the reference entry perfectly matches the *CRS* name, and both are equivalent. In which case a single result is returned. Note: in the case of a *GeographicCRS* whose axis order is implicit in the input definition (for example ESRI WKT), then axis order is ignored for the purpose of identification. That is the *CRS* built from GEOGCS["GCS_WGS_1984",DATUM["D_WGS_1984",SPHEROID["WGS_1984",6378137.0,298.257223563]], PRIMEM["Greenwich",0.0],UNIT["Degree",0.0174532925199433]] will be identified to EPSG:4326, but will not pass a isEquivalentTo(EPSG_4326, util::IComparable::Criterion::EQUIVALENT) test, but rather isEquivalentTo(EPSG_4326, *util::IComparable::Criterion::EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS*)
- 90% means that *CRS* are equivalent, but the names are not exactly the same.
- 70% means that *CRS* are equivalent (equivalent datum and coordinate system), but the names are not equivalent.
- 60% means that ellipsoid, prime meridian and coordinate systems are equivalent, but the *CRS* and datum names do not match.
- 25% means that the *CRS* are not equivalent, but there is some similarity in the names.

**Parameters**
**authorityFactory** -- Authority factory (or null, but degraded functionality)
**Returns**
a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match.

### Public Static Functions

static *GeodeticCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::GeodeticReferenceFrameNNPtr &datum, const *cs*::SphericalCSNNPtr &cs)

Instantiate a *GeodeticCRS* from a *datum::GeodeticReferenceFrame* and a *cs::SphericalCS*.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **datum** -- The datum of the *CRS*.
- **cs** -- a SphericalCS.

**Returns**
new *GeodeticCRS*.

static *GeodeticCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::GeodeticReferenceFrameNNPtr &datum, const *cs*::CartesianCSNNPtr &cs)

Instantiate a *GeodeticCRS* from a *datum::GeodeticReferenceFrame* and a *cs::CartesianCS*.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **datum** -- The datum of the *CRS*.
- **cs** -- a CartesianCS.

**Returns**
new *GeodeticCRS*.

static *GeodeticCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::GeodeticReferenceFramePtr &datum, const *datum*::DatumEnsemblePtr &datumEnsemble, const *cs*::SphericalCSNNPtr &cs)

Instantiate a *GeodeticCRS* from a *datum::GeodeticReferenceFrame* or *datum::DatumEnsemble* and a *cs::SphericalCS*.

One and only one of datum or datumEnsemble should be set to a non-null value.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **datum** -- The datum of the *CRS*, or nullptr
- **datumEnsemble** -- The datum ensemble of the *CRS*, or nullptr.
- **cs** -- a SphericalCS.

**Returns**
new *GeodeticCRS*.

static *GeodeticCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::GeodeticReferenceFramePtr &datum, const *datum*::DatumEnsemblePtr &datumEnsemble, const *cs*::CartesianCSNNPtr &cs)

Instantiate a *GeodeticCRS* from a *datum::GeodeticReferenceFrame* or *datum::DatumEnsemble* and a *cs::CartesianCS*.

One and only one of datum or datumEnsemble should be set to a non-null value.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **datum** -- The datum of the *CRS*, or nullptr
- **datumEnsemble** -- The datum ensemble of the *CRS*, or nullptr.
- **cs** -- a CartesianCS

**Returns**
new *GeodeticCRS*.

### Public Static Attributes

static const *GeodeticCRSNNPtr* **EPSG_4978**

EPSG:4978 / "WGS 84" Geocentric.

class **GeographicCRS** : public osgeo::proj::*crs*::*GeodeticCRS*

*#include <crs.hpp>* A coordinate reference system associated with a geodetic reference frame and a two- or three-dimensional ellipsoidal coordinate system.

If the geodetic reference frame is dynamic or if the geographic *CRS* has an association to a velocity model then the geodetic *CRS* is dynamic, else it is static.

**Remark**

Implements *GeographicCRS* from *ISO 19111:2019*

Subclassed by *osgeo::proj::crs::DerivedGeographicCRS*

### Public Functions

const *cs*::*EllipsoidalCSNNPtr* &**coordinateSystem**()

Return the *cs::EllipsoidalCS* associated with the *CRS*.

**Returns**

a EllipsoidalCS.

*GeographicCRSNNPtr* **demoteTo2D**(const std::string &newName, const *io*::*DatabaseContextPtr* &dbContext) const

Return a variant of this *CRS* "demoted" to a 2D one, if not already the case.

**Since**

6.3

**Parameters**

- **newName** -- Name of the new *CRS*. If empty, *nameStr()* will be used.
- **dbContext** -- Database context to look for potentially already registered 2D *CRS*. May be nullptr.

**Returns**

a new *CRS* demoted to 2D, or the current one if already 2D or not applicable.

### Public Static Functions

static *GeographicCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::GeodeticReferenceFrameNNPtr &datum, const *cs*::EllipsoidalCSNNPtr &cs)

Instantiate a *GeographicCRS* from a *datum::GeodeticReferenceFrameNNPtr* and a *cs::EllipsoidalCS*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **datum** -- The datum of the *CRS*.
- **cs** -- a EllipsoidalCS.

**Returns**

new *GeographicCRS*.

static *GeographicCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::GeodeticReferenceFramePtr &datum, const *datum*::DatumEnsemblePtr &datumEnsemble, const *cs*::EllipsoidalCSNNPtr &cs)

Instantiate a *GeographicCRS* from a *datum::GeodeticReferenceFramePtr* or *datum::DatumEnsemble* and a *cs::EllipsoidalCS*.

One and only one of datum or datumEnsemble should be set to a non-null value.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **datum** -- The datum of the *CRS*, or nullptr
- **datumEnsemble** -- The datum ensemble of the *CRS*, or nullptr.
- **cs** -- a EllipsoidalCS.

**Returns**

new *GeographicCRS*.

### Public Static Attributes

static const *GeographicCRSNNPtr* **EPSG_4267**

EPSG:4267 / "NAD27" 2D *GeographicCRS*.

static const *GeographicCRSNNPtr* **EPSG_4269**

EPSG:4269 / "NAD83" 2D *GeographicCRS*.

static const *GeographicCRSNNPtr* **EPSG_4326**

EPSG:4326 / "WGS 84" 2D *GeographicCRS*.

static const *GeographicCRSNNPtr* **OGC_CRS84**

OGC:CRS84 / "CRS 84" 2D *GeographicCRS* (long, lat)

static const *GeographicCRSNNPtr* **EPSG_4807**

EPSG:4807 / "NTF (Paris)" 2D *GeographicCRS*.

static const *GeographicCRSNNPtr* **EPSG_4979**

EPSG:4979 / "WGS 84" 3D *GeographicCRS*.

class **VerticalCRS** : public virtual osgeo::proj::*crs*::*SingleCRS*, public osgeo::proj::*io*::*IPROJStringExportable*

*#include <crs.hpp>* A coordinate reference system having a vertical reference frame and a one-dimensional vertical coordinate system used for recording gravity-related heights or depths.

Vertical CRSs make use of the direction of gravity to define the concept of height or depth, but the relationship with gravity may not be straightforward. If the vertical reference frame is dynamic or if the vertical *CRS* has an association to a velocity model then the *CRS* is dynamic, else it is static.

**Remark**

Implements *VerticalCRS* from *ISO 19111:2019*

**Note**

Ellipsoidal heights cannot be captured in a vertical coordinate reference system. They exist only as an inseparable part of a 3D coordinate tuple defined in a geographic 3D coordinate reference system.

Subclassed by *osgeo::proj::crs::DerivedVerticalCRS*

### Public Functions

const *datum*::VerticalReferenceFramePtr **datum**() const

Return the *datum::VerticalReferenceFrame* associated with the *CRS*.

**Returns**

a VerticalReferenceFrame.

const *cs*::*VerticalCSNNPtr* **coordinateSystem**() const

Return the *cs::VerticalCS* associated with the *CRS*.

**Returns**

a VerticalCS.

const std::vector<*operation*::*TransformationNNPtr*> &**geoidModel**()

Return the geoid model associated with the *CRS*.

Geoid height model or height correction model linked to a geoid-based vertical *CRS*.

**Returns**

a geoid model. might be null

const std::vector<*operation*::*PointMotionOperationNNPtr*> &**velocityModel**()

Return the velocity model associated with the *CRS*.

**Returns**

a velocity model. might be null.

std::list<std::pair<*VerticalCRSNNPtr*, int>> **identify**(const *io*::*AuthorityFactoryPtr* &authorityFactory) const

Identify the *CRS* with reference CRSs.

The candidate CRSs are looked in the database when authorityFactory is not null.

Note that the implementation uses a set of heuristics to have a good compromise of successful identifications over execution time. It might miss legitimate matches in some circumstances.

The method returns a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match. 100% means that the name of the reference entry perfectly matches the *CRS* name, and both are equivalent. In which case a single result is returned. 90% means that *CRS* are equivalent, but the names are not exactly the same. 70% means that *CRS* are equivalent (equivalent datum and coordinate system), but the names are not equivalent. 25% means that the *CRS* are not equivalent, but there is some similarity in the names.

**Parameters**

**authorityFactory** -- Authority factory (if null, will return an empty list)

**Returns**

a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match.

### Public Static Functions

static *VerticalCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::VerticalReferenceFrameNNPtr &datumIn, const *cs*::*VerticalCSNNPtr* &csIn)

Instantiate a *VerticalCRS* from a *datum::VerticalReferenceFrame* and a *cs::VerticalCS*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined. The GEOID_MODEL property can be set to a TransformationNNPtr object.
- **datumIn** -- The datum of the *CRS*.
- **csIn** -- a VerticalCS.

**Returns**

new *VerticalCRS*.

static *VerticalCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::VerticalReferenceFramePtr &datumIn, const *datum*::DatumEnsemblePtr &datumEnsembleIn, const *cs*::*VerticalCSNNPtr* &csIn)

Instantiate a *VerticalCRS* from a *datum::VerticalReferenceFrame* or *datum::DatumEnsemble* and a *cs::VerticalCS*.

One and only one of datum or datumEnsemble should be set to a non-null value.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined. The GEOID_MODEL property can be set to a TransformationNNPtr object.
- **datumIn** -- The datum of the *CRS*, or nullptr
- **datumEnsembleIn** -- The datum ensemble of the *CRS*, or nullptr.
- **csIn** -- a VerticalCS.

**Returns**

new *VerticalCRS*.

class **DerivedCRS** : public virtual osgeo::proj::*crs*::*SingleCRS*

*#include <crs.hpp>* Abstract class modelling a single coordinate reference system that is defined through the application of a specified coordinate conversion to the definition of a previously established single coordinate reference system referred to as the base *CRS*.

A derived coordinate reference system inherits its datum (or datum ensemble) from its base *CRS*. The coordinate conversion between the base and derived coordinate reference system is implemented using the parameters and formula(s) specified in the definition of the coordinate conversion.

**Remark**

Implements *DerivedCRS* from *ISO 19111:2019*

Subclassed by *osgeo::proj::crs::DerivedCRSTemplate< DerivedTemporalCRSTraits >*, *osgeo::proj::crs::DerivedCRSTemplate< DerivedParametricCRSTraits >*, *osgeo::proj::crs::DerivedCRSTemplate< DerivedEngineeringCRSTraits >*, *osgeo::proj::crs::DerivedCRSTemplate< DerivedCRSTraits >*, *osgeo::proj::crs::DerivedGeodeticCRS*, *osgeo::proj::crs::DerivedGeographicCRS*, *osgeo::proj::crs::DerivedProjectedCRS*, *osgeo::proj::crs::DerivedVerticalCRS*, *osgeo::proj::crs::ProjectedCRS*

### Public Functions

const *SingleCRSNNPtr* &**baseCRS**()

Return the base *CRS* of a *DerivedCRS*.

**Returns**

the base *CRS*.

const *operation*::*ConversionNNPtr* **derivingConversion**() const

Return the deriving conversion from the base *CRS* to this *CRS*.

**Returns**

the deriving conversion.

class **ProjectedCRS** : public osgeo::proj::*crs*::*DerivedCRS*, public osgeo::proj::*io*::*IPROJStringExportable*

*#include <crs.hpp>* A derived coordinate reference system which has a geodetic (usually geographic) co-

ordinate reference system as its base *CRS*, thereby inheriting a geodetic reference frame, and is converted using a map projection.

It has a Cartesian coordinate system, usually two-dimensional but may be three-dimensional; in the 3D case the base geographic CRSs ellipsoidal height is passed through unchanged and forms the vertical axis of the projected *CRS*'s Cartesian coordinate system.

**Remark**

Implements *ProjectedCRS* from *ISO 19111:2019*

### Public Functions

const *GeodeticCRSNNPtr* &**baseCRS**()

Return the base *CRS* (a *GeodeticCRS*, which is generally a *GeographicCRS*) of the *ProjectedCRS*.

**Returns**

the base *CRS*.

const *cs*::*CartesianCSNNPtr* &**coordinateSystem**()

Return the *cs::CartesianCS* associated with the *CRS*.

**Returns**

a CartesianCS

std::list<std::pair<*ProjectedCRSNNPtr*, int>> **identify**(const *io*::*AuthorityFactoryPtr* &authorityFactory) const

Identify the *CRS* with reference CRSs.

The candidate CRSs are either hard-coded, or looked in the database when authorityFactory is not null.

Note that the implementation uses a set of heuristics to have a good compromise of successful identifications over execution time. It might miss legitimate matches in some circumstances.

The method returns a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match. The list is sorted by decreasing confidence.

100% means that the name of the reference entry perfectly matches the *CRS* name, and both are equivalent. In which case a single result is returned. 90% means that *CRS* are equivalent, but the names are not exactly the same. 70% means that *CRS* are equivalent (equivalent base *CRS*, conversion and coordinate system), but the names are not equivalent. 60% means that *CRS* have strong similarity (equivalent base datum, conversion and coordinate system), but the names are not equivalent. 50% means that *CRS* have similarity (equivalent base ellipsoid and conversion), but the coordinate system do not match (e.g. different axis ordering or axis unit). 25% means that the *CRS* are not equivalent, but there is some similarity in the names.

For the purpose of this function, equivalence is tested with the *util::IComparable::Criterion::EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS*, that is to say that the axis order of the base *GeographicCRS* is ignored.

**Parameters**

**authorityFactory** -- Authority factory (or null, but degraded functionality)

**Returns**

a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match.

*ProjectedCRSNNPtr* **demoteTo2D**(const std::string &newName, const *io*::*DatabaseContextPtr* &dbContext) const

Return a variant of this *CRS* "demoted" to a 2D one, if not already the case.

**Since**
6.3

**Parameters**
- **newName** -- Name of the new *CRS*. If empty, *nameStr()* will be used.
- **dbContext** -- Database context to look for potentially already registered 2D *CRS*. May be nullptr.

**Returns**
a new *CRS* demoted to 2D, or the current one if already 2D or not applicable.

### Public Static Functions

static *ProjectedCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *GeodeticCRSNNPtr* &baseCRSIn, const *operation*::*ConversionNNPtr* &derivingConversionIn, const *cs*::*CartesianCSNNPtr* &csIn)

Instantiate a *ProjectedCRS* from a base *CRS*, a deriving *operation::Conversion* and a coordinate system.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **baseCRSIn** -- The base *CRS*, a *GeodeticCRS* that is generally a *GeographicCRS*.
- **derivingConversionIn** -- The deriving *operation::Conversion* (typically using a map projection method)
- **csIn** -- The coordniate system.

**Returns**
new *ProjectedCRS*.

class **TemporalCRS** : public virtual osgeo::proj::*crs*::*SingleCRS*

*#include <crs.hpp>* A coordinate reference system associated with a temporal datum and a one-dimensional temporal coordinate system.

**Remark**

Implements *TemporalCRS* from *ISO 19111:2019*

### Public Functions

const *datum*::TemporalDatumNNPtr **datum**() const

Return the *datum::TemporalDatum* associated with the *CRS*.

**Returns**
a TemporalDatum

const *cs*::*TemporalCSNNPtr* **coordinateSystem**() const

Return the *cs::TemporalCS* associated with the *CRS*.

**Returns**
a TemporalCS

### Public Static Functions

static *TemporalCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::TemporalDatumNNPtr &datumIn, const *cs*::*TemporalCSNNPtr* &csIn)

Instantiate a *TemporalCRS* from a datum and a coordinate system.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **datumIn** -- the datum.
- **csIn** -- the coordinate system.

**Returns**

new *TemporalCRS*.

class **EngineeringCRS** : public virtual osgeo::proj::*crs*::*SingleCRS*

*#include <crs.hpp>* Contextually local coordinate reference system associated with an engineering datum.

It is applied either to activities on or near the surface of the Earth without geodetic corrections, or on moving platforms such as road vehicles, vessels, aircraft or spacecraft, or as the internal *CRS* of an image.

In *WKT2 standard*, it maps to a ENGINEERINGCRS / ENGCRS keyword. In *WKT1 specification*, it maps to a LOCAL_CS keyword.

**Remark**

Implements *EngineeringCRS* from *ISO 19111:2019*

### Public Functions

const *datum*::EngineeringDatumNNPtr **datum**() const

Return the *datum::EngineeringDatum* associated with the *CRS*.

**Returns**

a EngineeringDatum

### Public Static Functions

static *EngineeringCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::EngineeringDatumNNPtr &datumIn, const *cs*::*CoordinateSystemNNPtr* &csIn)

Instantiate a *EngineeringCRS* from a datum and a coordinate system.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **datumIn** -- the datum.
- **csIn** -- the coordinate system.

**Returns**

new *EngineeringCRS*.

class **ParametricCRS** : public virtual osgeo::proj::*crs*::*SingleCRS*

*#include <crs.hpp>* Contextually local coordinate reference system associated with an engineering datum.

This is applied either to activities on or near the surface of the Earth without geodetic corrections, or on moving platforms such as road vehicles vessels, aircraft or spacecraft, or as the internal *CRS* of an image.

**Remark**

Implements *ParametricCRS* from *ISO 19111:2019*

### Public Functions

const *datum*::ParametricDatumNNPtr **datum**() const

Return the *datum::ParametricDatum* associated with the *CRS*.

**Returns**

a ParametricDatum

const *cs*::*ParametricCSNNPtr* **coordinateSystem**() const

Return the *cs::TemporalCS* associated with the *CRS*.

**Returns**

a TemporalCS

### Public Static Functions

static *ParametricCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *datum*::ParametricDatumNNPtr &datumIn, const *cs*::*ParametricCSNNPtr* &csIn)

Instantiate a *ParametricCRS* from a datum and a coordinate system.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **datumIn** -- the datum.
- **csIn** -- the coordinate system.

**Returns**

new *ParametricCRS*.

class **InvalidCompoundCRSException** : public osgeo::proj::*util*::*Exception*

*#include <crs.hpp>* Exception thrown when attempting to create an invalid compound *CRS*.

class **CompoundCRS** : public osgeo::proj::*crs*::*CRS*, public osgeo::proj::*io*::*IPROJStringExportable*

*#include <crs.hpp>* A coordinate reference system describing the position of points through two or more independent single coordinate reference systems.

**Remark**

Implements *CompoundCRS* from *ISO 19111:2019*

**Note**

Two coordinate reference systems are independent of each other if coordinate values in one cannot be converted or transformed into coordinate values in the other.

**Note**

As a departure to *ISO 19111:2019*, we allow to build a *CompoundCRS* from *CRS* objects, whereas ISO19111:2019 restricts the components to *SingleCRS*.

### Public Functions

const std::vector<*CRSNNPtr*> &**componentReferenceSystems**()

Return the components of a *CompoundCRS*.

**Returns**
the components.

std::list<std::pair<*CompoundCRSNNPtr*, int>> **identify**(const *io*::*AuthorityFactoryPtr* &authorityFactory) const

Identify the *CRS* with reference CRSs.

The candidate CRSs are looked in the database when authorityFactory is not null.

Note that the implementation uses a set of heuristics to have a good compromise of successful identifications over execution time. It might miss legitimate matches in some circumstances.

The method returns a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match. The list is sorted by decreasing confidence.

100% means that the name of the reference entry perfectly matches the *CRS* name, and both are equivalent. In which case a single result is returned. 90% means that *CRS* are equivalent, but the names are not exactly the same. 70% means that *CRS* are equivalent (equivalent horizontal and vertical *CRS*), but the names are not equivalent. 25% means that the *CRS* are not equivalent, but there is some similarity in the names.

**Parameters**
**authorityFactory** -- Authority factory (if null, will return an empty list)

**Returns**
a list of matching reference *CRS*, and the percentage (0-100) of confidence in the match.

### Public Static Functions

static *CompoundCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::vector<*CRSNNPtr*> &components)

Instantiate a *CompoundCRS* from a vector of *CRS*.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **components** -- the component *CRS* of the *CompoundCRS*.

**Throws**
*InvalidCompoundCRSException* -- if the object cannot be constructed.

**Returns**
new *CompoundCRS*.

class **BoundCRS** : public osgeo::proj::*crs*::*CRS*, public osgeo::proj::*io*::*IPROJStringExportable*

*#include <crs.hpp>* A coordinate reference system with an associated transformation to a target/hub *CRS*.

The definition of a *CRS* is not dependent upon any relationship to an independent *CRS*. However in an implementation that merges datasets referenced to differing CRSs, it is sometimes useful to associate the definition of the transformation that has been used with the *CRS* definition. This facilitates the interrelationship of *CRS* by concatenating transformations via a common or hub *CRS*. This is sometimes referred to as "early-binding". *WKT2 standard* permits the association of an abridged coordinate transformation

description with a coordinate reference system description in a single text string. In a *BoundCRS*, the abridged coordinate transformation is applied to the source *CRS* with the target *CRS* being the common or hub system.

Coordinates referring to a *BoundCRS* are expressed into its source/base *CRS*.

This abstraction can for example model the concept of TOWGS84 datum shift present in *WKT1 specification*.

**Remark**

Implements *BoundCRS* from *WKT2 standard*

**Note**

Contrary to other *CRS* classes of this package, there is no *ISO 19111:2019* modelling of a *BoundCRS*.

### Public Functions

const *CRSNNPtr* &**baseCRS**()

Return the base *CRS*.

This is the *CRS* into which coordinates of the *BoundCRS* are expressed.

**Returns**
the base *CRS*.

*CRSNNPtr* **baseCRSWithCanonicalBoundCRS**() const

Return a shallow clone of the base *CRS* that points to a shallow clone of this *BoundCRS*.

The base *CRS* is the *CRS* into which coordinates of the *BoundCRS* are expressed.

The returned *CRS* will actually be a shallow clone of the actual base *CRS*, with the extra property that *CRS::canonicalBoundCRS()* will point to a shallow clone of this *BoundCRS*. Use this only if you want to work with the base *CRS* object rather than the *BoundCRS*, but wanting to be able to retrieve the *BoundCRS* later.

**Returns**
the base *CRS*.

const *CRSNNPtr* &**hubCRS**()

Return the target / hub *CRS*.

**Returns**
the hub *CRS*.

const *operation*::*TransformationNNPtr* &**transformation**()

Return the transformation to the hub RS.

**Returns**
transformation.

### Public Static Functions

static *BoundCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *CRSNNPtr* &baseCRSIn, const *CRSNNPtr* &hubCRSIn, const *operation*::*TransformationNNPtr* &transformationIn)

Instantiate a *BoundCRS* from a base *CRS*, a hub *CRS* and a transformation.

**Since**
PROJ 8.2

**Parameters**
- **properties** -- See *General properties*.
- **baseCRSIn** -- base *CRS*.
- **hubCRSIn** -- hub *CRS*.
- **transformationIn** -- transformation from base *CRS* to hub *CRS*.

**Returns**
new *BoundCRS*.

static *BoundCRSNNPtr* **create**(const *CRSNNPtr* &baseCRSIn, const *CRSNNPtr* &hubCRSIn, const *operation*::*TransformationNNPtr* &transformationIn)

Instantiate a *BoundCRS* from a base *CRS*, a hub *CRS* and a transformation.

**Parameters**
- **baseCRSIn** -- base *CRS*.
- **hubCRSIn** -- hub *CRS*.
- **transformationIn** -- transformation from base *CRS* to hub *CRS*.

**Returns**
new *BoundCRS*.

static *BoundCRSNNPtr* **createFromTOWGS84**(const *CRSNNPtr* &baseCRSIn, const std::vector<double> &TOWGS84Parameters)

Instantiate a *BoundCRS* from a base *CRS* and TOWGS84 parameters.

**Parameters**
- **baseCRSIn** -- base *CRS*.
- **TOWGS84Parameters** -- a vector of 3 or 7 double values representing WKT1 TOWGS84 parameter.

**Returns**
new *BoundCRS*.

static *BoundCRSNNPtr* **createFromNadgrids**(const *CRSNNPtr* &baseCRSIn, const std::string &filename)

Instantiate a *BoundCRS* from a base *CRS* and nadgrids parameters.

**Parameters**
- **baseCRSIn** -- base *CRS*.
- **filename** -- Horizontal grid filename

**Returns**
new *BoundCRS*.

class **DerivedGeodeticCRS** : public osgeo::proj::*crs*::*GeodeticCRS*, public osgeo::proj::*crs*::*DerivedCRS*

*#include <crs.hpp>* A derived coordinate reference system which has either a geodetic or a geographic coordinate reference system as its base *CRS*, thereby inheriting a geodetic reference frame, and associated with a 3D Cartesian or spherical coordinate system.

**Remark**

Implements *DerivedGeodeticCRS* from *ISO 19111:2019*

### Public Functions

const *GeodeticCRSNNPtr* **baseCRS**() const

Return the base *CRS* (a *GeodeticCRS*) of a *DerivedGeodeticCRS*.

**Returns**

the base *CRS*.

### Public Static Functions

static *DerivedGeodeticCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *GeodeticCRSNNPtr* &baseCRSIn, const *operation*::*ConversionNNPtr* &derivingConversionIn, const *cs*::*CartesianCSNNPtr* &csIn)

Instantiate a *DerivedGeodeticCRS* from a base *CRS*, a deriving conversion and a *cs::CartesianCS*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **baseCRSIn** -- base *CRS*.
- **derivingConversionIn** -- the deriving conversion from the base *CRS* to this *CRS*.
- **csIn** -- the coordinate system.

**Returns**

new *DerivedGeodeticCRS*.

static *DerivedGeodeticCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *GeodeticCRSNNPtr* &baseCRSIn, const *operation*::*ConversionNNPtr* &derivingConversionIn, const *cs*::*SphericalCSNNPtr* &csIn)

Instantiate a *DerivedGeodeticCRS* from a base *CRS*, a deriving conversion and a *cs::SphericalCS*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **baseCRSIn** -- base *CRS*.
- **derivingConversionIn** -- the deriving conversion from the base *CRS* to this *CRS*.
- **csIn** -- the coordinate system.

**Returns**

new *DerivedGeodeticCRS*.

class **DerivedGeographicCRS** : public osgeo::proj::*crs*::*GeographicCRS*, public osgeo::proj::*crs*::*DerivedCRS*

*#include <crs.hpp>* A derived coordinate reference system which has either a geodetic or a geographic coordinate reference system as its base *CRS*, thereby inheriting a geodetic reference frame, and an ellipsoidal coordinate system.

A derived geographic *CRS* can be based on a geodetic *CRS* only if that geodetic *CRS* definition includes an ellipsoid.

**Remark**

Implements *DerivedGeographicCRS* from *ISO 19111:2019*

### Public Functions

const *GeodeticCRSNNPtr* **baseCRS**() const

Return the base *CRS* (a *GeodeticCRS*) of a *DerivedGeographicCRS*.

**Returns**
the base *CRS*.

*DerivedGeographicCRSNNPtr* **demoteTo2D**(const std::string &newName, const *io*::*DatabaseContextPtr* &dbContext) const

Return a variant of this *CRS* "demoted" to a 2D one, if not already the case.

**Since**
8.1.1

**Parameters**
- **newName** -- Name of the new *CRS*. If empty, *nameStr()* will be used.
- **dbContext** -- Database context to look for potentially already registered 2D *CRS*. May be nullptr.

**Returns**
a new *CRS* demoted to 2D, or the current one if already 2D or not applicable.

### Public Static Functions

static *DerivedGeographicCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *GeodeticCRSNNPtr* &baseCRSIn, const *operation*::*ConversionNNPtr* &derivingConversionIn, const *cs*::*EllipsoidalCSNNPtr* &csIn)

Instantiate a *DerivedGeographicCRS* from a base *CRS*, a deriving conversion and a *cs::EllipsoidalCS*.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **baseCRSIn** -- base *CRS*.
- **derivingConversionIn** -- the deriving conversion from the base *CRS* to this *CRS*.
- **csIn** -- the coordinate system.

**Returns**
new *DerivedGeographicCRS*.

class **DerivedProjectedCRS** : public osgeo::proj::*crs*::*DerivedCRS*

*#include <crs.hpp>* A derived coordinate reference system which has a projected coordinate reference system as its base *CRS*, thereby inheriting a geodetic reference frame, but also inheriting the distortion characteristics of the base projected *CRS*.

A *DerivedProjectedCRS* is not a *ProjectedCRS*.

**Remark**

Implements *DerivedProjectedCRS* from *ISO 19111:2019*

### Public Functions

const *ProjectedCRSNNPtr* **baseCRS**() const

Return the base *CRS* (a *ProjectedCRS*) of a *DerivedProjectedCRS*.

**Returns**
the base *CRS*.

*DerivedProjectedCRSNNPtr* **demoteTo2D**(const std::string &newName, const *io*::*DatabaseContextPtr* &dbContext) const

Return a variant of this *CRS* "demoted" to a 2D one, if not already the case.

**Since**
9.1.1

**Parameters**
- **newName** -- Name of the new *CRS*. If empty, *nameStr()* will be used.
- **dbContext** -- Database context to look for potentially already registered 2D *CRS*. May be nullptr.

**Returns**
a new *CRS* demoted to 2D, or the current one if already 2D or not applicable.

### Public Static Functions

static *DerivedProjectedCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *ProjectedCRSNNPtr* &baseCRSIn, const *operation*::*ConversionNNPtr* &derivingConversionIn, const *cs*::*CoordinateSystemNNPtr* &csIn)

Instantiate a *DerivedProjectedCRS* from a base *CRS*, a deriving conversion and a cs::CS.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **baseCRSIn** -- base *CRS*.
- **derivingConversionIn** -- the deriving conversion from the base *CRS* to this *CRS*.
- **csIn** -- the coordinate system.

**Returns**
new *DerivedProjectedCRS*.

class **DerivedVerticalCRS** : public osgeo::proj::*crs*::*VerticalCRS*, public osgeo::proj::*crs*::*DerivedCRS*

*#include <crs.hpp>* A derived coordinate reference system which has a vertical coordinate reference system as its base *CRS*, thereby inheriting a vertical reference frame, and a vertical coordinate system.

**Remark**

Implements *DerivedVerticalCRS* from *ISO 19111:2019*

### Public Functions

const *VerticalCRSNNPtr* **baseCRS**() const

Return the base *CRS* (a *VerticalCRS*) of a *DerivedVerticalCRS*.

**Returns**
the base *CRS*.

### Public Static Functions

static *DerivedVerticalCRSNNPtr* **create**(const *util*::*PropertyMap* &properties, const *VerticalCRSNNPtr* &baseCRSIn, const *operation*::*ConversionNNPtr* &derivingConversionIn, const *cs*::*VerticalCSNNPtr* &csIn)

Instantiate a *DerivedVerticalCRS* from a base *CRS*, a deriving conversion and a *cs::VerticalCS*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **baseCRSIn** -- base *CRS*.
- **derivingConversionIn** -- the deriving conversion from the base *CRS* to this *CRS*.
- **csIn** -- the coordinate system.

**Returns**

new *DerivedVerticalCRS*.

template<class **DerivedCRSTraits**>

class **DerivedCRSTemplate** : public *DerivedCRSTraits*::BaseType, public osgeo::proj::*crs*::*DerivedCRS*

*#include <crs.hpp>* Template representing a derived coordinate reference system.

### Public Types

typedef *util*::nn<std::shared_ptr<*DerivedCRSTemplate*>> **NNPtr**

Non-null shared pointer of *DerivedCRSTemplate*

typedef *util*::nn<std::shared_ptr<BaseType>> **BaseNNPtr**

Non-null shared pointer of BaseType

typedef *util*::nn<std::shared_ptr<CSType>> **CSNNPtr**

Non-null shared pointer of CSType

### Public Functions

const *BaseNNPtr* **baseCRS**() const

Return the base *CRS* of a *DerivedCRSTemplate*.

**Returns**

the base *CRS*.

### Public Static Functions

static *NNPtr* **create**(const *util*::*PropertyMap* &properties, const *BaseNNPtr* &baseCRSIn, const *operation*::*ConversionNNPtr* &derivingConversionIn, const *CSNNPtr* &csIn)

Instantiate a *DerivedCRSTemplate* from a base *CRS*, a deriving conversion and a *cs::CoordinateSystem*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **baseCRSIn** -- base *CRS*.
- **derivingConversionIn** -- the deriving conversion from the base *CRS* to this *CRS*.
- **csIn** -- the coordinate system.

**Returns**

new *DerivedCRSTemplate*.

class **DerivedEngineeringCRS** : public osgeo::proj::*crs*::*DerivedCRSTemplate*<DerivedEngineeringCRSTraits>

*#include <crs.hpp>* A derived coordinate reference system which has an engineering coordinate reference system as its base *CRS*, thereby inheriting an engineering datum, and is associated with one of the coordinate system types for an *EngineeringCRS*.

**Remark**

Implements *DerivedEngineeringCRS* from *ISO 19111:2019*

class **DerivedParametricCRS** : public osgeo::proj::*crs*::*DerivedCRSTemplate*<DerivedParametricCRSTraits>

*#include <crs.hpp>* A derived coordinate reference system which has a parametric coordinate reference system as its base *CRS*, thereby inheriting a parametric datum, and a parametric coordinate system.

**Remark**

Implements *DerivedParametricCRS* from *ISO 19111:2019*

class **DerivedTemporalCRS** : public osgeo::proj::*crs*::*DerivedCRSTemplate*<DerivedTemporalCRSTraits>

*#include <crs.hpp>* A derived coordinate reference system which has a temporal coordinate reference system as its base *CRS*, thereby inheriting a temporal datum, and a temporal coordinate system.

**Remark**

Implements *DerivedTemporalCRS* from *ISO 19111:2019*

#### 10.5.4.8 operation namespace

namespace **operation**

Coordinate operations (relationship between any two coordinate reference systems).

*osgeo.proj.operation* namespace

This covers *Conversion*, *Transformation*, *PointMotionOperation* or *ConcatenatedOperation*.

##### Typedefs

typedef std::shared_ptr<*CoordinateOperation*> **CoordinateOperationPtr**

Shared pointer of *CoordinateOperation*

typedef *util*::nn<*CoordinateOperationPtr*> **CoordinateOperationNNPtr**

Non-null shared pointer of *CoordinateOperation*

using **CoordinateTransformerPtr** = std::unique_ptr<*CoordinateTransformer*>

Shared pointer of *CoordinateTransformer*

using **CoordinateTransformerNNPtr** = *util*::nn<*CoordinateTransformerPtr*>

Non-null shared pointer of *CoordinateTransformer*

using **TransformationPtr** = std::shared_ptr<*Transformation*>

Shared pointer of *Transformation*

using **TransformationNNPtr** = *util*::nn<*TransformationPtr*>

Non-null shared pointer of *Transformation*

using **GeneralOperationParameterPtr** = std::shared_ptr<*GeneralOperationParameter*>

Shared pointer of *GeneralOperationParameter*

using **GeneralOperationParameterNNPtr** = *util*::nn<*GeneralOperationParameterPtr*>

Non-null shared pointer of *GeneralOperationParameter*

using **OperationParameterPtr** = std::shared_ptr<*OperationParameter*>

Shared pointer of *OperationParameter*

using **OperationParameterNNPtr** = *util*::nn<*OperationParameterPtr*>

Non-null shared pointer of *OperationParameter*

using **GeneralParameterValuePtr** = std::shared_ptr<*GeneralParameterValue*>

Shared pointer of *GeneralParameterValue*

using **GeneralParameterValueNNPtr** = *util*::nn<*GeneralParameterValuePtr*>

Non-null shared pointer of *GeneralParameterValue*

using **ParameterValuePtr** = std::shared_ptr<*ParameterValue*>

Shared pointer of *ParameterValue*

using **ParameterValueNNPtr** = *util*::nn<*ParameterValuePtr*>

Non-null shared pointer of *ParameterValue*

using **OperationParameterValuePtr** = std::shared_ptr<*OperationParameterValue*>

Shared pointer of *OperationParameterValue*

using **OperationParameterValueNNPtr** = *util*::nn<*OperationParameterValuePtr*>

Non-null shared pointer of *OperationParameterValue*

using **OperationMethodPtr** = std::shared_ptr<*OperationMethod*>

Shared pointer of *OperationMethod*

using **OperationMethodNNPtr** = *util*::nn<*OperationMethodPtr*>

Non-null shared pointer of *OperationMethod*

using **SingleOperationPtr** = std::shared_ptr<*SingleOperation*>

Shared pointer of *SingleOperation*

using **SingleOperationNNPtr** = *util*::nn<*SingleOperationPtr*>

Non-null shared pointer of *SingleOperation*

typedef std::shared_ptr<*Conversion*> **ConversionPtr**

Shared pointer of *Conversion*

typedef *util*::nn<*ConversionPtr*> **ConversionNNPtr**

Non-null shared pointer of *Conversion*

typedef std::shared_ptr<*PointMotionOperation*> **PointMotionOperationPtr**

Shared pointer of *PointMotionOperation*

typedef *util*::nn<*PointMotionOperationPtr*> **PointMotionOperationNNPtr**

Non-null shared pointer of *PointMotionOperation*

using **ConcatenatedOperationPtr** = std::shared_ptr<*ConcatenatedOperation*>

Shared pointer of *ConcatenatedOperation*

using **ConcatenatedOperationNNPtr** = *util*::nn<*ConcatenatedOperationPtr*>

Non-null shared pointer of *ConcatenatedOperation*

using **CoordinateOperationContextPtr** = std::unique_ptr<*CoordinateOperationContext*>

Unique pointer of *CoordinateOperationContext*

using **CoordinateOperationContextNNPtr** = *util*::nn<*CoordinateOperationContextPtr*>

Non-null unique pointer of *CoordinateOperationContext*

using **CoordinateOperationFactoryPtr** = std::unique_ptr<*CoordinateOperationFactory*>

Unique pointer of *CoordinateOperationFactory*

using **CoordinateOperationFactoryNNPtr** = *util*::nn<*CoordinateOperationFactoryPtr*>

Non-null unique pointer of *CoordinateOperationFactory*

### Functions

static std::list<*crs*::GeodeticCRSNNPtr> **findCandidateGeodCRSForDatum**(const *io*::*AuthorityFactoryPtr* &authFactory, const *crs*::GeodeticCRS *crs, const *datum*::GeodeticReferenceFrameNNPtr &datum)

static bool **isRegularVerticalGridMethod**(int methodEPSGCode, bool &reverseOffsetSign)

static double **negate**(double val)

static *CoordinateOperationPtr* **createApproximateInverseIfPossible**(const *Transformation* *op)

struct **GridDescription**

*#include <coordinateoperation.hpp>* Grid description.

### Public Members

std::string **shortName**

Grid short filename

std::string **fullName**

Grid full path name (if found)

std::string **packageName**

Package name (or empty)

std::string **url**

Grid URL (if packageName is empty), or package URL (or empty)

bool **directDownload**

Whether url can be fetched directly.

bool **openLicense**

Whether the grid is released with an open license.

bool **available**

Whether GRID is available.

class **CoordinateTransformer**

*#include <coordinateoperation.hpp>* Coordinate transformer.

Performs coordinate transformation of coordinate tuplies.

**Since**
9.3

### Public Functions

PJ_COORD **transform**(PJ_COORD coord)

Transforms a coordinate tuple.

PJ_COORD is a union of many structures. In the context of this method, it is prudent to only use the v[] array, with the understanding that the expected input values should be passed in the order and the unit of the successive axis of the input CRS. Similarly the values returned in the v[] array of the output PJ_COORD are in the order and the unit of the successive axis of the output CRS. For coordinate

operations involving a time-dependent operation, coord.v[3] is the decimal year of the coordinate epoch of the input (or HUGE_VAL to indicate none)

If an error occurs, HUGE_VAL is returned in the .v[0] member of the output coordinate tuple.

Example how to transform coordinates from EPSG:4326 (WGS 84 latitude/longitude) to EPSG:32631 (WGS 84 / UTM zone 31N).

```
auto authFactory =
    AuthorityFactory::create(DatabaseContext::create(), std::string());
auto coord_op_ctxt = CoordinateOperationContext::create(
    authFactory, nullptr, 0.0);
auto authFactoryEPSG =
    AuthorityFactory::create(DatabaseContext::create(), "EPSG");
auto list = CoordinateOperationFactory::create()->createOperations(
    authFactoryEPSG->createCoordinateReferenceSystem("4326"),
    authFactoryEPSG->createCoordinateReferenceSystem("32631"),
    coord_op_ctxt);
ASSERT_TRUE(!list.empty());
PJ_CONTEXT* ctx = proj_context_create();
auto transformer = list[0]->coordinateTransformer(ctx);
PJ_COORD c;
c.v[0] = 49; // latitude in degree
c.v[1] = 2;  // longitude in degree
c.v[2] = 0;
c.v[3] = HUGE_VAL;
c = transformer->transform(c);
EXPECT_NEAR(c.v[0], 426857.98771728, 1e-8); // easting in metre
EXPECT_NEAR(c.v[1], 5427937.52346492, 1e-8); // northing in metre
proj_context_destroy(ctx);
```

class **CoordinateOperation** : public osgeo::proj::*common*::*ObjectUsage*, public osgeo::proj::*io*::*IPROJStringExportable*, public osgeo::proj::*io*::*IJSONExportable*

*#include <coordinateoperation.hpp>* Abstract class for a mathematical operation on coordinates.

A mathematical operation:

- on coordinates that transforms or converts them from one coordinate reference system to another coordinate reference system
- or that describes the change of coordinate values within one coordinate reference system due to the motion of the point between one coordinate epoch and another coordinate epoch.

Many but not all coordinate operations (from CRS A to CRS B) also uniquely define the inverse coordinate operation (from CRS B to CRS A). In some cases, the coordinate operation method algorithm for the inverse coordinate operation is the same as for the forward algorithm, but the signs of some coordinate operation parameter values have to be reversed. In other cases, different algorithms are required for the forward and inverse coordinate operations, but the same coordinate operation parameter values are used. If (some) entirely different parameter values are needed, a different coordinate operation shall be defined.

**Remark**

Implements *CoordinateOperation* from *ISO 19111:2019*

Subclassed by *osgeo::proj::operation::ConcatenatedOperation*, *osgeo::proj::operation::SingleOperation*

### Public Functions

const *util*::*optional*<std::string> &**operationVersion**() const

Return the version of the coordinate transformation (i.e. instantiation due to the stochastic nature of the parameters).

Mandatory when describing a coordinate transformation or point motion operation, and should not be supplied for a coordinate conversion.

**Returns**

version or empty.

const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &**coordinateOperationAccuracies**() const

Return estimate(s) of the impact of this coordinate operation on point accuracy.

Gives position error estimates for target coordinates of this coordinate operation, assuming no errors in source coordinates.

**Returns**

estimate(s) or empty vector.

const *crs*::*CRSPtr* **sourceCRS**() const

Return the source CRS of this coordinate operation.

This should not be null, expect for of a derivingConversion of a DerivedCRS when the owning DerivedCRS has been destroyed.

**Returns**

source CRS, or null.

const *crs*::*CRSPtr* **targetCRS**() const

Return the target CRS of this coordinate operation.

This should not be null, expect for of a derivingConversion of a DerivedCRS when the owning DerivedCRS has been destroyed.

**Returns**

target CRS, or null.

const *crs*::*CRSPtr* &**interpolationCRS**() const

Return the interpolation CRS of this coordinate operation.

**Returns**

interpolation CRS, or null.

const *util*::*optional*<*common*::*DataEpoch*> &**sourceCoordinateEpoch**() const

Return the source epoch of coordinates.

**Returns**

source epoch of coordinates, or empty.

const *util*::*optional*<*common*::*DataEpoch*> &**targetCoordinateEpoch**() const

Return the target epoch of coordinates.

**Returns**

target epoch of coordinates, or empty.

*CoordinateTransformerNNPtr* **coordinateTransformer**(PJ_CONTEXT *ctx) const

Return a coordinate transformer for this operation.

The returned coordinate transformer is tied to the provided context, and should only be called by the thread "owning" the passed context. It should not be used after the context has been destroyed.

**Since**
9.3

**Parameters**
**ctx** -- Execution context to which the transformer will be tied to. If null, the default context will be used (only safe for single-threaded applications).
**Throws**
UnsupportedOperationException -- if the transformer cannot be instantiated.
**Returns**
a new *CoordinateTransformer* instance.

virtual *CoordinateOperationNNPtr* **inverse**() const = 0

Return the inverse of the coordinate operation.
**Throws**
*util*::*UnsupportedOperationException* -- if inverse is not available

virtual std::set<*GridDescription*> **gridsNeeded**(const *io*::*DatabaseContextPtr* &databaseContext, bool considerKnownGridsAsAvailable) const = 0

Return grids needed by an operation.

bool **isPROJInstantiable**(const *io*::*DatabaseContextPtr* &databaseContext, bool considerKnownGridsAsAvailable) const

Return whether a coordinate operation can be instantiated as a PROJ pipeline, checking in particular that referenced grids are available.

bool **hasBallparkTransformation**() const

Return whether a coordinate operation has a "ballpark" transformation, that is a very approximate one, due to lack of more accurate transformations.

Typically a null geographic offset between two horizontal datum, or a null vertical offset (or limited to unit changes) between two vertical datum. Errors of several tens to one hundred meters might be expected, compared to more accurate transformations.

bool **requiresPerCoordinateInputTime**() const

Return whether a coordinate operation requires coordinate tuples to have a valid input time for the coordinate transformation to succeed. (this applies for the forward direction)

Note: in the case of a time-dependent Helmert transformation, this function will return true, but when executing proj_trans(), execution will still succeed if the time information is missing, due to the transformation central epoch being used as a fallback.

**Since**
9.5

*CoordinateOperationNNPtr* **normalizeForVisualization**() const

Return a variation of the current coordinate operation whose axis order is the one expected for visualization purposes.

### Public Static Attributes

static const std::string **OPERATION_VERSION_KEY**

Key to set the operation version of a *operation::CoordinateOperation*.

The value is to be provided as a string.

class **GeneralOperationParameter** : public osgeo::proj::*common*::*IdentifiedObject*

*#include <coordinateoperation.hpp>* Abstract class modelling a parameter value (*OperationParameter*) or group of parameters.

**Remark**

Implements *GeneralOperationParameter* from *ISO 19111:2019*

Subclassed by *osgeo::proj::operation::OperationParameter*

class **OperationParameter** : public osgeo::proj::*operation*::*GeneralOperationParameter*

*#include <coordinateoperation.hpp>* The definition of a parameter used by a coordinate operation method.

Most parameter values are numeric, but other types of parameter values are possible.

**Remark**

Implements *OperationParameter* from *ISO 19111:2019*

### Public Functions

int **getEPSGCode**()

Return the EPSG code, either directly, or through the name.

**Returns**

code, or 0 if not found

### Public Static Functions

static *OperationParameterNNPtr* **create**(const *util*::*PropertyMap* &properties)

Instantiate a *OperationParameter*.

**Parameters**

**properties** -- See *General properties*. At minimum the name should be defined.

**Returns**

a new *OperationParameter*.

static const char ***getNameForEPSGCode**(int epsg_code) noexcept

Return the name of a parameter designed by its EPSG code.

**Returns**

name, or nullptr if not found

class **GeneralParameterValue** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*io*::*IWKTExportable*, public osgeo::proj::*io*::*IJSONExportable*, public osgeo::proj::*util*::*IComparable*

*#include <coordinateoperation.hpp>* Abstract class modelling a parameter value (*OperationParameterValue*) or group of parameter values.

**Remark**

Implements *GeneralParameterValue* from *ISO 19111:2019*

Subclassed by *osgeo::proj::operation::OperationParameterValue*

class **ParameterValue** : public osgeo::proj::*util*::*BaseObject*, public osgeo::proj::*io*::*IWKTExportable*, public osgeo::proj::*util*::*IComparable*

*#include <coordinateoperation.hpp>* The value of the coordinate operation parameter.

Most parameter values are numeric, but other types of parameter values are possible.

**Remark**

Implements *ParameterValue* from *ISO 19111:2019*

### Public Types

enum class **Type**

Type of the value.

*Values:*

enumerator **MEASURE**

Measure (i.e. value with a unit)

enumerator **STRING**

String

enumerator **INTEGER**

Integer

enumerator **BOOLEAN**

Boolean

enumerator **FILENAME**

Filename

### Public Functions

const *Type* &**type**()

Returns the type of a parameter value.

**Returns**

the type.

const *common*::*Measure* &**value**()

Returns the value as a Measure (assumes *type()* == *Type::MEASURE*)

**Returns**
the value as a Measure.

const std::string &**stringValue**()

Returns the value as a string (assumes *type()* == *Type::STRING*)

**Returns**
the value as a string.

const std::string &**valueFile**()

Returns the value as a filename (assumes *type()* == *Type::FILENAME*)

**Returns**
the value as a filename.

int **integerValue**()

Returns the value as a integer (assumes *type()* == *Type::INTEGER*)

**Returns**
the value as a integer.

bool **booleanValue**()

Returns the value as a boolean (assumes *type()* == *Type::BOOLEAN*)

**Returns**
the value as a boolean.

### Public Static Functions

static *ParameterValueNNPtr* **create**(const *common*::*Measure* &measureIn)

Instantiate a *ParameterValue* from a Measure (i.e. a value associated with a unit)

**Returns**
a new *ParameterValue*.

static *ParameterValueNNPtr* **create**(const char *stringValueIn)

Instantiate a *ParameterValue* from a string value.

**Returns**
a new *ParameterValue*.

static *ParameterValueNNPtr* **create**(const std::string &stringValueIn)

Instantiate a *ParameterValue* from a string value.

**Returns**
a new *ParameterValue*.

static *ParameterValueNNPtr* **create**(int integerValueIn)

Instantiate a *ParameterValue* from a integer value.

**Returns**
a new *ParameterValue*.

static *ParameterValueNNPtr* **create**(bool booleanValueIn)

Instantiate a *ParameterValue* from a boolean value.

**Returns**
a new *ParameterValue*.

static *ParameterValueNNPtr* **createFilename**(const std::string &stringValueIn)

Instantiate a *ParameterValue* from a filename.

**Returns**
a new *ParameterValue*.

class **OperationParameterValue** : public osgeo::proj::*operation*::*GeneralParameterValue*

*#include <coordinateoperation.hpp>* A parameter value, ordered sequence of values, or reference to a file of parameter values.

This combines a *OperationParameter* with the corresponding *ParameterValue*.

**Remark**

Implements *OperationParameterValue* from *ISO 19111:2019*

### Public Functions

const *OperationParameterNNPtr* &**parameter**()

Return the parameter (definition)

**Returns**

the parameter (definition).

const *ParameterValueNNPtr* &**parameterValue**()

Return the parameter value.

**Returns**

the parameter value.

### Public Static Functions

static *OperationParameterValueNNPtr* **create**(const *OperationParameterNNPtr* &parameterIn, const *ParameterValueNNPtr* &valueIn)

Instantiate a *OperationParameterValue*.

**Parameters**

- **parameterIn** -- Parameter (definition).
- **valueIn** -- Parameter value.

**Returns**

a new *OperationParameterValue*.

class **OperationMethod** : public osgeo::proj::*common*::*IdentifiedObject*, public osgeo::proj::*io*::*IJSONExportable*

*#include <coordinateoperation.hpp>* The method (algorithm or procedure) used to perform the coordinate operation.

For a projection method, this contains the name of the projection method and the name of the projection parameters.

**Remark**

Implements *OperationMethod* from *ISO 19111:2019*

### Public Functions

const *util*::*optional*<std::string> &**formula**()

Return the formula(s) or procedure used by this coordinate operation method.

This may be a reference to a publication (in which case use *formulaCitation()*).

Note that the operation method may not be analytic, in which case this attribute references or contains the procedure, not an analytic formula.

**Returns**
the formula, or empty.

const *util*::*optional*<*metadata*::*Citation*> &**formulaCitation**()

Return a reference to a publication giving the formula(s) or procedure used by the coordinate operation method.

**Returns**
the formula citation, or empty.

const std::vector<*GeneralOperationParameterNNPtr*> &**parameters**()

Return the parameters of this operation method.

**Returns**
the parameters.

int **getEPSGCode**()

Return the EPSG code, either directly, or through the name.

**Returns**
code, or 0 if not found

### Public Static Functions

static *OperationMethodNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::vector<*GeneralOperationParameterNNPtr*> &parameters)

Instantiate a operation method from a vector of *GeneralOperationParameter*.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **parameters** -- Vector of GeneralOperationParameterNNPtr.

**Returns**
a new *OperationMethod*.

static *OperationMethodNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::vector<*OperationParameterNNPtr*> &parameters)

Instantiate a operation method from a vector of *OperationParameter*.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **parameters** -- Vector of OperationParameterNNPtr.

**Returns**
a new *OperationMethod*.

class **InvalidOperation** : public osgeo::proj::*util*::*Exception*

*#include <coordinateoperation.hpp>* Exception that can be thrown when an invalid operation is attempted to be constructed.

class **SingleOperation** : public virtual osgeo::proj::*operation*::*CoordinateOperation*

*#include <coordinateoperation.hpp>* A single (not concatenated) coordinate operation (*CoordinateOperation*)

**Remark**

Implements *SingleOperation* from *ISO 19111:2019*

Subclassed by *osgeo::proj::operation::Conversion*, *osgeo::proj::operation::PointMotionOperation*, *osgeo::proj::operation::Transformation*

### Public Functions

const std::vector<*GeneralParameterValueNNPtr*> &**parameterValues**()

Return the parameter values.

**Returns**
the parameter values.

const *OperationMethodNNPtr* &**method**()

Return the operation method associated to the operation.

**Returns**
the operation method.

const *ParameterValuePtr* &**parameterValue**(const std::string &paramName, int epsg_code = 0) const noexcept

Return the parameter value corresponding to a parameter name or EPSG code.

**Parameters**
- **paramName** -- the parameter name (or empty, in which case epsg_code should be non zero)
- **epsg_code** -- the parameter EPSG code (possibly zero)

**Returns**
the value, or nullptr if not found.

const *ParameterValuePtr* &**parameterValue**(int epsg_code) const noexcept

Return the parameter value corresponding to a EPSG code.

**Parameters**
**epsg_code** -- the parameter EPSG code

**Returns**
the value, or nullptr if not found.

const *common*::*Measure* &**parameterValueMeasure**(const std::string &paramName, int epsg_code = 0) const noexcept

Return the parameter value, as a measure, corresponding to a parameter name or EPSG code.

**Parameters**
- **paramName** -- the parameter name (or empty, in which case epsg_code should be non zero)
- **epsg_code** -- the parameter EPSG code (possibly zero)

**Returns**
the measure, or the empty Measure() object if not found.

const *common*::*Measure* &**parameterValueMeasure**(int epsg_code) const noexcept

Return the parameter value, as a measure, corresponding to a EPSG code.

**Parameters**
**epsg_code** -- the parameter EPSG code

**Returns**
the measure, or the empty Measure() object if not found.

virtual std::set<*GridDescription*> **gridsNeeded**(const *io*::*DatabaseContextPtr* &databaseContext, bool considerKnownGridsAsAvailable) const override

Return grids needed by an operation.

std::list<std::string> **validateParameters**() const

Validate the parameters used by a coordinate operation.

Return whether the method is known or not, or a list of missing or extra parameters for the operations recognized by this implementation.

*TransformationNNPtr* **substitutePROJAlternativeGridNames**(*io*::*DatabaseContextNNPtr* databaseContext) const

Return an equivalent transformation to the current one, but using PROJ alternative grid names.

### Public Static Functions

static *SingleOperationNNPtr* **createPROJBased**(const *util*::*PropertyMap* &properties, const std::string &PROJString, const *crs*::*CRSPtr* &sourceCRS, const *crs*::*CRSPtr* &targetCRS, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies = std::vector<*metadata*::*PositionalAccuracyNNPtr*>())

Instantiate a PROJ-based single operation.

> **Note**
>
> The operation might internally be a pipeline chaining several operations. The use of the *SingleOperation* modeling here is mostly to be able to get the PROJ string as a parameter.

**Parameters**

- **properties** -- Properties
- **PROJString** -- the PROJ string.
- **sourceCRS** -- source CRS (might be null).
- **targetCRS** -- target CRS (might be null).
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**

the new instance

class **Conversion** : public osgeo::proj::*operation*::*SingleOperation*

*#include <coordinateoperation.hpp>* A mathematical operation on coordinates in which the parameter values are defined rather than empirically derived.

Application of the coordinate conversion introduces no error into output coordinates. The best-known example of a coordinate conversion is a map projection. For coordinate conversions the output coordinates are referenced to the same datum as are the input coordinates.

Coordinate conversions forming a component of a derived CRS have a source *crs::CRS* and a target *crs::CRS* that are NOT specified through the source and target associations, but through associations from *crs::DerivedCRS* to *crs::SingleCRS*.

**Remark**

Implements *Conversion* from *ISO 19111:2019*

### Projection parameters

#### Co-latitude of cone axis

The rotation applied to spherical coordinates for the oblique projection, measured on the conformal sphere in the plane of the meridian of origin.

EPSG:1036

#### Latitude of natural origin/Center Latitude

The latitude of the point from which the values of both the geographical coordinates on the ellipsoid and the grid coordinates on the projection are deemed to increment or decrement for computational purposes. Alternatively it may be considered as the latitude of the point which in the absence of application of false coordinates has grid coordinates of (0,0).

EPSG:8801

#### Longitude of natural origin/Central Meridian

The longitude of the point from which the values of both the geographical coordinates on the ellipsoid and the grid coordinates on the projection are deemed to increment or decrement for computational purposes. Alternatively it may be considered as the longitude of the point which in the absence of application of false coordinates has grid coordinates of (0,0). Sometimes known as "central meridian (CM)".

EPSG:8802

#### Scale Factor

The factor by which the map grid is reduced or enlarged during the projection process, defined by its value at the natural origin.

EPSG:8805

#### False Easting

Since the natural origin may be at or near the centre of the projection and under normal coordinate circumstances would thus give rise to negative coordinates over parts of the mapped area, this origin is usually given false coordinates which are large enough to avoid this inconvenience. The False Easting, FE, is the value assigned to the abscissa (east or west) axis of the projection grid at the natural origin.

EPSG:8806

#### False Northing

Since the natural origin may be at or near the centre of the projection and under normal coordinate circumstances would thus give rise to negative coordinates over parts of the mapped area, this origin is usually given false coordinates which are large enough to avoid this inconvenience. The False Northing, FN, is the value assigned to the ordinate (north or south) axis of the projection grid at the natural origin.

EPSG:8807

#### Latitude of projection centre

For an oblique projection, this is the latitude of the point at which the azimuth of the central line is defined.

EPSG:8811

#### Longitude of projection centre

For an oblique projection, this is the longitude of the point at which the azimuth of the central line is defined.

EPSG:8812

#### Azimuth of initial line

The azimuthal direction (north zero, east of north being positive) of the great circle which is the centre line of an oblique projection. The azimuth is given at the projection centre.

EPSG:8813

#### Angle from Rectified to Skew Grid

The angle at the natural origin of an oblique projection through which the natural coordinate reference system is rotated to make the projection north axis parallel with true north.

EPSG:8814

#### Scale factor on initial line

The factor by which the map grid is reduced or enlarged during the projection process, defined by its value at the projection center.

EPSG:8815

#### Easting at projection centre

The easting value assigned to the projection centre.

EPSG:8816

#### Northing at projection centre

The northing value assigned to the projection centre.

EPSG:8817

#### Latitude of pseudo standard

parallel

Latitude of the parallel on which the conic or cylindrical projection is based. This latitude is not geographic, but is defined on the conformal sphere AFTER its rotation to obtain the oblique aspect of the projection.

EPSG:8818

#### Scale factor on pseudo

standard parallel

The factor by which the map grid is reduced or enlarged during the projection process, defined by its value at the pseudo-standard parallel. EPSG:8819

#### Latitude of false origin

The latitude of the point which is not the natural origin and at which grid coordinate values false easting and false northing are defined.

EPSG:8821

#### Longitude of false origin

The longitude of the point which is not the natural origin and at which grid coordinate values false easting and false northing are defined.

EPSG:8822

#### Latitude of 1st standard parallel

For a conic projection with two standard parallels, this is the latitude of one of the parallels of intersection of the cone with the ellipsoid. It is normally but not necessarily that nearest to the pole. Scale is true along this parallel.

EPSG:8823

#### Latitude of 2nd standard parallel

For a conic projection with two standard parallels, this is the latitude of one of the parallels at which the cone intersects with the ellipsoid. It is normally but not necessarily that nearest to the equator. Scale is true along this parallel.

EPSG:8824

#### Easting of false origin

The easting value assigned to the false origin.

EPSG:8826

#### Northing of false origin

The northing value assigned to the false origin.

EPSG:8827

#### Latitude of standard parallel

For polar aspect azimuthal projections, the parallel on which the scale factor is defined to be unity.

EPSG:8832

### Longitude of origin

For polar aspect azimuthal projections, the meridian along which the northing axis increments and also across which parallels of latitude increment towards the north pole.

EPSG:8833

### Public Functions

virtual *CoordinateOperationNNPtr* **inverse**() const override

Return the inverse of the coordinate operation.

**Throws**

*util*::*UnsupportedOperationException* -- if inverse is not available

bool **isUTM**(int &zone, bool &north) const

Return whether a conversion is a Universal Transverse Mercator conversion.

**Parameters**

- **zone** -- **[out]** UTM zone number between 1 and 60.
- **north** -- **[out]** true for UTM northern hemisphere, false for UTM southern hemisphere.

**Returns**

true if it is a UTM conversion.

*ConversionNNPtr* **identify**() const

Return a *Conversion* object where some parameters are better identified.

**Returns**

a new *Conversion*.

*ConversionPtr* **convertToOtherMethod**(int targetEPSGCode) const

Return an equivalent projection.

Currently implemented:

- EPSG_CODE_METHOD_MERCATOR_VARIANT_A (1SP) to EPSG_CODE_METHOD_MERCATOR_VARIANT_B (2SP)
- EPSG_CODE_METHOD_MERCATOR_VARIANT_B (2SP) to EPSG_CODE_METHOD_MERCATOR_VARIANT_A (1SP)
- EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_1SP to EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_2SP
- EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_2SP to EPSG_CODE_METHOD_LAMBERT_CONIC_CONFORMAL_1SP

**Parameters**

**targetEPSGCode** -- EPSG code of the target method.

**Returns**

new conversion, or nullptr

### Public Static Functions

static *ConversionNNPtr* **create**(const *util*::*PropertyMap* &properties, const *OperationMethodNNPtr* &methodIn, const std::vector<*GeneralParameterValueNNPtr*> &values)

Instantiate a *Conversion* from a vector of *GeneralParameterValue*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **methodIn** -- the operation method.
- **values** -- the values.

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **create**(const *util*::*PropertyMap* &propertiesConversion, const *util*::*PropertyMap* &propertiesOperationMethod, const std::vector<*OperationParameterNNPtr*> &parameters, const std::vector<*ParameterValueNNPtr*> &values)

Instantiate a *Conversion* and its *OperationMethod*.

**Parameters**
- **propertiesConversion** -- See *General properties* of the conversion. At minimum the name should be defined.
- **propertiesOperationMethod** -- See *General properties* of the operation method. At minimum the name should be defined.
- **parameters** -- the operation parameters.
- **values** -- the operation values. Constraint: values.size() == parameters.size()

**Throws**
*InvalidOperation* -- if the object cannot be constructed.

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createUTM**(const *util*::*PropertyMap* &properties, int zone, bool north)

Instantiate a Universal Transverse Mercator conversion.

UTM is a family of conversions, of EPSG codes from 16001 to 16060 for the northern hemisphere, and 17001 to 17060 for the southern hemisphere, based on the Transverse Mercator projection method.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **zone** -- UTM zone number between 1 and 60.
- **north** -- true for UTM northern hemisphere, false for UTM southern hemisphere.

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createTransverseMercator**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Transverse Mercator projection method.

This method is defined as EPSG:9807.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGaussSchreiberTransverseMercator**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Gauss Schreiber Transverse Mercator projection method.

This method is also known as Gauss-Laborde Reunion.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createTransverseMercatorSouthOriented**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Transverse Mercator South Orientated projection method.

This method is defined as EPSG:9808.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createTwoPointEquidistant**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFirstPoint, const *common*::*Angle* &longitudeFirstPoint, const *common*::*Angle* &latitudeSecondPoint, const *common*::*Angle* &longitudeSeconPoint, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Two Point Equidistant projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFirstPoint** -- Latitude of first point.
- **longitudeFirstPoint** -- Longitude of first point.
- **latitudeSecondPoint** -- Latitude of second point.
- **longitudeSeconPoint** -- Longitude of second point.
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createTunisiaMappingGrid**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Tunisia Mapping Grid projection method.

This method is defined as EPSG:9816.

*Deprecated:*

. Use *createTunisiaMiningGrid()* instead

**Note**

There is currently no implementation of the method formulas in PROJ.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createTunisiaMiningGrid**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Tunisia Mining Grid projection method.

This method is defined as EPSG:9816.

**Since**

9.2

**Note**

There is currently no implementation of the method formulas in PROJ.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createAlbersEqualArea**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFalseOrigin, const *common*::*Angle* &longitudeFalseOrigin, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &latitudeSecondParallel, const *common*::*Length* &eastingFalseOrigin, const *common*::*Length* &northingFalseOrigin)

Instantiate a conversion based on the Albers Conic Equal Area projection method.

This method is defined as EPSG:9822.

**Note**

the order of arguments is conformant with the corresponding EPSG mode and different than OGRSpatialReference::setACEA() of GDAL <= 2.3

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFalseOrigin** -- See *Latitude of false origin*
- **longitudeFalseOrigin** -- See *Longitude of false origin*
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*
- **latitudeSecondParallel** -- See *Latitude of 2nd standard parallel*
- **eastingFalseOrigin** -- See *Easting of false origin*
- **northingFalseOrigin** -- See *Northing of false origin*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLambertConicConformal_1SP**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Lambert Conic Conformal 1SP projection method.

This method is defined as EPSG:9801.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.

- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createLambertConicConformal_1SP_VariantB**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeNatOrigin, const *common*::*Scale* &scale, const *common*::*Angle* &latitudeFalseOrigin, const *common*::*Angle* &longitudeFalseOrigin, const *common*::*Length* &eastingFalseOrigin, const *common*::*Length* &northingFalseOrigin)

Instantiate a conversion based on the Lambert Conic Conformal 1SP Variant B projection method.

This method is defined as EPSG:1102.

**Since**
9.2.1

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeNatOrigin** -- See *Latitude of natural origin/Center Latitude*
- **scale** -- See *Scale Factor*
- **latitudeFalseOrigin** -- See *Latitude of false origin*
- **longitudeFalseOrigin** -- See *Longitude of false origin*
- **eastingFalseOrigin** -- See *Easting of false origin*
- **northingFalseOrigin** -- See *Northing of false origin*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createLambertConicConformal_2SP**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFalseOrigin, const *common*::*Angle* &longitudeFalseOrigin, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &latitudeSecondParallel, const *common*::*Length* &eastingFalseOrigin, const *common*::*Length* &northingFalseOrigin)

Instantiate a conversion based on the Lambert Conic Conformal 2SP projection method.

This method is defined as EPSG:9802.

> **Note**
>
> the order of arguments is conformant with the corresponding EPSG mode and different than OGRSpatialReference::setLCC() of GDAL <= 2.3

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFalseOrigin** -- See *Latitude of false origin*
- **longitudeFalseOrigin** -- See *Longitude of false origin*
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*
- **latitudeSecondParallel** -- See *Latitude of 2nd standard parallel*
- **eastingFalseOrigin** -- See *Easting of false origin*
- **northingFalseOrigin** -- See *Northing of false origin*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLambertConicConformal_2SP_Michigan**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFalseOrigin, const *common*::*Angle* &longitudeFalseOrigin, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &latitudeSecondParallel, const *common*::*Length* &eastingFalseOrigin, const *common*::*Length* &northingFalseOrigin, const *common*::*Scale* &ellipsoidScalingFactor)

Instantiate a conversion based on the Lambert Conic Conformal (2SP Michigan) projection method.

This method is defined as EPSG:1051.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFalseOrigin** -- See *Latitude of false origin*
- **longitudeFalseOrigin** -- See *Longitude of false origin*
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*
- **latitudeSecondParallel** -- See *Latitude of 2nd standard parallel*
- **eastingFalseOrigin** -- See *Easting of false origin*
- **northingFalseOrigin** -- See *Northing of false origin*
- **ellipsoidScalingFactor** -- Ellipsoid scaling factor.

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLambertConicConformal_2SP_Belgium**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFalseOrigin, const *common*::*Angle* &longitudeFalseOrigin, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &latitudeSecondParallel, const *common*::*Length* &eastingFalseOrigin, const *common*::*Length* &northingFalseOrigin)

Instantiate a conversion based on the Lambert Conic Conformal (2SP Belgium) projection method.

This method is defined as EPSG:9803.

**Note**

the order of arguments is conformant with the corresponding EPSG mode and different than OGRSpatialReference::setLCCB() of GDAL <= 2.3

**Warning**

The formulas used currently in PROJ are, incorrectly, the ones of the regular LCC_2SP method.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFalseOrigin** -- See *Latitude of false origin*
- **longitudeFalseOrigin** -- See *Longitude of false origin*
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*
- **latitudeSecondParallel** -- See *Latitude of 2nd standard parallel*
- **eastingFalseOrigin** -- See *Easting of false origin*
- **northingFalseOrigin** -- See *Northing of false origin*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createAzimuthalEquidistant**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeNatOrigin, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Azimuthal Equidistant projection method.

This method is defined as EPSG:1125.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeNatOrigin** -- See *Latitude of natural origin/Center Latitude*
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*

- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createGuamProjection**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeNatOrigin, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Guam Projection method.

This method is defined as EPSG:9831.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeNatOrigin** -- See *Latitude of natural origin/Center Latitude*
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createBonne**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeNatOrigin, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Bonne projection method.

This method is defined as EPSG:9827.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeNatOrigin** -- See *Latitude of natural origin/Center Latitude* . PROJ calls its the standard parallel 1.
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLambertCylindricalEqualAreaSpherical**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Lambert Cylindrical Equal Area (Spherical) projection method.

This method is defined as EPSG:9834.

> **Warning**
>
> The PROJ cea computation code would select the ellipsoidal form if a non-spherical ellipsoid is used for the base GeographicCRS.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*.
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLambertCylindricalEqualArea**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Lambert Cylindrical Equal Area (ellipsoidal form) projection method.

This method is defined as EPSG:9835.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*.
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createCassiniSoldner**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Cassini-Soldner projection method.

This method is defined as EPSG:9806.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEquidistantConic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFalseOrigin, const *common*::*Angle* &longitudeFalseOrigin, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &latitudeSecondParallel, const *common*::*Length* &eastingFalseOrigin, const *common*::*Length* &northingFalseOrigin)

Instantiate a conversion based on the Equidistant Conic projection method.

This method is defined as EPSG:1119.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFalseOrigin** -- See *Latitude of false origin*
- **longitudeFalseOrigin** -- See *Longitude of false origin*
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*
- **latitudeSecondParallel** -- See *Latitude of 2nd standard parallel*
- **eastingFalseOrigin** -- See *Easting of false origin*
- **northingFalseOrigin** -- See *Northing of false origin*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEckertI**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Eckert I projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEckertII**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Eckert II projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEckertIII**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Eckert III projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEckertIV**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Eckert IV projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEckertV**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Eckert V projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEckertVI**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Eckert VI projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEquidistantCylindrical**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Equidistant Cylindrical projection method.

This is also known as the Equirectangular method, and in the particular case where the latitude of first parallel is 0.

This method is defined as EPSG:1028.

**Note**

This is the equivalent OGRSpatialReference::SetEquirectangular2( 0.0, latitudeFirstParallel, falseEasting, falseNorthing ) of GDAL <= 2.3, where the lat_0 / center_latitude parameter is forced to 0.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*.
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEquidistantCylindricalSpherical**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Equidistant Cylindrical (Spherical) projection method.

This is also known as the Equirectangular method, and in the particular case where the latitude of first parallel is 0.

This method is defined as EPSG:1029.

**Note**

This is the equivalent OGRSpatialReference::SetEquirectangular2( 0.0, latitudeFirstParallel, falseEasting, falseNorthing ) of GDAL <= 2.3, where the lat_0 / center_latitude parameter is forced to 0.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.

- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*.
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGall**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Gall (Stereographic) projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGoodeHomolosine**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Goode Homolosine projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createInterruptedGoodeHomolosine**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Interrupted Goode Homolosine projection method.

There is no equivalent in EPSG.

**Note**

OGRSpatialReference::SetIGH() of GDAL <= 2.3 assumes the 3 projection parameters to be zero and this is the nominal case.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*

- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGeostationarySatelliteSweepX**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &height, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Geostationary Satellite View projection method, with the sweep angle axis of the viewing instrument being x.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **height** -- Height of the view point above the Earth.
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGeostationarySatelliteSweepY**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &height, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Geostationary Satellite View projection method, with the sweep angle axis of the viewing instrument being y.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **height** -- Height of the view point above the Earth.
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGnomonic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Gnomonic projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createHotineObliqueMercatorVariantA**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeProjectionCentre, const *common*::*Angle* &longitudeProjectionCentre, const *common*::*Angle* &azimuthInitialLine, const *common*::*Angle* &angleFromRectifiedToSkrewGrid, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Hotine Oblique Mercator (Variant A) projection method.

This is the variant with the no_uoff parameter, which corresponds to GDAL >=2.3 Hotine_Oblique_Mercator projection. In this variant, the false grid coordinates are defined at the intersection of the initial line and the aposphere (the equator on one of the intermediate surfaces inherent in the method), that is at the natural origin of the coordinate system).

This method is defined as EPSG:9812.

> **Note**
>
> In the case where azimuthInitialLine = angleFromRectifiedToSkrewGrid = 90deg, this maps to the Swiss Oblique Mercator formulas.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeProjectionCentre** -- See *Latitude of projection centre*
- **longitudeProjectionCentre** -- See *Longitude of projection centre*
- **azimuthInitialLine** -- See *Azimuth of initial line*
- **angleFromRectifiedToSkrewGrid** -- See *Angle from Rectified to Skew Grid*
- **scale** -- See *Scale factor on initial line*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createHotineObliqueMercatorVariantB**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeProjectionCentre, const *common*::*Angle* &longitudeProjectionCentre, const *common*::*Angle* &azimuthInitialLine, const *common*::*Angle* &angleFromRectifiedToSkrewGrid, const *common*::*Scale* &scale, const *common*::*Length* &eastingProjectionCentre, const *common*::*Length* &northingProjectionCentre)

Instantiate a conversion based on the Hotine Oblique Mercator (Variant B) projection method.

This is the variant without the no_uoff parameter, which corresponds to GDAL >=2.3 Hotine_Oblique_Mercator_Azimuth_Center projection. In this variant, the false grid coordinates are defined at the projection centre.

This method is defined as EPSG:9815.

**Note**

In the case where azimuthInitialLine = angleFromRectifiedToSkrewGrid = 90deg, this maps to the Swiss Oblique Mercator formulas.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeProjectionCentre** -- See *Latitude of projection centre*
- **longitudeProjectionCentre** -- See *Longitude of projection centre*
- **azimuthInitialLine** -- See *Azimuth of initial line*
- **angleFromRectifiedToSkrewGrid** -- See *Angle from Rectified to Skew Grid*
- **scale** -- See *Scale factor on initial line*
- **eastingProjectionCentre** -- See *Easting at projection centre*
- **northingProjectionCentre** -- See *Northing at projection centre*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createHotineObliqueMercatorTwoPointNaturalOrigin**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeProjectionCentre, const *common*::*Angle* &latitudePoint1, const *common*::*Angle* &longitudePoint1, const *common*::*Angle* &latitudePoint2, const *common*::*Angle* &longitudePoint2, const *common*::*Scale* &scale, const *common*::*Length* &eastingProjectionCentre, const *common*::*Length* &northingProjectionCentre)

Instantiate a conversion based on the Hotine Oblique Mercator Two Point Natural Origin projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeProjectionCentre** -- See *Latitude of projection centre*
- **latitudePoint1** -- Latitude of point 1.
- **longitudePoint1** -- Latitude of point 1.
- **latitudePoint2** -- Latitude of point 2.
- **longitudePoint2** -- Longitude of point 2.
- **scale** -- See *Scale factor on initial line*
- **eastingProjectionCentre** -- See *Easting at projection centre*
- **northingProjectionCentre** -- See *Northing at projection centre*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLabordeObliqueMercator**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeProjectionCentre, const *common*::*Angle* &longitudeProjectionCentre, const *common*::*Angle* &azimuthInitialLine, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Laborde Oblique Mercator projection method.

This method is defined as EPSG:9813.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeProjectionCentre** -- See *Latitude of projection centre*
- **longitudeProjectionCentre** -- See *Longitude of projection centre*
- **azimuthInitialLine** -- See *Azimuth of initial line*
- **scale** -- See *Scale factor on initial line*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createInternationalMapWorldPolyconic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &latitudeSecondParallel, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the International Map of the World Polyconic projection method.

There is no equivalent in EPSG.

**Note**

the order of arguments is conformant with the corresponding EPSG mode and different than OGRSpatialReference::SetIWMPolyconic() of GDAL <= 2.3

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*
- **latitudeSecondParallel** -- See *Latitude of 2nd standard parallel*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createKrovakNorthOriented**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeProjectionCentre, const *common*::*Angle* &longitudeOfOrigin, const *common*::*Angle* &colatitudeConeAxis, const *common*::*Angle* &latitudePseudoStandardParallel, const *common*::*Scale* &scaleFactorPseudoStandardParallel, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Krovak (north oriented) projection method.

This method is defined as EPSG:1041.

The coordinates are returned in the "GIS friendly" order: easting, northing. This method is similar to *createKrovak()*, except that the later returns projected values as southing, westing, where southing(Krovak) = -northing(Krovak_North) and westing(Krovak) = -easting(Krovak_North).

**Note**

The current PROJ implementation of Krovak hard-codes colatitudeConeAxis = 30deg17'17.30311" and latitudePseudoStandardParallel = 78deg30'N, which are the values used for the ProjectedCRS S-JTSK (Ferro) / Krovak East North (EPSG:5221). It also hard-codes the parameters of the Bessel ellipsoid typically used for Krovak.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeProjectionCentre** -- See *Latitude of projection centre*
- **longitudeOfOrigin** -- See *Longitude of origin*
- **colatitudeConeAxis** -- See *Co-latitude of cone axis*
- **latitudePseudoStandardParallel** -- See *Latitude of pseudo standard*
- **scaleFactorPseudoStandardParallel** -- See *Scale factor on pseudo*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createKrovak**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeProjectionCentre, const *common*::*Angle* &longitudeOfOrigin, const *common*::*Angle* &colatitudeConeAxis, const *common*::*Angle* &latitudePseudoStandardParallel, const *common*::*Scale* &scaleFactorPseudoStandardParallel, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Krovak projection method.

This method is defined as EPSG:9819.

The coordinates are returned in the historical order: southing, westing This method is similar to *createKrovakNorthOriented()*, except that the later returns projected values as easting, northing, where easting(Krovak_North) = -westing(Krovak) and northing(Krovak_North) = -southing(Krovak).

**Note**

The current PROJ implementation of Krovak hard-codes colatitudeConeAxis = 30deg17'17.30311" and latitudePseudoStandardParallel = 78deg30'N, which are the values used for the ProjectedCRS S-JTSK (Ferro) / Krovak East North (EPSG:5221). It also hard-codes the parameters of the Bessel ellipsoid typically used for Krovak.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeProjectionCentre** -- See *Latitude of projection centre*
- **longitudeOfOrigin** -- See *Longitude of origin*
- **colatitudeConeAxis** -- See *Co-latitude of cone axis*
- **latitudePseudoStandardParallel** -- See *Latitude of pseudo standard*
- **scaleFactorPseudoStandardParallel** -- See *Scale factor on pseudo*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLambertAzimuthalEqualArea**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeNatOrigin, const *common*::*Angle* &longitudeNatOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Lambert Azimuthal Equal Area projection method.

This method is defined as EPSG:9820.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeNatOrigin** -- See *Latitude of natural origin/Center Latitude*
- **longitudeNatOrigin** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createMillerCylindrical**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Miller Cylindrical projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createMercatorVariantA**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Mercator (variant A) projection method.

This is the A variant, also known as Mercator (1SP), defined with the scale factor. Note that latitude of natural origin (centerLat) is a parameter, but unused in the transformation formulas.

This method is defined as EPSG:9804.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude* . Should be 0.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createMercatorVariantB**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeFirstParallel, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Mercator (variant B) projection method.

This is the B variant, also known as Mercator (2SP), defined with the latitude of the first standard parallel (the second standard parallel is implicitly the opposite value). The latitude of natural origin is fixed to zero.

This method is defined as EPSG:9805.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeFirstParallel** -- See *Latitude of 1st standard parallel*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createPopularVisualisationPseudoMercator**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Popular Visualisation Pseudo Mercator projection method.

Also known as WebMercator. Mostly/only used for Projected CRS EPSG:3857 (WGS 84 / Pseudo-Mercator)

This method is defined as EPSG:1024.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude* . Usually 0
- **centerLong** -- See *Longitude of natural origin/Central Meridian* . Usually 0
- **falseEasting** -- See *False Easting* . Usually 0
- **falseNorthing** -- See *False Northing* . Usually 0

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createMercatorSpherical**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Mercator projection method, using its spherical formulation.

When used with an ellipsoid, the radius used is the radius of the conformal sphere at centerLat.

This method is defined as EPSG:1026.

**Since**

9.3

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude* . Usually 0
- **centerLong** -- See *Longitude of natural origin/Central Meridian* . Usually 0
- **falseEasting** -- See *False Easting* . Usually 0
- **falseNorthing** -- See *False Northing* . Usually 0

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createMollweide**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Mollweide projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createNewZealandMappingGrid**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the New Zealand Map Grid projection method.

This method is defined as EPSG:9811.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createObliqueStereographic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Oblique Stereographic (alternative) projection method.

This method is defined as EPSG:9809.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createOrthographic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Orthographic projection method.

This method is defined as EPSG:9840.

> **Note**
>
> Before PROJ 7.2, only the spherical formulation was implemented.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createLocalOrthographic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Angle* &azimuthInitialLine, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Orthographic projection method.

This method is defined as EPSG:1130.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **azimuthInitialLine** -- See *Azimuth of initial line*
- **scale** -- See *Scale factor on initial line*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createAmericanPolyconic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the American Polyconic projection method.

This method is defined as EPSG:9818.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createPolarStereographicVariantA**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Polar Stereographic (Variant A) projection method.

This method is defined as EPSG:9810.

This is the variant of polar stereographic defined with a scale factor.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.

- **centerLat** -- See *Latitude of natural origin/Center Latitude* . Should be 90 deg ou -90 deg.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createPolarStereographicVariantB**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeStandardParallel, const *common*::*Angle* &longitudeOfOrigin, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Polar Stereographic (Variant B) projection method.

This method is defined as EPSG:9829.

This is the variant of polar stereographic defined with a latitude of standard parallel.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeStandardParallel** -- See *Latitude of standard parallel*
- **longitudeOfOrigin** -- See *Longitude of origin*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createRobinson**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Robinson projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createSinusoidal**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Sinusoidal projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*

- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createStereographic**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Scale* &scale, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Stereographic projection method.

There is no equivalent in EPSG. This method implements the original "Oblique

Stereographic" method described in "Snyder's Map Projections - A Working

manual", which is different from the "Oblique Stereographic (alternative)" method implemented in *createObliqueStereographic()*.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **scale** -- See *Scale Factor*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createVanDerGrinten**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Van der Grinten projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createWagnerI**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Wagner I projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*

- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createWagnerII**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Wagner II projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createWagnerIII**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &latitudeTrueScale, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Wagner III projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **latitudeTrueScale** -- Latitude of true scale.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createWagnerIV**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Wagner IV projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createWagnerV**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Wagner V projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createWagnerVI**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Wagner VI projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createWagnerVII**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Wagner VII projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createQuadrilateralizedSphericalCube**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLat, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Quadrilateralized Spherical Cube projection method.

There is no equivalent in EPSG.

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLat** -- See *Latitude of natural origin/Center Latitude*
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*

- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createSphericalCrossTrackHeight**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &pegPointLat, const *common*::*Angle* &pegPointLong, const *common*::*Angle* &pegPointHeading, const *common*::*Length* &pegPointHeight)

Instantiate a conversion based on the Spherical Cross-Track Height projection method.

There is no equivalent in EPSG.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **pegPointLat** -- Peg point latitude.
- **pegPointLong** -- Peg point longitude.
- **pegPointHeading** -- Peg point heading.
- **pegPointHeight** -- Peg point height.

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createEqualEarth**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &centerLong, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Equal Earth projection method.

This method is defined as EPSG:1078.

**Parameters**

- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **centerLong** -- See *Longitude of natural origin/Central Meridian*
- **falseEasting** -- See *False Easting*
- **falseNorthing** -- See *False Northing*

**Returns**

a new *Conversion*.

static *ConversionNNPtr* **createVerticalPerspective**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &topoOriginLat, const *common*::*Angle* &topoOriginLong, const *common*::*Length* &topoOriginHeight, const *common*::*Length* &viewPointHeight, const *common*::*Length* &falseEasting, const *common*::*Length* &falseNorthing)

Instantiate a conversion based on the Vertical Perspective projection method.

This method is defined as EPSG:9838.

The PROJ implementation of the EPSG Vertical Perspective has the current limitations with respect to the method described in EPSG:

- it is a 2D-only method, ignoring the ellipsoidal height of the point to project.
- it has only a spherical development.
- the height of the topocentric origin is ignored, and thus assumed to be 0.

For completeness, PROJ adds the falseEasting and falseNorthing parameter, which are not described in EPSG. They should usually be set to 0.

**Since**
6.3

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **topoOriginLat** -- Latitude of topocentric origin
- **topoOriginLong** -- Longitude of topocentric origin
- **topoOriginHeight** -- Ellipsoidal height of topocentric origin. Ignored by PROJ (that is assumed to be 0)
- **viewPointHeight** -- Viewpoint height with respect to the topocentric/mapping plane. In the case where topoOriginHeight = 0, this is the height above the ellipsoid surface at topoOriginLat, topoOriginLong.
- **falseEasting** -- See *False Easting* . (not in EPSG)
- **falseNorthing** -- See *False Northing* . (not in EPSG)

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createPoleRotationGRIBConvention**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &southPoleLatInUnrotatedCRS, const *common*::*Angle* &southPoleLongInUnrotatedCRS, const *common*::*Angle* &axisRotation)

Instantiate a conversion based on the Pole Rotation method, using the conventions of the GRIB 1 and GRIB 2 data formats.

Those are mentioned in the Note 2 of https://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_doc/grib2_temp3-1.shtml

Several conventions for the pole rotation method exists. The parameters provided in this method are remapped to the PROJ ob_tran operation with:

Another implementation of that convention is also in the netcdf-java library: https://github.com/Unidata/netcdf-java/blob/3ce72c0cd167609ed8c69152bb4a004d1daa9273/cdm/core/src/main/java/ucar/unidata/geoloc/projection/RotatedLatLon.java

The PROJ implementation of this method assumes a spherical ellipsoid.

**Since**
7.0

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **southPoleLatInUnrotatedCRS** -- Latitude of the point from the unrotated CRS, expressed in the unrotated CRS, that will become the south pole of the rotated CRS.
- **southPoleLongInUnrotatedCRS** -- Longitude of the point from the unrotated CRS, expressed in the unrotated CRS, that will become the south pole of the rotated CRS.
- **axisRotation** -- The angle of rotation about the new polar axis (measured clockwise when looking from the southern to the northern pole) of the coordinate system, assuming the new axis to have been obtained by first rotating the sphere through southPoleLongInUnrotatedCRS degrees about the geographic polar axis and then rotating through (90 + southPoleLatInUnrotatedCRS) degrees so that the southern pole moved along the (previously rotated) Greenwich meridian.

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createPoleRotationNetCDFCFConvention**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &gridNorthPoleLatitude, const *common*::*Angle* &gridNorthPoleLongitude, const *common*::*Angle* &northPoleGridLongitude)

Instantiate a conversion based on the Pole Rotation method, using the conventions of the netCDF CF convention for the netCDF format.

Those are mentioned in the Note 2 of https://cfconventions.org/Data/cf-conventions/cf-conventions-1.8/cf-conventions.html#_rotated_pole

Several conventions for the pole rotation method exists. The parameters provided in this method are remapped to the PROJ ob_tran operation with:

Another implementation of that convention is also in the netcdf-java library: https://github.com/Unidata/netcdf-java/blob/3ce72c0cd167609ed8c69152bb4a004d1daa9273/cdm/core/src/main/java/ucar/unidata/geoloc/projection/RotatedPole.java

The PROJ implementation of this method assumes a spherical ellipsoid.

**Since**
8.2

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **gridNorthPoleLatitude** -- True latitude of the north pole of the rotated grid
- **gridNorthPoleLongitude** -- True longitude of the north pole of the rotated grid.
- **northPoleGridLongitude** -- Longitude of the true north pole in the rotated grid.

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createChangeVerticalUnit**(const *util*::*PropertyMap* &properties, const *common*::*Scale* &factor)

Instantiate a conversion based on the Change of Vertical Unit method.

This method is defined as EPSG:1069 [DEPRECATED].

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **factor** -- *Conversion* factor

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createChangeVerticalUnit**(const *util*::*PropertyMap* &properties)

Instantiate a conversion based on the Change of Vertical Unit method (without explicit conversion factor)

This method is defined as EPSG:1104.

**Parameters**
**properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.

**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createHeightDepthReversal**(const *util*::*PropertyMap* &properties)

Instantiate a conversion based on the Height Depth Reversal method.

This method is defined as EPSG:1068.

**Since**
6.3

**Parameters**
**properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createAxisOrderReversal**(bool is3D)

Instantiate a conversion based on the Axis order reversal method.

This swaps the longitude, latitude axis.

This method is defined as EPSG:9843 for 2D or EPSG:9844 for Geographic3D horizontal.
**Parameters**
**is3D** -- Whether this should apply on 3D geographicCRS
**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGeographicGeocentric**(const *util*::*PropertyMap* &properties)

Instantiate a conversion based on the Geographic/Geocentric method.

This method is defined as EPSG:9602.
**Parameters**
**properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
**Returns**
a new *Conversion*.

static *ConversionNNPtr* **createGeographic2DOffsets**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &offsetLat, const *common*::*Angle* &offsetLong)

Instantiate a conversion with method Geographic 2D offsets.

This method is defined as EPSG:9619.
**Parameters**
- **properties** -- See *General properties* of the conversion. At minimum the name should be defined.
- **offsetLat** -- Latitude offset to add.
- **offsetLong** -- Longitude offset to add.

**Returns**
new conversion.

static *ConversionNNPtr* **createGeographic3DOffsets**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &offsetLat, const *common*::*Angle* &offsetLong, const *common*::*Length* &offsetHeight)

Instantiate a conversion with method Geographic 3D offsets.

This method is defined as EPSG:9660.

**Parameters**

- **properties** -- See *General properties* of the *Conversion*. At minimum the name should be defined.
- **offsetLat** -- Latitude offset to add.
- **offsetLong** -- Longitude offset to add.
- **offsetHeight** -- Height offset to add.

**Returns**

new *Conversion*.

static *ConversionNNPtr* **createGeographic2DWithHeightOffsets**(const *util*::*PropertyMap* &properties, const *common*::*Angle* &offsetLat, const *common*::*Angle* &offsetLong, const *common*::*Length* &offsetHeight)

Instantiate a conversion with method Geographic 2D with height offsets.

This method is defined as EPSG:9618.

**Parameters**

- **properties** -- See *General properties* of the *Conversion*. At minimum the name should be defined.
- **offsetLat** -- Latitude offset to add.
- **offsetLong** -- Longitude offset to add.
- **offsetHeight** -- Geoid undulation to add.

**Returns**

new *Conversion*.

static *ConversionNNPtr* **createVerticalOffset**(const *util*::*PropertyMap* &properties, const *common*::*Length* &offsetHeight)

Instantiate a conversion with method Vertical Offset.

This method is defined as EPSG:9616.

**Parameters**

- **properties** -- See *General properties* of the *Conversion*. At minimum the name should be defined.
- **offsetHeight** -- Geoid undulation to add.

**Returns**

new *Conversion*.

class **Transformation** : public osgeo::proj::*operation*::*SingleOperation*

*#include <coordinateoperation.hpp>* A mathematical operation on coordinates in which parameters are empirically derived from data containing the coordinates of a series of points in both coordinate reference systems.

This computational process is usually "over-determined", allowing derivation of error (or accuracy) estimates for the coordinate transformation. Also, the stochastic nature of the parameters may result in multiple (different) versions of the same coordinate transformations between the same source and target CRSs. Any single coordinate operation in which the input and output coordinates are referenced to different datums (reference frames) will be a coordinate transformation.

**Remark**

Implements *Transformation* from *ISO 19111:2019*

### Public Functions

const *crs*::*CRSNNPtr* &**sourceCRS**()

Return the source *crs::CRS* of the transformation.

**Returns**
the source CRS.

const *crs*::*CRSNNPtr* &**targetCRS**()

Return the target *crs::CRS* of the transformation.

**Returns**
the target CRS.

virtual *CoordinateOperationNNPtr* **inverse**() const override

Return the inverse of the coordinate operation.

**Throws**
*util*::*UnsupportedOperationException* -- if inverse is not available

### Public Static Functions

static *TransformationNNPtr* **create**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *crs*::*CRSPtr* &interpolationCRSIn, const *OperationMethodNNPtr* &methodIn, const std::vector<*GeneralParameterValueNNPtr*> &values, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation from a vector of *GeneralParameterValue*.

**Parameters**
- **properties** -- See *General properties*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **interpolationCRSIn** -- Interpolation CRS (might be null)
- **methodIn** -- Operation method.
- **values** -- Vector of GeneralOperationParameterNNPtr.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**
*InvalidOperation* -- if the object cannot be constructed.

**Returns**
new *Transformation*.

static *TransformationNNPtr* **create**(const *util*::*PropertyMap* &propertiesTransformation, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *crs*::*CRSPtr* &interpolationCRSIn, const *util*::*PropertyMap* &propertiesOperationMethod, const std::vector<*OperationParameterNNPtr*> &parameters, const std::vector<*ParameterValueNNPtr*> &values, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation and its *OperationMethod*.

**Parameters**
- **propertiesTransformation** -- The *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.

- **interpolationCRSIn** -- Interpolation CRS (might be null)
- **propertiesOperationMethod** -- The *General properties* of the *OperationMethod*. At minimum the name should be defined.
- **parameters** -- Vector of parameters of the operation method.
- **values** -- Vector of ParameterValueNNPtr. Constraint: values.size() == parameters.size()
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createGeocentricTranslations**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, double translationXMetre, double translationYMetre, double translationZMetre, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with Geocentric Translations method.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **translationXMetre** -- Value of the Translation_X parameter (in metre).
- **translationYMetre** -- Value of the Translation_Y parameter (in metre).
- **translationZMetre** -- Value of the Translation_Z parameter (in metre).
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createPositionVector**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, double translationXMetre, double translationYMetre, double translationZMetre, double rotationXArcSecond, double rotationYArcSecond, double rotationZArcSecond, double scaleDifferencePPM, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with Position vector transformation method.

This is similar to *createCoordinateFrameRotation()*, except that the sign of the rotation terms is inverted.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **translationXMetre** -- Value of the Translation_X parameter (in metre).
- **translationYMetre** -- Value of the Translation_Y parameter (in metre).

- **translationZMetre** -- Value of the Translation_Z parameter (in metre).
- **rotationXArcSecond** -- Value of the Rotation_X parameter (in arc-second).
- **rotationYArcSecond** -- Value of the Rotation_Y parameter (in arc-second).
- **rotationZArcSecond** -- Value of the Rotation_Z parameter (in arc-second).
- **scaleDifferencePPM** -- Value of the Scale_Difference parameter (in parts-per-million).
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createCoordinateFrameRotation**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, double translationXMetre, double translationYMetre, double translationZMetre, double rotationXArcSecond, double rotationYArcSecond, double rotationZArcSecond, double scaleDifferencePPM, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with Coordinate Frame Rotation method.

This is similar to *createPositionVector()*, except that the sign of the rotation terms is inverted.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **translationXMetre** -- Value of the Translation_X parameter (in metre).
- **translationYMetre** -- Value of the Translation_Y parameter (in metre).
- **translationZMetre** -- Value of the Translation_Z parameter (in metre).
- **rotationXArcSecond** -- Value of the Rotation_X parameter (in arc-second).
- **rotationYArcSecond** -- Value of the Rotation_Y parameter (in arc-second).
- **rotationZArcSecond** -- Value of the Rotation_Z parameter (in arc-second).
- **scaleDifferencePPM** -- Value of the Scale_Difference parameter (in parts-per-million).
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createTimeDependentPositionVector**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, double translationXMetre, double translationYMetre, double translationZMetre, double rotationXArcSecond, double rotationYArcSecond, double rotationZArcSecond, double scaleDifferencePPM, double rateTranslationX, double rateTranslationY, double rateTranslationZ, double rateRotationX, double rateRotationY, double rateRotationZ, double rateScaleDifference, double referenceEpochYear, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with Time Dependent position vector transformation method.

This is similar to *createTimeDependentCoordinateFrameRotation()*, except that the sign of the rotation terms is inverted.

This method is defined as EPSG:1053.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **translationXMetre** -- Value of the Translation_X parameter (in metre).
- **translationYMetre** -- Value of the Translation_Y parameter (in metre).
- **translationZMetre** -- Value of the Translation_Z parameter (in metre).
- **rotationXArcSecond** -- Value of the Rotation_X parameter (in arc-second).
- **rotationYArcSecond** -- Value of the Rotation_Y parameter (in arc-second).
- **rotationZArcSecond** -- Value of the Rotation_Z parameter (in arc-second).
- **scaleDifferencePPM** -- Value of the Scale_Difference parameter (in parts-per-million).
- **rateTranslationX** -- Value of the rate of change of X-axis translation (in metre/year)
- **rateTranslationY** -- Value of the rate of change of Y-axis translation (in metre/year)
- **rateTranslationZ** -- Value of the rate of change of Z-axis translation (in metre/year)
- **rateRotationX** -- Value of the rate of change of X-axis rotation (in arc-second/year)
- **rateRotationY** -- Value of the rate of change of Y-axis rotation (in arc-second/year)
- **rateRotationZ** -- Value of the rate of change of Z-axis rotation (in arc-second/year)
- **rateScaleDifference** -- Value of the rate of change of scale difference (in PPM/year)
- **referenceEpochYear** -- Parameter reference epoch (in decimal year)
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createTimeDependentCoordinateFrameRotation**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, double translationXMetre, double translationYMetre, double translationZMetre, double rotationXArcSecond, double rotationYArcSecond, double rotationZArcSecond, double scaleDifferencePPM, double rateTranslationX, double rateTranslationY, double rateTranslationZ, double rateRotationX, double rateRotationY, double rateRotationZ, double rateScaleDifference, double referenceEpochYear, const std::vector<*metadata*::*PositionalAccur*
&accuracies)

Instantiate a transformation with Time Dependent Position coordinate frame rotation transformation method.

This is similar to *createTimeDependentPositionVector()*, except that the sign of the rotation terms is inverted.

This method is defined as EPSG:1056.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **translationXMetre** -- Value of the Translation_X parameter (in metre).
- **translationYMetre** -- Value of the Translation_Y parameter (in metre).
- **translationZMetre** -- Value of the Translation_Z parameter (in metre).

- **rotationXArcSecond** -- Value of the Rotation_X parameter (in arc-second).
- **rotationYArcSecond** -- Value of the Rotation_Y parameter (in arc-second).
- **rotationZArcSecond** -- Value of the Rotation_Z parameter (in arc-second).
- **scaleDifferencePPM** -- Value of the Scale_Difference parameter (in parts-per-million).
- **rateTranslationX** -- Value of the rate of change of X-axis translation (in metre/year)
- **rateTranslationY** -- Value of the rate of change of Y-axis translation (in metre/year)
- **rateTranslationZ** -- Value of the rate of change of Z-axis translation (in metre/year)
- **rateRotationX** -- Value of the rate of change of X-axis rotation (in arc-second/year)
- **rateRotationY** -- Value of the rate of change of Y-axis rotation (in arc-second/year)
- **rateRotationZ** -- Value of the rate of change of Z-axis rotation (in arc-second/year)
- **rateScaleDifference** -- Value of the rate of change of scale difference (in PPM/year)
- **referenceEpochYear** -- Parameter reference epoch (in decimal year)
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createTOWGS84**(const *crs*::*CRSNNPtr* &sourceCRSIn, const std::vector<double> &TOWGS84Parameters)

Instantiate a transformation from TOWGS84 parameters.

This is a helper of *createPositionVector()* with the source CRS being the GeographicCRS of sourceCRSIn, and the target CRS being EPSG:4326

**Parameters**

- **sourceCRSIn** -- Source CRS.
- **TOWGS84Parameters** -- The vector of 3 double values (Translation_X,_Y,_Z) or 7 double values (Translation_X,_Y,_Z, Rotation_X,_Y,_Z, Scale_Difference) passed to *createPositionVector()*

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createNTv2**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const std::string &filename, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with NTv2 method.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **filename** -- NTv2 filename.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createMolodensky**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, double translationXMetre, double translationYMetre, double translationZMetre, double semiMajorAxisDifferenceMetre, double flattingDifference, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with Molodensky method.

This method is defined as EPSG:9604.

**See also**

*createAbridgedMolodensky()* for a related *method*.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **translationXMetre** -- Value of the Translation_X parameter (in metre).
- **translationYMetre** -- Value of the Translation_Y parameter (in metre).
- **translationZMetre** -- Value of the Translation_Z parameter (in metre).
- **semiMajorAxisDifferenceMetre** -- The difference between the semi-major axis values of the ellipsoids used in the target and source CRS (in metre).
- **flattingDifference** -- The difference between the flattening values of the ellipsoids used in the target and source CRS.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createAbridgedMolodensky**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, double translationXMetre, double translationYMetre, double translationZMetre, double semiMajorAxisDifferenceMetre, double flattingDifference, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with Abridged Molodensky method.

This method is defined as EPSG:9605.

**See also**

createdMolodensky() for a related *method*.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **translationXMetre** -- Value of the Translation_X parameter (in metre).
- **translationYMetre** -- Value of the Translation_Y parameter (in metre).
- **translationZMetre** -- Value of the Translation_Z parameter (in metre).
- **semiMajorAxisDifferenceMetre** -- The difference between the semi-major axis values of the ellipsoids used in the target and source CRS (in metre).
- **flattingDifference** -- The difference between the flattening values of the ellipsoids used in the target and source CRS.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createGravityRelatedHeightToGeographic3D**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *crs*::*CRSPtr* &interpolationCRSIn, const std::string &filename, const std::vector<*metadata*::*PositionalAccuracy*
&accuracies)

Instantiate a transformation from GravityRelatedHeight to Geographic3D.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **interpolationCRSIn** -- Interpolation CRS. (might be null)
- **filename** -- GRID filename.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**

new *Transformation*.

static *TransformationNNPtr* **createVERTCON**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const std::string &filename, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with method VERTCON.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.

- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **filename** -- GRID filename.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**
new *Transformation*.

static *TransformationNNPtr* **createLongitudeRotation**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *common*::*Angle* &offset)

Instantiate a transformation with method Longitude rotation.

This method is defined as EPSG:9601.

**Parameters**
- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **offset** -- Longitude offset to add.

**Returns**
new *Transformation*.

static *TransformationNNPtr* **createGeographic2DOffsets**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *common*::*Angle* &offsetLat, const *common*::*Angle* &offsetLong, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with method Geographic 2D offsets.

This method is defined as EPSG:9619.

**Parameters**
- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **offsetLat** -- Latitude offset to add.
- **offsetLong** -- Longitude offset to add.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**
new *Transformation*.

static *TransformationNNPtr* **createGeographic3DOffsets**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *common*::*Angle* &offsetLat, const *common*::*Angle* &offsetLong, const *common*::*Length* &offsetHeight, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with method Geographic 3D offsets.

This method is defined as EPSG:9660.

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **offsetLat** -- Latitude offset to add.
- **offsetLong** -- Longitude offset to add.
- **offsetHeight** -- Height offset to add.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**
new *Transformation*.

static *TransformationNNPtr* **createGeographic2DWithHeightOffsets**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *common*::*Angle* &offsetLat, const *common*::*Angle* &offsetLong, const *common*::*Length* &offsetHeight, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with method Geographic 2D with height offsets.

This method is defined as EPSG:9618.

**Parameters**
- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **offsetLat** -- Latitude offset to add.
- **offsetLong** -- Longitude offset to add.
- **offsetHeight** -- Geoid undulation to add.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**
new *Transformation*.

static *TransformationNNPtr* **createCartesianGridOffsets**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *common*::*Length* &eastingOffset, const *common*::*Length* &northingOffset, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with method Cartesian grid offsets.

This method is defined as EPSG:9656.

**Since**
PROJ 9.5.0

**Parameters**

- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **eastingOffset** -- Easting offset to add.
- **northingOffset** -- Northing offset to add.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**
new *Transformation*.

static *TransformationNNPtr* **createVerticalOffset**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *common*::*Length* &offsetHeight, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation with method Vertical Offset.

This method is defined as EPSG:9616.

**Parameters**
- **properties** -- See *General properties* of the *Transformation*. At minimum the name should be defined.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **offsetHeight** -- Geoid undulation to add.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**
new *Transformation*.

static *TransformationNNPtr* **createChangeVerticalUnit**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &sourceCRSIn, const *crs*::*CRSNNPtr* &targetCRSIn, const *common*::*Scale* &factor, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a transformation based on the Change of Vertical Unit method.

This method is defined as EPSG:1069 [DEPRECATED].

**Parameters**
- **properties** -- See *General properties* of the conversion. If the name is not provided, it is automatically set.
- **sourceCRSIn** -- Source CRS.
- **targetCRSIn** -- Target CRS.
- **factor** -- *Conversion* factor
- **accuracies** -- Vector of positional accuracy (might be empty).

**Returns**
a new *Transformation*.

class **PointMotionOperation** : public osgeo::proj::*operation*::*SingleOperation*

*#include <coordinateoperation.hpp>* A mathematical operation that describes the change of coordinate values within one coordinate reference system due to the motion of the point between one coordinate epoch and another coordinate epoch.

The motion is due to tectonic plate movement or deformation.

**Remark**

Implements *PointMotionOperation* from *ISO 19111:2019*

### Public Functions

const *crs*::*CRSNNPtr* &**sourceCRS**()

Return the source *crs::CRS* of the operation.

**Returns**

the source CRS.

virtual *CoordinateOperationNNPtr* **inverse**() const override

Return the inverse of the coordinate operation.

**Throws**

*util*::*UnsupportedOperationException* -- if inverse is not available

*PointMotionOperationNNPtr* **substitutePROJAlternativeGridNames**(*io*::*DatabaseContextNNPtr* databaseContext) const

Return an equivalent transformation to the current one, but using PROJ alternative grid names.

### Public Static Functions

static *PointMotionOperationNNPtr* **create**(const *util*::*PropertyMap* &properties, const *crs*::*CRSNNPtr* &crsIn, const *OperationMethodNNPtr* &methodIn, const std::vector<*GeneralParameterValueNNPtr*> &values, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a point motion operation from a vector of *GeneralParameterValue*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **crsIn** -- Source and target CRS.
- **methodIn** -- Operation method.
- **values** -- Vector of GeneralOperationParameterNNPtr.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *PointMotionOperation*.

static *PointMotionOperationNNPtr* **create**(const *util*::*PropertyMap* &propertiesOperation, const *crs*::*CRSNNPtr* &crsIn, const *util*::*PropertyMap* &propertiesOperationMethod, const std::vector<*OperationParameterNNPtr*> &parameters, const std::vector<*ParameterValueNNPtr*> &values, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a point motion operation and its *OperationMethod*.

**Parameters**

- **propertiesOperation** -- The *General properties* of the *PointMotionOperation*. At minimum the name should be defined.
- **crsIn** -- Source and target CRS.
- **propertiesOperationMethod** -- The *General properties* of the *OperationMethod*. At minimum the name should be defined.
- **parameters** -- Vector of parameters of the operation method.

- **values** -- Vector of ParameterValueNNPtr. Constraint: values.size() == parameters.size()
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *PointMotionOperation*.

class **ConcatenatedOperation** : public osgeo::proj::*operation*::*CoordinateOperation*

*#include <coordinateoperation.hpp>* An ordered sequence of two or more single coordinate operations (*SingleOperation*).

The sequence of coordinate operations is constrained by the requirement that the source coordinate reference system of step n+1 shall be the same as the target coordinate reference system of step n.

**Remark**

Implements *ConcatenatedOperation* from *ISO 19111:2019*

### Public Functions

const std::vector<*CoordinateOperationNNPtr*> &**operations**() const

Return the operation steps of the concatenated operation.

**Returns**

the operation steps.

virtual *CoordinateOperationNNPtr* **inverse**() const override

Return the inverse of the coordinate operation.

**Throws**

*util*::*UnsupportedOperationException* -- if inverse is not available

virtual std::set<*GridDescription*> **gridsNeeded**(const *io*::*DatabaseContextPtr* &databaseContext, bool considerKnownGridsAsAvailable) const override

Return grids needed by an operation.

### Public Static Functions

static *ConcatenatedOperationNNPtr* **create**(const *util*::*PropertyMap* &properties, const std::vector<*CoordinateOperationNNPtr*> &operationsIn, const std::vector<*metadata*::*PositionalAccuracyNNPtr*> &accuracies)

Instantiate a *ConcatenatedOperation*.

**Parameters**

- **properties** -- See *General properties*. At minimum the name should be defined.
- **operationsIn** -- Vector of the *CoordinateOperation* steps.
- **accuracies** -- Vector of positional accuracy (might be empty).

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

**Returns**

new *Transformation*.

static *CoordinateOperationNNPtr* **createComputeMetadata**(const std::vector<*CoordinateOperationNNPtr*> &operationsIn, bool checkExtent)

Instantiate a *ConcatenatedOperation*, or return a single coordinate operation.

This computes its accuracy from the sum of its member operations, its extent

**Parameters**

- **operationsIn** -- Vector of the *CoordinateOperation* steps.
- **checkExtent** -- Whether we should check the non-emptiness of the intersection of the extents of the operations

**Throws**

*InvalidOperation* -- if the object cannot be constructed.

class **CoordinateOperationContext**

*#include <coordinateoperation.hpp>* Context in which a coordinate operation is to be used.

**Remark**

Implements [*CoordinateOperationFactory* https://sis.apache.org/apidocs/org/apache/sis/referencing/operation/CoordinateOperationContext.html] from Apache SIS

## Public Types

enum class **SourceTargetCRSExtentUse**

Specify how source and target CRS extent should be used to restrict candidate operations (only taken into account if no explicit area of interest is specified.

*Values:*

enumerator **NONE**

Ignore CRS extent

enumerator **BOTH**

Test coordinate operation extent against both CRS extent.

enumerator **INTERSECTION**

Test coordinate operation extent against the intersection of both CRS extent.

enumerator **SMALLEST**

Test coordinate operation against the smallest of both CRS extent.

enum class **SpatialCriterion**

Spatial criterion to restrict candidate operations.

*Values:*

enumerator **STRICT_CONTAINMENT**

The area of validity of transforms should strictly contain the are of interest.

enumerator **PARTIAL_INTERSECTION**

The area of validity of transforms should at least intersect the area of interest.

enum class **GridAvailabilityUse**

Describe how grid availability is used.

*Values:*

enumerator **USE_FOR_SORTING**

Grid availability is only used for sorting results. Operations where some grids are missing will be sorted last.

enumerator **DISCARD_OPERATION_IF_MISSING_GRID**

Completely discard an operation if a required grid is missing.

enumerator **IGNORE_GRID_AVAILABILITY**

Ignore grid availability at all. Results will be presented as if all grids were available.

enumerator **KNOWN_AVAILABLE**

Results will be presented as if grids known to PROJ (that is registered in the grid_alternatives table of its database) were available. Used typically when networking is enabled.

enum class **IntermediateCRSUse**

Describe if and how intermediate CRS should be used

*Values:*

enumerator **ALWAYS**

Always search for intermediate CRS.

enumerator **IF_NO_DIRECT_TRANSFORMATION**

Only attempt looking for intermediate CRS if there is no direct transformation available.

enumerator **NEVER**

### Public Functions

const *io*::*AuthorityFactoryPtr* &**getAuthorityFactory**() const

Return the authority factory, or null.

const *metadata*::*ExtentPtr* &**getAreaOfInterest**() const

Return the desired area of interest, or null.

void **setAreaOfInterest**(const *metadata*::*ExtentPtr* &extent)

Set the desired area of interest, or null.

double **getDesiredAccuracy**() const

Return the desired accuracy (in metre), or 0.

void **setDesiredAccuracy**(double accuracy)

Set the desired accuracy (in metre), or 0.

void **setAllowBallparkTransformations**(bool allow)

Set whether ballpark transformations are allowed.

bool **getAllowBallparkTransformations**() const

Return whether ballpark transformations are allowed.

void **setSourceAndTargetCRSExtentUse**(*SourceTargetCRSExtentUse* use)

Set how source and target CRS extent should be used when considering if a transformation can be used (only takes effect if no area of interest is explicitly defined).

The default is *CoordinateOperationContext::SourceTargetCRSExtentUse::SMALLEST*.

*SourceTargetCRSExtentUse* **getSourceAndTargetCRSExtentUse**() const

Return how source and target CRS extent should be used when considering if a transformation can be used (only takes effect if no area of interest is explicitly defined).

The default is *CoordinateOperationContext::SourceTargetCRSExtentUse::SMALLEST*.

void **setSpatialCriterion**(*SpatialCriterion* criterion)

Set the spatial criterion to use when comparing the area of validity of coordinate operations with the area of interest / area of validity of source and target CRS.

The default is STRICT_CONTAINMENT.

*SpatialCriterion* **getSpatialCriterion**() const

Return the spatial criterion to use when comparing the area of validity of coordinate operations with the area of interest / area of validity of source and target CRS.

The default is STRICT_CONTAINMENT.

void **setUsePROJAlternativeGridNames**(bool usePROJNames)

Set whether PROJ alternative grid names should be substituted to the official authority names.

This only has effect is an authority factory with a non-null database context has been attached to this context.

If set to false, it is still possible to obtain later the substitution by using *io::PROJStringFormatter::create()* with a non-null database context.

The default is true.

bool **getUsePROJAlternativeGridNames**() const

Return whether PROJ alternative grid names should be substituted to the official authority names.

The default is true.

void **setDiscardSuperseded**(bool discard)

Set whether transformations that are superseded (but not deprecated) should be discarded.

The default is true.

bool **getDiscardSuperseded**() const

Return whether transformations that are superseded (but not deprecated) should be discarded.

The default is true.

void **setGridAvailabilityUse**(*GridAvailabilityUse* use)

Set how grid availability is used.

The default is USE_FOR_SORTING.

*GridAvailabilityUse* **getGridAvailabilityUse**() const

Return how grid availability is used.

The default is USE_FOR_SORTING.

void **setAllowUseIntermediateCRS**(*IntermediateCRSUse* use)

Set whether an intermediate pivot CRS can be used for researching coordinate operations between a source and target CRS.

Concretely if in the database there is an operation from A to C (or C to A), and another one from C to B (or B to C), but no direct operation between A and B, setting this parameter to ALWAYS/IF_NO_DIRECT_TRANSFORMATION, allow chaining both operations.

The current implementation is limited to researching one intermediate step.

By default, with the IF_NO_DIRECT_TRANSFORMATION strategy, all potential C candidates will be used if there is no direct transformation.

*IntermediateCRSUse* **getAllowUseIntermediateCRS**() const

Return whether an intermediate pivot CRS can be used for researching coordinate operations between a source and target CRS.

Concretely if in the database there is an operation from A to C (or C to A), and another one from C to B (or B to C), but no direct operation between A and B, setting this parameter to ALWAYS/IF_NO_DIRECT_TRANSFORMATION, allow chaining both operations.

The default is IF_NO_DIRECT_TRANSFORMATION.

void **setIntermediateCRS**(const std::vector<std::pair<std::string, std::string>> &intermediateCRSAuthCodes)

Restrict the potential pivot CRSs that can be used when trying to build a coordinate operation between two CRS that have no direct operation.

**Parameters**

**intermediateCRSAuthCodes** -- a vector of (auth_name, code) that can be used as potential pivot RS

const std::vector<std::pair<std::string, std::string>> &**getIntermediateCRS**() const

Return the potential pivot CRSs that can be used when trying to build a coordinate operation between two CRS that have no direct operation.

void **setSourceCoordinateEpoch**(const *util*::*optional*<*common*::*DataEpoch*> &epoch)

Set the source coordinate epoch.

const *util*::*optional*<*common*::*DataEpoch*> &**getSourceCoordinateEpoch**() const

Return the source coordinate epoch.

void **setTargetCoordinateEpoch**(const *util*::*optional*<*common*::*DataEpoch*> &epoch)

Set the target coordinate epoch.

const *util*::*optional*<*common*::*DataEpoch*> &**getTargetCoordinateEpoch**() const

Return the target coordinate epoch.

*CoordinateOperationContextNNPtr* **clone**() const

Clone a coordinate operation context.

**Since**
9.2

**Returns**
a new context.

### Public Static Functions

static *CoordinateOperationContextNNPtr* **create**(const *io*::*AuthorityFactoryPtr* &authorityFactory, const *metadata*::*ExtentPtr* &extent, double accuracy)

Creates a context for a coordinate operation.

If a non null authorityFactory is provided, the resulting context should not be used simultaneously by more than one thread.

If authorityFactory->getAuthority() is the empty string, then coordinate operations from any authority will be searched, with the restrictions set in the authority_to_authority_preference database table. If authorityFactory->getAuthority() is set to "any", then coordinate operations from any authority will be searched If authorityFactory->getAuthority() is a non-empty string different of "any", then coordinate operations will be searched only in that authority namespace.

**Parameters**
- **authorityFactory** -- Authority factory, or null if no database lookup is allowed. Use io::authorityFactory::create(context, std::string()) to allow all authorities to be used.
- **extent** -- Area of interest, or null if none is known.
- **accuracy** -- Maximum allowed accuracy in metre, as specified in or 0 to get best accuracy.

**Returns**
a new context.

class **CoordinateOperationFactory**

*#include <coordinateoperation.hpp>* Creates coordinate operations. This factory is capable to find coordinate transformations or conversions between two coordinate reference systems.

**Remark**

Implements (partially) *CoordinateOperationFactory* from *GeoAPI*

### Public Functions

*CoordinateOperationPtr* **createOperation**(const *crs*::*CRSNNPtr* &sourceCRS, const *crs*::*CRSNNPtr* &targetCRS) const

Find a *CoordinateOperation* from sourceCRS to targetCRS.

This is a helper of *createOperations()*, using a coordinate operation context with no authority factory (so no catalog searching is done), no desired accuracy and no area of interest. This returns the first operation of the result set of *createOperations()*, or null if none found.

**Parameters**
- **sourceCRS** -- source CRS.
- **targetCRS** -- source CRS.

**Returns**
a *CoordinateOperation* or nullptr.

std::vector<*CoordinateOperationNNPtr*> **createOperations**(const *crs*::*CRSNNPtr* &sourceCRS, const *crs*::*CRSNNPtr* &targetCRS, const *CoordinateOperationContextNNPtr* &context) const

Find a list of *CoordinateOperation* from sourceCRS to targetCRS.

The operations are sorted with the most relevant ones first: by descending area (intersection of the transformation area with the area of interest, or intersection of the transformation with the area of use of the CRS), and by increasing accuracy. Operations with unknown accuracy are sorted last, whatever their area.

When one of the source or target CRS has a vertical component but not the other one, the one that has no vertical component is automatically promoted to a 3D version, where its vertical axis is the ellipsoidal height in metres, using the ellipsoid of the base geodetic CRS.

**Parameters**
- **sourceCRS** -- source CRS.
- **targetCRS** -- target CRS.
- **context** -- Search context.

**Returns**
a list

std::vector<*CoordinateOperationNNPtr*> **createOperations**(const coordinates::CoordinateMetadataNNPtr &sourceCoordinateMetadata, const *crs*::*CRSNNPtr* &targetCRS, const *CoordinateOperationContextNNPtr* &context) const

Find a list of *CoordinateOperation* from a source coordinate metadata to targetCRS.

**Since**
9.2

**Parameters**
- **sourceCoordinateMetadata** -- source CoordinateMetadata.
- **targetCRS** -- target CRS.
- **context** -- Search context.

**Returns**
a list

std::vector<*CoordinateOperationNNPtr*> **createOperations**(const *crs*::*CRSNNPtr* &sourceCRS, const coordinates::CoordinateMetadataNNPtr &targetCoordinateMetadata, const *CoordinateOperationContextNNPtr* &context) const

Find a list of *CoordinateOperation* from a source CRS to a target coordinate metadata.

**Since**
9.2

**Parameters**
- **sourceCRS** -- source CRS.

- **targetCoordinateMetadata** -- target CoordinateMetadata.
- **context** -- Search context.

**Returns**
a list

std::vector<*CoordinateOperationNNPtr*> **createOperations**(const coordinates::CoordinateMetadataNNPtr &sourceCoordinateMetadata, const coordinates::CoordinateMetadataNNPtr &targetCoordinateMetadata, const *CoordinateOperationContextNNPtr* &context) const

Find a list of *CoordinateOperation* from a source coordinate metadata to a target coordinate metadata.

Both source_crs and target_crs can be a CoordinateMetadata with an associated coordinate epoch, to perform changes of coordinate epochs. Note however than this is in practice limited to use of velocity grids inside the same dynamic CRS.

**Since**
9.4

**Parameters**
- **sourceCoordinateMetadata** -- source CoordinateMetadata.
- **targetCoordinateMetadata** -- target CoordinateMetadata.
- **context** -- Search context.

**Returns**
a list

#### Public Static Functions

static *CoordinateOperationFactoryNNPtr* **create**()

Instantiate a *CoordinateOperationFactory*.

### 10.5.4.9 io namespace

namespace **io**

I/O classes.

*osgeo.proj.io* namespace.

#### Typedefs

using **DatabaseContextPtr** = std::shared_ptr<*DatabaseContext*>

Shared pointer of *DatabaseContext*.

using **DatabaseContextNNPtr** = *util*::nn<*DatabaseContextPtr*>

Non-null shared pointer of *DatabaseContext*.

using **WKTNodePtr** = std::unique_ptr<*WKTNode*>

Unique pointer of *WKTNode*.

using **WKTNodeNNPtr** = *util*::nn<*WKTNodePtr*>

Non-null unique pointer of *WKTNode*.

using **WKTFormatterPtr** = std::unique_ptr<*WKTFormatter*>

*WKTFormatter* unique pointer.

using **WKTFormatterNNPtr** = *util*::nn<*WKTFormatterPtr*>

Non-null *WKTFormatter* unique pointer.

using **PROJStringFormatterPtr** = std::unique_ptr<*PROJStringFormatter*>

*PROJStringFormatter* unique pointer.

using **PROJStringFormatterNNPtr** = *util*::nn<*PROJStringFormatterPtr*>

Non-null *PROJStringFormatter* unique pointer.

using **JSONFormatterPtr** = std::unique_ptr<*JSONFormatter*>

*JSONFormatter* unique pointer.

using **JSONFormatterNNPtr** = *util*::nn<*JSONFormatterPtr*>

Non-null *JSONFormatter* unique pointer.

using **IPROJStringExportablePtr** = std::shared_ptr<*IPROJStringExportable*>

Shared pointer of *IPROJStringExportable*.

using **IPROJStringExportableNNPtr** = *util*::nn<*IPROJStringExportablePtr*>

Non-null shared pointer of *IPROJStringExportable*.

using **AuthorityFactoryPtr** = std::shared_ptr<*AuthorityFactory*>

Shared pointer of *AuthorityFactory*.

using **AuthorityFactoryNNPtr** = *util*::nn<*AuthorityFactoryPtr*>

Non-null shared pointer of *AuthorityFactory*.

### Functions

static *crs*::*GeodeticCRSNNPtr* **cloneWithProps**(const *crs*::*GeodeticCRSNNPtr* &geodCRS, const *util*::*PropertyMap* &props)

static double **asDouble**(const std::string &val)

static void **ThrowNotEnoughChildren**(const std::string &nodeName)

static void **ThrowNotRequiredNumberOfChildren**(const std::string &nodeName)

static void **ThrowMissing**(const std::string &nodeName)

static void **ThrowNotExpectedCSType**(const std::string &expectedCSType)

static *ParsingException* **buildRethrow**(const char *funcName, const std::exception &e)

BaseObjectNNPtr **createFromUserInput**(const std::string &text, const *DatabaseContextPtr* &dbContext, bool usePROJ4InitRules)

Instantiate a sub-class of BaseObject from a user specified text.

The text can be a:

- WKT string
- PROJ string
- database code, prefixed by its authority. e.g. "EPSG:4326"
- OGC URN. e.g. "urn:ogc:def:crs:EPSG::4326", "urn:ogc:def:coordinateOperation:EPSG::1671", "urn:ogc:def:ellipsoid:EPSG::7001" or "urn:ogc:def:datum:EPSG::6326"
- OGC URN combining references for compound coordinate reference systems e.g. "urn:ogc:def:crs,crs:EPSG::2393,crs:EPSG::5717" We also accept a custom abbreviated syntax EPSG:2393+5717 or ESRI:103668+EPSG:5703
- OGC URN combining references for references for projected or derived CRSs e.g. for Projected 3D CRS "UTM zone 31N / WGS 84 (3D)" "urn:ogc:def:crs,crs:EPSG::4979,cs:PROJ::ENh,coordinateOperation:EPSG::16031"
- Extension of OGC URN for CoordinateMetadata. e.g. "urn:ogc:def:coordinateMetadata:NRCAN::NAD83_CSRS_1997
- OGC URN combining references for concatenated operations e.g. "urn:ogc:def:coordinateOperation,coordinateOperation:EPSG::3895,coordinateOperation:EPSG::1618"
- OGC URL for a single CRS. e.g. "http://www.opengis.net/def/crs/EPSG/0/4326"
- OGC URL for a compound CRS. e.g "http://www.opengis.net/def/crs-compound?1=http://www.opengis.net/def/crs/EPSG/0/4326&2=http://www.opengis.net/def/crs/EPSG/0/3855"
- an Object name. e.g "WGS 84", "WGS 84 / UTM zone 31N". In that case as uniqueness is not guaranteed, the function may apply heuristics to determine the appropriate best match.
- a CRS name and a coordinate epoch, separated with '@'. For example "ITRF2014@2025.0". (added in PROJ 9.2)
- a compound CRS made from two object names separated with " + ". e.g. "WGS 84 + EGM96 height"
- PROJJSON string

**Parameters**

- **text** -- One of the above mentioned text format
- **dbContext** -- Database context, or nullptr (in which case database lookups will not work)
- **usePROJ4InitRules** -- When set to true, init=epsg:XXXX syntax will be allowed and will be interpreted according to PROJ.4 and PROJ.5 rules, that is geodeticCRS will have longitude, latitude order and will expect/output coordinates in radians. ProjectedCRS will have easting, northing axis order (except the ones with Transverse Mercator South Orientated projection). In that mode, the epsg:XXXX syntax will be also interpreted the same way.

**Throws**

*ParsingException* -- if the string cannot be parsed.

BaseObjectNNPtr **createFromUserInput**(const std::string &text, PJ_CONTEXT *ctx)

Instantiate a sub-class of BaseObject from a user specified text.

The text can be a:

- WKT string
- PROJ string
- database code, prefixed by its authority. e.g. "EPSG:4326"
- OGC URN. e.g. "urn:ogc:def:crs:EPSG::4326", "urn:ogc:def:coordinateOperation:EPSG::1671", "urn:ogc:def:ellipsoid:EPSG::7001" or "urn:ogc:def:datum:EPSG::6326"
- OGC URN combining references for compound coordinate reference systems e.g. "urn:ogc:def:crs,crs:EPSG::2393,crs:EPSG::5717" We also accept a custom abbreviated syntax EPSG:2393+5717
- OGC URN combining references for references for projected or derived CRSs e.g. for Projected 3D CRS "UTM zone 31N / WGS 84 (3D)" "urn:ogc:def:crs,crs:EPSG::4979,cs:PROJ::ENh,coordinateOperation:EPSG::16031"
- Extension of OGC URN for CoordinateMetadata. e.g. "urn:ogc:def:coordinateMetadata:NRCAN::NAD83_CSRS_1997
- OGC URN combining references for concatenated operations e.g. "urn:ogc:def:coordinateOperation,coordinateOperation:EPSG::3895,coordinateOperation:EPSG::1618"
- an Object name. e.g "WGS 84", "WGS 84 / UTM zone 31N". In that case as uniqueness is not guaranteed, the function may apply heuristics to determine the appropriate best match.
- a compound CRS made from two object names separated with " + ". e.g. "WGS 84 + EGM96 height"
- PROJJSON string

**Parameters**

- **text** -- One of the above mentioned text format
- **ctx** -- PROJ context

**Throws**

*ParsingException* -- if the string cannot be parsed.

class **WKTFormatter**

*#include <io.hpp>* Formatter to WKT strings.

An instance of this class can only be used by a single thread at a time.

### Public Types

enum class **Convention_**

WKT variant.

*Values:*

enumerator **WKT2**

Full WKT2 string, conforming to ISO 19162:2015(E) / OGC 12-063r5 (WKT2:2015) with all possible nodes and new keyword names.

enumerator **_WKT2_2015**

enumerator **WKT2_SIMPLIFIED**

Same as WKT2 with the following exceptions:
- UNIT keyword used.
- ID node only on top element.
- No ORDER element in AXIS element.
- PRIMEM node omitted if it is Greenwich.
- ELLIPSOID.UNIT node omitted if it is UnitOfMeasure::METRE.
- PARAMETER.UNIT / PRIMEM.UNIT omitted if same as AXIS.
- AXIS.UNIT omitted and replaced by a common GEODCRS.UNIT if they are all the same on all axis.

enumerator **_WKT2_2015_SIMPLIFIED**

enumerator **_WKT2_2019**

Full WKT2 string, conforming to ISO 19162:2019 / OGC 18-010, with (WKT2:2019) all possible nodes and new keyword names. Non-normative list of differences:
- _WKT2_2019 uses GEOGCRS / BASEGEOGCRS keywords for GeographicCRS.

enumerator **_WKT2_2018**

Deprecated alias for _WKT2_2019

enumerator **_WKT2_2019_SIMPLIFIED**

_WKT2_2019 with the simplification rule of WKT2_SIMPLIFIED

enumerator **_WKT2_2018_SIMPLIFIED**

Deprecated alias for _WKT2_2019_SIMPLIFIED

enumerator **_WKT1_GDAL**

WKT1 as traditionally output by GDAL, deriving from OGC 01-009. A notable departure from _WKT1_GDAL with respect to OGC 01-009 is that in _WKT1_GDAL, the unit of the PRIMEM value is always degrees.

enumerator **_WKT1_ESRI**

WKT1 as traditionally output by ESRI software, deriving from OGC 99-049.

enum class **OutputAxisRule**

Rule for output AXIS nodes

*Values:*

enumerator **YES**

Always include AXIS nodes

enumerator **NO**

Never include AXIS nodes

enumerator **_WKT1_GDAL_EPSG_STYLE**

Includes them only on PROJCS node if it uses Easting/Northing ordering. Typically used for _WKT1_GDAL

### Public Functions

*WKTFormatter* &**setMultiLine**(bool multiLine) noexcept

Whether to use multi line output or not.

*WKTFormatter* &**setIndentationWidth**(int width) noexcept

Set number of spaces for each indentation level (defaults to 4).

*WKTFormatter* &**setOutputAxis**(*OutputAxisRule* outputAxis) noexcept

Set whether AXIS nodes should be output.

*WKTFormatter* &**setStrict**(bool strict) noexcept

Set whether the formatter should operate on strict more or not.

The default is strict mode, in which case a *FormattingException* can be thrown. In non-strict mode, a Geographic 3D CRS can be for example exported as _WKT1_GDAL with 3 axes, whereas this is normally not allowed.

bool **isStrict**() const noexcept

Returns whether the formatter is in strict mode.

*WKTFormatter* &**setAllowEllipsoidalHeightAsVerticalCRS**(bool allow) noexcept

Set whether the formatter should export, in WKT1, a Geographic or Projected 3D CRS as a compound CRS whose vertical part represents an ellipsoidal height.

bool **isAllowedEllipsoidalHeightAsVerticalCRS**() const noexcept

Return whether the formatter should export, in WKT1, a Geographic or Projected 3D CRS as a compound CRS whose vertical part represents an ellipsoidal height.

*WKTFormatter* &**setAllowLINUNITNode**(bool allow) noexcept

Set whether the formatter should export, in _WKT1_ESRI, a Geographic 3D CRS with the relatively new (ArcGIS Pro >= 2.7) LINUNIT node. Defaults to true.

**Since**
PROJ 9.1

bool **isAllowedLINUNITNode**() const noexcept

Return whether the formatter should export, in _WKT1_ESRI, a Geographic 3D CRS with the relatively new (ArcGIS Pro >= 2.7) LINUNIT node. Defaults to true.

**Since**
PROJ 9.1

const std::string &**toString**() const

Returns the WKT string from the formatter.

### Public Static Functions

static *WKTFormatterNNPtr* **create**(*Convention_* convention = *Convention_*::*WKT2*,
*DatabaseContextPtr* dbContext = nullptr)

Constructs a new formatter.

A formatter can be used only once (its internal state is mutated)

Its default behavior can be adjusted with the different setters.

**Parameters**

- **convention** -- WKT flavor. Defaults to *Convention_::WKT2*
- **dbContext** -- Database context, to allow queries in it if needed. This is used for example for _WKT1_ESRI output to do name substitutions.

**Returns**

new formatter.

static *WKTFormatterNNPtr* **create**(const *WKTFormatterNNPtr* &other)

Constructs a new formatter from another one.

A formatter can be used only once (its internal state is mutated)

Its default behavior can be adjusted with the different setters.

**Parameters**

**other** -- source formatter.

**Returns**

new formatter.

class **PROJStringFormatter**

*#include <io.hpp>* Formatter to PROJ strings.

An instance of this class can only be used by a single thread at a time.

### Public Types

enum class **Convention**

PROJ variant.

*Values:*

enumerator **PROJ_5**

PROJ v5 (or later versions) string.

enumerator **PROJ_4**

PROJ v4 string as output by GDAL exportToProj4()

### Public Functions

*PROJStringFormatter* &**setMultiLine**(bool multiLine) noexcept

Whether to use multi line output or not.

*PROJStringFormatter* &**setIndentationWidth**(int width) noexcept

Set number of spaces for each indentation level (defaults to 2).

*PROJStringFormatter* &**setMaxLineLength**(int maxLineLength) noexcept

Set the maximum size of a line (when multiline output is enable). Can be set to 0 for unlimited length.

void **setUseApproxTMerc**(bool flag)

Set whether approximate Transverse Mercator or UTM should be used.

const std::string &**toString**() const

Returns the PROJ string.

### Public Static Functions

static *PROJStringFormatterNNPtr* **create**(*Convention* conventionIn = *Convention*::*PROJ_5*, *DatabaseContextPtr* dbContext = nullptr)

Constructs a new formatter.

A formatter can be used only once (its internal state is mutated)

Its default behavior can be adjusted with the different setters.

**Parameters**

- **conventionIn** -- PROJ string flavor. Defaults to *Convention::PROJ_5*
- **dbContext** -- Database context (can help to find alternative grid names). May be nullptr

**Returns**

new formatter.

class **JSONFormatter**

*#include <io.hpp>* Formatter to JSON strings.

An instance of this class can only be used by a single thread at a time.

### Public Functions

*JSONFormatter* &**setMultiLine**(bool multiLine) noexcept

Whether to use multi line output or not.

*JSONFormatter* &**setIndentationWidth**(int width) noexcept

Set number of spaces for each indentation level (defaults to 4).

*JSONFormatter* &**setSchema**(const std::string &schema) noexcept

Set the value of the "$schema" key in the top level object.

If set to empty string, it will not be written.

const std::string &**toString**() const

Return the serialized JSON.

### Public Static Functions

static *JSONFormatterNNPtr* **create**(*DatabaseContextPtr* dbContext = nullptr)

Constructs a new formatter.

A formatter can be used only once (its internal state is mutated)

**Returns**

new formatter.

class **IJSONExportable**

*#include <io.hpp>* Interface for an object that can be exported to JSON.

Subclassed by osgeo::proj::coordinates::CoordinateMetadata, *osgeo::proj::crs::CRS*, *osgeo::proj::cs::CoordinateSystem*, *osgeo::proj::cs::CoordinateSystemAxis*, *osgeo::proj::cs::Meridian*, *osgeo::proj::datum::Datum*, *osgeo::proj::datum::DatumEnsemble*, *osgeo::proj::datum::Ellipsoid*, *osgeo::proj::datum::PrimeMeridian*, *osgeo::proj::metadata::Identifier*, *osgeo::proj::operation::CoordinateOperation*, *osgeo::proj::operation::GeneralParameterValue*, *osgeo::proj::operation::OperationMethod*

### Public Functions

std::string **exportToJSON**(*JSONFormatter* *formatter) const

Builds a JSON representation. May throw a *FormattingException*

class **FormattingException** : public osgeo::proj::*util*::*Exception*

*#include <io.hpp>* Exception possibly thrown by *IWKTExportable::exportToWKT()* or *IPROJStringExportable::exportToPROJString()*.

class **ParsingException** : public osgeo::proj::*util*::*Exception*

*#include <io.hpp>* Exception possibly thrown by *WKTNode::createFrom()* or *WKTParser::createFromWKT()*.

class **IWKTExportable**

*#include <io.hpp>* Interface for an object that can be exported to WKT.

Subclassed by *osgeo::proj::common::IdentifiedObject*, osgeo::proj::coordinates::CoordinateMetadata, *osgeo::proj::metadata::Identifier*, *osgeo::proj::operation::GeneralParameterValue*, *osgeo::proj::operation::ParameterValue*

### Public Functions

std::string **exportToWKT**(*WKTFormatter* *formatter) const

Builds a WKT representation. May throw a *FormattingException*

class **IPROJStringExportable**

*#include <io.hpp>* Interface for an object that can be exported to a PROJ string.

Subclassed by *osgeo::proj::crs::BoundCRS*, *osgeo::proj::crs::CompoundCRS*, *osgeo::proj::crs::GeodeticCRS*, *osgeo::proj::crs::ProjectedCRS*, *osgeo::proj::crs::VerticalCRS*, *osgeo::proj::datum::Ellipsoid*, *osgeo::proj::datum::PrimeMeridian*, *osgeo::proj::operation::CoordinateOperation*

### Public Functions

std::string **exportToPROJString**(*PROJStringFormatter* *formatter) const

Builds a PROJ string representation.

- For *PROJStringFormatter::Convention::PROJ_5* (the default),
  - For a *crs::CRS*, returns the same as *PROJStringFormatter::Convention::PROJ_4*. It should be noted that the export of a CRS as a PROJ string may cause loss of many important aspects of a CRS definition. Consequently it is discouraged to use it for interoperability in newer projects. The choice of a WKT representation will be a better option.
  - For *operation::CoordinateOperation*, returns a PROJ pipeline.

- For *PROJStringFormatter::Convention::PROJ_4*, format a string compatible with the OGRSpatialReference::exportToProj4() of GDAL <=2.3. It is only compatible of a few CRS objects. The PROJ string will also contain a +type=crs parameter to disambiguate the nature of the string from a CoordinateOperation.
  - For a *crs::GeographicCRS*, returns a proj=longlat string, with ellipsoid / datum / prime meridian information, ignoring axis order and unit information.
  - For a geocentric *crs::GeodeticCRS*, returns the transformation from geographic coordinates into geocentric coordinates.

- For a *crs::ProjectedCRS*, returns the projection method, ignoring axis order.
- For a *crs::BoundCRS*, returns the PROJ string of its source/base CRS, amended with towgs84 / nadgrids parameter when the deriving conversion can be expressed in that way.

**Parameters**
**formatter** -- PROJ string formatter.
**Throws**
*FormattingException* -- if cannot be exported as a PROJ string
**Returns**
a PROJ string.

class **WKTNode**

*#include <io.hpp>* Node in the tree-splitted WKT representation.

### Public Functions

explicit **WKTNode**(const std::string &valueIn)

Instantiate a *WKTNode*.
**Parameters**
**valueIn** -- the name of the node.

const std::string &**value**() const

Return the value of a node.

const std::vector<*WKTNodeNNPtr*> &**children**() const

Return the children of a node.

void **addChild**(*WKTNodeNNPtr* &&child)

Adds a child to the current node.
**Parameters**
**child** -- child to add. This should not be a parent of this node.

const *WKTNodePtr* &**lookForChild**(const std::string &childName, int occurrence = 0) const noexcept

Return the (occurrence-1)th sub-node of name childName.
**Parameters**
- **childName** -- name of the child.
- **occurrence** -- occurrence index (starting at 0)

**Returns**
the child, or nullptr.

int **countChildrenOfName**(const std::string &childName) const noexcept

Return the count of children of given name.
**Parameters**
**childName** -- name of the children to look for.
**Returns**
count

std::string **toString**() const

Return a WKT representation of the tree structure.

### Public Static Functions

static *WKTNodeNNPtr* **createFrom**(const std::string &wkt, size_t indexStart = 0)

Instantiate a *WKTNode* hierarchy from a WKT string.

**Parameters**

- **wkt** -- the WKT string to parse.
- **indexStart** -- the start index in the wkt string.

**Throws**

*ParsingException* -- if the string cannot be parsed.

class **WKTParser**

*#include <io.hpp>* Parse a WKT string into the appropriate subclass of *util::BaseObject*.

### Public Types

enum class **WKTGuessedDialect**

Guessed WKT "dialect"

*Values:*

enumerator **WKT2_2019**

*WKT2:2019*

enumerator **WKT2_2018**

Deprecated alias for WKT2_2019

enumerator **WKT2_2015**

*WKT2:2015*

enumerator **WKT1_GDAL**

*WKT1 specification*

enumerator **WKT1_ESRI**

ESRI variant of WKT1

enumerator **NOT_WKT**

Not WKT / unrecognized

### Public Functions

*WKTParser* &**attachDatabaseContext**(const *DatabaseContextPtr* &dbContext)

Attach a database context, to allow queries in it if needed.

*WKTParser* &**setStrict**(bool strict)

Set whether parsing should be done in strict mode.

std::list<std::string> **warningList**() const

Return the list of warnings found during parsing.

> **Note**
>
> The list might be non-empty only is setStrict(false) has been called.

std::list<std::string> **grammarErrorList**() const

Return the list of grammar errors found during parsing.

Grammar errors are non-compliance issues with respect to the WKT grammar.

**Since**
PROJ 9.5

> **Note**
>
> The list might be non-empty only is setStrict(false) has been called.

*WKTParser* &**setUnsetIdentifiersIfIncompatibleDef**(bool unset)

Set whether object identifiers should be unset when there is a contradiction between the definition from WKT and the one from the database.

At time of writing, this only applies to the base geographic CRS of a projected CRS, when comparing its coordinate system.

*util*::*BaseObjectNNPtr* **createFromWKT**(const std::string &wkt)

Instantiate a sub-class of BaseObject from a WKT string.

By default, validation is strict (to the extent of the checks that are actually implemented. Currently only WKT1 strict grammar is checked), and any issue detected will cause an exception to be thrown, unless setStrict(false) is called priorly.

In non-strict mode, non-fatal issues will be recovered and simply listed in *warningList()*. This does not prevent more severe errors to cause an exception to be thrown.

**Throws**
*ParsingException* -- if the string cannot be parsed.

*WKTGuessedDialect* **guessDialect**(const std::string &wkt) noexcept

Guess the "dialect" of the WKT string.

class **PROJStringParser**

*#include <io.hpp>* Parse a PROJ string into the appropriate subclass of *util::BaseObject*.

### Public Functions

*PROJStringParser* &**attachDatabaseContext**(const *DatabaseContextPtr* &dbContext)

Attach a database context, to allow queries in it if needed.

*PROJStringParser* &**setUsePROJ4InitRules**(bool enable)

Set how init=epsg:XXXX syntax should be interpreted.

**Parameters**
**enable** -- When set to true, init=epsg:XXXX syntax will be allowed and will be interpreted according to PROJ.4 and PROJ.5 rules, that is geodeticCRS will have longitude, latitude order and will expect/output coordinates in radians. ProjectedCRS will have easting, northing axis order (except the ones with Transverse Mercator South Orientated projection).

std::vector<std::string> **warningList**() const

Return the list of warnings found during parsing.

*util*::*BaseObjectNNPtr* **createFromPROJString**(const std::string &projString)

Instantiate a sub-class of BaseObject from a PROJ string.

The projString must contain +type=crs for the object to be detected as a CRS instead of a CoordinateOperation.

**Throws**

*ParsingException* -- if the string cannot be parsed.

class **DatabaseContext**

*#include <io.hpp>* Database context.

A database context should be used only by one thread at a time.

### Public Functions

const std::string &**getPath**() const

Return the path to the database.

const char ***getMetadata**(const char *key) const

Return a metadata item.

Value remains valid while this is alive and to the next call to getMetadata

std::set<std::string> **getAuthorities**() const

Return the list of authorities used in the database.

std::vector<std::string> **getDatabaseStructure**() const

Return the list of SQL commands (CREATE TABLE, CREATE TRIGGER, CREATE VIEW) needed to initialize a new database.

void **startInsertStatementsSession**()

Starts a session for *getInsertStatementsFor()*

Starts a new session for one or several calls to *getInsertStatementsFor()*. An insertion session guarantees that the inserted objects will not create conflicting intermediate objects.

The session must be stopped with *stopInsertStatementsSession()*.

Only one session may be active at a time for a given database context.

**Since**

8.1

**Throws**

*FactoryException* -- in case of error.

std::string **suggestsCodeFor**(const *common*::*IdentifiedObjectNNPtr* &object, const std::string &authName, bool numericCode)

Suggests a database code for the passed object.

Supported type of objects are PrimeMeridian, Ellipsoid, Datum, DatumEnsemble, GeodeticCRS, ProjectedCRS, VerticalCRS, CompoundCRS, BoundCRS, Conversion.

**Since**

8.1

**Parameters**
- **object** -- Object for which to suggest a code.
- **authName** -- Authority name into which the object will be inserted.
- **numericCode** -- Whether the code should be numeric, or derived from the object name.

**Throws**
*FactoryException* -- in case of error.

**Returns**
the suggested code, that is guaranteed to not conflict with an existing one.

std::vector<std::string> **getInsertStatementsFor**(const *common*::*IdentifiedObjectNNPtr* &object, const std::string &authName, const std::string &code, bool numericCode, const std::vector<std::string> &allowedAuthorities = {"EPSG", "PROJ"})

Returns SQL statements needed to insert the passed object into the database.

*startInsertStatementsSession()* must have been called previously.

**Since**
8.1

**Parameters**
- **object** -- The object to insert into the database. Currently only PrimeMeridian, Ellipsoid, Datum, GeodeticCRS, ProjectedCRS, VerticalCRS, CompoundCRS or BoundCRS are supported.
- **authName** -- Authority name into which the object will be inserted.
- **code** -- Code with which the object will be inserted.
- **numericCode** -- Whether intermediate objects that can be created should use numeric codes (true), or may be alphanumeric (false)
- **allowedAuthorities** -- Authorities to which intermediate objects are allowed to refer to. authName will be implicitly added to it. Note that unit, coordinate systems, projection methods and parameters will in any case be allowed to refer to EPSG.

**Throws**
*FactoryException* -- in case of error.

void **stopInsertStatementsSession**()

Stops an insertion session started with *startInsertStatementsSession()*

**Since**
8.1

### Public Static Functions

static *DatabaseContextNNPtr* **create**(const std::string &databasePath = std::string(), const std::vector<std::string> &auxiliaryDatabasePaths = std::vector<std::string>(), PJ_CONTEXT *ctx = nullptr)

Instantiate a database context.

This database context should be used only by one thread at a time.

**Parameters**
- **databasePath** -- Path and filename of the database. Might be empty string for the default rules to locate the default proj.db
- **auxiliaryDatabasePaths** -- Path and filename of auxiliary databases. Might be empty. Starting with PROJ 8.1, if this parameter is an empty array, the PROJ_AUX_DB environment variable will be used, if set. It must contain one or several paths. If several

paths are provided, they must be separated by the colon (:) character on Unix, and on Windows, by the semi-colon (;) character.

- **ctx** -- Context used for file search.

**Throws**

*FactoryException* -- if the database cannot be opened.

class **AuthorityFactory**

*#include <io.hpp>* Builds object from an authority database.

A *AuthorityFactory* should be used only by one thread at a time.

**Remark**

Implements AuthorityFactory from *GeoAPI*

### Public Types

enum class **ObjectType**

Object type.

*Values:*

enumerator **PRIME_MERIDIAN**

Object of type *datum::PrimeMeridian*

enumerator **ELLIPSOID**

Object of type *datum::Ellipsoid*

enumerator **DATUM**

Object of type *datum::Datum* (and derived classes)

enumerator **GEODETIC_REFERENCE_FRAME**

Object of type *datum::GeodeticReferenceFrame* (and derived classes)

enumerator **VERTICAL_REFERENCE_FRAME**

Object of type *datum::VerticalReferenceFrame* (and derived classes)

enumerator **ENGINEERING_DATUM**

Object of type *datum::EngineeringDatum*

enumerator **CRS**

Object of type *crs::CRS* (and derived classes)

enumerator **GEODETIC_CRS**

Object of type *crs::GeodeticCRS* (and derived classes)

enumerator **GEOCENTRIC_CRS**

GEODETIC_CRS of type geocentric

enumerator **GEOGRAPHIC_CRS**

Object of type *crs::GeographicCRS* (and derived classes)

enumerator **GEOGRAPHIC_2D_CRS**

GEOGRAPHIC_CRS of type Geographic 2D

enumerator **GEOGRAPHIC_3D_CRS**

GEOGRAPHIC_CRS of type Geographic 3D

enumerator **PROJECTED_CRS**

Object of type *crs::ProjectedCRS* (and derived classes)

enumerator **VERTICAL_CRS**

Object of type *crs::VerticalCRS* (and derived classes)

enumerator **ENGINEERING_CRS**

Object of type *crs::CompoundCRS* (and derived classes)

enumerator **COMPOUND_CRS**

Object of type *crs::EngineeringCRS*

enumerator **COORDINATE_OPERATION**

Object of type *operation::CoordinateOperation* (and derived classes)

enumerator **CONVERSION**

Object of type *operation::Conversion* (and derived classes)

enumerator **TRANSFORMATION**

Object of type *operation::Transformation* (and derived classes)

enumerator **CONCATENATED_OPERATION**

Object of type *operation::ConcatenatedOperation* (and derived classes)

enumerator **DYNAMIC_GEODETIC_REFERENCE_FRAME**

Object of type *datum::DynamicGeodeticReferenceFrame*

enumerator **DYNAMIC_VERTICAL_REFERENCE_FRAME**

Object of type *datum::DynamicVerticalReferenceFrame*

enumerator **DATUM_ENSEMBLE**

Object of type *datum::DatumEnsemble*

### Public Functions

*util*::*BaseObjectNNPtr* **createObject**(const std::string &code) const

Returns an arbitrary object from a code.

The returned object will typically be an instance of Datum, CoordinateSystem, ReferenceSystem or CoordinateOperation. If the type of the object is know at compile time, it is recommended to invoke the most precise method instead of this one (for example createCoordinateReferenceSystem(code) instead of createObject(code) if the caller know he is asking for a coordinate reference system).

If there are several objects with the same code, a *FactoryException* is thrown.

**Parameters**
**code** -- Object code allocated by authority. (e.g. "4326")

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*common*::*UnitOfMeasureNNPtr* **createUnitOfMeasure**(const std::string &code) const

Returns a *common::UnitOfMeasure* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*metadata*::*ExtentNNPtr* **createExtent**(const std::string &code) const

Returns a *metadata::Extent* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*datum*::*PrimeMeridianNNPtr* **createPrimeMeridian**(const std::string &code) const

Returns a *datum::PrimeMeridian* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

std::string **identifyBodyFromSemiMajorAxis**(double a, double tolerance) const

Identify a celestial body from an approximate radius.

**Parameters**
- **semi_major_axis** -- Approximate semi-major axis.
- **tolerance** -- Relative error allowed.

**Throws**
*FactoryException* -- in case of error.

**Returns**
celestial body name if one single match found.

*datum*::*EllipsoidNNPtr* **createEllipsoid**(const std::string &code) const

Returns a *datum::Ellipsoid* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*datum*::*DatumNNPtr* **createDatum**(const std::string &code) const

Returns a *datum::Datum* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*datum*::*DatumEnsembleNNPtr* **createDatumEnsemble**(const std::string &code, const std::string &type = std::string()) const

Returns a *datum::DatumEnsemble* from the specified code.

**Parameters**
- **code** -- Object code allocated by authority.
- **type** -- "geodetic_datum", "vertical_datum" or empty string if unknown

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*datum*::*GeodeticReferenceFrameNNPtr* **createGeodeticDatum**(const std::string &code) const

Returns a *datum::GeodeticReferenceFrame* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*datum*::*VerticalReferenceFrameNNPtr* **createVerticalDatum**(const std::string &code) const

Returns a *datum::VerticalReferenceFrame* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*datum*::*EngineeringDatumNNPtr* **createEngineeringDatum**(const std::string &code) const

Returns a *datum::EngineeringDatum* from the specified code.

**Since**
9.6

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*cs*::*CoordinateSystemNNPtr* **createCoordinateSystem**(const std::string &code) const

Returns a *cs::CoordinateSystem* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*crs*::*GeodeticCRSNNPtr* **createGeodeticCRS**(const std::string &code) const

Returns a *crs::GeodeticCRS* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*crs*::*GeographicCRSNNPtr* **createGeographicCRS**(const std::string &code) const

Returns a *crs::GeographicCRS* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*crs*::*VerticalCRSNNPtr* **createVerticalCRS**(const std::string &code) const

Returns a *crs::VerticalCRS* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*crs*::*EngineeringCRSNNPtr* **createEngineeringCRS**(const std::string &code) const

Returns a *crs::EngineeringCRS* from the specified code.

**Since**
9.6

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*operation*::*ConversionNNPtr* **createConversion**(const std::string &code) const

Returns a *operation::Conversion* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*crs*::*ProjectedCRSNNPtr* **createProjectedCRS**(const std::string &code) const

Returns a *crs::ProjectedCRS* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*crs*::*CompoundCRSNNPtr* **createCompoundCRS**(const std::string &code) const

Returns a *crs::CompoundCRS* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*crs*::*CRSNNPtr* **createCoordinateReferenceSystem**(const std::string &code) const

Returns a *crs::CRS* from the specified code.

**Parameters**
**code** -- Object code allocated by authority.
**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

coordinates::CoordinateMetadataNNPtr **createCoordinateMetadata**(const std::string &code) const

Returns a coordinates::CoordinateMetadata from the specified code.

**Since**
9.4

**Parameters**
**code** -- Object code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

*operation*::*CoordinateOperationNNPtr* **createCoordinateOperation**(const std::string &code, bool usePROJAlternativeGridNames) const

Returns a *operation::CoordinateOperation* from the specified code.

**Parameters**
- **code** -- Object code allocated by authority.
- **usePROJAlternativeGridNames** -- Whether PROJ alternative grid names should be substituted to the official grid names.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
object.

std::vector<*operation*::*CoordinateOperationNNPtr*> **createFromCoordinateReferenceSystemCodes**(const std::string &sourceCRSCode, const std::string &targetCRSCode) const

Returns a list *operation::CoordinateOperation* between two CRS.

The list is ordered with preferred operations first. No attempt is made at inferring operations that are not explicitly in the database.

Deprecated operations are rejected.

**Parameters**
- **sourceCRSCode** -- Source CRS code allocated by authority.
- **targetCRSCode** -- Source CRS code allocated by authority.

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
list of coordinate operations

std::list<std::string> **getGeoidModels**(const std::string &code) const

Returns a list of geoid models available for that crs.

The list includes the geoid models connected directly with the crs, or via "Height Depth Reversal" or "Change of Vertical Unit" transformations

**Parameters**
**code** -- crs code allocated by authority.

**Throws**
*FactoryException* -- in case of error.

**Returns**
list of geoid model names

const std::string &**getAuthority**()

Returns the authority name associated to this factory.

**Returns**
name.

std::set<std::string> **getAuthorityCodes**(const *ObjectType* &type, bool allowDeprecated = true) const

Returns the set of authority codes of the given object type.

**Parameters**
- **type** -- Object type.
- **allowDeprecated** -- whether we should return deprecated objects as well.

**Throws**
*FactoryException* -- in case of error.

**Returns**
the set of authority codes for spatial reference objects of the given type

std::string **getDescriptionText**(const std::string &code) const

Gets a description of the object corresponding to a code.

> **Note**
>
> In case of several objects of different types with the same code, one of them will be arbitrarily selected. But if a CRS object is found, it will be selected.

**Parameters**
**code** -- Object code allocated by authority. (e.g. "4326")

**Throws**
- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**
description.

std::list<*CRSInfo*> **getCRSInfoList**() const

Return a list of information on CRS objects.

This is functionally equivalent of listing the codes from an authority, instantiating a CRS object for each of them and getting the information from this CRS object, but this implementation has much less overhead.

**Throws**
*FactoryException* -- in case of error.

std::list<*UnitInfo*> **getUnitList**() const

Return the list of units.

**Since**
7.1

**Throws**
*FactoryException* -- in case of error.

std::list<*CelestialBodyInfo*> **getCelestialBodyList**() const

Return the list of celestial bodies.

**Since**
8.1

**Throws**
*FactoryException* -- in case of error.

const *DatabaseContextNNPtr* &**databaseContext**() const

Returns the database context.

std::vector<*operation*::*CoordinateOperationNNPtr*> **createFromCoordinateReferenceSystemCodes**(const std::string &sourceCRSAuthName, const std::string &sourceCRSCode, const std::string &targetCRSAuthName, const std::string &targetCRSCode, bool usePROJAlternativeGridNames, bool discardIfMissingGrid, bool considerKnownGridsAsAvailable, bool discardSuperseded, bool tryReverseOrder = false, bool reportOnly-

Returns a list *operation::CoordinateOperation* between two CRS.

The list is ordered with preferred operations first. No attempt is made at inferring operations that are not explicitly in the database (see *createFromCRSCodesWithIntermediates()* for that), and only source -> target operations are searched (i.e. if target -> source is present, you need to call this method with the arguments reversed, and apply the reverse transformations).

Deprecated operations are rejected.

If *getAuthority()* returns empty, then coordinate operations from all authorities are considered.

**Parameters**

- **sourceCRSAuthName** -- Authority name of sourceCRSCode
- **sourceCRSCode** -- Source CRS code allocated by authority sourceCRSAuthName.
- **targetCRSAuthName** -- Authority name of targetCRSCode
- **targetCRSCode** -- Source CRS code allocated by authority targetCRSAuthName.
- **usePROJAlternativeGridNames** -- Whether PROJ alternative grid names should be substituted to the official grid names.
- **discardIfMissingGrid** -- Whether coordinate operations that reference missing grids should be removed from the result set.
- **considerKnownGridsAsAvailable** -- Whether known grids should be considered as available (typically when network is enabled).
- **discardSuperseded** -- Whether coordinate operations that are superseded (but not deprecated) should be removed from the result set.
- **tryReverseOrder** -- whether to search in the reverse order too (and thus inverse results found that way)
- **reportOnlyIntersectingTransformations** -- if intersectingExtent1 and intersectingExtent2 should be honored in a strict way.
- **intersectingExtent1** -- Optional extent that the resulting operations must intersect.
- **intersectingExtent2** -- Optional extent that the resulting operations must intersect.

**Throws**

- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**

list of coordinate operations

std::vector<*operation*::*CoordinateOperationNNPtr*> **createFromCRSCodesWithIntermediates**(const std::string &sourceCRSAuthName, const std::string &sourceCRSCode, const std::string &targetCRSAuthName, const std::string &targetCRSCode, bool usePROJAlternativeGridNames, bool discardIfMissingGrid, bool considerKnownGridsAsAvailable, bool discardSuperseded, const std::vector<std::pair<std::string,std::string>> &intermediateCRSAuthCodes, const std::string &allowedAuthorities, double spatialExtentPadding, const *util*::*optional*<*metadata*::*Extent*> &restrictToSpatialExtent, const *util*::*optional*<*common*::*DataEpoch*> &sourceCoordinateEpoch, const *util*::*optional*<*common*::*DataEpoch*> &targetCoordinateEpoch)

Returns a list *operation::CoordinateOperation* between two CRS, using intermediate codes.

The list is ordered with preferred operations first.

Deprecated operations are rejected.

The method will take care of considering all potential combinations (i.e. contrary to *createFromCoordinateReferenceSystemCodes()*, you do not need to call it with sourceCRS and targetCRS switched)

If *getAuthority()* returns empty, then coordinate operations from all authorities are considered.

**Parameters**

- **sourceCRSAuthName** -- Authority name of sourceCRSCode
- **sourceCRSCode** -- Source CRS code allocated by authority sourceCRSAuthName.
- **targetCRSAuthName** -- Authority name of targetCRSCode
- **targetCRSCode** -- Source CRS code allocated by authority targetCRSAuthName.
- **usePROJAlternativeGridNames** -- Whether PROJ alternative grid names should be substituted to the official grid names.
- **discardIfMissingGrid** -- Whether coordinate operations that reference missing grids should be removed from the result set.
- **considerKnownGridsAsAvailable** -- Whether known grids should be considered as available (typically when network is enabled).
- **discardSuperseded** -- Whether coordinate operations that are superseded (but not deprecated) should be removed from the result set.
- **intermediateCRSAuthCodes** -- List of (auth_name, code) of CRS that can be used as potential intermediate CRS. If the list is empty, the database will be used to find common CRS in operations involving both the source and target CRS.
- **allowedIntermediateObjectType** -- Restrict the type of the intermediate object considered. Only *ObjectType::CRS* and *ObjectType::GEOGRAPHIC_CRS* supported currently
- **allowedAuthorities** -- One or several authority name allowed for the two coordinate operations that are going to be searched. When this vector is no empty, it overrides the authority of this object. This is useful for example when the coordinate operations to chain belong to two different allowed authorities.
- **intersectingExtent1** -- Optional extent that the resulting operations must intersect.
- **intersectingExtent2** -- Optional extent that the resulting operations must intersect.

**Throws**

- *NoSuchAuthorityCodeException* -- if there is no matching object.
- *FactoryException* -- in case of other errors.

**Returns**

list of coordinate operations

std::string **getOfficialNameFromAlias**(const std::string &aliasedName, const std::string &tableName, const std::string &source, bool tryEquivalentNameSpelling, std::string &outTableName, std::string &outAuthName, std::string &outCode) const

Gets the official name from a possibly alias name.

**Parameters**

- **aliasedName** -- Alias name.
- **tableName** -- Table name/category. Can help in case of ambiguities. Or empty otherwise.
- **source** -- Source of the alias. Can help in case of ambiguities. Or empty otherwise.
- **tryEquivalentNameSpelling** -- whether the comparison of aliasedName with the alt_name column of the alias_name table should be done with using *metadata::Identifier::isEquivalentName()* rather than strict string comparison;
- **outTableName** -- Table name in which the official name has been found.

- **outAuthName** -- Authority name of the official name that has been found.
- **outCode** -- Code of the official name that has been found.

**Throws**

*FactoryException* -- in case of error.

**Returns**

official name (or empty if not found).

std::list<*common*::*IdentifiedObjectNNPtr*> **createObjectsFromName**(const std::string &name, const std::vector<*ObjectType*> &allowedObjectTypes = std::vector<*ObjectType*>(), bool approximateMatch = true, size_t limitResultCount = 0) const

Return a list of objects, identified by their name.

**Parameters**

- **searchedName** -- Searched name. Must be at least 2 character long.
- **allowedObjectTypes** -- List of object types into which to search. If empty, all object types will be searched.
- **approximateMatch** -- Whether approximate name identification is allowed.
- **limitResultCount** -- Maximum number of results to return. Or 0 for unlimited.

**Throws**

*FactoryException* -- in case of error.

**Returns**

list of matched objects.

std::list<std::pair<std::string, std::string>> **listAreaOfUseFromName**(const std::string &name, bool approximateMatch) const

Return a list of area of use from their name.

**Parameters**

- **name** -- Searched name.
- **approximateMatch** -- Whether approximate name identification is allowed.

**Throws**

*FactoryException* -- in case of error.

**Returns**

list of (auth_name, code) of matched objects.

### Public Static Functions

static *AuthorityFactoryNNPtr* **create**(const *DatabaseContextNNPtr* &context, const std::string &authorityName)

Instantiate a *AuthorityFactory*.

The authority name might be set to the empty string in the particular case where createFromCoordinateReferenceSystemCodes(const std::string&,const std::string&,const std::string&,const std::string&) const is called.

**Parameters**

- **context** -- Context.
- **authorityName** -- Authority name.

**Returns**

new *AuthorityFactory*.

struct **CelestialBodyInfo**

*#include <io.hpp>* Celestial Body information

### Public Members

std::string **authName**

Authority name

std::string **name**

Name

struct **CRSInfo**

*#include <io.hpp>* CRS information

### Public Members

std::string **authName**

Authority name

std::string **code**

Code

std::string **name**

Name

*ObjectType* **type**

Type

bool **deprecated**

Whether the object is deprecated

bool **bbox_valid**

Whereas the west_lon_degree, south_lat_degree, east_lon_degree and north_lat_degree fields are valid.

double **west_lon_degree**

Western-most longitude of the area of use, in degrees.

double **south_lat_degree**

Southern-most latitude of the area of use, in degrees.

double **east_lon_degree**

Eastern-most longitude of the area of use, in degrees.

double **north_lat_degree**

Northern-most latitude of the area of use, in degrees.

std::string **areaName**

Name of the area of use.

std::string **projectionMethodName**

Name of the projection method for a projected CRS. Might be empty even for projected CRS in some cases.

std::string **celestialBodyName**

Name of the celestial body of the CRS (e.g. "Earth")

struct **UnitInfo**

*#include <io.hpp>* Unit information

### Public Members

std::string **authName**

Authority name

std::string **code**

Code

std::string **name**

Name

std::string **category**

Category: one of "linear", "linear_per_time", "angular", "angular_per_time", "scale", "scale_per_time" or "time"

double **convFactor**

Conversion factor to the SI unit. It might be 0 in some cases to indicate no known conversion factor.

std::string **projShortName**

PROJ short name (may be empty)

bool **deprecated**

Whether the object is deprecated

class **FactoryException** : public osgeo::proj::*util*::*Exception*

*#include <io.hpp>* Exception thrown when a factory can't create an instance of the requested object.

Subclassed by *osgeo::proj::io::NoSuchAuthorityCodeException*

class **NoSuchAuthorityCodeException** : public osgeo::proj::*io*::*FactoryException*

*#include <io.hpp>* Exception thrown when an authority factory can't find the requested authority code.

### Public Functions

const std::string &**getAuthority**() const

Returns authority name.

const std::string &**getAuthorityCode**() const

Returns authority code.

## 10.6 Using PROJ in CMake projects

The recommended way to use the PROJ library in a CMake project is to link to the imported library target `PROJ::proj` provided by the CMake configuration which comes with the library. Typical usage is:

```
find_package(PROJ REQUIRED CONFIG)

target_link_libraries(MyApp PRIVATE PROJ::proj)
```

By adding the imported library target `PROJ::proj` to the target link libraries, CMake will also pass the include directories to the compiler.

The CMake command `find_package` will look for the configuration in a number of places. The lookup can be adjusted for all packages by setting the cache variable or environment variable `CMAKE_PREFIX_PATH`. In particular, CMake will consult (and set) the cache variable `PROJ_DIR`.

The old CMake name for the PROJ project was "PROJ4" and the switch to the name "PROJ" was made with version 7.0. As of PROJ 9.1, using `find_package(PROJ4)` will show a CMake Deprecation Warning. The old project name PROJ4 name will be phased out after the PROJ 9.x series.

## 10.7 Language bindings

PROJ bindings are available for a number of different development platforms.

### 10.7.1 Excel

proj.xll: Excel add-in for PROJ map projections

### 10.7.2 Fortran

Fortran-Proj: Bindings for PROJ in Fortran (By João Macedo @likeno)

### 10.7.3 Go (Golang)

go-proj: Go bindings for PROJ.

### 10.7.4 Java

PROJ-JNI: Java Native Interface for PROJ

### 10.7.5 Julia

Proj.jl: Julia bindings and idiomatic wrapper for PROJ.

### 10.7.6 MySQL

fProj4: Bindings for PROJ in MySQL

### 10.7.7 Perl

Geo::LibProj::FFI: Foreign function interface for PROJ (version 6 and newer) in Perl

### 10.7.8 Python

pyproj: Python interface (wrapper for PROJ)

### 10.7.9 Ruby

proj4rb: Bindings for PROJ in ruby

### 10.7.10 Rust

proj: Rust bindings for the latest stable version of PROJ

### 10.7.11 TCL

proj4tcl: Bindings for PROJ in tcl (critcl source)

### 10.7.12 Visual Basic

PROJ VB Wrappers: By Eric G. Miller.

## 10.8 Version 4 to 6 API Migration

This is a transition guide for developers wanting to migrate their code to use PROJ version 6.

### 10.8.1 Code example

The difference between the old and new API is shown here with a few examples. Below we implement the same program with the two different API's. The program reads input longitude and latitude from the command line and convert them to projected coordinates with the Mercator projection.

We start by writing the program for PROJ 4:

```
#include <proj_api.h>

main(int argc, char **argv) {
    projPJ pj_merc, pj_longlat;
    double x, y;
    int p;

    if (!(pj_longlat = pj_init_plus("+proj=longlat +ellps=clrk66")) )
        return 1;
    if (!(pj_merc = pj_init_plus("+proj=merc +datum=clrk66 +lat_ts=33")) )
        return 1;

    while (scanf("%lf %lf", &x, &y) == 2) {
        x *= DEG_TO_RAD; /* longitude */
        y *= DEG_TO_RAD; /* latitude */
        p = pj_transform(pj_longlat, pj_merc, 1, 1, &x, &y, NULL);
        printf("%.2f\t%.2f\n", x, y);
    }
```

(continues on next page)

(continued from previous page)

```
    pj_free(pj_longlat);
    pj_free(pj_merc);

    return 0;
}
```

The same program implemented using PROJ 6:

```
#include <proj.h>

main(int argc, char **argv) {
    PJ *P;
    PJ_COORD c, c_out;

    /* NOTE: the use of PROJ strings to describe CRS is strongly discouraged */
    /* in PROJ 6, as PROJ strings are a poor way of describing a CRS, and */
    /* more precise its geodetic datum. */
    /* Use of codes provided by authorities (such as "EPSG:4326", etc...) */
    /* or WKT strings will bring the full power of the "transformation */
    /* engine" used by PROJ to determine the best transformation(s) between */
    /* two CRS. */
    P = proj_create_crs_to_crs(PJ_DEFAULT_CTX,
                               "+proj=longlat +ellps=clrk66",
                               "+proj=merc +ellps=clrk66 +lat_ts=33",
                               NULL);
    if (P==0)
        return 1;

    {
        /* For that particular use case, this is not needed. */
        /* proj_normalize_for_visualization() ensures that the coordinate */
        /* order expected and returned by proj_trans() will be longitude, */
        /* latitude for geographic CRS, and easting, northing for projected */
        /* CRS. If instead of using PROJ strings as above, "EPSG:XXXX" codes */
        /* had been used, this might had been necessary. */
        PJ* P_for_GIS = proj_normalize_for_visualization(PJ_DEFAULT_CTX, P);
        if( 0 == P_for_GIS )  {
            proj_destroy(P);
            return 1;
        }
        proj_destroy(P);
        P = P_for_GIS;
    }

    /* For reliable geographic <--> geocentric conversions, z shall not */
    /* be some random value. Also t shall be initialized to HUGE_VAL to */
    /* allow for proper selection of time-dependent operations if one of */
    /* the CRS is dynamic. */
    c.lpzt.z = 0.0;
    c.lpzt.t = HUGE_VAL;
```

(continues on next page)

(continued from previous page)

```
    while (scanf("%lf %lf", &c.lpzt.lam, &c.lpzt.phi) == 2) {
        /* No need to convert to radian */
        c_out = proj_trans(P, PJ_FWD, c);
        printf("%.2f\t%.2f\n", c_out.xy.x, c_out.xy.y);
    }

    proj_destroy(P);

    return 0;
}
```

Further examples of using the PROJ API are in the *quick start* document.

### 10.8.2 Function mapping from old to new API

| Old API functions | New API functions |
|---|---|
| pj_fwd | *proj_trans()* |
| pj_inv | *proj_trans()* |
| pj_fwd3 | *proj_trans()* |
| pj_inv3 | *proj_trans()* |
| pj_transfor | *proj_create_crs_to_crs()* or *proj_create_crs_to_crs_from_pj()* + (*proj_normalize_for_visualization()* +) *proj_trans()*, *proj_trans_array()* or *proj_trans_generic()* |
| pj_init | *proj_create()* / *proj_create_crs_to_crs()* |
| pj_init | *proj_create()* / *proj_create_crs_to_crs()* |
| pj_free | *proj_destroy()* |
| pj_is_latlo | *proj_get_type()* |
| pj_is_geoce | *proj_get_type()* |
| pj_get_def | *proj_pj_info()* |
| pj_latlong_ | *No direct equivalent*, but can be accomplished by chaining *proj_create()*, *proj_crs_get_horizontal_datum()* and *proj_create_geographic_crs_from_datum()* |
| pj_set_find | *proj_context_set_file_finder()* |
| pj_set_sear | *proj_context_set_search_paths()* |
| pj_dealloca | *No equivalent* |
| pj_strerrno | *No equivalent* |
| pj_get_errn | *proj_errno()* |
| pj_get_rele | *proj_info()* |

### 10.8.3 Backward incompatibilities

Access to the `proj_api.h` is still possible but requires to define the `ACCEPT_USE_OF_DEPRECATED_PROJ_API_H` macro.

The emulation of the now deprecated `+init=epsg:XXXX` syntax in PROJ 6 is not fully compatible with previous versions.

In particular, when used with the `pj_transform()` function, no datum shift term (`towgs84`, `nadgrids`, `geoidgrids`) will be added during the expansion of the `+init=epsg:XXXX` string to `+proj=YYYY .....`. If you still uses `pj_transform()` and want datum shift to be applied, then you need to provide a fully expanded string with appropriate `towgs84`, `nadgrids` or `geoidgrids` terms to `pj_init()`.

To use the `+init=epsg:XXXX` syntax with *proj_create()* and then *proj_create_crs_to_crs()*, `proj_context_use_proj4_init_rules(ctx, TRUE)` or the `PROJ_USE_PROJ4_INIT_RULES=YES` environment variable must have been previously set. In that context, datum shift will be researched. However they might be different than with PROJ 4 or PROJ 5, since a "late-binding" approach will be used (that is trying to find as much as possible the most direct transformation between the source and target datum), whereas PROJ 4 or PROJ 5 used an "early-binding" approach consisting in always going to EPSG:4326 / WGS 84.

### 10.8.4 Feedback from downstream projects on the PROJ 6 migration

- PROJ 6 adoption by Spatialite
- On GDA2020, PROJ 6 and QGIS: Lessons learnt and recommendations for handling GDA2020 within geospatial software development

## 10.9 Version 4 to 5 API Migration

This is a transition guide for developers wanting to migrate their code to use PROJ version 5.

### 10.9.1 Background

Before we go on, a bit of background is needed. The new API takes a different view of the world than the old because it is needed in order to obtain high accuracy transformations. The old API is constructed in such a way that any transformation between two coordinate reference systems *must* pass through the ill-defined WGS84 reference frame, using it as a hub. The new API does away with this limitation to transformations in PROJ. It is still possible to do that type of transformations but in many cases there will be a better alternative.

The world view represented by the old API is always sufficient if all you care about is meter level accuracy - and in many cases it will provide much higher accuracy than that. But the view that "WGS84 is the *true* foundation of the world, and everything else can be transformed natively to and from WGS84" is inherently flawed.

First and foremost because any time WGS84 is mentioned, you should ask yourself "Which of the six WGS84 realizations are we talking about here?".

Second, because for many (especially legacy) systems, it may not be straightforward to transform to WGS84 (or actually ITRF-something, ETRS-something or NAD-something which appear to be the practical meaning of the term WGS84 in everyday PROJ related work), while centimeter-level accurate transformations may exist between pairs of older systems.

The concept of a hub reference frame ("datum") is not inherently bad, but in many cases you need to handle and select that datum with more care than the old API allows. The primary aim of the new API is to allow just that. And to do that, you must realize that the world is inherently 4 dimensional. You may in many cases assume one or more of the coordinates to be constant, but basically, to obtain geodetic accuracy transformations, you need to work in 4 dimensions.

Now, having described the background for introducing the new API, let's try to show how to use it. First note that in order to go from system A to system B, the old API starts by doing an **inverse** transformation from system A to WGS84, then does a **forward** transformation from WGS84 to system B.

With *cs2cs* being the command line interface to the old API, and *cct* being the same for the new, this example of doing the same thing in both world views will should give an idea of the differences:

```
$ echo 300000 6100000 | cs2cs +proj=utm +zone=33 +ellps=GRS80 +to +proj=utm +zone=32␣
↪+ellps=GRS80
683687.87       6099299.66 0.00


$ echo 300000 6100000 0 0 | cct +proj=pipeline +step +inv +proj=utm +zone=33␣
↪+ellps=GRS80 +step +proj=utm +zone=32 +ellps=GRS80
683687.8667   6099299.6624    0.0000    0.0000
```

Lookout for the +inv in the first +step, indicating an inverse transform.

### 10.9.2 Code example

The difference between the old and new API is shown here with a few examples. Below we implement the same program with the two different API's. The program reads input longitude and latitude from the command line and convert them to projected coordinates with the Mercator projection.

We start by writing the program for PROJ v. 4:

```
#include <proj_api.h>

main(int argc, char **argv) {
    projPJ pj_merc, pj_longlat;
    double x, y;

    if (!(pj_longlat = pj_init_plus("+proj=longlat +ellps=clrk66")) )
        return 1;
    if (!(pj_merc = pj_init_plus("+proj=merc +ellps=clrk66 +lat_ts=33")) )
        return 1;

    while (scanf("%lf %lf", &x, &y) == 2) {
        x *= DEG_TO_RAD; /* longitude */
        y *= DEG_TO_RAD; /* latitude */
        p = pj_transform(pj_longlat, pj_merc, 1, 1, &x, &y, NULL );
        printf("%.2f\t%.2f\n", x, y);
    }

    pj_free(pj_longlat);
    pj_free(pj_merc);

    return 0;
}
```

The same program implemented using PROJ v. 5:

```
#include <proj.h>

main(int argc, char **argv) {
    PJ *P;
    PJ_COORD c;

    P = proj_create(PJ_DEFAULT_CTX, "+proj=merc +ellps=clrk66 +lat_ts=33");
    if (P==0)
        return 1;

    while (scanf("%lf %lf", &c.lp.lam, &c.lp.phi) == 2) {
        c.lp.lam = proj_torad(c.lp.lam);
        c.lp.phi = proj_torad(c.lp.phi);
        c = proj_trans(P, PJ_FWD, c);
        printf("%.2f\t%.2f\n", c.xy.x, c.xy.y);
    }

    proj_destroy(P);
}
```

Looking at the two different programs, there's a few immediate differences that catches the eye. First off, the included header file describing the API has changed from `proj_api.h` to simply `proj.h`. All functions in `proj.h` belongs to the `proj_` namespace.

With the new API also comes new datatypes. E.g. the transformation object `projPJ` which has been changed to a pointer of type `PJ`. This is done to highlight the actual nature of the object, instead of hiding it away behind a typedef. New data types for handling coordinates have also been introduced. In the above example we use the `PJ_COORD`, which is a union of various types. The benefit of this is that it is possible to use the various structs in the union to communicate what state the data is in at different points in the program. For instance as in the above example where the coordinate is read from STDIN as a geodetic coordinate, communicated to the reader of the code by using the `c.lp` struct. After it has been projected we print it to STDOUT by accessing the individual elements in `c.xy` to illustrate that the coordinate is now in projected space. Data types are prefixed with *PJ_*.

The final, and perhaps biggest, change is that the fundamental concept of transformations in PROJ are now handled in a single transformation object (`PJ`) and not by stating the source and destination systems as previously. It is of course still possible to do just that, but the transformation object now captures the whole transformation from source to destination in one. In the example with the old API the source system is described as `+proj=latlon +ellps=clrk66` and the destination system is described as `+proj=merc +ellps=clrk66 +lat_ts=33`. Since the Mercator projection accepts geodetic coordinates as its input, the description of the source in this case is superfluous. We use that to our advantage in the new API and simply state the destination. This is simple at a glance, but is actually a big conceptual change. We are now focused on the path between two systems instead of what the source and destination systems are.

### 10.9.3 Function mapping from old to new API

| Old API functions | New API functions |
|---|---|
| `pj_fwd` | *proj_trans()* |
| `pj_inv` | *proj_trans()* |
| `pj_fwd3` | *proj_trans()* |
| `pj_inv3` | *proj_trans()* |
| `pj_transform` | *proj_trans_array()* or *proj_trans_generic()* |
| `pj_init` | *proj_create()* |
| `pj_init_plus` | *proj_create()* |
| `pj_free` | *proj_destroy()* |
| `pj_is_latlong` | *proj_angular_output()* |
| `pj_is_geocent` | *proj_angular_output()* |
| `pj_get_def` | *proj_pj_info()* |
| `pj_latlong_from_proj` | *No equivalent* |
| `pj_set_finder` | *No equivalent* |
| `pj_set_searchpath` | *No equivalent* |
| `pj_deallocate_grids` | *No equivalent* |
| `pj_strerrno` | *No equivalent* |
| `pj_get_errno_ref` | *proj_errno()* |
| `pj_get_release` | *proj_info()* |

The source code for PROJ is maintained in a git repository on GitHub. Additionally, a collection of PROJ-compatible transformation grids are maintained in a separate git repository.

**Attention**

The `projects.h` header and the functions related to it is considered deprecated from version 5.0.0 and onwards. The header has been removed PROJ in version 6.0.0 released February 1st 2019.

**Attention**

The nmake build system on Windows is on longer supported in version 6.0.0 on onwards. Use CMake instead.

**Attention**

The `proj_api.h` header and the functions related to it is considered deprecated from version 5.0.0 and onwards. The header has been removed in version 8.0.0 released March 1st 2021.

**Attention**

With the introduction of PROJ 5, behavioural changes has been made to existing functionality. Consult *Known differences between versions* for the details.

CHAPTER

ELEVEN

# SPECIFICATIONS

PROJ implements a number of extensions to standards, that are described below for the sake of broader interoperability.

## 11.1 PROJJSON

### 11.1.1 Introduction

PROJJSON is a JSON encoding of WKT2:2019 / ISO-19162:2019: Geographic information - Well-known text representation of coordinate reference systems, which itself implements the model of [ISO19111]. Apart from the difference of encodings, the semantics is intended to be exactly the same as WKT2:2019, and PROJJSON can be morphed losslessly from/into WKT2:2019.

PROJJSON is aimed at encoding definitions of coordinate reference systems (and their composing objects: datums, datum ensembles, coordinate systems, conversion) and coordinate operations.

### 11.1.2 Normative references

The PROJJSON specification requires prior knowledge of the following normative specifications:

- The JavaScript Object Notation (JSON) Data Interchange Format / IETF RFC 7159
- WKT2:2019 / ISO-19162:2019: Geographic information - Well-known text representation of coordinate reference systems

### 11.1.3 Definitions

- JavaScript Object Notation (JSON), and the terms object, member, name, value, array, number, true, false, and null, are to be interpreted as defined in [RFC7159].
- integer: JSON number whose value has no fractional/exponent part.
- All term and definitions from WKT2:2019 apply.

### 11.1.4 Schema

A JSON schema of PROJJSON grammar is available at https://proj.org/schemas/v0.7/projjson.schema.json

This schema defines a minimum set of constraints that apply to well-formed PROJJSON. Number of specific CRS and coordinate operation domain constraints are not expressed as JSON schema constraints: unless otherwise stated, the constraints (optional/mandatory/conditional character of information, restricted set of allowed values, etc.) defined in the WKT2:2019 specification also apply, as supplement to the JSON schema constraints.

### 11.1.5 History of the schema

- v0.7: added "accuracy" property in ConcatenatedOperation. Implemented in PROJ 9.2.1
- **v0.6:**
  - Implemented in PROJ 9.2
  - Additional optional "source_crs" property in "abridged_transformation".
  - Added CoordinateMetadata
  - Added "datum_epoch" property to GeodeticReferenceFrame and VerticalReferenceFrame
  - Added "minimum_value", "maximum_value" and "range_meaning" properties to Axis
  - Added "affine" in the CoordinateSystem.subtype enumeration.
- **v0.5:**
  - **Implemented in PROJ 9.1:**
    - add "meridian" member in Axis object type.
    - add "temporal_extent" and "vertical_extent" members in object usage.
    - add "geoid_models" for VerticalCRS if several models are needed.
    - add "deformation_models" members to GeodeticCRS and VerticalCRS (replaces "deformation_model" in DynamicGeodeticReferenceFrame/DynamicVerticalReferenceFrame)
  - **Added in schema, but not implemented in PROJ:**
    - add top-level object PointMotionOperation
- v0.4: additional properties allowed in id object (version, authority_citation, uri). Implemented in PROJ 8.2.
- v0.3: additional properties allowed in BoundCRS object (name, scope, area, bbox, usages, remarks, id, ids). Implemented in PROJ 8.1.
- v0.2: addition of geoid_model in VerticalCRS object.
- v0.1: initial version for PROJ 6.2

### 11.1.6 Specification

A PROJJSON text is a JSON object which has, at a minimum, a required `type` member, whose value is a string describing the nature of the encoded geodetic object.

An optional `$schema` member may be present, with its value being a string with a URL that points to the JSON schema that applies.

Objects may be composed of sub-objects (e.g a GeographicCRS is made of a Datum or DatumEnsemble and a coordinate system). The `type` member of the sub-objects can be omitted when there is no ambiguity. For example, in the object which is the value of a `coordinate_system` member, the `type` may be omitted. However, the value of the `datum` object of a GeographicCRS the `type` should be specified, as it can be either a GeodeticReferenceFrame or a DynamicGeodeticReferenceFrame. More formally, the `type` should be specified if the JSON schema specifies alternative types for the value of a member using the oneOf construct and those alternative types have a `type` member. Otherwise it may be omitted.

#### 11.1.6.1 High level objects

Objects described at the first level of a PROJJSON text have the following potential values of the `type` member:

- Coordinate Reference Systems (CRS):
  - Common ones:
    - `GeographicCRS`
    - `GeodeticCRS`
    - `ProjectedCRS`
    - `CompoundCRS`
    - `BoundCRS`
  - More esoteric ones:
    - `VerticalCRS`
    - `EngineeringCRS`
    - `TemporalCRS`
    - `ParametricCRS`
    - `DerivedGeographicCRS`
    - `DerivedGeodeticCRS`
    - `DerivedProjectedCRS`
    - `DerivedVerticalCRS`
    - `DerivedEngineeringCRS`
    - `DerivedTemporalCRS`
    - `DerivedParametricCRS`
- Coordinate operations:
  - `Transformation`
  - `Conversion`
  - `ConcatenatedOperation`
- Others:
  - `PrimeMeridian`
  - `Ellipsoid`
  - `Datum`
  - `DatumEnsemble`

#### 11.1.6.2 Identifiers

All objects mentioned above can have an optional `id` or `ids` member.

The value of `id` is a JSON object with the following members:

- `authority`: (required) value of type string. e.g "EPSG", "OGC", "IGNF", etc.
- `code`: (required) value of type string or integer. e.g 4326 or "CRS84"
- `authority_citation`: (optional) value of type string that may be used to give further details of the authority.

- `uri`: (optional) value of type string that may be used to give reference to an online resource.

An object can sometimes be identified in different ways, in which case the `ids` member can be used to specify a JSON array of objects with the same type of `id`.

Identifiers are allowed in top-level objects and inner objects. The WKT2:2019 specification recommends that if an object has an identifier, its inner objects should omit their identifiers, with the exceptions mentioned at http://docs.opengeospatial.org/is/18-010r7/18-010r7.html#37.

#### 11.1.6.3 Object usages

CRS and coordinate operation objects are derived classes (in object modeling terminology) of a "object usage" class. An object usage has the following optional members:

- `scope`: (optional) value of type string describing the purpose or purposes of the object. e.g "Geodesy, topographic mapping and cadastre"
- `area`: (optional) value of type string which describes a geographic area over which a CRS or coordinate operation is applicable. e.g. "World"
- `bbox`: (optional) value of type object, with 4 required members:
  - `east_longitude`: (required) number expressing the longitude in degrees of the eastern most part of the extent, within [-180,180] range.
  - `west_longitude`: (required) number expressing the longitude in degrees of the western most part of the extent, within [-180,180] range.. For an extent crossing the anti-meridian, west_longitude is lower than east_longitude.
  - `south_latitude`: (required) number expressing the latitude in degrees of the southern most part of the extent, within [-90,90] range.
  - `north_latitude`: (required) number expressing the latitude in degrees of the northern most part of the extent, within [-90,90] range.

  The coordinates are expressed in a unspecified datum, with the longitudes relative to the international reference meridian.
- `remarks`: (optional) value of type string with an informative text that does not modify the defining parameters of the object. e.g "Use NTv2 file for better accuracy"
- `id` (mutually exclusive with `ids`): (optional) Identifier of the object, as defined in *Identifiers*
- `ids` (mutually exclusive with `id`): (optional) Identifiers of the object, as defined in *Identifiers*

If several extents and scopes apply to an object, the `scope`, `area` and `bbox` members should not be used. Instead a `usages` member should be used, whose value is an array of objects, each of them accepting `scope` and/or `area` and/or `bbox` as members. While it is acceptable to use the `usages` construct for a single usage, it is recommended to avoid it and rather use instead the `scope`, `area` and `bbox` members.

#### 11.1.6.4 Units

A unit may be described either as:

- an object with the following members:
  - `type`: (required) one of the following types: `LinearUnit`, `AngularUnit`, `ScaleUnit`, `TimeUnit`, `ParametricUnit`, `Unit`
  - `name`: (required) string.

  - `conversion_factor`: (required in most cases, except in the temporal quantities of http://docs.opengeospatial.org/is/18-010r7/18-010r7.html#42) number that expresses a multiplicative factor to convert from the specified unit to a reference unit, as specified in http://docs.opengeospatial.org/is/18-010r7/18-010r7.html#41
  - `id` or `ids`: (optional, mutually exclusive)
- a string among the following enumeration: `metre`, `degree`, `unity`

Using a string value, when applicable, is recommended for brevity of the object definition.

#### 11.1.6.5 Omitted units in measured parameters

Most numeric parameters should generally be accompanied with the corresponding unit.

For example, for a projection parameter:

```
{
    "name": "False easting",
    "value": 500000,
    "unit": "metre"
}
```

or:

```
{
    "name": "False easting",
    "value": 700000,
    "unit": {
      "type": "LinearUnit",
      "name": "foot",
      "conversion_factor": 0.3048
    }
}
```

For the following cases, the unit may be omitted if it is metre: `semi_major_axis`, `semi_minor_axis` and `radius` members of an ellipsoid

For the following cases, the unit may be omitted if it is degree: `longitude` of a prime meridian.

#### 11.1.6.6 Coordinate system

In WKT, a ORDER keyword may be present in an axis definition. As the value of that element is equal to the index of axis in the list of axis of the coordinate system (with 1 as the value of the first index), it is absent from the PROJJSON encoding, to avoid any risk of misuse.

### 11.1.7 Examples

#### 11.1.7.1 GeographicCRS

##### Using a datum member, implicit prime meridian

The EPSG:6318 / "NAD83(2011)" geographic CRS can be expressed as

```
{
  "$schema": "https://proj.org/schemas/v0.4/projjson.schema.json",
  "type": "GeographicCRS",
```

(continues on next page)

(continued from previous page)

```
  "name": "NAD83(2011)",
  "datum": {
    "type": "GeodeticReferenceFrame",
    "name": "NAD83 (National Spatial Reference System 2011)",
    "ellipsoid": {
      "name": "GRS 1980",
      "semi_major_axis": 6378137,
      "inverse_flattening": 298.257222101
    }
  },
  "coordinate_system": {
    "subtype": "ellipsoidal",
    "axis": [
      {
        "name": "Geodetic latitude",
        "abbreviation": "Lat",
        "direction": "north",
        "unit": "degree"
      },
      {
        "name": "Geodetic longitude",
        "abbreviation": "Lon",
        "direction": "east",
        "unit": "degree"
      }
    ]
  },
  "scope": "Horizontal component of 3D system.",
  "area": "Puerto Rico - onshore and offshore. United States (USA) onshore and offshore -
↪ Alabama; Alaska; Arizona; Arkansas; California; Colorado; Connecticut; Delaware;␣
↪Florida; Georgia; Idaho; Illinois; Indiana; Iowa; Kansas; Kentucky; Louisiana; Maine;␣
↪Maryland; Massachusetts; Michigan; Minnesota; Mississippi; Missouri; Montana; Nebraska;
↪ Nevada; New Hampshire; New Jersey; New Mexico; New York; North Carolina; North Dakota;
↪ Ohio; Oklahoma; Oregon; Pennsylvania; Rhode Island; South Carolina; South Dakota;␣
↪Tennessee; Texas; Utah; Vermont; Virginia; Washington; West Virginia; Wisconsin;␣
↪Wyoming. US Virgin Islands - onshore and offshore.",
  "bbox": {
    "south_latitude": 14.92,
    "west_longitude": 167.65,
    "north_latitude": 74.71,
    "east_longitude": -63.88
  },
  "id": {
    "authority": "EPSG",
    "code": 6318
  }
}
```

Note the omission of a prime meridian member, which is conformant with the WKT2:2019 conditionality rules, as the prime meridian of the WGS 84 datum is the reference meridian / Greenwich.

### Using a datum member with an explicit prime meridian

For the EPSG:4806 / "Monte Mario (Rome)" geographic CRS, the prime meridian must be specified:

```
{
  "$schema": "https://proj.org/schemas/v0.4/projjson.schema.json",
  "type": "GeographicCRS",
  "name": "Monte Mario (Rome)",
  "datum": {
    "type": "GeodeticReferenceFrame",
    "name": "Monte Mario (Rome)",
    "ellipsoid": {
      "name": "International 1924",
      "semi_major_axis": 6378388,
      "inverse_flattening": 297
    },
    "prime_meridian": {
      "name": "Rome",
      "longitude": 12.4523333333333
    }
  },
  "coordinate_system": {
    "subtype": "ellipsoidal",
    "axis": [
      {
        "name": "Geodetic latitude",
        "abbreviation": "Lat",
        "direction": "north",
        "unit": "degree"
      },
      {
        "name": "Geodetic longitude",
        "abbreviation": "Lon",
        "direction": "east",
        "unit": "degree"
      }
    ]
  },
  "scope": "Geodesy, onshore minerals management.",
  "area": "Italy - onshore and offshore; San Marino, Vatican City State.",
  "bbox": {
    "south_latitude": 34.76,
    "west_longitude": 5.93,
    "north_latitude": 47.1,
    "east_longitude": 18.99
  },
  "id": {
    "authority": "EPSG",
    "code": 4806
  }
}
```

### Using a datum ensemble member

The WGS 84 geographic CRS may also be specified using a datum ensemble representation of the WGS 84 datum:


```
{
  "$schema": "https://proj.org/schemas/v0.4/projjson.schema.json",
  "type": "GeographicCRS",
  "name": "WGS 84",
  "datum_ensemble": {
    "name": "World Geodetic System 1984 ensemble",
    "members": [
      {
        "name": "World Geodetic System 1984 (Transit)",
        "id": {
          "authority": "EPSG",
          "code": 1166
        }
      },
      {
        "name": "World Geodetic System 1984 (G730)",
        "id": {
          "authority": "EPSG",
          "code": 1152
        }
      },
      {
        "name": "World Geodetic System 1984 (G873)",
        "id": {
          "authority": "EPSG",
          "code": 1153
        }
      },
      {
        "name": "World Geodetic System 1984 (G1150)",
        "id": {
          "authority": "EPSG",
          "code": 1154
        }
      },
      {
        "name": "World Geodetic System 1984 (G1674)",
        "id": {
          "authority": "EPSG",
          "code": 1155
        }
      },
      {
        "name": "World Geodetic System 1984 (G1762)",
        "id": {
          "authority": "EPSG",
          "code": 1156
        }
      },
      {
```


(continues on next page)

(continued from previous page)


```
        "name": "World Geodetic System 1984 (G2139)",
        "id": {
          "authority": "EPSG",
          "code": 1309
        }
      }
    ],
    "ellipsoid": {
      "name": "WGS 84",
      "semi_major_axis": 6378137,
      "inverse_flattening": 298.257223563
    },
    "accuracy": "2.0",
    "id": {
      "authority": "EPSG",
      "code": 6326
    }
  },
  "coordinate_system": {
    "subtype": "ellipsoidal",
    "axis": [
      {
        "name": "Geodetic latitude",
        "abbreviation": "Lat",
        "direction": "north",
        "unit": "degree"
      },
      {
        "name": "Geodetic longitude",
        "abbreviation": "Lon",
        "direction": "east",
        "unit": "degree"
      }
    ]
  },
  "scope": "Horizontal component of 3D system.",
  "area": "World.",
  "bbox": {
    "south_latitude": -90,
    "west_longitude": -180,
    "north_latitude": 90,
    "east_longitude": 180
  },
  "id": {
    "authority": "EPSG",
    "code": 4326
  }
}
```


The above is the output of the following invocation of the projinfo utility of the PROJ software version 9.0.0

```
projinfo EPSG:4326 -o PROJJSON -q
```

> **Note**
>
> PROJ versions prior to PROJ 8.0.0 used versions of the EPSG dataset that did not have the datum ensemble concept. Consequently they used a `datum` member instead of a `datum_ensemble`. The number of elements in the datum ensemble may also vary over time when new realizations of WGS 84 are added to the ensemble.

#### 11.1.7.2 ProjectedCRS

The EPSG:32631 / "WGS 84 / UTM zone 31N" projected CRS can be expressed as


```
{
    "$schema": "https://proj.org/schemas/v0.1/projjson.schema.json",
    "type": "ProjectedCRS",
    "name": "WGS 84 / UTM zone 31N",
    "base_crs": {
        "type": "GeographicCRS",
        "name": "WGS 84",
        "datum": {
            "type": "GeodeticReferenceFrame",
            "name": "World Geodetic System 1984",
            "ellipsoid": {
                "name": "WGS 84",
                "semi_major_axis": 6378137,
                "inverse_flattening": 298.257223563
            }
        },
        "coordinate_system": {
            "subtype": "ellipsoidal",
            "axis": [
            {
                "name": "Geodetic latitude",
                "abbreviation": "Lat",
                "direction": "north",
                "unit": "degree"
            },
            {
                "name": "Geodetic longitude",
                "abbreviation": "Lon",
                "direction": "east",
                "unit": "degree"
            }
            ]
        },
        "id": {
            "authority": "EPSG",
            "code": 4326
        }
    },
    "conversion": {
        "name": "UTM zone 31N",
        "method": {
            "name": "Transverse Mercator",
            "id": {
```


(continues on next page)

(continued from previous page)

```
                "authority": "EPSG",
                "code": 9807
            }
        },
        "parameters": [
        {
            "name": "Latitude of natural origin",
            "value": 0,
            "unit": "degree",
            "id": {
                "authority": "EPSG",
                "code": 8801
            }
        },
        {
            "name": "Longitude of natural origin",
            "value": 3,
            "unit": "degree",
            "id": {
                "authority": "EPSG",
                "code": 8802
            }
        },
        {
            "name": "Scale factor at natural origin",
            "value": 0.9996,
            "unit": "unity",
            "id": {
                "authority": "EPSG",
                "code": 8805
            }
        },
        {
            "name": "False easting",
            "value": 500000,
            "unit": "metre",
            "id": {
                "authority": "EPSG",
                "code": 8806
            }
        },
        {
            "name": "False northing",
            "value": 0,
            "unit": "metre",
            "id": {
                "authority": "EPSG",
                "code": 8807
            }
        }
        ]
    },
```

(continues on next page)

(continued from previous page)

```
    "coordinate_system": {
        "subtype": "Cartesian",
        "axis": [
        {
            "name": "Easting",
            "abbreviation": "E",
            "direction": "east",
            "unit": "metre"
        },
        {
            "name": "Northing",
            "abbreviation": "N",
            "direction": "north",
            "unit": "metre"
        }
        ]
    },
    "area": "World - N hemisphere - 0°E to 6°E - by country",
    "bbox": {
        "south_latitude": 0,
        "west_longitude": 0,
        "north_latitude": 84,
        "east_longitude": 6
    },
    "id": {
        "authority": "EPSG",
        "code": 32631
    }
}
```

**Note**

The `type` member of the `base_crs` member was omitted in PROJ export of PROJJSON prior to PROJ 9.6.

#### 11.1.7.3 CompoundCRS

The EPSG:9518 / "WGS 84 + EGM2008 height" compound CRS can be expressed as:

```
{
  "$schema": "https://proj.org/schemas/v0.4/projjson.schema.json",
  "type": "CompoundCRS",
  "name": "WGS 84 + EGM2008 height",
  "components": [
    {
      "type": "GeographicCRS",
      "name": "WGS 84",
      "datum_ensemble": {
        "name": "World Geodetic System 1984 ensemble",
        "members": [
          {
            "name": "World Geodetic System 1984 (Transit)",
```

(continues on next page)

(continued from previous page)


```
            "id": {
              "authority": "EPSG",
              "code": 1166
            }
          },
          {
            "name": "World Geodetic System 1984 (G730)",
            "id": {
              "authority": "EPSG",
              "code": 1152
            }
          },
          {
            "name": "World Geodetic System 1984 (G873)",
            "id": {
              "authority": "EPSG",
              "code": 1153
            }
          },
          {
            "name": "World Geodetic System 1984 (G1150)",
            "id": {
              "authority": "EPSG",
              "code": 1154
            }
          },
          {
            "name": "World Geodetic System 1984 (G1674)",
            "id": {
              "authority": "EPSG",
              "code": 1155
            }
          },
          {
            "name": "World Geodetic System 1984 (G1762)",
            "id": {
              "authority": "EPSG",
              "code": 1156
            }
          },
          {
            "name": "World Geodetic System 1984 (G2139)",
            "id": {
              "authority": "EPSG",
              "code": 1309
            }
          }
        ],
        "ellipsoid": {
          "name": "WGS 84",
          "semi_major_axis": 6378137,
          "inverse_flattening": 298.257223563
```


(continues on next page)

(continued from previous page)


```
        },
        "accuracy": "2.0",
        "id": {
          "authority": "EPSG",
          "code": 6326
        }
      },
      "coordinate_system": {
        "subtype": "ellipsoidal",
        "axis": [
          {
            "name": "Geodetic latitude",
            "abbreviation": "Lat",
            "direction": "north",
            "unit": "degree"
          },
          {
            "name": "Geodetic longitude",
            "abbreviation": "Lon",
            "direction": "east",
            "unit": "degree"
          }
        ]
      }
    },
    {
      "type": "VerticalCRS",
      "name": "EGM2008 height",
      "datum": {
        "type": "VerticalReferenceFrame",
        "name": "EGM2008 geoid"
      },
      "coordinate_system": {
        "subtype": "vertical",
        "axis": [
          {
            "name": "Gravity-related height",
            "abbreviation": "H",
            "direction": "up",
            "unit": "metre"
          }
        ]
      }
    }
  ],
  "scope": "Spatial referencing.",
  "area": "World.",
  "bbox": {
    "south_latitude": -90,
    "west_longitude": -180,
    "north_latitude": 90,
    "east_longitude": 180
```


(continues on next page)

(continued from previous page)

```
  },
  "id": {
    "authority": "EPSG",
    "code": 9518
  }
}
```

#### 11.1.7.4 BoundCRS

The Bound CRS, using as a base EPSG:4258 "ETRS89" geographic CRS, with an explicit transformation to WGS 84 using a null Helmert transformation, can be expressed as

```
{
  "$schema": "https://proj.org/schemas/v0.4/projjson.schema.json",
  "type": "BoundCRS",
  "source_crs": {
    "type": "GeographicCRS",
    "name": "ETRS89",
    "datum_ensemble": {
      "name": "European Terrestrial Reference System 1989 ensemble",
      "members": [
        {
          "name": "European Terrestrial Reference Frame 1989"
        },
        {
          "name": "European Terrestrial Reference Frame 1990"
        },
        {
          "name": "European Terrestrial Reference Frame 1991"
        },
        {
          "name": "European Terrestrial Reference Frame 1992"
        },
        {
          "name": "European Terrestrial Reference Frame 1993"
        },
        {
          "name": "European Terrestrial Reference Frame 1994"
        },
        {
          "name": "European Terrestrial Reference Frame 1996"
        },
        {
          "name": "European Terrestrial Reference Frame 1997"
        },
        {
          "name": "European Terrestrial Reference Frame 2000"
        },
        {
          "name": "European Terrestrial Reference Frame 2005"
        },
        {
```

(continues on next page)

(continued from previous page)


```
          "name": "European Terrestrial Reference Frame 2014"
        }
      ],
      "ellipsoid": {
        "name": "GRS 1980",
        "semi_major_axis": 6378137,
        "inverse_flattening": 298.257222101
      },
      "accuracy": "0.1"
    },
    "coordinate_system": {
      "subtype": "ellipsoidal",
      "axis": [
        {
          "name": "Geodetic latitude",
          "abbreviation": "Lat",
          "direction": "north",
          "unit": "degree"
        },
        {
          "name": "Geodetic longitude",
          "abbreviation": "Lon",
          "direction": "east",
          "unit": "degree"
        }
      ]
    },
    "id": {
      "authority": "EPSG",
      "code": 4258
    }
  },
  "target_crs": {
    "type": "GeographicCRS",
    "name": "WGS 84",
    "datum": {
      "type": "GeodeticReferenceFrame",
      "name": "World Geodetic System 1984",
      "ellipsoid": {
        "name": "WGS 84",
        "semi_major_axis": 6378137,
        "inverse_flattening": 298.257223563
      }
    },
    "coordinate_system": {
      "subtype": "ellipsoidal",
      "axis": [
        {
          "name": "Geodetic latitude",
          "abbreviation": "Lat",
          "direction": "north",
          "unit": "degree"
```


(continues on next page)

(continued from previous page)

```
        },
        {
          "name": "Geodetic longitude",
          "abbreviation": "Lon",
          "direction": "east",
          "unit": "degree"
        }
      ]
    },
    "id": {
      "authority": "EPSG",
      "code": 4326
    }
  },
  "transformation": {
    "name": "Transformation from unknown to WGS84",
    "method": {
      "name": "Position Vector transformation (geog2D domain)",
      "id": {
        "authority": "EPSG",
        "code": 9606
      }
    },
    "parameters": [
      {
        "name": "X-axis translation",
        "value": 0,
        "unit": "metre",
        "id": {
          "authority": "EPSG",
          "code": 8605
        }
      },
      {
        "name": "Y-axis translation",
        "value": 0,
        "unit": "metre",
        "id": {
          "authority": "EPSG",
          "code": 8606
        }
      },
      {
        "name": "Z-axis translation",
        "value": 0,
        "unit": "metre",
        "id": {
          "authority": "EPSG",
          "code": 8607
        }
      },
      {
```

(continues on next page)

(continued from previous page)

```
      "name": "X-axis rotation",
      "value": 0,
      "unit": {
        "type": "AngularUnit",
        "name": "arc-second",
        "conversion_factor": 4.84813681109536e-06
      },
      "id": {
        "authority": "EPSG",
        "code": 8608
      }
    },
    {
      "name": "Y-axis rotation",
      "value": 0,
      "unit": {
        "type": "AngularUnit",
        "name": "arc-second",
        "conversion_factor": 4.84813681109536e-06
      },
      "id": {
        "authority": "EPSG",
        "code": 8609
      }
    },
    {
      "name": "Z-axis rotation",
      "value": 0,
      "unit": {
        "type": "AngularUnit",
        "name": "arc-second",
        "conversion_factor": 4.84813681109536e-06
      },
      "id": {
        "authority": "EPSG",
        "code": 8610
      }
    },
    {
      "name": "Scale difference",
      "value": 0,
      "unit": {
        "type": "ScaleUnit",
        "name": "parts per million",
        "conversion_factor": 1e-06
      },
      "id": {
        "authority": "EPSG",
        "code": 8611
      }
    }
  ]
```

(continues on next page)

(continued from previous page)

```
  }
}
```

#### 11.1.7.5 Transformation

The EPSG:8549 / "NAD27 to NAD83 (8)" transformation can be expressed as:

```
{
  "$schema": "https://proj.org/schemas/v0.4/projjson.schema.json",
  "type": "Transformation",
  "name": "NAD27 to NAD83 (8)",
  "source_crs": {
    "type": "GeographicCRS",
    "name": "NAD27",
    "datum": {
      "type": "GeodeticReferenceFrame",
      "name": "North American Datum 1927",
      "ellipsoid": {
        "name": "Clarke 1866",
        "semi_major_axis": 6378206.4,
        "semi_minor_axis": 6356583.8
      }
    },
    "coordinate_system": {
      "subtype": "ellipsoidal",
      "axis": [
        {
          "name": "Geodetic latitude",
          "abbreviation": "Lat",
          "direction": "north",
          "unit": "degree"
        },
        {
          "name": "Geodetic longitude",
          "abbreviation": "Lon",
          "direction": "east",
          "unit": "degree"
        }
      ]
    },
    "id": {
      "authority": "EPSG",
      "code": 4267
    }
  },
  "target_crs": {
    "type": "GeographicCRS",
    "name": "NAD83",
    "datum": {
      "type": "GeodeticReferenceFrame",
      "name": "North American Datum 1983",
      "ellipsoid": {
```

(continues on next page)

(continued from previous page)

```
      "name": "GRS 1980",
      "semi_major_axis": 6378137,
      "inverse_flattening": 298.257222101
    }
  },
  "coordinate_system": {
    "subtype": "ellipsoidal",
    "axis": [
      {
        "name": "Geodetic latitude",
        "abbreviation": "Lat",
        "direction": "north",
        "unit": "degree"
      },
      {
        "name": "Geodetic longitude",
        "abbreviation": "Lon",
        "direction": "east",
        "unit": "degree"
      }
    ]
  },
  "id": {
    "authority": "EPSG",
    "code": 4269
  }
},
"method": {
  "name": "NADCON5 (2D)",
  "id": {
    "authority": "EPSG",
    "code": 1074
  }
},
"parameters": [
  {
    "name": "Latitude difference file",
    "value": "nadcon5.nad27.nad83_1986.alaska.lat.trn.20160901.b",
    "id": {
      "authority": "EPSG",
      "code": 8657
    }
  },
  {
    "name": "Longitude difference file",
    "value": "nadcon5.nad27.nad83_1986.alaska.lon.trn.20160901.b",
    "id": {
      "authority": "EPSG",
      "code": 8658
    }
  }
],
```

(continues on next page)

(continued from previous page)

```
  "accuracy": "0.5",
  "scope": "Geodesy.",
  "area": "United States (USA) - Alaska.",
  "bbox": {
    "south_latitude": 51.3,
    "west_longitude": 172.42,
    "north_latitude": 71.4,
    "east_longitude": -129.99
  },
  "id": {
    "authority": "EPSG",
    "code": 8549
  },
  "remarks": "Uses NADCON5 method which expects longitudes positive east in range 0-360°;
↪ source and target CRSs have longitudes positive east in range -180° to +180°.␣
↪Accuracy at 67% confidence level is 0.5m onshore, 5m nearshore and undetermined␣
↪farther offshore."
}
```

### 11.1.8 Deviations with the WKT2:2019 specification

While most of this specification is intended to be interoperable with WKT2:2019, there are a few deviations, reflecting the needs of the PROJ software implementation.

#### 11.1.8.1 PROJJSON extensions

This specification allows a Bound CRS to be used wherever a CRS object is allowed in the OGC Topic 2 abstract specification / ISO-19111:2019. In particular, the members of a compound CRS can be a Bound CRS in this specification, whereas OGC Topic 2 abstract specification restricts it to single CRS. A Bound CRS can also be used as the source or target of a coordinate operation.

#### 11.1.8.2 PROJJSON omissions

This specification does not define an encoding for triaxial ellipsoid (`TRIAXIAL` WKT keyword)

### 11.1.9 Reference implementation

PROJJSON is available as input and output of the PROJ software since PROJ 6.2.

### 11.1.10 Online CRS definitions as PROJJSON

The https://spatialreference.org/ website features CRS definitions for various authorities under a number of formats, including PROJJSON. For example: https://spatialreference.org/ref/epsg/4326/projjson.json

### 11.1.11 Copyright and License information

This document is copyright Even Rouault and PROJ contributors, 2019-2024, and subject to the MIT license.

## 11.2 Geodetic TIFF grids (GTG)

Added in version 7.0.

### 11.2.1 Introduction

The Geodetic TIFF grid format has been introduced per *PROJ RFC 4: Remote access to grids and GeoTIFF grids*. It is a profile of the TIFF and GeoTIFF formats that addresses the specific requirements of geodetic grids: horizontal shifts, vertical shifts, velocity grids, etc... It also follows the Cloud Optimized GeoTIFF principles.

Such grids are available on a *CDN of GeoTIFF grids*.

### 11.2.2 General description

The general principles that guide the following requirements and recommendations are such that files will be properly recognized by PROJ, and also by GDAL which is an easy way to inspect such grid files:

- TIFF 6.0 based (could possibly be BigTIFF without code changes, if we ever need some day to handle grids larger than 4GB)
- GeoTIFF 1.1 for the georeferencing. GeoTIFF 1.1 is a recent standard, compared to the original GeoTIFF 1.0 version, but its backward compatibility is excellent, so that should not cause much trouble to readers that are not official GeoTIFF 1.1 compliant.
- Files hosted on the CDN will use a Geographic 2D CRS for the GeoTIFF GeoKeys. That CRS is intended to be the interpolation CRS as defined in [ISO19111], that is the CRS to which grid values are referred to.

  Given that they will nominally be related to the EPSG dataset, the GeodeticCRSGeoKey will be used to store the EPSG code of the CRS. If the CRS cannot be reliably encoded through that key or other geokeys, the `interpolation_crs_wkt` metadata item detailed afterwards should be used.

  This CRS will be generally the source CRS (for geographic to geographic horizontal shift grids, or geographic to vertical shift grids), but for vertical to vertical CRS adjustment, this will be the geographic CRS to which the grid is referenced. In some very rare cases of geographic to vertical shift grids, the interpolation CRS might be a geographic CRS that is not the same as the source CRS (into which ellipsoidal height are expressed). The only instance we have in mind is for the EPSG:7001 "ETRS89 to NAP height (1)" transformation using the naptrans2008 VDatum-grid which is referenced to Amersfoort EPSG:4289 instead of ETRS89...

  On the reading side, PROJ will ignore that information: the CRS is already stored in the source_crs or interpolation_crs column of the grid_transformation table.

  For geographic to vertical shift files (geoid models), the GeoTIFF 1.1 convention will be used to store the value of the VerticalGeoKey So a geoid model that apply to WGS 84 EPSG:4979 will have GeodeticCRSGeoKey = 4326 and VerticalGeoKey = 4979.
- Files hosted on the CDN will use the GeoTIFF defined ModelTiepointTag and ModelPixelScaleTag TIFF tags to store the coordinates of the upper-left pixel and the resolution of the pixels. On the reading side, they will be required and ModelTransformationTag will be ignored.

  **Note**

  Regarding anti-meridian handling, a variety of possibilities exist. We do not attempt to standardize this and files hosted on the CDN will use a georeferencing close to the original data producer. For example, NOAA vertical grids that apply to Conterminous USA might even have a top-left longitude beyond 180 (for consistency with Alaska grids, whose origin is < 180) Anti-meridian handling in PROJ has probably issues. This RFC does not attempt to address them in particular, as they are believed to be orthogonal to the topics it covers, and being mostly implementation issues.

- Files hosted on the CDN will use the GTRasterTypeGeoKey = PixelIsPoint convention. This is the convention used by most existing grid formats currently. Note that GDAL typically use a PixelIsArea convention (but can handle both conventions), so the georeferencing it displays when opening a .gsb or .gtx file appears to have a half-pixel shift regarding to the coordinates stored in the original grid file. On the reading side, PROJ will accept

both conventions (for equivalent georeferencing, the value of the origin in a PixelIsArea convention is shifted by a half-pixel towards the upper-left direction). Unspecified behavior if this GeoKey is absent.

- Files hosted on the CDN will be tiled, presumably with 256x256 tiles (small grids that are smaller than 256x256 will use a single strip). On the reading side, PROJ will accept TIFF files with any strip or tile organization. Tiling is expressed by specifying the TileWidth, TileHeight, TileOffsets and TileByteCounts tags. Strip organization is expressed by specifying the RowsPerStrip, StripByteCounts and StripOffsets tags.
- Files hosted on the CDN will use Compression = DEFLATE or LZW (to be determined, possibly with Predictor = 2 or 3) On the reading side, PROJ will accept TIFF files with any compression method (appropriate for the data types and PhotometricInterpretation considered) supported by the libtiff build used by PROJ. Of course uncompressed files will be supported.
- Files hosted on the CDN will use little-endian byte ordering. On the reading side, libtiff will transparently handle both little-endian and big-endian ordering.
- Files hosted on the CDN will use PlanarConfiguration=Separate. The tools described in a later section will order blocks so that blocks needed for a given location are close to each other. On the reading side, PROJ will handle also PlanarConfiguration=Contig.
- Files hosted on the CDN will generally use Float32 (BitsPerSample=32 and SampleFormat=IEEEFP) Files may be created using Signed Int 16 ( BitsPerSample =16 and SampleFormat = INT), Unsigned Int 16 (BitsPerSample=16 and SampleFormat=UINT), Signed Int 32 or Unsigned Int 32 generally with an associate scale/offset. On the reading side, only those three data types will be supported as well.
- Files hosted on the CDN will have a PhotometricInterpretation = MinIsBlack. It will be assumed, and ignored on the reading side.
- Files hosted on the CDN will nominally have:
  - SamplesPerPixel = 2 for horizontal shift grid, with the first sample being the longitude offset and the second sample being the latitude offset.
  - SamplesPerPixel = 1 for vertical shift grids.
  - SamplesPerPixel = 3 for deformation models combining horizontal and vertical adjustments.

  And even for the current identified needs of horizontal or vertical shifts, more samples may be present (to indicate for example uncertainties), but will be ignored by PROJ.

  The ExtraSamples tag should be set to a value of SamplesPerPixel - 1 (given the rules that apply for PhotometricInterpretation = MinIsBlack)
- The ImageDescription tag may be used to convey extra information about the name, provenance, version and last updated date of the grid. Will be set when possible for files hosted on the CDN. Ignored by PROJ.
- The Copyright tag may be used to convey extra information about the copyright and license of the grid. Will be set when possible for files hosted on the CDN. Ignored by PROJ.
- The DateTime tag may be used to convey the date at which the file has been created or converted. In case of a file conversion, for example from NTv2, this will be the date at which the conversion has been performed. The `ImageDescription` tag however will contain the latest of the CREATED or UPDATED fields from the NTv2 file. Will be set when possible for files hosted on the CDN. Ignored by PROJ.
- Files hosted on the CDN will use the GDAL_NODATA tag to encode the value of the nodata / missing value, when it applies to the grid.

  If offset and/or scaling is used, the nodata value corresponds to the raw value, before applying offset and scaling. The value found in this tag, if present, will be honoured (to the extent to which current PROJ code makes use of nodata). For floating point data, writers are strongly discouraged to use non-finite values (+/- infinity, NaN) of nodata to maximimize interoperability. The GDAL_NODATA value applies to all samples of a given TIFF IFD.

- Files hosted on the CDN will use the GDAL_METADATA tag to encode extra metadata not supported by baseline or extended TIFF.
  - The root XML node should be `GDALMetadata`
  - Zero, one or several child XML nodes `Item` may be present.
  - A Item should have a `name` attribute, and a child text node with its value. `role` and `sample` attributes may be present for attributes that have a special semantics (recognized by GDAL). The value of *sample* should be a integer value between 0 and number_of_samples - 1.
  - Scale and offset to convert integer raw values to floating point values may be expressed with XML *Item* elements whose name attribute is respectively `SCALE` and `OFFSET`, and their `role` attribute is respectively `scale` and `offset`. The decoded value will be: {offset} + {scale} * raw_value_from_geotiff_file

    For a offset value of 1 and scaling of 2, the following payload should be stored:

```
<GDALMetadata>
    <Item name="OFFSET" sample="0" role="offset">1</Item>
    <Item name="SCALE" sample="0" role="scale">2</Item>
</GDALMetadata>
```

  - The type of the grid must be specified with a *Item* whose `name` is set to `TYPE`.

    Values recognized by PROJ currently are:

    * `HORIZONTAL_OFFSET`: implies the presence of at least two samples. For grids referenced in geographic coordinates, the first sample must contain the latitude offset and the second sample must contain the longitude offset. For grids referenced in projected coordinates (supported since PROJ 9.4), the first sample must contain the easting offset and the second sample must contain the northing offset. The offset may also be expressed as a speed per year for temporal gridshifting. Corresponds to PROJ *Horizontal grid shift* (only for grids referenced in geographic coordinates) and *General grid shift* methods (for grids referenced both in geographic and projected coordinates)
    * `GEOGRAPHIC_3D_OFFSET`: implies the presence of at least 3 samples. The first sample must contain the latitude offset, the second sample must contain the longitude offset and the third one the ellipsoidal height difference. Typically used for NADCON5 grids. Added in PROJ 9.2 Corresponds to PROJ *General grid shift* method.
    * `ELLIPSOIDAL_HEIGHT_OFFSET`: implies the presence of one sample with the ellipsoidal height difference. Generally used in combination with another grid of type `HORIZONTAL_OFFSET` to perform Geographic 3D offsetting when the horizontal and vertical grids do not have the same resolution, as found in some NADCON5 grids. Added in PROJ 9.2 Corresponds to PROJ *General grid shift* method.
    * `VERTICAL_OFFSET_GEOGRAPHIC_TO_VERTICAL`: implies the presence of at least one sample. The first sample must contain the vertical adjustment. Must be used when the source/interpolation CRS is a Geographic CRS and the target CRS a Vertical CRS. Corresponds to PROJ *Vertical grid shift* method.
    * `VERTICAL_OFFSET_VERTICAL_TO_VERTICAL`: implies the presence of at least one sample. The first sample must contain the vertical adjustment. Must be used when the source and target CRS are Vertical CRS. Corresponds to PROJ *Vertical grid shift* method.
    * `GEOCENTRIC_TRANSLATION`: implies the presence of at least 3 samples. The first 3 samples must be respectively the geocentric adjustments along the X, Y and Z axis. Must be used when the source and target CRS are geocentric CRS. The interpolation CRS must be a geographic CRS. Corresponds to PROJ *Geocentric grid shift* method.
    * `VELOCITY`: implies the presence of at least 3 samples. The first 3 samples must be respectively the velocities along the E(ast), N(orth), U(p) axis in the local topocentric coordinate system. Corresponds to PROJ *Kinematic datum shifting utilizing a deformation model* method.

    * DEFORMATION_MODEL: implies the presence of the DISPLACEMENT_TYPE and UNCERTAINTY_TYPE metadata items. Corresponds to PROJ *Multi-component time-based deformation model* method.

  For example:

```
<Item name="TYPE">HORIZONTAL_OFFSET</Item>
```

- The description of each sample must be specified with a Item whose name attribute is set to DESCRIPTION and role attribute to description.

  Values recognized by PROJ for this Item are currently:

  * latitude_offset: valid for TYPE=HORIZONTAL_OFFSET or GEOGRAPHIC_3D_OFFSET . Sample values should be the value to add a latitude expressed in the CRS encoded in the GeoKeys to obtain a latitude value expressed in the target CRS.
  * longitude_offset: valid for TYPE=HORIZONTAL_OFFSET or GEOGRAPHIC_3D_OFFSET . Sample values should be the value to add a longitude expressed in the CRS encoded in the GeoKeys to obtain a longitude value expressed in the target CRS.
  * easting_offset: valid for TYPE=HORIZONTAL_OFFSET. Sample values should be the value to add a easting expressed in the CRS encoded in the GeoKeys to obtain a easting value expressed in the target CRS.
  * northing_offset: valid for TYPE=HORIZONTAL_OFFSET. Sample values should be the value to add a northing expressed in the CRS encoded in the GeoKeys to obtain a northing value expressed in the target CRS.
  * ellipsoidal_height_offset: valid for TYPE=ELLIPSOIDAL_HEIGHT_OFFSET or GEOGRAPHIC_3D_OFFSET . Sample values should be the value to add to the ellipsoidal height of the source CRS to obtain the ellipsoidal height of the target CRS.
  * geoid_undulation: valid for TYPE=VERTICAL_OFFSET_GEOGRAPHIC_TO_VERTICAL. For a source CRS being a geographic CRS and a target CRS being a vertical CRS, sample values should be the value to add to a geoid-related height (that is expressed in the target CRS) to get an ellipsoidal height (that is expressed in the source CRS), also called the geoid undulation. The value is positive upward from the reference ellipsoid. Note the possible confusion related to what is the source CRS and target CRS and the semantics of the value stored (to convert from the source to the target, one must subtract the value contained in the grid). This is the convention used by the EPSG:9665 operation method.
  * hydroid_height: valid for TYPE=VERTICAL_OFFSET_GEOGRAPHIC_TO_VERTICAL. For a source CRS being a geographic CRS and a target CRS being a vertical CRS representing a tidal surface (typically Chart Datum, Highest/Lowest Astronomical Tide (HAT/LAT), High/Low Waters, Mean Sea Level), the hydroid height is the value to add to a height related to a tidal surface (that is expressed in the target CRS) to get an ellipsoidal height (that is expressed in the source CRS). The value is positive upward from the reference ellipsoid. Note the possible confusion related to what is the source CRS and target CRS and the semantics of the value stored (to convert from the source to the target, one must subtract the value contained in the grid). This is the convention used by the EPSG:1115 operation method.
  * vertical_offset: valid for TYPE=VERTICAL_OFFSET_VERTICAL_TO_VERTICAL. For a source and target CRS being vertical CRS, sample values should be the value to add to an elevation expressed in the source CRS to obtain a longitude value expressed in the target CRS.
  * x_translation / y_translation / z_translation: valid for TYPE=GEOCENTRIC_TRANSLATION. Sample values should be the value to add to the input geocentric coordinates expressed in the source CRS to geocentric coordinates expressed in the target CRS.

  - `east_velocity` / `north_velocity` / `up_velocity`: valid for TYPE=VELOCITY. Sample values should be the velocity in a linear/time unit in a ENU local topocentric coordinate system.
  - `east_offset` / `north_offset` / `vertical_offset`: valid for TYPE=DEFORMATION_MODEL. For east_offset and north_offset, the unit might be degree or metre. For vertical_offset, the unit must be metre.

  For example:

```
<Item name="DESCRIPTION" sample="0" role="description">latitude_offset</Item>
<Item name="DESCRIPTION" sample="1" role="description">longitude_offset</Item>
```

  Other values may be used (not used by PROJ):

  - `latitude_offset_accuracy`: valid for TYPE=HORIZONTAL_OFFSET. Sample values should be the accuracy of corresponding latitude_offset samples. Generally in metre (if converted from NTv2)
  - `longitude_offset_accuracy`: valid for TYPE=HORIZONTAL_OFFSET. Sample values should be the accuracy of corresponding longitude_offset samples. Generally in metre (if converted from NTv2)

- The sign convention for the values of the `longitude_offset` channel should be indicated with an Item named `positive_value` whose value can be `west` or `east`. NTv2 products originally use a `west` convention, but when converting from them to GeoTIFF, the sign of those samples will be inverted so they use a more natural `east` convention. If this item is absent, the default value is `east`.
- The unit of the values stored in the grid must be specified for each sample through an Item of name `UNITTYPE` and role `unittype` Valid values should be the name of entries from the EPSG `unitofmeasure` table. To maximize interoperability, writers are strongly encouraged to limit themselves to the following values:

  For linear units:

  - `metre` (default value assumed if absent for vertical shift grid files, and value used for files stored on PROJ CDN)
  - `US survey foot`

  For angular units:

  - `degree`
  - `arc-second` (default value assumed if absent for longitude and latitude offset samples of horizontal shift grid files, and value used for files stored on PROJ CDN)
  - `arc-seconds per year` (when used with the *Horizontal grid shift* method with temporal gridshifting)

  For velocity units:

  - `millimetres per year`

  The longitude and latitude offset samples should use the same unit. The geocentric translation samples should use the same unit. The velocity samples should use the same unit.

  Example:

```
<Item name="UNITTYPE" sample="0" role="unittype">arc-second</Item>
<Item name="UNITTYPE" sample="1" role="unittype">arc-second</Item>
```

- For grids with TYPE=HORIZONTAL_OFFSET and with a `easting_offset` and `northing_offset` channel, an extra offset to apply after the grid correction in the forward direction of the transformation can

be specified with a `constant_offset` metadata item, expressed in the same units as the grid values (only metre supported at the moment).

Example:

```
<Item name="constant_offset" sample="0">-5000000</Item>
<Item name="constant_offset" sample="1">-5000000</Item>
```

- For TYPE=DEFORMATION_MODEL, the type of the displacement must be specified with a *Item* whose `name` is set to `DISPLACEMENT_TYPE`.

  The accepted values are: `HORIZONTAL`, `VERTICAL`, `3D` or `NONE`

- For TYPE=DEFORMATION_MODEL, the type of the uncertainty must be specified with a *Item* whose `name` is set to `UNCERTAINTY_TYPE`.

  The accepted values are: `HORIZONTAL`, `VERTICAL`, `3D` or `NONE`

- The `target_crs_epsg_code` metadata item should be present. For a horizontal shift grid, this is the EPSG code of the target geographic CRS. For a vertical shift grid, this is the EPSG code of a the target vertical CRS. If the target CRS has no associated EPSG code, `target_crs_wkt` must be used. Ignored by PROJ currently.
- The `target_crs_wkt` metadata item must be present if the `target_crs_epsg_code` cannot be used. Its value should be a valid WKT string according to WKT:2015 or WKT:2019 Ignored by PROJ currently.
- The `source_crs_epsg_code` metadata item must be present if the source and interpolation CRS are not the same (typical use case is vertical CRS to vertical CRS transformation), because the GeoKeys encode the interpolation CRS and not the source CRS. If the source CRS has no associated EPSG code, `source_crs_wkt` must be used. Ignored by PROJ currently.
- The `source_crs_wkt` metadata item must be present if the `source_crs_epsg_code` cannot be used. Its value should be a valid WKT string according to WKT:2015 or WKT:2019. Ignored by PROJ currently.
- The `interpolation_crs_wkt` metadata item may be present if the GeoKeys cannot be used to express reliably the interpolation CRS. Its value should be a valid WKT string according to WKT:2015 or WKT:2019. Ignored by PROJ currently.
- The `recommended_interpolation_method` metadata item may be present to describe the method to use to interpolation values at locations not coincident with nodes stored in the grid file. Potential values: `bilinear`, `bicubic`. Ignored by PROJ currently.
- The `area_of_use` metadata item can be used to indicate plain text information about the area of use of the grid (like "USA - Wisconsin"). In case of multiple subgrids, it should be set only on the first one, but applies to the whole set of grids, not just the first one.
- The `grid_name` metadata item should be present if there are subgrids for this grid (that is grids whose extent is contained in the extent of this grid), or if this is a subgrid. It is intended to be a relatively short identifier Will be ignored by PROJ (this information can be inferred by the grids extent)
- The `parent_grid_name` metadata item should be present if this is a subgrid and its value should be equal to the parent's `grid_name` Will be ignored by PROJ (this information can be inferred by the grids extent)
- The `number_of_nested_grids` metadata item should be present if there are subgrids for this grid (that is grids whose extent is contained in the extent of this grid). Will be ignored by PROJ (this information can be inferred by the grids extent)
- The `interpolation_method` metadata item may be present to indicate the interpolation method to apply. `bilinear` or `biquadratic` are supported. If not specified, defaults to `bilinear`.

### 11.2.3 Example

https://github.com/OSGeo/PROJ-data/blob/master/fr_ign/fr_ign_ntf_r93.tif has been converted from https://github.com/OSGeo/proj-datumgrid/blob/master/ntf_r93.gsb with https://github.com/OSGeo/PROJ-data/blob/master/grid_tools/ntv2_to_gtiff.py

```
$ tiffinfo ntf_r93.tif

TIFF Directory at offset 0x4e (78)
Image Width: 156 Image Length: 111
Bits/Sample: 32
Sample Format: IEEE floating point
Compression Scheme: AdobeDeflate
Photometric Interpretation: min-is-black
Extra Samples: 3<unspecified, unspecified, unspecified>
Samples/Pixel: 4
Rows/Strip: 111
Planar Configuration: separate image planes
ImageDescription: NTF (EPSG:4275) to RGF93 (EPSG:4171). Converted from ntf_r93.gsb␣
↪(version IGN07_01, last updated on 2007-10-31)
DateTime: 2019:12:09 00:00:00
Copyright: Derived from work by IGN France. Open License https://www.etalab.gouv.fr/wp-
↪content/uploads/2014/05/Open_Licence.pdf
Tag 33550: 0.100000,0.100000,0.000000
Tag 33922: 0.000000,0.000000,0.000000,-5.500000,52.000000,0.000000
Tag 34735: 1,1,1,3,1024,0,1,2,1025,0,1,2,2048,0,1,4275
Tag 42112: <GDALMetadata>
<Item name="grid_name">FRANCE</Item>
<Item name="target_crs_epsg_code">4171</Item>
<Item name="TYPE">HORIZONTAL_OFFSET</Item>
<Item name="UNITTYPE" sample="0" role="unittype">arc-second</Item>
<Item name="DESCRIPTION" sample="0" role="description">latitude_offset</Item>
<Item name="positive_value" sample="1">east</Item>
<Item name="UNITTYPE" sample="1" role="unittype">arc-second</Item>
<Item name="DESCRIPTION" sample="1" role="description">longitude_offset</Item>
<Item name="UNITTYPE" sample="2" role="unittype">arc-second</Item>
<Item name="DESCRIPTION" sample="2" role="description">latitude_offset_accuracy</Item>
<Item name="UNITTYPE" sample="3" role="unittype">arc-second</Item>
<Item name="DESCRIPTION" sample="3" role="description">longitude_offset_accuracy</Item>
</GDALMetadata>

Predictor: floating point predictor 3 (0x3)
```

```
$ listgeo ntf_r93.tif

Geotiff_Information:
    Version: 1
    Key_Revision: 1.1
    Tagged_Information:
        ModelTiepointTag (2,3):
            0                 0                 0
            -5.5              52                0
        ModelPixelScaleTag (1,3):
            0.1               0.1               0
```

(continues on next page)

(continued from previous page)

```
        End_Of_Tags.
    Keyed_Information:
        GTModelTypeGeoKey (Short,1): ModelTypeGeographic
        GTRasterTypeGeoKey (Short,1): RasterPixelIsPoint
        GeodeticCRSGeoKey (Short,1): Code-4275 (NTF)
        End_Of_Keys.
    End_Of_Geotiff.

GCS: 4275/NTF
Datum: 6275/Nouvelle Triangulation Francaise
Ellipsoid: 7011/Clarke 1880 (IGN) (6378249.20,6356515.00)
Prime Meridian: 8901/Greenwich (0.000000/  0d 0' 0.00"E)
Projection Linear Units: User-Defined (1.000000m)

Corner Coordinates:
Upper Left    (  5d30' 0.00"W, 52d 0' 0.00"N)
Lower Left    (  5d30' 0.00"W, 40d54' 0.00"N)
Upper Right   ( 10d 6' 0.00"E, 52d 0' 0.00"N)
Lower Right   ( 10d 6' 0.00"E, 40d54' 0.00"N)
Center        (  2d18' 0.00"E, 46d27' 0.00"N)
```

```
$ gdalinfo ntf_r93.tif

Driver: GTiff/GeoTIFF
Files: ntf_r93.tif
Size is 156, 111
Coordinate System is:
GEOGCRS["NTF",
    DATUM["Nouvelle Triangulation Francaise",
        ELLIPSOID["Clarke 1880 (IGN)",6378249.2,293.466021293627,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4275]]
Data axis to CRS axis mapping: 2,1
Origin = (-5.550000000000000,52.049999999999997)
Pixel Size = (0.100000000000000,-0.100000000000000)
Metadata:
  AREA_OR_POINT=Point
  grid_name=FRANCE
  target_crs_epsg_code=4171
  TIFFTAG_DATETIME=2019:12:09 00:00:00
  TIFFTAG_IMAGEDESCRIPTION=NTF (EPSG:4275) to RGF93 (EPSG:4171). Converted from ntf_r93.
↪gsb (version IGN07_01, last updated on 2007-10-31)
  TYPE=HORIZONTAL_OFFSET
```

(continues on next page)

(continued from previous page)

```
Image Structure Metadata:
  COMPRESSION=DEFLATE
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  (  -5.5500000,  52.0500000) (  5d33' 0.00"W, 52d 3' 0.00"N)
Lower Left  (  -5.5500000,  40.9500000) (  5d33' 0.00"W, 40d57' 0.00"N)
Upper Right (  10.0500000,  52.0500000) ( 10d 3' 0.00"E, 52d 3' 0.00"N)
Lower Right (  10.0500000,  40.9500000) ( 10d 3' 0.00"E, 40d57' 0.00"N)
Center      (   2.2500000,  46.5000000) (  2d15' 0.00"E, 46d30' 0.00"N)
Band 1 Block=156x111 Type=Float32, ColorInterp=Gray
  Description = latitude_offset
  Unit Type: arc-second
Band 2 Block=156x111 Type=Float32, ColorInterp=Undefined
  Description = longitude_offset
  Unit Type: arc-second
  Metadata:
    positive_value=east
Band 3 Block=156x111 Type=Float32, ColorInterp=Undefined
  Description = latitude_offset_accuracy
  Unit Type: arc-second
Band 4 Block=156x111 Type=Float32, ColorInterp=Undefined
  Description = longitude_offset_accuracy
  Unit Type: arc-second
```

### 11.2.4 Multi-grid storage

Formats like NTv2 can contain multiple subgrids. This can be transposed to TIFF by using several IFD chained together with the last 4 bytes (or 8 bytes for BigTIFF) of an IFD pointing to the offset of the next one.

The first IFD should have a full description according to the *General description*. Subsequent IFD might have a more compact description, omitting for example, CRS information if it is identical to the main IFD (which should be the case for the currently envisionned use cases), or Copyright / ImageDescription metadata items.

Each IFD will have its NewSubfileType tag set to 0.

If a low-resolution grid is available, it should be put before subgrids of higher-resolution in the chain of IFD linking. On reading, PROJ will use the value from the highest-resolution grid that contains the point of interest.

For efficient reading from the network, files hosted on the CDN will use a layout similar to the one described in the low level paragraph of the Cloud Optimized GeoTIFF GDAL driver page

The layout for a file converted from NTv2 will for example be:

- TIFF/BigTIFF header/signature and pointer to first IFD (Image File Directory)
- "ghost area" indicating the generated process
- IFD of the first grid, followed by TIFF tags values, excluding the TileOffsets and TileByteCounts arrays
- ...
- IFD of the last grid, followed by TIFF tags values, excluding the GDAL_METADATA tag, TileOffsets and TileByteCounts arrays
- TileOffsets and TileByteCounts arrays for first IFD
- ...
- TileOffsets and TileByteCounts arrays for last IFD

- Value of GDAL_METADATA tag for IFDs following the first IFD
- First IFD: Data corresponding to latitude offset of Block_0_0
- First IFD: Data corresponding to longitude offset of Block_0_0
- First IFD: Data corresponding to latitude offset of Block_0_1
- First IFD: Data corresponding to longitude offset of Block_0_1
- ...
- First IFD: Data corresponding to latitude offset of Block_n_m
- First IFD: Data corresponding to longitude offset of Block_n_m
- ...
- Last IFD: Data corresponding to latitude offset of Block_0_0
- Last IFD: Data corresponding to longitude offset of Block_0_0
- Last IFD: Data corresponding to latitude offset of Block_0_1
- Last IFD: Data corresponding to longitude offset of Block_0_1
- ...
- Last IFD: Data corresponding to latitude offset of Block_n_m
- Last IFD: Data corresponding to longitude offset of Block_n_m

If longitude_offset_accuracy and latitude_offset_accuracy are present, this will be followed by:

- First IFD: Data corresponding to latitude offset accuracy of Block_0_0
- First IFD: Data corresponding to longitude offset accuracy of Block_0_0
- ...
- First IFD: Data corresponding to latitude offset accuracy of Block_n_m
- First IFD: Data corresponding to longitude offset accuracy of Block_n_m
- ...
- Last IFD: Data corresponding to latitude offset accuracy of Block_0_0
- Last IFD: Data corresponding to longitude offset accuracy of Block_0_0
- ...
- Last IFD: Data corresponding to latitude offset accuracy of Block_n_m
- Last IFD: Data corresponding to longitude offset accuracy of Block_n_m

**Note**

TIFF has another mechanism to link IFDs, the SubIFD tag. This potentially enables to define a hierarchy of IFDs (similar to HDF5 groups). There is no support for that in most TIFF-using software, notably GDAL, and no compelling need to have a nested hierarchy, so "flat" organization with the standard IFD chaining mechanism is adopted.

### 11.2.5 Examples of multi-grid dataset

https://github.com/OSGeo/PROJ-data/blob/master/au_icsm/au_icsm_GDA94_GDA2020_conformal.tif has been converted from https://github.com/OSGeo/proj-datumgrid/blob/master/oceania/GDA94_GDA2020_conformal.gsb with https://github.com/OSGeo/PROJ-data/blob/master/grid_tools/ntv2_to_gtiff.py

It contains 5 subgrids. All essential metadata to list the subgrids and their georeferencing is contained within the first 3 KB of the file.

The file size is 4.8 MB using DEFLATE compression and floating-point predictor. It results from a lossless conversion from the 83 MB of the original .gsb file.

https://github.com/OSGeo/PROJ-data/blob/master/ca_nrc/ca_nrc_ntv2_0.tif has been converted from https://github.com/OSGeo/proj-datumgrid/blob/master/north-america/ntv2_0.gsb

It contains 114 subgrids. All essential metadata to list the subgrids and their georeferencing is contained within the first 40 KB of the file.

### 11.2.6 Revisions

- **v0.3: addition of TYPE=GEOGRAPHIC_3D_OFFSET, ELLIPSOIDAL_HEIGHT_OFFSET and** interpolation_method (PROJ 9.2)
- v0.2: addition of "arc-seconds per year" as a valid unit (PROJ 9.1.1)
- v0.1: initial version for PROJ 7.0

CHAPTER

# TWELVE

# COMMUNITY

The PROJ community is what makes the software stand out from its competitors. PROJ is used and developed by group of very enthusiastic, knowledgeable and friendly people. Whether you are a first time user of PROJ or a long-time contributor the community is always very welcoming.

## 12.1 Communication channels

### 12.1.1 Mailing list

Users and developers of the library are using the mailing list to discuss all things related to PROJ. The mailing list is the primary forum for asking for help with use of PROJ. The mailing list is also used for announcements, discussions about the development of the library and from time to time interesting discussions on geodesy appear as well. You are more than welcome to join in on the discussions!

The PROJ mailing list can be found at https://lists.osgeo.org/mailman/listinfo/proj

### 12.1.2 GitHub

GitHub is the development platform we use for collaborating on the PROJ code. We use GitHub to keep track of the changes in the code and to index bug reports and feature requests. We are happy to take contributions in any form, either as code, bug reports, documentation or feature requests. See *Contributing* for more info on how you can help improve PROJ.

The PROJ GitHub page can be found at https://github.com/OSGeo/PROJ

**Note**

The issue tracker on GitHub is only meant to keep track of bugs, feature request and other things related to the development of PROJ. Please ask your questions about the use of PROJ on the mailing list instead.

### 12.1.3 Gitter

Gitter is the instant messaging alternative to the mailing list. PROJ has a room under the OSGeo organization. Most of the core developers stop by from time to time for an informal chat. You are more than welcome to join the discussion.

The Gitter room can be found at https://gitter.im/OSGeo/proj.4

## 12.2 Contributing

PROJ has a wide and varied user base. Some are highly skilled geodesists with a deep knowledge of map projections and reference systems, some are GIS software developers and others are GIS users. All users, regardless of the profession or skill level, has the ability to contribute to PROJ. Here's a few suggestion on how:

- Help PROJ-users that is less experienced than yourself.
- Write a bug report
- Request a new feature
- Write documentation for your favorite map projection
- Fix a bug
- Implement a new feature

In the following sections you can find some guidelines on how to contribute. As PROJ is managed on GitHub most contributions require that you have a GitHub account. Familiarity with issues and the GitHub Flow is an advantage.

### 12.2.1 Help a fellow PROJ user

The main forum for support for PROJ is the mailing list. You can subscribe to the mailing list here and read the archive here.

If you have questions about the usage of PROJ the mailing list is also the place to go. Please *do not* use the GitHub issue tracker as a support forum. Your question is much more likely to be answered on the mailing list, as many more people follow that than the issue tracker.

### 12.2.2 Adding bug reports

Bug reports are handled in the issue tracker on PROJ's home on GitHub. Writing a good bug report is not easy. But fixing a poorly documented bug is not easy either, so please put in the effort it takes to create a thorough bug report.

A good bug report includes at least:

- A title that quickly explains the problem
- A description of the problem and how it can be reproduced
- Version of PROJ being used
- Version numbers of any other relevant software being used, e.g. operating system
- A description of what already has been done to solve the problem

The more information that is given up front, the more likely it is that a developer will find interest in solving the problem. You will probably get follow-up questions after submitting a bug report. Please answer them in a timely manner if you have an interest in getting the issue solved.

Finally, please only submit bug reports that are actually related to PROJ. If the issue materializes in software that uses PROJ it is likely a problem with that particular software. Make sure that it actually is a PROJ problem before you submit an issue. If you can reproduce the problem only by using tools from PROJ it is definitely a problem with PROJ.

### 12.2.3 Feature requests

Got an idea for a new feature in PROJ? Submit a thorough description of the new feature in the issue tracker. Please include any technical documents that can help the developer make the new feature a reality. An example of this could be a publicly available academic paper that describes a new projection. Also, including a numerical test case will make it much easier to verify that an implementation of your requested feature actually works as you expect.

Note that not all feature requests are accepted.

### 12.2.4 Write documentation

PROJ is in dire need of better documentation. Any contributions of documentation are greatly appreciated. The PROJ documentation is available on proj.org. The website is generated with Sphinx. Contributions to the documentation should be made as Pull Requests on GitHub.

If you intend to document one of PROJ's supported projections please use the *Mercator projection* as a template.

### 12.2.5 Code contributions

See *Code contributions*

#### 12.2.5.1 Legalese

Committers are the front line gatekeepers to keep the code base clear of improperly contributed code. It is important to the PROJ users, developers and the OSGeo foundation to avoid contributing any code to the project without it being clearly licensed under the project license.

Generally speaking the key issues are that those providing code to be included in the repository understand that the code will be released under the MIT/X license, and that the person providing the code has the right to contribute the code. For the committer themselves understanding about the license is hopefully clear. For other contributors, the committer should verify the understanding unless the committer is very comfortable that the contributor understands the license (for instance frequent contributors).

If the contribution was developed on behalf of an employer (on work time, as part of a work project, etc) then it is important that an appropriate representative of the employer understand that the code will be contributed under the MIT/X license. The arrangement should be cleared with an authorized supervisor/manager, etc.

The code should be developed by the contributor, or the code should be from a source which can be rightfully contributed such as from the public domain, or from an open source project under a compatible license.

All unusual situations need to be discussed and/or documented.

Committers should adhere to the following guidelines, and may be personally legally liable for improperly contributing code to the source repository:

- Make sure the contributor (and possibly employer) is aware of the contribution terms.
- Code coming from a source other than the contributor (such as adapted from another project) should be clearly marked as to the original source, copyright holders, license terms and so forth. This information can be in the file headers, but should also be added to the project licensing file if not exactly matching normal project licensing (COPYING).
- Existing copyright headers and license text should never be stripped from a file. If a copyright holder wishes to give up copyright they must do so in writing to the foundation before copyright messages are removed. If license terms are changed it has to be by agreement (written in email is ok) of the copyright holders.
- Code with licenses requiring credit, or disclosure to users should be added to COPYING.
- When substantial contributions are added to a file (such as substantial patches) the author/contributor should be added to the list of copyright holders for the file.
- If there is uncertainty about whether a change is proper to contribute to the code base, please seek more information from the project steering committee, or the foundation legal counsel.

### 12.2.6 Additional Resources

- General GitHub documentation
- GitHub pull request documentation

### 12.2.7 Acknowledgements

The *code contribution* section of this CONTRIBUTING file is inspired by PDAL's and the *legalese* section is modified from GDAL committer guidelines

## 12.3 Guidelines for PROJ code contributors

This is a guide for PROJ, casual or regular, code contributors.

### 12.3.1 Code contributions.

Code contributions can be either bug fixes or new features. The process is the same for both, so they will be discussed together in this section.

#### 12.3.1.1 Making Changes

- Create a topic branch from where you want to base your work.
- You usually should base your topic branch off of the master branch.
- To quickly create a topic branch: `git checkout -b my-topic-branch`
- Make commits of logical units.
- Check for unnecessary whitespace with `git diff --check` before committing.
- Make sure your commit messages are in the proper format.
- Make sure you have added the necessary tests for your changes.
- Make sure that all tests pass

#### 12.3.1.2 Submitting Changes

- Push your changes to a topic branch in your fork of the repository.
- Submit a pull request to the PROJ repository in the OSGeo organization.
- If your pull request fixes/references an issue, include that issue number in the pull request. For example:

```
Wiz the bang

Fixes #123.
```

- PROJ developers will look at your patch and take an appropriate action.

#### 12.3.1.3 Coding conventions

##### Programming language

PROJ was originally developed in ANSI C. Today PROJ is mostly developed in C++11, with a few parts of the code base still being C. Most of the older parts of the code base is effectively C with a few modifications so that it compiles better as C++.

##### Coding style

The parts of the code base that has started its life as C++ is formatted with `clang-format` using the script `scripts/reformat_cpp.sh`. This is mostly contained to the code in *src/iso19111/* but a few other *.cpp*-files are covered as well.

For the rest of the code base, which has its origin in C, we don't enforce any particular coding style, but please try to keep it as simple as possible. If improving existing code, please try to conform with the style of the locally surrounding code.

#### Whitespace

Throughout the PROJ code base you will see differing whitespace use. The general rule is to keep whitespace in whatever form it is in the file you are currently editing. If the file has a mix of tabs and space please convert the tabs to space in a separate commit before making any other changes. This makes it a lot easier to see the changes in diffs when evaluating the changed code. New files should use spaces as whitespace.

### 12.3.2 Tools

#### 12.3.2.1 Reformatting C++ code

The script in `scripts/reformat_cpp.sh` will reformat C++ code in accordance to the project preference.

If you are writing a new `.cpp`-file it should be added to the list in the reformatting script.

#### 12.3.2.2 cppcheck static analyzer

You can run locally `scripts/cppcheck.sh` that is a wrapper script around the cppcheck utility. This tool is used as part of the quality control of the code.

cppcheck can have false positives. In general, it is preferable to rework the code a bit to make it more 'obvious' and avoid those false positives. When not possible, you can add a comment in the code like

```
/* cppcheck-suppress duplicateBreak */
```

in the preceding line. Replace duplicateBreak with the actual name of the violated rule emitted by cppcheck.

#### 12.3.2.3 Clang Static Analyzer (CSA)

CSA is run by a GitHub Actions workflow. You may also run it locally.

Preliminary step: install clang tools. For example, on a Debian-like OS:

```
sudo apt install clang-tools libfindbin-libs-perl
```

Configure PROJ with the scan-build utility of clang:

```
mkdir csa_build
cd csa_build
scan-build cmake ..
```

Build using `scan-build`:

```
scan-build make [-j8]
```

If CSA finds errors, they will be emitted during the build. And in which case, at the end of the build process, `scan-build` will emit a warning message indicating errors have been found and how to display the error report. This is with something like

```
scan-view /tmp/scan-build-2021-03-15-121416-17476-1
```

This will open a web browser with the interactive report.

CSA may also have false positives. In general, this happens when the code is non-trivial / makes assumptions that hard to check at first sight. You will need to add extra checks or rework it a bit to make it more "obvious" for CSA. This will also help humans reading your code !

#### 12.3.2.4 Typo detection and fixes

Run `scripts/fix_typos.sh`

#### 12.3.2.5 Include What You Use (IWYU)

Managing C includes is a pain. IWYU makes updating headers a bit easier. IWYU scans the code for functions that are called and makes sure that the headers for all those functions are present and in sorted order. However, you cannot blindly apply IWYU to PROJ. It does not understand ifdefs, other platforms, or the order requirements of PROJ internal headers. So the way to use it is to run it on a copy of the source and merge in only the changes that make sense. Additions of standard headers should always be safe to merge. The rest require careful evaluation. See the IWYU documentation for motivation and details.

IWYU docs

## 12.4 Code of Conduct

The PROJ project has adopted the Contributor Covenant Code of Conduct. Everyone who participates in the PROJ community is expected to follow the code of conduct as written below.

### 12.4.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to make participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

### 12.4.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

### 12.4.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

### 12.4.4 Scope

This Code of Conduct applies within all project spaces, and it also applies when an individual is representing the project or its community in public spaces. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

### 12.4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at kristianevers@gmail.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

### 12.4.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

## 12.5 PROJ Service Providers

PROJ is developed and supported by a rich ecosystem of businesses and individuals around the world.

In this page we provide a list of service providers who can assist you in getting the best out of your PROJ investment. Their services can range from training and technical support to help you get started, all the way to specialized development and support services to advance the software and support your organization's mission-critical applications.

**Note**

This list is non-exhaustive, and there are quite likely other service providers in your region of the world who are not listed here. If you think your organization should be listed here, please submit a pull request to add it to this page.

**Note**

We take responsibility for the accuracy, reliability or completeness of the provided information for Core Contributors only.

### 12.5.1 Core Contributors

Core Contributor organizations have PROJ Committers and/or PSC members as part of their team and as such they are the closest to the core development of the software and are usually the best to go to for support.

They are close enough to the architecture to know what is possible, and to ensure that any improvement to be made to PROJ gets done in the right way and will be included in future releases of the software.

They wrote the software that you use, so you can trust them to help you get the most out of it.

Spatialys (France) is a company founded by Even Rouault that is dedicated to providing to its clients the best of its expertise around Open Source geospatial software - such as GDAL/OGR, PROJ, MapServer, and QGIS - and Open standards.

### 12.5.2 Contributors

The Contributors listed here usually have power users and experienced integrators as part of their team who are recognized in the PROJ community for their contributions to various aspects of the software over the years and are very well positioned to assist you with PROJ.

None listed yet.

### 12.5.3 Other Service Providers

The service providers in this category offer services around PROJ and related Open Source technologies.

None listed yet.

**See also**

Search for PROJ service providers with the OSGeo Service Providers

### 12.5.4 How to Add a Company

The steps to add a company to the service providers list are outlined in the GDAL RFC 79 document

## 12.6 Request for Comments

A PROJ RFC describes a major change in the technological underpinnings of PROJ, major additions to functionality, or changes in the direction of the project.

### 12.6.1 PROJ RFC 1: Project Committee Guidelines

**Author**
Frank Warmerdam, Howard Butler

**Contact**
howard@hobu.co

**Status**
Passed

**Last Updated**
2018-06-08

#### 12.6.1.1 Summary

This document describes how the PROJ Project Steering Committee (PSC) determines membership, and makes decisions on all aspects of the PROJ project - both technical and non-technical.

Examples of PSC management responsibilities:

- setting the overall development road map
- developing technical standards and policies (e.g. coding standards, file naming conventions, etc...)
- ensuring regular releases (major and maintenance) of PROJ software
- reviewing RFC for technical enhancements to the software
- project infrastructure (e.g. GitHub, continuous integration hosting options, etc...)
- formalization of affiliation with external entities such as OSGeo
- setting project priorities, especially with respect to project sponsorship
- creation and oversight of specialized sub-committees (e.g. project infrastructure, training)

In brief the project team votes on proposals on the proj mailing list. Proposals are available for review for at least two days, and a single veto is sufficient delay progress though ultimately a majority of members can pass a proposal.

#### 12.6.1.2 List of PSC Members

(up-to-date as of 2023-09)

- Kristian Evers @kbevers (DK) **Chair**
- Howard Butler @hobu (USA)
- Charles Karney @cffk (USA)
- Thomas Knudsen @busstoptaktik (DK)
- Even Rouault @rouault (FR)
- Kurt Schwehr @schwehr (USA)
- Alan Snow @snowman2 (USA)
- Javier Jimenez Shaw @jjimenezshaw (ES)
- Frank Warmerdam @warmerdam (USA) **Emeritus**

#### 12.6.1.3 Detailed Process

- Proposals are written up and submitted on the proj mailing list for discussion and voting, by any interested party, not just committee members.
- Proposals need to be available for review for at least two business days before a final decision can be made.
- Respondents may vote "+1" to indicate support for the proposal and a willingness to support implementation.
- Respondents may vote "-1" to veto a proposal, but must provide clear reasoning and alternate approaches to resolving the problem within the two days.
- A vote of -0 indicates mild disagreement, but has no effect. A 0 indicates no opinion. A +0 indicate mild support, but has no effect.
- Anyone may comment on proposals on the list, but only members of the Project Steering Committee's votes will be counted.
- A proposal will be accepted if it receives +2 (including the author) and no vetoes (-1).
- If a proposal is vetoed, and it cannot be revised to satisfy all parties, then it can be resubmitted for an override vote in which a majority of all eligible voters indicating +1 is sufficient to pass it. Note that this is a majority of all committee members, not just those who actively vote.
- Upon completion of discussion and voting the author should announce whether they are proceeding (proposal accepted) or are withdrawing their proposal (vetoed).
- The Chair gets a vote.
- The Chair is responsible for keeping track of who is a member of the Project Steering Committee (perhaps as part of a PSC file in CVS).
- Addition and removal of members from the committee, as well as selection of a Chair should be handled as a proposal to the committee.
- The Chair adjudicates in cases of disputes about voting.

#### RFC Origin

PROJ RFC and Project Steering Committee is derived from similar governance bodies in both the GDAL and MapServer software projects.

#### 12.6.1.4 When is Vote Required?

- Any change to committee membership (new members, removing inactive members)
- Changes to project infrastructure (e.g. tool, location or substantive configuration)
- Anything that could cause backward compatibility issues.
- Adding substantial amounts of new code.
- Changing inter-subsystem APIs, or objects.
- Issues of procedure.
- When releases should take place.
- Anything dealing with relationships with external entities such as OSGeo
- Anything that might be controversial.

#### 12.6.1.5 Observations

- The Chair is the ultimate adjudicator if things break down.
- The absolute majority rule can be used to override an obstructionist veto, but it is intended that in normal circumstances vetoers need to be convinced to withdraw their veto. We are trying to reach consensus.

#### 12.6.1.6 Committee Membership

The PSC is made up of individuals consisting of technical contributors (e.g. developers) and prominent members of the PROJ user community. There is no set number of members for the PSC although the initial desire is to set the membership at 6.

##### Adding Members

Any member of the proj mailing list may nominate someone for committee membership at any time. Only existing PSC committee members may vote on new members. Nominees must receive a majority vote from existing members to be added to the PSC.

##### Stepping Down

If for any reason a PSC member is not able to fully participate then they certainly are free to step down. If a member is not active (e.g. no voting, no IRC or email participation) for a period of two months then the committee reserves the right to seek nominations to fill that position. Should that person become active again (hey, it happens) then they would certainly be welcome, but would require a nomination.

#### 12.6.1.7 Membership Responsibilities

##### Guiding Development

Members should take an active role guiding the development of new features they feel passionate about. Once a change request has been accepted and given a green light to proceed does not mean the members are free of their obligation. PSC members voting "+1" for a change request are expected to stay engaged and ensure the change is implemented and documented in a way that is most beneficial to users. Note that this applies not only to change requests that affect code, but also those that affect the web site, technical infrastructure, policies and standards.

##### Mailing List Participation

PSC members are expected to be active on the proj mailing list, subject to Open Source mailing list etiquette. Non-developer members of the PSC are not expected to respond to coding level questions on the developer mailing list, however they are expected to provide their thoughts and opinions on user level requirements and compatibility issues when RFC discussions take place.

#### 12.6.1.8 Updates

**June 2018**

RFC 1 was ratified by the following members

### 12.6.2 PROJ RFC 2: Initial integration of "GDAL SRS barn" work

**Author**
Even Rouault

**Contact**
even.rouault at spatialys.com

**Status**
Adopted, implemented in PROJ 6.0

**Initial version**
2018-10-09

**Last Updated**
2018-10-31

#### 12.6.2.1 Summary

This RFC is the result of a first phase of the GDAL Coordinate System Barn Raising efforts. In its current state, this work mostly consists of:

- a C++ implementation of the ISO-19111:2018 / OGC Topic 2 "Referencing by coordinates" classes to represent Datums, Coordinate systems, CRSs (Coordinate Reference Systems) and Coordinate Operations.
- methods to convert between this C++ modeling and WKT1, WKT2 and PROJ string representations of those objects
- management and query of a SQLite3 database of CRS and Coordinate Operation definition
- a C API binding part of those capabilities

#### 12.6.2.2 Related standards

Consult Applicable standards

(They will be linked from the PROJ documentation)

#### 12.6.2.3 Details

##### Structure in packages / namespaces

The C++ implementation of the (upcoming) ISO-19111:2018 / OGC Topic 2 "Referencing by coordinates" classes follows this abstract modeling as much as possible, using package names as C++ namespaces, abstract classes and method names. A new BoundCRS class has been added to cover the modeling of the WKT2 BoundCRS construct, that is a generalization of the WKT1 TOWGS84 concept. It is strongly recommended to have the ISO-19111 standard open to have an introduction for the concepts when looking at the code. A few classes have also been inspired by the GeoAPI

The classes are organized into several namespaces:

- **osgeo::proj::util**
  A set of base types from ISO 19103, GeoAPI and other PROJ "technical" specific classes

  Template optional<T>, classes BaseObject, IComparable, BoxedValue, ArrayOfBaseObject, PropertyMap, LocalName, NameSpace, GenericName, NameFactory, CodeList, Exception, InvalidValueTypeException, UnsupportedOperationException
- **osgeo::proj::metadata:**
  Common classes from ISO 19115 (Metadata) standard

  Classes Citation, GeographicExtent, GeographicBoundingBox, TemporalExtent, VerticalExtent, Extent, Identifier, PositionalAccuracy,
- **osgeo::proj::common:**
  Common classes: UnitOfMeasure, Measure, Scale, Angle, Length, DateTime, DateEpoch, IdentifiedObject, ObjectDomain, ObjectUsage
- **osgeo::proj::cs:**
  Coordinate systems and their axis

Classes AxisDirection, Meridian, CoordinateSystemAxis, CoordinateSystem, SphericalCS, EllipsoidalCS, VerticalCS, CartesianCS, OrdinalCS, ParametricCS, TemporalCS, DateTimeTemporalCS, TemporalCountCS, TemporalMeasureCS

- **osgeo::proj::datum:**
  Datum (the relationship of a coordinate system to the body)

  Classes Ellipsoid, PrimeMeridian, Datum, DatumEnsemble, GeodeticReferenceFrame, DynamicGeodeticReferenceFrame, VerticalReferenceFrame, DynamicVerticalReferenceFrame, TemporalDatum, EngineeringDatum, ParametricDatum

- **osgeo::proj::crs:**
  CRS = coordinate reference system = coordinate system with a datum

  Classes CRS, GeodeticCRS, GeographicCRS, DerivedCRS, ProjectedCRS, VerticalCRS, CompoundCRS, BoundCRS, TemporalCRS, EngineeringCRS, ParametricCRS, DerivedGeodeticCRS, DerivedGeographicCRS, DerivedProjectedCRS, DerivedVerticalCRS

- **osgeo::proj::operation:**
  Coordinate operations (relationship between any two coordinate reference systems)

  Classes CoordinateOperation, GeneralOperationParameter, OperationParameter, GeneralParameterValue, ParameterValue, OperationParameterValue, OperationMethod, InvalidOperation, SingleOperation, Conversion, Transformation, PointMotionOperation, ConcatenatedOperation

- **osgeo::proj::io:**
  I/O classes: WKTFormatter, PROJStringFormatter, FormattingException, ParsingException, IWKTExportable, IPROJStringExportable, WKTNode, WKTParser, PROJStringParser, DatabaseContext, AuthorityFactory, FactoryException, NoSuchAuthorityCodeException

### What does what?

The code to parse WKT and PROJ strings and build ISO-19111 objects is contained in io.cpp

The code to format WKT and PROJ strings from ISO-19111 objects is mostly contained in the related exportToWKT() and exportToPROJString() methods overridden in the applicable classes. io.cpp contains the general mechanics to build such strings.

Regarding WKT strings, three variants are handled in import and export:

- WKT2_2018: variant corresponding to the upcoming ISO-19162:2018 standard
- WKT2_2015: variant corresponding to the current ISO-19162:2015 standard
- WKT1_GDAL: variant corresponding to the way GDAL understands the OGC 01-099 and OGC 99-049 standards

Regarding PROJ strings, two variants are handled in import and export:

- PROJ5: variant used by PROJ >= 5, possibly using pipeline constructs, and avoiding +towgs84 / +nadgrids legacy constructs. This variant honours axis order and input/output units. That is the pipeline for the conversion of EPSG:4326 to EPSG:32631 will assume that the input coordinates are in latitude, longitude order, with degrees.
- PROJ4: variant used by PROJ 4.x

The raw query of the proj.db database and the upper level construction of ISO-19111 objects from the database contents is done in factory.cpp

### A few design principles

Methods generally take and return xxxNNPtr objects, that is non-null shared pointers (pointers with internal reference counting). The advantage of this approach is that the user has not to care about the life-cycle of the instances (and this makes the code leak-free by design). The only point of attention is to make sure no reference cycles are made. This is the case for all classes, except the CoordinateOperation class that point to CRS for sourceCRS and targetCRS members, whereas DerivedCRS point to a Conversion instance (which derives from CoordinateOperation). This issue was detected in the ISO-19111 standard. The solution adopted here is to use std::weak_ptr in the CoordinateOperation class to avoid the cycle. This design artifact is transparent to users.

Another important design point is that all ISO19111 objects are immutable after creation, that is they only have getters that do not modify their states. Consequently they could possibly use in a thread-safe way. There are however classes like PROJStringFormatter, WKTFormatter, DatabaseContext, AuthorityFactory and CoordinateOperationContext whose instances are mutable and thus can not be used by multiple threads at once.

Example how to build the EPSG:4326 / WGS84 Geographic2D definition from scratch:

```
auto greenwich = PrimeMeridian::create(
    util::PropertyMap()
        .set(metadata::Identifier::CODESPACE_KEY,
            metadata::Identifier::EPSG)
        .set(metadata::Identifier::CODE_KEY, 8901)
        .set(common::IdentifiedObject::NAME_KEY, "Greenwich"),
    common::Angle(0));
// actually predefined as PrimeMeridian::GREENWICH constant

auto ellipsoid = Ellipsoid::createFlattenedSphere(
    util::PropertyMap()
        .set(metadata::Identifier::CODESPACE_KEY, metadata::Identifier::EPSG)
        .set(metadata::Identifier::CODE_KEY, 7030)
        .set(common::IdentifiedObject::NAME_KEY, "WGS 84"),
    common::Length(6378137),
    common::Scale(298.257223563));
// actually predefined as Ellipsoid::WGS84 constant

auto datum = GeodeticReferenceFrame::create(
    util::PropertyMap()
        .set(metadata::Identifier::CODESPACE_KEY, metadata::Identifier::EPSG)
        .set(metadata::Identifier::CODE_KEY, 6326)
        .set(common::IdentifiedObject::NAME_KEY, "World Geodetic System 1984");
    ellipsoid
    util::optional<std::string>(), // anchor
    greenwich);
// actually predefined as GeodeticReferenceFrame::EPSG_6326 constant

auto geogCRS = GeographicCRS::create(
    util::PropertyMap()
        .set(metadata::Identifier::CODESPACE_KEY, metadata::Identifier::EPSG)
        .set(metadata::Identifier::CODE_KEY, 4326)
        .set(common::IdentifiedObject::NAME_KEY, "WGS 84"),
    datum,
    cs::EllipsoidalCS::createLatitudeLongitude(scommon::UnitOfMeasure::DEGREE));
// actually predefined as GeographicCRS::EPSG_4326 constant
```

### Algorithmic focus

On the algorithmic side, a somewhat involved logic is the CoordinateOperationFactory::createOperations() in coordinateoperation.cpp that takes a pair of source and target CRS and returns a set of possible coordinate operations (either single operations like a Conversion or a Transformation, or concatenated operations). It uses the intrinsic structure of those objects to create the coordinate operation pipeline. That is, if going from a ProjectedCRS to another one, by doing first the inverse conversion from the source ProjectedCRS to its base GeographicCRS, then finding the appropriate transformation(s) from this base GeographicCRS to the base GeographicCRS of the target CRS, and then applying the conversion from this base GeographicCRS to the target ProjectedCRS. At each step, it queries the database to find if one or several transformations are available. The resulting coordinate operations are filtered, and sorted, with user provided hints:

- desired accuracy
- area of use, defined as a bounding box in longitude, latitude space (its actual CRS does not matter for the intended use)
- if no area of use is defined, if and how the area of use of the source and target CRS should be used. By default, the smallest area of use is used. The rationale is for example when transforming between a national ProjectedCRS and a world-scope GeographicCRS to use the are of use of this ProjectedCRS to select the appropriate datum shifts.
- how the area of use of the candidate transformations and the desired area of use (either explicitly or implicitly defined, as explained above) are compared. By default, only transformations whose area of use is fully contained in the desired area of use are selected. It is also possible to relax this test by specifying that only an intersection test must be used.
- whether PROJ transformation grid names should be substituted to the official names, when a match is found in the *grid_alternatives* table of the database. Defaults to true
- whether the availability of those grids should be used to filter and sort the results. By default, the transformations using grids available in the system will be presented first.

The results are sorted, with the most relevant ones appearing first in the result vector. The criteria used are in that order

- grid actual availability: operations referencing grids not available will be listed after ones with available grids
- grid potential availability: operation referencing grids not known at all in the proj.db will be listed after operations with grids known, but not available.
- known accuracy: operations with unknown accuracies will be listed after operations with known accuracy
- area of use: operations with smaller area of use (the intersection of the operation area of used with the desired area of use) will be listed after the ones with larger area of use
- accuracy: operations with lower accuracy will be listed after operations with higher accuracy (caution: lower accuracy actually means a higher numeric value of the accuracy property, since it is a precision in metre)

All those settings can be specified in the CoordinateOperationContext instance passed to createOperations().

An interesting example to understand how those parameters play together is to use *projinfo -s EPSG:4267 -t EPSG:4326* (NAD27 to WGS84 conversions), and see how specifying desired area of use, spatial criterion, grid availability, etc. affects the results.

The following command currently returns 78 results:

```
projinfo -s EPSG:4267 -t EPSG:4326 --summary --spatial-test intersects
```

The createOperations() algorithm also does a kind of "CRS routing". A typical example is if wanting to transform between CRS A and CRS B, but no direct transformation is referenced in proj.db between those. But if there are transformations between A <--> C and B <--> C, then it is possible to build a concatenated operation A --> C --> B. The typical example is when C is WGS84, but the implementation is generic and just finds a common pivot from the

database. An example of finding a non-WGS84 pivot is when searching a transformation between EPSG:4326 and EPSG:6668 (JGD2011 - Japanese Geodetic Datum 2011), which has no direct transformation registered in the EPSG database . However there are transformations between those two CRS and JGD2000 (and also Tokyo datum, but that one involves less accurate transformations)

```
projinfo -s EPSG:4326 -t EPSG:6668  --grid-check none --bbox 135.42,34.84,142.14,41.58 --
↪summary

Candidate operations found: 7
unknown id, Inverse of JGD2000 to WGS 84 (1) + JGD2000 to JGD2011 (1), 1.2 m, Japan -␣
↪northern Honshu
unknown id, Inverse of JGD2000 to WGS 84 (1) + JGD2000 to JGD2011 (2), 2 m, Japan␣
↪excluding northern main province
unknown id, Inverse of Tokyo to WGS 84 (108) + Tokyo to JGD2011 (2), 9.2 m, Japan␣
↪onshore excluding northern main province
unknown id, Inverse of Tokyo to WGS 84 (108) + Tokyo to JGD2000 (2) + JGD2000 to JGD2011␣
↪(1), 9.4 m, Japan - northern Honshu
unknown id, Inverse of Tokyo to WGS 84 (2) + Tokyo to JGD2011 (2), 13.2 m, Japan -␣
↪onshore mainland and adjacent islands
unknown id, Inverse of Tokyo to WGS 84 (2) + Tokyo to JGD2000 (2) + JGD2000 to JGD2011␣
↪(1), 13.4 m, Japan - northern Honshu
unknown id, Inverse of Tokyo to WGS 84 (1) + Tokyo to JGD2011 (2), 29.2 m, Asia - Japan␣
↪and South Korea
```

#### 12.6.2.4 Code repository

The current state of the work can be found in the iso19111 branch of rouault/proj.4 repository , and is also available as a GitHub pull request at https://github.com/OSGeo/proj.4/pull/1040

Here is a not-so-usable comparison with a fixed snapshot of master branch

#### 12.6.2.5 Database

##### Content

The database contains CRS and coordinate operation definitions from the EPSG database (IOGP's EPSG Geodetic Parameter Dataset) v9.5.3, IGNF registry (French National Geographic Institute), ESRI database, as well as a few customizations.

##### Building (for PROJ developers creating the database)

The building of the database is a several stage process:

##### Construct SQL scripts for EPSG

The first stage consists in constructing .sql scripts mostly with CREATE TABLE and INSERT statements to create the database structure and populate it. There is one .sql file for each database table, populated with the content of the EPSG database, automatically generated with the build_db.py script, which processes the PostgreSQL dumps issued by IOGP. A number of other scripts are dedicated to manual editing, for example grid_alternatives.sql file that binds official grid names to PROJ grid names

### Concert UTF8 SQL to sqlite3 db

The second stage is done automatically by the make process. It pipes the .sql script, in the right order, to the sqlite3 binary to generate a first version of the proj.db SQLite3 database.

### Add extra registries

The third stage consists in creating additional .sql files from the content of other registries. For that process, we need to bind some definitions of those registries to those of the EPSG database, to be able to link to existing objects and detect some boring duplicates. The ignf.sql file has been generated using the build_db_create_ignf.py script from the current data/IGNF file that contains CRS definitions (and implicit transformations to WGS84) as PROJ.4 strings. The esri.sql file has been generated using the build_db_from_esri.py script, from the .csv files in https://github.com/Esri/projection-engine-db-doc/tree/master/csv

### Finalize proj.db

The last stage runs make again to incorporate the new .sql files generated in the previous stage (so the process of building the database involves a kind of bootstrapping...)

### Building (for PROJ users)

The make process just runs the second stage mentioned above from the .sql files. The resulting proj.db is currently 5.3 MB large.

### Structure

The database is structured into the following tables and views. They generally match a ISO-19111 concept, and is generally close to the general structure of the EPSG database. Regarding identification of objects, where the EPSG database only contains a 'code' numeric column, the PROJ database identifies objects with a (auth_name, code) tuple of string values, allowing several registries to be combined together.

- **Technical:**
  - *authority_list*: view enumerating the authorities present in the database. Currently: EPSG, IGNF, PROJ
  - *metadata*: a few key/value pairs, for example to indicate the version of the registries imported in the database
  - *object_view*: synthetic view listing objects (ellipsoids, datums, CRS, coordinate operations...) code and name, and the table name where they are further described
  - *alias_names*: list possible alias for the *name* field of object table
  - *link_from_deprecated_to_non_deprecated*: to handle the link between old ESRI to new ESRI/EPSG codes
- **Common:**
  - *unit_of_measure*: table with UnitOfMeasure definitions.
  - *area*: table with area-of-use (bounding boxes) applicable to CRS and coordinate operations.
- **Coordinate systems:**
  - *axis*: table with CoordinateSystemAxis definitions.
  - *coordinate_system*: table with CoordinateSystem definitions.
- **Ellipsoid and datums:**

  - *ellipsoid*: table with ellipsoid definitions.
  - *prime_meridian*: table with PrimeMeridian definitions.
  - *geodetic_datum*: table with GeodeticReferenceFrame definitions.
  - *vertical_datum*: table with VerticalReferenceFrame definitions.
- **CRS:**
  - *geodetic_crs*: table with GeodeticCRS and GeographicCRS definitions.
  - *projected_crs*: table with ProjectedCRS definitions.
  - *vertical_crs*: table with VerticalCRS definitions.
  - *compound_crs*: table with CompoundCRS definitions.
- **Coordinate operations:**
  - *coordinate_operation_view*: view giving a number of common attributes shared by the concrete tables implementing CoordinateOperation
  - *conversion*: table with definitions of Conversion (mostly parameter and values of Projection)
  - *concatenated_operation*: table with definitions of ConcatenatedOperation.
  - *grid_transformation*: table with all grid-based transformations.
  - *grid_packages*: table listing packages in which grids can be found. ie "proj-datumgrid", "proj-datumgrid-europe", ...
  - *grid_alternatives*: table binding official grid names to PROJ grid names. e.g "Und_min2.5x2.5_egm2008_isw=82_WGS84_TideFree.gz" --> "egm08_25.gtx"
  - *helmert_transformation*: table with all Helmert-based transformations.
  - *other_transformation*: table with other type of transformations.

The main departure with the structure of the EPSG database is the split of the various coordinate operations over several tables. This was done mostly for human-readability as the EPSG organization of coordoperation, coordoperationmethod, coordoperationparam, coordoperationparamusage, coordoperationparamvalue tables makes it hard to grasp at once all the parameters and values for a given operation.

#### 12.6.2.6 Utilities

A new *projinfo* utility has been added. It enables the user to enter a CRS or coordinate operation by a AUTHORITY:CODE, PROJ string or WKT string, and see it translated in the different flavors of PROJ and WKT strings. It also enables to build coordinate operations between two CRSs.

##### Usage

```
usage: projinfo [-o formats] [-k crs|operation] [--summary] [-q]
                [--bbox min_long,min_lat,max_long,max_lat]
                [--spatial-test contains|intersects]
                [--crs-extent-use none|both|intersection|smallest]
                [--grid-check none|discard_missing|sort]
                [--boundcrs-to-wgs84]
                {object_definition} | (-s {srs_def} -t {srs_def})

-o: formats is a comma separated combination of: all,default,PROJ4,PROJ,WKT_ALL,WKT2_
↪2015,WKT2_2018,WKT1_GDAL
    Except 'all' and 'default', other format can be preceded by '-' to disable them
```

### Examples

#### Specify CRS by AUTHORITY:CODE

```
$ projinfo EPSG:4326

PROJ string:
+proj=pipeline +step +proj=longlat +ellps=WGS84 +step +proj=unitconvert +xy_in=rad +xy_
↪out=deg +step +proj=axisswap +order=2,1

WKT2_2015 string:
GEODCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    AREA["World"],
    BBOX[-90,-180,90,180],
    ID["EPSG",4326]]
```

#### Specify CRS by PROJ string and specify output formats

```
$ projinfo -o PROJ4,PROJ,WKT1_GDAL,WKT2_2018 "+title=IGN 1972 Nuku Hiva - UTM fuseau 7␣
↪Sud +proj=tmerc +towgs84=165.7320,216.7200,180.5050,-0.6434,-0.4512,-0.0791,7.420400␣
↪+a=6378388.0000 +rf=297.0000000000000 +lat_0=0.000000000 +lon_0=-141.000000000 +k_0=0.
↪99960000 +x_0=500000.000 +y_0=10000000.000 +units=m +no_defs"

PROJ string:
Error when exporting to PROJ string: BoundCRS cannot be exported as a PROJ.5 string, but␣
↪its baseCRS might

PROJ.4 string:
+proj=utm +zone=7 +south +ellps=intl +towgs84=165.732,216.72,180.505,-0.6434,-0.4512,-0.
↪0791,7.4204

WKT2_2018 string:
BOUNDCRS[
    SOURCECRS[
        PROJCRS["IGN 1972 Nuku Hiva - UTM fuseau 7 Sud",
            BASEGEOGCRS["unknown",
                DATUM["unknown",
                    ELLIPSOID["International 1909 (Hayford)",6378388,297,
                        LENGTHUNIT["metre",1,
                            ID["EPSG",9001]]]],
```

(continues on next page)

(continued from previous page)

```
            PRIMEM["Greenwich",0,
                ANGLEUNIT["degree",0.0174532925199433],
                ID["EPSG",8901]]],
        CONVERSION["unknown",
            METHOD["Transverse Mercator",
                ID["EPSG",9807]],
            PARAMETER["Latitude of natural origin",0,
                ANGLEUNIT["degree",0.0174532925199433],
                ID["EPSG",8801]],
            PARAMETER["Longitude of natural origin",-141,
                ANGLEUNIT["degree",0.0174532925199433],
                ID["EPSG",8802]],
            PARAMETER["Scale factor at natural origin",0.9996,
                SCALEUNIT["unity",1],
                ID["EPSG",8805]],
            PARAMETER["False easting",500000,
                LENGTHUNIT["metre",1],
                ID["EPSG",8806]],
            PARAMETER["False northing",10000000,
                LENGTHUNIT["metre",1],
                ID["EPSG",8807]]],
        CS[Cartesian,2],
            AXIS["(E)",east,
                ORDER[1],
                LENGTHUNIT["metre",1,
                    ID["EPSG",9001]]],
            AXIS["(N)",north,
                ORDER[2],
                LENGTHUNIT["metre",1,
                    ID["EPSG",9001]]]]],
TARGETCRS[
    GEOGCRS["WGS 84",
        DATUM["World Geodetic System 1984",
            ELLIPSOID["WGS 84",6378137,298.257223563,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        CS[ellipsoidal,2],
            AXIS["latitude",north,
                ORDER[1],
                ANGLEUNIT["degree",0.0174532925199433]],
            AXIS["longitude",east,
                ORDER[2],
                ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4326]]],
ABRIDGEDTRANSFORMATION["Transformation from unknown to WGS84",
    METHOD["Position Vector transformation (geog2D domain)",
        ID["EPSG",9606]],
    PARAMETER["X-axis translation",165.732,
        ID["EPSG",8605]],
    PARAMETER["Y-axis translation",216.72,
        ID["EPSG",8606]],
```

(continues on next page)

(continued from previous page)

```
        PARAMETER["Z-axis translation",180.505,
            ID["EPSG",8607]],
        PARAMETER["X-axis rotation",-0.6434,
            ID["EPSG",8608]],
        PARAMETER["Y-axis rotation",-0.4512,
            ID["EPSG",8609]],
        PARAMETER["Z-axis rotation",-0.0791,
            ID["EPSG",8610]],
        PARAMETER["Scale difference",1.0000074204,
            ID["EPSG",8611]]]]

WKT1_GDAL:
PROJCS["IGN 1972 Nuku Hiva - UTM fuseau 7 Sud",
    GEOGCS["unknown",
        DATUM["unknown",
            SPHEROID["International 1909 (Hayford)",6378388,297],
            TOWGS84[165.732,216.72,180.505,-0.6434,-0.4512,-0.0791,7.4204]],
        PRIMEM["Greenwich",0,
            AUTHORITY["EPSG","8901"]],
        UNIT["degree",0.0174532925199433,
            AUTHORITY["EPSG","9122"]],
        AXIS["Longitude",EAST],
        AXIS["Latitude",NORTH]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",-141],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",500000],
    PARAMETER["false_northing",10000000],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AXIS["Easting",EAST],
    AXIS["Northing",NORTH]]
```

### Find transformations between 2 CRS

Between "Poland zone I" (based on Pulkovo 42 datum) and "UTM WGS84 zone 34N"

Summary view:

```
$ projinfo -s EPSG:2171 -t EPSG:32634 --summary

Candidate operations found: 1
unknown id, Inverse of Poland zone I + Pulkovo 1942(58) to WGS 84 (1) + UTM zone 34N, 1␣
↪m, Poland - onshore
```

Display of pipelines:

```
$ PROJ_LIB=data src/projinfo -s EPSG:2171 -t EPSG:32634 -o PROJ

PROJ string:
+proj=pipeline +step +proj=axisswap +order=2,1 +step +inv +proj=sterea +lat_0=50.625␣
```

(continues on next page)

(continued from previous page)

```
↪+lon_0=21.0833333333333 +k=0.9998 +x_0=4637000 +y_0=5647000 +ellps=krass +step␣
↪+proj=cart +ellps=krass +step +proj=helmert +x=33.4 +y=-146.6 +z=-76.3 +rx=-0.359 +ry=-
↪0.053 +rz=0.844 +s=-0.84 +convention=position_vector +step +inv +proj=cart␣
↪+ellps=WGS84 +step +proj=utm +zone=34 +ellps=WGS84
```

#### 12.6.2.7 Impacted files

New files (excluding makefile.am, CMakeLists.txt and other build infrastructure artifacts):

- **include/proj/: Public installed C++ headers**
  - common.hpp: declarations of osgeo::proj::common namespace.
  - coordinateoperation.hpp: declarations of osgeo::proj::operation namespace.
  - coordinatesystem.hpp: declarations of osgeo::proj::cs namespace.
  - crs.hpp: declarations of osgeo::proj::crs namespace.
  - datum.hpp: declarations of osgeo::proj::datum namespace.
  - io.hpp: declarations of osgeo::proj::io namespace.
  - metadata.hpp: declarations of osgeo::proj::metadata namespace.
  - util.hpp: declarations of osgeo::proj::util namespace.
  - nn.hpp: Code from https://github.com/dropbox/nn to manage Non-nullable pointers for C++
- **include/proj/internal: Private non-installed C++ headers**
  - coordinateoperation_internal.hpp: classes InverseCoordinateOperation, InverseConversion, InverseTransformation, PROJBasedOperation, and functions to get conversion mappings between WKT and PROJ syntax
  - coordinateoperation_constants.hpp: Select subset of conversion/transformation EPSG names and codes for the purpose of translating them to PROJ strings
  - coordinatesystem_internal.hpp: classes AxisDirectionWKT1, AxisName and AxisAbbreviation
  - internal.hpp: a few helper functions, mostly to do string-based operations
  - io_internal.hpp: class WKTConstants
  - helmert_constants.hpp: Helmert-based transformation & parameters names and codes.
  - lru_cache.hpp: code from https://github.com/mohaps/lrucache11 to have a generic Least-Recently-Used cache of objects
- **src/:**
  - c_api.cpp: C++ API mapped to C functions
  - common.cpp: implementation of common.hpp
  - coordinateoperation.cpp: implementation of coordinateoperation.hpp
  - coordinatesystem.cpp: implementation of coordinatesystem.hpp
  - crs.cpp: implementation of crs.hpp
  - datum.cpp: implementation of datum.hpp

  - factory.cpp: implementation of AuthorityFactory class (from io.hpp)
  - internal.cpp: implementation of internal.hpp
  - io.cpp: implementation of io.hpp
  - metadata.cpp: implementation of metadata.hpp
  - static.cpp: a number of static constants (like pre-defined well-known ellipsoid, datum and CRS), put in the right order for correct static initializations
  - util.cpp: implementation of util.hpp
  - projinfo.cpp: new 'projinfo' binary
  - general.dox: generic introduction documentation.

- **data/sql/:**
  - area.sql: generated by build_db.py
  - axis.sql: generated by build_db.py
  - begin.sql: hand generated (trivial)
  - commit.sql: hand generated (trivial)
  - compound_crs.sql: generated by build_db.py
  - concatenated_operation.sql: generated by build_db.py
  - conversion.sql: generated by build_db.py
  - coordinate_operation.sql: generated by build_db.py
  - coordinate_system.sql: generated by build_db.py
  - crs.sql: generated by build_db.py
  - customizations.sql: hand generated (empty)
  - ellipsoid.sql: generated by build_db.py
  - geodetic_crs.sql: generated by build_db.py
  - geodetic_datum.sql: generated by build_db.py
  - grid_alternatives.sql: hand-generated. Contains links between official registry grid names and PROJ ones
  - grid_transformation.sql: generated by build_db.py
  - grid_transformation_custom.sql: hand-generated
  - helmert_transformation.sql: generated by build_db.py
  - ignf.sql: generated by build_db_create_ignf.py
  - esri.sql: generated by build_db_from_esri.py
  - metadata.sql: hand-generated
  - other_transformation.sql: generated by build_db.py
  - prime_meridian.sql: generated by build_db.py
  - proj_db_table_defs.sql: hand-generated. Database structure: CREATE TABLE / CREATE VIEW / CREATE TRIGGER
  - projected_crs.sql: generated by build_db.py

  - unit_of_measure.sql: generated by build_db.py
  - vertical_crs.sql: generated by build_db.py
  - vertical_datum.sql: generated by build_db.py
- **scripts/:**
  - build_db.py : generate .sql files from EPSG database dumps
  - build_db_create_ignf.py: generates data/sql/ignf.sql
  - build_db_from_esri.py: generates data/sql/esri.sql
  - doxygen.sh: generates Doxygen documentation
  - gen_html_coverage.sh: generates HTML report of the coverage for --coverage build
  - filter_lcov_info.py: utility used by gen_html_coverage.sh
  - reformat.sh: used by reformat_cpp.sh
  - reformat_cpp.sh: reformat all .cpp/.hpp files according to LLVM-style formatting rules
- **tests/unit/**
  - test_c_api.cpp: test of src/c_api.cpp
  - test_common.cpp: test of src/common.cpp
  - test_util.cpp: test of src/util.cpp
  - test_crs.cpp: test of src/crs.cpp
  - test_datum.cpp: test of src/datum.cpp
  - test_factory.cpp: test of src/factory.cpp
  - test_io.cpp: test of src/io.cpp
  - test_metadata.cpp: test of src/metadata.cpp
  - test_operation.cpp: test of src/operation.cpp

#### 12.6.2.8 C API

proj.h has been extended to bind a number of C++ classes/methods to a C API.

The main structure is an opaque PJ_OBJ* roughly encapsulating a osgeo::proj::BaseObject, that can represent a CRS or a CoordinateOperation object. A number of the C functions will work only if the right type of underlying C++ object is used with them. Misuse will be properly handled at runtime. If a user passes a PJ_OBJ* representing a coordinate operation to a pj_obj_crs_xxxx() function, it will properly error out. This design has been chosen over creating a dedicate PJ_xxx object for each C++ class, because such an approach would require adding many conversion and free functions for little benefit.

This C API is incomplete. In particular, it does not allow to build ISO19111 objects at hand. However it currently permits a number of actions:

- building CRS and coordinate operations from WKT and PROJ strings, or from the proj.db database
- exporting CRS and coordinate operations as WKT and PROJ strings
- querying main attributes of those objects
- finding coordinate operations between two CRS.

test_c_api.cpp should demonstrates simple usage of the API (note: for the conveniency of writing the tests in C++, test_c_api.cpp wraps the C PJ_OBJ* instances in C++ 'keeper' objects that automatically call the pj_obj_unref() function at function end. In a pure C use, the caller must use pj_obj_unref() to prevent leaks.)

#### 12.6.2.9 Documentation

All public C++ classes and methods and C functions are documented with Doxygen.

Current snapshot of Class list

Spaghetti inheritance diagram

A basic integration of the Doxygen XML output into the general PROJ documentation (using reStructuredText format) has been done with the Sphinx extension Breathe, producing:

- One section with the C++ API
- One section with the C API

#### 12.6.2.10 Testing

Nearly all exported methods are tested by a unit test. Global line coverage of the new files is 92%. Those tests represent 16k lines of codes.

#### 12.6.2.11 Build requirements

The new code leverages on a number of C++11 features (auto keyword, constexpr, initializer list, std::shared_ptr, lambda functions, etc.), which means that a C++11-compliant compiler must be used to generate PROJ:

- gcc >= 4.8
- clang >= 3.3
- Visual Studio >= 2015.

Compilers tested by the Travis-CI and AppVeyor continuous integration environments:

- GCC 4.8
- mingw-w64-x86-64 4.8
- clang 5.0
- Apple LLVM version 9.1.0 (clang-902.0.39.2)
- MSVC 2015 32 and 64 bit
- MSVC 2017 32 and 64 bit

The libsqlite3 >= 3.7 development package must also be available. And the sqlite3 binary must be available to build the proj.db files from the .sql files.

#### 12.6.2.12 Runtime requirements

- libc++/libstdc++/MSVC runtime consistent with the compiler used
- libsqlite3 >= 3.7

#### 12.6.2.13 Backward compatibility

At this stage, no backward compatibility issue is foreseen, as no existing functional C code has been modified to use the new capabilities

#### 12.6.2.14 Future work

The work described in this RFC will be pursued in a number of directions. Non-exhaustively:

- Support for ESRI WKT1 dialect (PROJ currently ingest the ProjectedCRS in esri.sql in that dialect, but there is no mapping between it and EPSG operation and parameter names, so conversion to PROJ strings does not always work.
- closer integration with the existing code base. In particular, the +init=dict:code syntax should now go first to the database (then the *epsg* and *IGNF* files can be removed). Similarly proj_create_crs_to_crs() could use the new capabilities to find an appropriate coordinate transformation.
- and whatever else changes are needed to address GDAL and libgeotiff needs

#### 12.6.2.15 Adoption status

The RFC has been adopted with support from PSC members Kurt Schwehr, Kristian Evers, Howard Butler and Even Rouault.

### 12.6.3 PROJ RFC 3: Dependency management

**Author**
Kristian Evers

**Contact**
kreve@sdfe.dk

**Status**
Adopted

**Last Updated**
2025-01-07

#### 12.6.3.1 Summary

This document defines a set of guidelines for dependency management in PROJ. With PROJ being a core component in many downstream software packages clearly stating which dependencies the library has is of great value. This document concern both programming language standards as well as minimum required versions of library dependencies and build tools.

It is proposed to adopt a rolling update scheme that ensures that PROJ is sufficiently accessible, even on older systems, as well as keeping up with the technological evolution. The scheme is divided in two parts, one concerning versions of used programming languages within PROJ and the other concerning software packages that PROJ depend on.

With adoption of this RFC, versions used for

1. programming languages will always be at least two revisions behind the most recent standard
2. software packages will always be at least two years old (patch releases are exempt)

Changes for programming languages or software package dependencies can be introduced with minor version releases of PROJ. Changing the version requirements for a dependency needs to be approved by the PSC. Following a change in dependency version the build requirements needs to be updated and the changes be clearly stated in the release notes for the affected PROJ version.

Following the above rule set will ensure that all but the most conservative users of PROJ will be able to build and use the most recent version of the library.

In the sections below details concerning programming languages and software dependencies are outlined. The RFC is concluded with a bootstrapping section that details the state of dependencies after the accept of the RFC.

#### 12.6.3.2 Background

PROJ has traditionally been written in C89. Until recently, no formal requirements of e.g. build systems has been defined and formally accepted by the project. *RFC2* formally introduces dependencies on C++11 and SQLite 3.7.

In this RFC a rolling update of version or standard requirements is described. The reasoning behind a rolling update scheme is that it has become increasingly evident that C89 is becoming outdated and creating a less than optimal development environment for contributors. It has been noted that the most commonly used compilers all now support more recent versions of C, so the strict usage of C89 is no longer as critical as it used to be.

Similarly, rolling updates to other tools and libraries that PROJ depend on will ensure that the code base can be kept modern and in line with the rest of the open source software ecosphere.

#### 12.6.3.3 C and C++

Following *RFC2* PROJ is written in both C and C++. At the time of writing the core library is C based and the code described in RFC2 is written in C++. While the core library is mostly written in C it is compiled as C++. Minor sections of PROJ, like the geodesic algorithms are still compiled as C since there is no apparent benefit of compiling with a C++ compiler. This may change in the future.

Both the C and C++ standards are updated with regular intervals. After an update of a standard it takes time for compiler manufacturers to implement the standards fully, which makes adaption of new standards potentially troublesome if done too soon. On the other hand, waiting too long to adopt new standards will eventually make the code base feel old and new contributors are more likely to stay away because they don't want to work using tools of the past. With a rolling update scheme both concerns can be managed by always staying behind the most recent standard, but not so far away that potential contributors are scared away. Keeping a policy of always lagging behind be two iterations of the standard is thought to be the best comprise between the two concerns.

C comes in four ISO standardised varieties: C89, C99, C11, C18. In this document we refer to their informal names for ease of reading. C++ exists in five varieties: C++98, C++03, C++11, C++14, C++17. Before adoption of this RFC PROJ uses C89 and C++11. For C, that means that the used standard is three iterations behind the most recent standard. C++ is two iterations behind. Following the rules in this RFC the required C standard used in PROJ is at allowed to be two iterations behind the most recent standard. That means that a change to C99 is possible, as long as the PROJ PSC acknowledges such a change.

When a new standard for either C or C++ is released PROJ should consider changing its requirement to the next standard in the line. For C++ that means a change in standard roughly every three years, for C the periods between standard updates is expected to be longer.

#### 12.6.3.4 Software dependencies

At the time of writing PROJ is dependent on very few external packages. In fact only one runtime dependency is present: SQLite. Building PROJ also requires one of two external dependencies for configuration: Autotools or CMake.

As with programming language standards it is preferable that software dependencies are a bit behind the most recent development. For this reason it is required that the minimum version supported in PROJ dependencies is at least two years old, preferably more. It is not a requirement that the minimum version of dependencies is always kept strictly two years behind current development, but it is allowed in case future development of PROJ warrants an update. Changes in minimum version requirements are allowed to happen with minor version releases of PROJ.

At the time of writing the minimum version required for SQLite it 3.7 which was released in 2010. CMake currently is required to be at least at version 2.8.3 which was also released in 2010.

#### 12.6.3.5 Bootstrapping

This RFC comes with a set of guidelines for handling dependencies for PROJ in the future. Up until now dependencies hasn't been handled consistently, with some dependencies not being approved formally by the projects governing body. Therefore minimum versions of PROJ dependencies is proposed so that at the approval of this RFC PROJ will have the following external requirements:

- C99 (was C89)
- C++11 (already approved in *RFC2*)
- SQLite 3.7 (already approved in *RFC2*)
- CMake 3.5 (was 2.8.3)

#### 12.6.3.6 Adoption status

The RFC was adopted on 2018-01-19 with +1's from the following PSC members

- Kristian Evers
- Even Rouault
- Thomas Knudsen
- Howard Butler

Amendment on January 2025 to allow updating programming language standard in minor versions.

### 12.6.4 PROJ RFC 4: Remote access to grids and GeoTIFF grids

**Author**
Even Rouault, Howard Butler

**Contact**
even.rouault@spatialys.com, howard@hobu.co

**Status**
Adopted

**Implementation target**
PROJ 7

**Last Updated**
2020-01-10

#### 12.6.4.1 Motivation

PROJ 6 brings undeniable advances in the management of coordinate transformations between datums by relying and applying information available in the PROJ database. PROJ's rapid evolution from a cartographic projections library with a little bit of geodetic capability to a full geodetic transformation and description environment has highlighted the importance of the support data. Users desire the convenience of software doing the right thing with the least amount of fuss, and survey organizations wish to deliver their models across as wide a software footprint as possible. To get results with the highest precision, a grid file that defines a model that provides dimension shifts is often needed. The proj-datumgrid project centralizes grids available under an open data license and bundles them in different archives split along major geographical regions of the world .

It is assumed that a PROJ user has downloaded and installed grid files that are referred to in the PROJ database. These files can be quite large in aggregate, and packaging support by major distribution channels is somewhat uneven due to their size, sometimes ambiguous licensing story, and difficult-to-track versioning and lineage. It is not always clear to the user, especially to those who may not be so familiar with geodetic operations, that the highest precision transformation may not always being applied if grid data is not available. Users want both convenience and correctness, and management of the shift files can be challenging to those who may not be aware of their importance to the process.

The computing environment in which PROJ operates is also changing. Because the shift data can be so large (currently more than 700 MB of uncompressed data, and growing), deployment of high accuracy operations can be limited due to deployment size constraints (serverless operations, for example). Changing to a delivery format that supports incremental access over a network along with convenient access and compression will ease the resource burden the shift

files present while allowing the project to deliver transformation capability with the highest known precision provided by the survey organizations.

Adjustment grids also tend to be provided in many different formats depending on the organization and country that produced them. In PROJ, we have over time "standardized" on using horizontal shift grids as NTv2 and vertical shift grids using GTX. Both have poor general support as dedicated formats, limited metadata capabilities, and neither are not necessarily "cloud optimized" for incremental access across a network.

#### 12.6.4.2 Summary of work planned by this RFC

- Grids will be hosted by one or several Content Delivery Networks (CDN)
- Grid loading mechanism will be reworked to be able to download grids or parts of grids from a online repository. When opted in, users will no longer have to manually fetch grid files and place them in PROJ_LIB. Full and accurate capability of the software will no longer require hundreds of megabytes of grid shift files in advance, even if only just a few of them are needed for the transformations done by the user.
- Local caching of grid files, or even part of files, so that users end up mirroring what they actually use.
- A grid shift format, for both horizontal and vertical shift grids (and in potential future steps, for other needs, such as deformation models) will be implemented.

The use of grids locally available will of course still be available, and will be the default behavior.

#### 12.6.4.3 Network access to grids

curl will be an optional build dependency of PROJ, added in autoconf and cmake build systems. It can be disabled at build time, but this must be an explicit setting of configure/cmake as the resulting builds have less functionality. When curl is enabled at build time, download of grids themselves will not be enabled by default at runtime. It will require explicit consent of the user, either through the API (*proj_context_set_enable_network()*) through the PROJ_NETWORK=ON environment variable, or the `network = on` setting of proj.ini.

Regarding the minimum version of libcurl required, given GDAL experience that can build with rather ancient libcurl for similar functionality, we can aim for libcurl >= 7.29.0 (as being available in RHEL 7).

An alternate pluggable network interface can also be set by the user in case support for libcurl was not built in, or if for the desired context of use, the user wishes to provide the network implementation (a typical use case could be QGIS that would use its QT-based networking facilities to solve issues with SSL, proxy, authentication, etc.)

A text configuration file, installed in ${installation_prefix}/share/proj/proj.ini (or ${PROJ_LIB}/proj.ini) will contain the URL of the CDN that will be used. The user may also override this setting with the *proj_context_set_url_endpoint()* or through the PROJ_NETWORK_ENDPOINT environment variable.

The rationale for putting proj.ini in that location is that it is a well-known place by PROJ users, with the existing PROJ_LIB mechanics for systems like Windows where hardcoded paths at runtime aren't generally usable.

#### C API

The preliminary C API for the above is:

```
/** Enable or disable network access.
*
* @param ctx PROJ context, or NULL
* @return TRUE if network access is possible. That is either libcurl is
*         available, or an alternate interface has been set.
*/
int proj_context_set_enable_network(PJ_CONTEXT* ctx, int enable);
```

(continues on next page)

(continued from previous page)

```
/** Define URL endpoint to query for remote grids.
*
* This overrides the default endpoint in the PROJ configuration file or with
* the PROJ_NETWORK_ENDPOINT environment variable.
*
* @param ctx PROJ context, or NULL
* @param url Endpoint URL. Must NOT be NULL.
*/
void proj_context_set_url_endpoint(PJ_CONTEXT* ctx, const char* url);

/** Opaque structure for PROJ. Implementations might cast it to their
 * structure/class of choice. */
typedef struct PROJ_NETWORK_HANDLE PROJ_NETWORK_HANDLE;

/** Network access: open callback
*
* Should try to read the size_to_read first bytes at the specified offset of
* the file given by URL url,
* and write them to buffer. *out_size_read should be updated with the actual
* amount of bytes read (== size_to_read if the file is larger than size_to_read).
* During this read, the implementation should make sure to store the HTTP
* headers from the server response to be able to respond to
* proj_network_get_header_value_cbk_type callback.
*
* error_string_max_size should be the maximum size that can be written into
* the out_error_string buffer (including terminating nul character).
*
* @return a non-NULL opaque handle in case of success.
*/
typedef PROJ_NETWORK_HANDLE* (*proj_network_open_cbk_type)(
                                                PJ_CONTEXT* ctx,
                                                const char* url,
                                                unsigned long long offset,
                                                size_t size_to_read,
                                                void* buffer,
                                                size_t* out_size_read,
                                                size_t error_string_max_size,
                                                char* out_error_string,
                                                void* user_data);

/** Network access: close callback */
typedef void (*proj_network_close_cbk_type)(PJ_CONTEXT* ctx,
                                            PROJ_NETWORK_HANDLE* handle,
                                            void* user_data);

/** Network access: get HTTP headers */
typedef const char* (*proj_network_get_header_value_cbk_type)(
                                            PJ_CONTEXT* ctx,
                                            PROJ_NETWORK_HANDLE* handle,
                                            const char* header_name,
                                            void* user_data);
```

(continues on next page)

(continued from previous page)

```
/** Network access: read range
*
* Read size_to_read bytes from handle, starting at offset, into
* buffer.
* During this read, the implementation should make sure to store the HTTP
* headers from the server response to be able to respond to
* proj_network_get_header_value_cbk_type callback.
*
* error_string_max_size should be the maximum size that can be written into
* the out_error_string buffer (including terminating nul character).
*
* @return the number of bytes actually read (0 in case of error)
*/
typedef size_t (*proj_network_read_range_type)(
                                            PJ_CONTEXT* ctx,
                                            PROJ_NETWORK_HANDLE* handle,
                                            unsigned long long offset,
                                            size_t size_to_read,
                                            void* buffer,
                                            size_t error_string_max_size,
                                            char* out_error_string,
                                            void* user_data);

/** Define a custom set of callbacks for network access.
*
* All callbacks should be provided (non NULL pointers).
*
* @param ctx PROJ context, or NULL
* @param open_cbk Callback to open a remote file given its URL
* @param close_cbk Callback to close a remote file.
* @param get_header_value_cbk Callback to get HTTP headers
* @param read_range_cbk Callback to read a range of bytes inside a remote file.
* @param user_data Arbitrary pointer provided by the user, and passed to the
* above callbacks. May be NULL.
* @return TRUE in case of success.
*/
int proj_context_set_network_callbacks(
    PJ_CONTEXT* ctx,
    proj_network_open_cbk_type open_cbk,
    proj_network_close_cbk_type close_cbk,
    proj_network_get_header_value_cbk_type get_header_value_cbk,
    proj_network_read_range_type read_range_cbk,
    void* user_data);
```

To make network access efficient, PROJ will internally have a in-memory cache of file ranges to only issue network requests by chunks of 16 KB or multiple of them, to limit the number of HTTP GET requests and minimize latency caused by network access. This is very similar to the behavior of the GDAL /vsicurl/ I/O layer. The plan is to mostly copy GDAL's vsicurl implementation inside PROJ, with needed adjustments and proper namespacing of it.

A retry strategy (typically a delay with an exponential back-off and some random jitter) will be added to account for intermittent network or server-side failure.

### URL building

The PROJ database has a `grid_transformation` grid whose column `grid_name` (and possibly `grid2_name`) contain the name of the grid as indicated by the authority having registered the transformation (typically EPSG). As those grid names are not generally directly usable by PROJ, the PROJ database has also a `grid_alternatives` table that link original grid names to the ones used by PROJ. When network access will be available and needed due to lack of a local grid, the full URL will be the endpoint from the configuration or set by the user, the basename of the PROJ usable filename, and the "tif" suffix. So if the CDN is at http://example.com and the name from `grid_alternatives` is egm96_15.gtx, then the URL will be http://example.com/egm96_15.tif

### Grid loading

The following files will be affected, in one way or another, by the above describes changes: nad_cvt.cpp, nad_intr.cpp, nad_init.cpp, grid_info.cpp, grid_list.cpp, apply_gridshift.cpp, apply_vgridshift.cpp.

In particular the current logic that consists to ingest all the values of a grid/subgrid in the ct->cvs array will be completely modified, to enable access to grid values at a specified (x,y) location.

### proj_create_crs_to_crs() / proj_create_operations() impacts

Once network access is available, all grids known to the PROJ database (grid_transformation + grid_alternatives table) will be assumed to be available, when computing the potential pipelines between two CRS.

Concretely, this will be equivalent to calling `proj_operation_factory_context_set_grid_availability_use()` with the `use` argument set to a new enumeration value

```
/** Results will be presented as if grids known to PROJ (that is
* registered in the grid_alternatives table of its database) were
* available. Used typically when networking is enabled.
*/
PROJ_GRID_AVAILABILITY_KNOWN_AVAILABLE
```

### Local on-disk caching of remote grids

As many workflows will tend to use the same grids over and over, a local on-disk caching of remote grids will be added. The cache will be a single SQLite3 database, in a user-writable directory shared by all applications using PROJ.

Its total size will be configurable, with a default maximum size of 100 MB in proj.ini. The cache will also keep the timestamp of the last time it checked various global properties of the file (its size, Last-Modified and ETag headers). A time-to-live parameter, with a default of 1 day in proj.ini, will be used to determine whether the CDN should be hit to verify if the information in the cache is still up-to-date.

```
/** Enable or disable the local cache of grid chunks
*
* This overrides the setting in the PROJ configuration file.
*
* @param ctx PROJ context, or NULL
* @param enabled TRUE if the cache is enabled.
*/
void proj_grid_cache_set_enable(PJ_CONTEXT *ctx, int enabled);

/** Override, for the considered context, the path and file of the local
* cache of grid chunks.
*
* @param ctx PROJ context, or NULL
```

(continues on next page)

(continued from previous page)

```
* @param fullname Full name to the cache (encoded in UTF-8). If set to NULL,
*                 caching will be disabled.
*/
void proj_grid_cache_set_filename(PJ_CONTEXT* ctx, const char* fullname);

/** Override, for the considered context, the maximum size of the local
* cache of grid chunks.
*
* @param ctx PROJ context, or NULL
* @param max_size_MB Maximum size, in mega-bytes (1024*1024 bytes), or
*                    negative value to set unlimited size.
*/
void proj_grid_cache_set_max_size(PJ_CONTEXT* ctx, int max_size_MB);

/** Override, for the considered context, the time-to-live delay for
* re-checking if the cached properties of files are still up-to-date.
*
* @param ctx PROJ context, or NULL
* @param ttl_seconds Delay in seconds. Use negative value for no expiration.
*/
void proj_grid_cache_set_ttl(PJ_CONTEXT* ctx, int ttl_seconds);

/** Clear the local cache of grid chunks.
 *
 * @param ctx PROJ context, or NULL.
 */
void proj_grid_cache_clear(PJ_CONTEXT* ctx);
```

The planned database structure is:

```
-- General properties on a file
CREATE TABLE properties(
 url          TEXT PRIMARY KEY NOT NULL,
 lastChecked  TIMESTAMP NOT NULL,
 fileSize     INTEGER NOT NULL,
 lastModified TEXT,
 etag         TEXT
);

-- Store chunks of data. To avoid any potential fragmentation of the
-- cache, the data BLOB is always set to the maximum chunk size of 16 KB
-- (right padded with 0-byte)
-- The actual size is stored in chunks.data_size
CREATE TABLE chunk_data(
 id        INTEGER PRIMARY KEY AUTOINCREMENT CHECK (id > 0),
 data      BLOB NOT NULL
);

-- Record chunks of data by (url, offset)
CREATE TABLE chunks(
 id        INTEGER PRIMARY KEY AUTOINCREMENT CHECK (id > 0),
 url       TEXT NOT NULL,
```

(continues on next page)

(continued from previous page)

```
 offset    INTEGER NOT NULL,
 data_id   INTEGER NOT NULL,
 data_size INTEGER NOT NULL,
 CONSTRAINT fk_chunks_url FOREIGN KEY (url) REFERENCES properties(url),
 CONSTRAINT fk_chunks_data FOREIGN KEY (data_id) REFERENCES chunk_data(id)
);
CREATE INDEX idx_chunks ON chunks(url, offset);

-- Doubly linked list of chunks. The next link is to go to the least-recently
-- used entries.
CREATE TABLE linked_chunks(
 id        INTEGER PRIMARY KEY AUTOINCREMENT CHECK (id > 0),
 chunk_id  INTEGER NOT NULL,
 prev      INTEGER,
 next      INTEGER,
 CONSTRAINT fk_links_chunkid FOREIGN KEY (chunk_id) REFERENCES chunks(id),
 CONSTRAINT fk_links_prev FOREIGN KEY (prev) REFERENCES linked_chunks(id),
 CONSTRAINT fk_links_next FOREIGN KEY (next) REFERENCES linked_chunks(id)
);
CREATE INDEX idx_linked_chunks_chunk_id ON linked_chunks(chunk_id);

-- Head and tail pointers of the linked_chunks. The head pointer is for
-- the most-recently used chunk.
-- There should be just one row in this table.
CREATE TABLE linked_chunks_head_tail(
  head       INTEGER,
  tail       INTEGER,
  CONSTRAINT lht_head FOREIGN KEY (head) REFERENCES linked_chunks(id),
  CONSTRAINT lht_tail FOREIGN KEY (tail) REFERENCES linked_chunks(id)
);
INSERT INTO linked_chunks_head_tail VALUES (NULL, NULL);
```

The chunks table will store 16 KB chunks (or less for terminating chunks). The linked_chunks and linked_chunks_head_tail table swill act as a doubly linked list of chunks, with the least recently used ones at the end of the list, which will be evicted when the cache saturates.

The directory used to locate this database will be ${XDG_DATA_HOME}/proj (per https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html) where ${XDG_DATA_HOME} defaults to ${HOME}/.local/share on Unix builds and ${LOCALAPPDATA} on Windows builds. Exact details to be sorted out, but https://github.com/ActiveState/appdirs/blob/a54ea98feed0a7593475b94de3a359e9e1fe8fdb/appdirs.py#L45-L97 can be a good reference.

As this database might be accessed by several threads or processes at the same time, the code accessing to it will carefully honour SQLite3 errors regarding to locks, to do appropriate retries if another thread/process is currently locking the database. Accesses requiring a modification of the database will start with a BEGIN IMMEDIATE transaction so as to acquire a write lock.

**Note**

This database should be hosted on a local disk, not a network one. Otherwise SQLite3 locking issues are to be expected.

### CDN provider

Amazon Public Datasets has offered to be a storage and CDN provider.

The program covers storage and egress (bandwidth) of the data. They generally don't allow usage of CloudFront (their CDN) as part of the program (we would usually look to have it covered by credits), but in this instance, they would be fine to provide it. They'd only ask that we keep the CloudFront URL "visible" (as appropriate for the use case) so people can see where the data is hosted in case they go looking. Their terms can be seen at https://aws.amazon.com/service-terms/ and CloudFront has its own, small section. Those terms may change a bit from time to time for minor changes. Major changing service terms is assumed to be unfrequent. There are also the Public Dataset Program terms at http://aws.amazon.com/public-datasets/terms/. Those also do not effectively change over time and are renewed on a 2 year basis.

### Criteria for grid hosting

The grids hosted on the CDN will be exactly the ones collected, currently and in the future, by the proj-datumgrid initiative. In particular, new grids are accepted as long as they are released under a license that is compatible with the Open Source Definition and the source of the grid is clearly stated and verifiable. Suitable licenses include:

- Public domain
- X/MIT
- BSD 2/3/4 clause
- CC0
- CC-BY (v3.0 or later)
- CC-BY-SA (v3.0 or later)

For new grids to be transparently used by the proj_create_crs_to_crs() mechanics, they must be registered in the PROJ database (proj.db) in the `grid_transformation` and `grid_alternatives` table. The nominal path to have a new record in the grid_transformation is to have a transformation being registered in the EPSG dataset (if there is no existing one), which will be subsequently imported into the PROJ database.

### Versioning, historical preservation of grids

The policy regarding this should be similar to the one applied to proj-datumgrid, which even if not formalized, is around the following lines:

- Geodetic agencies release regularly new version of grids. Typically for the USA, NOAA has released GEOID99, GEOID03, GEOID06, GEOID09, GEOID12A, GEOID12B, GEOID18 for the NAVD88 to NAD83/NAD83(2011) vertical adjustments. Each of these grids is considered by EPSG and PROJ has a separate object, with distinct filenames. The release of a new version does not cause the old grid to be automatically removed. That said, due to advertised accuracies and supersession rules of the EPSG dataset, the most recent grid will generally be used for a CRS -> CRS transformation if the user uses proj_create_crs_to_crs() (with the exception that if a VERT_CRS WKT includes a GEOID_MODEL known to PROJ, an old version of the grid will be used). If the user specifies a whole pipeline with an explicit grid name, it will be of course strictly honoured. As time goes, the size of the datasets managed by proj-datumgrid will be increasing, we will have to explore on we managed that for the distributed .zip / .tar.gz archives. This should not be a concern for CDN hosted content.
- In case software-related conversion errors from the original grid format to the one used by PROJ (be it GTX, NTv2 or GeoTIFF) would happen, the previous erroneous version of the dataset would be replaced by the corrected one. In that situation, this might have an effect with the local on-disk caching of remote grids. We will have to see with the CDN providers used if we can use for example the ETag HTTP header on the client to detect a change, so that old cached content is not erroneously reused (if not possible, we'll have to use some text file listing the grid names and their current md5sum)

#### 12.6.4.4 Grids in GeoTIFF format

##### Limitations of current formats

Several formats exist depending on the ad-hoc needs and ideas of the original data producer. It would be appropriate to converge on a common format able to address the different use cases.

- Not tiled. Tiling is a nice to have property for cloud-friendly access to large files.
- No support for compression
- The NTv2 structures is roughly: header of main grid, data of main grid, header of subgrid 1, data of subgrid 1, header of subgrid 2, data of subgrid 2, etc.Due to the headers being scattered through the file, it is not possibly to retrieve with a single HTTP GET request all header information.
- GTX format has no provision to store metadata besides the minimum georeferencing of the grid. NTv2 is a bit richer, but no extensible metadata possible.

##### Discussion on choice of format

We have been made recently aware of other initiatives from the industry to come with a common format to store geodetic adjustment data. Some discussions have happen recently within the OGC CRS Working group. Past efforts include the Esri's proposed Geodetic data Grid eXchange Format, GGXF, briefly mentioned at page 86 of https://iag.dgfi.tum.de/fileadmin/IAG-docs/Travaux2015/01_Travaux_Template_Comm_1_tvd.pdf and page 66 of ftp://ftp.iaspei.org/pub/meetings/2010-2019/2015-Prague/IAG-Geodesy.pdf The current trend of those works would be to use a netCDF / HDF5 container.

So, for the sake of completeness, we list hereafter a few potential candidate formats and their pros and cons.

##### TIFF/GeoTIFF

Strong points:

- TIFF is a well-known and widespread format.
- The GeoTIFF encoding is a widely industry supported scheme to encode georeferencing. It is now a OGC standard
- There are independent initiatives to share grids as GeoTIFF, like that one
- TIFF can contain multiple images (IFD: Image File Directory) chained together. This is the mechanism used for multiple-page scanned TIFF files, or in the geospatial field to store multi-resolution/pyramid rasters. So it can be used with sub-grids as in the NTv2 format.
- Extensive experience with the TIFF format, and its appropriateness for network access, in particular through the Cloud Optimized GeoTIFF initiative whose layout can make use of sub-grids efficient from a network access perspective, because grid headers can be put at the beginning of the file, and so being retrieved in a single HTTP GET request.
- TIFF can be tiled.
- TIFF can be compressed. Commonly found compression formats are DEFLATE, LZW, combined with differential integer or floating point predictors
- A TIFF image can contain a configurable number of channels/bands/samples. In the rest of the document, we will use the sample terminology for this concept.
- TIFF sample organization can be configured: either the values of different samples are packed together (PlanarConfiguration = Contig), or put in separate tiles/strips (PlanarConfiguration = Separate)
- libtiff is a dependency commonly found in binary distributions of the "ecosystem" to which PROJ belongs too

- libtiff benefits from many years of efforts to increase its security, for example being integrated to the oss-fuzz initiative. Given the potential fetching of grids, using security tested components is an important concern.
- Browser-side: there are "ports" of libtiff/libgeotiff in the browser such as https://geotiffjs.github.io/ which could potentially make a port of PROJ easier.

Weak points:

- we cannot use libgeotiff, since it depends itself on PROJ (to resolve CRS or components of CRS from their EPSG codes). That said, for PROJ intended use, we only need to decode the ModelTiepointTag and ModelPixelScaleTag TIFF tags, so this can be done "at hand"
- the metadata capabilities of TIFF baseline are limited. The TIFF format comes with a predefined set of metadata items whose keys have numeric values. That said, GDAL has used for the last 20 years or so a dedicated tag, GDAL_METADATA of code 42112 that holds a XML-formatted string being able to store arbitrary key-pair values.

### netCDF v3

Strong points:

- The binary format description as given in OGC 10-092r3 is relatively simple, but it would still probably be necessary to use libnetcdf-c to access it
- Metadata can be stored easily in netCDF attributes

Weak points:

- No compression in netCDF v3
- No tiling in netCDF v3
- Multi-samples variables are located in different sections of the files (correspond to TIFF PlanarConfiguration = Separate)
- No natural way of having hierarchical / multigrids. They must be encoded as separate variables
- georeferencing in netCDF is somewhat less standardized than TIFF/GeoTIFF. The generally used model is the conventions for CF (Climate and Forecast) metadata but there is nothing really handy in them for simple georeferencing with the coordinate of the upper-left pixel and the resolution. The practice is to write explicit long and lat variables with all values taken by the grid. GDAL has for many years supported a simpler syntax, using a GeoTransform attribute.
- From the format description, its layout could be relatively cloud friendly, except that libnetcdf has no API to plug an alternate I/O layer.
- Most binary distributions of netCDF nowadays are based on libnetcdf v4, which implies the HDF5 dependency.
- From a few issues we identified a few years ago regarding crashes on corrupted datasets, we contacted libnetcdf upstream, but they did not seem to be interested in addressing those security issues.

### netCDF v4 / HDF5

Note: The netCDF v4 format is a profile of the HDF5 file format.

Strong points:

- Compression supported (ZLIB and SZIP predefined)
- Tiling (chunking) supported
- Values of Multi-sample variables can be interleaved together (similarly to TIFF PlanarConfiguration = Contig) by using compound data types.

- Hierarchical organization with groups
- While the netCDF API does not provide an alternate I/O layer, this is possible with the HDF5 API.
- Grids can be indexed by more than 2 dimensions (for current needs, we don't need more than 2D support)

Weak points:

- The HDF 5 File format is more complex than netCDF v3, and likely more than TIFF. We do not have in-depth expertise of it to assess its cloud-friendliness.
- The ones mentioned for netCDF v3 regarding georeferencing and security apply.

### GeoPackage

As PROJ has already a SQLite3 dependency, GeoPackage could be examined as a potential solution.

Strong points:

- SQLite3 dependency
- OGC standard
- Multi-grid capabilities
- Tiling
- Compression
- Metadata capabilities

Weak points:

- GeoPackage mostly address the RGB(A) Byte use case, or via the tile gridded data extension, single-sample non-Byte data. No native support for multi-sample non-Byte data: each sample should be put in a separate raster table.
- Experience shows that SQLite3 layout (at least the layout adopted when using the standard libsqlite3) is not cloud friendly. Indices may be scattered in different places of the file.

### Conclusions

The 2 major contenders regarding our goals and constraints are GeoTIFF and HDF5. Given past positive experience and its long history, GeoTIFF remains our preferred choice.

### Format description

The format description is available in a dedicated *Geodetic TIFF grids (GTG)* document.

### Tooling

A script will be developed to accept a list of individual grids to combine together into a single file.

A ntv2_to_gtiff.py convenience script will be created to convert NTv2 grids, including their subgrids, to the above described GeoTIFF layout.

A validation Python script will be created to check that a file meets the above described requirements and recommendations.

#### Build requirements

The minimum libtiff version will be 4.0 (RHEL 7 ships with libtiff 4.0.3). To be able to read grids stored on the CDN, libtiff will need to build against zlib to have DEFLATE and LZW support, which is met by all known binary distributions of libtiff.

The libtiff dependency can be disabled at build time, but this must be an explicit setting of configure/cmake as the resulting builds have less functionality.

#### 12.6.4.5 Dropping grid catalog functionality

While digging through existing code, I more or less discovered that the PROJ code base has the concept of a grid catalog. This is a feature apparently triggered by using the +catalog=somefilename.csv in a PROJ string, where the CSV file list grid names, their extent, priority and date. It seems to be an alternative to using +nadgrids with multiple grids, with the extra ability to interpolate shift values between several grids if a +date parameter is provided and the grid catalog mentions a date for each grids. It was added in June 2012 per commit fcb186942ec8532655ff6cf4cc990e5da669a3bc

This feature is likely unknown to most users as there is no known documentation for it (neither in current documentation, nor in historic one). It is not either tested by PROJ tests, so its working status is unknown. It would likely make implementation of this RFC easier if this was removed. This would result in completely dropping the gridcatalog.cpp and gc_reader.cpp files, their call sites and the catalog_name and datum_date parameter from the PJ structure.

In case similar functionality would be be needed, it might be later reintroduced as an extra mode of *Horizontal grid shift*, or using a dedicated transformation method, similarly to the *Kinematic datum shifting utilizing a deformation model* one, and possibly combining the several grids to interpolate among in the same file, with a date metadata item.

#### 12.6.4.6 Backward compatibility issues

None anticipated, except the removal of the (presumably little used) grid catalog functionality.

#### 12.6.4.7 Potential future related work

The foundations set in the definition of the GeoTIFF grid format should hopefully be reused to extend them to support deformation models (was initially discussed per https://github.com/OSGeo/PROJ/issues/1001).

Definition of such an extension is out of scope of this RFC.

#### 12.6.4.8 Documentation

- New API function will be documented.
- A dedicated documentation page will be created to explain the working of network-based access.
- A dedicated documentation page will be created to describe the GeoTIFF based grid format. Mostly reusing above material.

#### 12.6.4.9 Testing

Number of GeoTIFF formulations (tiled vs untiled, PlanarConfiguration Separate vs Contig, data types, scale+offset vs not, etc.) will be tested.

For testing of network capabilities, a mix of real hits to the CDN and use of the alternate pluggable network interface to test edge cases will be used.

#### 12.6.4.10 Proposed implementation

A proposed implementation is available at https://github.com/OSGeo/PROJ/pull/1817

Tooling scripts are currently available at https://github.com/rouault/sample_proj_gtiff_grids/ (will be ultimately stored in PROJ repository)

#### 12.6.4.11 Adoption status

The RFC was adopted on 2020-01-10 with +1's from the following PSC members

- Kristian Evers
- Even Rouault
- Thomas Knudsen
- Howard Butler
- Kurt Schwehr

### 12.6.5 PROJ RFC 5: Adopt GeoTIFF-based grids for grids delivered with PROJ

**Author**
Even Rouault

**Contact**
even.rouault@spatialys.com

**Status**
Adopted

**Implementation target**
PROJ 7

**Last Updated**
2020-01-28

#### 12.6.5.1 Motivation

This RFC is a continuation of *PROJ RFC 4: Remote access to grids and GeoTIFF grids*. With RFC4, PROJ can, upon request of the user, download grids from a CDN in a progressive way. There is also API, such as `proj_download_file()` to be able to download a GeoTIFF grid in the user writable directory. The content of the CDN at https://cdn.proj.org is https://github.com/OSGeo/PROJ-data , which has the same content as https://github.com/OSGeo/proj-datumgrid converted in GeoTIFF files. In the current state, we could have a somewhat inconsistency between users relying on the proj-datumgrid, proj-datumgrid-[world,northamerica,oceania,europe] packages of mostly NTv2 and GTX files, and what is shipped through the CDN. Maintaining two repositories is also a maintenance burden in the long term.

It is thus desirable to have a single source of truth, and we propose it to be based on the GeoTIFF grids.

#### 12.6.5.2 Summary of work planned by this RFC and related decisions

- https://github.com/OSGeo/PROJ-data/ will be used, starting with PROJ 7.0, to create "static" grid packages.
- For now, a single package of, mostly GeoTIFF grids (a few text files for PROJ init style files, as well as a few edge cases for deformation models where grids have not been converted), will be delivered. Its size at the time of writing is 486 MB (compared to 1.5 GB of uncompressed NTv2 + GTX content, compressed to ~ 700 MB currently)
- The content of this archive will be flat, i.e. no subdirectories
- Each file will be named according to the following pattern `${agency_name}_${filename}[.ext]`. For example fr_ign_ntf_r93.tif This convention should allow packagers, if the need arise, to be able to split the monolithic package in smaller ones, based on criterion related to the country.

  The agency name is the one you can see from the directory names at https://github.com/OSGeo/PROJ-data/. `${agency_name}` itself is structure like

`${two_letter_country_code_of_agency_nationality}_${some_abbreviation}` (with the exception of eur_nkg, for the Nordic Geodetic Commission which isn't affiliated to a single country but to some European countries, and follows the general scheme)

- https://github.com/OSGeo/proj-datumgrid and related packages will only be maintained during the remaining lifetime of PROJ 6.x. After that, the repository will no longer receive any update and will be put in archiving state (see https://help.github.com/en/github/creating-cloning-and-archiving-repositories/about-archiving-repositories)
- PROJ database `grid_alternatives` table will be updated to point to the new TIFF filenames. It will also maintain the old names as used by current proj-datumgrid packages to be able to provide backward compatibility when a PROJ string refers to a grid by its previous name.
- Upon adoption of this RFC, new grids referenced by PROJ database will only point to GeoTIFF grid names.
- Related to the above point, if a PROJ string refers to a grid name, let's say foo.gsb. This grid will first be looked for in all the relevant locations under this name. If no match is found, then a lookup in the `grid_alternatives` table will be done to retrieve the potential new name (GeoTIFF file), and if there's such match, a new look-up in the file system will be done with the name of this GeoTIFF file.
- The `package_name` column of grid_alternatives will no longer be filled. And `url` will be filled with the direct URL to the grid in the CDN, for example: https://cdn.proj.org/fr_ign_ntf_r93.tif
- The Python scripts to convert grids (NTv2, GTX) to GeoTIFF currently available at https://github.com/rouault/sample_proj_gtiff_grids/ will be moved to a grid_tools/ subdirectories of https://github.com/OSGeo/PROJ-data/ Documentation for those utilities will be added to PROJ documentation.
- Obviously, all the above assumes PROJ builds to have libtiff enabled. Non-libtiff builds are not considered as nominal PROJ builds (if a PROJ master build is attempted and libtiff is not detected, it fails. The user has to explicitly ask to disable TIFF support), and users deciding to go through that route will have to deal with the consequences (that is that grid-based transformations generated by PROJ will likely be non working)

#### 12.6.5.3 Backward compatibility

This change is considered to be *mostly* backward compatible. There might be impacts for software using `proj_coordoperation_get_grid_used()` and assuming that the url returned is one of the proj-datumgrid-xxx files at https://download.osgeo.org. As mentioned in https://lists.osgeo.org/pipermail/proj/2020-January/009274.html , this assumption was not completely bullet-proof either. There will be impacts on software checking the value of PROJ pipeline strings resulting `proj_create_crs_to_crs()`. The new grid names will now be returned (the most impacted software will likely be PROJ's own test suite)

Although discouraged, people not using the new proj-datumgrid-geotiff-XXX.zip archives, should still be able to use the old archives made of NTv2/GTX files, at least as long as the PROJ database does not only point to a GeoTIFF grid. So this might be a short-term partly working solution, but at time goes, it will become increasingly non-working. The nominal combination will be PROJ 7.0 + proj-datumgrid-geotiff-1.0.zip

#### 12.6.5.4 Testing

PROJ test suite will have to be adapted for the new TIFF based filenames.

Mechanism to auto-promote existing NTv2/GTX names to TIFF ones will be exercised.

#### 12.6.5.5 Proposed implementation

https://github.com/OSGeo/PROJ/pull/1891 and https://github.com/OSGeo/PROJ-data/pull/5

#### 12.6.5.6 Adoption status

The RFC was adopted on 2020-01-28 with +1's from the following PSC members

- Kristian Evers
- Even Rouault
- Thomas Knudsen
- Howard Butler
- Kurt Schwehr

### 12.6.6 PROJ RFC 6: Triangulation-based transformations

**Author**
Even Rouault

**Contact**
even.rouault@spatialys.com

**Status**
Adopted

**Implementation target**
PROJ 7.2

**Last Updated**
2020-09-02

#### 12.6.6.1 Summary

This RFC adds a new transformation method, `tinshift` (TIN stands for Triangulated Irregular Network)

The motivation for this work is to be able to handle the official transformations created by National Land Survey of Finland, for:

- horizontal transformation between the KKJ and ETRS89 horizontal datums
- vertical transformations between N43 and N60 heights, and N60 and N2000 heights.

Such transformations are somehow related to traditional grid-based transformations, except that the correction values are hold by the vertices of the triangulation, instead of being at nodes of a grid.

Triangulation are in a number of cases the initial product of a geodesic adjustment, with grids being a derived product. The Swiss grids have for example derived products of an original triangulation.

Grid-based transformations remain very convenient to use because accessing correction values is really easy and efficient, so triangulation-based transformations are not meant as replacing them, but more about it being a complement, that is sometimes necessary to be able to replicate the results of a officially vetted transformation to a millimetric or better precision (speaking here about reproducibility of numeric results, rather than the physical accuracy of the transformation that might rather be centimetric). It is always possible to approach the result of the triangulation with a grid, but that may require to adopt a small grid step, and thus generate a grid that can be much larger than the original triangulation.

#### 12.6.6.2 Details

##### Transformation

A new transformation method, `tinshift`, is added. It takes one mandatory argument, `file`, that points to a JSON file, which contains the triangulation and associated metadata. Input and output coordinates must be geographic or

projected. Depending on the content of the JSON file, horizontal, vertical or both components of the coordinates may be transformed.

The transformation is used like:

```
$ echo 3210000.0000 6700000.0000 0 2020 | cct +proj=tinshift +file=./triangulation_kkj.
↪json

209948.3217     6697187.0009    0.0000     2020
```

The transformation is invertible, with the same computational complexity than the forward transformation.

### Algorithm

Internally, `tinshift` ingest the whole file into memory. It is considered that triangulation should be small enough for that. The above mentioned KKJ to ETRS89 triangulation fits into 65 KB of JSON, for 1449 triangles and 767 vertices.

When a point is transformed, one must find the triangle into which it falls into. Instead of iterating over all triangles, we build a in-memory quadtree to speed-up the identification of candidates triangles. On the above mentioned KKJ -> ETRS89 triangulation, this speeds up the whole transformation by a factor of 10. The quadtree structure is a very good compromise between the performance gain it brings and the simplicity of its implementation (we have ported the implementation coming from GDAL, inherit from the one used for shapefile .spx spatial indices).

To determine if a point falls into a triangle, one computes its 3 barycentric coordinates from its projected coordinates, $\lambda_i$ for $i = 1, 2, 3$. They are real values (in the [0,1] range for a point inside the triangle), giving the weight of each of the 3 vertices of the triangles.

Once those weights are known, interpolating the target horizontal coordinate is a matter of doing the linear combination of those weights with the target horizontal coordinates at the 3 vertices of the triangle ($Xt_i$ and $Yt_i$):

$$X_{target} = Xt_1 * \lambda_1 + Xt_2 * \lambda_2 + Xt_3 * \lambda_3$$
$$Y_{target} = Yt_1 * \lambda_1 + Yt_2 * \lambda_2 + Yt_3 * \lambda_3$$

This interpolation is exact at the vertices of the triangulation, and has linear properties inside each triangle. It is completely equivalent to other formulations of triangular interpolation, such as

$$X_{target} = A + X_{source} * B + Y_{source} * C$$
$$Y_{target} = D + X_{source} * E + Y_{source} * F$$

where the A, B, C, D, E, F constants (for a given triangle) are found by solving the 2 systems of 3 linear equations, constraint by the source and target coordinate pairs of the 3 vertices of the triangle:

$$Xt_i = A + Xs_i * B + Ys_i * C$$
$$Yt_i = D + Xs_i * E + Ys_i * F$$

**Note**

From experiments, the interpolation using barycentric coordinates is slightly more numerically robust when interpolating projected coordinates of amplitude of the order of 1e5 / 1e6, due to computations involving differences of coordinates. Whereas the formulation with the A, B, C, D, E, F tends to have big values for the A and D constants, and values clause to 0 for C and E, and close to 1 for B and F. However, the difference between the two approaches is negligible for practical purposes (below micrometre precision)

Similarly for a vertical coordinate transformation, where $Zoff_i$ is the vertical offset at each vertex of the triangle:

$$Z_{target} = Z_{source} + Zoff_1 * \lambda_1 + Zoff_2 * \lambda_2 + Zoff_3 * \lambda_3$$

### Constraints on the triangulation

No check is done on the consistence of the triangulation. It is highly recommended that triangles do not overlap each other (when considering the source coordinates or the forward transformation, or the target coordinates for the inverse transformation), otherwise which triangle will be selected is unspecified. Besides that, the triangulation does not need to have particular properties (like being a Delaunay triangulation)

### File format

To the best of our knowledge, there are no established file formats to convey geodetic transformations as triangulations. Potential similar formats to store TINs are ITF or XMS. Both of them would need to be extended in order to handle datum shift information, since they are both intended for mostly DEM use.

We thus propose a text-based format, using JSON as a serialization. Using a text-based format could potentially be thought as a limitation performance-wise compared to binary formats, but for the size of triangulations considered (a few thousands triangles / vertices), there is no issue. Loading such file is a matter of 20 milliseconds or so. For reference, loading a triangulation of about 115 000 triangles and 71 000 vertices takes 450 ms.

Using JSON provides generic formatting and parsing rules, and convenience to create it from Python script for examples. This could also be easily generated "at hand" by non-JSON aware writers.

For generic metadata, we reuse closely what has been used for the Deformation model master file

Below a minimal example, from the KKJ to ETRS89 transformation, with just a single triangle:

```
{
    "file_type": "triangulation_file",
    "format_version": "1.0",
    "name": "Name",
    "version": "Version",
    "publication_date": "2018-07-01T00:00:00Z",
    "license": "Creative Commons Attribution 4.0 International",
    "description": "Test triangulation",
    "authority": {
        "name": "Authority name",
        "url": "http://example.com",
        "address": "Address",
        "email": "test@example.com"
    },
    "links": [
    {
        "href": "https://example.com/about.html",
        "rel": "about",
        "type": "text/html",
        "title": "About"
    },
    {
        "href": "https://example.com/download",
        "rel": "source",
        "type": "application/zip",
        "title": "Authoritative source"
    },
    {
        "href": "https://creativecommons.org/licenses/by/4.0/",
        "rel": "license",
        "type": "text/html",
```

(continues on next page)

(continued from previous page)

```
        "title": "Creative Commons Attribution 4.0 International license"
    },
    {
        "href": "https://example.com/metadata.xml",
        "rel": "metadata",
        "type": "application/xml",
    "title": " ISO 19115 XML encoded metadata regarding the triangulation"
        }
    ],
    "input_crs": "EPSG:2393",
    "target_crs": "EPSG:3067",
    "transformed_components": [ "horizontal" ],
    "vertices_columns": [ "source_x", "source_y", "target_x", "target_y" ],
    "triangles_columns": [ "idx_vertex1", "idx_vertex2", "idx_vertex3" ],
    "vertices": [ [3244102.707, 6693710.937, 244037.137, 6690900.686],
                  [3205290.722, 6715311.822, 205240.895, 6712492.577],
                  [3218328.492, 6649538.429, 218273.648, 6646745.973] ],
    "triangles": [ [0, 1, 2] ]
}
```

So after the generic metadata, we define the input and output CRS (informative only), and that the transformation affects horizontal components of coordinates. We name the columns of the `vertices` and `triangles` arrays. We defined the source and target coordinates of each vertex, and define a triangle by referring to the index of its vertices in the `vertices` array.

More formally, the specific items for the triangulation file are:

**input_crs**
: String identifying the CRS of source coordinates in the vertices. Typically `EPSG:XXXX`. If the transformation is for vertical component, this should be the code for a compound CRS (can be EPSG:XXXX+YYYY where XXXX is the code of the horizontal CRS and YYYY the code of the vertical CRS). For example, for the KKJ->ETRS89 transformation, this is EPSG:2393 (`KKJ / Finland Uniform Coordinate System`). The input coordinates are assumed to be passed in the "normalized for visualisation" / "GIS friendly" order, that is longitude, latitude for geographic coordinates and easting, northing for projected coordinates.

**output_crs**
: String identifying the CRS of target coordinates in the vertices. Typically `EPSG:XXXX`. If the transformation is for vertical component, this should be the code for a compound CRS (can be EPSG:XXXX+YYYY where XXXX is the code of the horizontal CRS and YYYY the code of the vertical CRS). For example, for the KKJ->ETRS89 transformation, this is EPSG:3067 ("ETRS89 / TM35FIN(E,N)"). The output coordinates will be returned in the "normalized for visualisation" / "GIS friendly" order, that is longitude, latitude for geographic coordinates and easting, northing for projected coordinates.

**transformed_components**
: Array which may contain one or two strings: "horizontal" when horizontal components of the coordinates are transformed and/or "vertical" when the vertical component is transformed.

**vertices_columns**
: Specify the name of the columns of the rows in the `vertices` array. There must be exactly as many elements in `vertices_columns` as in a row of `vertices`. The following names have a special meaning: `source_x`, `source_y`, `target_x`, `target_y`, `source_z`, `target_z` and `offset_z`. `source_x` and `source_y` are compulsory. `source_x` is for the source longitude (in degree) or easting. `source_y` is for the source latitude (in degree) or northing. `target_x` and `target_y` are compulsory when `horizontal` is specified in `transformed_components`. (`source_z` and `target_z`) or `offset_z` are compulsory when `vertical` is specified in `transformed_components`

**triangles_columns**
Specify the name of the columns of the rows in the `triangles` array. There must be exactly as many elements in `triangles_columns` as in a row of `triangles`. The following names have a special meaning: `idx_vertex1`, `idx_vertex2`, `idx_vertex3`. They are compulsory.

**vertices**
An array whose items are themselves arrays with as many columns as described in `vertices_columns`.

**triangles**
An array whose items are themselves arrays with as many columns as described in `triangles_columns`. The value of the `idx_vertexN` columns must be indices (between 0 and len(`vertices`-1) of items of the `vertices` array.

#### Code impacts

The following new files are added in src/transformations:

- tinshift.cpp: PROJ specific code for defining the new operation. Takes care of the input and output coordinate conversions (between input_crs and triangulation_source_crs, and triangulation_target_crs and output_crs), when needed.
- tinshift.hpp: Header-based implementation. This file contains the API.
- tinshift_exceptions.hpp: Exceptions that can be raised during file parsing
- tinshift_impl.hpp: Implementation of file loading, triangle search and interpolation.

This is the approach that has been followed for the deformation model implementation, and which makes it easier to do unit test.

src/quadtree.hpp contains a quadtree implementation.

#### Performance indications

Tested on Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, transforming 4 million points

For the KKJ to ETRS89 transformation (1449 triangles and 767 vertices), 4.4 million points / sec can be transformed.

For comparison, the Helmert-based KKJ to ETRS89 transformation operates at 1.6 million points / sec.

A triangulation with about 115 000 triangles and 71 000 vertices operates at 2.2 million points / sec (throughput on more points would be better since the initial loading of the triangulation is non-negligible here)

#### 12.6.6.3 Backward compatibility

New functionality fully backward compatible.

#### 12.6.6.4 Testing

The PROJ test suite will be enhanced to test the new transformation, with a new .gie file, and a C++ unit test to test at a lower level.

#### 12.6.6.5 Documentation

- The tinshift method will be documented.
- The JSON format will be documented under https://proj.org/specifications/
- A JSON schema will also be provided.

#### 12.6.6.6 Proposed implementation

An initial implementation is available at https://github.com/rouault/PROJ/tree/tinshift

#### 12.6.6.7 References

Finnish coordinate transformation (automated translation to English)

#### 12.6.6.8 Adoption status

The RFC was adopted on 2020-09-02 with +1's from the following PSC members

- Kristian Evers
- Charles Karney
- Thomas Knudsen
- Even Rouault

#### 12.6.6.9 Funding

This work is funded by National Land Survey of Finland

### 12.6.7 PROJ RFC 7: Drop Autotools, maintain CMake

**Author**
Mike Taves

**Contact**
mwtoews@gmail.com

**Status**
Adopted

**Implementation target**
PROJ 9.0

**Last Updated**
2021-10-27

#### 12.6.7.1 Summary

This RFC proposes to drop Autotools for PROJ 9.0, and to maintain CMake for build automation, testing and packaging. This will reduce the overall maintenance for PROJ and will enable the library to be integrated into other projects that use CMake.

#### 12.6.7.2 Background

Here is a short timeline of the build tools used for PROJ:

- Throughout the mid-1990s, Gerald Evenden maintained a Unix build system with a few scripts (some derived from Autoconf), and Makefile templates.
- In 2000, Frank Warmerdam wrote Autoconf and Automake configurations for PROJ 4.4.0.
- This was followed by a NMake configuration to build PROJ 4.4.2 for Windows.
- In 2014, a CMake build setup was introduced by Howard Butler for PROJ 4.9.0RC1. The CMake configuration was improved for the 4.9.1 release, but not considered at feature parity with the Autotools builds at the time.
- The NMake build setup was removed for PROJ 6.0.0, as its functionality had been replaced by CMake.

#### 12.6.7.3 Motivation

The primary motivation in removing Autotools is to reduce the burden of maintaining multiple build configurations, which requires developers to be familiar with different tools and configuration files. There are several other benefits in maintaining a single build system:

- Remove extra configuration and m4 macro files from source repository,
- Simplify scripts used for running tests for CI services (GitHub Actions, TravisCI),
- Reduce compilation time (and carbon footprint) used for testing on CI services,
- Ease development effort, particularly with new contributors.

#### 12.6.7.4 Why drop Autotools?

The GNU Build System or Autotools consist of a suite of tools including Autoconf and Automake, which can be used to build software on Unix-like systems. These tools are not cross-platform, and do not naively integrate with development environments on Microsoft Windows. Furthermore, the existing PROJ Autotools builds do not install the CMake configuration files required to find PROJ from other projects that use CMake (#2546).

#### 12.6.7.5 Why use CMake?

CMake is an open source cross-platform tool for build automation, testing and packaging of software. It does not directly compile the software, but manages the build process using generators, including Unix Makefiles and Ninja among other command-based and IDE tools. The CMake software has been under active development since its origins in 2000. The CMake language is carefully developed with backwards-compatible policies that aim to provide consistent behavior across different versions. CMake is currently the preferred build tool for PROJ for the following reasons:

- It has existed in the PROJ code base since 2014, and is familiar to active PROJ contributors,
- It can install configuration files that can be used by other software that use CMake to find PROJ for linking via `find_package()`,
- CMake configurations are used in 3rd-party binary packages of PROJ, including conda-forge and vcpkg,
- It can be used to build PROJ on all major operating systems and compiler combinations (where compatible),
- It has integration with modern IDEs and tools, including Microsoft Visual Studio and Qt Creator.

#### 12.6.7.6 Why not CMake?

Other modern cross-platform build systems exist, including Meson and Bazel, which have many advantages over CMake. However, they are currently not widely used by active PROJ contributors. This RFC should not restrict future build system configurations being introduced to PROJ, if they are proven to have advantages to CMake over time.

#### 12.6.7.7 Potential impacts

Binary packagers that currently rely on Autotools would obviously need to transition building and testing PROJ with CMake. Issues related to multiarch builds of PROJ may become apparent, which can be patched and/or reported to PROJ developers. One feature of Autotools is that both static and dynamic (shared) libraries are built, which packagers may distribute. This feature is currently not set-up for PROJ, as it would need to be configured and built twice.

End-users that use binary packages of PROJ should not be impacted. PROJ should be discoverable via both pkg-config and CMake's `find_package()`. Other projects that use Autotools will continue to work as expected, linking statically or dynamically to PROJ built by CMake.

#### 12.6.7.8 Transition plan

If this proposal is approved, the following tasks should be completed:

- Rewrite CI tests to only use CMake for packaging, building, testing, installation and post-install tests,
- Remove files only used by Autotools, also update `.gitignore`,
- Update documentation and `HOWTORELEASE` notes.

Related issues will be tracked on GitHub with a tag RFC7: Autotools→CMake.

#### 12.6.7.9 Adoption status

The RFC was adopted on 2021-10-26 with +1's from the following PSC members

- Kristian Evers
- Even Rouault
- Howard Butler
- Thomas Knudsen
- Kurt Schwehr
- Charles Karney
- Thomas Knudsen

### 12.6.8 PROJ RFC-8: Embedding resource files into libproj

**Author**
Even Rouault

**Contact**
even.rouault @ spatialys.com

**Status**
Adopted, implemented

**Implementation target**
PROJ 9.6

**Last Updated**
2024-Oct-01

#### 12.6.8.1 Summary

This RFC provides an optional way of embedding `proj.db` and `proj.ini` files directly into libproj, either using C23 `#embed` pre-processor directive when supported by compilers, or falling back to a CMake-based script for older compilers.

#### 12.6.8.2 Motivation

Most common practical PROJ use critically depends on the availability of `proj.db`. Locating that resource files on the file system can be painful in some use cases of PROJ, that involve relocating the PROJ binary at installation time. One such case could be the PROJ embedded in Rasterio or Fiona binary wheels where `PROJ_DATA` must be correctly set currently. DuckDB Spatial also patches PROJ to embed `proj.db` in its static build. Web-assembly (WASM) use cases come also to mind as users of PROJ builds where resources are directly included in libproj. Given the existence of several out-of-tree patches to support embedding `proj.db` (such as https://github.com/OSGeo/PROJ/issues/2998#issuecomment-1004185741), it makes sense to have a upstream-vetted solution the community can build around, and potentially use as a point for further works.

#### 12.6.8.3 Technical solution

##### C23 #embed

The C23 standard includes a #embed "filename" pre-processor directive that ingests the specified filename and returns its content as tokens that can be stored in a unsigned char or char array.

Getting the content of a file into a variable is as simple as the following:

```
static const unsigned char proj_db[] = {
#embed "data/proj.db"
};
```

Support for that directive is still very new. clang 19.1 is the first compiler which has a release including it, and has an efficient implementation of it, able to embed very large files with minimum RAM and CPU usage.

The development version of GCC 15 also supports it, but in a non-optimized way for now. i.e. trying to include large files, of several tens of megabytes could cause significant compilation time, but without impact on runtime. There is expressed intent from GCC developers to improve this in the future.

Embedding PROJ's `proj.db` of size 9.1 MB with GCC 15dev at time of writing takes 18 seconds and 1.7 GB RAM, compared to 0.4 second and 400 MB RAM for clang 19, which is still reasonable (Generating `proj.db` itself from its source .sql files takes one minute on the same system).

There is no timeline for Visual Studio C/C++ at time of writing (it has been requested by users)

To be noted that currently clang 19.1 only supports `#embed` in .c files, not C++ ones (the C++ standard has not yet adopted this feature). So embedding resources must be done in a .c file, which is obviously not a problem since we can easily export symbols/functions from a .c file to be available by C++.

##### Fallback C99/C++11 mode

For non C23 capable compilers, a CMake script, derived from https://jonathanhamberg.com/post/cmake-file-embedding/ (which is MIT licensed) is used that generates a .c and .h file per file to embed. The C file consists of a `const unsigned uint8_t content[] = { .... }` array, which matches what a non-optimization implementation of C23 #embed typically does.

This script has been improved because it performed very poorly on large files such as `proj.db`. Its execution time is now 8 seconds for `proj.db`.

##### memvfs

Loading of the embedded `proj.db` involves using the SQLite3 memvfs, as done by DuckDB Spatial

#### 12.6.8.4 New CMake options

Resources will only be embedded if the new `EMBED_RESOURCE_FILES` CMake option is set to `ON`. This option will default to `ON` for static library builds and if *C23 `#embed`* is detected to be available. Users might also turn it to ON for shared library builds. A CMake error is emitted if the option is turned on but the compiler lacks support for it.

A complementary CMake option `USE_ONLY_EMBEDDED_RESOURCE_FILES` will also be added. It will default to `OFF`. When set to ON, PROJ will not try to locate resource files in the PROJ_DATA directory burnt at build time into libproj (`${install_prefix}/share/proj`), or by the `PROJ_DATA` configuration option.

Said otherwise, if `EMBED_RESOURCE_FILES=ON` but `USE_ONLY_EMBEDDED_RESOURCE_FILES=OFF`, PROJ will first try to locate resource files from the file system, and fallback to the embedded version if not found.

The resource files will still be installed in `${install_prefix}/share/proj`, unless `USE_ONLY_EMBEDDED_RESOURCE_FILES` is set to ON.

#### 12.6.8.5 Impacted code

- `cmake/FileEmbed.cmake`: compatibility script for non-C23 mode to generate embedded resources
- `data/CMakeLists.txt`: take into account USE_ONLY_EMBEDDED_RESOURCE_FILES to not install proj.db/proj.ini when it is ON
- `docs/source/install.rst`: document EMBED_RESOURCE_FILES and USE_ONLY_EMBEDDED_RESOURCE_FILES
- `src/embedded_resources.c` and .h: new files that use #embed or make a bridge to files generated by FileEmbed.cmake
- `src/filemanager.cpp`: to take into account EMBED_RESOURCE_FILES for proj.ini
- `src/iso19111/factory.cpp`: to take into account EMBED_RESOURCE_FILES for proj.db
- `src/lib_proj.cmake`: takes into account EMBED_RESOURCE_FILES and USE_ONLY_EMBEDDED_RESOURCE_FILES in both C23 and non-C23 modes
- `src/memvfs.c` and .h: code originating from https://www.sqlite.org/src/file/ext/misc/memvfs.c to handle a in-memory proj.db, with bug fixes, and adaptation for PROJ needs
- `src/sqlite3_utils.cpp` and .hpp: interface layer of memvfs with src/iso19111/factory.cpp

#### 12.6.8.6 Out of scope

Embedding of resource files in PROJ is currently limited to `proj.db` and `proj.ini`, as those are the ones which are expected to be the most needed ones in typical embedded use cases. Extension to other resources (ITRFxxxx file) could potentially be done as follow-up enhancements if the need arose, although supporting dual C23/non-C23 mode for too many files could be a bit tedious.

The sky is the limit, so potentially grid files could also be embedded. That would require developing a MemFile implementation in filemanager.cpp (in parallel to the existing FileStdio, FileWin32 or NetworkFile).

#### 12.6.8.7 Backward compatibility

Fully backwards compatible with default settings.

Static builds will default to EMBED_RESOURCE_FILES=ON, but USE_ONLY_EMBEDDED_RESOURCE_FILES will default to OFF. So an external `proj.db` and `proj.ini` found by existing search mechanisms will still have precedence over the embedded files.

Even when EMBED_RESOURCE_FILES an/or USE_ONLY_EMBEDDED_RESOURCE_FILES is enabled, the user can still use `proj_context_set_database_path()` to provide an alternate database. Network based fetching of grids is also orthogonal to those settings.

C23 is not required: it is just an opportunity for faster build time when available.

#### 12.6.8.8 Documentation

The 2 new CMake variables will be documented.

#### 12.6.8.9 Testing

The existing fedora:rawhide continuous integration target, which has now clang 19.1 available, will be modified to test the effect of the new variables.

Local builds using GCC 15dev builds of https://jwakely.github.io/pkg-gcc-latest/ have also be successfully done during the development of the candidate implementation

#### 12.6.8.10 Related issues and PRs

- GDAL RFC 102 text "Embedding resource files into libgdal"
- PROJ candidate implementation
- ticket "RFC 8 (resource embedding) follow-up: update requirements to C23 when generally availability of it"

#### 12.6.8.11 Voting history

+1 from PSC members KurtS, KristianE, JavierJS, ThomasK and EvenR

CHAPTER

# THIRTEEN

# TUTORIALS

## 13.1 Transformation tutorial

### 13.1.1 Introduction

This tutorial was developed for the EUREF 2019 Symposium as an introduction to geodetic transformations in PROJ. The intended audience was geodesist in governmental geodetic agencies but the tutorial should be of interest to a broader group of people within GIS, remote sensing and related fields.

The tutorials gives insight into how transformations are set up using the "PROJ language" as well as introduce most of the command line applications packaged with PROJ. In the tutorial properties of several common projections and transformations are investigated.

The tutorial requires a working installation of PROJ. Consult the *chapter on installing PROJ*.

### 13.1.2 How to fill out the PROJ tutorial exercises

All the exercises are made as *gie* tests where one or more inputs are missing. Replace occurrences of `<your answer here>` with the relevant input. Below is an annotated example of a complete *gie* test. Your objective is to fill out the exercises from 1 and onwards in a similar fashion.

Below is a simple exercise before the answer has been filled in:

```
1. Set up an operation that returns the input it is given
--------------------------------------------------------------------------------
operation   <your answer here>
tolerance   10 mm

accept      12.3    45.6
expect      12.3    45.6
--------------------------------------------------------------------------------
```

An annotated answer could look like this:

```
<gie>
operation   +proj=latlon    # The latlon "projection" returns the same output as
                            # given in the input
tolerance   10 mm

accept      12.3    45.6
expect      12.3    45.6
</gie>
```

The *gie* application can be used to check if the exercise has been answered correctly:

```
gie exercises\<exercise_name>.gie
```

If all answers are correct *gie* will return output similar to:

```
------------------------------------------------------------------------------
Reading file 'exercises\instructions.md'
------------------------------------------------------------------------------
total:  1 tests succeeded,  0 tests skipped,  0 tests failed.
------------------------------------------------------------------------------
```

If incorrect, *gie* will output something like:

```
------------------------------------------------------------------------------
Reading file 'exercises\instructions.md'
     -----
     FAILURE in instructions.md(48):
     expected: 12.3 45.6
     got:      1369229.736757265171   5685937.873489554040
     deviation:  999999999.999000 mm,  expected:  10.000000 mm
------------------------------------------------------------------------------
total:  0 tests succeeded,  0 tests skipped,  1 tests FAILED!
------------------------------------------------------------------------------
```

Where the number of failing exercises (or tests in *gie* terminology) is given, as well as the line number of the failing test (in parenthesis after the file name).

### 13.1.3 Exercises

#### 13.1.3.1 The UTM projection

In this exercise we will learn how to define a simple projection using PROJ. We will investigate a few aspects of the UTM projection, as well as learn how to use a different ellipsoid than the default (GRS80).

Download the *gie* file for the exercise: `projections1.gie`.

##### Exercise 1: Setting up a projection

Set up a UTM projection for use in Estonia.

> **Hint**
>
> See *Universal Transverse Mercator (UTM)* for more on the projection

```
operation   <your answer here>
tolerance   1 cm

accept      24.745          59.437       # Talinn, Estonia
expect      372106.37       6590881.40
```

#### Exercise 2: UTM on the southern hemisphere

Set up a UTM projection for use on the North Island of New Zealand.

> **Hint**
>
> See *Universal Transverse Mercator (UTM)* for more on the projection

```
operation   <your answer here>
tolerance   1 cm

accept      174.740        -36.841      # Auckland, New Zealand
expect      298481.34       5920382.04
```

#### Exercise 3: Using a non-standard ellipsoid

Set up a UTM projection based on the Hayford ellipsoid for use in Estonia.

> **Hint**
>
> See *Universal Transverse Mercator (UTM)* for more on the projection

> **Hint**
>
> Run `proj -le` to get a list of available ellipsoids

```
operation   <your answer here>
tolerance   1 cm

accept      24.745          59.437       # Talinn, Estonia
expect      372099.99       6591034.35
```

### 13.1.3.2 The Lambert Conic Conformal projection

In this exercise we will introduce the Lambert Conic Conformal (LCC) projection as well as general map projection concepts such as projection center, scale factor, false easting/northing and ellipsoid models.

We will do so by defining various versions of the LCC optimised for use in and around the Gulf of Finland.

In case you are not familiar with the LCC projection, Wikipedia describes the Lambert Conic Conformal projection as:

> "A Lambert conformal conic projection (LCC) is a conic map projection used for aeronautical charts, portions of the State Plane Coordinate System, and many national and regional mapping systems. It is one of seven projections introduced by Johann Heinrich Lambert in his 1772 publication Anmerkungen und Zusätze zur Entwerfung der Land- und Himmelscharten (Notes and Comments on the Composition of Terrestrial and Celestial Maps).
>
> Conceptually, the projection seats a cone over the sphere of the Earth and projects the surface conformally onto the cone. The cone is unrolled, and the parallel that was touching the sphere is assigned unit scale. That parallel is called the reference parallel or standard parallel.

> By scaling the resulting map, two parallels can be assigned unit scale, with scale decreasing between the two parallels and increasing outside them. This gives the map two standard parallels. In this way, deviation from unit scale can be minimized within a region of interest that lies largely between the two standard parallels. Unlike other conic projections, no true secant form of the projection exists because using a secant cone does not yield the same scale along both standard parallels"

Download the *gie* file for the exercise: `projections2.gie`.

### Exercise 1: Standard parallel

Set up a LCC projection with one standard parallel placed halfway between Helsinki and Tallinn.

**Hint**

- Approximate location of Helsinki: 60.171N 24.938E
- Approximate location of Tallinn: 59.437N 24.745E
- Approximate mid point between Helsinki and Tallinn: 59.8N 24.8E

**Hint**

See *Lambert Conformal Conic* for more on the Lambert Conformal Conic projection

```
operation   <your answer here>
tolerance   1 cm

accept      24.938              60.171  # Helsinki
expect      1351962.01          298663.54

accept      24.745              59.437  # Tallinn
expect      1371811.19          218598.22
```

### Exercise 2: Distance distortion

Improve the distance distortion of the projection.

The projection defined above is conformal, i.e. has no angular distortions for infinitesimal objects - but there is a minor distortion of distances. This cannot be avoided in general, but locally distortions can be minimized by adjusting the scale factor.

With the default factor of 1, the distance error is 0 for (infinitesimal) line segments at the latitude of the standard parallel.

By reducing the scale factor, the line of zero distance distortion degenerates into two lines, north and south of the standard parallel.

For the above-mentioned point in Tallinn, the distance error is around 0.002%, but this can be reduced to zero by varying the scaling factor.

Even by trial and error it is not hard to reduce it to a few millionths of a percent. So try to find a more optimal scaling factor by trial and error using the projection analysis functionality of the proj application.

**Hint**

Modify and use the command below to find a better scaling factor:
`echo 24.745 59.437 | proj -V <your projection setup here>`

**Hint**

Look for the lines displaying the meridian and parallel scale.

**Hint**

Consult the LCC documentation to find out how to set the scaling factor.

**Hint**

Scaling factors usually deviates from unity by the order of 1e-3 to 1e-5

```
operation   <your answer here>
tolerance   1 um

accept      24.745              59.437
expect      1371783.759883408   218593.850591891
```

### Exercise 3: Using two standard parallels

Set up a similar LCC projection, this time using two standard parallels. The first passing through Helsinki, the second through Tallinn.

**Hint**

- Approximate location of Helsinki: 60.171N 24.938E
- Approximate location of Talinn: 59.437N 24.745E

```
operation   <your answer here>
tolerance   1 cm

accept      24.8                59.8
expect      1359907.80          8024082.260
```

### Exercise 4: Projection center

Change the projection center to the midway point between Helsinki and Tallinn.

By changing the projection center we move the origin of the output coordinates to be coinciding with the position where the projection is most accurate.

**Hint**

Start with the projection you designed in exercise 1 above

**Hint**

The mid point between Helsinki and Tallinn is roughly 59.8N 24.8E

**Hint**

*Lambert Conformal Conic* explains how to change the projection center

```
operation   <your answer here>
tolerance   1 cm

accept      24.8            59.8
expect       0.0             0.0
```

#### Exercie 5: False easting and northing

Introduce false eastings and northings.

Oftentimes it is preferable to keep coordinate values positive even when moving west and south of the projection center. This can be achieved by introducing false easting and northing. Add false easting and northing of 1000000

**Hint**

*Lambert Conformal Conic* explains how to set false easting/northing

```
operation   <your answer here>
tolerance   1 cm

accept      24.8            59.8
expect      1000000.00      1000000.00
```

### 13.1.3.3 Specifying ellipsoid models

In this exercise we will introduce some of the many ways one can specify the ellipsoid model used in projections and transformations.

We will be using the Mercator projection to illustrate the different ellipsoid settings, but note that the same parameter settings can be used for setting the ellipsoid used for most PROJ projections and transformations.

Download the *gie* file for the exercise: `ellipsoids.gie`.

### Exercise 1: Ellipsoid radius

Use a sphere of radius 1 as the ellipsoid model

**Hint**

Check the *Mercator documentation* to find out how to specify an ellipsoid

```
operation   +proj=merc  <your answer here>
tolerance   1 cm

accept      24.745          59.437          # Talinn
expect      0.4319          1.2975
```

### Exercise 2. Specifying a ellipsoid

Set up a Mercator projection that uses the Hayford (International) ellipsoid

**Hint**

Check the *Mercator documentation* to find out how to specify an ellipsoid

**Hint**

Running `proj -le` will return a list of ellipsoids and their parameters.

```
operation   +proj=merc  <your answer here>
tolerance   1 cm

accept      24.745          59.437          # Talinn
expect      2754709.2020    8238786.4803
```

### Exercise 3. Semimajor axis and inverse flattening

Specify the Hayford ellipsoid by its semimajor axis radius and inverse flattening

**Hint**

Find the parameters in the list returned by `proj -le`

```
operation   +proj=merc  <your answer here>
tolerance   1 cm

accept      24.745          59.437          # Talinn
expect      2754709.2020    8238786.4803
```

#### Exercise 4. Semimajor and semiminor ellipsoid axes

Set the Clarke 1866 ellipsoid by its semimajor and semiminor axis radii

```
operation   +proj=merc  <your answer here>
tolerance   1 cm

accept      24.745          59.437          # Talinn
expect      2754630.7723    8238298.4968
```

### 13.1.3.4 Gridshifting

Sometimes it is simpler to define a transformation as a gridded model of displacements rather than a mathematical model. This approach, pioneered by the US National Geodetic Survey in their 1980s era transformation from the old NAD27 datum to the (back then) new NAD83, has become increasingly popular as disk space and CPU cycles have gotten cheaper.

Gridded models of displacements offer the ability to make local adjustments that are not possible with e.g. a Helmert transformation. This fact is especially leveraged in transformations of heights since the geoid is too bumpy to retain enough detail with a simple mathematical definition.

The main problem with grid shifting is availability of the grids needed for a given transformation. The PROJ package requires that the *PROJ-data package* is installed alongside it.

Download the *gie* file for the exercise: `gridshift.gie`.

#### Exercises 1. Horizontal gridshifting

In this exercise we will apply a horizontal grid shift as it is done in the transformation from the German DHDN to ETRS89 transformation.

Define the operation that converts from DE_DHDN to ETRS89 using a gridshift.

**Hint**

See *Horizontal grid shift*

**Hint**

Find the relevant grid name by inspecting the output of

```
projinfo -s DHDN -t ETRS89 -o WKT2_2019
```

Note that two operations are returned, look for the one which uses the "NTv2" method.

```
operation   <your answer here>
tolerance   1 mm

accept      13.0            52.0            0.0
expect      12.9983317082   51.9986488216   0.0
```

### 2. Vertical gridshifting

The most common use case for vertical grid shifts is transformation from ellipsoidal heights to physical heights. In most cases this is equivalent to applying an offset from a geoid model, which is exactly what we will do in this exercise.

Set up an operation that transforms ellipsoidal heights to physical heights using the EGM96 model.

**Hint**

See *Vertical grid shift*

**Hint**

The relevant grid name is "us_nga_egm96_15.tif"

```
operation   <your answer here>
tolerance   1 mm

accept      13.0            52.0          100.0
expect      13.0            52.0           58.1917
```

#### 13.1.3.5 The Helmert transformation

In this set of exercises we investigate the Helmert Transformation and some of its properties. The Helmert transformation covers a wide range of configurations, from simple translations of coordinates to the fully-fledged spatiotemporal transformation. Consult Wikipedia and *Helmert transform* for in-depth technical details on the transformation.

The Helmert transformation works on geocentric, cartesian coordinates. See the *Coordinate conversions* exercise for more info. In this exercise all coordinates are already given as cartesian coordinates, so you only need to focus on the actual transformation setup.

Download the *gie* file for the exercise: `helmert.gie`.

### 1. Simple coordinate translation

The most basic application of the Helmert transformation is a translation of the x-, y- and z-components of the coordinate. This is rarely a good fit for large areas but locally a 3 parameter Helmert shift can be very effective. Even though the 3 parameter translation doesn't fit particularly well in large areas of use it is still commonly used between legacy and modern systems. In this exercise we will use a transformation between ED50 in Italy (Sardinia) and WGS84.

**Hint**

You can find the transformations parameters by running the following command:

```
projinfo -o WKT2_2019 -k operation EPSG:1142
```

which returns the WKT2 definition of the transformation.

**Hint**

Remember that the input to the Helmert transform is cartesian geocentric coordinates (x, y z).

**Hint**

See *Helmert transform* to find out how to set up the transformation

```
operation   <your answer here>
tolerance   1 m

accept      4826177.7574   4049643.9762   991162.2529  # Sardinia (40N, 9E)
expect      4826080.7574   4049540.9762   991042.2529
```

### 2. Translation and rotation

For larger areas it is not enough to just rely on translation of coordinates. In that case the solution is to include rotations of the axes and a scale adjustment. The rotational aspect of the transformation is handled by a set of three rotation matrices. Two conventions for the application of the rotations are in common use: Position Vector and Coordinate Frame. In this exercise we will using Position Vector. See the PROJ documentation on the Helmert transformation for further detail.

**Hint**

You can find the transformations parameters by running the following command:

```
projinfo -o WKT2_2019 -k operation EPSG:1626
```

**Hint**

Remember to specify the rotation convention with +convention

```
operation   <your answer here>
tolerance   1 m

accept      3496723.5936    743251.5442  5264442.2361
expect      3496639.7297    743156.3657  5264324.9341
```

### 3. Position vector/Coordinate frame

Effectively, the difference between the two rotation conventions is the sign of the rotation parameters. In this exercise we will examine this property by looking at a transformation between the German DHDN reference system and ETRS89.

First find the parameters for the EPSG:1309 transformation. In 3a below, enter the parameters as reported by projinfo (using the coordinate frame convention). Adapt the parameters to the position vector convention in 3b.

**Hint**

You can find the transformations parameters by running the following command:

```
projinfo -o WKT2_2019 -k operation EPSG:1309
```

```
* 3a:
operation   <your answer here> +convention=coordinate_frame

accept      4067886.6403    571704.1839  4862789.0376
expect      4068519.1921    571728.6671  4863239.3787

* 3b:
operation   <your answer here> +convention=position_vector
tolerance 1 m

accept      4067886.6403    571704.1839  4862789.0376
expect      4068519.1921    571728.6671  4863239.3787
```

### 4. Kinematic transformation between ITRF2008 and ITRF2014

The Helmert transformation also exists in a kinematic, or spatiotemporal, version. This takes the evolution of a coordinate reference system over time into account as well. This is done by pivoting about a reference epoch. The 14-parameter spatiotemporal Helmert is used when super high accuracy transformations are needed. A good example of such a use case is transformations between various realizations of ITRS and ETRS89. In this exercise we look at the transformation from ITRF2008 to ITRF2014.

**Hint**

You can find the transformations parameters by running the following command:

```
projinfo -o WKT2_2019 -s ITRF2008 -t ITRF2014
```

which returns the WKT2 definition of the transformation.

**Hint**

Not all parameters need to be set.

**Hint**

Pay attention to the units of the parameters output by projinfo, you may have to convert them to standard units (e.g. mm -> m).

**Hint**

Note that the coordinates in the test now also includes a time tag in the form of a decimalyear - This is always needed when doing spatiotemporal transformations in PROJ. The time tag is the observation time of the coordinate. Note that the time component of the coordinate is never affected by the transformation.

```
operation   <your answer here>
tolerance   1 mm
```

(continues on next page)

(continued from previous page)

```
accept      2952736.3768   1360917.6894  5468849.5615       2019.5
expect      2952736.3744   1360917.6871  5468849.5586       2019.5
```

#### 13.1.3.6 Coordinate conversions

In ISO19111 terms, a coordinate conversion is an operation that changes coordinates in a source coordinate reference system to coordinates in a target coordinate reference system in which both coordinate reference systems are based on the same datum. Technically this is also true for PROJ although the term is usually narrowed a bit, excluding projections since those are the origin of PROJ they get special treatment.

In this exercise we will be looking at three coordinate conversions that in combination with other operations are very powerful: Unit conversions, axis swapping and geodetic to cartesian conversion. Here we only look at the conversions by themselves which isn't particularly useful, apart from the educational purpose. When used in transformation pipelines they reveal their true power. We will explore that further in the pipelines.gie exercises.

See *Conversions* for a list of all conversions available in PROJ.

Download the *gie* file for the exercise: `conversions.gie`.

#### 1. Unit conversion from meters to feet

Many projected coordinate systems are defined in terms of units other than the meter. In this exercise we will transform the horizontal part of a coordinate from meters to feet.

**Hint**

Consult *Unit conversion* to find out how to set up the transformation

**Hint**

You can use `proj -lu` to learn which units is supported by PROJ.

**Hint**

Note that the horizontal, vertical and temporal parts are treated separately by the unit convert operator.

```
operation   <your answer here>
tolerance   1 mm

accept      100.0       123.0       432.0
expect      328.0833    403.5425    432.0
```

#### 2. Swapping axes

Many coordinate reference systems are defined such that the axis order is different than the (east, north, up, time) PROJ defaults to. An example of this is the standard representation of a latitude/longitude-pair, where the north component of the coordinate comes first. For this reason we need an operation that can swap the axes around so that a given coordinate reference system can be represented correctly, conforming to the intention of the defining authority.

Set up an operation that puts a longitude/latitude pair on the standard latitude/longitude form.

**Hint**

Consult *Axis swap* to find out how to set up the transformation

**Hint**

Remember that the axisswap operation doesn't know anything about the nature of the coordinate that is passed to it - it only cares about the the order of the input.

```
operation   <your answer here>
tolerance   1 mm

accept      140.0     75.0    # somewhere in Siberia
expect       75.0    140.0
```

### 3. Geodetic to cartesian conversion

Some transformations, most notably the Helmert transformation, operate on cartesian geocentric coordinates. Geodetic coordinates (latitude and longitude) is the most commonly used coordinate representation. It is only natural to have a way to convert between the two representations.

Set up a transformation that convert geodetic coordinates to cartesian geocentric coordinates on the Hayford ellipsoid.

**Hint**

Consult *Geodetic to cartesian conversion* to find out how to set up the transformation

**Hint**

Remember that *proj -le* returns a list of available ellipsoid models

**Hint**

Remember that the Hayford ellipsoid is known under a number of other names - most of them including the term "international".

```
operation   <your answer here>
tolerance   1 mm

accept      24.745          59.437          0  # Talinn, Estonia
expect      2952883.7000    1360985.5908    5468966.6589
```

#### 13.1.3.7 Transformation pipelines

In these exercises we will learn how to use Transformation Pipelines. Transformation Pipelines is a powerful construct in PROJ that allow users to combine several operations into one "super operation". Pipelines are used extensively internally in PROJ when a user asks for a transformation between two CRS's (for example when using *cs2cs*). However, users can also write their own custom pipelines for specific purposes.

Before moving on to the exercises below, familiarize yourself with the *pipeline operation in the documentation*. Pay special attention to the rules that govern the use of the pipeline operation.

Download the *gie* file for the exercise: `pipelines.gie`.

##### 1. Geodetic -> Helmert -> Geodetic

The signature use case for transformation pipelines is the Helmert transformation of geodetic coordinates.

As we have seen in the previous exercises, fundamentally, the Helmert operation works on cartesian coordinates. So when the input coordinates are geodetic, we must convert them to the cartesian space before applying the Helmert transformation.

Similarly, we must convert the transformed cartesian coordinates back to geodetic space to be able to compare the coordinates before and after transformation.

With that in mind, now create a transformation pipeline that take geodetic (longitude/latitude) coordinates, applies a Helmert transform, and returns geodetic coordinates (in a different datum).

To keep it simple we will use the GRS80 ellipsoid for both input and output datum, and a basic 3 parameter transformation, with translations 100,200,300, and using the position vector convention.

**Hint**

Remember that in a pipeline step, the inverse of an operation is selected by adding `+inv` to the parameter list of the operation.

```
operation   <your answer here>
tolerance   1 m

accept      12.0           53.0          75.0
expect      12.00260406    53.00062190  398.48468
```

##### 2. Sequential grid shifts

Create a transformation that applies both a horizontal and a vertical grid adjustment. Use the `de_adv_BETA2007.tif` and the `us_nga_egm96_15.tif` grids.

1. Does it matter which grid is applied first?
2. If yes, which goes first? Why?

**Hint**

See *Horizontal grid shift* and *Vertical grid shift*

```
operation   <your answer here>
tolerance   1 cm

accept      12.0            53.0        75.0
expect      11.99846399     52.99852627 34.541488647
```

### 3. ED50/UTM32 -> ETRS89/UTM33

A transformation from one coordinate reference system to another. A very common use case of transformation pipelines.

In this exercise we have UTM Zone 32 coordinates related to the ED50 datum, which we want to transform to UTM Zone 33 related to ETRS89.

The transformation goes

1. Back-projection
2. Geodetic to cartesian conversion
3. Helmert transformation
4. Cartesian to geodetic conversion
5. Re-projection

which is a very common recipe for transformations between projected coordinate systems.

**Hint**

Find the relevant Helmert parameters by running:

```
projinfo -k operation EPSG:1626 -o WKT2_2019
```

**Hint**

If you can't remember which ellipsoids is used by ED50 and ETRS89 you can look them up with:

```
projinfo <datum>
```

**Hint**

Relevant documentation:

1. *Universal Transverse Mercator (UTM)*
2. *Geodetic to cartesian conversion*
3. *Helmert transform*

```
operation   <your answer here>
tolerance   1 cm
```

(continues on next page)

(continued from previous page)

```
accept      687080.63      6210278.55   0
expect      312871.16      6210214.58   34.08
```

#### 4. A pipeline from the Real World

The transformation between the German DHDN and ETRS89 is a transformation pipeline consisting of eight steps. Lookup the transformation with projinfo:

```
projinfo -k operation EPSG:1776
```

and copy and paste the pipeline definition as your answer below. Try to decipher the pipeline and answer the following questions:

1. What is the input coordinate type and units?
2. What is the output coordinate type and units?
3. What is the purpose of the +proj=push and +proj=pop steps?

**Hint**

A pipeline definition can span several lines in the gie format. Use this to your advantage and put each step on it's own separate line for easier reading

**Hint**

You can use projinfo to get more knowledge about a particular CRS:

```
projinfo <CRS name or code>
```

**Hint**

Look at *Push coordinate value to pipeline stack* and *Pop coordinate value to pipeline stack* to find out how to set up the transformation

```
operation   <your answer here>
tolerance   3 m

accept      12.0           53.0          75.0
expect      12.0032        52.9953       75.0000
```

### 13.1.3.8 The Transverse Mercator projection

In this exercise we will introduce the Transverse Mercator projection, the UTM projection and the relationship between the two.

In addition we will investigate the differences between the two Transverse Mercator implementations available in PROJ and when one should be used in favour of the other.

Download the *gie* file for the exercise: `projections3.gie`.

### Exercise 1. Transverse Mercator with default parameters

Set up a Transverse Mercator projection using the default parameters.

**Hint**

Consult *Transverse Mercator*

```
operation   <your answer here>
tolerance   1 mm

accept      24.745          56.437       # Talinn, Estonia
expect      1506742.2481    6535299.3398
```

### Exercise 2: Use the Transverse Mercator to model the UTM projection

The backbone of the UTM projection is a Transverse Mercator projection. In this exercise we will model the behavior of the UTM projection using the Transverse Mercator.

**Hint**

Remember that the UTM projection on the Northern Hemisphere uses a scale factor of 0.9996, a false easting of 500000 and a false northing of 0. UTM on the Southern Hemisphere is similar but with a false northing of 10000000.

**Hint**

The projection center is determined from the UTM zone and can be determined by zone*6 - 183

**Hint**

Transverse Mercator parameters are documented at *Transverse Mercator*

```
operation  <your answer here>
tolerance   1 mm

accept      24.745          56.437       # Talinn, Estonia
expect      360965.5942     6256998.5609
```

### Exercise 3: The less accurate, but faster, version of the Transverse Mercator

As mentioned in the introduction to this set of exercises, two versions of the Transverse Mercator is implemented in PROJ. The default uses the Engsager/Poder algorithm which is accurate far away from the central meridian of the projection. The downside to this accuracy is that the algorithm is slower. The alternative algorithm, which is toggled by the +approx parameter, is faster but usage is recommended only within a few degrees away from the central meridian.

In this and the following exercises we will explore the accuracy of the two algorithms by checking the roundtrip stability of a number of transformations. A coordinate in Greenland will be used, as it is common practice to store geospatial

data covering the whole country in the same UTM zone. This is only possible when using the correct algorithm. For the sake of simplicity, all operations in the following exercises are expressed as UTM projections. The UTM projection also has the +approx parameter which toggles the use of the faster, less accurate transverse mercator algorithm.

We will try to determine the approximate roundtrip accuracy of the +approx algorithm several UTM zones away from the actual zone for the given coordinate. For all the exercises below the aim is to find the lowest tolerance for each roundtrip test. You can of course make all tests pass by setting a tolerance of 1000 km - that's not the point: How low can you go?

After you have answered all exercise 3 questions below, based on your findings consider in which situation use of each of the algorithms is appropriate.

**Hint**

*gie* accepts most common SI unit prefixes to the meter when specifying the tolerance, e.g. km, m, dm, cm, mm, um, nm.

**Hint**

Look at the output *gie* produces - the difference between the actual result and the expected result is reported when tests fail.

### Exercise 3a

As a baseline, determine the roundtrip accuracy of the default algorithm using UTM zone 22.

```
operation   +proj=utm +zone=22
tolerance   <your answer here>  # Hint: You can go *very* low here


accept      -20.0   74      # Daneborg, Greenland (~30 degrees from central meridian)
roundtrip   1000
```

### Exercise 3b

Determine the roundtrip accuracy of the +approx algorithm using UTM zone 22:

```
operation   +proj=utm +zone=22 +approx
tolerance   <your answer here>

accept      -20.0   74      # Daneborg, Greenland (~30 degrees from central meridian)
roundtrip   1000
```

### Exercise 3c

Determine the roundtrip accuracy of the +approx algorithm using UTM zone 23.

```
operation   +proj=utm +zone=23 +approx
tolerance   <your answer here>

accept      -20.0   74      # Daneborg, Greenland
roundtrip   1000
```

#### Exercise 3d

Determine the roundtrip accuracy of the +approx algorithm using UTM zone 24:

```
operation   +proj=utm +zone=24 +approx
tolerance   <your answer here>

accept      -20.0   74      # Daneborg, Greenland
roundtrip   1000
```

#### Exercise 3e

Determine the roundtrip accuracy of the +approx algorithm using UTM zone 25:

```
operation   +proj=utm +zone=25 +approx
tolerance   <your answer here>

accept      -20.0   74      # Daneborg, Greenland
roundtrip   1000
```

#### Exercise 3f

Determine the roundtrip accuracy of the +approx algorithm using UTM zone 26:

```
operation   +proj=utm +zone=26 +approx
tolerance   <your answer here>

accept      -20.0   74      # Daneborg, Greenland
roundtrip   1000
```

#### Exercise 3g

Determine the roundtrip accuracy of the +approx algorithm using UTM zone 27:

```
operation   +proj=utm +zone=27 +approx
tolerance   <your answer here>

accept      -20.0   74      # Daneborg, Greenland
roundtrip   1000
```

### 13.1.3.9 Answers to the PROJ tutorial exercises

#### Projections1: UTM

```
1.      +proj=utm +zone=35
2.      +proj=utm +zone=60 +south
3.      +proj=utm +zone=35 +ellps=intl
```

#### Projections2: LCC

```
1.      +proj=lcc +lat_1=59.8
2.      +proj=lcc +lat_1=59.8   +k_0=0.99998
```

(continues on next page)

(continued from previous page)

```
3.      +proj=lcc +lat_1=59.437 +lat_2=60.171
4.      +proj=lcc +lat_1=59.8   +lat_0=59.8   +lon_0=24.8
5.      +proj=lcc +lat_1=59.8   +lat_0=59.8   +lon_0=24.8  +x_0=1000000 +y_0=1000000
```

### Ellipsoids

```
1.      +R=1
2.      +ellps=intl
3.      +a=6378388.0 +rf=297.0
4.      +a=6378206.4 +b=6356583.8
```

### Helmert

```
1.      +proj=helmert +x=-97 +y=-103 +z=-120
2.      +proj=helmert +x=-81.1 +y=-89.4 +z=-115.8 +rx=0.485 +ry=0.024 +rz=0.413 +s=-0.54␣
↪+convention=position_vector
3a.     +proj=helmert +x=582 +y=105 +z=414 +rx=-1.04 +ry=-0.35 +rz=3.08 +s=8.3
3b.     +proj=helmert +x=582 +y=105 +z=414 +rx=1.04 +ry=0.35 +rz=-3.08 +s=8.3
1.      +proj=helmert   +convention=position_vector +x=-0.0016  +y=-0.0019  +z=-0.0024␣
↪+s=2e-05 +dz=0.0001 +ds=-3e-05 +t_epoch=2010
```

### Conversions

```
1.      +proj=unitconvert +xy_in=m  +xy_out=us-ft
2.      +proj=axisswap +order=2,1
3.      +proj=cart +ellps=intl
```

### Gridshifts

```
1.      +proj=hgridshift +grids=de_adv_BETA2007.tif
2.      +proj=vgridshift +grids=us_nga_egm96_15.tif
```

### Pipelines

```
1.      +proj=pipeline +ellps=GRS80
        +step +proj=cart
        +step +proj=helmert +x=100 +y=200 +z=300 +convention=position_vector
        +step +inv +proj=cart

2.      +proj=pipeline
        +step +proj=hgridshift +grids=de_adv_BETA2007.tif
        +step +proj=vgridshift +grids=us_nga_egm96_15.tif

2.1:    In principle, yes - practically, no.
2.2:    Which grids is applied first depends on how the systems are defined.
        If a geoid model is fitted to a certain datum, you need to make sure that
        the input to the vgridshift operation is in that datum. That is your
        responsibility to check - PROJ can't figure that out on its own. In this
```

(continues on next page)

(continued from previous page)

```
        case, the EGM96 geoid is fitted to WGS84. The BETA2007 grid results in
        ETRS89 coordinates which for all but the most high accuracy application
        can be regarded as equivalent to WGS84. Hence we do the horizontal grid
        shift first.

3.      +proj=pipeline
        +step +inv +proj=utm +zone=32
        +step +proj=cart +ellps=intl
        +step +proj=helmert +x=-81.1 +y=-89.4 +z=-115.8
        +rx=0.485 +ry=0.024 +rz=0.413 +s=-0.54 +convention=position_vector
        +step +inv +proj=cart +ellps=GRS80
        +step +proj=utm +zone=33


4.      +proj=pipeline
        +step +proj=axisswap +order=2,1
        +step +proj=unitconvert +xy_in=deg +xy_out=rad
        +step +proj=push +v_3 +step +proj=cart +ellps=bessel
        +step +proj=helmert +x=598.1 +y=73.7 +z=418.2
        +rx=0.202 +ry=0.045 +rz=-2.455 +s=6.7 +convention=position_vector
        +step +inv +proj=cart +ellps=GRS80
        +step +proj=pop +v_3
        +step +proj=unitconvert +xy_in=rad +xy_out=deg
        +step +proj=axisswap +order=2,1

4.1:    Latitude (deg), longitude (deg), height (m)
4.2:    Latitude (deg), longitude (deg), height (m)
4.3:    Retain the original height of the input coordinate:
        Without the push/pop steps the height would be affected by
        the Helmert step.
```

#### Projections3: Transverse Mercator

```
1.      +proj=tmerc
2.      +proj=tmerc +k_0=0.9996 +lon_0=27 +x_0=500000
3a.     1 cm
3b.     250 km
3c.     100 km
3d.      10 km
3e.     500  m
3f.       2  m
3g.       5 um
```

## 13.2 Geodesics and planar approximations using PROJ and geod

The PROJ system supports computations with *Geodesic calculations*, through API calls or through the *geod* command line interface. This exercise will proceed mostly by having you copy sample commands from the text onto the command line, and noting the results.

The tutorial is written with the Windows command line in mind but it should be straight forward to use it on Unix-based systems as well.

**Hint**

Users of Unix-based systems can replace `set` with `export` and `%VAR%` with `$VAR` to follow the tutorial.

### 13.2.1 1. Objective

The objective of this exercise is to investigate how well a straight line in a projected plane approximates the geodesic between the same end points.

### 13.2.2 2. The geodesic

The geodesic we will consider runs from Helsinki to Tallinn, using the following coordinates:

```
Helsinki  60.171 N 24.938 E
Tallinn   59.437 N 24.745 E
```

So to save some typing, let's define some environment variables. On Windows:

```
set HEL="60.171  24.938"
set TAL="59.437  24.745"
```

```
export HEL="60.171  24.938"
export TAL="59.437  24.745"
```

(i.e. mark, copy, and paste these lines into the command prompt)

First, we want to solve the "inverse geodetic problem" for the geodesic, i.e. finding the azimuth and distance between the two points:

```
echo %HEL% %TAL% | geod -I +ellps=GRS80
```

where the `-I` option indicates "the inverse geodetic problem".

*geod* replies with these 3 numbers:

```
-172d22'12.772"    7d27'46.693"    82488.500
```

The first is the forward azimuth from Helsinki to Tallinn, the second is the return azimuth, while the third is the distance in meters.

We would actually rather have the azimuths in fractional degrees, so we provide an explicit output format, using the `-f` option to geod:

```
echo %HEL% %TAL% | geod -I -f "%.12f" +ellps=GRS80
```

getting us:

```
-172.370214337896    7.462970382137    82488.500
```

We assert that the planar approximation is worst at the midpoint of the geodesic, i.e. `(82488.5 m) / 2 = 41244.25 m`, from Helsinki. So now, we want to find the coordinates of this point, by solving the "direct geodetic problem":

```
echo %HEL%  -172.370214337896    41244.25 | geod -f "%.12f" +ellps=GRS80
```

resulting in:

```
59.804045696236    24.840439590098    7.545306054713
```

i.e. the latitude, the longitude, and the return azimuth of the mid point of the geodesic.

Now, for convenience, define this environment variable:

```
set MID="59.804045696236  24.840439590098"
```

**Note**

You can save a bit of manual copying by capturing the geod output into the clipboard, using the "clip" command on Windows, i.e. by appending `| clip`

to the `geod` command above.

### 13.2.3 1. The planar approximation

We will be working on ETRS89/UTM35 coordinates, so first define an environment variable, to save some typing:

```
set utm35=proj -rs +proj=utm +zone=35 +ellps=GRS80
```

**Note**

The `-rs` option switches the proj coordinate i/o order to latitude/longitude and northing/easting, respectively. We do this to comply with the expected order for the geod program.

Now, find the UTM coordinates of the two end points:

```
echo %HEL% | %utm35%
echo %TAL% | %utm35%
```

Note your results and compute the mean of the northings, and the mean of the eastings, to obtain the planar approximation of the mid point

**Hint**

you can use python as a makeshift command line calculator by saying:

```
python -c print((4+8)/2)
```

Now, compute the geographical coordinates corresponding to the UTM mid point:

```
echo <your northing   your easting> | %utm35% -f "%.12f" -I
```

(note the -I option for doing the inverse projection)

For convenience define:

```
set MID_APPROX=<your result here>
```

Finally, compute the distance between the geodesic mid point and its planar approximation, by stating it as another case of the inverse geodetic problem:

```
echo %MID%  %MID_APPROX% | geod -I +ellps=GRS80
```

resulting in:

```
107d0'16.673"    -72d59'43.193"    2.527
```

i.e. a deviation of 2.5 m over a stretch of about 85 km.

### 13.2.4 4. Suggested meditations

Consider these aspects:

1. Given that geod is available, fast, and reliable - is it really worth the effort doing approximate calculations in the projected plane?
2. *geod* includes functionality for computing intermediate points along a geodesic. Check the manual, especially the description of the `n_S` option, and try to compute the geodesic mid point directly, by setting `n_S=2`.
3. One should actually expect the result of meditation 2 above to be slightly superior to the result obtained in the exercise. Why?

### 13.2.5 Answers

```
Helsinki,  UTM:  6672241.54   385592.95
Tallinn,   UTM:  6590881.40   372106.37
Mid point, UTM:  6631561.47   378849.66
Mid point, GEO:  59.804039062602   24.840482644156
```

Meditations:

1. Probably not
2. `geod +lat_1=60.171 +lon_1=24.938 +lat_2=59.437 +lon_2=24.745 +n_S=2 +ellps=GRS80 -f "%.12f"`
3. Avoids a bit of truncation of the azimuth, and *may* use a slightly superior algorithm (check the code!)

CHAPTER

# FOURTEEN

# FAQ

## 14.1 Which file formats does PROJ support?

The *command line applications* that come with PROJ only support text input and output (apart from *proj* which accepts a simple binary data stream as well). *proj*, *cs2cs* and *cct* expects text files with one coordinate per line with each coordinate dimension in a separate column.

**Note**

If your data is stored in a common geodata file format chances are that you can use GDAL as a frontend to PROJ and transform your data with the ogr2ogr application.

## 14.2 Can I transform from *abc* to *xyz*?

Probably. PROJ supports transformations between most coordinate reference systems registered in the EPSG registry, as well as a number of other coordinate reference systems. The best way to find out is to test it with the *projinfo* application. Here's an example checking if there's a transformation between ETRS89/UTM32N (EPSG:25832) and ETRS89/DKTM1 (EPSG:4093):

```
$ ./projinfo -s EPSG:25832 -t EPSG:4093 -o PROJ
Candidate operations found: 1
-------------------------------------
Operation No. 1:

unknown id, Inverse of UTM zone 32N + DKTM1, 0 m, World

PROJ string:
+proj=pipeline
  +step +inv +proj=utm +zone=32 +ellps=GRS80
  +step +proj=tmerc +lat_0=0 +lon_0=9 +k=0.99998 +x_0=200000 +y_0=-5000000
        +ellps=GRS80
```

See the *projinfo* documentation for more info on how to use it.

## 14.3 Coordinate reference system *xyz* is not in the EPSG registry, what do I do?

Generally PROJ will accept coordinate reference system descriptions in the form of WKT, WKT2 and PROJ strings. If you are able to describe your desired CRS in either of those formats there's a good chance that PROJ will be able to

make sense of it.

If it is important to you that a given CRS is added to the EPSG registry, you should contact your local geodetic authority and ask them to submit the CRS for inclusion in the registry.

## 14.4 I found a bug in PROJ, how do I get it fixed?

Please report bugs that you find to the issue tracker on GitHub. *Here's how*.

If you know how to program you can also try to fix it yourself. You are welcome to ask for guidance on one of the *communication channels* used by the project.

## 14.5 How do I contribute to PROJ?

Any contributions from the PROJ community is welcome. See *Contributing* for more details.

## 14.6 How do I calculate distances/directions on the surface of the earth?

These are called geodesic calculations. There is a page about it here: *Geodesic calculations*.

## 14.7 What is the best format for describing coordinate reference systems?

A coordinate reference system (CRS) can in PROJ be described in several ways: As PROJ strings, Well-Known Text (WKT) and as spatial reference ID's (such as EPSG codes). Generally, WKT or SRID's are preferred over PROJ strings as they can contain more information about a given CRS. Conversions between WKT and PROJ strings will in most cases cause a loss of information, potentially leading to erroneous transformations.

For compatibility reasons PROJ supports several WKT dialects (see `projinfo -o`). If possible WKT2 should be used.

## 14.8 Which CRS apply to a given location?

You can use the webpage CRS Explorer to view a list of all coordinate reference systems in *proj.db*, and filter by type, authority, name and location (clicking on the map). It provides WKTs for every coordinate reference system and quick links to epsg.org.

## 14.9 Why is the axis ordering in PROJ not consistent?

PROJ respects the axis ordering as it was defined by the authority in charge of a given coordinate reference system. This is in accordance to the ISO19111 standard [ISO19111]. Unfortunately most GIS software on the market doesn't follow this standard. Before version 6, PROJ did not respect the standard either. This causes some problems while the rest of the industry conforms to the standard. PROJ intends to spearhead this effort, hopefully setting a good example for the rest of the geospatial industry.

Customarily in GIS the first component in a coordinate tuple has been aligned with the east/west direction and the second component with the north/south direction. For many coordinate reference systems this is also what is defined by the authority. There are however exceptions, especially when dealing with coordinate systems that don't align with the cardinal directions of a compass. For example it is not obvious which coordinate component aligns to which axis in a skewed coordinate system with a 45 degrees angle against the north direction. Similarly, a geocentric cartesian coordinate system usually has the z-component aligned with the rotational axis of the earth and hence the axis points

towards north. Both cases are incompatible with the convention of always having the x-component be the east/west axis, the y-component the north/south axis and the z-component the up/down axis.

In most cases coordinate reference systems with geodetic coordinates expect the input ordered as latitude/longitude (typically with the EPSG dataset), however, internally PROJ expects an longitude/latitude ordering for all projections. This is generally hidden for users but in a few cases it is exposed at the surface level of PROJ, most prominently in the *proj* utility which expects longitude/latitude ordering of input data (unless `proj -r` is used). (And *geod*, which on the contrary expects latitude/longitude, and lacks any *-r* style workarounds.)

In case of doubt about the axis order of a specific CRS *projinfo* is able to provide an answer. Simply look up the CRS and examine the axis specification of the Well-Known Text output:

```
projinfo EPSG:4326
PROJ.4 string:
+proj=longlat +datum=WGS84 +no_defs +type=crs

WKT2:2019 string:
GEOGCRS["WGS 84",
    DATUM["World Geodetic System 1984",
        ELLIPSOID["WGS 84",6378137,298.257223563,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["geodetic latitude (Lat)",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["geodetic longitude (Lon)",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    USAGE[
        SCOPE["unknown"],
        AREA["World"],
        BBOX[-90,-180,90,180]],
    ID["EPSG",4326]]
```

## 14.10 Why am I getting the error "Cannot find proj.db"?

The file *proj.db* must be readable for the library to properly function. Like other *resource files*, it is located using a set of search paths. In most cases, the following paths are checked in order:

- A path provided by the environment variable *PROJ_DATA* (called `PROJ_LIB` before PROJ 9.1)
- A path built into PROJ as its resource installation directory (typically ../share/proj relative to the PROJ library).
- The current directory.

Note that if you're using conda, activating an environment sets *PROJ_DATA* to a resource directory located in that environment.

## 14.11 What happened to PROJ.4?

The first incarnation of PROJ saw the light of day in 1983. Back then it was simply known as PROJ. Eventually a new version was released, known as PROJ.2 in order to distinguish between the two versions. Later on both PROJ.3 and PROJ.4 was released. By the time PROJ.4 was released the software had matured enough that a new major version

release wasn't an immediate necessity. PROJ.4 was around for more than 25 years before it again became time for an update. This left the project in a bit of a conundrum regarding the name. For the majority of the life-time of the product it was known as PROJ.4, but with the release of version 5 the name was no longer aligned with the version number. As a consequence, it was decided to decouple the name from the version number and once again simply call the software PROJ.

Use of name PROJ.4 is now strictly reserved for describing legacy behavior of the software, e.g. "PROJ.4 strings" as seen in *projinfo* output.

## 14.12 Who uses PROJ ?

See *Software and organizations using or contributing to PROJ*

### 14.12.1 Software and organizations using or contributing to PROJ

#### 14.12.1.1 Free and open source software

- GDAL Translator library for raster and vector geospatial data formats, used by > 100 other software
- GRASS GIS A raster/vector open source GIS
- libgeotiff Library to read/write GeoTIFF keys
- MapServer A popular web mapping application
- Mergin Maps Field data collection app built on top of QGIS with cloud sync
- PDAL Point Cloud Data Abstraction Library
- PostGIS Spatial database extender for PostgreSQL
- QGIS A cross platform desktop GIS.
- QField Mobile data collection app for Android, iOS and Windows.
- R A free software environment for statistical computing and graphics
- mundipy A Python framework for spatial data analysis

#### 14.12.1.2 PROJ bindings in other languages

See *Language bindings*

#### 14.12.1.3 Proprietary licensed software

- FME A GIS translator package. Includes a PROJ based transformer.
- Avenza Maps Top mobile mapping app for offline navigation and data collection.
- Bunting Labs No-code GIS infrastructure
- Pix4D Cloud, desktop and mobile photogrammetry processing and data capture software.

#### 14.12.1.4 Geodetic organizations

- Agency for Data Supply and Infrastructure of Denmark (SDFI)
- Land Information New Zealand (LINZ)
- National Land Survey of Finland (NLS)
- National Land Survey of Iceland (NLS)
- Netherlands Partnership Geodetic Infrastructure (NSGI)

  - Netherlands' Cadastre, Land Registry and Mapping Agency (Kadaster)
  - Directorate-General for Public Works and Water Management (Rijkswaterstaat)
  - Hydrographic Service of the Royal Netherlands Navy
- Nordic Geodetic commission (NKG)
- Norwegian Mapping Authority (Kartverket)
- Swedish mapping, cadastral and land registration authority (Lantmäteriet)
- USGS: PROJ initial author, Gerald E. Evenden, worked at USGS

#### 14.12.1.5 Public organizations

- Istituto Nazioanle di Astrofisica (INAF) : Planetary Mapping and GIS Laboratory
- Ecole Nationale des Sciences Géographiques (ENSG-géomatique) : French geomatic school which depends on the National Institute of Geographic and Forest Information (IGN)

(add yourself here if you are a direct user of PROJ API or utilities)

#### 14.12.1.6 Private organizations

(add yourself here if you are a direct user of PROJ API or utilities)

#### 14.12.1.7 Sponsors

PROJ development, as an upstream dependency of GDAL, benefits from funding brought by its sponsors

CHAPTER

# FIFTEEN

# GLOSSARY

**Ballpark transformation**

For a transformation between two geographic CRS, a ballpark transformation is a coordinate operation that only takes into account potential difference of axis orders (long-lat vs lat-long), units (degree vs grads) and prime meridian (Greenwich vs Paris/Rome/other historic prime meridians). It does not attempt any datum shift, hence the "ballpark" qualifier in its name. Its accuracy is unknown, and could lead in some cases to errors of a few hundreds of metres.

For a transformation between two vertical CRS or a vertical CRS and a geographic CRS, a ballpark transformation only takes into account potential different in units (e.g. metres vs feet). Its accuracy is unknown, and could lead in some cases to errors of a few tens of metres.

> **Note**
>
> The term "Ballpark transformation" is specific to PROJ.

**Pseudocylindrical Projection**

Pseudocylindrical projections have the mathematical characteristics of

$$x = f(\lambda, \phi)$$
$$y = g(\phi)$$

where the parallels of latitude are straight lines, like cylindrical projections, but the meridians are curved toward the center as they depart from the equator. This is an effort to minimize the distortion of the polar regions inherent in the cylindrical projections.

Pseudocylindrical projections are almost exclusively used for small scale global displays and, except for the Sinusoidal projection, only derived for a spherical Earth. Because of the basic definition none of the pseudocylindrical projections are conformal but many are equal area.

To further reduce distortion, pseudocylindrical are often presented in interrupted form that are made by joining several regions with appropriate central meridians and false easting and clipping boundaries. Interrupted Homolosine constructions are suited for showing respective global land and oceanic regions, for example. To reduce the lateral size of the map, some uses remove an irregular, North-South strip of the mid-Atlantic region so that the western tip of Africa is plotted north of the eastern tip of South America.

# BIBLIOGRAPHY


[Altamimi2002] Altamimi, Z., Sillard, P., and Boucher, C. ITRF2000: a new release of the International Terrestrial Reference Frame for earth science applications. *Journal of Geophysical Research: Solid Earth*, 2002. doi:10.1029/2001JB000561.

[Bessel1825] Bessel, F. W. The calculation of longitude and latitude from geodesic measurements. *Astronomische Nachrichten*, 4(86):241–254, 1825. arXiv:0908.1824.

[CalabrettaGreisen2002] Calabretta, M. R. and Greisen, E. W. Representations of celestial coordinates in FITS. *Astronomy & Astrophysics*, 395(3):1077–1122, 2002. doi:10.1051/0004-6361:20021327.

[ChanONeil1975] Chan, F. K. and O'Neill, E. M. Feasibility study of a quadrilateralized spherical cube earth data base. Tech. Rep. EPRF 2-75 (CSC), Computer Sciences Corporation, System Sciences Division, Silver Spring, Md, 1975. URL: https://archive.org/details/ADA010232.

[Danielsen1989] Danielsen, J. The area under the geodesic. *Survey Review*, 30(232):61–66, 1989. doi:10.1179/sre.1989.30.232.61.

[Deakin2004] Deakin, R. E. The standard and abridged Molodensky coordinate transformation formulae. Technical Report, Department of Mathematical and Geospatial Sciences, RMIT University, Melborne, Australia, 2004. URL: http://www.mygeodesy.id.au/documents/Molodensky%20V2.pdf.

[EberHewitt1979] Eber, L. E. and Hewitt, R. P. Conversion algorithms for the CalCOFI station grid. *California Cooperative Oceanic Fisheries Investigations Reports*, 20:135–137, 1979. URL: http://www.calcofi.org/publications/calcofireports/v20/Vol_20_Eber___Hewitt.pdf.

[Engsager2007] Engsager, K. E. and Poder, K. A highly accurate world wide algorithm for the transverse Mercator mapping (almost). In *Proc. XXIII Intl. Cartographic Conf. (ICC2007), Moscow*, 2.1.2. August 2007.

[Evenden1995] Evenden, G. I. *Cartographic Projection Procedures for the UNIX Environment — A User's Manual*. 1995. URL: https://pubs.usgs.gov/of/1990/of90-284/ofr90-284.pdf.

[Evenden2005] Evenden, G. I. *libproj4: A Comprehensive Library of Cartographic Projection Functions (Preliminary Draft)*. 2005. URL: https://github.com/OSGeo/PROJ/blob/master/docs/old/libproj.pdf.

[EversKnudsen2017] Evers, K. and Knudsen, T. Transformation pipelines for PROJ.4. In *FIG Working Week 2017 Proceedings*. Helsinki, Finland, 2017. URL: http://www.fig.net/resources/proceedings/fig_proceedings/fig2017/papers/iss6b/ISS6B_evers_knudsen_9156.pdf.

[Goode1919] Goode, J. P. Studies in projections: adapting the homolographic projection to the portrayal of the earth's entire surface. *Bul. Geog. SOC.Phila.*, XWIJ(3):103–113, 1919.

[Goode1925] Goode, J. P. The homolosine projection: a new device for portraying the earth's surface entire. *Annals of the Association of American Geographers*, 3(15):119–125, 1925. doi:10.1080/00045602509356949.

[Gray1995] Gray, R. W. Exact transformation equations for Fuller's World Map. *Cartographica*, 32(3):17–25, 1995. doi:10.3138/1677-3273-Q862-1885.

[Helmert1880] Helmert, F. R. *Mathematical and Physical Theories of Higher Geodesy*. Volume 1. Teubner, Leipzig, 1880. doi:10.5281/zenodo.32050.

[Hensley2002] Hensley, S., Chapin, E., Freedman, A., and Michel, T. Improved processing of AIRSAR data based on the GeoSAR processor. In *AIRSAR Earth Science and Application Workshop*. Pasadena, California, 2002. Jet Propulsion Laboratory. URL: https://airsar.jpl.nasa.gov/documents/workshop2002/papers/T3.pdf.

[Hakli2016] Häkli, P., Lidberg, M., Jivall, L., Nørbech, T., Tangen, O., Weber, M., Pihlak, P., Aleksejenko, I., and Paršeliunas, E. The NKG2008 GPS campaign – final transformation results and a new common Nordic reference frame. *Journal of Geodetic Science*, 6(1):1–33, 2016. doi:10.1515/jogs-2016-0001.

[NTF_88] IGN. Grille de parametres de transformation de coordonnees - GR3DF97A - notice d'utilisation. Technical Report, Service de Geodesie et Nivellement, Institut Geographique National, 1997. URL: https://geodesie.ign.fr/contenu/fichiers/documentation/algorithmes/notice/NTG_88.pdf.

[IOGP2018] IOGP. Geomatics guidance note 7, part 2: coordinate conversions & transformations including formulas. IOGP Publication 373-7-2, International Association For Oil And Gas Producers, 2018. URL: https://www.iogp.org/bookstore/product/coordinate-conversions-and-transformation-including-formulas/.

[IOGP2019] IOGP. Geomatics guidance note 7, part 2: coordinate conversions & transformations including formulas. IOGP Publication 373-7-2, International Association For Oil And Gas Producers, 2019. URL: https://www.iogp.org/wp-content/uploads/2019/09/373-07-02.pdf.

[ISO19111] ISO. Geographic information – Referencing by coordinates. Standard, International Organization for Standardization, Geneva, CH, January 2019. URL: http://docs.opengeospatial.org/as/18-005r5/18-005r5.html.

[Jenny2015] Jenny, B., Šavrič, B., and Patterson, T. A compromise aspect-adaptive cylindrical projection for world maps. *International Journal of Geographical Information Science*, 29(6):935–952, 2015. URL: http://www.cartography.oregonstate.edu/pdf/2015_Jenny_etal_ACompromiseAspect-adaptiveCylindricalProjectionForWorldMaps.pdf, doi:10.1080/13658816.2014.997734.

[JimenezShaw2023] Jimenez Shaw, J., Hernando, J., and Strecha, C. Site calibration with proj and wkt2. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVIII-4/W7-2023:75–81, 2023. URL: https://isprs-archives.copernicus.org/articles/XLVIII-4-W7-2023/75/2023/, doi:10.5194/isprs-archives-XLVIII-4-W7-2023-75-2023.

[Karney2011] Karney, C. F. F. Geodesics on an ellipsoid of revolution. *ArXiv e-prints*, 2011. arXiv:1102.1215.

[Karney2011tm] Karney, C. F. F. Transverse Mercator with an accuracy of a few nanometers. *J. Geod.*, 85(8):475–485, August 2011. arXiv:1002.1417, doi:10.1007/s00190-011-0445-3.

[Karney2013] Karney, C. F. F. Algorithms for geodesics. *Journal of Geodesy*, 87(1):43–55, 2013. doi:10.1007/s00190-012-0578-z.

[Komsta2016] Komsta, Ł. ATPOL geobotanical grid revisited – a proposal of coordinate conversion algorithms. *Annales UMCS Sectio E Agricultura*, 71(1):31–37, 2016.

[Krueger1912] Krüger, J. H. L. Konforme Abbildung des Erdellipsoids in der Ebene. New Series 52, Royal Prussian Geodetic Institute, Potsdam, 1912. doi:10.2312/GFZ.b103-krueger28.

[LambersKolb2012] Lambers, M. and Kolb, A. Ellipsoidal cube maps for accurate rendering of planetary-scale terrain data. In Bregler, C., Sander, P., and Wimmer, M., editors, *Pacific Graphics Short Papers*. The Eurographics Association, 2012. doi:10.2312/PE/PG/PG2012short/005-010.

[ONeilLaubscher1976] O'Neill, E. M. and Laubscher, R. E. Extended studies of a quadrilateralized spherical cube earth data base. Tech. Rep. EPRF 3-76 (CSC), Computer Sciences Corporation, System Sciences Division, Silver Spring, Md, 1976. URL: https://archive.org/details/DTIC_ADA026294.

[Patterson2014] Patterson, T., Šavrič, B., and Jenny, B. Introducing the Patterson cylindrical projection. *Cartographic Perspectives*, 2014. doi:10.14714/CP78.1270.

[Poder1998] Poder, K. and Engsager, K. Some conformal mappings and transformations for geodesy and topographic cartography. National Survey and Cadastre Publications, National Survey and Cadastre, Copenhagen, Denmark, 1998.

[Rittri2012] Rittri, M. New omerc approximations of Denmark System 34. e-mail, 2012. URL: https://lists.osgeo.org/pipermail/proj/2012-June/005926.html.

[Ruffhead2016] Ruffhead, A. C. Introduction to multiple regression equations in datum transformations and their reversibility. *Survey Review*, 50(358):82–90, 2016. doi:10.1080/00396265.2016.1244143.

[Snyder1987] Snyder, J. P. Map projections — A working manual. Professional Paper 1395, U.S. Geological Survey, 1987. doi:10.3133/pp1395.

[Snyder1988] Snyder, J. P. New equal-area map projections for noncircular regions. *The American Cartographer*, 15(4):341–356, 1988. doi:10.1559/152304088783886784.

[Snyder1992] Snyder, J. P. An equal-area map projection for polyhedral globes. *Cartographica*, 29(1):10–21, 1992. doi:10.3138/27H7-8K88-4882-1752.

[Snyder1993] Snyder, J. P. *Flattening the Earth*. University of Chicago Press, 1993.

[Steers1970] Steers, J. A. *An introduction to the study of map projections*. University of London Press, 15th edition, 1970.

[Tobler2018] Tobler, W. A new companion for Mercator. *Cartography and Geographic Information Science*, 45(3):284–285, 2018. doi:10.1080/15230406.2017.1308837.

[Verey2017] Verey, M. Theoretical analysis and practical consequences of adopting a model ATPOL grid as a conical projection defining the conversion of plane coordinates to the WGS 84 ellipsoid. *Fragmenta Floristica et Geobotanica Polonica*, 24(2):469–488, 2017. URL: http://bomax.botany.pl/pubs-new/#article-4279.

[Vincenty1975] Vincenty, T. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*, 23(176):88–93, 1975. doi:10.1179/sre.1975.23.176.88.

[WeberMoore2013] Weber, E. D. and Moore, T. J. Corrected conversion algorithms for the CalCOFI station grid and their implementation in several computer languages. *California Cooperative Oceanic Fisheries Investigations Reports*, 54:1–10, 2013. URL: http://calcofi.org/publications/calcofireports/v54/Vol_54_Weber.pdf.

[Zajac1978] Zajaç, A. Atlas of distribution of vascular plants in Poland (ATPOL). *Taxon*, 27(5/6):481–484, 1978. doi:10.2307/1219899.

[Savric2015] Šavrič, B., Patterson, T., and Jenny, B. The Natural Earth II world map projection. *International Journal of Cartography*, 1(2):123–133, 2015. URL: https://www.researchgate.net/publication/290447301_The_Natural_Earth_II_world_map_projection, doi:10.1080/23729333.2015.1093312.

[Savric2018] Šavrič, B., Patterson, T., and Jenny, B. The Equal Earth map projection. *International Journal of Geographical Information Science*, 33(3):454–465, 2018. URL: https://www.researchgate.net/publication/326879978_The_Equal_Earth_map_projection, doi:10.1080/13658816.2018.1504949.

# INDEX

## Symbols


+M
    command line option, 194
+R
    command line option, 67, 109, 111, 114, 117, 119, 121, 122, 125, 126, 128, 129, 131, 132, 134, 136, 139, 142, 144, 147, 149, 150, 153, 154, 156–158, 160–162, 164, 166, 169, 171, 173, 175–177, 179, 183, 185, 186, 188, 190, 194–196, 199, 202, 204–206, 208, 210, 212, 213, 216, 220, 222, 224, 225, 228, 230, 231, 235, 236, 238–241, 243, 246, 250, 253, 256, 258, 260, 262, 264–266, 268, 269, 271, 274, 276, 277, 283, 286, 287, 289, 293, 295–297, 299–302, 304–307, 312, 314, 315, 320, 321, 323, 325, 327, 330, 332, 334, 337, 338, 340, 343, 349, 351, 353, 355, 357, 358, 362, 364, 366, 367, 369, 371, 372, 374–376, 378, 381, 383, 384, 386
+R_A
    command line option, 68
+R_C
    command line option, 68
+R_V
    command line option, 68
+R_a
    command line option, 68
+R_g
    command line option, 68
+R_h
    command line option, 68
+R_lat_a
    command line option, 68
+R_lat_g
    command line option, 68
+UVtoST
    command line option, 318
+W
    command line option, 194, 220
+X_0
    command line option, 396
+Y_0
    command line option, 396
+Z_0
    command line option, 396
+a
    command line option, 67
+abridged

command line option, 416
+algo
command line option, 342, 360
+alpha
command line option, 275, 281, 285, 355
+aperture
command line option, 210
+approx
command line option, 342, 360
+asc_lon
command line option, 323
+azi
command line option, 209, 216, 327, 353
+b
command line option, 68
+convention
command line option, 407, 417
+czech
command line option, 214, 255
+da
command line option, 415
+datum
command line option, 389
+deg
command line option, 413
+df
command line option, 415
+dh
command line option, 405, 423
+dlat
command line option, 405
+dlon
command line option, 405
+drx
command line option, 408
+dry
command line option, 408
+drz
command line option, 408
+ds
command line option, 408
+dt
command line option, 403
+dx
command line option, 408, 415
+dy
command line option, 408, 416
+dz
command line option, 408, 416
+e
command line option, 68
+ellps
command line option, 114, 117, 124, 134, 136, 139, 146, 152, 166, 169, 171, 180, 183, 188, 191, 199, 201, 220, 222, 227, 229, 231, 232, 236, 246, 249, 253, 286, 295, 311, 313, 318, 320, 323, 325, 330, 332, 333,

343, 355, 357, 360, 379, 388, 389, 396, 413, 416, 424, 427
+es
    command line option, 68
+exact
    command line option, 408
+f
    command line option, 68
+file
    command line option, 420
+fwd_c
    command line option, 414
+fwd_origin
    command line option, 413
+fwd_u
    command line option, 413
+fwd_v
    command line option, 413
+gamma
    command line option, 281
+grid_ref
    command line option, 427
+grids
    command line option, 403, 406, 419, 426, 427
+guam
    command line option, 115
+h
    command line option, 183, 271, 351
+h_0
    command line option, 152, 319, 396
+hyperbolic
    command line option, 139
+inc_angle
    command line option, 323
+interpolation
    command line option, 406
+inv
    command line option, 429
+inv_c
    command line option, 414
+inv_origin
    command line option, 414
+inv_tolerance
    command line option, 414
+inv_u
    command line option, 414
+inv_v
    command line option, 414
+k_0
    command line option, 147, 214, 228, 245, 255, 276, 282, 285, 325, 327, 330, 332, 333, 337, 343, 348
+lat_0
    command line option, 115, 120, 125, 137, 151, 166, 188, 210, 214, 216, 218, 227, 229, 255, 271, 281, 285, 310, 318, 327, 328, 331, 333, 343, 353, 396, 423
+lat_1

command line option, 114, 133, 143, 149, 167, 172, 207, 222, 226, 231, 234, 257, 259, 261, 276, 281, 288, 339, 351, 368, 385
+lat_2
command line option, 114, 149, 169, 173, 208, 227, 258, 259, 262, 276, 281, 289, 340, 351, 369
+lat_3
command line option, 149
+lat_b
command line option, 120
+lat_ts
command line option, 146, 166, 245, 315, 328, 373, 382, 383
+lon_0
command line option, 107, 109, 111, 114, 115, 119, 121, 122, 125–127, 131, 134, 137, 142, 143, 146, 149–152, 154–156, 158–160, 162, 163, 165, 169, 171, 173–175, 177, 179, 183, 184, 186, 188, 189, 192, 194, 196, 198, 201–204, 206, 208, 210, 212–214, 216, 218, 220, 223, 224, 227, 228, 231, 234, 235, 237, 239–242, 245, 250, 253, 255, 256, 258, 260, 262, 263, 265–268, 271, 274, 276, 277, 283, 285, 286, 289, 290, 294, 295, 297–300, 302–305, 307, 310, 312, 313, 315, 317, 319, 321, 323, 324, 327, 330, 331, 333, 334, 337, 338, 340, 342, 348, 353–355, 358, 362, 364, 366, 369, 370, 372–374, 376, 377, 379, 381, 382, 384, 386, 396, 423
+lon_1
command line option, 149, 276, 281, 350
+lon_2
command line option, 149, 276, 281, 351
+lon_3
command line option, 149
+lonc
command line option, 275, 281
+lsat
command line option, 235
+m
command line option, 277
+mode
command line option, 210
+model
command line option, 401
+multiplier
command line option, 426, 427
+n
command line option, 177, 277, 355, 358
+no_cut
command line option, 120
+no_off
command line option, 282
+no_rot
command line option, 282
+no_z_transform
command line option, 406
+north_square
command line option, 200
+ns
command line option, 130
+o_alpha
command line option, 274
+o_lat_1
command line option, 274

+o_lat_2
    command line option, 274
+o_lat_c
    command line option, 274
+o_lat_p
    command line option, 274
+o_lon_1
    command line option, 274
+o_lon_2
    command line option, 274
+o_lon_c
    command line option, 274
+o_lon_p
    command line option, 274
+o_proj
    command line option, 273
+omit_fwd
    command line option, 429
+omit_inv
    command line option, 430
+order
    command line option, 387
+orient
    command line option, 119, 209
+path
    command line option, 235, 251
+phdg_0
    command line option, 319
+plat_0
    command line option, 319
+plon_0
    command line option, 319
+ps_rev
    command line option, 323
+px
    command line option, 417
+py
    command line option, 417
+pz
    command line option, 417
+q
    command line option, 355
+range
    command line option, 414
+resolution
    command line option, 210
+rf
    command line option, 68
+rot
    command line option, 327
+rot_xy
    command line option, 198
+rx
    command line option, 408, 417

+ry
    command line option, 408, 417
+rz
    command line option, 408, 417
+s
    command line option, 408, 417
+s11
    command line option, 400
+s12
    command line option, 400
+s13
    command line option, 400
+s21
    command line option, 400
+s22
    command line option, 400
+s23
    command line option, 400
+s31
    command line option, 400
+s32
    command line option, 400
+s33
    command line option, 400
+scrollx
    command line option, 293
+scrolly
    command line option, 293
+shape
    command line option, 290
+slope_lat
    command line option, 423
+slope_lon
    command line option, 424
+south
    command line option, 231, 354, 360
+south_square
    command line option, 201
+step
    command line option, 429
+sweep
    command line option, 183
+t_epoch
    command line option, 403, 408, 419, 426
+t_final
    command line option, 419, 426
+t_in
    command line option, 397
+t_out
    command line option, 397
+theta
    command line option, 277, 408
+tilt
    command line option, 353

+toff
    command line option, 400
+towgs84
    command line option, 390
+tscale
    command line option, 400
+uneg
    command line option, 414
+v_1
    command line option, 391–393
+v_2
    command line option, 391–393
+v_3
    command line option, 391, 392, 394
+v_4
    command line option, 391, 392, 394
+vneg
    command line option, 414
+x
    command line option, 408, 417
+x_0
    command line option, 109, 111, 113, 115, 117, 119, 121–123, 125, 126, 128, 129, 131, 132, 134, 139, 142, 144, 147, 149, 150, 152, 153, 155–157, 159–161, 163, 164, 166, 169, 171, 173, 175–177, 179, 183, 185, 186, 188, 190, 191, 194, 195, 197, 199, 201, 202, 204–206, 208, 210, 212–214, 218, 220, 222, 224, 225, 228, 230–232, 235, 236, 238–240, 242, 243, 245, 249, 250, 253, 255, 256, 258, 260, 262, 264, 265, 267–269, 271, 274, 276, 277, 282, 283, 286, 287, 289, 293, 295–297, 299–302, 304–307, 311, 312, 314, 315, 318, 319, 321, 323, 325, 327, 330, 332, 334, 335, 337, 338, 340, 343, 349, 351, 353, 355, 357, 358, 362, 364, 366, 367, 369, 371, 372, 374–376, 378, 380, 381, 383, 384, 386
+xoff
    command line option, 400
+xy_grids
    command line option, 403
+xy_in
    command line option, 397
+xy_out
    command line option, 397
+y
    command line option, 408, 417
+y_0
    command line option, 109, 111, 113, 115, 117, 119, 121, 122, 124, 125, 127–129, 131, 132, 134, 139, 142, 144, 147, 149, 151–153, 155–157, 159–161, 163, 164, 166, 170, 171, 173, 175, 176, 178, 180, 183, 185, 186, 188, 190, 191, 194, 195, 197, 199, 201, 202, 204, 205, 207, 208, 211–213, 216, 218, 220, 222, 224, 225, 228, 230–232, 235, 236, 238–240, 242, 243, 245, 249, 250, 253, 255, 256, 258, 260, 262, 264, 265, 267–269, 271, 275–277, 282, 283, 286, 287, 289, 293, 295–297, 299–302, 304–307, 311, 312, 314, 315, 318, 320, 321, 323, 325, 328, 330, 332, 334, 335, 337, 339, 340, 343, 349, 351, 353, 355, 357, 358, 362, 364, 366, 367, 369, 371, 372, 374–376, 378, 380, 381, 383, 384, 386
+yoff
    command line option, 400
+z
    command line option, 408, 417
+z_grids
    command line option, 404
+z_in
    command line option, 397

+z_out
    command line option, 397
+zoff
    command line option, 400
+zone
    command line option, 360
-E
    cs2cs command line option, 79
    proj command line option, 91
-F
    geod command line option, 84
-I
    cct command line option, 76
    cs2cs command line option, 79
    geod command line option, 83
    proj command line option, 91
-S
    proj command line option, 91
-V
    proj command line option, 92
-W<n>
    cs2cs command line option, 80
    geod command line option, 84
    proj command line option, 92
--3d
    cs2cs command line option, 81
    projinfo command line option, 98
--accuracy
    cs2cs command line option, 81
    projinfo command line option, 97
--all
    projsync command line option, 105
--allow-ellipsoidal-height-as-vertical-crs
    projinfo command line option, 97
--area
    cs2cs command line option, 80
    projinfo command line option, 95
--area-of-use
    projsync command line option, 105
--authority
    cs2cs command line option, 81
    projinfo command line option, 98
--aux-db-path
    projinfo command line option, 98
--bbox
    cs2cs command line option, 80
    projinfo command line option, 96
    projsync command line option, 104
--boundcrs-to-wgs84
    projinfo command line option, 98
--c-ify
    projinfo command line option, 99
--crs-extent-use
    projinfo command line option, 96

--dry-run
    projsync command line option, 105
--dump-db-structure
    projinfo command line option, 98
--endpoint
    projsync command line option, 104
--exclude-world-coverage
    projsync command line option, 105
--file
    projsync command line option, 105
--grid-check
    projinfo command line option, 96
--height
    cct command line option, 76
--help
    command line option, 85
--hide-ballpark
    projinfo command line option, 97
--identify
    projinfo command line option, 98
--list
    command line option, 86
--list-crs
    projinfo command line option, 98
--list-files
    projsync command line option, 105
--local-geojson-file
    projsync command line option, 104
--main-db-path
    projinfo command line option, 98
--no-ballpark
    cs2cs command line option, 80
--no-version-filtering
    projsync command line option, 105
--only-best
    cs2cs command line option, 80
--output
    cct command line option, 76
    command line option, 85
--output-id
    projinfo command line option, 98
--pivot-crs
    projinfo command line option, 97
--quiet
    command line option, 86
--remote-data
    projinfo command line option, 99
--s_epoch
    cs2cs command line option, 81
    projinfo command line option, 99
--searchpaths
    projinfo command line option, 99
--show-superseded
    projinfo command line option, 97

--single-line
    projinfo command line option, 99
--skip-lines
    cct command line option, 76
--source-id
    projsync command line option, 105
--spatial-test
    projinfo command line option, 96
    projsync command line option, 104
--summary
    projinfo command line option, 95
--system-directory
    projsync command line option, 104
--t_epoch
    cs2cs command line option, 81
    projinfo command line option, 99
--target-dir
    projsync command line option, 104
--time
    cct command line option, 76
--user-writable-directory
    projsync command line option, 104
--verbose
    cct command line option, 76
    command line option, 86
    projsync command line option, 105
--version
    cct command line option, 76
    command line option, 86
-a
    geod command line option, 83
-b
    proj command line option, 91
-c
    cct command line option, 76
-d
    cct command line option, 76
    cs2cs command line option, 79
    proj command line option, 91
-e
    cs2cs command line option, 79
    proj command line option, 91
-f
    cs2cs command line option, 80
    geod command line option, 84
    proj command line option, 92
-h
    command line option, 85
-i
    proj command line option, 91
-k
    projinfo command line option, 95
-l
    command line option, 86

-lP
    cs2cs command line option, 79
    proj command line option, 91
-l<[
    cs2cs command line option, 79
    proj command line option, 91
-le
    cs2cs command line option, 79
    geod command line option, 84
    proj command line option, 91
-lm
    cs2cs command line option, 79
-lp
    cs2cs command line option, 79
    proj command line option, 91
-lu
    cs2cs command line option, 79
    geod command line option, 84
    proj command line option, 91
-m
    proj command line option, 92
-o
    cct command line option, 76
    command line option, 85
    proj command line option, 91
    projinfo command line option, 95
-p
    geod command line option, 84
-q
    command line option, 86
    projinfo command line option, 95
    projsync command line option, 105
-r
    cs2cs command line option, 79
    proj command line option, 91
-s
    cct command line option, 76
    cs2cs command line option, 80
    proj command line option, 91
-t
    cct command line option, 76
-t<a>
    cs2cs command line option, 79
    geod command line option, 83
    proj command line option, 91
-v
    cct command line option, 76
    command line option, 86
    cs2cs command line option, 80
    proj command line option, 92
-w<n>
    cs2cs command line option, 80
    geod command line option, 84
    proj command line option, 92

-z
    cct command line option, 76

## A

accept
    command line option, 87
APPEND_SOVERSION
    command line option, 52

## B

Ballpark transformation, **885**
BUILD_APPS
    command line option, 50
BUILD_CCT
    command line option, 51
BUILD_CS2CS
    command line option, 51
BUILD_GEOD
    command line option, 51
BUILD_GIE
    command line option, 51
BUILD_PROJ
    command line option, 51
BUILD_PROJINFO
    command line option, 51
BUILD_PROJSYNC
    command line option, 51
BUILD_SHARED_LIBS
    command line option, 51
BUILD_TESTING
    command line option, 51

## C

cct, 75
cct command line option
    -I, 76
    --height, 76
    --output, 76
    --skip-lines, 76
    --time, 76
    --verbose, 76
    --version, 76
    -c, 76
    -d, 76
    -o, 76
    -s, 76
    -t, 76
    -v, 76
    -z, 76
CFLAGS, 52
CMAKE_BUILD_TYPE
    command line option, 51
CMAKE_C_COMPILER
    command line option, 52

CMAKE_C_FLAGS
    command line option, 52
CMAKE_CXX_COMPILER
    command line option, 52
CMAKE_CXX_FLAGS
    command line option, 52
CMAKE_INSTALL_MANDIR, 15
CMAKE_INSTALL_PREFIX
    command line option, 52
CMAKE_PREFIX_PATH
    command line option, 52
CMAKE_UNITY_BUILD
    command line option, 52
command line option
    +M, 194
    +R, 67, 109, 111, 114, 117, 119, 121, 122, 125, 126, 128, 129, 131, 132, 134, 136, 139, 142, 144, 147, 149, 150, 153, 154, 156–158, 160–162, 164, 166, 169, 171, 173, 175–177, 179, 183, 185, 186, 188, 190, 194–196, 199, 202, 204–206, 208, 210, 212, 213, 216, 220, 222, 224, 225, 228, 230, 231, 235, 236, 238–241, 243, 246, 250, 253, 256, 258, 260, 262, 264–266, 268, 269, 271, 274, 276, 277, 283, 286, 287, 289, 293, 295–297, 299–302, 304–307, 312, 314, 315, 320, 321, 323, 325, 327, 330, 332, 334, 337, 338, 340, 343, 349, 351, 353, 355, 357, 358, 362, 364, 366, 367, 369, 371, 372, 374–376, 378, 381, 383, 384, 386
    +R_A, 68
    +R_C, 68
    +R_V, 68
    +R_a, 68
    +R_g, 68
    +R_h, 68
    +R_lat_a, 68
    +R_lat_g, 68
    +UVtoST, 318
    +W, 194, 220
    +X_0, 396
    +Y_0, 396
    +Z_0, 396
    +a, 67
    +abridged, 416
    +algo, 342, 360
    +alpha, 275, 281, 285, 355
    +aperture, 210
    +approx, 342, 360
    +asc_lon, 323
    +azi, 209, 216, 327, 353
    +b, 68
    +convention, 407, 417
    +czech, 214, 255
    +da, 415
    +datum, 389
    +deg, 413
    +df, 415
    +dh, 405, 423
    +dlat, 405
    +dlon, 405
    +drx, 408
    +dry, 408

+drz, 408
+ds, 408
+dt, 403
+dx, 408, 415
+dy, 408, 416
+dz, 408, 416
+e, 68
+ellps, 114, 117, 124, 134, 136, 139, 146, 152, 166, 169, 171, 180, 183, 188, 191, 199, 201, 220, 222, 227, 229, 231, 232, 236, 246, 249, 253, 286, 295, 311, 313, 318, 320, 323, 325, 330, 332, 333, 343, 355, 357, 360, 379, 388, 389, 396, 413, 416, 424, 427
+es, 68
+exact, 408
+f, 68
+file, 420
+fwd_c, 414
+fwd_origin, 413
+fwd_u, 413
+fwd_v, 413
+gamma, 281
+grid_ref, 427
+grids, 403, 406, 419, 426, 427
+guam, 115
+h, 183, 271, 351
+h_0, 152, 319, 396
+hyperbolic, 139
+inc_angle, 323
+interpolation, 406
+inv, 429
+inv_c, 414
+inv_origin, 414
+inv_tolerance, 414
+inv_u, 414
+inv_v, 414
+k_0, 147, 214, 228, 245, 255, 276, 282, 285, 325, 327, 330, 332, 333, 337, 343, 348
+lat_0, 115, 120, 125, 137, 151, 166, 188, 210, 214, 216, 218, 227, 229, 255, 271, 281, 285, 310, 318, 327, 328, 331, 333, 343, 353, 396, 423
+lat_1, 114, 133, 143, 149, 167, 172, 207, 222, 226, 231, 234, 257, 259, 261, 276, 281, 288, 339, 351, 368, 385
+lat_2, 114, 149, 169, 173, 208, 227, 258, 259, 262, 276, 281, 289, 340, 351, 369
+lat_3, 149
+lat_b, 120
+lat_ts, 146, 166, 245, 315, 328, 373, 382, 383
+lon_0, 107, 109, 111, 114, 115, 119, 121, 122, 125–127, 131, 134, 137, 142, 143, 146, 149–152, 154–156, 158–160, 162, 163, 165, 169, 171, 173–175, 177, 179, 183, 184, 186, 188, 189, 192, 194, 196, 198, 201–204, 206, 208, 210, 212–214, 216, 218, 220, 223, 224, 227, 228, 231, 234, 235, 237, 239–242, 245, 250, 253, 255, 256, 258, 260, 262, 263, 265–268, 271, 274, 276, 277, 283, 285, 286, 289, 290, 294, 295, 297–300, 302–305, 307, 310, 312, 313, 315, 317, 319, 321, 323, 324, 327, 330, 331, 333, 334, 337, 338, 340, 342, 348, 353–355, 358, 362, 364, 366, 369, 370, 372–374, 376, 377, 379, 381, 382, 384, 386, 396, 423
+lon_1, 149, 276, 281, 350
+lon_2, 149, 276, 281, 351
+lon_3, 149
+lonc, 275, 281
+lsat, 235
+m, 277
+mode, 210

+model, 401
+multiplier, 426, 427
+n, 177, 277, 355, 358
+no_cut, 120
+no_off, 282
+no_rot, 282
+no_z_transform, 406
+north_square, 200
+ns, 130
+o_alpha, 274
+o_lat_1, 274
+o_lat_2, 274
+o_lat_c, 274
+o_lat_p, 274
+o_lon_1, 274
+o_lon_2, 274
+o_lon_c, 274
+o_lon_p, 274
+o_proj, 273
+omit_fwd, 429
+omit_inv, 430
+order, 387
+orient, 119, 209
+path, 235, 251
+phdg_0, 319
+plat_0, 319
+plon_0, 319
+ps_rev, 323
+px, 417
+py, 417
+pz, 417
+q, 355
+range, 414
+resolution, 210
+rf, 68
+rot, 327
+rot_xy, 198
+rx, 408, 417
+ry, 408, 417
+rz, 408, 417
+s, 408, 417
+s11, 400
+s12, 400
+s13, 400
+s21, 400
+s22, 400
+s23, 400
+s31, 400
+s32, 400
+s33, 400
+scrollx, 293
+scrolly, 293
+shape, 290
+slope_lat, 423

+slope_lon, 424
+south, 231, 354, 360
+south_square, 201
+step, 429
+sweep, 183
+t_epoch, 403, 408, 419, 426
+t_final, 419, 426
+t_in, 397
+t_out, 397
+theta, 277, 408
+tilt, 353
+toff, 400
+towgs84, 390
+tscale, 400
+uneg, 414
+v_1, 391–393
+v_2, 391–393
+v_3, 391, 392, 394
+v_4, 391, 392, 394
+vneg, 414
+x, 408, 417
+x_0, 109, 111, 113, 115, 117, 119, 121–123, 125, 126, 128, 129, 131, 132, 134, 139, 142, 144, 147, 149, 150, 152, 153, 155–157, 159–161, 163, 164, 166, 169, 171, 173, 175–177, 179, 183, 185, 186, 188, 190, 191, 194, 195, 197, 199, 201, 202, 204–206, 208, 210, 212–214, 218, 220, 222, 224, 225, 228, 230–232, 235, 236, 238–240, 242, 243, 245, 249, 250, 253, 255, 256, 258, 260, 262, 264, 265, 267–269, 271, 274, 276, 277, 282, 283, 286, 287, 289, 293, 295–297, 299–302, 304–307, 311, 312, 314, 315, 318, 319, 321, 323, 325, 327, 330, 332, 334, 335, 337, 338, 340, 343, 349, 351, 353, 355, 357, 358, 362, 364, 366, 367, 369, 371, 372, 374–376, 378, 380, 381, 383, 384, 386
+xoff, 400
+xy_grids, 403
+xy_in, 397
+xy_out, 397
+y, 408, 417
+y_0, 109, 111, 113, 115, 117, 119, 121, 122, 124, 125, 127–129, 131, 132, 134, 139, 142, 144, 147, 149, 151–153, 155–157, 159–161, 163, 164, 166, 170, 171, 173, 175, 176, 178, 180, 183, 185, 186, 188, 190, 191, 194, 195, 197, 199, 201, 202, 204, 205, 207, 208, 211–213, 216, 218, 220, 222, 224, 225, 228, 230–232, 235, 236, 238–240, 242, 243, 245, 249, 250, 253, 255, 256, 258, 260, 262, 264, 265, 267–269, 271, 275–277, 282, 283, 286, 287, 289, 293, 295–297, 299–302, 304–307, 311, 312, 314, 315, 318, 320, 321, 323, 325, 328, 330, 332, 334, 335, 337, 339, 340, 343, 349, 351, 353, 355, 357, 358, 362, 364, 366, 367, 369, 371, 372, 374–376, 378, 380, 381, 383, 384, 386
+yoff, 400
+z, 408, 417
+z_grids, 404
+z_in, 397
+z_out, 397
+zoff, 400
+zone, 360
--help, 85
--list, 86
--output, 85
--quiet, 86
--verbose, 86
--version, 86
-h, 85

-l, 86
-o, 85
-q, 86
-v, 86
accept, 87
APPEND_SOVERSION, 52
BUILD_APPS, 50
BUILD_CCT, 51
BUILD_CS2CS, 51
BUILD_GEOD, 51
BUILD_GIE, 51
BUILD_PROJ, 51
BUILD_PROJINFO, 51
BUILD_PROJSYNC, 51
BUILD_SHARED_LIBS, 51
BUILD_TESTING, 51
CMAKE_BUILD_TYPE, 51
CMAKE_C_COMPILER, 52
CMAKE_C_FLAGS, 52
CMAKE_CXX_COMPILER, 52
CMAKE_CXX_FLAGS, 52
CMAKE_INSTALL_PREFIX, 52
CMAKE_PREFIX_PATH, 52
CMAKE_UNITY_BUILD, 52
CURL_INCLUDE_DIR, 53
CURL_LIBRARY, 53
direction, 88
echo, 89
EMBED_PROJ_DATA_PATH, 53
EMBED_RESOURCE_DIRECTORY, 53
EMBED_RESOURCE_FILES, 53
ENABLE_CURL, 53
ENABLE_IPO, 52
ENABLE_TIFF, 53
EXE_SQLITE3, 52
expect, 87
ignore, 89
operation, 87
PROJ_DB_CACHE_DIR, 53
PROJ_OUTPUT_NAME, 51
require_grid, 89
roundtrip, 88
skip, 89
TESTING_USE_NETWORK, 53
TIFF_INCLUDE_DIR, 53
TIFF_LIBRARY_RELEASE, 53
tolerance, 87
USE_CCACHE, 53
USE_ONLY_EMBEDDED_RESOURCE_FILES, 54
cs2cs command line option
-E, 79
-I, 79
-W<n>, 80
--3d, 81

--accuracy, 81
--area, 80
--authority, 81
--bbox, 80
--no-ballpark, 80
--only-best, 80
--s_epoch, 81
--t_epoch, 81
-d, 79
-e, 79
-f, 80
-lP, 79
-l<[, 79
-le, 79
-lm, 79
-lp, 79
-lu, 79
-r, 79
-s, 80
-t<a>, 79
-v, 80
-w<n>, 80
CURL_INCLUDE_DIR
command line option, 53
CURL_LIBRARY
command line option, 53
CXXFLAGS, 52

## D

direction
command line option, 88

## E

echo
command line option, 89
EMBED_PROJ_DATA_PATH
command line option, 53
EMBED_RESOURCE_DIRECTORY
command line option, 53
EMBED_RESOURCE_FILES, 4
command line option, 53
ENABLE_CURL
command line option, 53
ENABLE_IPO
command line option, 52
ENABLE_TIFF
command line option, 53
environment variable
CFLAGS, 52
CMAKE_INSTALL_MANDIR, 15
CXXFLAGS, 52
EMBED_RESOURCE_FILES, 4
OSGEO4W_ROOT, 52
PROJ_AUX_DB, 70, 99

PROJ_CURL_CA_BUNDLE, 70
PROJ_DATA, 69, 70, 81, 92, 96, 447, 448, 450, 453, 881
PROJ_DEBUG, 9, 70
PROJ_LIB, 22, 34, 36, 70, 447
PROJ_NATIVE_CA, 4, 70
PROJ_NETWORK, 50, 70, 73, 77, 82, 96, 448
PROJ_NETWORK_ENDPOINT, 70, 73
PROJ_ONLY_BEST_DEFAULT, 80, 497
SSL_OPTIONS, 7
USE_ONLY_EMBEDDED_RESOURCE_FILES, 4
XDG_DATA_HOME, 447
EXE_SQLITE3
command line option, 52
expect
command line option, 87

## G

geod command line option
-F, 84
-I, 83
-W<n>, 84
-a, 83
-f, 84
-le, 84
-lu, 84
-p, 84
-t<a>, 83
-w<n>, 84
gie, 85

## I

ignore
command line option, 89

## O

operation
command line option, 87
OSGEO4W_ROOT, 52
osgeo::proj::common (*C++ type*), 583
osgeo::proj::common::Angle (*C++ class*), 587
osgeo::proj::common::Angle::Angle (*C++ function*), 587
osgeo::proj::common::DataEpoch (*C++ class*), 588
osgeo::proj::common::DataEpoch::coordinateEpoch (*C++ function*), 588
osgeo::proj::common::DateTime (*C++ class*), 588
osgeo::proj::common::DateTime::create (*C++ function*), 588
osgeo::proj::common::DateTime::isISO_8601 (*C++ function*), 588
osgeo::proj::common::DateTime::toString (*C++ function*), 588
osgeo::proj::common::IdentifiedObject (*C++ class*), 588
osgeo::proj::common::IdentifiedObject::alias (*C++ function*), 589
osgeo::proj::common::IdentifiedObject::ALIAS_KEY (*C++ member*), 590
osgeo::proj::common::IdentifiedObject::aliases (*C++ function*), 589
osgeo::proj::common::IdentifiedObject::DEPRECATED_KEY (*C++ member*), 590
osgeo::proj::common::IdentifiedObject::getEPSGCode (*C++ function*), 589
osgeo::proj::common::IdentifiedObject::identifiers (*C++ function*), 589

osgeo::proj::common::IdentifiedObject::IDENTIFIERS_KEY (C++ member), 590
osgeo::proj::common::IdentifiedObject::isDeprecated (C++ function), 589
osgeo::proj::common::IdentifiedObject::name (C++ function), 589
osgeo::proj::common::IdentifiedObject::NAME_KEY (C++ member), 590
osgeo::proj::common::IdentifiedObject::nameStr (C++ function), 589
osgeo::proj::common::IdentifiedObject::remarks (C++ function), 589
osgeo::proj::common::IdentifiedObject::REMARKS_KEY (C++ member), 590
osgeo::proj::common::IdentifiedObjectNNPtr (C++ type), 583
osgeo::proj::common::IdentifiedObjectPtr (C++ type), 583
osgeo::proj::common::Length (C++ class), 588
osgeo::proj::common::Length::Length (C++ function), 588
osgeo::proj::common::Measure (C++ class), 586
osgeo::proj::common::Measure::_isEquivalentTo (C++ function), 587
osgeo::proj::common::Measure::convertToUnit (C++ function), 586
osgeo::proj::common::Measure::DEFAULT_MAX_REL_ERROR (C++ member), 587
osgeo::proj::common::Measure::getSIValue (C++ function), 586
osgeo::proj::common::Measure::Measure (C++ function), 586
osgeo::proj::common::Measure::operator== (C++ function), 587
osgeo::proj::common::Measure::unit (C++ function), 586
osgeo::proj::common::Measure::value (C++ function), 586
osgeo::proj::common::ObjectDomain (C++ class), 590
osgeo::proj::common::ObjectDomain::create (C++ function), 591
osgeo::proj::common::ObjectDomain::domainOfValidity (C++ function), 590
osgeo::proj::common::ObjectDomain::scope (C++ function), 590
osgeo::proj::common::ObjectDomainNNPtr (C++ type), 583
osgeo::proj::common::ObjectDomainPtr (C++ type), 583
osgeo::proj::common::ObjectUsage (C++ class), 591
osgeo::proj::common::ObjectUsage::DOMAIN_OF_VALIDITY_KEY (C++ member), 591
osgeo::proj::common::ObjectUsage::domains (C++ function), 591
osgeo::proj::common::ObjectUsage::OBJECT_DOMAIN_KEY (C++ member), 591
osgeo::proj::common::ObjectUsage::SCOPE_KEY (C++ member), 591
osgeo::proj::common::ObjectUsageNNPtr (C++ type), 584
osgeo::proj::common::ObjectUsagePtr (C++ type), 583
osgeo::proj::common::Scale (C++ class), 587
osgeo::proj::common::Scale::Scale (C++ function), 587
osgeo::proj::common::UnitOfMeasure (C++ class), 584
osgeo::proj::common::UnitOfMeasure::ARC_SECOND (C++ member), 586
osgeo::proj::common::UnitOfMeasure::ARC_SECOND_PER_YEAR (C++ member), 586
osgeo::proj::common::UnitOfMeasure::code (C++ function), 585
osgeo::proj::common::UnitOfMeasure::codeSpace (C++ function), 585
osgeo::proj::common::UnitOfMeasure::conversionToSI (C++ function), 584
osgeo::proj::common::UnitOfMeasure::DEGREE (C++ member), 586
osgeo::proj::common::UnitOfMeasure::FOOT (C++ member), 585
osgeo::proj::common::UnitOfMeasure::GRAD (C++ member), 586
osgeo::proj::common::UnitOfMeasure::METRE (C++ member), 585
osgeo::proj::common::UnitOfMeasure::METRE_PER_YEAR (C++ member), 585
osgeo::proj::common::UnitOfMeasure::MICRORADIAN (C++ member), 586
osgeo::proj::common::UnitOfMeasure::name (C++ function), 584
osgeo::proj::common::UnitOfMeasure::NONE (C++ member), 585
osgeo::proj::common::UnitOfMeasure::operator!= (C++ function), 585
osgeo::proj::common::UnitOfMeasure::operator== (C++ function), 585
osgeo::proj::common::UnitOfMeasure::PARTS_PER_MILLION (C++ member), 585
osgeo::proj::common::UnitOfMeasure::PPM_PER_YEAR (C++ member), 585
osgeo::proj::common::UnitOfMeasure::RADIAN (C++ member), 586

osgeo::proj::common::UnitOfMeasure::SCALE_UNITY (*C++ member*), 585
osgeo::proj::common::UnitOfMeasure::SECOND (*C++ member*), 586
osgeo::proj::common::UnitOfMeasure::Type (*C++ enum*), 584
osgeo::proj::common::UnitOfMeasure::type (*C++ function*), 585
osgeo::proj::common::UnitOfMeasure::Type::ANGULAR (*C++ enumerator*), 584
osgeo::proj::common::UnitOfMeasure::Type::LINEAR (*C++ enumerator*), 584
osgeo::proj::common::UnitOfMeasure::Type::PARAMETRIC (*C++ enumerator*), 584
osgeo::proj::common::UnitOfMeasure::Type::SCALE (*C++ enumerator*), 584
osgeo::proj::common::UnitOfMeasure::Type::TIME (*C++ enumerator*), 584
osgeo::proj::common::UnitOfMeasure::Type::Type::NONE (*C++ enumerator*), 584
osgeo::proj::common::UnitOfMeasure::Type::UNKNOWN (*C++ enumerator*), 584
osgeo::proj::common::UnitOfMeasure::UnitOfMeasure (*C++ function*), 584
osgeo::proj::common::UnitOfMeasure::US_FOOT (*C++ member*), 586
osgeo::proj::common::UnitOfMeasure::YEAR (*C++ member*), 586
osgeo::proj::common::UnitOfMeasureNNPtr (*C++ type*), 583
osgeo::proj::common::UnitOfMeasurePtr (*C++ type*), 583
osgeo::proj::crs (*C++ type*), 638
osgeo::proj::crs::BoundCRS (*C++ class*), 655
osgeo::proj::crs::BoundCRS::baseCRS (*C++ function*), 656
osgeo::proj::crs::BoundCRS::baseCRSWithCanonicalBoundCRS (*C++ function*), 656
osgeo::proj::crs::BoundCRS::create (*C++ function*), 656, 657
osgeo::proj::crs::BoundCRS::createFromNadgrids (*C++ function*), 657
osgeo::proj::crs::BoundCRS::createFromTOWGS84 (*C++ function*), 657
osgeo::proj::crs::BoundCRS::hubCRS (*C++ function*), 656
osgeo::proj::crs::BoundCRS::transformation (*C++ function*), 656
osgeo::proj::crs::BoundCRSNNPtr (*C++ type*), 638
osgeo::proj::crs::BoundCRSPtr (*C++ type*), 638
osgeo::proj::crs::CompoundCRS (*C++ class*), 654
osgeo::proj::crs::CompoundCRS::componentReferenceSystems (*C++ function*), 655
osgeo::proj::crs::CompoundCRS::create (*C++ function*), 655
osgeo::proj::crs::CompoundCRS::identify (*C++ function*), 655
osgeo::proj::crs::CompoundCRSNNPtr (*C++ type*), 638
osgeo::proj::crs::CompoundCRSPtr (*C++ type*), 638
osgeo::proj::crs::CRS (*C++ class*), 640
osgeo::proj::crs::CRS::canonicalBoundCRS (*C++ function*), 642
osgeo::proj::crs::CRS::createBoundCRSToWGS84IfPossible (*C++ function*), 641
osgeo::proj::crs::CRS::demoteTo2D (*C++ function*), 643
osgeo::proj::crs::CRS::extractGeodeticCRS (*C++ function*), 641
osgeo::proj::crs::CRS::extractGeographicCRS (*C++ function*), 641
osgeo::proj::crs::CRS::extractVerticalCRS (*C++ function*), 641
osgeo::proj::crs::CRS::getNonDeprecated (*C++ function*), 642
osgeo::proj::crs::CRS::identify (*C++ function*), 642
osgeo::proj::crs::CRS::isDynamic (*C++ function*), 641
osgeo::proj::crs::CRS::promoteTo3D (*C++ function*), 642
osgeo::proj::crs::CRS::stripVerticalComponent (*C++ function*), 642
osgeo::proj::crs::CRSNNPtr (*C++ type*), 638
osgeo::proj::crs::CRSPtr (*C++ type*), 638
osgeo::proj::crs::DerivedCRS (*C++ class*), 650
osgeo::proj::crs::DerivedCRS::baseCRS (*C++ function*), 650
osgeo::proj::crs::DerivedCRS::derivingConversion (*C++ function*), 650
osgeo::proj::crs::DerivedCRSNNPtr (*C++ type*), 639
osgeo::proj::crs::DerivedCRSPtr (*C++ type*), 639
osgeo::proj::crs::DerivedCRSTemplate (*C++ class*), 661
osgeo::proj::crs::DerivedCRSTemplate::baseCRS (*C++ function*), 661

osgeo::proj::crs::DerivedCRSTemplate::BaseNNPtr (*C++ type*), 661
osgeo::proj::crs::DerivedCRSTemplate::create (*C++ function*), 661
osgeo::proj::crs::DerivedCRSTemplate::CSNNPtr (*C++ type*), 661
osgeo::proj::crs::DerivedCRSTemplate::NNPtr (*C++ type*), 661
osgeo::proj::crs::DerivedEngineeringCRS (*C++ class*), 661
osgeo::proj::crs::DerivedEngineeringCRSNNPtr (*C++ type*), 640
osgeo::proj::crs::DerivedEngineeringCRSPtr (*C++ type*), 640
osgeo::proj::crs::DerivedGeodeticCRS (*C++ class*), 657
osgeo::proj::crs::DerivedGeodeticCRS::baseCRS (*C++ function*), 658
osgeo::proj::crs::DerivedGeodeticCRS::create (*C++ function*), 658
osgeo::proj::crs::DerivedGeodeticCRSNNPtr (*C++ type*), 640
osgeo::proj::crs::DerivedGeodeticCRSPtr (*C++ type*), 639
osgeo::proj::crs::DerivedGeographicCRS (*C++ class*), 658
osgeo::proj::crs::DerivedGeographicCRS::baseCRS (*C++ function*), 659
osgeo::proj::crs::DerivedGeographicCRS::create (*C++ function*), 659
osgeo::proj::crs::DerivedGeographicCRS::demoteTo2D (*C++ function*), 659
osgeo::proj::crs::DerivedGeographicCRSNNPtr (*C++ type*), 640
osgeo::proj::crs::DerivedGeographicCRSPtr (*C++ type*), 640
osgeo::proj::crs::DerivedParametricCRS (*C++ class*), 662
osgeo::proj::crs::DerivedParametricCRSNNPtr (*C++ type*), 640
osgeo::proj::crs::DerivedParametricCRSPtr (*C++ type*), 640
osgeo::proj::crs::DerivedProjectedCRS (*C++ class*), 659
osgeo::proj::crs::DerivedProjectedCRS::baseCRS (*C++ function*), 660
osgeo::proj::crs::DerivedProjectedCRS::create (*C++ function*), 660
osgeo::proj::crs::DerivedProjectedCRS::demoteTo2D (*C++ function*), 660
osgeo::proj::crs::DerivedProjectedCRSNNPtr (*C++ type*), 640
osgeo::proj::crs::DerivedProjectedCRSPtr (*C++ type*), 640
osgeo::proj::crs::DerivedTemporalCRS (*C++ class*), 662
osgeo::proj::crs::DerivedTemporalCRSNNPtr (*C++ type*), 640
osgeo::proj::crs::DerivedTemporalCRSPtr (*C++ type*), 640
osgeo::proj::crs::DerivedVerticalCRS (*C++ class*), 660
osgeo::proj::crs::DerivedVerticalCRS::baseCRS (*C++ function*), 660
osgeo::proj::crs::DerivedVerticalCRS::create (*C++ function*), 661
osgeo::proj::crs::DerivedVerticalCRSNNPtr (*C++ type*), 640
osgeo::proj::crs::DerivedVerticalCRSPtr (*C++ type*), 640
osgeo::proj::crs::EngineeringCRS (*C++ class*), 653
osgeo::proj::crs::EngineeringCRS::create (*C++ function*), 653
osgeo::proj::crs::EngineeringCRS::datum (*C++ function*), 653
osgeo::proj::crs::EngineeringCRSNNPtr (*C++ type*), 639
osgeo::proj::crs::EngineeringCRSPtr (*C++ type*), 639
osgeo::proj::crs::GeodeticCRS (*C++ class*), 644
osgeo::proj::crs::GeodeticCRS::create (*C++ function*), 645, 646
osgeo::proj::crs::GeodeticCRS::datum (*C++ function*), 644
osgeo::proj::crs::GeodeticCRS::ellipsoid (*C++ function*), 644
osgeo::proj::crs::GeodeticCRS::EPSG_4978 (*C++ member*), 646
osgeo::proj::crs::GeodeticCRS::identify (*C++ function*), 645
osgeo::proj::crs::GeodeticCRS::isGeocentric (*C++ function*), 644
osgeo::proj::crs::GeodeticCRS::isSphericalPlanetocentric (*C++ function*), 645
osgeo::proj::crs::GeodeticCRS::primeMeridian (*C++ function*), 644
osgeo::proj::crs::GeodeticCRS::velocityModel (*C++ function*), 644
osgeo::proj::crs::GeodeticCRSNNPtr (*C++ type*), 639
osgeo::proj::crs::GeodeticCRSPtr (*C++ type*), 639
osgeo::proj::crs::GeographicCRS (*C++ class*), 646
osgeo::proj::crs::GeographicCRS::coordinateSystem (*C++ function*), 647

osgeo::proj::crs::GeographicCRS::create (*C++ function*), 647
osgeo::proj::crs::GeographicCRS::demoteTo2D (*C++ function*), 647
osgeo::proj::crs::GeographicCRS::EPSG_4267 (*C++ member*), 648
osgeo::proj::crs::GeographicCRS::EPSG_4269 (*C++ member*), 648
osgeo::proj::crs::GeographicCRS::EPSG_4326 (*C++ member*), 648
osgeo::proj::crs::GeographicCRS::EPSG_4807 (*C++ member*), 648
osgeo::proj::crs::GeographicCRS::EPSG_4979 (*C++ member*), 648
osgeo::proj::crs::GeographicCRS::OGC_CRS84 (*C++ member*), 648
osgeo::proj::crs::GeographicCRSNNPtr (*C++ type*), 638
osgeo::proj::crs::GeographicCRSPtr (*C++ type*), 638
osgeo::proj::crs::InvalidCompoundCRSException (*C++ class*), 654
osgeo::proj::crs::ParametricCRS (*C++ class*), 653
osgeo::proj::crs::ParametricCRS::coordinateSystem (*C++ function*), 654
osgeo::proj::crs::ParametricCRS::create (*C++ function*), 654
osgeo::proj::crs::ParametricCRS::datum (*C++ function*), 654
osgeo::proj::crs::ParametricCRSNNPtr (*C++ type*), 639
osgeo::proj::crs::ParametricCRSPtr (*C++ type*), 639
osgeo::proj::crs::ProjectedCRS (*C++ class*), 650
osgeo::proj::crs::ProjectedCRS::baseCRS (*C++ function*), 651
osgeo::proj::crs::ProjectedCRS::coordinateSystem (*C++ function*), 651
osgeo::proj::crs::ProjectedCRS::create (*C++ function*), 652
osgeo::proj::crs::ProjectedCRS::demoteTo2D (*C++ function*), 651
osgeo::proj::crs::ProjectedCRS::identify (*C++ function*), 651
osgeo::proj::crs::ProjectedCRSNNPtr (*C++ type*), 639
osgeo::proj::crs::ProjectedCRSPtr (*C++ type*), 639
osgeo::proj::crs::SingleCRS (*C++ class*), 643
osgeo::proj::crs::SingleCRS::coordinateSystem (*C++ function*), 644
osgeo::proj::crs::SingleCRS::datum (*C++ function*), 643
osgeo::proj::crs::SingleCRS::datumEnsemble (*C++ function*), 643
osgeo::proj::crs::SingleCRSNNPtr (*C++ type*), 639
osgeo::proj::crs::SingleCRSPtr (*C++ type*), 639
osgeo::proj::crs::TemporalCRS (*C++ class*), 652
osgeo::proj::crs::TemporalCRS::coordinateSystem (*C++ function*), 652
osgeo::proj::crs::TemporalCRS::create (*C++ function*), 653
osgeo::proj::crs::TemporalCRS::datum (*C++ function*), 652
osgeo::proj::crs::TemporalCRSNNPtr (*C++ type*), 639
osgeo::proj::crs::TemporalCRSPtr (*C++ type*), 639
osgeo::proj::crs::VerticalCRS (*C++ class*), 648
osgeo::proj::crs::VerticalCRS::coordinateSystem (*C++ function*), 649
osgeo::proj::crs::VerticalCRS::create (*C++ function*), 649
osgeo::proj::crs::VerticalCRS::datum (*C++ function*), 649
osgeo::proj::crs::VerticalCRS::geoidModel (*C++ function*), 649
osgeo::proj::crs::VerticalCRS::identify (*C++ function*), 649
osgeo::proj::crs::VerticalCRS::velocityModel (*C++ function*), 649
osgeo::proj::crs::VerticalCRSNNPtr (*C++ type*), 638
osgeo::proj::crs::VerticalCRSPtr (*C++ type*), 638
osgeo::proj::cs (*C++ type*), 606
osgeo::proj::cs::AffineCS (*C++ class*), 619
osgeo::proj::cs::AffineCS::create (*C++ function*), 620
osgeo::proj::cs::AffineCS::WKT2_TYPE (*C++ member*), 620
osgeo::proj::cs::AffineCSNNPtr (*C++ type*), 607
osgeo::proj::cs::AffineCSPtr (*C++ type*), 607
osgeo::proj::cs::AxisDirection (*C++ class*), 608
osgeo::proj::cs::AxisDirection::AFT (*C++ member*), 610

osgeo::proj::cs::AxisDirection::AWAY_FROM (C++ member), 611
osgeo::proj::cs::AxisDirection::CLOCKWISE (C++ member), 610
osgeo::proj::cs::AxisDirection::COLUMN_NEGATIVE (C++ member), 610
osgeo::proj::cs::AxisDirection::COLUMN_POSITIVE (C++ member), 609
osgeo::proj::cs::AxisDirection::COUNTER_CLOCKWISE (C++ member), 610
osgeo::proj::cs::AxisDirection::DISPLAY_DOWN (C++ member), 610
osgeo::proj::cs::AxisDirection::DISPLAY_LEFT (C++ member), 610
osgeo::proj::cs::AxisDirection::DISPLAY_RIGHT (C++ member), 610
osgeo::proj::cs::AxisDirection::DISPLAY_UP (C++ member), 610
osgeo::proj::cs::AxisDirection::DOWN (C++ member), 609
osgeo::proj::cs::AxisDirection::EAST (C++ member), 608
osgeo::proj::cs::AxisDirection::EAST_NORTH_EAST (C++ member), 608
osgeo::proj::cs::AxisDirection::EAST_SOUTH_EAST (C++ member), 608
osgeo::proj::cs::AxisDirection::FORWARD (C++ member), 610
osgeo::proj::cs::AxisDirection::FUTURE (C++ member), 611
osgeo::proj::cs::AxisDirection::GEOCENTRIC_X (C++ member), 609
osgeo::proj::cs::AxisDirection::GEOCENTRIC_Y (C++ member), 609
osgeo::proj::cs::AxisDirection::GEOCENTRIC_Z (C++ member), 609
osgeo::proj::cs::AxisDirection::NORTH (C++ member), 608
osgeo::proj::cs::AxisDirection::NORTH_EAST (C++ member), 608
osgeo::proj::cs::AxisDirection::NORTH_NORTH_EAST (C++ member), 608
osgeo::proj::cs::AxisDirection::NORTH_NORTH_WEST (C++ member), 609
osgeo::proj::cs::AxisDirection::NORTH_WEST (C++ member), 609
osgeo::proj::cs::AxisDirection::PAST (C++ member), 611
osgeo::proj::cs::AxisDirection::PORT (C++ member), 610
osgeo::proj::cs::AxisDirection::ROW_NEGATIVE (C++ member), 610
osgeo::proj::cs::AxisDirection::ROW_POSITIVE (C++ member), 610
osgeo::proj::cs::AxisDirection::SOUTH (C++ member), 609
osgeo::proj::cs::AxisDirection::SOUTH_EAST (C++ member), 608
osgeo::proj::cs::AxisDirection::SOUTH_SOUTH_EAST (C++ member), 608
osgeo::proj::cs::AxisDirection::SOUTH_SOUTH_WEST (C++ member), 609
osgeo::proj::cs::AxisDirection::SOUTH_WEST (C++ member), 609
osgeo::proj::cs::AxisDirection::STARBOARD (C++ member), 610
osgeo::proj::cs::AxisDirection::TOWARDS (C++ member), 610
osgeo::proj::cs::AxisDirection::UNSPECIFIED (C++ member), 611
osgeo::proj::cs::AxisDirection::UP (C++ member), 609
osgeo::proj::cs::AxisDirection::WEST (C++ member), 609
osgeo::proj::cs::AxisDirection::WEST_NORTH_WEST (C++ member), 609
osgeo::proj::cs::AxisDirection::WEST_SOUTH_WEST (C++ member), 609
osgeo::proj::cs::CartesianCS (C++ class), 618
osgeo::proj::cs::CartesianCS::create (C++ function), 618
osgeo::proj::cs::CartesianCS::createEastingNorthing (C++ function), 618
osgeo::proj::cs::CartesianCS::createGeocentric (C++ function), 619
osgeo::proj::cs::CartesianCS::createNorthingEasting (C++ function), 618
osgeo::proj::cs::CartesianCS::createNorthPoleEastingSouthNorthingSouth (C++ function), 619
osgeo::proj::cs::CartesianCS::createSouthPoleEastingNorthNorthingNorth (C++ function), 619
osgeo::proj::cs::CartesianCS::createWestingSouthing (C++ function), 619
osgeo::proj::cs::CartesianCS::WKT2_TYPE (C++ member), 619
osgeo::proj::cs::CartesianCSNNPtr (C++ type), 607
osgeo::proj::cs::CartesianCSPtr (C++ type), 607
osgeo::proj::cs::CoordinateSystem (C++ class), 614
osgeo::proj::cs::CoordinateSystem::axisList (C++ function), 614
osgeo::proj::cs::CoordinateSystemAxis (C++ class), 612
osgeo::proj::cs::CoordinateSystemAxis::abbreviation (C++ function), 612

osgeo::proj::cs::CoordinateSystemAxis::create (*C++ function*), 613
osgeo::proj::cs::CoordinateSystemAxis::direction (*C++ function*), 612
osgeo::proj::cs::CoordinateSystemAxis::maximumValue (*C++ function*), 613
osgeo::proj::cs::CoordinateSystemAxis::meridian (*C++ function*), 613
osgeo::proj::cs::CoordinateSystemAxis::minimumValue (*C++ function*), 613
osgeo::proj::cs::CoordinateSystemAxis::rangeMeaning (*C++ function*), 613
osgeo::proj::cs::CoordinateSystemAxis::unit (*C++ function*), 612
osgeo::proj::cs::CoordinateSystemAxisNNPtr (*C++ type*), 606
osgeo::proj::cs::CoordinateSystemAxisPtr (*C++ type*), 606
osgeo::proj::cs::CoordinateSystemNNPtr (*C++ type*), 606
osgeo::proj::cs::CoordinateSystemPtr (*C++ type*), 606
osgeo::proj::cs::DateTimeTemporalCS (*C++ class*), 621
osgeo::proj::cs::DateTimeTemporalCS::create (*C++ function*), 622
osgeo::proj::cs::DateTimeTemporalCS::WKT2_2019_TYPE (*C++ member*), 622
osgeo::proj::cs::DateTimeTemporalCSNNPtr (*C++ type*), 607
osgeo::proj::cs::DateTimeTemporalCSPtr (*C++ type*), 607
osgeo::proj::cs::EllipsoidalCS (*C++ class*), 615
osgeo::proj::cs::EllipsoidalCS::create (*C++ function*), 616
osgeo::proj::cs::EllipsoidalCS::createLatitudeLongitude (*C++ function*), 616
osgeo::proj::cs::EllipsoidalCS::createLatitudeLongitudeEllipsoidalHeight (*C++ function*), 616
osgeo::proj::cs::EllipsoidalCS::createLongitudeLatitude (*C++ function*), 616
osgeo::proj::cs::EllipsoidalCS::createLongitudeLatitudeEllipsoidalHeight (*C++ function*), 616
osgeo::proj::cs::EllipsoidalCS::WKT2_TYPE (*C++ member*), 617
osgeo::proj::cs::EllipsoidalCSNNPtr (*C++ type*), 606
osgeo::proj::cs::EllipsoidalCSPtr (*C++ type*), 606
osgeo::proj::cs::Meridian (*C++ class*), 611
osgeo::proj::cs::Meridian::create (*C++ function*), 612
osgeo::proj::cs::Meridian::longitude (*C++ function*), 612
osgeo::proj::cs::MeridianNNPtr (*C++ type*), 606
osgeo::proj::cs::MeridianPtr (*C++ type*), 606
osgeo::proj::cs::OrdinalCS (*C++ class*), 620
osgeo::proj::cs::OrdinalCS::create (*C++ function*), 620
osgeo::proj::cs::OrdinalCS::WKT2_TYPE (*C++ member*), 621
osgeo::proj::cs::OrdinalCSNNPtr (*C++ type*), 607
osgeo::proj::cs::OrdinalCSPtr (*C++ type*), 607
osgeo::proj::cs::ParametricCS (*C++ class*), 621
osgeo::proj::cs::ParametricCS::create (*C++ function*), 621
osgeo::proj::cs::ParametricCS::WKT2_TYPE (*C++ member*), 621
osgeo::proj::cs::ParametricCSNNPtr (*C++ type*), 607
osgeo::proj::cs::ParametricCSPtr (*C++ type*), 607
osgeo::proj::cs::RangeMeaning (*C++ class*), 611
osgeo::proj::cs::RangeMeaning::EXACT (*C++ member*), 611
osgeo::proj::cs::RangeMeaning::WRAPAROUND (*C++ member*), 611
osgeo::proj::cs::SphericalCS (*C++ class*), 614
osgeo::proj::cs::SphericalCS::create (*C++ function*), 615
osgeo::proj::cs::SphericalCS::WKT2_TYPE (*C++ member*), 615
osgeo::proj::cs::SphericalCSNNPtr (*C++ type*), 606
osgeo::proj::cs::SphericalCSPtr (*C++ type*), 606
osgeo::proj::cs::TemporalCountCS (*C++ class*), 622
osgeo::proj::cs::TemporalCountCS::create (*C++ function*), 622
osgeo::proj::cs::TemporalCountCS::WKT2_2019_TYPE (*C++ member*), 623
osgeo::proj::cs::TemporalCountCSNNPtr (*C++ type*), 608
osgeo::proj::cs::TemporalCountCSPtr (*C++ type*), 607
osgeo::proj::cs::TemporalCS (*C++ class*), 621

osgeo::proj::cs::TemporalCS::WKT2_2015_TYPE (C++ member), 621
osgeo::proj::cs::TemporalCSNNPtr (C++ type), 607
osgeo::proj::cs::TemporalCSPtr (C++ type), 607
osgeo::proj::cs::TemporalMeasureCS (C++ class), 623
osgeo::proj::cs::TemporalMeasureCS::create (C++ function), 623
osgeo::proj::cs::TemporalMeasureCS::WKT2_2019_TYPE (C++ member), 623
osgeo::proj::cs::TemporalMeasureCSNNPtr (C++ type), 608
osgeo::proj::cs::TemporalMeasureCSPtr (C++ type), 608
osgeo::proj::cs::VerticalCS (C++ class), 617
osgeo::proj::cs::VerticalCS::create (C++ function), 617
osgeo::proj::cs::VerticalCS::createGravityRelatedHeight (C++ function), 617
osgeo::proj::cs::VerticalCS::WKT2_TYPE (C++ member), 618
osgeo::proj::cs::VerticalCSNNPtr (C++ type), 607
osgeo::proj::cs::VerticalCSPtr (C++ type), 607
osgeo::proj::datum (C++ type), 623
osgeo::proj::datum::Datum (C++ class), 625
osgeo::proj::datum::Datum::anchorDefinition (C++ function), 625
osgeo::proj::datum::Datum::anchorEpoch (C++ function), 625
osgeo::proj::datum::Datum::conventionalRS (C++ function), 626
osgeo::proj::datum::Datum::publicationDate (C++ function), 626
osgeo::proj::datum::DatumEnsemble (C++ class), 626
osgeo::proj::datum::DatumEnsemble::create (C++ function), 627
osgeo::proj::datum::DatumEnsemble::datums (C++ function), 626
osgeo::proj::datum::DatumEnsemble::positionalAccuracy (C++ function), 626
osgeo::proj::datum::DatumEnsembleNNPtr (C++ type), 623
osgeo::proj::datum::DatumEnsemblePtr (C++ type), 623
osgeo::proj::datum::DatumNNPtr (C++ type), 623
osgeo::proj::datum::DatumPtr (C++ type), 623
osgeo::proj::datum::DynamicGeodeticReferenceFrame (C++ class), 632
osgeo::proj::datum::DynamicGeodeticReferenceFrame::create (C++ function), 633
osgeo::proj::datum::DynamicGeodeticReferenceFrame::deformationModelName (C++ function), 633
osgeo::proj::datum::DynamicGeodeticReferenceFrame::frameReferenceEpoch (C++ function), 633
osgeo::proj::datum::DynamicGeodeticReferenceFrameNNPtr (C++ type), 624
osgeo::proj::datum::DynamicGeodeticReferenceFramePtr (C++ type), 624
osgeo::proj::datum::DynamicVerticalReferenceFrame (C++ class), 635
osgeo::proj::datum::DynamicVerticalReferenceFrame::create (C++ function), 636
osgeo::proj::datum::DynamicVerticalReferenceFrame::deformationModelName (C++ function), 635
osgeo::proj::datum::DynamicVerticalReferenceFrame::frameReferenceEpoch (C++ function), 635
osgeo::proj::datum::DynamicVerticalReferenceFrameNNPtr (C++ type), 624
osgeo::proj::datum::DynamicVerticalReferenceFramePtr (C++ type), 624
osgeo::proj::datum::Ellipsoid (C++ class), 628
osgeo::proj::datum::Ellipsoid::celestialBody (C++ function), 629
osgeo::proj::datum::Ellipsoid::CLARKE_1866 (C++ member), 630
osgeo::proj::datum::Ellipsoid::computedInverseFlattening (C++ function), 629
osgeo::proj::datum::Ellipsoid::computeSemiMinorAxis (C++ function), 629
osgeo::proj::datum::Ellipsoid::createFlattenedSphere (C++ function), 630
osgeo::proj::datum::Ellipsoid::createSphere (C++ function), 630
osgeo::proj::datum::Ellipsoid::createTwoAxis (C++ function), 630
osgeo::proj::datum::Ellipsoid::EARTH (C++ member), 630
osgeo::proj::datum::Ellipsoid::GRS1980 (C++ member), 631
osgeo::proj::datum::Ellipsoid::identify (C++ function), 629
osgeo::proj::datum::Ellipsoid::inverseFlattening (C++ function), 628
osgeo::proj::datum::Ellipsoid::isSphere (C++ function), 629
osgeo::proj::datum::Ellipsoid::semiMajorAxis (C++ function), 628

osgeo::proj::datum::Ellipsoid::semiMedianAxis (*C++ function*), 629
osgeo::proj::datum::Ellipsoid::semiMinorAxis (*C++ function*), 629
osgeo::proj::datum::Ellipsoid::squaredEccentricity (*C++ function*), 629
osgeo::proj::datum::Ellipsoid::WGS84 (*C++ member*), 630
osgeo::proj::datum::EllipsoidNNPtr (*C++ type*), 624
osgeo::proj::datum::EllipsoidPtr (*C++ type*), 624
osgeo::proj::datum::EngineeringDatum (*C++ class*), 637
osgeo::proj::datum::EngineeringDatum::create (*C++ function*), 637
osgeo::proj::datum::EngineeringDatumNNPtr (*C++ type*), 625
osgeo::proj::datum::EngineeringDatumPtr (*C++ type*), 624
osgeo::proj::datum::GeodeticReferenceFrame (*C++ class*), 631
osgeo::proj::datum::GeodeticReferenceFrame::create (*C++ function*), 632
osgeo::proj::datum::GeodeticReferenceFrame::ellipsoid (*C++ function*), 631
osgeo::proj::datum::GeodeticReferenceFrame::EPSG_6267 (*C++ member*), 632
osgeo::proj::datum::GeodeticReferenceFrame::EPSG_6269 (*C++ member*), 632
osgeo::proj::datum::GeodeticReferenceFrame::EPSG_6326 (*C++ member*), 632
osgeo::proj::datum::GeodeticReferenceFrame::primeMeridian (*C++ function*), 631
osgeo::proj::datum::GeodeticReferenceFrameNNPtr (*C++ type*), 624
osgeo::proj::datum::GeodeticReferenceFramePtr (*C++ type*), 624
osgeo::proj::datum::ParametricDatum (*C++ class*), 637
osgeo::proj::datum::ParametricDatum::create (*C++ function*), 638
osgeo::proj::datum::ParametricDatumNNPtr (*C++ type*), 625
osgeo::proj::datum::ParametricDatumPtr (*C++ type*), 625
osgeo::proj::datum::PrimeMeridian (*C++ class*), 627
osgeo::proj::datum::PrimeMeridian::create (*C++ function*), 628
osgeo::proj::datum::PrimeMeridian::GREENWICH (*C++ member*), 628
osgeo::proj::datum::PrimeMeridian::longitude (*C++ function*), 627
osgeo::proj::datum::PrimeMeridian::PARIS (*C++ member*), 628
osgeo::proj::datum::PrimeMeridian::REFERENCE_MERIDIAN (*C++ member*), 628
osgeo::proj::datum::PrimeMeridianNNPtr (*C++ type*), 624
osgeo::proj::datum::PrimeMeridianPtr (*C++ type*), 624
osgeo::proj::datum::RealizationMethod (*C++ class*), 633
osgeo::proj::datum::RealizationMethod::GEOID (*C++ member*), 634
osgeo::proj::datum::RealizationMethod::LEVELLING (*C++ member*), 634
osgeo::proj::datum::RealizationMethod::TIDAL (*C++ member*), 634
osgeo::proj::datum::TemporalDatum (*C++ class*), 636
osgeo::proj::datum::TemporalDatum::calendar (*C++ function*), 636
osgeo::proj::datum::TemporalDatum::CALENDAR_PROLEPTIC_GREGORIAN (*C++ member*), 637
osgeo::proj::datum::TemporalDatum::create (*C++ function*), 636
osgeo::proj::datum::TemporalDatum::temporalOrigin (*C++ function*), 636
osgeo::proj::datum::TemporalDatumNNPtr (*C++ type*), 624
osgeo::proj::datum::TemporalDatumPtr (*C++ type*), 624
osgeo::proj::datum::VerticalReferenceFrame (*C++ class*), 634
osgeo::proj::datum::VerticalReferenceFrame::create (*C++ function*), 634, 635
osgeo::proj::datum::VerticalReferenceFrame::realizationMethod (*C++ function*), 634
osgeo::proj::datum::VerticalReferenceFrameNNPtr (*C++ type*), 624
osgeo::proj::datum::VerticalReferenceFramePtr (*C++ type*), 624
osgeo::proj::io (*C++ type*), 732
osgeo::proj::io::asDouble (*C++ function*), 733
osgeo::proj::io::AuthorityFactory (*C++ class*), 746
osgeo::proj::io::AuthorityFactory::CelestialBodyInfo (*C++ struct*), 759
osgeo::proj::io::AuthorityFactory::CelestialBodyInfo::authName (*C++ member*), 760
osgeo::proj::io::AuthorityFactory::CelestialBodyInfo::name (*C++ member*), 760
osgeo::proj::io::AuthorityFactory::create (*C++ function*), 759

osgeo::proj::io::AuthorityFactory::createCompoundCRS (C++ function), 751
osgeo::proj::io::AuthorityFactory::createConversion (C++ function), 751
osgeo::proj::io::AuthorityFactory::createCoordinateMetadata (C++ function), 751
osgeo::proj::io::AuthorityFactory::createCoordinateOperation (C++ function), 752
osgeo::proj::io::AuthorityFactory::createCoordinateReferenceSystem (C++ function), 751
osgeo::proj::io::AuthorityFactory::createCoordinateSystem (C++ function), 750
osgeo::proj::io::AuthorityFactory::createDatum (C++ function), 749
osgeo::proj::io::AuthorityFactory::createDatumEnsemble (C++ function), 749
osgeo::proj::io::AuthorityFactory::createEllipsoid (C++ function), 749
osgeo::proj::io::AuthorityFactory::createEngineeringCRS (C++ function), 750
osgeo::proj::io::AuthorityFactory::createEngineeringDatum (C++ function), 749
osgeo::proj::io::AuthorityFactory::createExtent (C++ function), 748
osgeo::proj::io::AuthorityFactory::createFromCoordinateReferenceSystemCodes (C++ function), 752, 754
osgeo::proj::io::AuthorityFactory::createFromCRSCodesWithIntermediates (C++ function), 756
osgeo::proj::io::AuthorityFactory::createGeodeticCRS (C++ function), 750
osgeo::proj::io::AuthorityFactory::createGeodeticDatum (C++ function), 749
osgeo::proj::io::AuthorityFactory::createGeographicCRS (C++ function), 750
osgeo::proj::io::AuthorityFactory::createObject (C++ function), 748
osgeo::proj::io::AuthorityFactory::createObjectsFromName (C++ function), 759
osgeo::proj::io::AuthorityFactory::createPrimeMeridian (C++ function), 748
osgeo::proj::io::AuthorityFactory::createProjectedCRS (C++ function), 751
osgeo::proj::io::AuthorityFactory::createUnitOfMeasure (C++ function), 748
osgeo::proj::io::AuthorityFactory::createVerticalCRS (C++ function), 750
osgeo::proj::io::AuthorityFactory::createVerticalDatum (C++ function), 749
osgeo::proj::io::AuthorityFactory::CRSInfo (C++ struct), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::areaName (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::authName (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::bbox_valid (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::celestialBodyName (C++ member), 761
osgeo::proj::io::AuthorityFactory::CRSInfo::code (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::deprecated (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::east_lon_degree (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::name (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::north_lat_degree (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::projectionMethodName (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::south_lat_degree (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::type (C++ member), 760
osgeo::proj::io::AuthorityFactory::CRSInfo::west_lon_degree (C++ member), 760
osgeo::proj::io::AuthorityFactory::databaseContext (C++ function), 754
osgeo::proj::io::AuthorityFactory::getAuthority (C++ function), 753
osgeo::proj::io::AuthorityFactory::getAuthorityCodes (C++ function), 753
osgeo::proj::io::AuthorityFactory::getCelestialBodyList (C++ function), 754
osgeo::proj::io::AuthorityFactory::getCRSInfoList (C++ function), 753
osgeo::proj::io::AuthorityFactory::getDescriptionText (C++ function), 753
osgeo::proj::io::AuthorityFactory::getGeoidModels (C++ function), 752
osgeo::proj::io::AuthorityFactory::getOfficialNameFromAlias (C++ function), 758
osgeo::proj::io::AuthorityFactory::getUnitList (C++ function), 753
osgeo::proj::io::AuthorityFactory::identifyBodyFromSemiMajorAxis (C++ function), 748
osgeo::proj::io::AuthorityFactory::listAreaOfUseFromName (C++ function), 759
osgeo::proj::io::AuthorityFactory::ObjectType (C++ enum), 746
osgeo::proj::io::AuthorityFactory::ObjectType::COMPOUND_CRS (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::CONCATENATED_OPERATION (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::CONVERSION (C++ enumerator), 747

osgeo::proj::io::AuthorityFactory::ObjectType::COORDINATE_OPERATION (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::CRS (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::DATUM (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::DATUM_ENSEMBLE (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::DYNAMIC_GEODETIC_REFERENCE_FRAME (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::DYNAMIC_VERTICAL_REFERENCE_FRAME (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::ELLIPSOID (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::ENGINEERING_CRS (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::ENGINEERING_DATUM (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::GEOCENTRIC_CRS (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::GEODETIC_CRS (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::GEODETIC_REFERENCE_FRAME (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::GEOGRAPHIC_2D_CRS (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::GEOGRAPHIC_3D_CRS (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::GEOGRAPHIC_CRS (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::PRIME_MERIDIAN (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::ObjectType::PROJECTED_CRS (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::TRANSFORMATION (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::VERTICAL_CRS (C++ enumerator), 747
osgeo::proj::io::AuthorityFactory::ObjectType::VERTICAL_REFERENCE_FRAME (C++ enumerator), 746
osgeo::proj::io::AuthorityFactory::UnitInfo (C++ struct), 761
osgeo::proj::io::AuthorityFactory::UnitInfo::authName (C++ member), 761
osgeo::proj::io::AuthorityFactory::UnitInfo::category (C++ member), 761
osgeo::proj::io::AuthorityFactory::UnitInfo::code (C++ member), 761
osgeo::proj::io::AuthorityFactory::UnitInfo::convFactor (C++ member), 761
osgeo::proj::io::AuthorityFactory::UnitInfo::deprecated (C++ member), 761
osgeo::proj::io::AuthorityFactory::UnitInfo::name (C++ member), 761
osgeo::proj::io::AuthorityFactory::UnitInfo::projShortName (C++ member), 761
osgeo::proj::io::AuthorityFactoryNNPtr (C++ type), 733
osgeo::proj::io::AuthorityFactoryPtr (C++ type), 733
osgeo::proj::io::buildRethrow (C++ function), 733
osgeo::proj::io::cloneWithProps (C++ function), 733
osgeo::proj::io::createFromUserInput (C++ function), 733, 734
osgeo::proj::io::DatabaseContext (C++ class), 744
osgeo::proj::io::DatabaseContext::create (C++ function), 745
osgeo::proj::io::DatabaseContext::getAuthorities (C++ function), 744
osgeo::proj::io::DatabaseContext::getDatabaseStructure (C++ function), 744
osgeo::proj::io::DatabaseContext::getInsertStatementsFor (C++ function), 745
osgeo::proj::io::DatabaseContext::getMetadata (C++ function), 744
osgeo::proj::io::DatabaseContext::getPath (C++ function), 744
osgeo::proj::io::DatabaseContext::startInsertStatementsSession (C++ function), 744
osgeo::proj::io::DatabaseContext::stopInsertStatementsSession (C++ function), 745
osgeo::proj::io::DatabaseContext::suggestsCodeFor (C++ function), 744
osgeo::proj::io::DatabaseContextNNPtr (C++ type), 732
osgeo::proj::io::DatabaseContextPtr (C++ type), 732
osgeo::proj::io::FactoryException (C++ class), 761
osgeo::proj::io::FormattingException (C++ class), 740
osgeo::proj::io::IJSONExportable (C++ class), 739
osgeo::proj::io::IJSONExportable::exportToJSON (C++ function), 740
osgeo::proj::io::IPROJStringExportable (C++ class), 740
osgeo::proj::io::IPROJStringExportable::exportToPROJString (C++ function), 740
osgeo::proj::io::IPROJStringExportableNNPtr (C++ type), 733

osgeo::proj::io::IPROJStringExportablePtr (*C++ type*), 733
osgeo::proj::io::IWKTExportable (*C++ class*), 740
osgeo::proj::io::IWKTExportable::exportToWKT (*C++ function*), 740
osgeo::proj::io::JSONFormatter (*C++ class*), 739
osgeo::proj::io::JSONFormatter::create (*C++ function*), 739
osgeo::proj::io::JSONFormatter::setIndentationWidth (*C++ function*), 739
osgeo::proj::io::JSONFormatter::setMultiLine (*C++ function*), 739
osgeo::proj::io::JSONFormatter::setSchema (*C++ function*), 739
osgeo::proj::io::JSONFormatter::toString (*C++ function*), 739
osgeo::proj::io::JSONFormatterNNPtr (*C++ type*), 733
osgeo::proj::io::JSONFormatterPtr (*C++ type*), 733
osgeo::proj::io::NoSuchAuthorityCodeException (*C++ class*), 761
osgeo::proj::io::NoSuchAuthorityCodeException::getAuthority (*C++ function*), 761
osgeo::proj::io::NoSuchAuthorityCodeException::getAuthorityCode (*C++ function*), 761
osgeo::proj::io::ParsingException (*C++ class*), 740
osgeo::proj::io::PROJStringFormatter (*C++ class*), 738
osgeo::proj::io::PROJStringFormatter::Convention (*C++ enum*), 738
osgeo::proj::io::PROJStringFormatter::Convention::PROJ_4 (*C++ enumerator*), 738
osgeo::proj::io::PROJStringFormatter::Convention::PROJ_5 (*C++ enumerator*), 738
osgeo::proj::io::PROJStringFormatter::create (*C++ function*), 739
osgeo::proj::io::PROJStringFormatter::setIndentationWidth (*C++ function*), 738
osgeo::proj::io::PROJStringFormatter::setMaxLineLength (*C++ function*), 738
osgeo::proj::io::PROJStringFormatter::setMultiLine (*C++ function*), 738
osgeo::proj::io::PROJStringFormatter::setUseApproxTMerc (*C++ function*), 738
osgeo::proj::io::PROJStringFormatter::toString (*C++ function*), 738
osgeo::proj::io::PROJStringFormatterNNPtr (*C++ type*), 733
osgeo::proj::io::PROJStringFormatterPtr (*C++ type*), 733
osgeo::proj::io::PROJStringParser (*C++ class*), 743
osgeo::proj::io::PROJStringParser::attachDatabaseContext (*C++ function*), 743
osgeo::proj::io::PROJStringParser::createFromPROJString (*C++ function*), 744
osgeo::proj::io::PROJStringParser::setUsePROJ4InitRules (*C++ function*), 743
osgeo::proj::io::PROJStringParser::warningList (*C++ function*), 743
osgeo::proj::io::ThrowMissing (*C++ function*), 733
osgeo::proj::io::ThrowNotEnoughChildren (*C++ function*), 733
osgeo::proj::io::ThrowNotExpectedCSType (*C++ function*), 733
osgeo::proj::io::ThrowNotRequiredNumberOfChildren (*C++ function*), 733
osgeo::proj::io::WKTFormatter (*C++ class*), 735
osgeo::proj::io::WKTFormatter::Convention_ (*C++ enum*), 735
osgeo::proj::io::WKTFormatter::Convention_::_WKT1_ESRI (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::Convention_::_WKT1_GDAL (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::Convention_::_WKT2_2015 (*C++ enumerator*), 735
osgeo::proj::io::WKTFormatter::Convention_::_WKT2_2015_SIMPLIFIED (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::Convention_::_WKT2_2018 (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::Convention_::_WKT2_2018_SIMPLIFIED (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::Convention_::_WKT2_2019 (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::Convention_::_WKT2_2019_SIMPLIFIED (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::Convention_::WKT2 (*C++ enumerator*), 735
osgeo::proj::io::WKTFormatter::Convention_::WKT2_SIMPLIFIED (*C++ enumerator*), 735
osgeo::proj::io::WKTFormatter::create (*C++ function*), 737, 738
osgeo::proj::io::WKTFormatter::isAllowedEllipsoidalHeightAsVerticalCRS (*C++ function*), 737
osgeo::proj::io::WKTFormatter::isAllowedLINUNITNode (*C++ function*), 737
osgeo::proj::io::WKTFormatter::isStrict (*C++ function*), 737
osgeo::proj::io::WKTFormatter::OutputAxisRule (*C++ enum*), 736
osgeo::proj::io::WKTFormatter::OutputAxisRule::_WKT1_GDAL_EPSG_STYLE (*C++ enumerator*), 736

osgeo::proj::io::WKTFormatter::OutputAxisRule::NO (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::OutputAxisRule::YES (*C++ enumerator*), 736
osgeo::proj::io::WKTFormatter::setAllowEllipsoidalHeightAsVerticalCRS (*C++ function*), 737
osgeo::proj::io::WKTFormatter::setAllowLINUNITNode (*C++ function*), 737
osgeo::proj::io::WKTFormatter::setIndentationWidth (*C++ function*), 737
osgeo::proj::io::WKTFormatter::setMultiLine (*C++ function*), 737
osgeo::proj::io::WKTFormatter::setOutputAxis (*C++ function*), 737
osgeo::proj::io::WKTFormatter::setStrict (*C++ function*), 737
osgeo::proj::io::WKTFormatter::toString (*C++ function*), 737
osgeo::proj::io::WKTFormatterNNPtr (*C++ type*), 733
osgeo::proj::io::WKTFormatterPtr (*C++ type*), 733
osgeo::proj::io::WKTNode (*C++ class*), 741
osgeo::proj::io::WKTNode::addChild (*C++ function*), 741
osgeo::proj::io::WKTNode::children (*C++ function*), 741
osgeo::proj::io::WKTNode::countChildrenOfName (*C++ function*), 741
osgeo::proj::io::WKTNode::createFrom (*C++ function*), 742
osgeo::proj::io::WKTNode::lookForChild (*C++ function*), 741
osgeo::proj::io::WKTNode::toString (*C++ function*), 741
osgeo::proj::io::WKTNode::value (*C++ function*), 741
osgeo::proj::io::WKTNode::WKTNode (*C++ function*), 741
osgeo::proj::io::WKTNodeNNPtr (*C++ type*), 732
osgeo::proj::io::WKTNodePtr (*C++ type*), 732
osgeo::proj::io::WKTParser (*C++ class*), 742
osgeo::proj::io::WKTParser::attachDatabaseContext (*C++ function*), 742
osgeo::proj::io::WKTParser::createFromWKT (*C++ function*), 743
osgeo::proj::io::WKTParser::grammarErrorList (*C++ function*), 743
osgeo::proj::io::WKTParser::guessDialect (*C++ function*), 743
osgeo::proj::io::WKTParser::setStrict (*C++ function*), 742
osgeo::proj::io::WKTParser::setUnsetIdentifiersIfIncompatibleDef (*C++ function*), 743
osgeo::proj::io::WKTParser::warningList (*C++ function*), 742
osgeo::proj::io::WKTParser::WKTGuessedDialect (*C++ enum*), 742
osgeo::proj::io::WKTParser::WKTGuessedDialect::NOT_WKT (*C++ enumerator*), 742
osgeo::proj::io::WKTParser::WKTGuessedDialect::WKT1_ESRI (*C++ enumerator*), 742
osgeo::proj::io::WKTParser::WKTGuessedDialect::WKT1_GDAL (*C++ enumerator*), 742
osgeo::proj::io::WKTParser::WKTGuessedDialect::WKT2_2015 (*C++ enumerator*), 742
osgeo::proj::io::WKTParser::WKTGuessedDialect::WKT2_2018 (*C++ enumerator*), 742
osgeo::proj::io::WKTParser::WKTGuessedDialect::WKT2_2019 (*C++ enumerator*), 742
osgeo::proj::metadata (*C++ type*), 597
osgeo::proj::metadata::Citation (*C++ class*), 598
osgeo::proj::metadata::Citation::Citation (*C++ function*), 599
osgeo::proj::metadata::Citation::title (*C++ function*), 599
osgeo::proj::metadata::Extent (*C++ class*), 602
osgeo::proj::metadata::Extent::contains (*C++ function*), 602
osgeo::proj::metadata::Extent::create (*C++ function*), 603
osgeo::proj::metadata::Extent::createFromBBOX (*C++ function*), 603
osgeo::proj::metadata::Extent::description (*C++ function*), 602
osgeo::proj::metadata::Extent::geographicElements (*C++ function*), 602
osgeo::proj::metadata::Extent::intersection (*C++ function*), 602
osgeo::proj::metadata::Extent::intersects (*C++ function*), 602
osgeo::proj::metadata::Extent::temporalElements (*C++ function*), 602
osgeo::proj::metadata::Extent::verticalElements (*C++ function*), 602
osgeo::proj::metadata::Extent::WORLD (*C++ member*), 603
osgeo::proj::metadata::ExtentNNPtr (*C++ type*), 598
osgeo::proj::metadata::ExtentPtr (*C++ type*), 598

osgeo::proj::metadata::GeographicBoundingBox (*C++ class*), 599
osgeo::proj::metadata::GeographicBoundingBox::contains (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBox::create (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBox::eastBoundLongitude (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBox::intersection (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBox::intersects (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBox::northBoundLatitude (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBox::southBoundLatitude (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBox::westBoundLongitude (*C++ function*), 600
osgeo::proj::metadata::GeographicBoundingBoxNNPtr (*C++ type*), 598
osgeo::proj::metadata::GeographicBoundingBoxPtr (*C++ type*), 598
osgeo::proj::metadata::GeographicExtent (*C++ class*), 599
osgeo::proj::metadata::GeographicExtent::contains (*C++ function*), 599
osgeo::proj::metadata::GeographicExtent::intersection (*C++ function*), 599
osgeo::proj::metadata::GeographicExtent::intersects (*C++ function*), 599
osgeo::proj::metadata::GeographicExtentNNPtr (*C++ type*), 598
osgeo::proj::metadata::GeographicExtentPtr (*C++ type*), 598
osgeo::proj::metadata::Identifier (*C++ class*), 603
osgeo::proj::metadata::Identifier::authority (*C++ function*), 603
osgeo::proj::metadata::Identifier::AUTHORITY_KEY (*C++ member*), 605
osgeo::proj::metadata::Identifier::code (*C++ function*), 603
osgeo::proj::metadata::Identifier::CODE_KEY (*C++ member*), 605
osgeo::proj::metadata::Identifier::codeSpace (*C++ function*), 604
osgeo::proj::metadata::Identifier::CODESPACE_KEY (*C++ member*), 605
osgeo::proj::metadata::Identifier::create (*C++ function*), 604
osgeo::proj::metadata::Identifier::description (*C++ function*), 604
osgeo::proj::metadata::Identifier::DESCRIPTION_KEY (*C++ member*), 605
osgeo::proj::metadata::Identifier::EPSG (*C++ member*), 605
osgeo::proj::metadata::Identifier::isEquivalentName (*C++ function*), 604
osgeo::proj::metadata::Identifier::OGC (*C++ member*), 605
osgeo::proj::metadata::Identifier::uri (*C++ function*), 604
osgeo::proj::metadata::Identifier::URI_KEY (*C++ member*), 605
osgeo::proj::metadata::Identifier::version (*C++ function*), 604
osgeo::proj::metadata::Identifier::VERSION_KEY (*C++ member*), 605
osgeo::proj::metadata::IdentifierNNPtr (*C++ type*), 598
osgeo::proj::metadata::IdentifierPtr (*C++ type*), 598
osgeo::proj::metadata::PositionalAccuracy (*C++ class*), 605
osgeo::proj::metadata::PositionalAccuracy::create (*C++ function*), 606
osgeo::proj::metadata::PositionalAccuracy::value (*C++ function*), 606
osgeo::proj::metadata::PositionalAccuracyNNPtr (*C++ type*), 598
osgeo::proj::metadata::PositionalAccuracyPtr (*C++ type*), 598
osgeo::proj::metadata::TemporalExtent (*C++ class*), 600
osgeo::proj::metadata::TemporalExtent::contains (*C++ function*), 601
osgeo::proj::metadata::TemporalExtent::create (*C++ function*), 601
osgeo::proj::metadata::TemporalExtent::intersects (*C++ function*), 601
osgeo::proj::metadata::TemporalExtent::start (*C++ function*), 601
osgeo::proj::metadata::TemporalExtent::stop (*C++ function*), 601
osgeo::proj::metadata::TemporalExtentNNPtr (*C++ type*), 598
osgeo::proj::metadata::TemporalExtentPtr (*C++ type*), 598
osgeo::proj::metadata::VerticalExtent (*C++ class*), 601
osgeo::proj::metadata::VerticalExtent::contains (*C++ function*), 601
osgeo::proj::metadata::VerticalExtent::create (*C++ function*), 601
osgeo::proj::metadata::VerticalExtent::intersects (*C++ function*), 601
osgeo::proj::metadata::VerticalExtent::maximumValue (*C++ function*), 601

osgeo::proj::metadata::VerticalExtent::minimumValue (C++ function), 601
osgeo::proj::metadata::VerticalExtent::unit (C++ function), 601
osgeo::proj::metadata::VerticalExtentNNPtr (C++ type), 598
osgeo::proj::metadata::VerticalExtentPtr (C++ type), 598
osgeo::proj::operation (C++ type), 662
osgeo::proj::operation::ConcatenatedOperation (C++ class), 725
osgeo::proj::operation::ConcatenatedOperation::create (C++ function), 725
osgeo::proj::operation::ConcatenatedOperation::createComputeMetadata (C++ function), 725
osgeo::proj::operation::ConcatenatedOperation::gridsNeeded (C++ function), 725
osgeo::proj::operation::ConcatenatedOperation::inverse (C++ function), 725
osgeo::proj::operation::ConcatenatedOperation::operations (C++ function), 725
osgeo::proj::operation::ConcatenatedOperationNNPtr (C++ type), 664
osgeo::proj::operation::ConcatenatedOperationPtr (C++ type), 664
osgeo::proj::operation::Conversion (C++ class), 675
osgeo::proj::operation::Conversion::convertToOtherMethod (C++ function), 679
osgeo::proj::operation::Conversion::create (C++ function), 679, 680
osgeo::proj::operation::Conversion::createAlbersEqualArea (C++ function), 683
osgeo::proj::operation::Conversion::createAmericanPolyconic (C++ function), 703
osgeo::proj::operation::Conversion::createAxisOrderReversal (C++ function), 711
osgeo::proj::operation::Conversion::createAzimuthalEquidistant (C++ function), 686
osgeo::proj::operation::Conversion::createBonne (C++ function), 687
osgeo::proj::operation::Conversion::createCassiniSoldner (C++ function), 688
osgeo::proj::operation::Conversion::createChangeVerticalUnit (C++ function), 710
osgeo::proj::operation::Conversion::createEckertI (C++ function), 689
osgeo::proj::operation::Conversion::createEckertII (C++ function), 689
osgeo::proj::operation::Conversion::createEckertIII (C++ function), 689
osgeo::proj::operation::Conversion::createEckertIV (C++ function), 690
osgeo::proj::operation::Conversion::createEckertV (C++ function), 690
osgeo::proj::operation::Conversion::createEckertVI (C++ function), 690
osgeo::proj::operation::Conversion::createEqualEarth (C++ function), 708
osgeo::proj::operation::Conversion::createEquidistantConic (C++ function), 688
osgeo::proj::operation::Conversion::createEquidistantCylindrical (C++ function), 690
osgeo::proj::operation::Conversion::createEquidistantCylindricalSpherical (C++ function), 691
osgeo::proj::operation::Conversion::createGall (C++ function), 692
osgeo::proj::operation::Conversion::createGaussSchreiberTransverseMercator (C++ function), 680
osgeo::proj::operation::Conversion::createGeographic2DOffsets (C++ function), 711
osgeo::proj::operation::Conversion::createGeographic2DWithHeightOffsets (C++ function), 712
osgeo::proj::operation::Conversion::createGeographic3DOffsets (C++ function), 711
osgeo::proj::operation::Conversion::createGeographicGeocentric (C++ function), 711
osgeo::proj::operation::Conversion::createGeostationarySatelliteSweepX (C++ function), 693
osgeo::proj::operation::Conversion::createGeostationarySatelliteSweepY (C++ function), 693
osgeo::proj::operation::Conversion::createGnomonic (C++ function), 693
osgeo::proj::operation::Conversion::createGoodeHomolosine (C++ function), 692
osgeo::proj::operation::Conversion::createGuamProjection (C++ function), 687
osgeo::proj::operation::Conversion::createHeightDepthReversal (C++ function), 711
osgeo::proj::operation::Conversion::createHotineObliqueMercatorTwoPointNaturalOrigin (C++ function), 695
osgeo::proj::operation::Conversion::createHotineObliqueMercatorVariantA (C++ function), 694
osgeo::proj::operation::Conversion::createHotineObliqueMercatorVariantB (C++ function), 694
osgeo::proj::operation::Conversion::createInternationalMapWorldPolyconic (C++ function), 697
osgeo::proj::operation::Conversion::createInterruptedGoodeHomolosine (C++ function), 692
osgeo::proj::operation::Conversion::createKrovak (C++ function), 698
osgeo::proj::operation::Conversion::createKrovakNorthOriented (C++ function), 697

osgeo::proj::operation::Conversion::createLabordeObliqueMercator (*C++ function*), 696
osgeo::proj::operation::Conversion::createLambertAzimuthalEqualArea (*C++ function*), 699
osgeo::proj::operation::Conversion::createLambertConicConformal_1SP (*C++ function*), 683
osgeo::proj::operation::Conversion::createLambertConicConformal_1SP_VariantB (*C++ function*), 684
osgeo::proj::operation::Conversion::createLambertConicConformal_2SP (*C++ function*), 684
osgeo::proj::operation::Conversion::createLambertConicConformal_2SP_Belgium (*C++ function*), 685
osgeo::proj::operation::Conversion::createLambertConicConformal_2SP_Michigan (*C++ function*), 685
osgeo::proj::operation::Conversion::createLambertCylindricalEqualArea (*C++ function*), 688
osgeo::proj::operation::Conversion::createLambertCylindricalEqualAreaSpherical (*C++ function*), 687
osgeo::proj::operation::Conversion::createLocalOrthographic (*C++ function*), 702
osgeo::proj::operation::Conversion::createMercatorSpherical (*C++ function*), 701
osgeo::proj::operation::Conversion::createMercatorVariantA (*C++ function*), 699
osgeo::proj::operation::Conversion::createMercatorVariantB (*C++ function*), 700
osgeo::proj::operation::Conversion::createMillerCylindrical (*C++ function*), 699
osgeo::proj::operation::Conversion::createMollweide (*C++ function*), 701
osgeo::proj::operation::Conversion::createNewZealandMappingGrid (*C++ function*), 701
osgeo::proj::operation::Conversion::createObliqueStereographic (*C++ function*), 702
osgeo::proj::operation::Conversion::createOrthographic (*C++ function*), 702
osgeo::proj::operation::Conversion::createPolarStereographicVariantA (*C++ function*), 703
osgeo::proj::operation::Conversion::createPolarStereographicVariantB (*C++ function*), 704
osgeo::proj::operation::Conversion::createPoleRotationGRIBConvention (*C++ function*), 709
osgeo::proj::operation::Conversion::createPoleRotationNetCDFCFConvention (*C++ function*), 710
osgeo::proj::operation::Conversion::createPopularVisualisationPseudoMercator (*C++ function*), 700
osgeo::proj::operation::Conversion::createQuadrilateralizedSphericalCube (*C++ function*), 707
osgeo::proj::operation::Conversion::createRobinson (*C++ function*), 704
osgeo::proj::operation::Conversion::createSinusoidal (*C++ function*), 704
osgeo::proj::operation::Conversion::createSphericalCrossTrackHeight (*C++ function*), 708
osgeo::proj::operation::Conversion::createStereographic (*C++ function*), 705
osgeo::proj::operation::Conversion::createTransverseMercator (*C++ function*), 680
osgeo::proj::operation::Conversion::createTransverseMercatorSouthOriented (*C++ function*), 681
osgeo::proj::operation::Conversion::createTunisiaMappingGrid (*C++ function*), 682
osgeo::proj::operation::Conversion::createTunisiaMiningGrid (*C++ function*), 682
osgeo::proj::operation::Conversion::createTwoPointEquidistant (*C++ function*), 681
osgeo::proj::operation::Conversion::createUTM (*C++ function*), 680
osgeo::proj::operation::Conversion::createVanDerGrinten (*C++ function*), 705
osgeo::proj::operation::Conversion::createVerticalOffset (*C++ function*), 712
osgeo::proj::operation::Conversion::createVerticalPerspective (*C++ function*), 708
osgeo::proj::operation::Conversion::createWagnerI (*C++ function*), 705
osgeo::proj::operation::Conversion::createWagnerII (*C++ function*), 706
osgeo::proj::operation::Conversion::createWagnerIII (*C++ function*), 706
osgeo::proj::operation::Conversion::createWagnerIV (*C++ function*), 706
osgeo::proj::operation::Conversion::createWagnerV (*C++ function*), 706
osgeo::proj::operation::Conversion::createWagnerVI (*C++ function*), 707
osgeo::proj::operation::Conversion::createWagnerVII (*C++ function*), 707
osgeo::proj::operation::Conversion::identify (*C++ function*), 679
osgeo::proj::operation::Conversion::inverse (*C++ function*), 679
osgeo::proj::operation::Conversion::isUTM (*C++ function*), 679
osgeo::proj::operation::ConversionNNPtr (*C++ type*), 664
osgeo::proj::operation::ConversionPtr (*C++ type*), 664

osgeo::proj::operation::CoordinateOperation (*C++ class*), 666
osgeo::proj::operation::CoordinateOperation::coordinateOperationAccuracies (*C++ function*), 667
osgeo::proj::operation::CoordinateOperation::coordinateTransformer (*C++ function*), 667
osgeo::proj::operation::CoordinateOperation::gridsNeeded (*C++ function*), 668
osgeo::proj::operation::CoordinateOperation::hasBallparkTransformation (*C++ function*), 668
osgeo::proj::operation::CoordinateOperation::interpolationCRS (*C++ function*), 667
osgeo::proj::operation::CoordinateOperation::inverse (*C++ function*), 668
osgeo::proj::operation::CoordinateOperation::isPROJInstantiable (*C++ function*), 668
osgeo::proj::operation::CoordinateOperation::normalizeForVisualization (*C++ function*), 668
osgeo::proj::operation::CoordinateOperation::OPERATION_VERSION_KEY (*C++ member*), 668
osgeo::proj::operation::CoordinateOperation::operationVersion (*C++ function*), 667
osgeo::proj::operation::CoordinateOperation::requiresPerCoordinateInputTime (*C++ function*), 668
osgeo::proj::operation::CoordinateOperation::sourceCoordinateEpoch (*C++ function*), 667
osgeo::proj::operation::CoordinateOperation::sourceCRS (*C++ function*), 667
osgeo::proj::operation::CoordinateOperation::targetCoordinateEpoch (*C++ function*), 667
osgeo::proj::operation::CoordinateOperation::targetCRS (*C++ function*), 667
osgeo::proj::operation::CoordinateOperationContext (*C++ class*), 726
osgeo::proj::operation::CoordinateOperationContext::clone (*C++ function*), 729
osgeo::proj::operation::CoordinateOperationContext::create (*C++ function*), 730
osgeo::proj::operation::CoordinateOperationContext::getAllowBallparkTransformations (*C++ function*), 728
osgeo::proj::operation::CoordinateOperationContext::getAllowUseIntermediateCRS (*C++ function*), 729
osgeo::proj::operation::CoordinateOperationContext::getAreaOfInterest (*C++ function*), 727
osgeo::proj::operation::CoordinateOperationContext::getAuthorityFactory (*C++ function*), 727
osgeo::proj::operation::CoordinateOperationContext::getDesiredAccuracy (*C++ function*), 727
osgeo::proj::operation::CoordinateOperationContext::getDiscardSuperseded (*C++ function*), 728
osgeo::proj::operation::CoordinateOperationContext::getGridAvailabilityUse (*C++ function*), 729
osgeo::proj::operation::CoordinateOperationContext::getIntermediateCRS (*C++ function*), 729
osgeo::proj::operation::CoordinateOperationContext::getSourceAndTargetCRSExtentUse (*C++ function*), 728
osgeo::proj::operation::CoordinateOperationContext::getSourceCoordinateEpoch (*C++ function*), 729
osgeo::proj::operation::CoordinateOperationContext::getSpatialCriterion (*C++ function*), 728
osgeo::proj::operation::CoordinateOperationContext::getTargetCoordinateEpoch (*C++ function*), 729
osgeo::proj::operation::CoordinateOperationContext::getUsePROJAlternativeGridNames (*C++ function*), 728
osgeo::proj::operation::CoordinateOperationContext::GridAvailabilityUse (*C++ enum*), 727
osgeo::proj::operation::CoordinateOperationContext::GridAvailabilityUse::DISCARD_OPERATION_IF_MISSING_GI (*C++ enumerator*), 727
osgeo::proj::operation::CoordinateOperationContext::GridAvailabilityUse::IGNORE_GRID_AVAILABILITY (*C++ enumerator*), 727
osgeo::proj::operation::CoordinateOperationContext::GridAvailabilityUse::KNOWN_AVAILABLE (*C++ enumerator*), 727
osgeo::proj::operation::CoordinateOperationContext::GridAvailabilityUse::USE_FOR_SORTING (*C++ enumerator*), 727
osgeo::proj::operation::CoordinateOperationContext::IntermediateCRSUse (*C++ enum*), 727
osgeo::proj::operation::CoordinateOperationContext::IntermediateCRSUse::ALWAYS (*C++ enumerator*), 727

osgeo::proj::operation::CoordinateOperationContext::IntermediateCRSUse::IF_NO_DIRECT_TRANSFORMATION (C++ enumerator), 727
osgeo::proj::operation::CoordinateOperationContext::IntermediateCRSUse::NEVER (C++ enumerator), 727
osgeo::proj::operation::CoordinateOperationContext::setAllowBallparkTransformations (C++ function), 728
osgeo::proj::operation::CoordinateOperationContext::setAllowUseIntermediateCRS (C++ function), 729
osgeo::proj::operation::CoordinateOperationContext::setAreaOfInterest (C++ function), 727
osgeo::proj::operation::CoordinateOperationContext::setDesiredAccuracy (C++ function), 727
osgeo::proj::operation::CoordinateOperationContext::setDiscardSuperseded (C++ function), 728
osgeo::proj::operation::CoordinateOperationContext::setGridAvailabilityUse (C++ function), 728
osgeo::proj::operation::CoordinateOperationContext::setIntermediateCRS (C++ function), 729
osgeo::proj::operation::CoordinateOperationContext::setSourceAndTargetCRSExtentUse (C++ function), 728
osgeo::proj::operation::CoordinateOperationContext::setSourceCoordinateEpoch (C++ function), 729
osgeo::proj::operation::CoordinateOperationContext::setSpatialCriterion (C++ function), 728
osgeo::proj::operation::CoordinateOperationContext::setTargetCoordinateEpoch (C++ function), 729
osgeo::proj::operation::CoordinateOperationContext::setUsePROJAlternativeGridNames (C++ function), 728
osgeo::proj::operation::CoordinateOperationContext::SourceTargetCRSExtentUse (C++ enum), 726
osgeo::proj::operation::CoordinateOperationContext::SourceTargetCRSExtentUse::BOTH (C++ enumerator), 726
osgeo::proj::operation::CoordinateOperationContext::SourceTargetCRSExtentUse::INTERSECTION (C++ enumerator), 726
osgeo::proj::operation::CoordinateOperationContext::SourceTargetCRSExtentUse::NONE (C++ enumerator), 726
osgeo::proj::operation::CoordinateOperationContext::SourceTargetCRSExtentUse::SMALLEST (C++ enumerator), 726
osgeo::proj::operation::CoordinateOperationContext::SpatialCriterion (C++ enum), 726
osgeo::proj::operation::CoordinateOperationContext::SpatialCriterion::PARTIAL_INTERSECTION (C++ enumerator), 726
osgeo::proj::operation::CoordinateOperationContext::SpatialCriterion::STRICT_CONTAINMENT (C++ enumerator), 726
osgeo::proj::operation::CoordinateOperationContextNNPtr (C++ type), 664
osgeo::proj::operation::CoordinateOperationContextPtr (C++ type), 664
osgeo::proj::operation::CoordinateOperationFactory (C++ class), 730
osgeo::proj::operation::CoordinateOperationFactory::create (C++ function), 732
osgeo::proj::operation::CoordinateOperationFactory::createOperation (C++ function), 730
osgeo::proj::operation::CoordinateOperationFactory::createOperations (C++ function), 731, 732
osgeo::proj::operation::CoordinateOperationFactoryNNPtr (C++ type), 664
osgeo::proj::operation::CoordinateOperationFactoryPtr (C++ type), 664
osgeo::proj::operation::CoordinateOperationNNPtr (C++ type), 662
osgeo::proj::operation::CoordinateOperationPtr (C++ type), 662
osgeo::proj::operation::CoordinateTransformer (C++ class), 665
osgeo::proj::operation::CoordinateTransformer::transform (C++ function), 665
osgeo::proj::operation::CoordinateTransformerNNPtr (C++ type), 663
osgeo::proj::operation::CoordinateTransformerPtr (C++ type), 662
osgeo::proj::operation::createApproximateInverseIfPossible (C++ function), 665
osgeo::proj::operation::findCandidateGeodCRSForDatum (C++ function), 664

osgeo::proj::operation::GeneralOperationParameter (*C++ class*), 668
osgeo::proj::operation::GeneralOperationParameterNNPtr (*C++ type*), 663
osgeo::proj::operation::GeneralOperationParameterPtr (*C++ type*), 663
osgeo::proj::operation::GeneralParameterValue (*C++ class*), 669
osgeo::proj::operation::GeneralParameterValueNNPtr (*C++ type*), 663
osgeo::proj::operation::GeneralParameterValuePtr (*C++ type*), 663
osgeo::proj::operation::GridDescription (*C++ struct*), 665
osgeo::proj::operation::GridDescription::available (*C++ member*), 665
osgeo::proj::operation::GridDescription::directDownload (*C++ member*), 665
osgeo::proj::operation::GridDescription::fullName (*C++ member*), 665
osgeo::proj::operation::GridDescription::openLicense (*C++ member*), 665
osgeo::proj::operation::GridDescription::packageName (*C++ member*), 665
osgeo::proj::operation::GridDescription::shortName (*C++ member*), 665
osgeo::proj::operation::GridDescription::url (*C++ member*), 665
osgeo::proj::operation::InvalidOperation (*C++ class*), 673
osgeo::proj::operation::isRegularVerticalGridMethod (*C++ function*), 664
osgeo::proj::operation::negate (*C++ function*), 665
osgeo::proj::operation::OperationMethod (*C++ class*), 672
osgeo::proj::operation::OperationMethod::create (*C++ function*), 673
osgeo::proj::operation::OperationMethod::formula (*C++ function*), 673
osgeo::proj::operation::OperationMethod::formulaCitation (*C++ function*), 673
osgeo::proj::operation::OperationMethod::getEPSGCode (*C++ function*), 673
osgeo::proj::operation::OperationMethod::parameters (*C++ function*), 673
osgeo::proj::operation::OperationMethodNNPtr (*C++ type*), 663
osgeo::proj::operation::OperationMethodPtr (*C++ type*), 663
osgeo::proj::operation::OperationParameter (*C++ class*), 669
osgeo::proj::operation::OperationParameter::create (*C++ function*), 669
osgeo::proj::operation::OperationParameter::getEPSGCode (*C++ function*), 669
osgeo::proj::operation::OperationParameter::getNameForEPSGCode (*C++ function*), 669
osgeo::proj::operation::OperationParameterNNPtr (*C++ type*), 663
osgeo::proj::operation::OperationParameterPtr (*C++ type*), 663
osgeo::proj::operation::OperationParameterValue (*C++ class*), 671
osgeo::proj::operation::OperationParameterValue::create (*C++ function*), 672
osgeo::proj::operation::OperationParameterValue::parameter (*C++ function*), 672
osgeo::proj::operation::OperationParameterValue::parameterValue (*C++ function*), 672
osgeo::proj::operation::OperationParameterValueNNPtr (*C++ type*), 663
osgeo::proj::operation::OperationParameterValuePtr (*C++ type*), 663
osgeo::proj::operation::ParameterValue (*C++ class*), 670
osgeo::proj::operation::ParameterValue::booleanValue (*C++ function*), 671
osgeo::proj::operation::ParameterValue::create (*C++ function*), 671
osgeo::proj::operation::ParameterValue::createFilename (*C++ function*), 671
osgeo::proj::operation::ParameterValue::integerValue (*C++ function*), 671
osgeo::proj::operation::ParameterValue::stringValue (*C++ function*), 671
osgeo::proj::operation::ParameterValue::Type (*C++ enum*), 670
osgeo::proj::operation::ParameterValue::type (*C++ function*), 670
osgeo::proj::operation::ParameterValue::Type::BOOLEAN (*C++ enumerator*), 670
osgeo::proj::operation::ParameterValue::Type::FILENAME (*C++ enumerator*), 670
osgeo::proj::operation::ParameterValue::Type::INTEGER (*C++ enumerator*), 670
osgeo::proj::operation::ParameterValue::Type::MEASURE (*C++ enumerator*), 670
osgeo::proj::operation::ParameterValue::Type::STRING (*C++ enumerator*), 670
osgeo::proj::operation::ParameterValue::value (*C++ function*), 670
osgeo::proj::operation::ParameterValue::valueFile (*C++ function*), 671
osgeo::proj::operation::ParameterValueNNPtr (*C++ type*), 663
osgeo::proj::operation::ParameterValuePtr (*C++ type*), 663

osgeo::proj::operation::PointMotionOperation (*C++ class*), 723
osgeo::proj::operation::PointMotionOperation::create (*C++ function*), 724
osgeo::proj::operation::PointMotionOperation::inverse (*C++ function*), 724
osgeo::proj::operation::PointMotionOperation::sourceCRS (*C++ function*), 724
osgeo::proj::operation::PointMotionOperation::substitutePROJAlternativeGridNames (*C++ function*), 724
osgeo::proj::operation::PointMotionOperationNNPtr (*C++ type*), 664
osgeo::proj::operation::PointMotionOperationPtr (*C++ type*), 664
osgeo::proj::operation::SingleOperation (*C++ class*), 673
osgeo::proj::operation::SingleOperation::createPROJBased (*C++ function*), 675
osgeo::proj::operation::SingleOperation::gridsNeeded (*C++ function*), 674
osgeo::proj::operation::SingleOperation::method (*C++ function*), 674
osgeo::proj::operation::SingleOperation::parameterValue (*C++ function*), 674
osgeo::proj::operation::SingleOperation::parameterValueMeasure (*C++ function*), 674
osgeo::proj::operation::SingleOperation::parameterValues (*C++ function*), 674
osgeo::proj::operation::SingleOperation::substitutePROJAlternativeGridNames (*C++ function*), 675
osgeo::proj::operation::SingleOperation::validateParameters (*C++ function*), 675
osgeo::proj::operation::SingleOperationNNPtr (*C++ type*), 664
osgeo::proj::operation::SingleOperationPtr (*C++ type*), 664
osgeo::proj::operation::Transformation (*C++ class*), 712
osgeo::proj::operation::Transformation::create (*C++ function*), 713
osgeo::proj::operation::Transformation::createAbridgedMolodensky (*C++ function*), 719
osgeo::proj::operation::Transformation::createCartesianGridOffsets (*C++ function*), 722
osgeo::proj::operation::Transformation::createChangeVerticalUnit (*C++ function*), 723
osgeo::proj::operation::Transformation::createCoordinateFrameRotation (*C++ function*), 715
osgeo::proj::operation::Transformation::createGeocentricTranslations (*C++ function*), 714
osgeo::proj::operation::Transformation::createGeographic2DOffsets (*C++ function*), 721
osgeo::proj::operation::Transformation::createGeographic2DWithHeightOffsets (*C++ function*), 722
osgeo::proj::operation::Transformation::createGeographic3DOffsets (*C++ function*), 721
osgeo::proj::operation::Transformation::createGravityRelatedHeightToGeographic3D (*C++ function*), 720
osgeo::proj::operation::Transformation::createLongitudeRotation (*C++ function*), 721
osgeo::proj::operation::Transformation::createMolodensky (*C++ function*), 718
osgeo::proj::operation::Transformation::createNTv2 (*C++ function*), 718
osgeo::proj::operation::Transformation::createPositionVector (*C++ function*), 714
osgeo::proj::operation::Transformation::createTimeDependentCoordinateFrameRotation (*C++ function*), 716
osgeo::proj::operation::Transformation::createTimeDependentPositionVector (*C++ function*), 715
osgeo::proj::operation::Transformation::createTOWGS84 (*C++ function*), 718
osgeo::proj::operation::Transformation::createVERTCON (*C++ function*), 720
osgeo::proj::operation::Transformation::createVerticalOffset (*C++ function*), 723
osgeo::proj::operation::Transformation::inverse (*C++ function*), 713
osgeo::proj::operation::Transformation::sourceCRS (*C++ function*), 713
osgeo::proj::operation::Transformation::targetCRS (*C++ function*), 713
osgeo::proj::operation::TransformationNNPtr (*C++ type*), 663
osgeo::proj::operation::TransformationPtr (*C++ type*), 663
osgeo::proj::util (*C++ type*), 591
osgeo::proj::util::ArrayOfBaseObject (*C++ class*), 594
osgeo::proj::util::ArrayOfBaseObject::add (*C++ function*), 594
osgeo::proj::util::ArrayOfBaseObject::create (*C++ function*), 594
osgeo::proj::util::ArrayOfBaseObjectNNPtr (*C++ type*), 592
osgeo::proj::util::ArrayOfBaseObjectPtr (*C++ type*), 592

osgeo::proj::util::BaseObject (*C++ class*), 593
osgeo::proj::util::BaseObjectNNPtr (*C++ struct*), 593
osgeo::proj::util::BaseObjectPtr (*C++ type*), 592
osgeo::proj::util::BoxedValue (*C++ class*), 594
osgeo::proj::util::BoxedValue::BoxedValue (*C++ function*), 594
osgeo::proj::util::BoxedValueNNPtr (*C++ type*), 592
osgeo::proj::util::BoxedValuePtr (*C++ type*), 592
osgeo::proj::util::CodeList (*C++ class*), 597
osgeo::proj::util::CodeList::operator std::string (*C++ function*), 597
osgeo::proj::util::CodeList::toString (*C++ function*), 597
osgeo::proj::util::Exception (*C++ class*), 597
osgeo::proj::util::Exception::what (*C++ function*), 597
osgeo::proj::util::GenericName (*C++ class*), 595
osgeo::proj::util::GenericName::scope (*C++ function*), 595
osgeo::proj::util::GenericName::toFullyQualifiedName (*C++ function*), 595
osgeo::proj::util::GenericName::toString (*C++ function*), 595
osgeo::proj::util::GenericNameNNPtr (*C++ type*), 592
osgeo::proj::util::GenericNamePtr (*C++ type*), 592
osgeo::proj::util::IComparable (*C++ class*), 593
osgeo::proj::util::IComparable::Criterion (*C++ enum*), 593
osgeo::proj::util::IComparable::Criterion::EQUIVALENT (*C++ enumerator*), 593
osgeo::proj::util::IComparable::Criterion::EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS (*C++ enumerator*), 593
osgeo::proj::util::IComparable::Criterion::STRICT (*C++ enumerator*), 593
osgeo::proj::util::IComparable::isEquivalentTo (*C++ function*), 594
osgeo::proj::util::InvalidValueTypeException (*C++ class*), 597
osgeo::proj::util::LocalName (*C++ class*), 596
osgeo::proj::util::LocalName::scope (*C++ function*), 596
osgeo::proj::util::LocalName::toFullyQualifiedName (*C++ function*), 596
osgeo::proj::util::LocalName::toString (*C++ function*), 596
osgeo::proj::util::LocalNameNNPtr (*C++ type*), 592
osgeo::proj::util::LocalNamePtr (*C++ type*), 592
osgeo::proj::util::NameFactory (*C++ class*), 596
osgeo::proj::util::NameFactory::createGenericName (*C++ function*), 597
osgeo::proj::util::NameFactory::createLocalName (*C++ function*), 596
osgeo::proj::util::NameFactory::createNameSpace (*C++ function*), 596
osgeo::proj::util::NameSpace (*C++ class*), 595
osgeo::proj::util::NameSpace::isGlobal (*C++ function*), 595
osgeo::proj::util::NameSpace::name (*C++ function*), 595
osgeo::proj::util::NameSpaceNNPtr (*C++ type*), 592
osgeo::proj::util::NameSpacePtr (*C++ type*), 592
osgeo::proj::util::optional (*C++ class*), 592
osgeo::proj::util::optional::has_value (*C++ function*), 593
osgeo::proj::util::optional::operator bool (*C++ function*), 592
osgeo::proj::util::optional::operator* (*C++ function*), 592
osgeo::proj::util::optional::operator-> (*C++ function*), 592
osgeo::proj::util::PropertyMap (*C++ class*), 594
osgeo::proj::util::PropertyMap::set (*C++ function*), 594, 595
osgeo::proj::util::UnsupportedOperationException (*C++ class*), 597

## P

PJ (*C type*), 470
PJ_AREA (*C type*), 470
PJ_AXIS_DESCRIPTION (*C struct*), 494

PJ_CARTESIAN_CS_2D_TYPE (*C enum*), 493
PJ_CARTESIAN_CS_2D_TYPE.PJ_CART2D_EASTING_NORTHING (*C enumerator*), 493
PJ_CARTESIAN_CS_2D_TYPE.PJ_CART2D_NORTH_POLE_EASTING_SOUTH_NORTHING_SOUTH (*C enumerator*), 493
PJ_CARTESIAN_CS_2D_TYPE.PJ_CART2D_NORTHING_EASTING (*C enumerator*), 493
PJ_CARTESIAN_CS_2D_TYPE.PJ_CART2D_SOUTH_POLE_EASTING_NORTH_NORTHING_NORTH (*C enumerator*), 493
PJ_CARTESIAN_CS_2D_TYPE.PJ_CART2D_WESTING_SOUTHING (*C enumerator*), 493
PJ_CATEGORY (*C enum*), 485
PJ_CATEGORY.PJ_CATEGORY_COORDINATE_OPERATION (*C enumerator*), 485
PJ_CATEGORY.PJ_CATEGORY_CRS (*C enumerator*), 485
PJ_CATEGORY.PJ_CATEGORY_DATUM (*C enumerator*), 485
PJ_CATEGORY.PJ_CATEGORY_DATUM_ENSEMBLE (*C enumerator*), 485
PJ_CATEGORY.PJ_CATEGORY_ELLIPSOID (*C enumerator*), 485
PJ_CATEGORY.PJ_CATEGORY_PRIME_MERIDIAN (*C enumerator*), 485
PJ_COMPARISON_CRITERION (*C enum*), 487
PJ_COMPARISON_CRITERION.PJ_COMP_EQUIVALENT (*C enumerator*), 487
PJ_COMPARISON_CRITERION.PJ_COMP_EQUIVALENT_EXCEPT_AXIS_ORDER_GEOGCRS (*C enumerator*), 487
PJ_COMPARISON_CRITERION.PJ_COMP_STRICT (*C enumerator*), 487
PJ_CONTEXT (*C type*), 470
PJ_COORD (*C type*), 474
PJ_COORD.PJ_COORD.enu (*C member*), 475
PJ_COORD.PJ_COORD.geod (*C member*), 475
PJ_COORD.PJ_COORD.lp (*C member*), 475
PJ_COORD.PJ_COORD.lpz (*C member*), 475
PJ_COORD.PJ_COORD.lpzt (*C member*), 475
PJ_COORD.PJ_COORD.opk (*C member*), 475
PJ_COORD.PJ_COORD.uv (*C member*), 475
PJ_COORD.PJ_COORD.uvw (*C member*), 475
PJ_COORD.PJ_COORD.uvwt (*C member*), 475
PJ_COORD.PJ_COORD.xy (*C member*), 475
PJ_COORD.PJ_COORD.xyz (*C member*), 475
PJ_COORD.PJ_COORD.xyzt (*C member*), 474
PJ_COORD.v (*C member*), 474
PJ_COORDINATE_SYSTEM_TYPE (*C enum*), 489
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_CARTESIAN (*C enumerator*), 489
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_DATETIMETEMPORAL (*C enumerator*), 490
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_ELLIPSOIDAL (*C enumerator*), 489
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_ORDINAL (*C enumerator*), 489
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_PARAMETRIC (*C enumerator*), 489
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_SPHERICAL (*C enumerator*), 489
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_TEMPORALCOUNT (*C enumerator*), 490
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_TEMPORALMEASURE (*C enumerator*), 490
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_UNKNOWN (*C enumerator*), 489
PJ_COORDINATE_SYSTEM_TYPE.PJ_CS_TYPE_VERTICAL (*C enumerator*), 489
PJ_DIRECTION (*C type*), 470
PJ_ELLIPSOIDAL_CS_2D_TYPE (*C enum*), 494
PJ_ELLIPSOIDAL_CS_2D_TYPE.PJ_ELLPS2D_LATITUDE_LONGITUDE (*C enumerator*), 494
PJ_ELLIPSOIDAL_CS_2D_TYPE.PJ_ELLPS2D_LONGITUDE_LATITUDE (*C enumerator*), 494
PJ_ELLIPSOIDAL_CS_3D_TYPE (*C enum*), 494
PJ_ELLIPSOIDAL_CS_3D_TYPE.PJ_ELLPS3D_LATITUDE_LONGITUDE_HEIGHT (*C enumerator*), 494
PJ_ELLIPSOIDAL_CS_3D_TYPE.PJ_ELLPS3D_LONGITUDE_LATITUDE_HEIGHT (*C enumerator*), 494
PJ_ELLPS (*C type*), 477
PJ_ELLPS.ell (*C member*), 477
PJ_ELLPS.id (*C member*), 477
PJ_ELLPS.major (*C member*), 477

PJ_ELLPS.name (*C member*), 477
PJ_ENU (*C type*), 474
PJ_ENU.PJ_ENU.e (*C member*), 474
PJ_ENU.PJ_ENU.n (*C member*), 474
PJ_ENU.PJ_ENU.u (*C member*), 474
PJ_FACTORS (*C type*), 475
PJ_FACTORS.PJ_FACTORS.angular_distortion (*C member*), 476
PJ_FACTORS.PJ_FACTORS.areal_scale (*C member*), 476
PJ_FACTORS.PJ_FACTORS.dx_dlam (*C member*), 476
PJ_FACTORS.PJ_FACTORS.dx_dphi (*C member*), 476
PJ_FACTORS.PJ_FACTORS.dy_dlam (*C member*), 476
PJ_FACTORS.PJ_FACTORS.dy_dphi (*C member*), 476
PJ_FACTORS.PJ_FACTORS.meridian_parallel_angle (*C member*), 476
PJ_FACTORS.PJ_FACTORS.meridian_parallel_angle.PJ_FACTORS.meridian_convergence (*C member*), 476
PJ_FACTORS.PJ_FACTORS.meridional_scale (*C member*), 476
PJ_FACTORS.PJ_FACTORS.parallel_scale (*C member*), 476
PJ_FACTORS.PJ_FACTORS.tissot_semimajor (*C member*), 476
PJ_FACTORS.PJ_FACTORS.tissot_semiminor (*C member*), 476
PJ_FWD (*C++ enumerator*), 470
PJ_GEOD (*C type*), 474
PJ_GEOD.PJ_GEOD.a1 (*C member*), 474
PJ_GEOD.PJ_GEOD.a2 (*C member*), 474
PJ_GEOD.PJ_GEOD.s (*C member*), 474
PJ_GRID_INFO (*C type*), 479
PJ_GRID_INFO.PJ_GRID_INFO.cs_lat (*C member*), 480
PJ_GRID_INFO.PJ_GRID_INFO.cs_lon (*C member*), 479
PJ_GRID_INFO.PJ_GRID_INFO.filename (*C member*), 479
PJ_GRID_INFO.PJ_GRID_INFO.format (*C member*), 479
PJ_GRID_INFO.PJ_GRID_INFO.gridname (*C member*), 479
PJ_GRID_INFO.PJ_GRID_INFO.lowerleft (*C member*), 479
PJ_GRID_INFO.PJ_GRID_INFO.n_lat (*C member*), 479
PJ_GRID_INFO.PJ_GRID_INFO.n_lon (*C member*), 479
PJ_GRID_INFO.PJ_GRID_INFO.upperright (*C member*), 479
PJ_GUESSED_WKT_DIALECT (*C enum*), 484
PJ_GUESSED_WKT_DIALECT.PJ_GUESSED_NOT_WKT (*C enumerator*), 485
PJ_GUESSED_WKT_DIALECT.PJ_GUESSED_WKT1_ESRI (*C enumerator*), 485
PJ_GUESSED_WKT_DIALECT.PJ_GUESSED_WKT1_GDAL (*C enumerator*), 484
PJ_GUESSED_WKT_DIALECT.PJ_GUESSED_WKT2_2015 (*C enumerator*), 484
PJ_GUESSED_WKT_DIALECT.PJ_GUESSED_WKT2_2018 (*C enumerator*), 484
PJ_GUESSED_WKT_DIALECT.PJ_GUESSED_WKT2_2019 (*C enumerator*), 484
PJ_IDENT (*C++ enumerator*), 470
PJ_INFO (*C type*), 478
PJ_INFO.PJ_INFO.major (*C member*), 478
PJ_INFO.PJ_INFO.minor (*C member*), 478
PJ_INFO.PJ_INFO.patch (*C member*), 478
PJ_INFO.PJ_INFO.release (*C member*), 478
PJ_INFO.PJ_INFO.searchpath (*C member*), 478
PJ_INFO.PJ_INFO.version (*C member*), 478
PJ_INIT_INFO (*C type*), 480
PJ_INIT_INFO.PJ_INIT_INFO.filename (*C member*), 480
PJ_INIT_INFO.PJ_INIT_INFO.lastupdate (*C member*), 480
PJ_INIT_INFO.PJ_INIT_INFO.name (*C member*), 480
PJ_INIT_INFO.PJ_INIT_INFO.origin (*C member*), 480

PJ_INIT_INFO.PJ_INIT_INFO.version (*C member*), 480
PJ_INV (*C++ enumerator*), 470
PJ_LOG_DEBUG (*C++ enumerator*), 482
PJ_LOG_ERROR (*C++ enumerator*), 482
PJ_LOG_FUNC (*C type*), 482
PJ_LOG_LEVEL (*C type*), 481
PJ_LOG_NONE (*C++ enumerator*), 481
PJ_LOG_TELL (*C++ enumerator*), 482
PJ_LOG_TRACE (*C++ enumerator*), 482
PJ_LP (*C type*), 471
PJ_LP.PJ_LP.lam (*C member*), 471
PJ_LP.PJ_LP.phi (*C member*), 471
PJ_LPZ (*C type*), 471
PJ_LPZ.PJ_LPZ.lam (*C member*), 471
PJ_LPZ.PJ_LPZ.phi (*C member*), 471
PJ_LPZ.PJ_LPZ.z (*C member*), 472
PJ_LPZT (*C type*), 472
PJ_LPZT.PJ_LPZT.lam (*C member*), 472
PJ_LPZT.PJ_LPZT.phi (*C member*), 472
PJ_LPZT.PJ_LPZT.t (*C member*), 472
PJ_LPZT.PJ_LPZT.z (*C member*), 472
PJ_OPERATIONS (*C type*), 476
PJ_OPERATIONS.descr (*C member*), 477
PJ_OPERATIONS.id (*C member*), 476
PJ_OPERATIONS.op (*C member*), 476
PJ_OPK (*C type*), 473
PJ_OPK.PJ_OPK.k (*C member*), 473
PJ_OPK.PJ_OPK.o (*C member*), 473
PJ_OPK.PJ_OPK.p (*C member*), 473
PJ_PARAM_DESCRIPTION (*C struct*), 494
PJ_PRIME_MERIDIANS (*C type*), 477
PJ_PRIME_MERIDIANS.def (*C member*), 478
PJ_PRIME_MERIDIANS.id (*C member*), 478
PJ_PROJ_INFO (*C type*), 478
PJ_PROJ_INFO.PJ_PROJ_INFO.accuracy (*C member*), 479
PJ_PROJ_INFO.PJ_PROJ_INFO.definition (*C member*), 479
PJ_PROJ_INFO.PJ_PROJ_INFO.description (*C member*), 479
PJ_PROJ_INFO.PJ_PROJ_INFO.has_inverse (*C member*), 479
PJ_PROJ_INFO.PJ_PROJ_INFO.id (*C member*), 479
PJ_PROJ_STRING_TYPE (*C enum*), 488
PJ_PROJ_STRING_TYPE.PJ_PROJ_4 (*C enumerator*), 489
PJ_PROJ_STRING_TYPE.PJ_PROJ_5 (*C enumerator*), 488
PJ_TYPE (*C enum*), 485
PJ_TYPE.PJ_TYPE_BOUND_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_COMPOUND_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_CONCATENATED_OPERATION (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_CONVERSION (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_COORDINATE_METADATA (*C enumerator*), 487
PJ_TYPE.PJ_TYPE_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_DATUM_ENSEMBLE (*C enumerator*), 485
PJ_TYPE.PJ_TYPE_DERIVED_PROJECTED_CRS (*C enumerator*), 487
PJ_TYPE.PJ_TYPE_DYNAMIC_GEODETIC_REFERENCE_FRAME (*C enumerator*), 485
PJ_TYPE.PJ_TYPE_DYNAMIC_VERTICAL_REFERENCE_FRAME (*C enumerator*), 485
PJ_TYPE.PJ_TYPE_ELLIPSOID (*C enumerator*), 485

PJ_TYPE.PJ_TYPE_ENGINEERING_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_ENGINEERING_DATUM (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_GEOCENTRIC_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_GEODETIC_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_GEODETIC_REFERENCE_FRAME (*C enumerator*), 485
PJ_TYPE.PJ_TYPE_GEOGRAPHIC_2D_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_GEOGRAPHIC_3D_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_GEOGRAPHIC_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_OTHER_COORDINATE_OPERATION (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_OTHER_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_PARAMETRIC_DATUM (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_PRIME_MERIDIAN (*C enumerator*), 485
PJ_TYPE.PJ_TYPE_PROJECTED_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_TEMPORAL_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_TEMPORAL_DATUM (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_TRANSFORMATION (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_UNKNOWN (*C enumerator*), 485
PJ_TYPE.PJ_TYPE_VERTICAL_CRS (*C enumerator*), 486
PJ_TYPE.PJ_TYPE_VERTICAL_REFERENCE_FRAME (*C enumerator*), 485
PJ_UNIT_TYPE (*C enum*), 493
PJ_UNIT_TYPE.PJ_UT_ANGULAR (*C enumerator*), 493
PJ_UNIT_TYPE.PJ_UT_LINEAR (*C enumerator*), 493
PJ_UNIT_TYPE.PJ_UT_PARAMETRIC (*C enumerator*), 493
PJ_UNIT_TYPE.PJ_UT_SCALE (*C enumerator*), 493
PJ_UNIT_TYPE.PJ_UT_TIME (*C enumerator*), 493
PJ_UNITS (*C type*), 477
PJ_UNITS.factor (*C member*), 477
PJ_UNITS.id (*C member*), 477
PJ_UNITS.name (*C member*), 477
PJ_UNITS.to_meter (*C member*), 477
PJ_UV (*C type*), 471
PJ_UV.PJ_UV.u (*C member*), 471
PJ_UV.PJ_UV.v (*C member*), 471
PJ_UVW (*C type*), 472
PJ_UVW.PJ_UVW.u (*C member*), 472
PJ_UVW.PJ_UVW.v (*C member*), 472
PJ_UVW.PJ_UVW.w (*C member*), 472
PJ_UVWT (*C type*), 473
PJ_UVWT.PJ_UVWT.e (*C member*), 473
PJ_UVWT.PJ_UVWT.n (*C member*), 473
PJ_UVWT.PJ_UVWT.t (*C member*), 473
PJ_UVWT.PJ_UVWT.w (*C member*), 473
PJ_WKT_TYPE (*C enum*), 487
PJ_WKT_TYPE.PJ_WKT1_ESRI (*C enumerator*), 488
PJ_WKT_TYPE.PJ_WKT1_GDAL (*C enumerator*), 487
PJ_WKT_TYPE.PJ_WKT2_2015 (*C enumerator*), 487
PJ_WKT_TYPE.PJ_WKT2_2015_SIMPLIFIED (*C enumerator*), 487
PJ_WKT_TYPE.PJ_WKT2_2018 (*C enumerator*), 487
PJ_WKT_TYPE.PJ_WKT2_2018_SIMPLIFIED (*C enumerator*), 487
PJ_WKT_TYPE.PJ_WKT2_2019 (*C enumerator*), 487
PJ_WKT_TYPE.PJ_WKT2_2019_SIMPLIFIED (*C enumerator*), 487
PJ_XY (*C type*), 471
PJ_XY.PJ_XY.x (*C member*), 471
PJ_XY.PJ_XY.y (*C member*), 471

PJ_XYZ (*C type*), 472
PJ_XYZ.PJ_XYZ.x (*C member*), 472
PJ_XYZ.PJ_XYZ.y (*C member*), 472
PJ_XYZ.PJ_XYZ.z (*C member*), 472
PJ_XYZT (*C type*), 473
PJ_XYZT.PJ_XYZT.t (*C member*), 473
PJ_XYZT.PJ_XYZT.x (*C member*), 473
PJ_XYZT.PJ_XYZT.y (*C member*), 473
PJ_XYZT.PJ_XYZT.z (*C member*), 473
proj, 90
proj command line option
    -E, 91
    -I, 91
    -S, 91
    -V, 92
    -W<n>, 92
    -b, 91
    -d, 91
    -e, 91
    -f, 92
    -i, 91
    -lP, 91
    -l<[, 91
    -le, 91
    -lp, 91
    -lu, 91
    -m, 92
    -o, 91
    -r, 91
    -s, 91
    -t<a>, 91
    -v, 92
    -w<n>, 92
proj_alter_id (*C function*), 555
proj_alter_name (*C function*), 555
proj_angular_input (*C function*), 510
proj_angular_output (*C function*), 511
proj_area_create (*C function*), 498
proj_area_destroy (*C function*), 498
proj_area_set_bbox (*C function*), 498
proj_as_proj_string (*C function*), 527
proj_as_projjson (*C function*), 528
proj_as_wkt (*C function*), 526
PROJ_AT_LEAST_VERSION (*C macro*), 470
PROJ_AUX_DB, 99
PROJ_CELESTIAL_BODY_INFO (*C struct*), 492
PROJ_CELESTIAL_BODY_INFO.auth_name (*C var*), 493
PROJ_CELESTIAL_BODY_INFO.name (*C var*), 493
proj_celestial_body_list_destroy (*C function*), 531
proj_cleanup (*C function*), 518
proj_clone (*C function*), 522
PROJ_COMPUTE_VERSION (*C macro*), 470
proj_concatoperation_get_step (*C function*), 550
proj_concatoperation_get_step_count (*C function*), 549

proj_context_clone (*C function*), 494
proj_context_create (*C function*), 494
proj_context_destroy (*C function*), 494
proj_context_errno (*C function*), 503
proj_context_errno_string (*C function*), 505
proj_context_get_database_metadata (*C function*), 519
proj_context_get_database_path (*C function*), 519
proj_context_get_database_structure (*C function*), 519
proj_context_get_url_endpoint (*C function*), 514
proj_context_get_user_writable_directory (*C function*), 515
proj_context_guess_wkt_dialect (*C function*), 520
proj_context_is_network_enabled (*C function*), 514
proj_context_set_autoclose_database (*C function*), 518
proj_context_set_ca_bundle_path (*C function*), 513
proj_context_set_database_path (*C function*), 518
proj_context_set_enable_network (*C function*), 514
proj_context_set_file_finder (*C function*), 512
proj_context_set_fileapi (*C function*), 511
proj_context_set_network_callbacks (*C function*), 513
proj_context_set_search_paths (*C function*), 512
proj_context_set_sqlite3_vfs_name (*C function*), 512
proj_context_set_url_endpoint (*C function*), 514
proj_convert_conversion_to_other_method (*C function*), 561
proj_coord (*C function*), 508
proj_coordinate_metadata_create (*C function*), 550
proj_coordinate_metadata_get_epoch (*C function*), 550
proj_coordoperation_create_inverse (*C function*), 549
proj_coordoperation_get_accuracy (*C function*), 549
proj_coordoperation_get_grid_used (*C function*), 548
proj_coordoperation_get_grid_used_count (*C function*), 548
proj_coordoperation_get_method_info (*C function*), 546
proj_coordoperation_get_param (*C function*), 547
proj_coordoperation_get_param_count (*C function*), 547
proj_coordoperation_get_param_index (*C function*), 547
proj_coordoperation_get_towgs84_values (*C function*), 549
proj_coordoperation_has_ballpark_transformation (*C function*), 546
proj_coordoperation_is_instantiable (*C function*), 546
proj_coordoperation_requires_per_coordinate_input_time (*C function*), 546
proj_create (*C function*), 495
proj_create_argv (*C function*), 495
proj_create_cartesian_2D_cs (*C function*), 551
proj_create_compound_crs (*C function*), 560
proj_create_conversion (*C function*), 560
proj_create_conversion_albers_equal_area (*C function*), 565
proj_create_conversion_american_polyconic (*C function*), 576
proj_create_conversion_azimuthal_equidistant (*C function*), 567
proj_create_conversion_bonne (*C function*), 567
proj_create_conversion_cassini_soldner (*C function*), 568
proj_create_conversion_eckert_i (*C function*), 568
proj_create_conversion_eckert_ii (*C function*), 569
proj_create_conversion_eckert_iii (*C function*), 569
proj_create_conversion_eckert_iv (*C function*), 569
proj_create_conversion_eckert_v (*C function*), 569
proj_create_conversion_eckert_vi (*C function*), 569

proj_create_conversion_equal_earth (*C function*), 580
proj_create_conversion_equidistant_conic (*C function*), 568
proj_create_conversion_equidistant_cylindrical (*C function*), 570
proj_create_conversion_equidistant_cylindrical_spherical (*C function*), 570
proj_create_conversion_gall (*C function*), 570
proj_create_conversion_gauss_schreiber_transverse_mercator (*C function*), 563
proj_create_conversion_geostationary_satellite_sweep_x (*C function*), 571
proj_create_conversion_geostationary_satellite_sweep_y (*C function*), 571
proj_create_conversion_gnomonic (*C function*), 571
proj_create_conversion_goode_homolosine (*C function*), 570
proj_create_conversion_guam_projection (*C function*), 567
proj_create_conversion_hotine_oblique_mercator_two_point_natural_origin (*C function*), 572
proj_create_conversion_hotine_oblique_mercator_variant_a (*C function*), 571
proj_create_conversion_hotine_oblique_mercator_variant_b (*C function*), 572
proj_create_conversion_international_map_world_polyconic (*C function*), 573
proj_create_conversion_interrupted_goode_homolosine (*C function*), 570
proj_create_conversion_krovak (*C function*), 574
proj_create_conversion_krovak_north_oriented (*C function*), 574
proj_create_conversion_laborde_oblique_mercator (*C function*), 573
proj_create_conversion_lambert_azimuthal_equal_area (*C function*), 574
proj_create_conversion_lambert_conic_conformal_1sp (*C function*), 565
proj_create_conversion_lambert_conic_conformal_1sp_variant_b (*C function*), 565
proj_create_conversion_lambert_conic_conformal_2sp (*C function*), 566
proj_create_conversion_lambert_conic_conformal_2sp_belgium (*C function*), 566
proj_create_conversion_lambert_conic_conformal_2sp_michigan (*C function*), 566
proj_create_conversion_lambert_cylindrical_equal_area (*C function*), 568
proj_create_conversion_lambert_cylindrical_equal_area_spherical (*C function*), 567
proj_create_conversion_local_orthographic (*C function*), 576
proj_create_conversion_mercator_variant_a (*C function*), 575
proj_create_conversion_mercator_variant_b (*C function*), 575
proj_create_conversion_miller_cylindrical (*C function*), 574
proj_create_conversion_mollweide (*C function*), 575
proj_create_conversion_new_zealand_mapping_grid (*C function*), 575
proj_create_conversion_oblique_stereographic (*C function*), 576
proj_create_conversion_orthographic (*C function*), 576
proj_create_conversion_polar_stereographic_variant_a (*C function*), 577
proj_create_conversion_polar_stereographic_variant_b (*C function*), 577
proj_create_conversion_pole_rotation_grib_convention (*C function*), 580
proj_create_conversion_pole_rotation_netcdf_cf_convention (*C function*), 580
proj_create_conversion_popular_visualisation_pseudo_mercator (*C function*), 575
proj_create_conversion_quadrilateralized_spherical_cube (*C function*), 579
proj_create_conversion_robinson (*C function*), 577
proj_create_conversion_sinusoidal (*C function*), 577
proj_create_conversion_spherical_cross_track_height (*C function*), 579
proj_create_conversion_stereographic (*C function*), 577
proj_create_conversion_transverse_mercator (*C function*), 563
proj_create_conversion_transverse_mercator_south_oriented (*C function*), 564
proj_create_conversion_tunisia_mapping_grid (*C function*), 564
proj_create_conversion_tunisia_mining_grid (*C function*), 565
proj_create_conversion_two_point_equidistant (*C function*), 564
proj_create_conversion_utm (*C function*), 563
proj_create_conversion_van_der_grinten (*C function*), 578
proj_create_conversion_vertical_perspective (*C function*), 580
proj_create_conversion_wagner_i (*C function*), 578

proj_create_conversion_wagner_ii (*C function*), 578
proj_create_conversion_wagner_iii (*C function*), 578
proj_create_conversion_wagner_iv (*C function*), 578
proj_create_conversion_wagner_v (*C function*), 578
proj_create_conversion_wagner_vi (*C function*), 579
proj_create_conversion_wagner_vii (*C function*), 579
proj_create_crs_to_crs (*C function*), 496
proj_create_crs_to_crs_from_pj (*C function*), 497
proj_create_cs (*C function*), 550
proj_create_derived_geographic_crs (*C function*), 554
proj_create_ellipsoidal_2D_cs (*C function*), 551
proj_create_ellipsoidal_3D_cs (*C function*), 551
proj_create_engineering_crs (*C function*), 558
proj_create_from_database (*C function*), 520
proj_create_from_name (*C function*), 522
proj_create_from_wkt (*C function*), 520
proj_create_geocentric_crs (*C function*), 553
proj_create_geocentric_crs_from_datum (*C function*), 554
proj_create_geographic_crs (*C function*), 552
proj_create_geographic_crs_from_datum (*C function*), 553
proj_create_operation_factory_context (*C function*), 534
proj_create_operations (*C function*), 538
proj_create_projected_crs (*C function*), 562
proj_create_transformation (*C function*), 561
proj_create_vertical_crs (*C function*), 559
proj_create_vertical_crs_ex (*C function*), 559
proj_crs_alter_cs_angular_unit (*C function*), 556
proj_crs_alter_cs_linear_unit (*C function*), 556
proj_crs_alter_geodetic_crs (*C function*), 555
proj_crs_alter_parameters_linear_unit (*C function*), 557
proj_crs_create_bound_crs (*C function*), 562
proj_crs_create_bound_crs_to_WGS84 (*C function*), 562
proj_crs_create_bound_vertical_crs (*C function*), 563
proj_crs_create_projected_3D_crs_from_2D (*C function*), 558
proj_crs_demote_to_2D (*C function*), 558
PROJ_CRS_EXTENT_USE (*C enum*), 488
PROJ_CRS_EXTENT_USE.PJ_CRS_EXTENT_BOTH (*C enumerator*), 488
PROJ_CRS_EXTENT_USE.PJ_CRS_EXTENT_INTERSECTION (*C enumerator*), 488
PROJ_CRS_EXTENT_USE.PJ_CRS_EXTENT_NONE (*C enumerator*), 488
PROJ_CRS_EXTENT_USE.PJ_CRS_EXTENT_SMALLEST (*C enumerator*), 488
proj_crs_get_coordinate_system (*C function*), 543
proj_crs_get_coordoperation (*C function*), 545
proj_crs_get_datum (*C function*), 540
proj_crs_get_datum_ensemble (*C function*), 541
proj_crs_get_datum_forced (*C function*), 541
proj_crs_get_geodetic_crs (*C function*), 539
proj_crs_get_horizontal_datum (*C function*), 540
proj_crs_get_sub_crs (*C function*), 540
proj_crs_has_point_motion_operation (*C function*), 541
PROJ_CRS_INFO (*C struct*), 490
PROJ_CRS_INFO.area_name (*C var*), 491
PROJ_CRS_INFO.auth_name (*C var*), 490
PROJ_CRS_INFO.bbox_valid (*C var*), 490
PROJ_CRS_INFO.celestial_body_name (*C var*), 491

PROJ_CRS_INFO.code (*C var*), 490
PROJ_CRS_INFO.deprecated (*C var*), 490
PROJ_CRS_INFO.east_lon_degree (*C var*), 490
PROJ_CRS_INFO.name (*C var*), 490
PROJ_CRS_INFO.north_lat_degree (*C var*), 490
PROJ_CRS_INFO.projection_method_name (*C var*), 491
PROJ_CRS_INFO.south_lat_degree (*C var*), 490
PROJ_CRS_INFO.type (*C var*), 490
PROJ_CRS_INFO.west_lon_degree (*C var*), 490
proj_crs_info_list_destroy (*C function*), 532
proj_crs_is_derived (*C function*), 539
PROJ_CRS_LIST_PARAMETERS (*C struct*), 491
PROJ_CRS_LIST_PARAMETERS.allow_deprecated (*C var*), 492
PROJ_CRS_LIST_PARAMETERS.bbox_valid (*C var*), 491
PROJ_CRS_LIST_PARAMETERS.celestial_body_name (*C var*), 492
PROJ_CRS_LIST_PARAMETERS.crs_area_of_use_contains_bbox (*C var*), 491
PROJ_CRS_LIST_PARAMETERS.east_lon_degree (*C var*), 491
PROJ_CRS_LIST_PARAMETERS.north_lat_degree (*C var*), 491
PROJ_CRS_LIST_PARAMETERS.south_lat_degree (*C var*), 491
PROJ_CRS_LIST_PARAMETERS.types (*C var*), 491
PROJ_CRS_LIST_PARAMETERS.typesCount (*C var*), 491
PROJ_CRS_LIST_PARAMETERS.west_lon_degree (*C var*), 491
proj_crs_promote_to_3D (*C function*), 557
proj_cs_get_axis_count (*C function*), 543
proj_cs_get_axis_info (*C function*), 543
proj_cs_get_type (*C function*), 543
PROJ_DATA, 69, 70, 81, 92, 96, 447, 448, 450, 453, 881
proj_datum_ensemble_get_accuracy (*C function*), 542
proj_datum_ensemble_get_member (*C function*), 542
proj_datum_ensemble_get_member_count (*C function*), 542
PROJ_DB_CACHE_DIR
    command line option, 53
PROJ_DEBUG, 9
proj_degree_input (*C function*), 511
proj_degree_output (*C function*), 511
proj_destroy (*C function*), 498
proj_dmstor (*C function*), 510
proj_download_file (*C function*), 517
proj_dynamic_datum_get_frame_reference_epoch (*C function*), 542
proj_ellipsoid_get_parameters (*C function*), 544
PROJ_ERR_COORD_TRANSFM (*C macro*), 481
PROJ_ERR_COORD_TRANSFM_GRID_AT_NODATA (*C macro*), 481
PROJ_ERR_COORD_TRANSFM_INVALID_COORD (*C macro*), 481
PROJ_ERR_COORD_TRANSFM_NO_OPERATION (*C macro*), 481
PROJ_ERR_COORD_TRANSFM_OUTSIDE_GRID (*C macro*), 481
PROJ_ERR_COORD_TRANSFM_OUTSIDE_PROJECTION_DOMAIN (*C macro*), 481
PROJ_ERR_INVALID_OP (*C macro*), 480
PROJ_ERR_INVALID_OP_FILE_NOT_FOUND_OR_INVALID (*C macro*), 481
PROJ_ERR_INVALID_OP_ILLEGAL_ARG_VALUE (*C macro*), 481
PROJ_ERR_INVALID_OP_MISSING_ARG (*C macro*), 481
PROJ_ERR_INVALID_OP_MUTUALLY_EXCLUSIVE_ARGS (*C macro*), 481
PROJ_ERR_INVALID_OP_WRONG_SYNTAX (*C macro*), 480
PROJ_ERR_OTHER (*C macro*), 481
PROJ_ERR_OTHER_API_MISUSE (*C macro*), 481

PROJ_ERR_OTHER_NETWORK_ERROR (*C macro*), 481
PROJ_ERR_OTHER_NO_INVERSE_OP (*C macro*), 481
proj_errno (*C function*), 503
proj_errno_reset (*C function*), 504
proj_errno_restore (*C function*), 504
proj_errno_set (*C function*), 504
proj_errno_string (*C function*), 505
proj_factors (*C function*), 509
PROJ_FILE_API (*C struct*), 482
PROJ_FILE_API.close_cbk (*C var*), 483
PROJ_FILE_API.exists_cbk (*C var*), 483
PROJ_FILE_API.mkdir_cbk (*C var*), 483
PROJ_FILE_API.open_cbk (*C var*), 482
PROJ_FILE_API.read_cbk (*C var*), 482
PROJ_FILE_API.rename_cbk (*C var*), 483
PROJ_FILE_API.seek_cbk (*C var*), 482
PROJ_FILE_API.tell_cbk (*C var*), 483
PROJ_FILE_API.unlink_cbk (*C var*), 483
PROJ_FILE_API.version (*C var*), 482
PROJ_FILE_API.write_cbk (*C var*), 482
PROJ_FILE_HANDLE (*C type*), 483
proj_geod (*C function*), 508
proj_get_area_of_use (*C function*), 525
proj_get_area_of_use_ex (*C function*), 526
proj_get_authorities_from_database (*C function*), 530
proj_get_celestial_body_list_from_database (*C function*), 530
proj_get_celestial_body_name (*C function*), 544
proj_get_codes_from_database (*C function*), 530
proj_get_crs_info_list_from_database (*C function*), 531
proj_get_crs_list_parameters_create (*C function*), 531
proj_get_crs_list_parameters_destroy (*C function*), 531
proj_get_domain_count (*C function*), 524
proj_get_ellipsoid (*C function*), 544
proj_get_geoid_models_from_database (*C function*), 529
proj_get_id_auth_name (*C function*), 524
proj_get_id_code (*C function*), 524
proj_get_insert_statements (*C function*), 533
proj_get_name (*C function*), 524
proj_get_non_deprecated (*C function*), 523
proj_get_prime_meridian (*C function*), 545
proj_get_remarks (*C function*), 524
proj_get_scope (*C function*), 525
proj_get_scope_ex (*C function*), 525
proj_get_source_crs (*C function*), 528
proj_get_suggested_operation (*C function*), 539
proj_get_target_crs (*C function*), 528
proj_get_type (*C function*), 522
proj_get_units_from_database (*C function*), 532
PROJ_GRID_AVAILABILITY_USE (*C enum*), 488
PROJ_GRID_AVAILABILITY_USE.PROJ_GRID_AVAILABILITY_DISCARD_OPERATION_IF_MISSING_GRID (*C enumerator*), 488
PROJ_GRID_AVAILABILITY_USE.PROJ_GRID_AVAILABILITY_IGNORED (*C enumerator*), 488
PROJ_GRID_AVAILABILITY_USE.PROJ_GRID_AVAILABILITY_KNOWN_AVAILABLE (*C enumerator*), 488
PROJ_GRID_AVAILABILITY_USE.PROJ_GRID_AVAILABILITY_USED_FOR_SORTING (*C enumerator*), 488

proj_grid_cache_clear (*C function*), 516
proj_grid_cache_set_enable (*C function*), 515
proj_grid_cache_set_filename (*C function*), 515
proj_grid_cache_set_max_size (*C function*), 516
proj_grid_cache_set_ttl (*C function*), 516
proj_grid_get_info_from_database (*C function*), 521
proj_grid_info (*C function*), 506
proj_identify (*C function*), 529
proj_info (*C function*), 506
proj_init_info (*C function*), 506
proj_insert_object_session_create (*C function*), 532
proj_insert_object_session_destroy (*C function*), 533
proj_int_list_destroy (*C function*), 530
PROJ_INTERMEDIATE_CRS_USE (*C enum*), 489
PROJ_INTERMEDIATE_CRS_USE.PROJ_INTERMEDIATE_CRS_USE_ALWAYS (*C enumerator*), 489
PROJ_INTERMEDIATE_CRS_USE.PROJ_INTERMEDIATE_CRS_USE_IF_NO_DIRECT_TRANSFORMATION (*C enumerator*), 489
PROJ_INTERMEDIATE_CRS_USE.PROJ_INTERMEDIATE_CRS_USE_NEVER (*C enumerator*), 489
proj_is_crs (*C function*), 524
proj_is_deprecated (*C function*), 523
proj_is_derived_crs (*C function*), 555
proj_is_download_needed (*C function*), 516
proj_is_equivalent_to (*C function*), 523
proj_is_equivalent_to_with_ctx (*C function*), 523
PROJ_LIB, 22, 34, 36, 70, 447
proj_list_destroy (*C function*), 539
proj_list_ellps (*C function*), 506
proj_list_get (*C function*), 538
proj_list_get_count (*C function*), 538
proj_list_operations (*C function*), 506
proj_list_prime_meridians (*C function*), 507
proj_list_units (*C function*), 506
proj_log_func (*C function*), 505
proj_log_level (*C function*), 505
proj_lp_dist (*C function*), 507
proj_lpz_dist (*C function*), 507
PROJ_NATIVE_CA, 4
PROJ_NETWORK, 50, 70, 73, 77, 82, 96, 448
proj_network_close_cbk_type (*C type*), 484
PROJ_NETWORK_ENDPOINT, 73
proj_network_get_header_value_cbk_type (*C type*), 484
PROJ_NETWORK_HANDLE (*C type*), 483
proj_network_open_cbk_type (*C type*), 483
proj_network_read_range_type (*C type*), 484
proj_normalize_for_visualization (*C function*), 497
PROJ_ONLY_BEST_DEFAULT, 80, 497
PROJ_OPEN_ACCESS (*C enum*), 483
PROJ_OPEN_ACCESS.PROJ_OPEN_ACCESS_CREATE (*C enumerator*), 483
PROJ_OPEN_ACCESS.PROJ_OPEN_ACCESS_READ_ONLY (*C enumerator*), 483
PROJ_OPEN_ACCESS.PROJ_OPEN_ACCESS_READ_UPDATE (*C enumerator*), 483
proj_operation_factory_context_destroy (*C function*), 535
proj_operation_factory_context_set_allow_ballpark_transformations (*C function*), 538
proj_operation_factory_context_set_allow_use_intermediate_crs (*C function*), 537
proj_operation_factory_context_set_allowed_intermediate_crs (*C function*), 537

proj_operation_factory_context_set_area_of_interest (*C function*), 535
proj_operation_factory_context_set_area_of_interest_name (*C function*), 535
proj_operation_factory_context_set_crs_extent_use (*C function*), 536
proj_operation_factory_context_set_desired_accuracy (*C function*), 535
proj_operation_factory_context_set_discard_superseded (*C function*), 537
proj_operation_factory_context_set_grid_availability_use (*C function*), 536
proj_operation_factory_context_set_spatial_criterion (*C function*), 536
proj_operation_factory_context_set_use_proj_alternative_grid_names (*C function*), 536
PROJ_OUTPUT_NAME
    command line option, 51
proj_pj_info (*C function*), 506
proj_prime_meridian_get_parameters (*C function*), 545
proj_query_geodetic_crs_from_datum (*C function*), 552
proj_roundtrip (*C function*), 509
proj_rtodms (*C function*), 510
proj_rtodms2 (*C function*), 510
PROJ_SPATIAL_CRITERION (*C enum*), 489
PROJ_SPATIAL_CRITERION.PROJ_SPATIAL_CRITERION_PARTIAL_INTERSECTION (*C enumerator*), 489
PROJ_SPATIAL_CRITERION.PROJ_SPATIAL_CRITERION_STRICT_CONTAINMENT (*C enumerator*), 489
proj_string_destroy (*C function*), 534
PROJ_STRING_LIST (*C type*), 490
proj_string_list_destroy (*C function*), 518
proj_suggests_code_for (*C function*), 534
proj_todeg (*C function*), 509
proj_torad (*C function*), 509
proj_trans (*C function*), 499
proj_trans_array (*C function*), 501
proj_trans_bounds (*C function*), 501
proj_trans_bounds_3D (*C function*), 502
proj_trans_generic (*C function*), 499
proj_trans_get_last_used_operation (*C function*), 499
PROJ_UNIT_INFO (*C struct*), 492
PROJ_UNIT_INFO.auth_name (*C var*), 492
PROJ_UNIT_INFO.category (*C var*), 492
PROJ_UNIT_INFO.code (*C var*), 492
PROJ_UNIT_INFO.conv_factor (*C var*), 492
PROJ_UNIT_INFO.deprecated (*C var*), 492
PROJ_UNIT_INFO.name (*C var*), 492
PROJ_UNIT_INFO.proj_short_name (*C var*), 492
proj_unit_list_destroy (*C function*), 532
proj_uom_get_info_from_database (*C function*), 521
PROJ_VERSION_MAJOR (*C macro*), 469
PROJ_VERSION_MINOR (*C macro*), 469
PROJ_VERSION_NUMBER (*C macro*), 470
PROJ_VERSION_PATCH (*C macro*), 469
proj_xy_dist (*C function*), 507
proj_xyz_dist (*C function*), 508
projinfo, 93
projinfo command line option
    --3d, 98
    --accuracy, 97
    --allow-ellipsoidal-height-as-vertical-crs, 97
    --area, 95
    --authority, 98

--aux-db-path, 98
--bbox, 96
--boundcrs-to-wgs84, 98
--c-ify, 99
--crs-extent-use, 96
--dump-db-structure, 98
--grid-check, 96
--hide-ballpark, 97
--identify, 98
--list-crs, 98
--main-db-path, 98
--output-id, 98
--pivot-crs, 97
--remote-data, 99
--s_epoch, 99
--searchpaths, 99
--show-superseded, 97
--single-line, 99
--spatial-test, 96
--summary, 95
--t_epoch, 99
-k, 95
-o, 95
-q, 95
projsync, 104
projsync command line option
--all, 105
--area-of-use, 105
--bbox, 104
--dry-run, 105
--endpoint, 104
--exclude-world-coverage, 105
--file, 105
--list-files, 105
--local-geojson-file, 104
--no-version-filtering, 105
--source-id, 105
--spatial-test, 104
--system-directory, 104
--target-dir, 104
--user-writable-directory, 104
--verbose, 105
-q, 105
Pseudocylindrical Projection, **885**

## R

require_grid
command line option, 89
roundtrip
command line option, 88

## S

skip
command line option, 89

SSL_OPTIONS, 7

## T

TESTING_USE_NETWORK
    command line option, 53
TIFF_INCLUDE_DIR
    command line option, 53
TIFF_LIBRARY_RELEASE
    command line option, 53
tolerance
    command line option, 87

## U

USE_CCACHE
    command line option, 53
USE_ONLY_EMBEDDED_RESOURCE_FILES, 4
    command line option, 54

## X

XDG_DATA_HOME, 447