PROJ coordinate transformation software library

Release 5.2.0

PROJ contributors

September 2018

CONTENTS

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 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 let 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 it's way into PROJ as well. Today PROJ support 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 (2018). PROJ coordinate transformation software library. Open Source Geospatial Foundation. URL https://proj4.org/.

A BibTeX entry for LaTeX users is

```
@Manual{,
  title = {{PROJ} coordinate transformation software library},
  author = {{PROJ contributors}},
  organization = {Open Source Geospatial Foundation},
  year = {2018},
  url = {https://proj4.org/},
}
```

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 license is as follows:

> Copyright (c) 2000, Frank Warmerdam
>
> 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

CHAPTER TWO

NEWS

2.1 PROJ 5.2.0

September 15th 2018

2.1.1 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)

2.1.2 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)

2.2 PROJ 5.1.0

June 1st 2018

2.2.1 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)

2.2.2 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)

2.3 PROJ 5.0.1

March 1st 2018

2.3.1 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)

2.4 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.

2.4.1 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.

2.4.2 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.

2.4.3 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

Here you can download current and previous releases of PROJ. We only supply a distribution of the source code and various resource files archives. See *Installation* for information on how to get pre-built packages of PROJ.

3.1 Current Release

- **2018-09-15** proj-5.2.0.tar.gz (md5)
- **2018-09-15** proj-datumgrid-1.8.zip
- **2018-09-15** proj-datumgrid-europe-1.1.zip
- **2018-09-15** proj-datumgrid-north-america-1.1.zip
- **2018-03-01** proj-datumgrid-oceania-1.0.zip

3.2 Past Releases

- **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
- **2016-09-11** proj-datumgrid-1.6.zip
- **2018-03-01** proj-datumgrid-1.7.zip
- **2018-03-01** proj-datumgrid-europe-1.0.zip
- **2018-03-01** proj-datumgrid-north-america-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.

Conda

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

```
conda install -c conda-forge proj4
```

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 it is likely that PROJ is already installed.

1. Download either the 32 bit or 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 packacke directory”. The suggestions is fine in most cases. Press Next.
7. Select “Direct connection” and press Next.
8. Choose the download.osgeo.org 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-x86-64.exe -q -k -r -A -s http://download.osgeo.org/osgeo4w/ -a␣
↪x86_64 -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:

Debian

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

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

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 classical way of installing PROJ is via the source code distribution. The most recent version is available from the *download page*. You will need that and at least the standard *proj-datumgrid* package for a successful installation. The following guides show how to compile and install the software using the Autotools and CMake build systems.

4.2.1 Autotools

FSF's configuration procedure is used to ease installation of the PROJ system.

Note: The Autotools build system is only available on UNIX-like systems. Follow the CMake installation guide if you are not using a UNIX-like operating system.

The default destination path prefix for installed files is `/usr/local`. Results from the installation script will be placed into subdirectories `bin`, `include`, `lib`, `man/man1` and `man/man3`. If this default path prefix is proper, then execute:

```
./configure
```

If another path prefix is required, then execute:

```
./configure --prefix=/my/path
```

In either case, the directory of the prefix path must exist and be writable by the installer.

Before proceeding with the installation we need to add the datum grids. Unzip the contents of the *proj-datumgrid* package into `nad/`:

```
unzip proj-datumgrid-1.7.zip -d proj-5.0.1/nad/
```

The installation will automatically move the grid files to the correct location. Alternatively the grids can be installed manually in the directory pointed to by the `PROJ_LIB` environment variable. The default location is `/usr/local/share/proj`.

With the grid files in place we can now build and install PROJ:

```
make
make install
```

The install target will create, if necessary, all required sub-directories.

Tests are run with:

```
make check
```

The test suite requires that the proj-datumgrid package is installed in `PROJ_LIB`.

4.2.2 CMake

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-5.0.1
```

Create a build directory and step into it:

```
mkdir build
cd build
```

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

```
cmake ..
cmake --build .
```

On Windows, one may need to specify generator:

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

Tests are run with:

```
ctest
```

The test suite requires that the proj-datumgrid package is installed in *PROJ_LIB*.

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
+axis	Axis orientation
+b	Semiminor radius of the ellipsoid axis
+ellps	Ellipsoid name (see `proj -le`)
+k	Scaling factor (deprecated)
+k_0	Scaling factor
+lat_0	Latitude of origin
+lon_0	Central meridian
+lon_wrap	Center longitude to use for wrapping (see below)
+no_defs	Don't use the /usr/share/proj/proj_def.dat defaults file
+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	meters, US survey feet, etc.
+vunits	vertical units.
+x_0	False easting
+y_0	False northing

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

5.2.1 Units

Horizontal units can be specified using the `+units` keyword with a symbolic name for a unit (ie. `us-ft`). Alternatively the translation to meters can be specified with the `+to_meter` keyword (ie. 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 behaviour, most notably the latlong pseudo-projection that returns degrees.

Vertical (Z) units can be specified using the `+vunits` keyword with a symbolic name for a unit (ie. `us-ft`). Alternatively the translation to meters can be specified with the `+vto_meter` keyword (ie. 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` do 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. The returned coordinates are scaled by the value assigned with the `+k_0` parameter.

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* where it would be desirable for X values past -20000000 (roughly) to continue past -180 instead of wrapping to +180.

The `+lon_wrap` option can be used to provide an alternative means of doing longitude wrapping within `pj_transform()`. The argument to this option is a center longitude. So `+lon_wrap=180` means wrap longitudes in the range 0 to 360. Note that `+over` does **not** disable `+lon_wrap`.

5.2.4 Prime Meridian

A prime meridian may be declared indicating the 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 a symbolic name, or the longitude of the alternative prime meridian relative to greenwich.

Currently prime meridian declarations are only utilized by the `pj_transform()` API call, not the `pj_inv()` and `pj_fwd()` calls. Consequently the user utility `cs2cs` does honour prime meridians but the `proj` user utility ignores them.

The following predeclared prime meridian names are supported. These can be listed using with `cs2cs -lm`.

Meridian	Longitude
greenwich	0dE
lisbon	9d07’54.862”W
paris	2d20’14.025”E
bogota	74d04’51.3”E
madrid	3d41’16.48”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

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

```
cs2cs +proj=latlong +datum=WGS84 +to +proj=latlong +datum=WGS84 +pm=madrid
0 0
3d41'16.48"E    0dN 0.000
```

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.

5.3 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 *cs2cs* 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). 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
     rx=-0.48488 ry=-0.02436 rz=-0.41321 s=-0.540645
```

(continues on next page)

(continued from previous page)

```
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 cs2cs paradigm

Parameter	Description
+datum	Datum name (see `proj -ld`)
+geoidgrids	Filename of GTX grid file to use for vertical datum transforms
+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

The *cs2cs* framework delivers a subset of the geodetic transformations available with the *pipeline* framework. Coordinate transformations done in this framework are 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

In contrast to the *transformation pipeline* framework, transformations with the *cs2cs* framework are 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
20 35
20d0'5.467"E    35d0'9.575"N 8.570
```

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 3.223
```

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 (the binary format produced by the PROJ `nad2bin` program), 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
```

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
```

The null Grid

A special `null` grid shift file is shift with releases after 4.4.6 (not inclusive). 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 *Transformation grids*.

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
- There are additional grids for shifting between NAD83 and various HPGN versions of the NAD83 datum. Use of these haven't been tried recently so you may encounter problems. The FL.lla, WO.lla, MD.lla, TN.lla and WI.lla are examples of high precision grid shifts. Take care!
- 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
- The *cs2cs* framework always assumes that grids contain a shift **to** NAD83 (essentially WGS84). Other types of grids can be used with the *pipeline* framework.

5.4 Environment variables

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

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_LIB
The location of PROJ *resource files*. It is only possible to specify a single library in *PROJ_LIB*; e.g. it does not behave like PATH. PROJ is hardcoded to look for resource files in other locations as well, amongst those are the users home directory, `/usr/share/proj` and the current folder.

PROJ_DEBUG
Set the debug level of PROJ. The default debug level is zero, which results in no debug output when using PROJ. A number from 1-3, whit 3 being the most verbose setting.

5.5 Known differences between versions

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

5.5.1 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.

5.5.2 Version 5.0.0

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
-70 0
-20037508.34    0.00
290 0
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
-70 0
-20037508.34    0.00
290 0
-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 behaviour.

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 basic datum translation. The `geod` program provides the ability to do geodesic (great circle) computations. `gie` is the program used for regression tests in PROJ.

6.1 proj

6.1.1 Synopsis

> **proj** [**-beEfiIlmorsStTvVwW**] [args]] [+*args*] file[s]
>
> **invproj** [**-beEfiIlmorsStTvVwW**] [args]] [+*args*] file[s]

6.1.2 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>`

New in version 5.2.0: Specify the number of decimals 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.

-ld
List of datums that can be selected with the *+datum* 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. Should not be used with the `-T` option.

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

-T `<ulow,uhi,vlow,vhi,res[,umax,vmax]>`
This option creates a set of bivariate Chebyshev polynomial coefficients that approximate the selected cartographic projection on stdout. The values *low* and *hi* denote the range of the input where the *u* or *v* prefixes apply to respective longitude-x or latitude-y depending upon whether a forward or inverse projection is selected. The integer *res* is a number specifying the power of 10 precision of the approximation. For example, a *res* of -3 specifies an approximation with an accuracy better than 0.001. Optional *umax*, and *vmax* specify maximum degree of the polynomials (default: 15).

The *+args* 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_LIB` establishes the default directory for a file reference without an absolute path. This is also used for supporting files like datum shift files.

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 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.

6.1.3 Example

The following script

```
proj +proj=utm +lon_0=112w +ellps=clrk66 -r <<EOF
45d15'33.1" 111.5W
45d15.551666667N -111d30
+45.25919444444 111d30'000w
EOF
```

will perform UTM forward projection with a standard UTM central meridian nearest longitude 112W. 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:

```
460769.27     5011648.45
```

6.2 cct

6.2.1 Synopsis

cct [**-cIostvz** [args]] +*opts[=arg]* file[s]

6.2.2 Description

`cct` 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.

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>`

New in version 5.2.0: Specify the number of decimals 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>`
New 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 +*args* 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).

6.2.3 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 +proj=utm +ellps=GRS80 +zone=32 +step +step +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 forward 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
```

6.2.4 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).

6.3 cs2cs

6.3.1 Synopsis

cs2cs [**-eEfIlrstvwW** [args]] [*+opts[=arg]*] [+to [*+opts[=arg]*]] file[s]

6.3.2 Description

`cs2cs` performs transformation between the source and destination cartographic coordinate system 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.

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>`

New in version 5.2.0: Specify the number of decimals 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.

`-lu`
: List of all distance units that can be selected with the *+units* parameter.

`-ld`
: List of datums that can be selected with the *+datum* 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.

-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.

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

The **cs2cs** program requires two coordinate system 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 coordinate system. If there is no second coordinate system defined, a geographic coordinate system based on the datum and ellipsoid of the source coordinate system is assumed. Note that the source and destination coordinate system can both be projections, both be geographic, or one of each and may have the same or different datums.

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_LIB* establishes the default directory for a file reference without an absolute path. This is also used for supporting files like datum shift files.

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.

6.3.3 Example

The following script

```
cs2cs +proj=latlong +datum=NAD83 +to +proj=utm +zone=10 +datum=NAD27 -r <<EOF
45d15'33.1" 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:

```
1402285.98  5076292.42 -0.00
```

6.4 geod

6.4.1 Synopsis

geod +*ellps*=<*ellipse*> [**-afFIlptwW** [args]] [+*args*] file[s]

invgeod +*ellps*=<*ellipse*> [**-afFIlptwW** [args]] [+*args*] file[s]

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.

-f `<format>`
Where *format* is a printf format string to control the output form of the geographic coordinate values. The default mode is DMS for geographic coordinates and `"%.3f"` for distance.

-F `<format>`
Where *format* is a printf format string to control the output form of the distance value (`-F`). The default mode is DMS for geographic coordinates and `"%.3f"` for distance.

-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 +*args* 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 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=lon* parameters and either a terminus point *+lat_2=lat* and *+lon_2=lon* 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 <<EOF -I +units=us-mi
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 <<EOF +units=us-mi
42d15'N 71d07'W -66d31'50.141" 2587.504
EOF
```

which gives:

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

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 -l` 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
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 -l` 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 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.

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 Albers Equal Area

Classification	Conic
Available forms	Forward and inverse, spherical and elliptical projection
Defined area	Global
Alias	aea
Domain	2D
Input type	Geodetic coordinates
Output type	Projected coordinates

Options

Note: All options are optional for the Albers Equal Area projection.

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

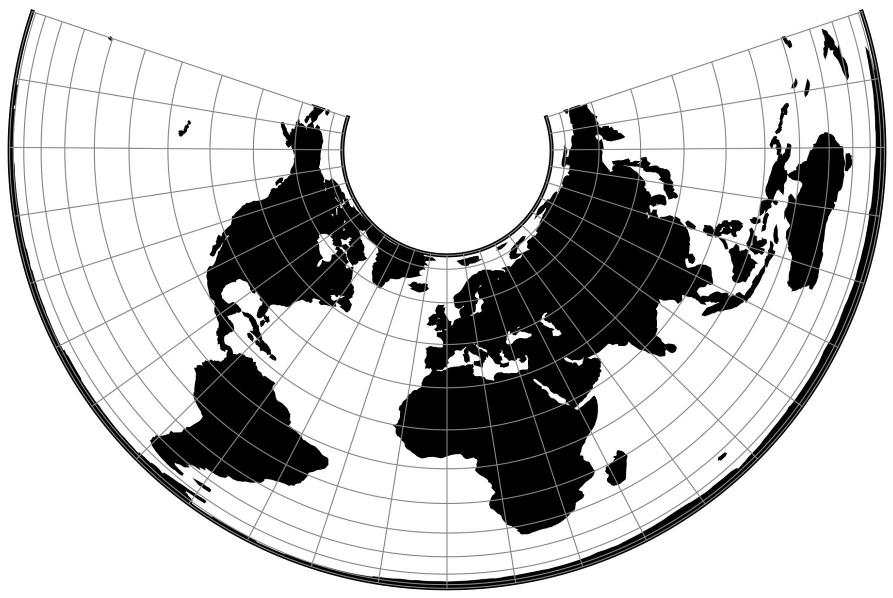


Fig. 1: proj-string: `+proj=aea`

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.2 Azimuthal Equidistant

Classification	Azimuthal
Available forms	Forward and inverse, spherical and elliptical projection
Alias	aeqd
Domain	2D
Input type	Geodetic coordinates
Output type	Projected coordinates

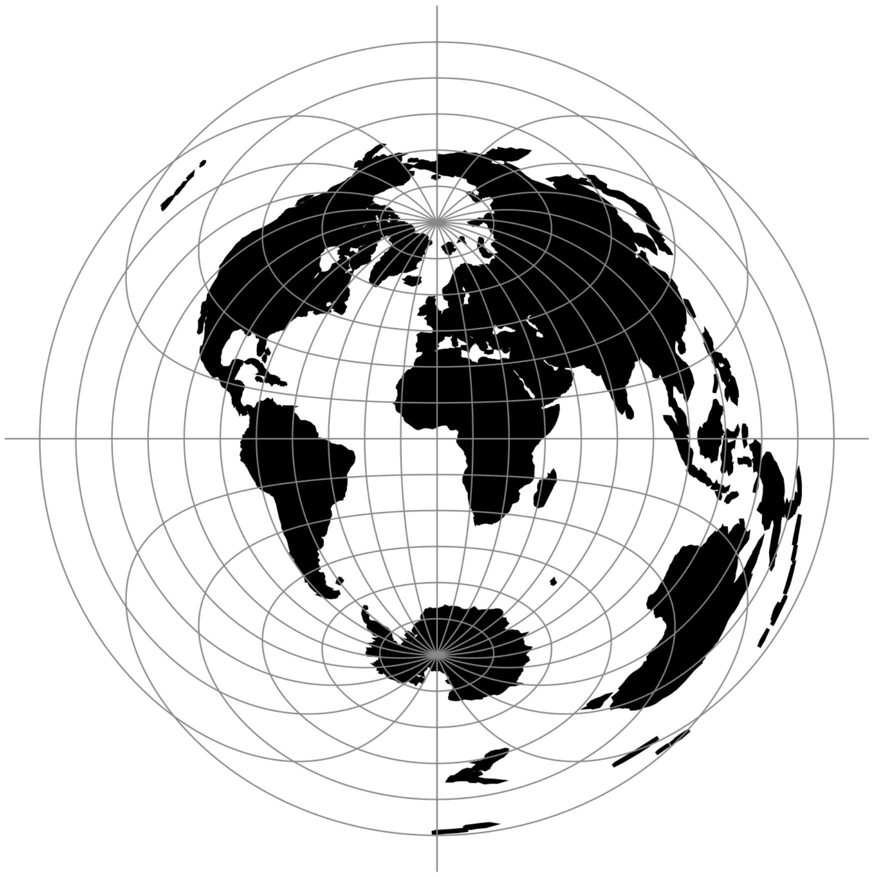


Fig. 2: proj-string: `+proj=aeqd`

Options

Note: All options are optional for the Azimuthal Equidistant projection.

+guam
Use Guam elliptical formulas. Only accurate near the Island of Guam ($\lambda \approx 144.5°$, $\phi \approx 13.5°$)

+k_0=<value>
Scale factor. Determines scale factor used in the projection.

Defaults to 1.0.

+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.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

7.1.3 Airy

The Airy projection is an azimuthal minimum error projection for the region within the small or great circle defined by an angular distance, ϕ_b, from the tangency point of the plane (λ_0, ϕ_0).

Classification	Azimuthal
Available forms	Forward spherical projection
Defined area	Global
Alias	airy
Domain	2D
Input type	Geodetic coordinates
Output type	Projected coordinates

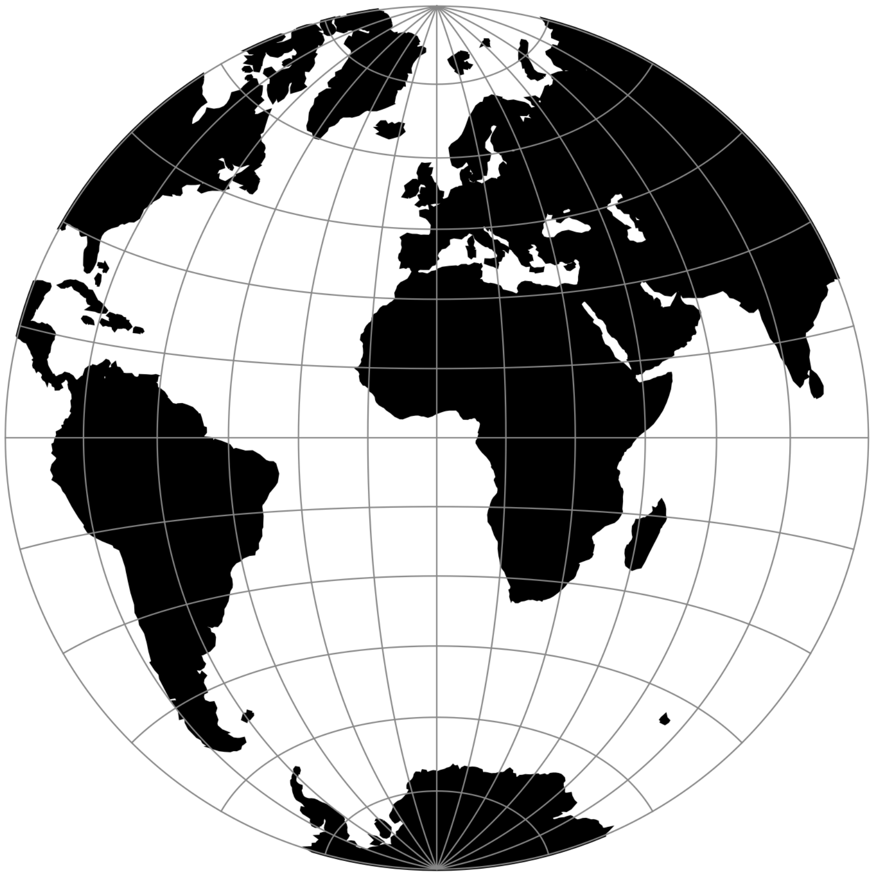


Fig. 3: proj-string: `+proj=airy`

Options

+lat_b
Angular distance from tangency point of the plane (λ_0, ϕ_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 projection center.

Defaults to 0.0.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

7.1.4 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

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

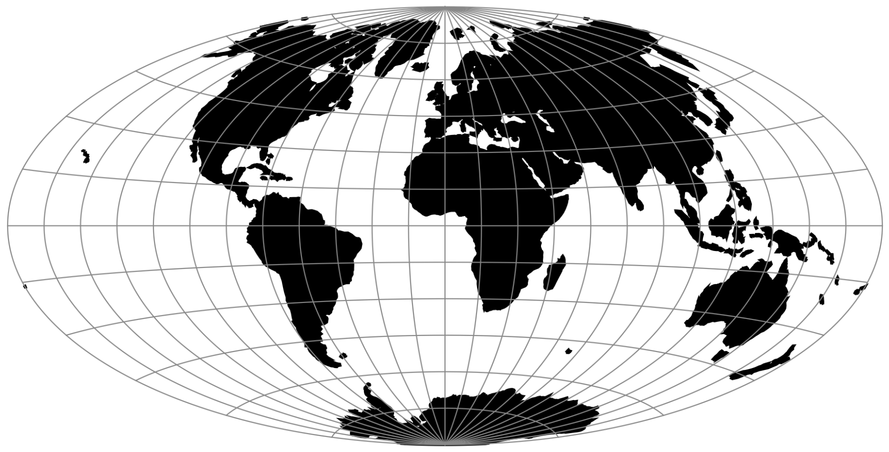


Fig. 4: proj-string: `+proj=aitoff`

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.5 Modified Stererographics of Alaska

Classification	Modified azimuthal
Available forms	Forward and inverse, spherical and elliptical projection
Defined area	Alaska
Alsk	alsk
Domain	2D
Input type	Geodetic coordinates
Output type	Projected coordinates

Options

Note: All options are optional for the projection.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

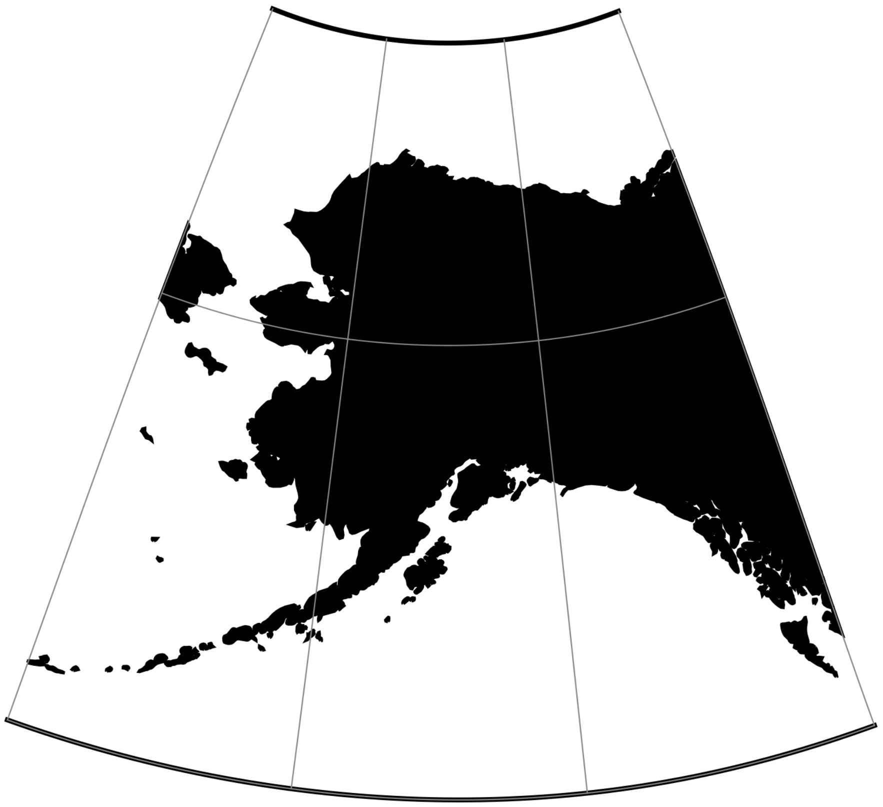


Fig. 5: proj-string: `+proj=alsk`

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

7.1.6 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

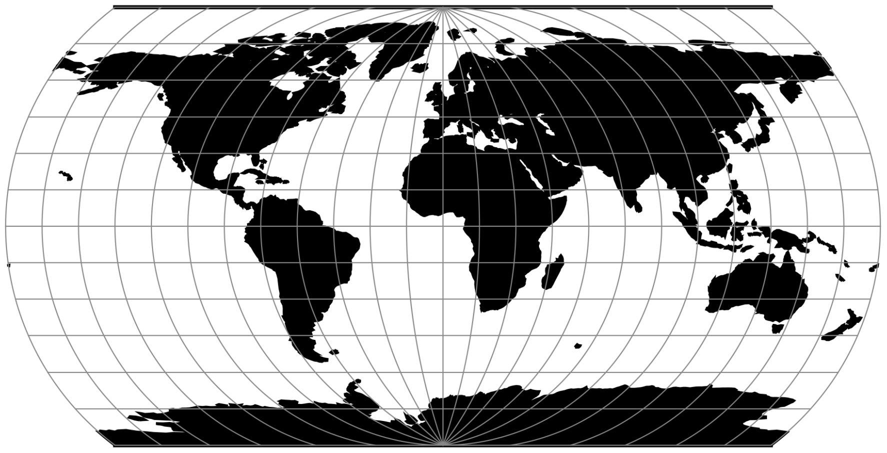


Fig. 6: proj-string: `+proj=apian`

Options

Note: All options are optional for the Apian Globular projection.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.7 August Epicycloidal

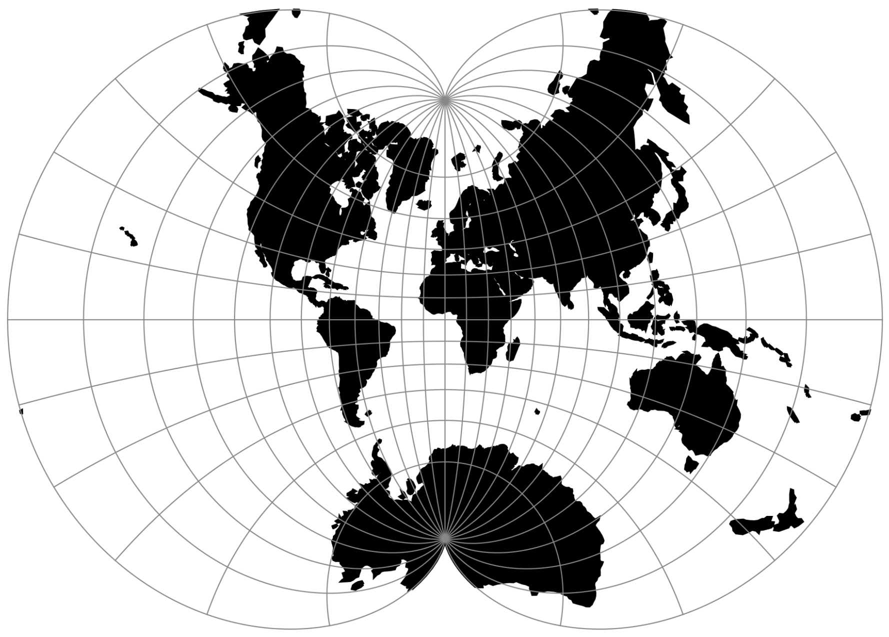


Fig. 7: proj-string: `+proj=august`

Parameters

Note: All options are optional for the August Epicycloidal projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.8 Bacon Globular

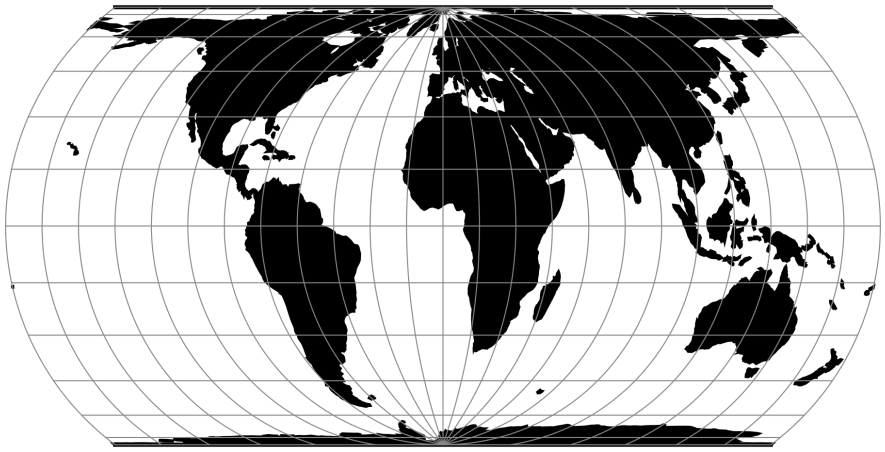


Fig. 8: proj-string: `+proj=bacon`

Parameters

Note: All parameters are optional for the Bacon Globular projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.9 Bipolar conic of western hemisphere

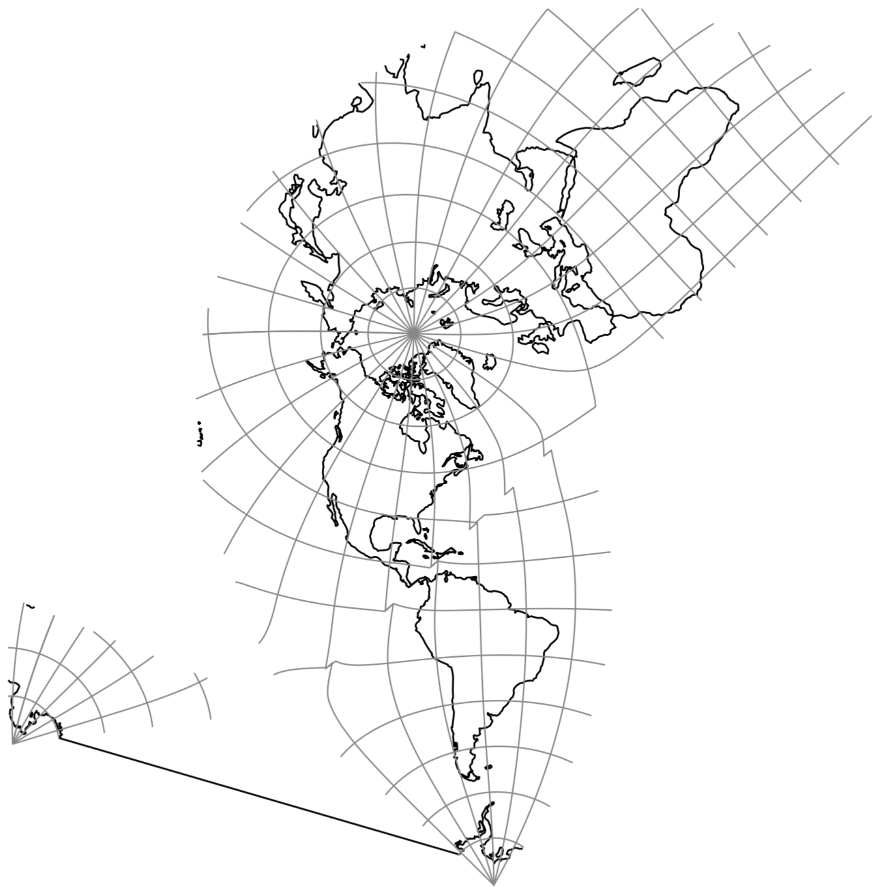


Fig. 9: proj-string: `+proj=bipc +ns`

Parameters

Note: All options are optional for the Bipolar Conic projection.

+ns
    Return non-skewed cartesian coordinates.

+lon_0=<value>
    Longitude of projection center.

    Defaults to 0.0.

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.10 Boggs Eumorphic

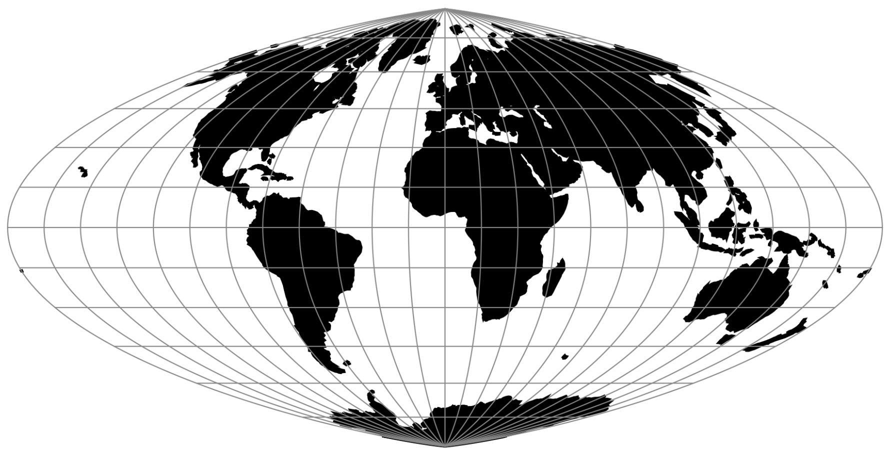


Fig. 10: proj-string: `+proj=boggs`

Parameters

Note: All options are optional for the Boggs Eumorphic projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.11 Bonne (Werner lat_1=90)

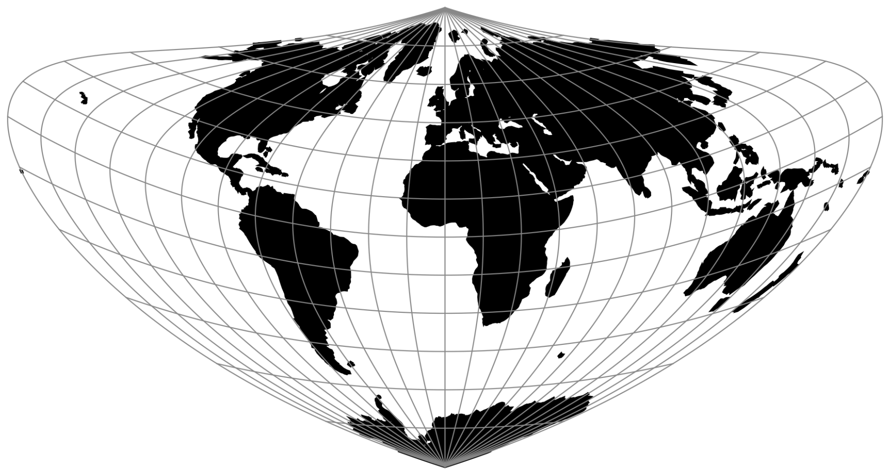


Fig. 11: proj-string: `+proj=bonne +lat_1=10`

Parameters

Required

+lat_1=<value>
    First standard parallel.

    Defaults to 0.0.

Optional

+lon_0=<value>
    Longitude of projection center.

    Defaults to 0.0.

+ellps=<value>
    See `proj -le` for a list of available ellipsoids.

    Defaults to "WGS84".

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
    False easting.

    Defaults to 0.0.

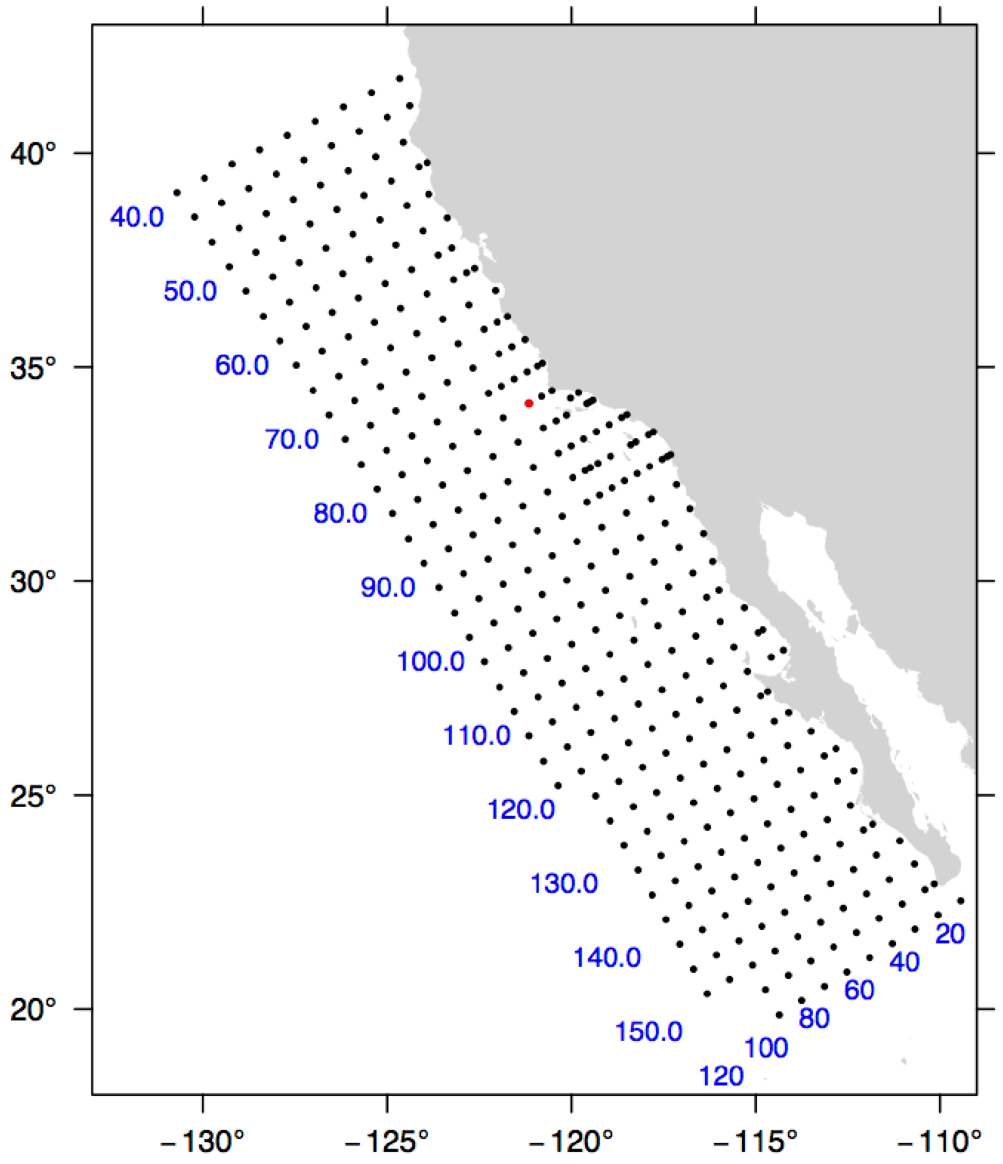
+y_0=<value>
    False northing.

    Defaults to 0.0.

7.1.12 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 elliptical projection
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.

Usage

A typical forward CalCOFI projection would be from lon/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 lon/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
```

Options

Note: All options are optional for the CalCOFI projection.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

Mathematical definition

The algorithm used to make conversions is described in [EberHewitt1979] with a few corrections reported in [Weber-Moore2013].

Further reading

1. General information about the CalCOFI program
2. The Investigaciones Mexicanas de la Corriente de California

7.1.13 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 Elliptical
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

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 +no_defs
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 +datum=potsdam +units=m +no_defs
31343.05    7932.76
```

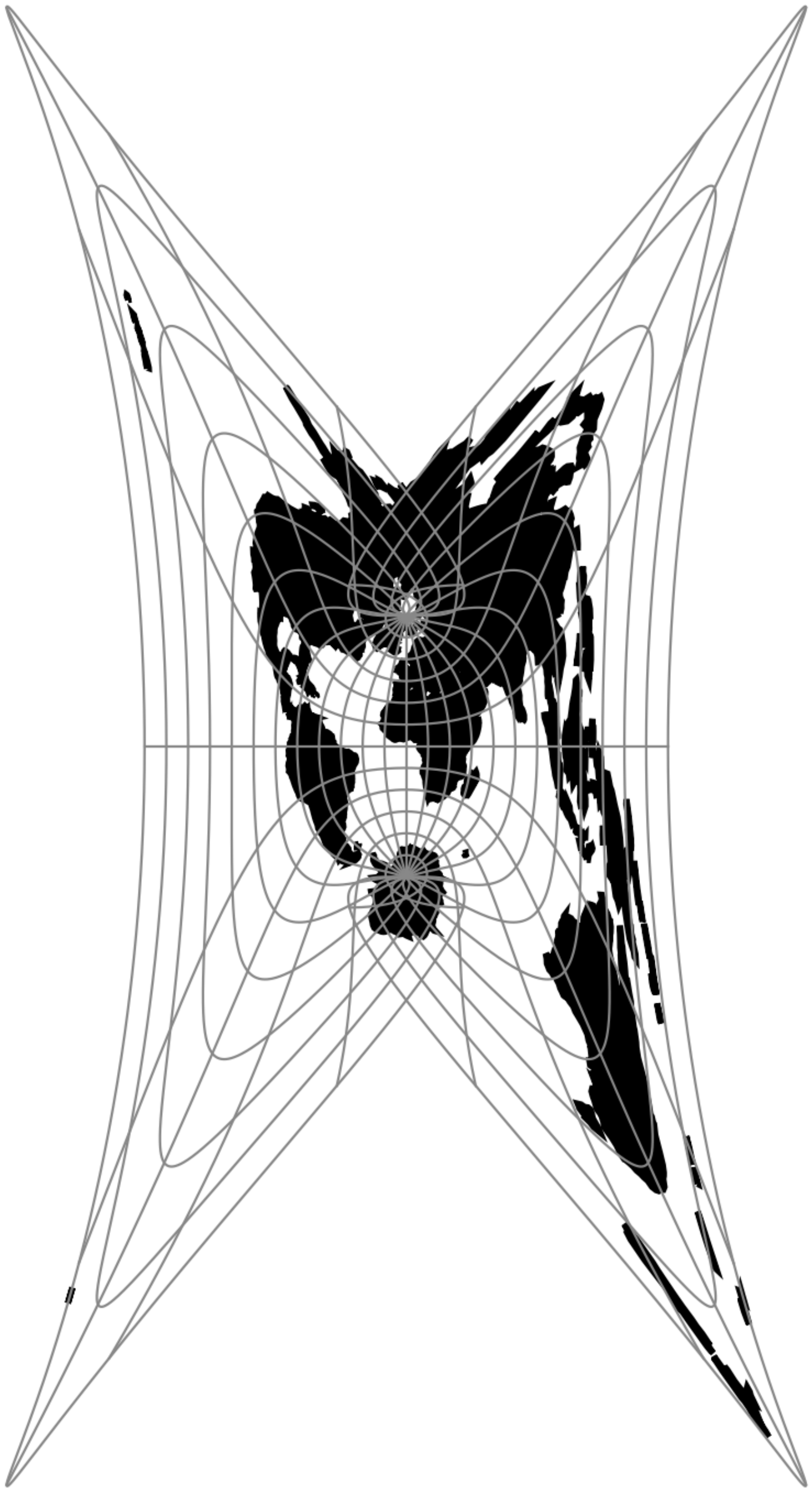


Fig. 12: proj-string: `+proj=cass`

Options

Note: All options are optional for the Cassini projection.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

+ellps=<value>
See *proj -le* for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

Mathematical definition

The formulas describing the Cassini projection are taken from [Snyder1987].

ϕ_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

$$\phi = \arcsin(\sin(y + \phi_0)\cos(x))$$

$$\lambda = \arctan 2(\tan(x), \cos(y + \phi_0))$$

Elliptical 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 ϕ' and

$$R = (1 - e^2)(1 - e^2 sin^2\phi')^{-3/2}$$

$$D = x/N$$

$$\phi = \phi' - \tan\phi' \frac{N}{R}(\frac{D^2}{2} - (1 + 3T)\frac{D^4}{24})$$

$$\lambda = \frac{(D - T\frac{D^3}{3} + (1 + 3T)T\frac{D^5}{15})}{\cos\phi'}$$

Further reading

1. Wikipedia
2. EPSG, POSC literature pertaining to Coordinate Conversions and Transformations including Formulas

7.1.14 Central Cylindrical

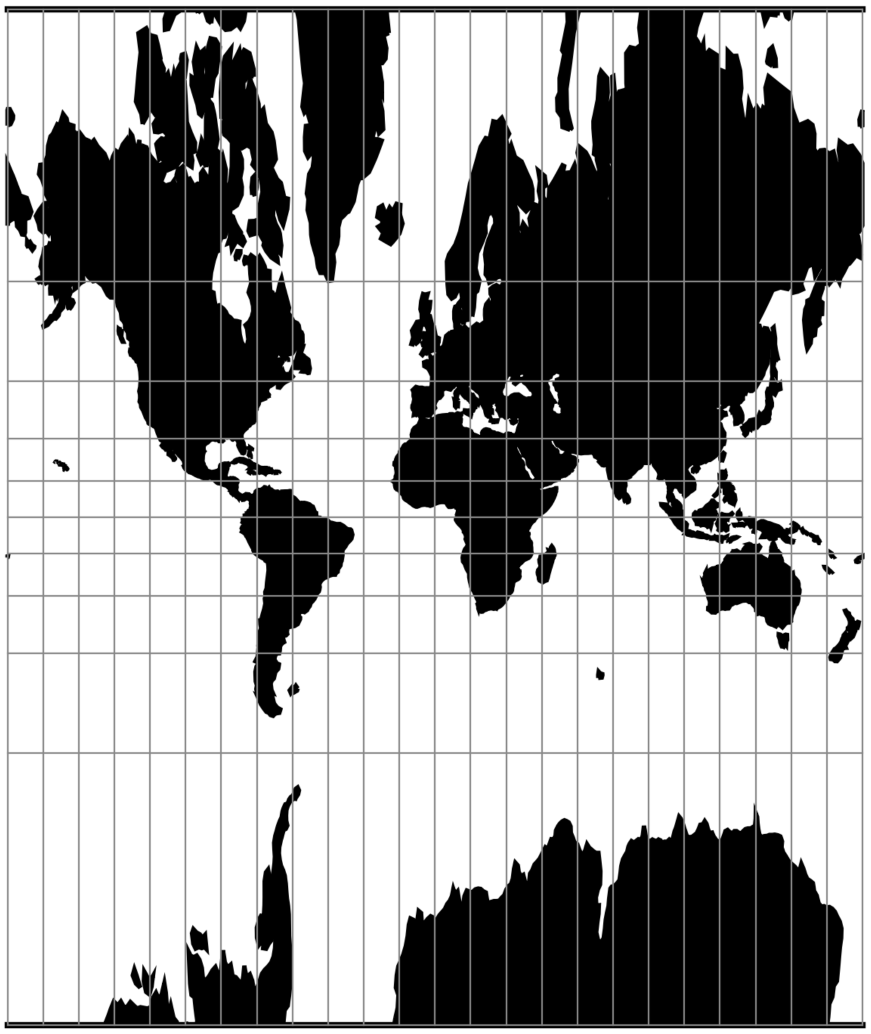


Fig. 13: proj-string: `+proj=cc`

Parameters

Note: All parameters are optional for the Central Cylindricla projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.15 Central Conic

New 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

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].

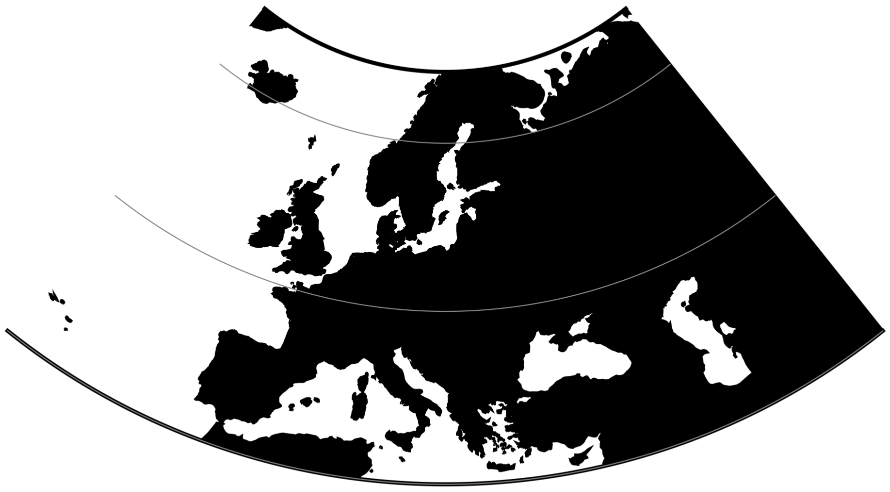


Fig. 14: proj-string: `+proj=ccon +lat_1=52 +lon_0=19`

Parameters

Required

+lat_1=<value>
: Standard parallel of projection.

Optional

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

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

$$y = \cot\phi_0 - y$$

$$\phi = \phi_0 - \tan^{-1}(\sqrt{x^2+y^2} - \cot\phi_0)$$

$$\lambda = \frac{\tan^{-1}\sqrt{x^2+y^2}}{\sin\phi_0}$$

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 +datum=WGS84 +no_defs
0 0
0 700000
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 +datum=WGS84 +no_defs +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
```

7.1.16 Equal Area Cylindrical

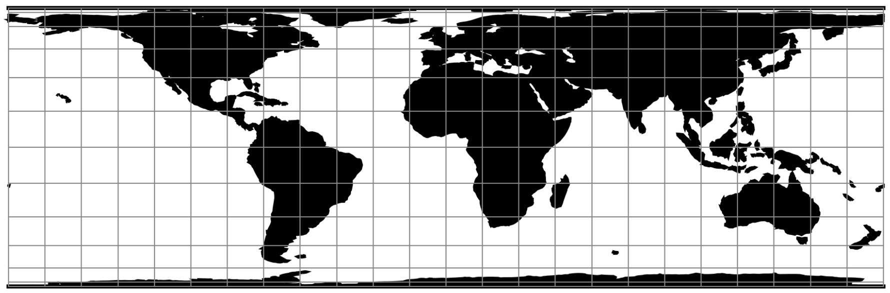


Fig. 15: proj-string: `+proj=cea`

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.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+k_0=<value>
: Scale factor. Determines scale factor used in the projection.

  Defaults to 1.0.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.17 Chamberlin Trimetric

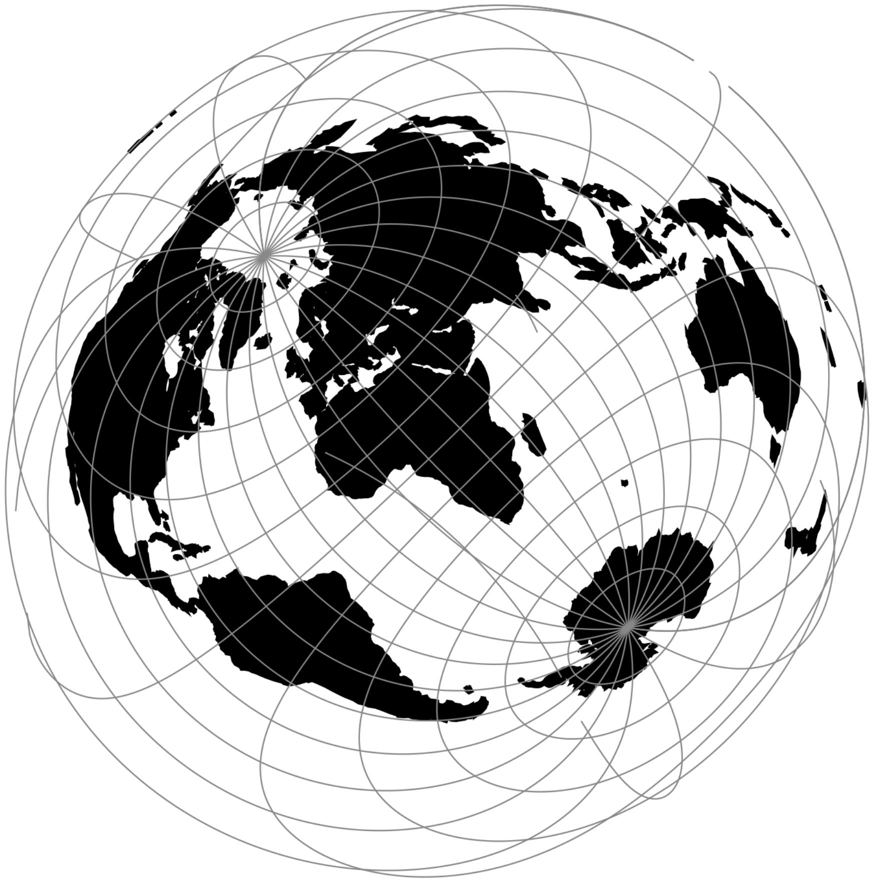


Fig. 16: proj-string: `+proj=chamb +lat_1=10 +lon_1=30 +lon_2=40`

Parameters

Required

+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>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.18 Collignon

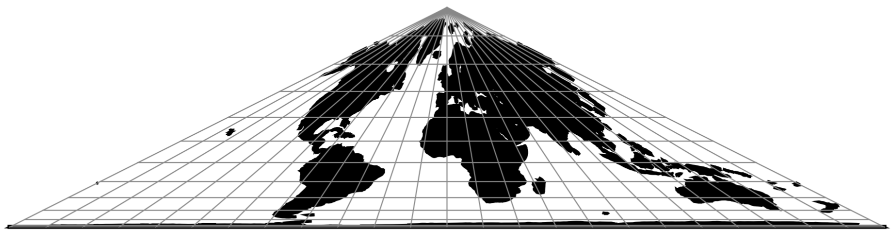


Fig. 17: proj-string: `+proj=collg`

Parameters

Note: All parameters are optional for the Collignon projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.19 Craster Parabolic (Putnins P4)

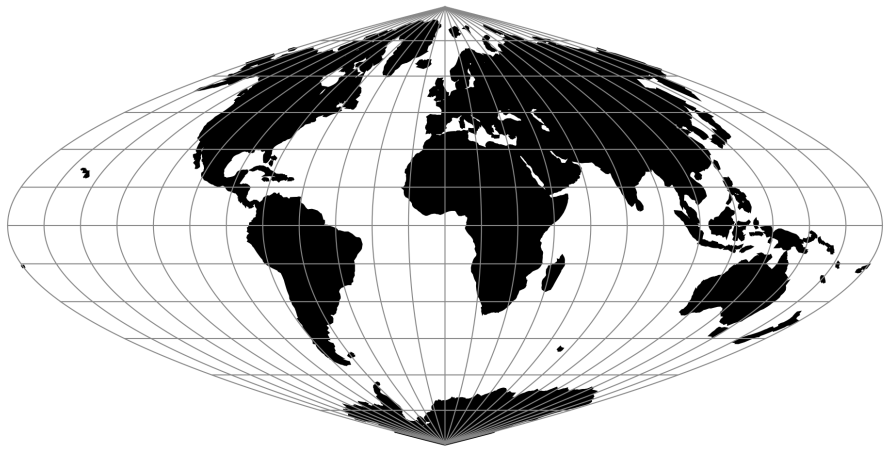


Fig. 18: proj-string: `+proj=crast`

Parameters

Note: All parameters are optional for the Craster Parabolic projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.20 Denoyer Semi-Elliptical

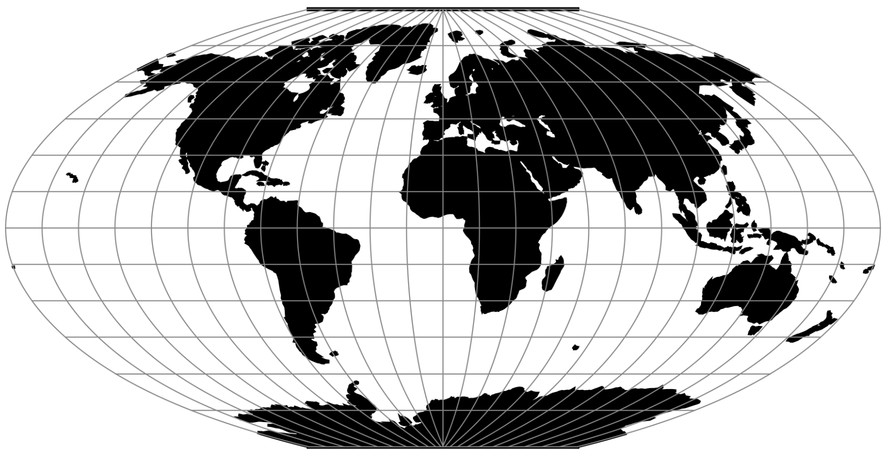


Fig. 19: proj-string: `+proj=denoy`

Parameters

Note: All parameters are optional for the Denoyer Semi-Elliptical projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.21 Eckert I

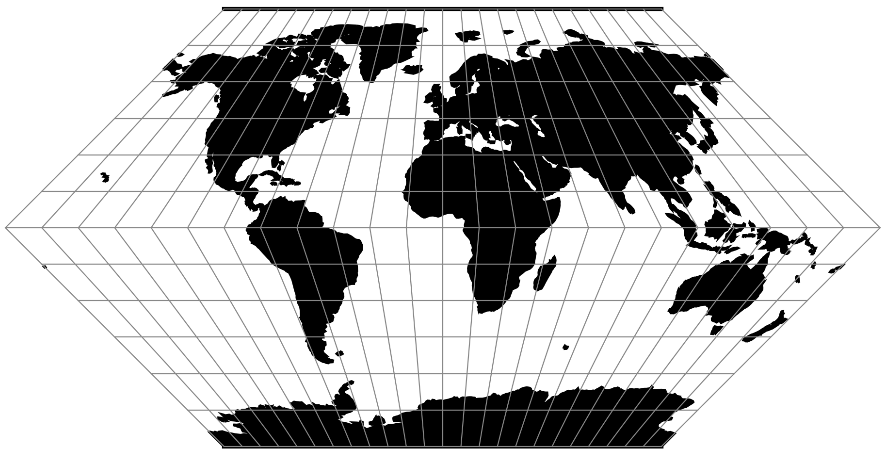


Fig. 20: proj-string: `+proj=eck1`

$$x = 2\sqrt{2/3\pi}\lambda(1 - |\phi|/\pi)$$
$$y = 2\sqrt{2/3\pi}\phi$$

Parameters

Note: All parameters are optional for the Eckert I projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.22 Eckert II

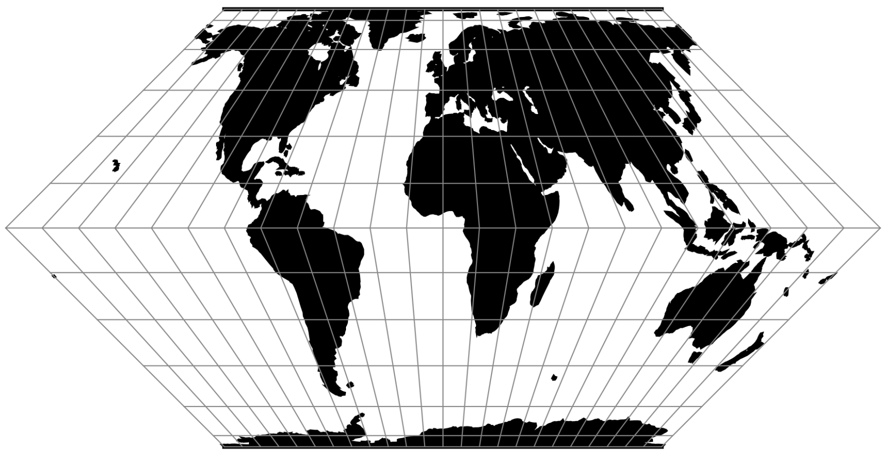


Fig. 21: proj-string: `+proj=eck2`

Parameters

Note: All parameters are optional for the Eckert II projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.23 Eckert III

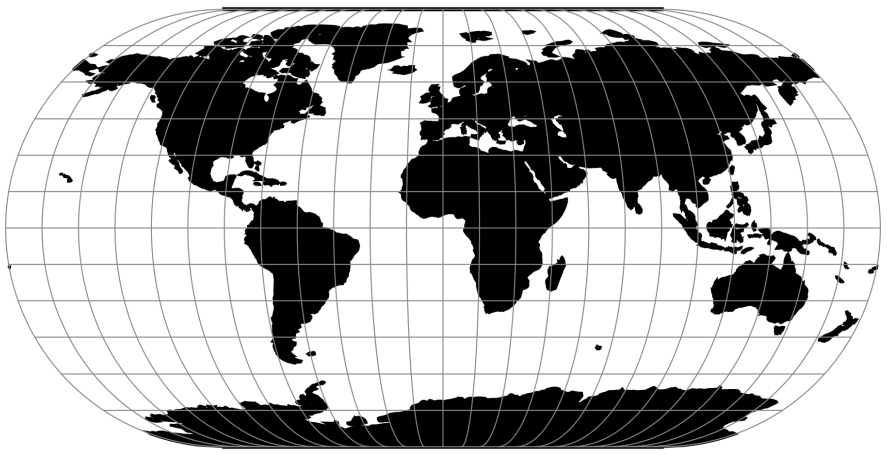


Fig. 22: proj-string: `+proj=eck3`

Parameters

Note: All parameters are optional for the Eckert III projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.24 Eckert IV

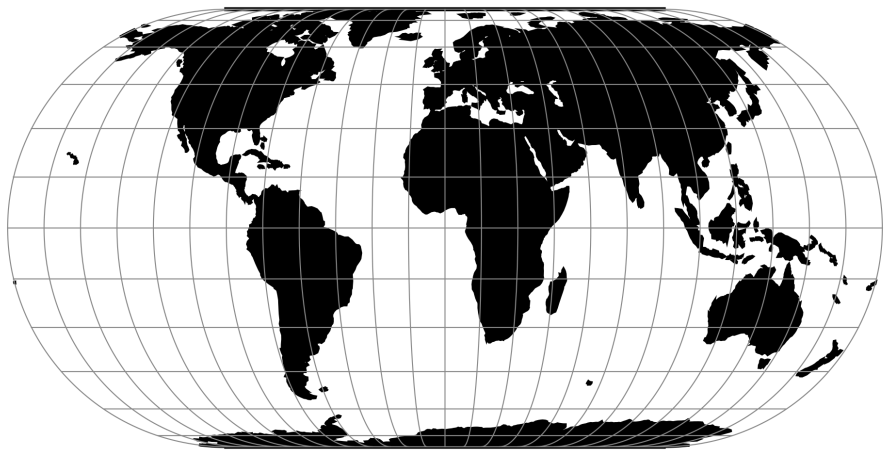


Fig. 23: proj-string: `+proj=eck4`

$$x = \lambda(1 + cos\phi)/\sqrt{2+\pi}$$

$$y = 2\phi/\sqrt{2+\pi}$$

Parameters

Note: All parameters are optional for the Eckert IV projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.25 Eckert V

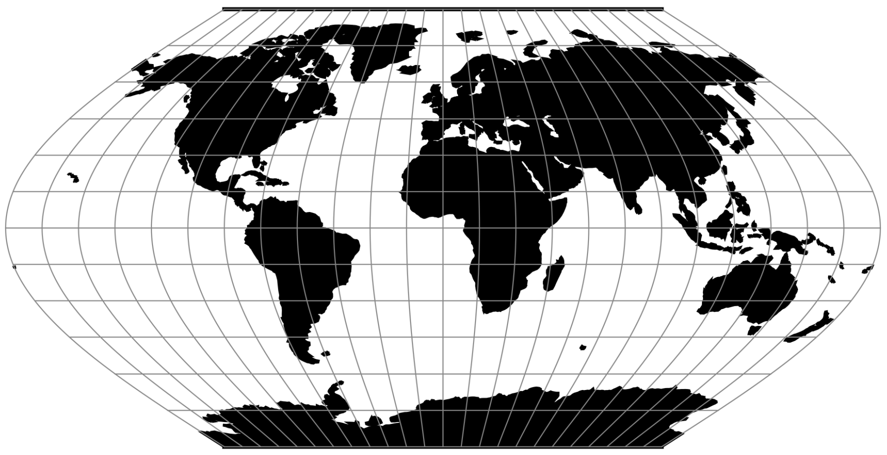


Fig. 24: proj-string: `+proj=eck5`

Parameters

Note: All parameters are optional for the Eckert V projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.26 Eckert VI

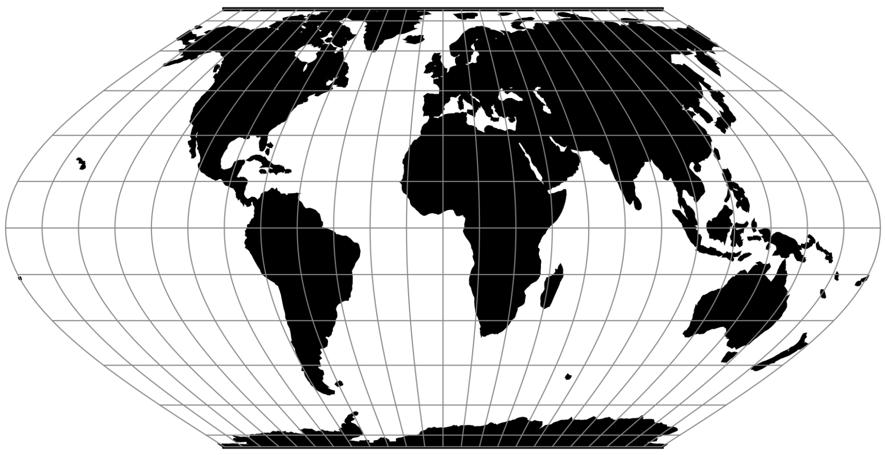


Fig. 25: proj-string: `+proj=eck6`

Parameters

Note: All parameters are optional for the Eckert VI projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.27 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

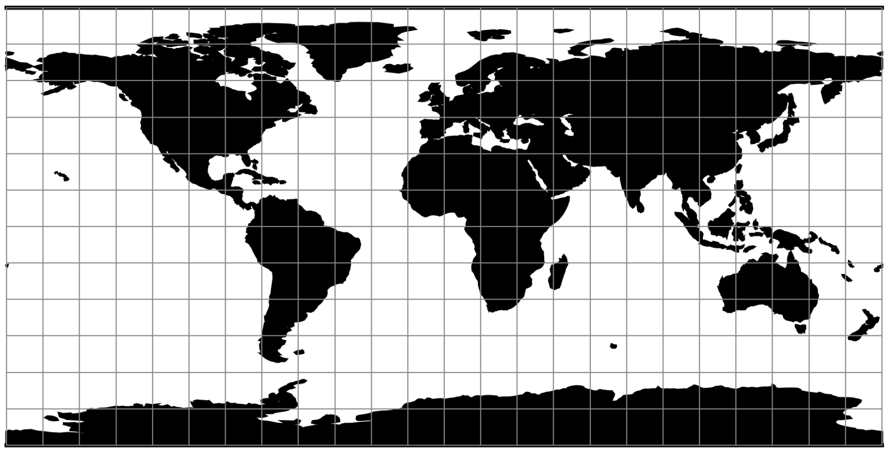


Fig. 26: proj-string: `+proj=eqc`

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.

Projection Name	(lat ts=) ϕ_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 +lat_ts=0 +lat_0=0 +lon_0=0 +x_0=0 +y_0=0 +ellps=WGS84␣
↪+datum=WGS84 +units=m +no_defs
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
```

Parameters

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+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.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

Mathematical definition

The formulas describing the Equidistant Cylindrical projection are all taken from [Snyder1987].

ϕ_{ts} is the latitude of true scale, that mean the standard parallels where the scale of the projection is true. It can be set with `+lat_ts`.

ϕ_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

$$\lambda = x / cos \phi_{ts}$$

$$\phi = y + \phi_0$$

Further reading

1. Wikipedia
2. Wolfram Mathworld

7.1.28 Equidistant Conic

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

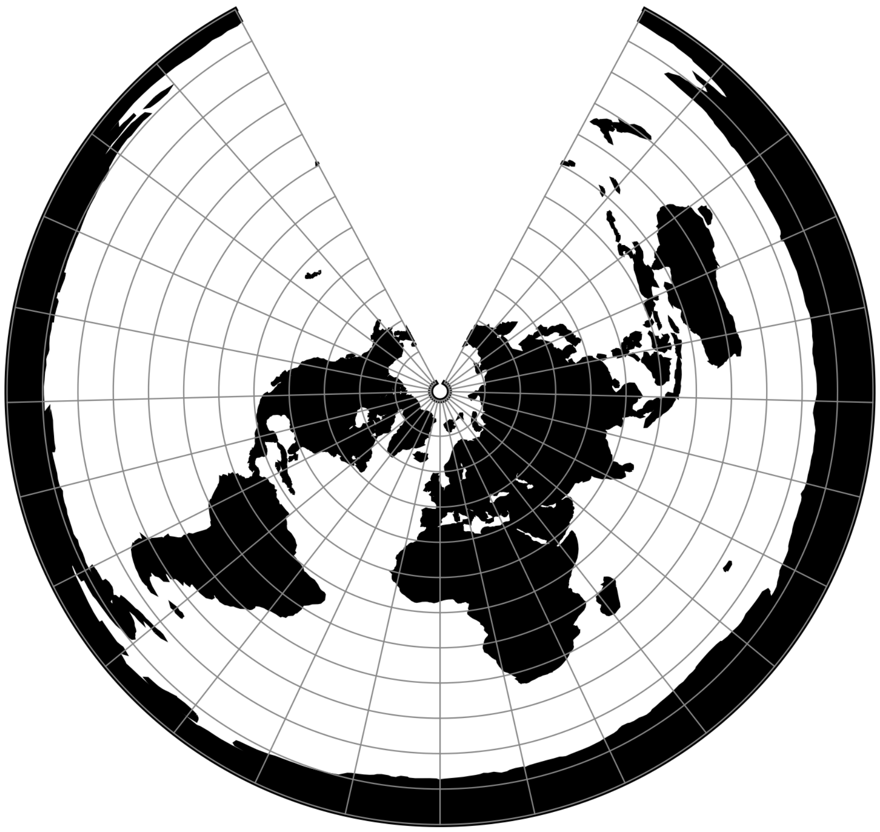


Fig. 27: proj-string: `+proj=eqdc +lat_1=55 +lat_2=60`

Optional

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.29 Equal Earth

New 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

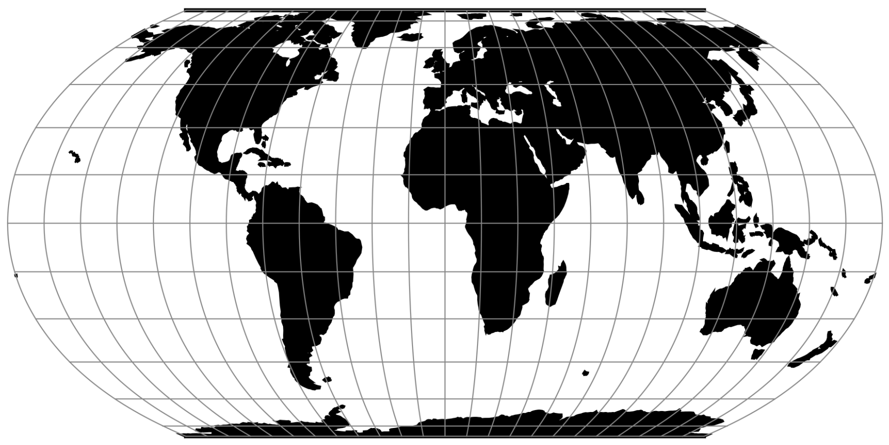


Fig. 28: proj-string: `+proj=eqearth`

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].

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 | src/proj +proj=eqearth +R=1
1.55        0.89
```

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

Further reading

1. Bojan Savric, Tom Patterson & Bernhard Jenny (2018). The Equal Earth map projection, International Journal of Geographical Information Science

7.1.30 Euler

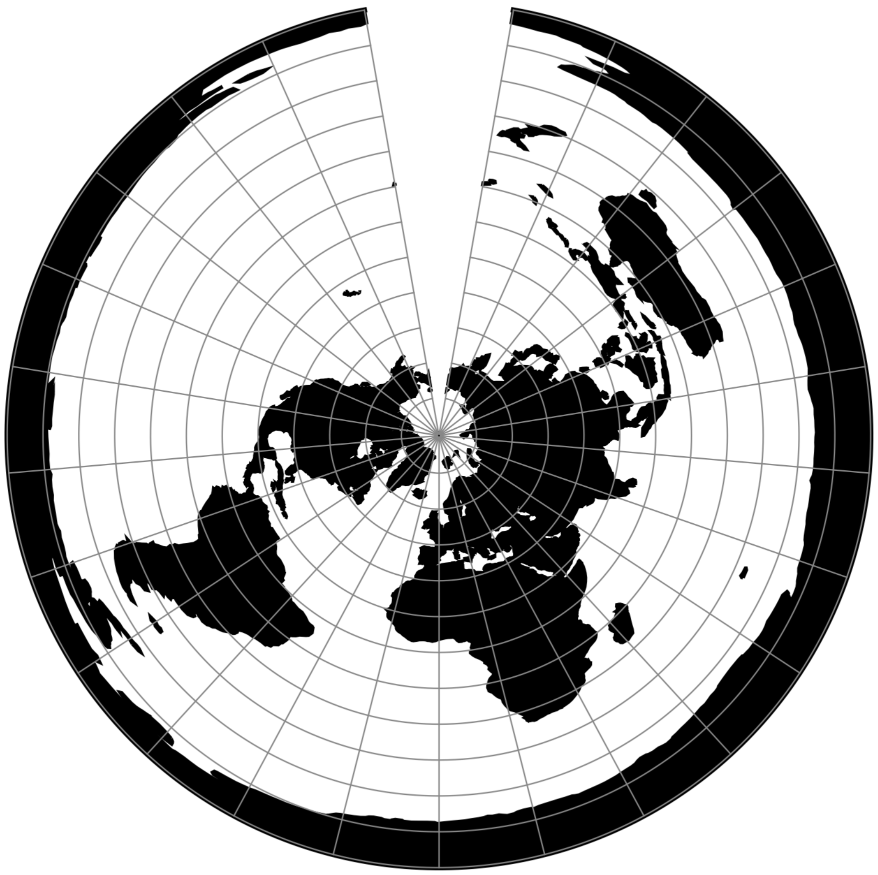


Fig. 29: proj-string: `+proj=euler +lat_1=67 +lat_2=75`

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

Optional

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.31 Extended Transverse Mercator

Parameters

Note: All parameters for the projection are optional.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+lat_0``=<value>`
Latitude of projection center.

Defaults to 0.0.

`+ellps``=<value>`
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+k_0``=<value>`
Scale factor. Determines scale factor used in the projection.

Defaults to 1.0.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

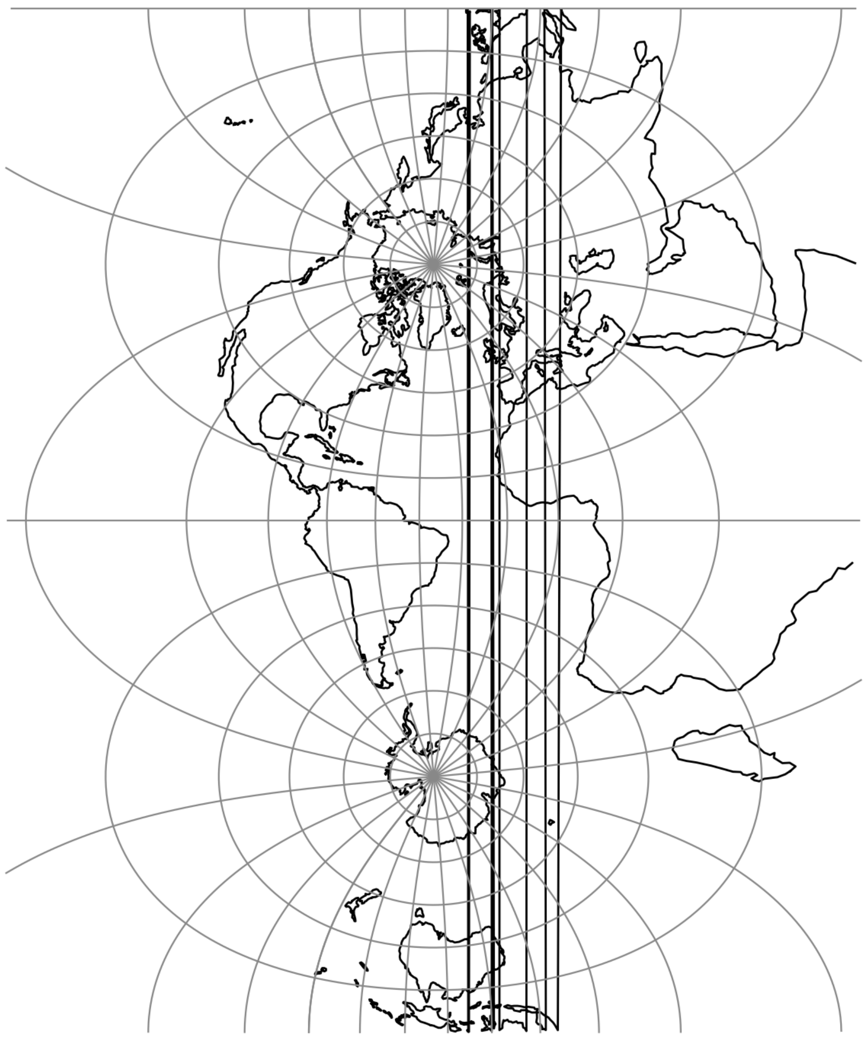


Fig. 30: proj-string: `+proj=etmerc +lon_0=-40`

7.1.32 Fahey

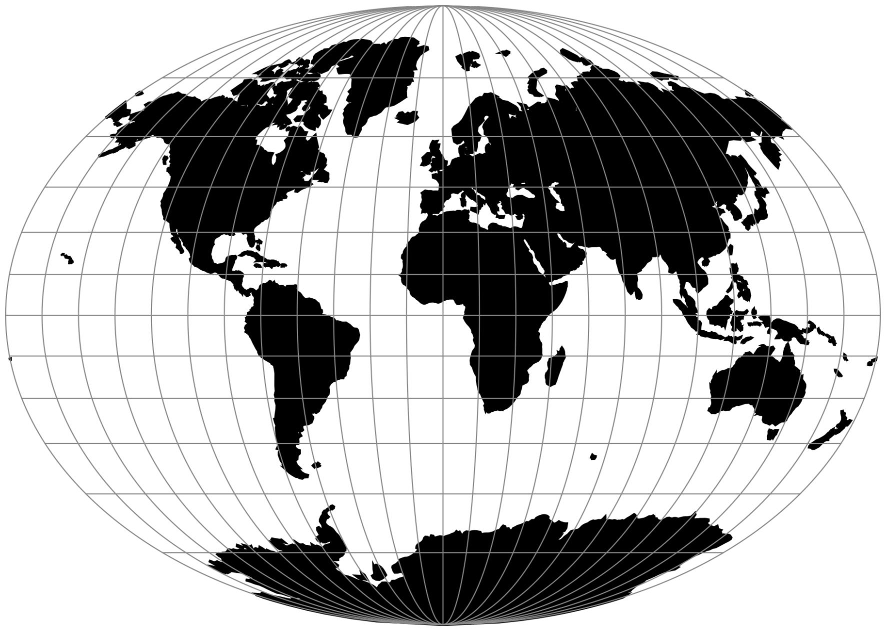


Fig. 31: proj-string: `+proj=fahey`

Parameters

Note: All parameters are optional for the Fahey projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.33 Foucaut

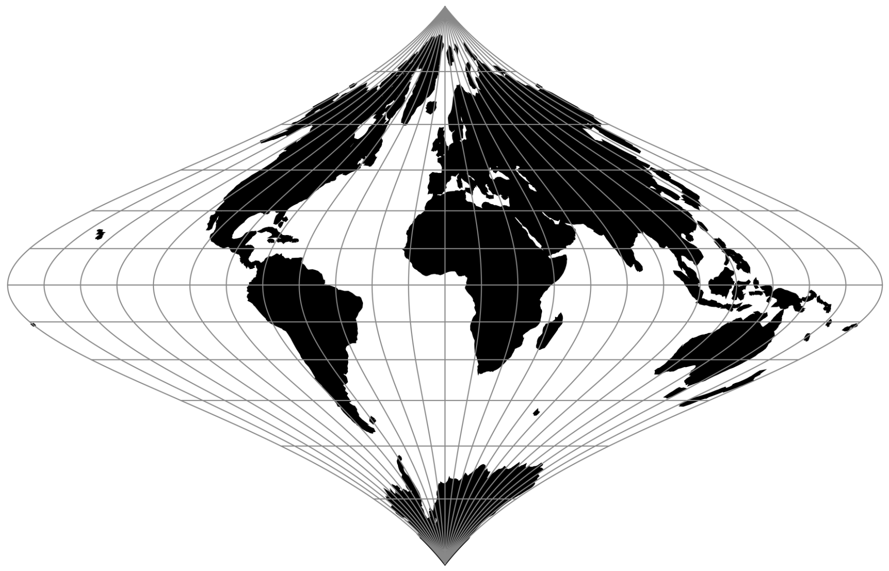


Fig. 32: proj-string: `+proj=fouc`

Parameters

Note: All parameters are optional for the Foucaut projection.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.34 Foucaut Sinusoidal

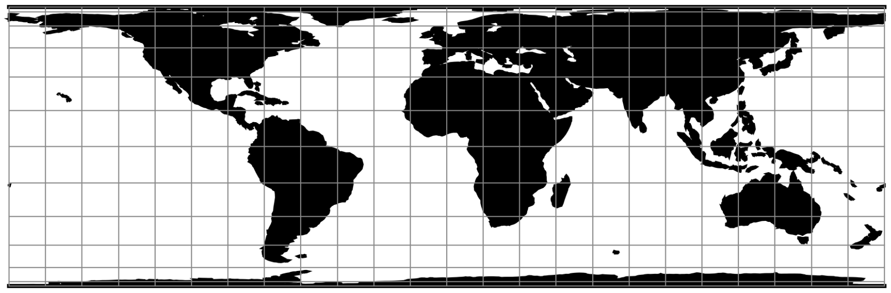


Fig. 33: proj-string: `+proj=fouc_s`

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 ϕ from the equation for y above. As $n \to 0$ and $\phi \to \pi/2$, convergence is slow but for $n = 0$, $\phi = \sin^1 y$

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>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.35 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

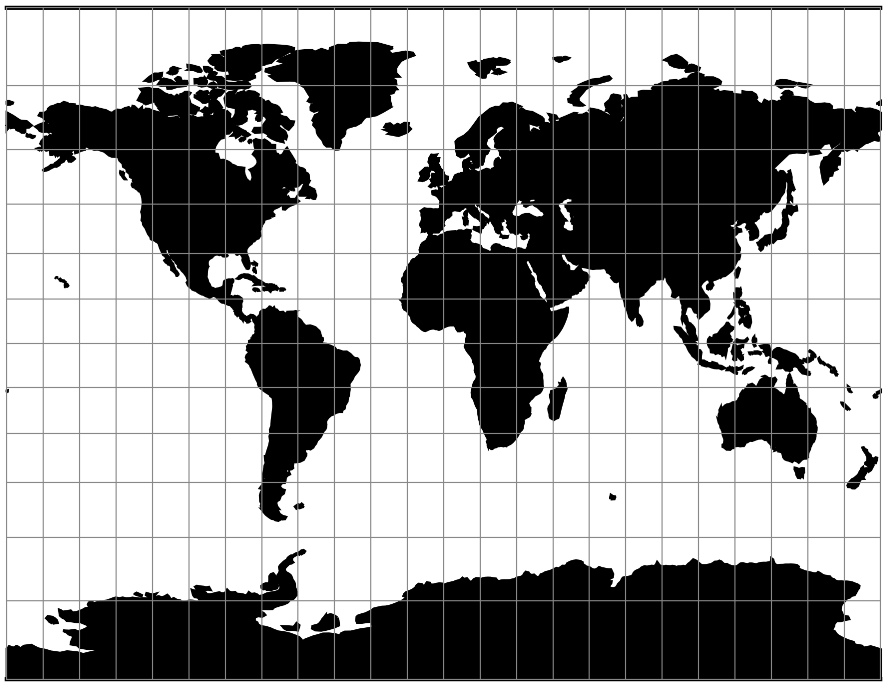


Fig. 34: proj-string: `+proj=gall`

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.

Example using Gall Stereographical

```
$ echo 9 51 | proj +proj=gall +lon_0=0 +x_0=0 +y_0=0 +ellps=WGS84 +datum=WGS84␣
↪+units=m +no_defs
708432.90   5193386.36
```

Example using Gall Stereographical (Central meridian 90°W)

```
$ echo 9 51 | proj +proj=gall +lon_0=90w +x_0=0 +y_0=0 +ellps=WGS84 +datum=WGS84␣
↪+units=m +no_defs
7792761.91  5193386.36
```

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

+ellps=<value>
See *proj -le* for a list of available ellipsoids.

Defaults to "WGS84".

Mathematical definition

The formulas describing the Gall Stereographical 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

$$\phi = 2\arctan(\frac{y}{1+\frac{\sqrt{2}}{2}})$$

$$\lambda = \sqrt{2}x$$

Further reading

1. Wikipedia
2. Cartographic Projection Procedures for the UNIX Environment-A User's Manual

7.1.36 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 elliptical projection
Defined area	Global
Alias	geos
Domain	2D
Input type	Geodetic coordinates
Output type	Projected coordinates

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`).

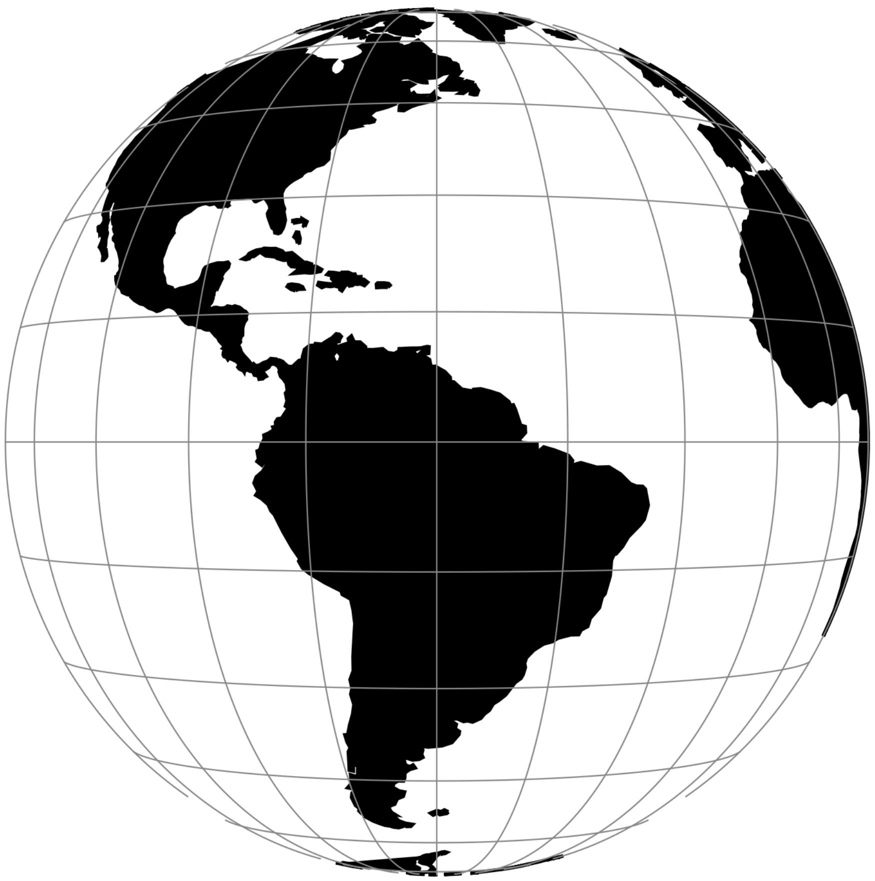
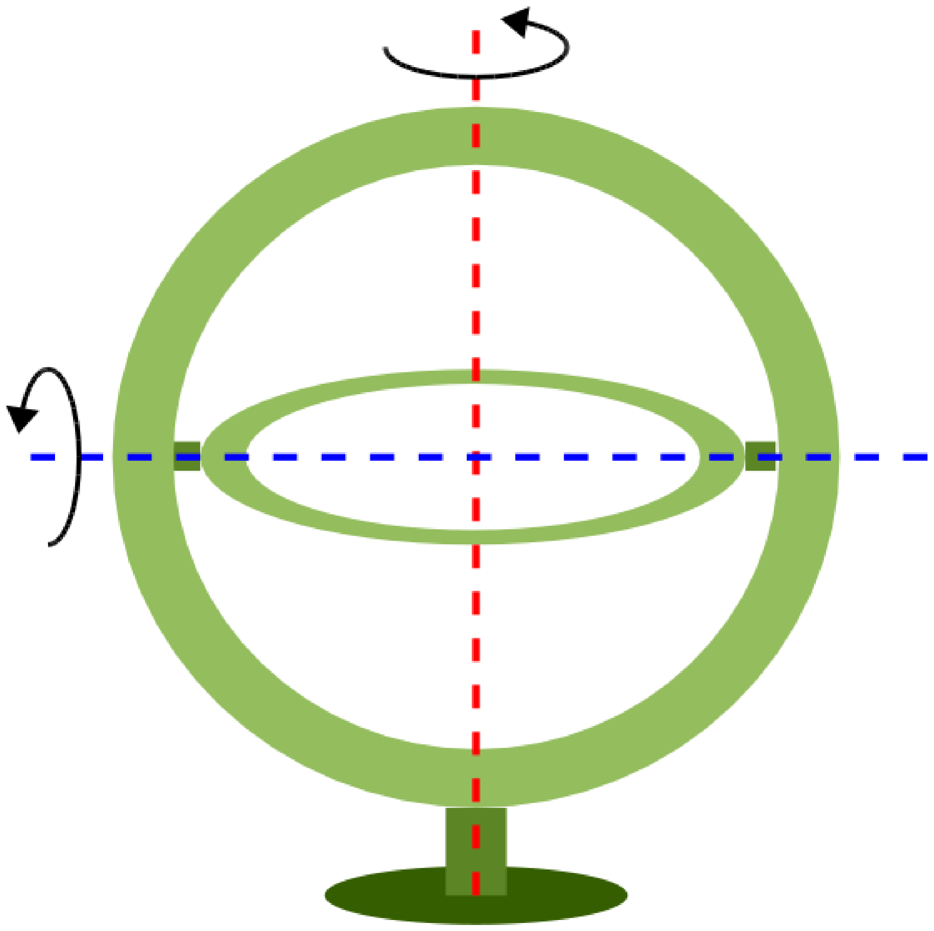


Fig. 35: proj-string: `+proj=geos +h=35785831.0 +lon_0=-60 +sweep=y`



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.

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>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.37 Ginsburg VIII (TsNIIGAiK)

Parameters

Note: All parameters are optional for the Ginsburg VIII projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

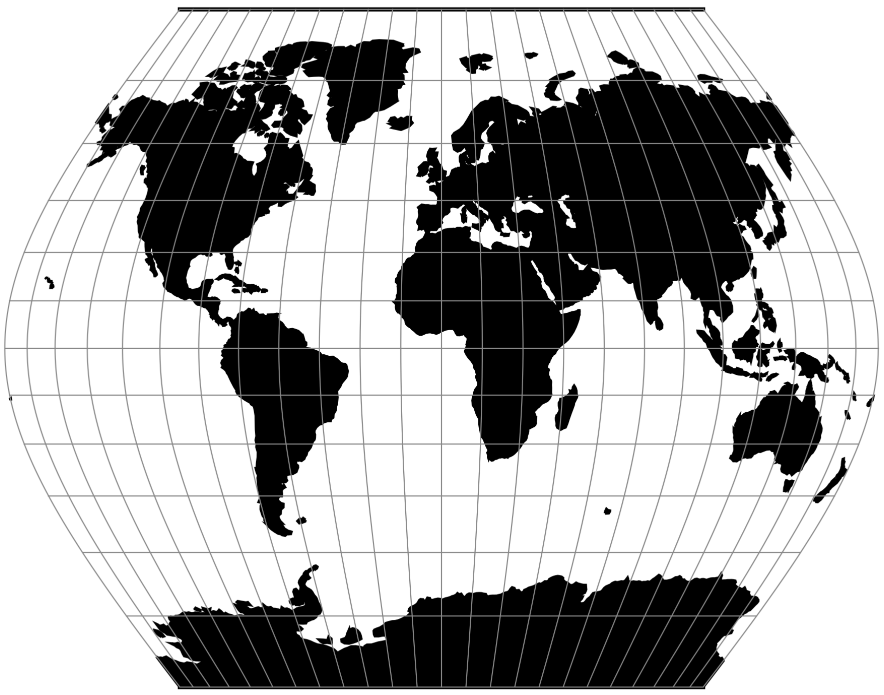


Fig. 36: proj-string: `+proj=gins8`

7.1.38 General Sinusoidal Series

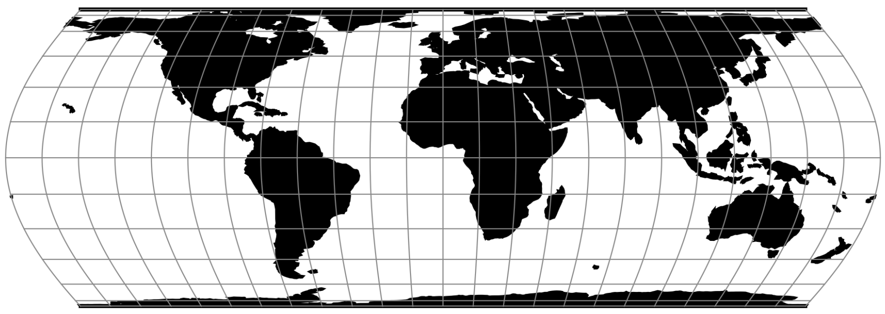


Fig. 37: proj-string: `+proj=gn_sinu +m=2 +n=3`

Parameters

Note: All parameters are optional for the General Sinusoidal Series projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.39 Gnomonic

Parameters

Note: All parameters are optional for the Gnomomic projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

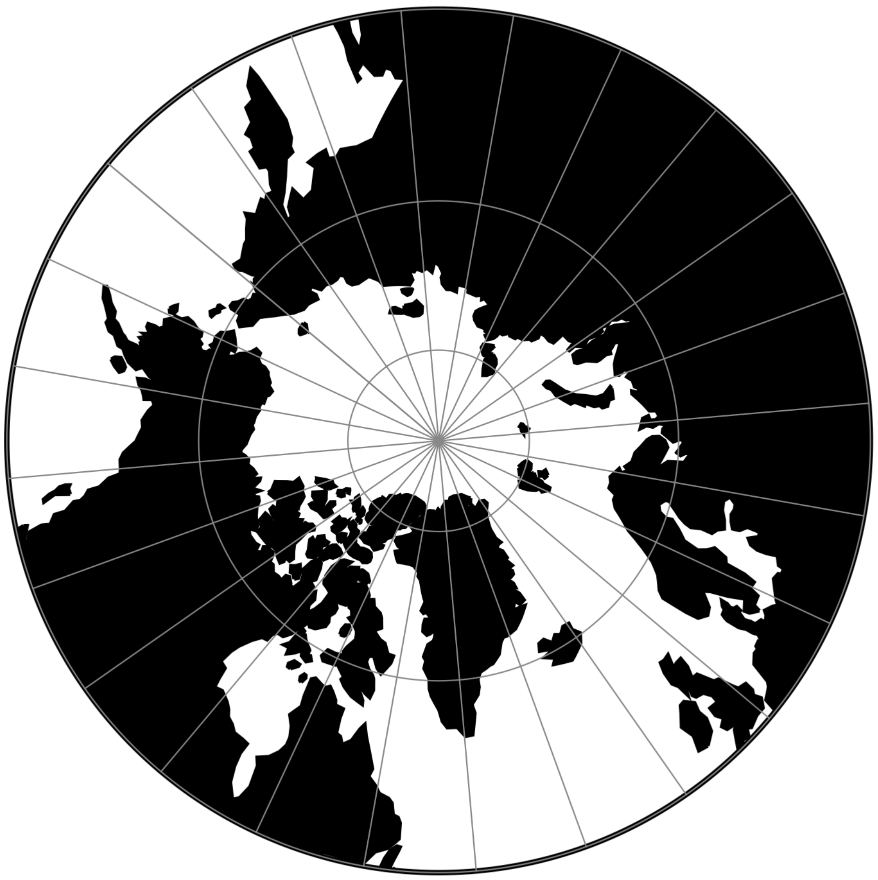


Fig. 38: proj-string: `+proj=gnom +lat_0=90 +lon_0=-50`

7.1.40 Goode Homolosine

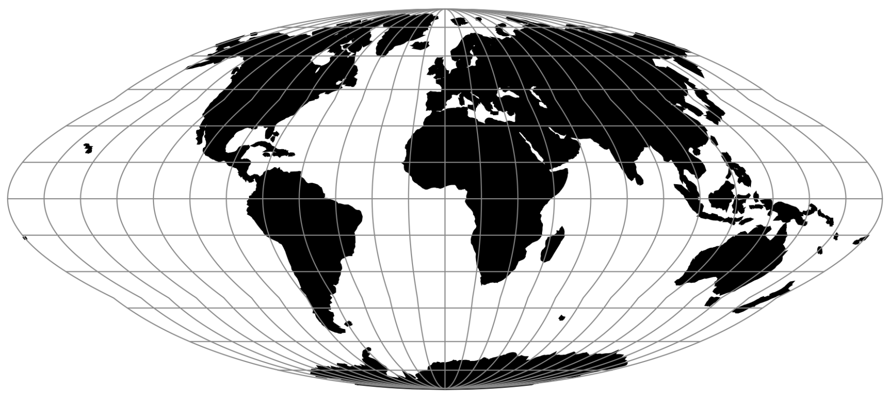


Fig. 39: proj-string: `+proj=goode`

Parameters

Note: All parameters are optional for the Goode Homolosine projection.

+lon_0=<value>
    Longitude of projection center.

    Defaults to 0.0.

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
    False easting.

    Defaults to 0.0.

+y_0=<value>
    False northing.

    Defaults to 0.0.

7.1.41 Mod. Stererographics of 48 U.S.

Parameters

Note: All parameters are optional for the projection.

+ellps=<value>
    See `proj -le` for a list of available ellipsoids.

    Defaults to "WGS84".

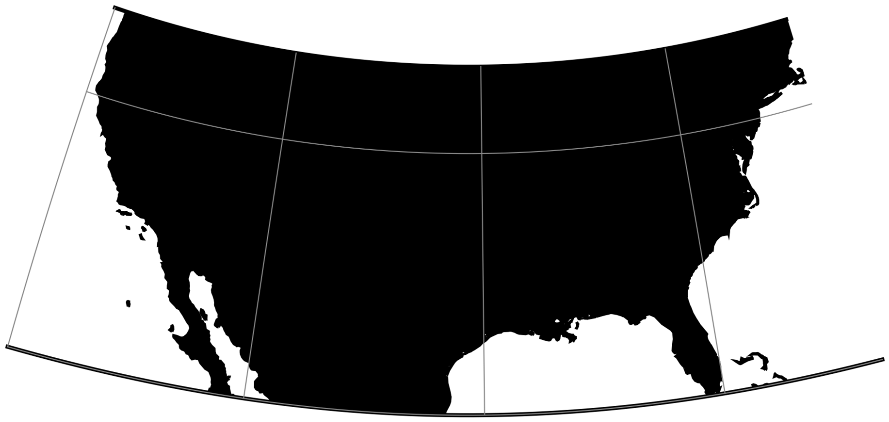


Fig. 40: proj-string: `+proj=gs48`

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.42 Mod. Stererographics of 50 U.S.

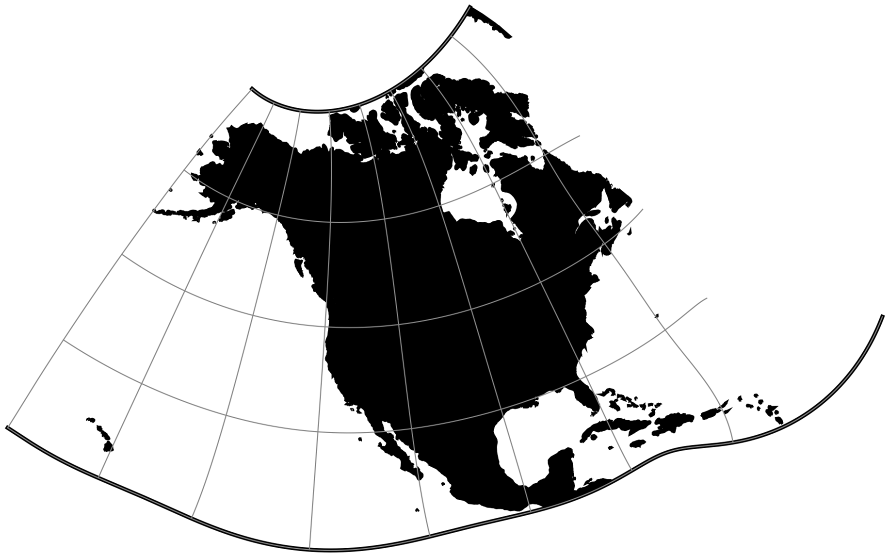


Fig. 41: proj-string: `+proj=gs50`

Parameters

Note: All parameters are optional for the projection.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.43 Hammer & Eckert-Greifendorff

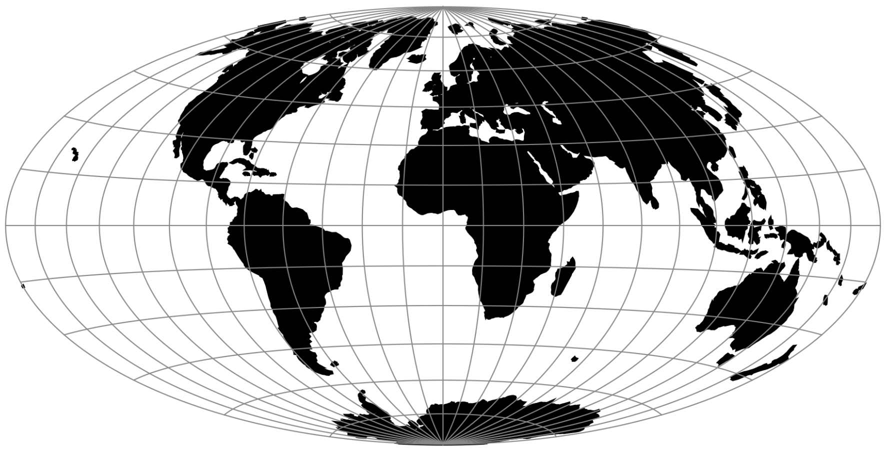


Fig. 42: proj-string: `+proj=hammer`

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.

+lon_0=<value>

Longitude of projection center.

Defaults to 0.0.

+R=<value>

Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>

False easting.

Defaults to 0.0.

+y_0=<value>

False northing.

Defaults to 0.0.

7.1.44 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

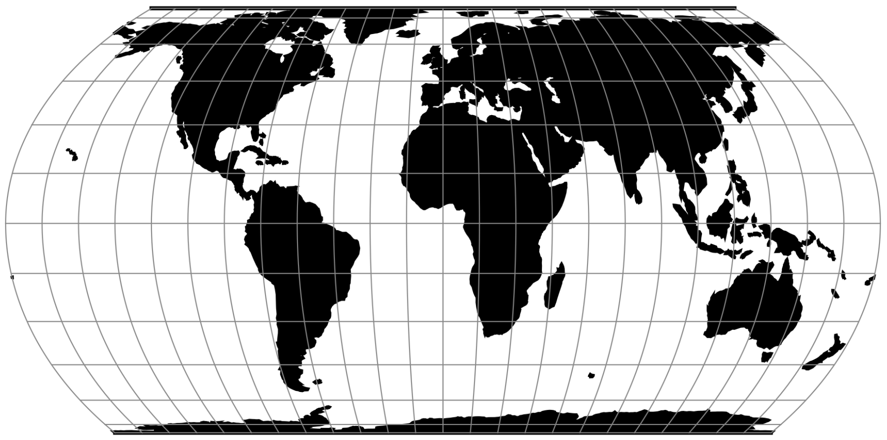


Fig. 43: proj-string: `+proj=hatano`

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

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

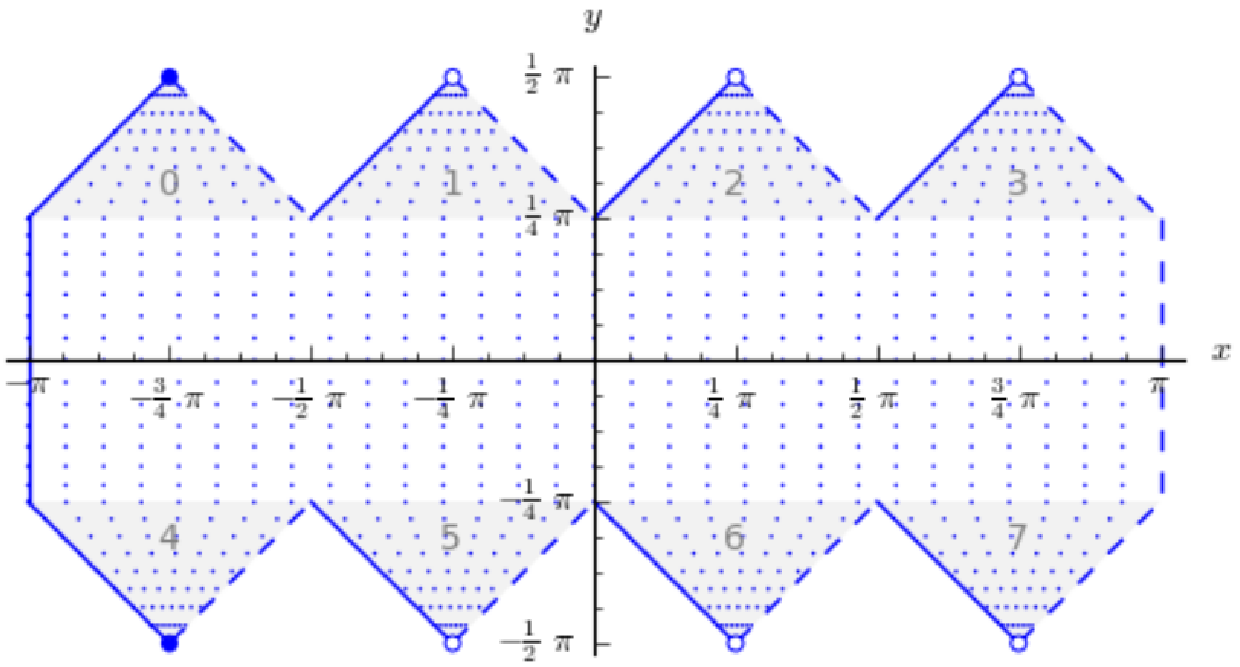
For $\phi = 0$, $y \leftarrow 0$, and $x \leftarrow 0.85\lambda$.

Further reading

1. Compare Map Projections
2. Mathworks

7.1.45 HEALPix

Classification	Miscellaneous
Available forms	Forward and inverse, spherical and elliptical projection
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.

Usage

To run a forward HEALPix projection on a unit sphere model, use the following command:

```
proj +proj=healpix +lon_0=0 +a=1 -E <<EOF
0 0
EOF
# output
0 0 0.00 0.00
```

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

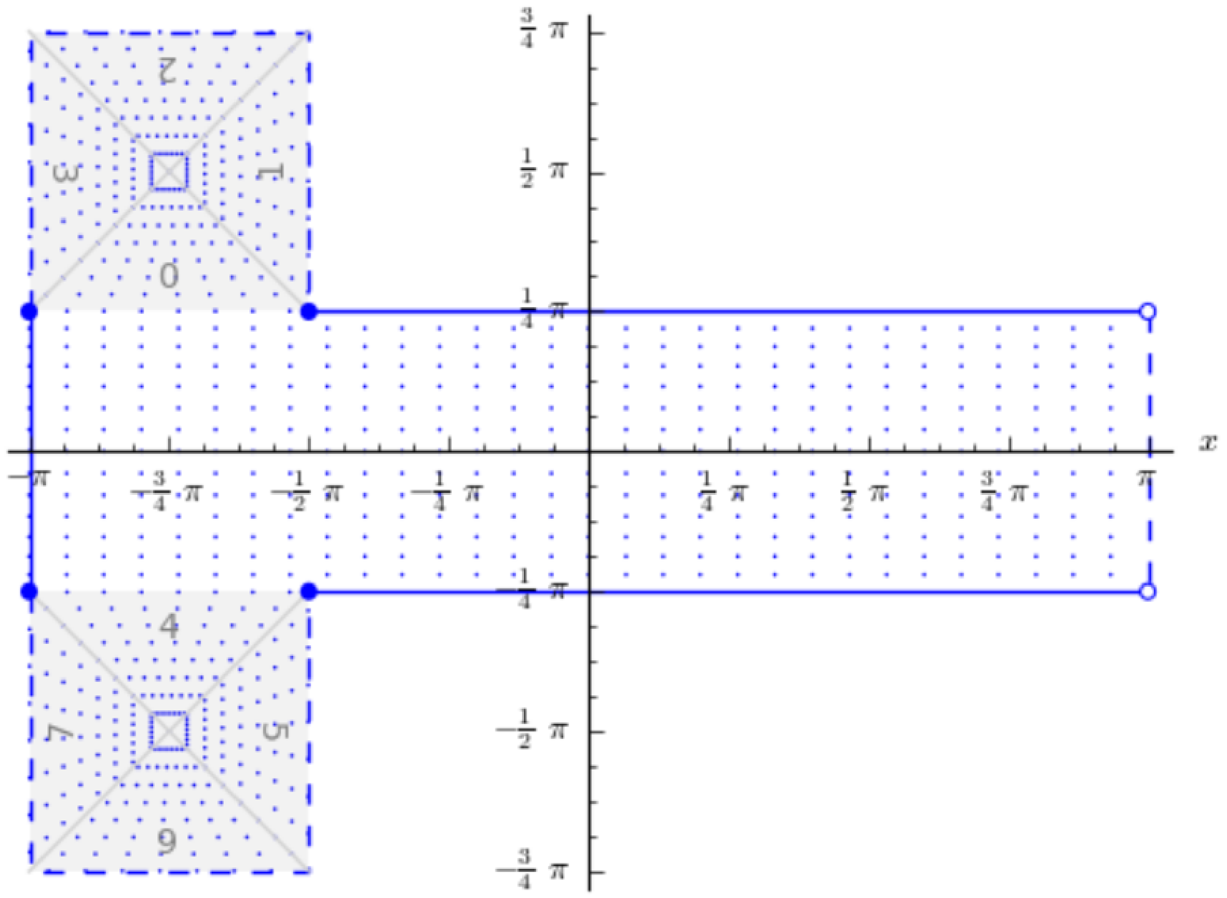
+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

Further reading

1. NASA
2. Wikipedia

7.1.46 rHEALPix

Classification	Miscellaneous
Available forms	Forward and inverse, spherical and elliptical projection
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).

Usage

To run a rHEALPix projection on a WGS84 ellipsoidal model, use the following command:

```
proj +proj=rhealpix -f '%.2f' +ellps=WGS84 +south_square=0 +north_square=2  -E << EOF
> 55 12
> EOF
55 12   6115727.86  1553840.13
```

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>
Longitude of projection center.

Defaults to 0.0.

+ellps=<value>
See *proj -le* for a list of available ellipsoids.

Defaults to "WGS84".

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

Further reading

1. NASA
2. Wikipedia

7.1.47 Interrupted Goode Homolosine

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

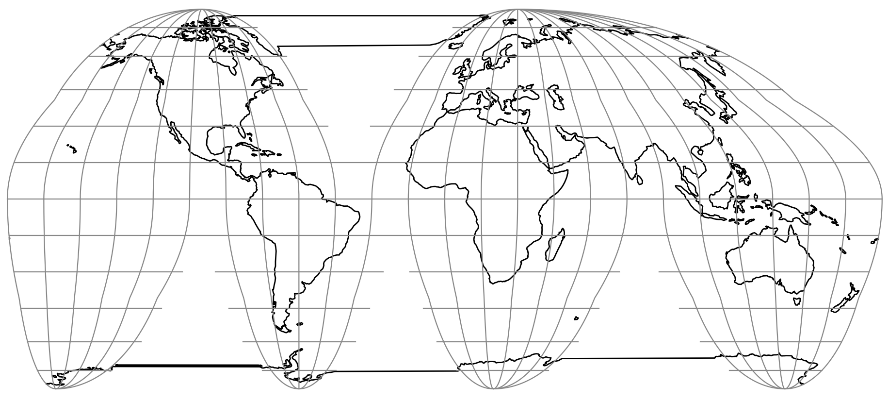


Fig. 44: proj-string: `+proj=igh`

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.48 International Map of the World Polyconic

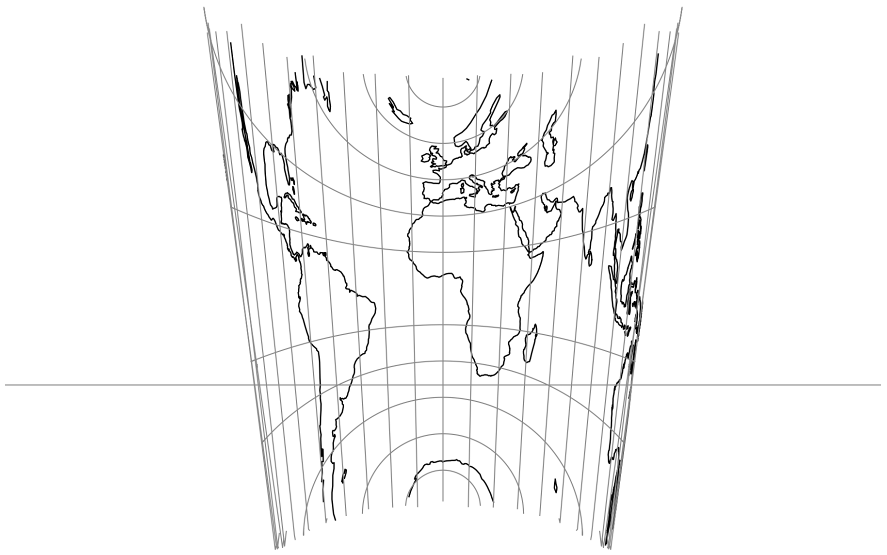


Fig. 45: proj-string: `+proj=imw_p +lat_1=30 +lat_2=-40`

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

Optional

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.49 Icosahedral Snyder Equal Area

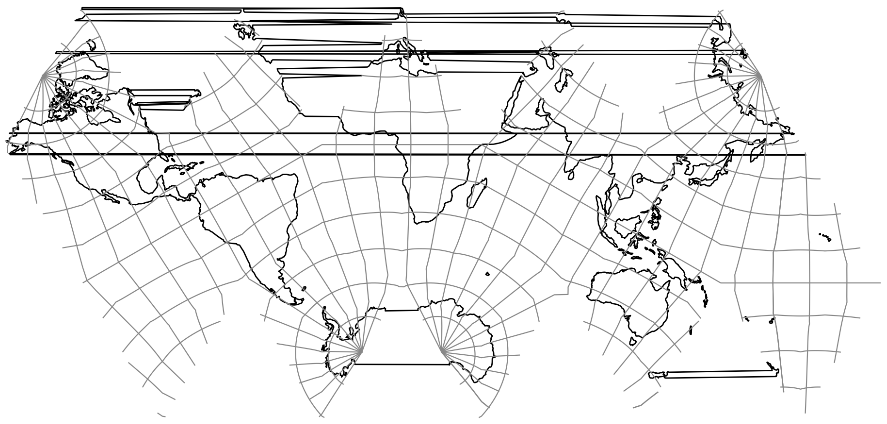


Fig. 46: proj-string: `+proj=isea`

Parameters

Note: All parameters are optional for the projection.

+orient=<string>
: Can be set to either `isea` or `pole`.

+azi=<value>
: Azimuth.

  Defaults to 0.0

+aperture=<value>
: *Defaults to 3.0*

+resolution=<value>
: *Defaults to 4.0*

+mode=<string>
: Can be either `plane`, `di`, `dd` or `hex`.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+lat_0=<value>
: Latitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.50 Kavraisky V

Parameters

Note: All parameters are optional for the Kavraisky V projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

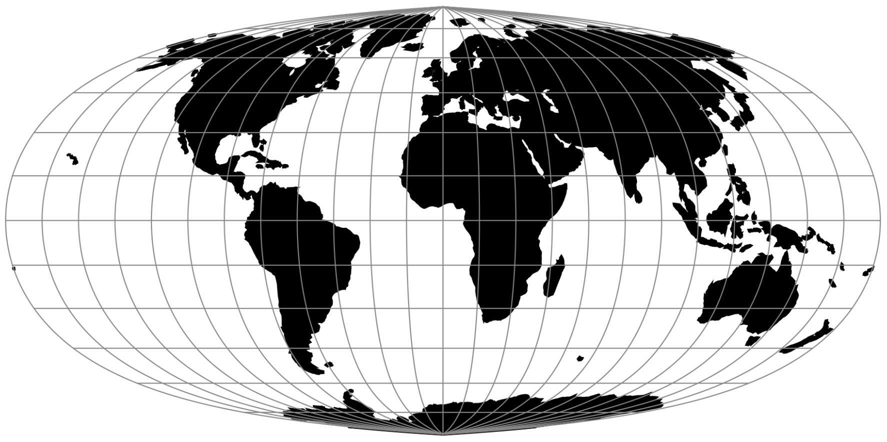


Fig. 47: proj-string: `+proj=kav5`

+x_0=`<value>`
False easting.

Defaults to 0.0.

+y_0=`<value>`
False northing.

Defaults to 0.0.

7.1.51 Kavraisky VII

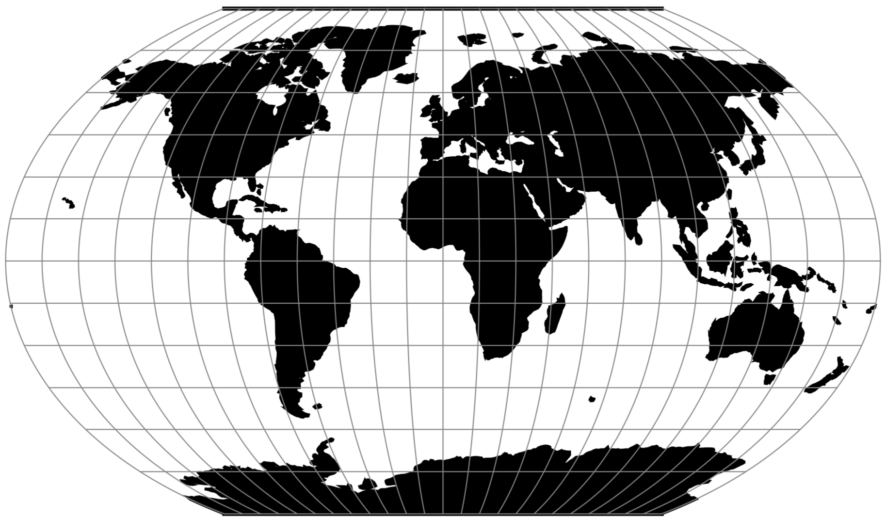


Fig. 48: proj-string: `+proj=kav7`

Parameters

Note: All parameters are optional for the Kavraisky VII projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.52 Krovak

Parameters

Note: All parameters are optional for the Krovak projection.

+czech
Reverse the sign of the output coordinates, as is tradition in the Czech Republic.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+k_0=<value>
Scale factor. Determines scale factor used in the projection.

Defaults to 0.9999.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

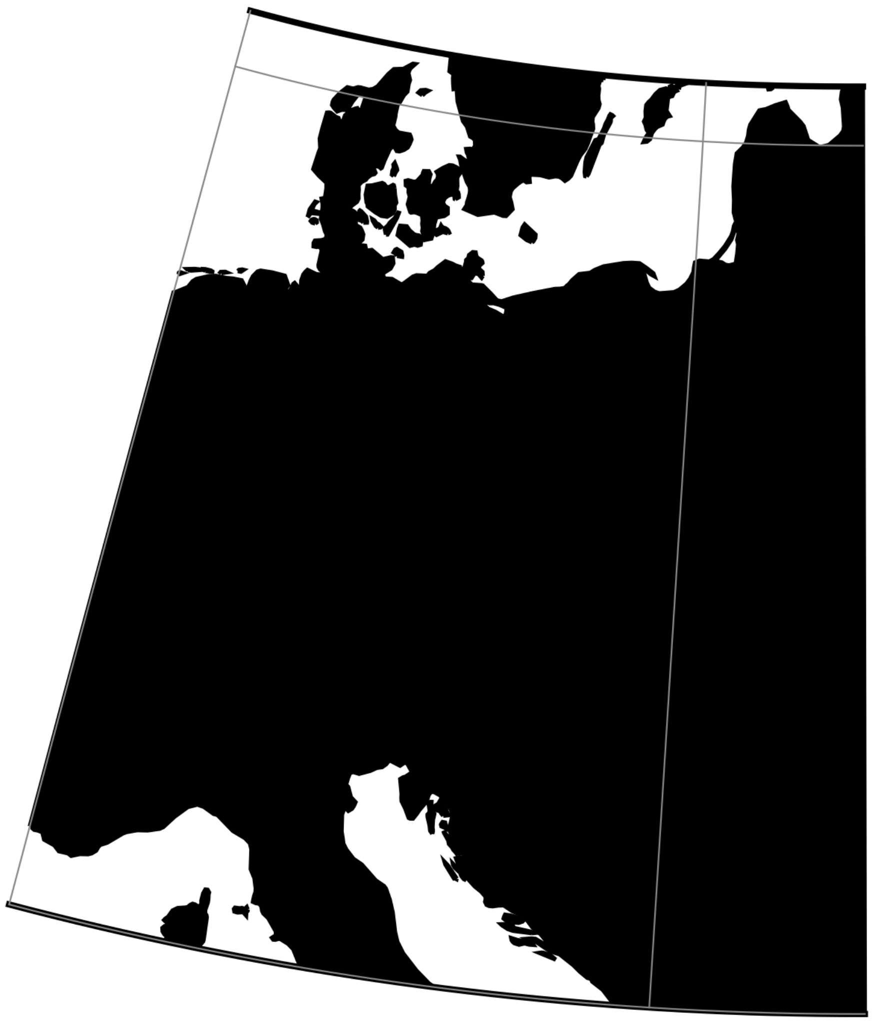


Fig. 49: proj-string: `+proj=krovak`

7.1.53 Laborde

Parameters

Required

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

Optional

+azi=<value>
Azimuth of the central line.

Defaults to 0.0

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.54 Lambert Azimuthal Equal Area

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

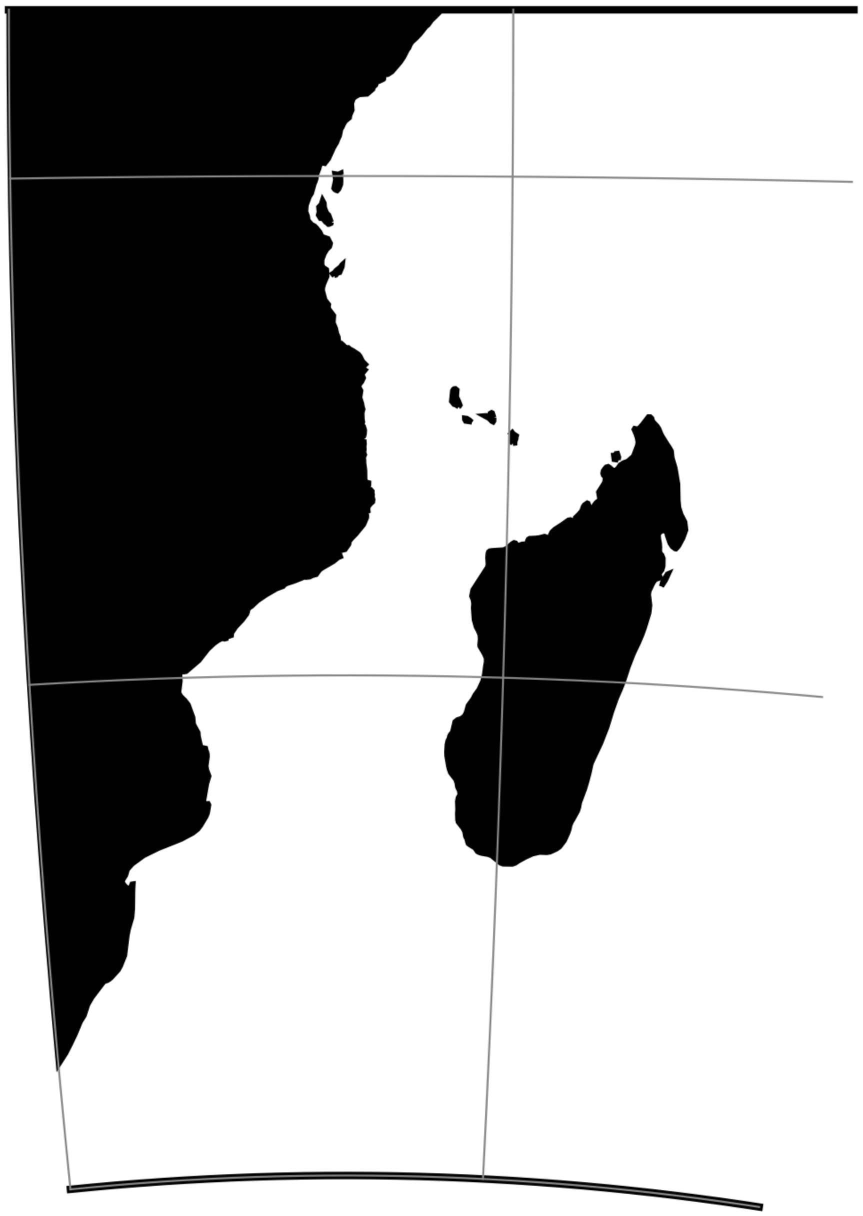


Fig. 50: proj-string: `+proj=labrd +lon_0=40 +lat_0=-10`

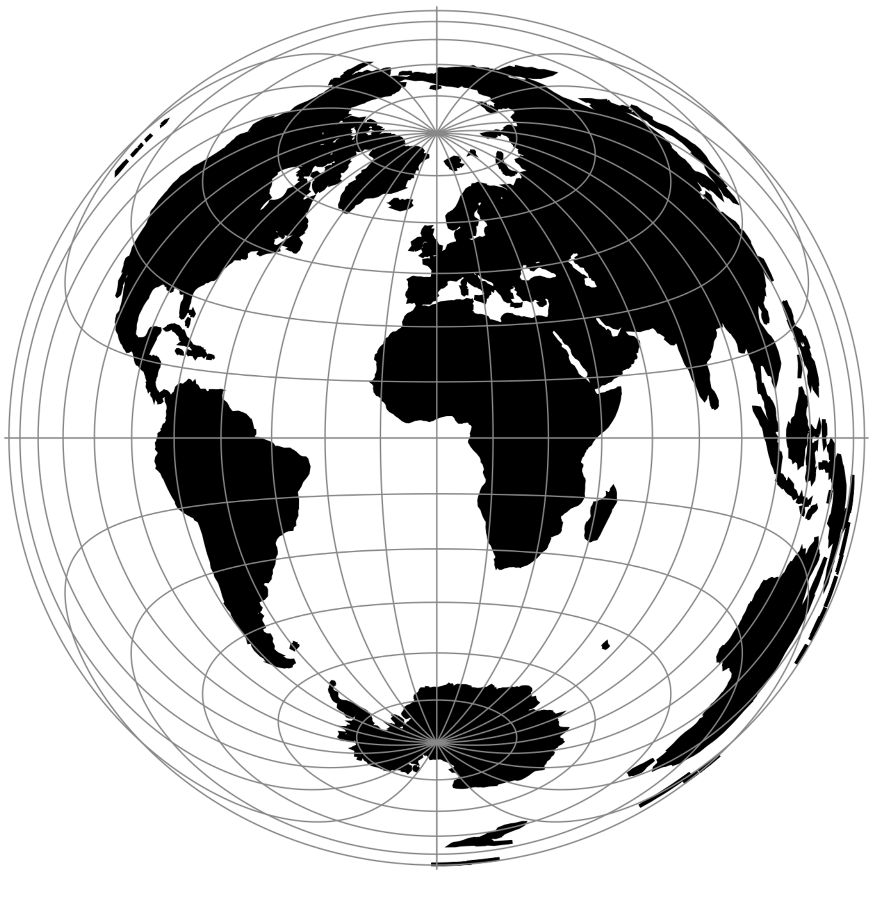


Fig. 51: proj-string: `+proj=laea`

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.55 Lagrange

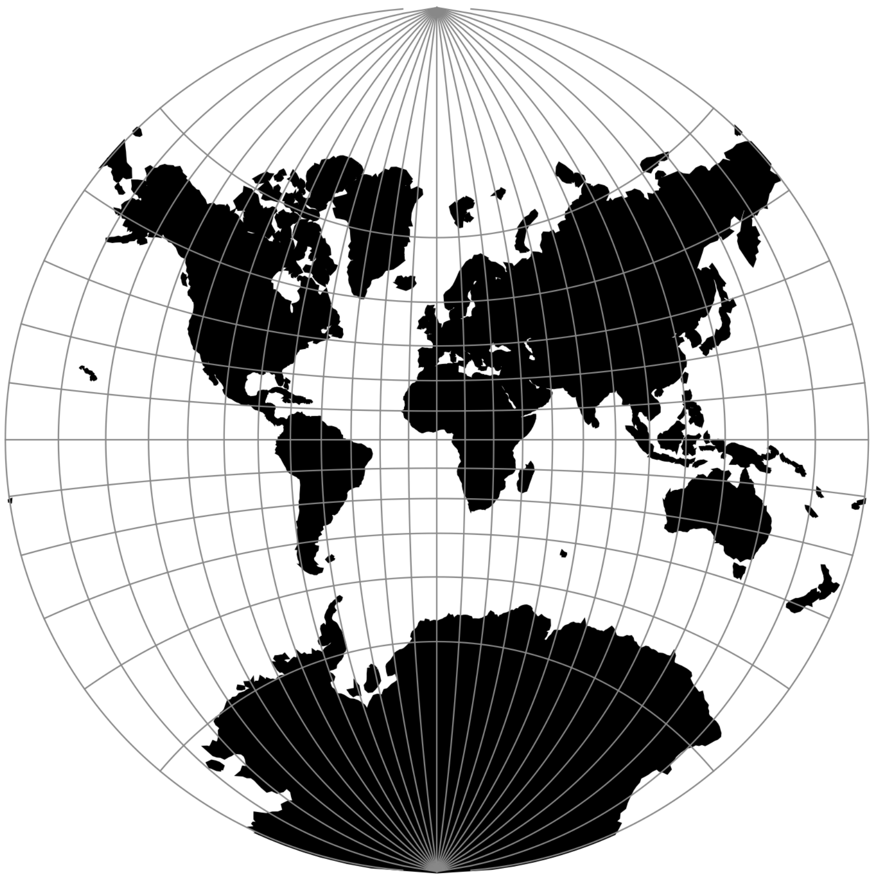


Fig. 52: proj-string: `+proj=lagrng`

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

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.56 Larrivee

Parameters

Note: All parameters are optional for the Larrivee projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

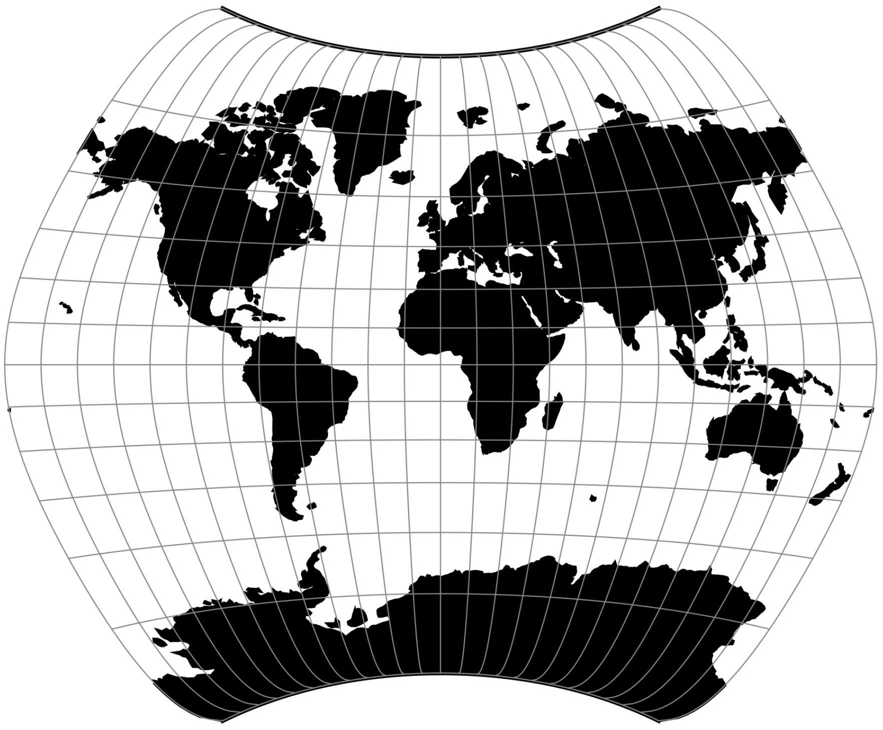


Fig. 53: proj-string: `+proj=larr`

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.57 Laskowski

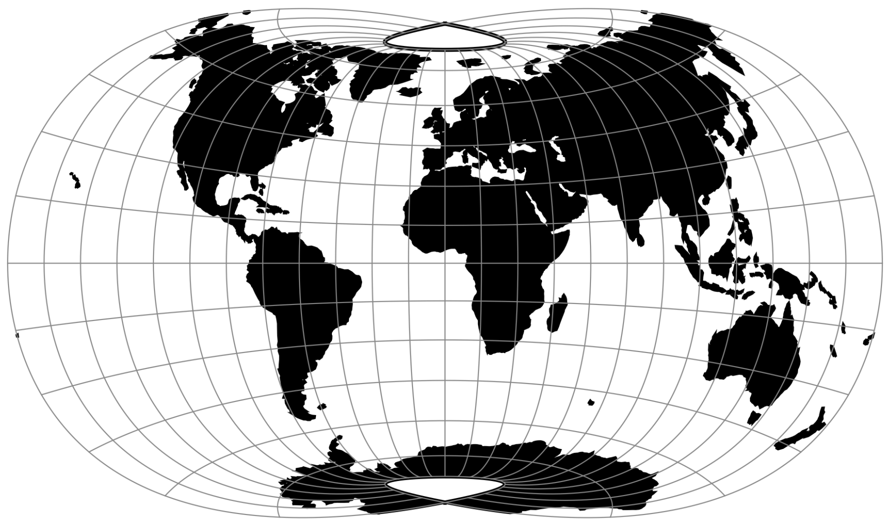


Fig. 54: proj-string: `+proj=lask`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.58 Lambert Conformal Conic

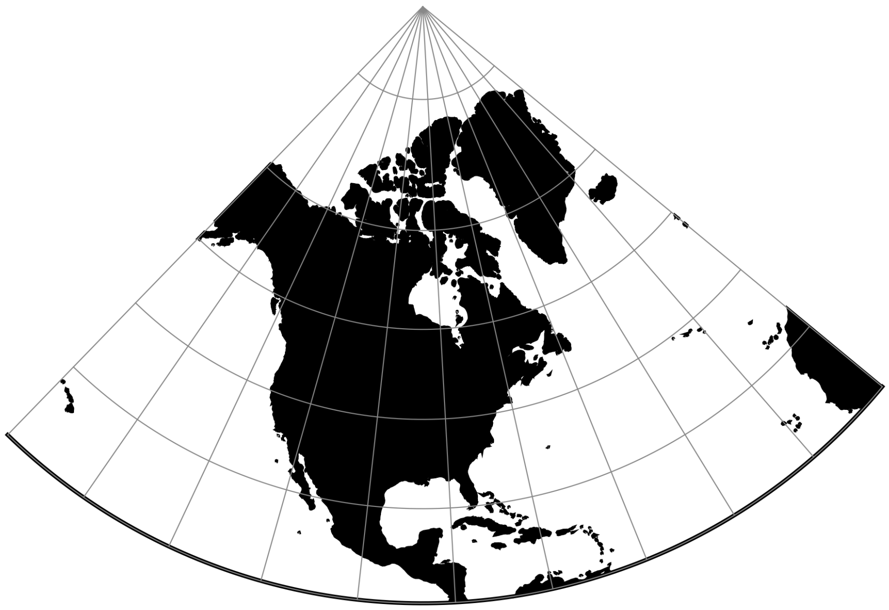


Fig. 55: proj-string: `+proj=lcc +lon_0=-90`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+lat_0=<value>
: Latitude of projection center.

  Defaults to 0.0.

+lat_1=<value>
: First standard parallel.

  Defaults to 0.0.

+lat_2=<value>
: Second standard parallel.

  Defaults to 0.0.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.59 Lambert Conformal Conic Alternative

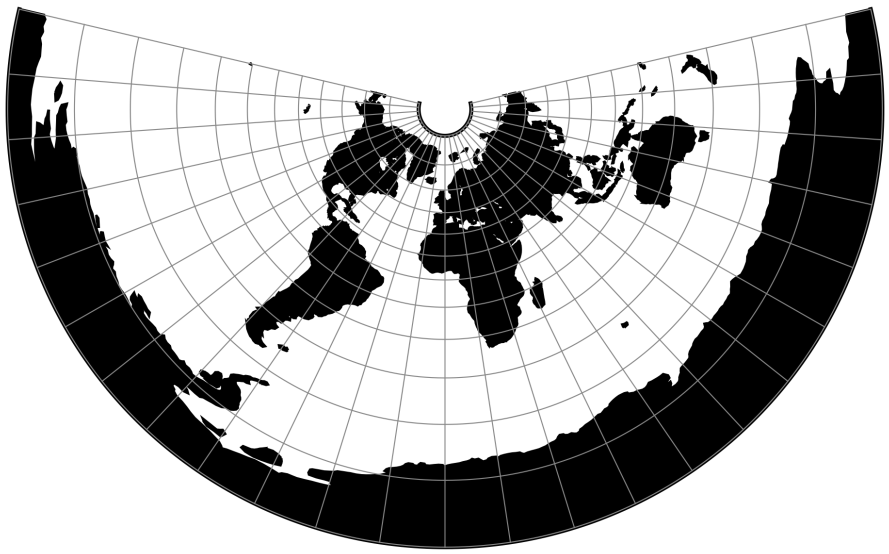


Fig. 56: proj-string: `+proj=lcca +lat_0=35`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+lat_0=<value>
: Latitude of projection center.

  Defaults to 0.0.

+ellps=<value>
: See *proj -le* for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.60 Lambert Equal Area Conic

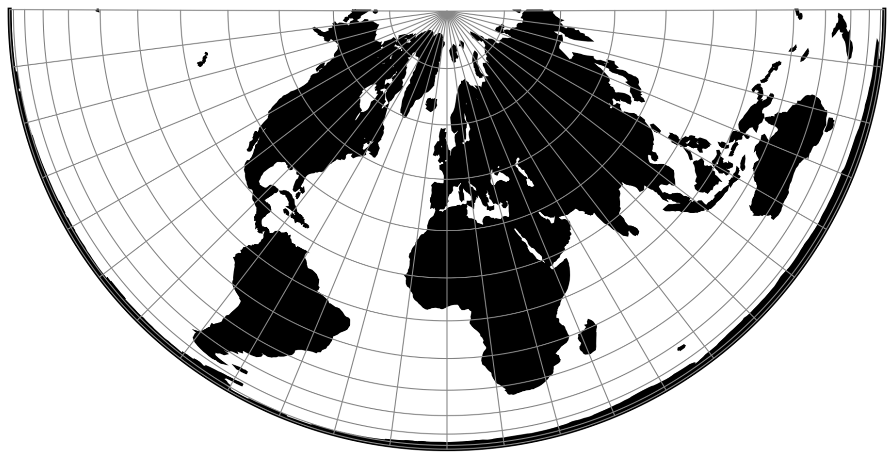


Fig. 57: proj-string: `+proj=leac`

Parameters

Note: All parameters are optional for the Lambert Equal Area Conic projection.

+lat_1=<value>
: First standard parallel.

  Defaults to 0.0.

+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>
: Longitude of projection center.

  Defaults to 0.0.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.61 Lee Oblated Stereographic

Parameters

Note: All parameters are optional for the projection.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.62 Loximuthal

Parameters

Note: All parameters are optional for the Loximuthal projection.

+lat_1=<value>
: First standard parallel.

  Defaults to 0.0.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

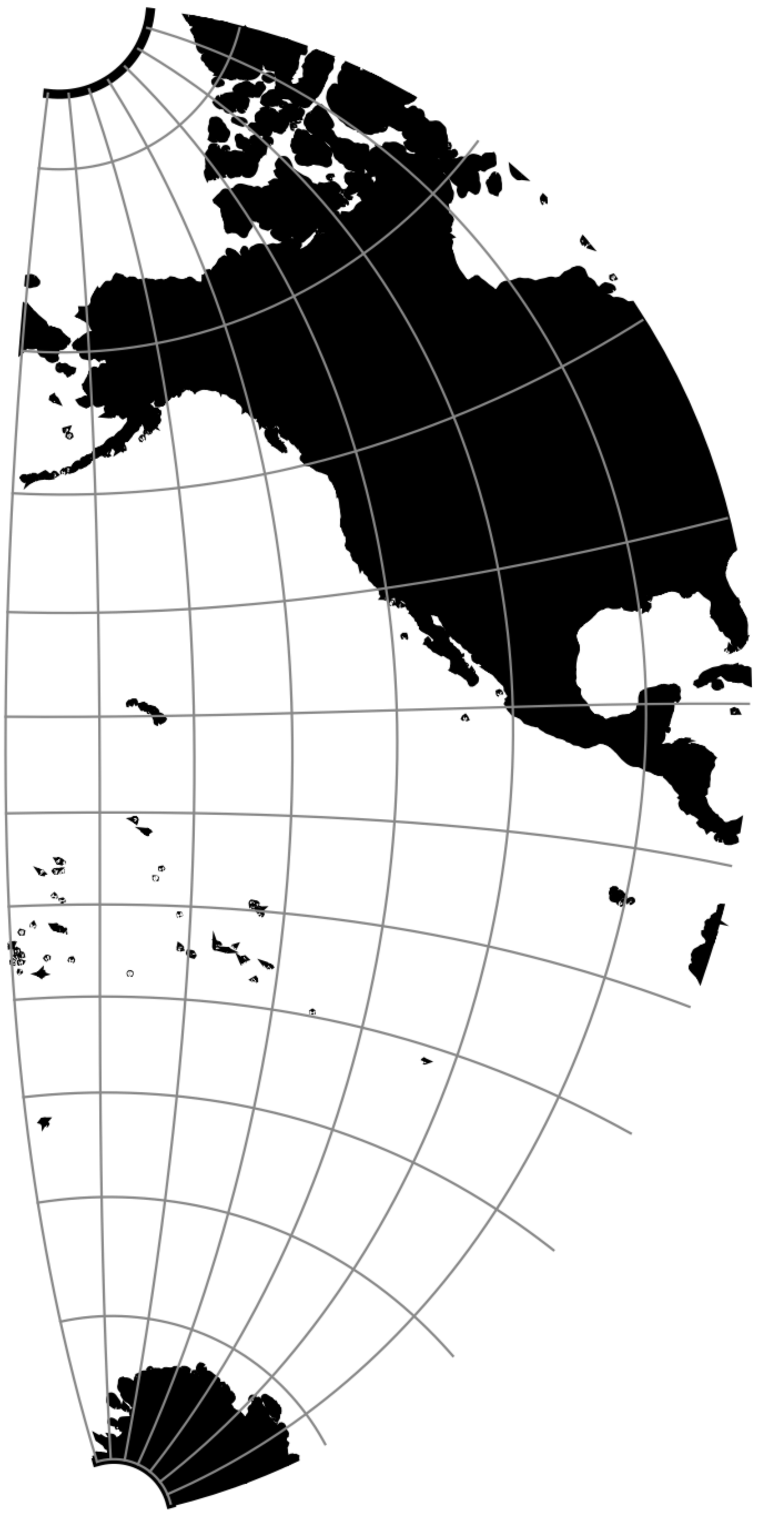


Fig. 58: proj-string: `+proj=lee_os`

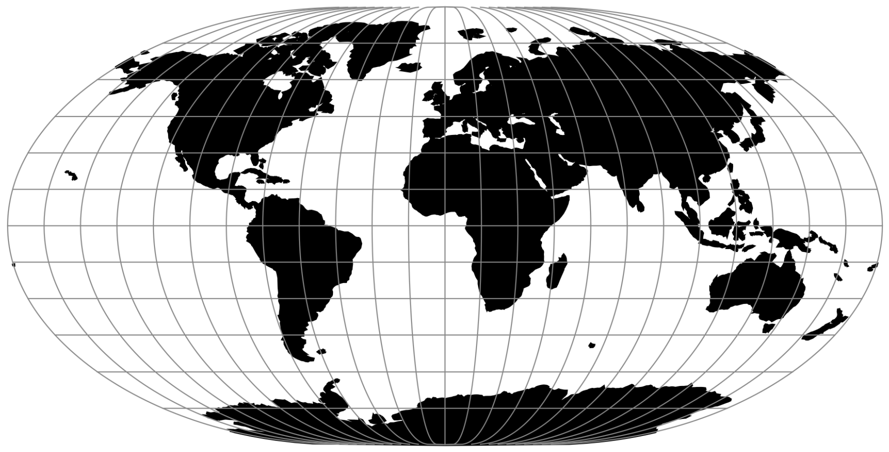


Fig. 59: proj-string: `+proj=loxim`

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.63 Space oblique for LANDSAT

Parameters

Required

+lsat=<value>
: Landsat satelite used for the projection. Value between 1 and 5.

+path=<value>
: Selected path of satelite. Value between 1 and 253 when *+lsat* is set to 1,2 or 3, otherwise valid input is between 1 and 233.

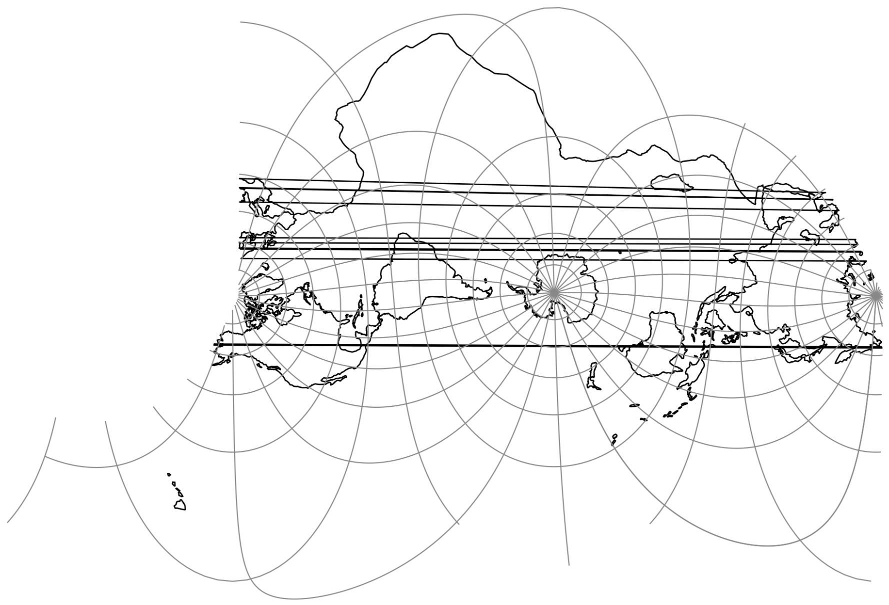


Fig. 60: proj-string: `+proj=lsat +ellps=GRS80 +lat_1=-60 +lat_2=60 +lsat=2 +path=2`

Optional

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.64 McBryde-Thomas Flat-Polar Sine (No. 1)

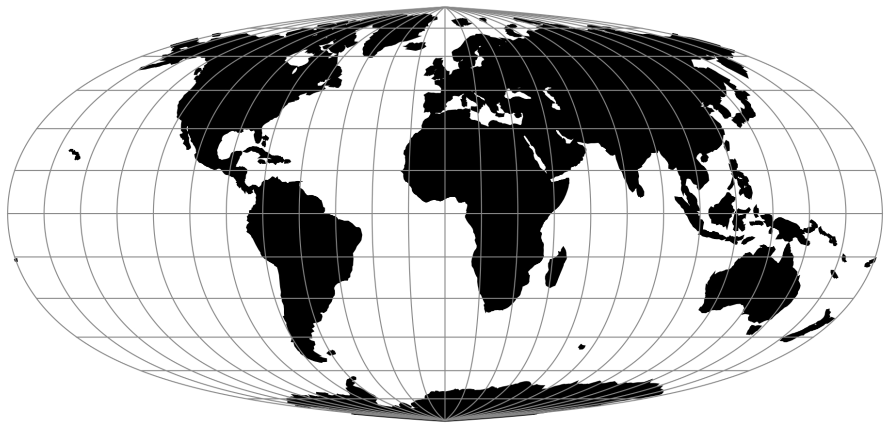


Fig. 61: proj-string: `+proj=mbt_s`

Parameters

Note: All parameters are optional for the McBryde-Thomas Flat-Polar Sine projection.

`+lon_0``=<value>`
: Longitude of projection center.

  Defaults to 0.0.

`+R``=<value>`
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
: False easting.

  Defaults to 0.0.

`+y_0``=<value>`
: False northing.

  Defaults to 0.0.

7.1.65 McBryde-Thomas Flat-Pole Sine (No. 2)

Parameters

Note: All parameters are optional.

`+lon_0``=<value>`
: Longitude of projection center.

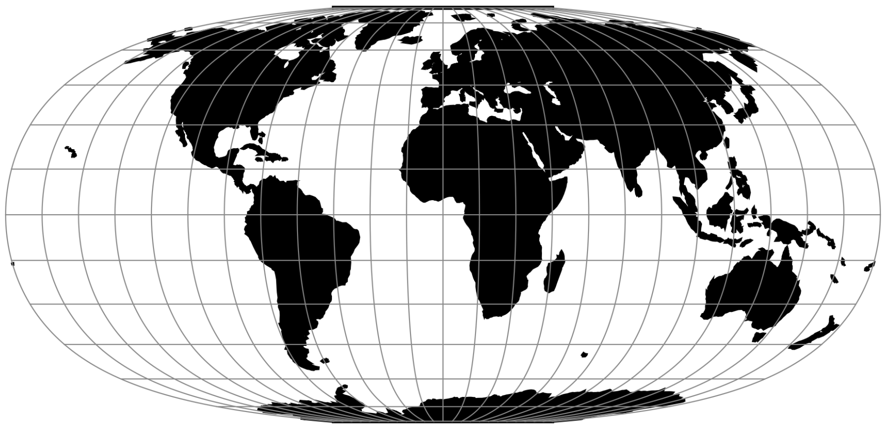


Fig. 62: proj-string: `+proj=mbt_fps`

Defaults to 0.0.

+R=`<value>`
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=`<value>`
: False easting.

Defaults to 0.0.

+y_0=`<value>`
: False northing.

Defaults to 0.0.

7.1.66 McBride-Thomas Flat-Polar Parabolic

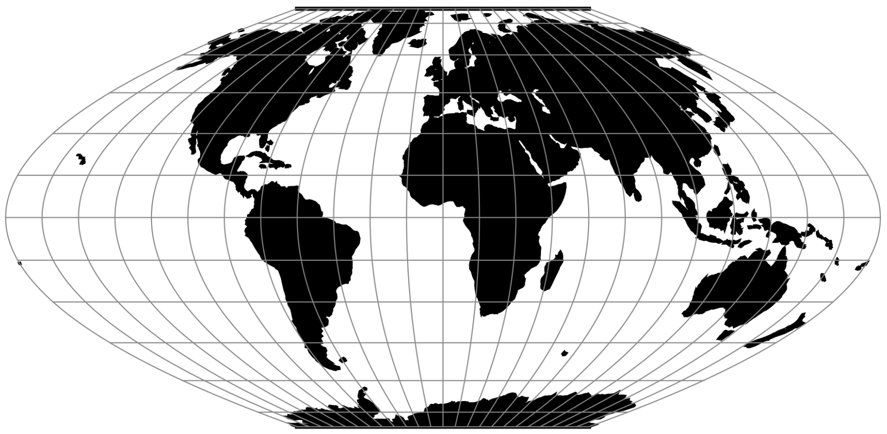


Fig. 63: proj-string: `+proj=mbtfpp`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.67 McBryde-Thomas Flat-Polar Quartic

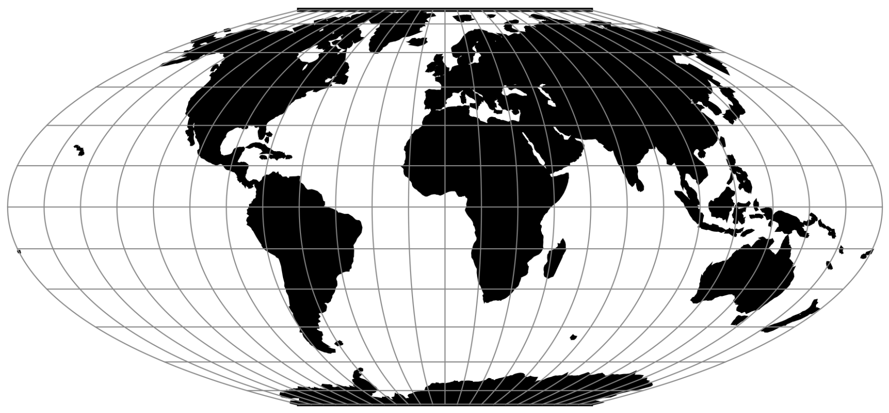


Fig. 64: proj-string: `+proj=mbtfpq`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.68 McBryde-Thomas Flat-Polar Sinusoidal

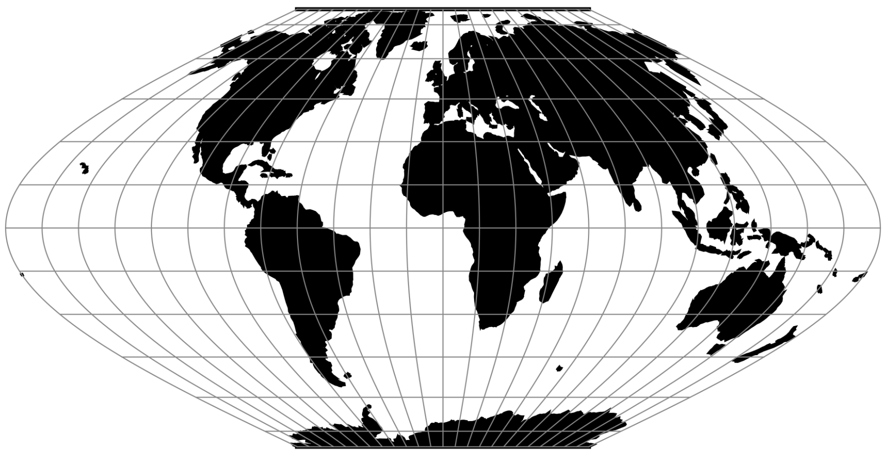


Fig. 65: proj-string: `+proj=mbtfps`

Parameters

Note: All parameters are optional for the McBryde-Thomas Flat-Polar Sinusoidal projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.69 Mercator

The Mercator projection is a cylindrical map projection that origins from the 15th century. It is widely recognized as the first regularly used map projection. The projection is conformal which makes it suitable for navigational purposes.

Classification	Conformal cylindrical
Available forms	Forward and inverse, spherical and elliptical projection
Defined area	Global, but best used near the equator
Alias	merc
Domain	2D
Input type	Geodetic coordinates
Output type	Projected coordinates

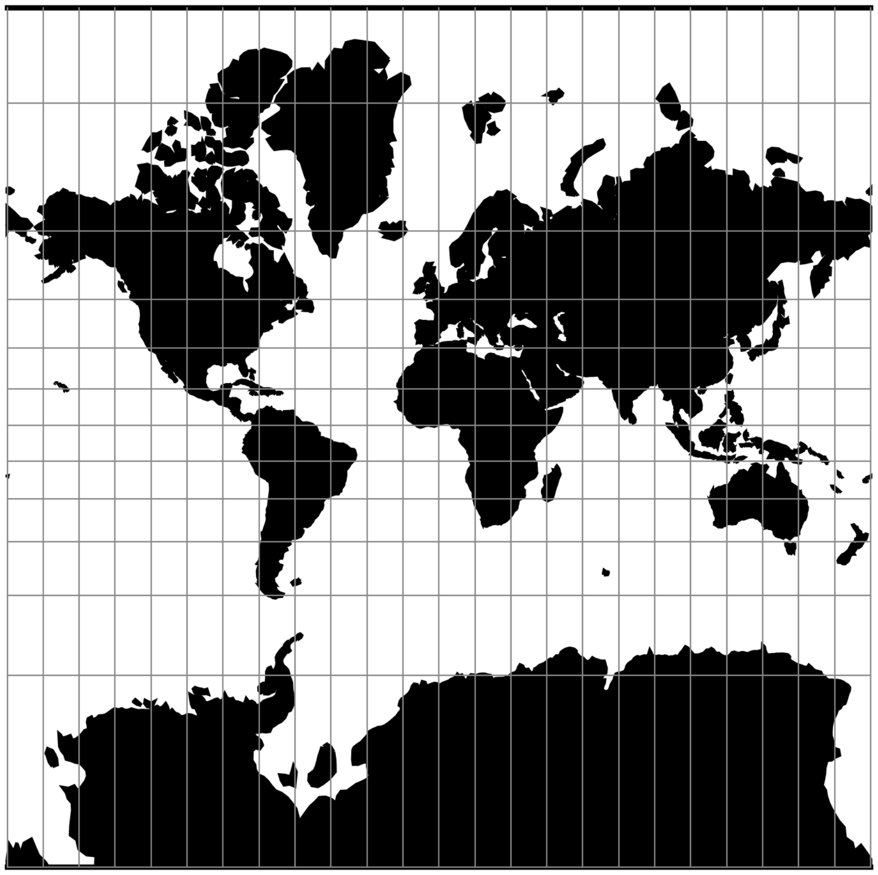


Fig. 66: proj-string: `+proj=merc`

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. Often inappropriately used for world maps since the regions near the poles cannot be shown [Evenden1995].

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:

```
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*.

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.

+k_0=<value>
Scale factor. Determines scale factor used in the projection.

Defaults to 1.0.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

+ellps=<value>
See *proj -le* for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with +ellps +R takes precedence.

Mathematical definition

The formulas describing the Mercator projection are all taken from G. Evenden's libproj manuals [Evenden2005].

Spherical form

For the spherical form of the projection we introduce the scaling factor:

$$k_0 = \cos \phi_{ts}$$

Forward projection

$$x = k_0 \lambda$$

$$y = k_0 \ln \left[\tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right) \right]$$

Inverse projection

$$\lambda = \frac{x}{k_0}$$

$$\phi = \frac{\pi}{2} - 2 \arctan \left[e^{-y/k_0} \right]$$

Elliptical form

For the elliptical form of the projection we introduce the scaling factor:

$$k_0 = m\left(\phi_{ts}\right)$$

where $m\left(\phi\right)$ is the parallel radius at latitude ϕ.

We also use the Isometric Latitude kernel function $t()$.

Note: m() and t() should be described properly on a separate page about the theory of projections on the ellipsoid.

Forward projection

$$x = k_0 \lambda$$

$$y = k_0 \ln t\left(\phi\right)$$

Inverse projection

$$\lambda = \frac{x}{k_0}$$

$$\phi = t^{-1}\left[e^{-y/k_0}\right]$$

Further reading

1. Wikipedia
2. Wolfram Mathworld

7.1.70 Miller Oblated Stereographic

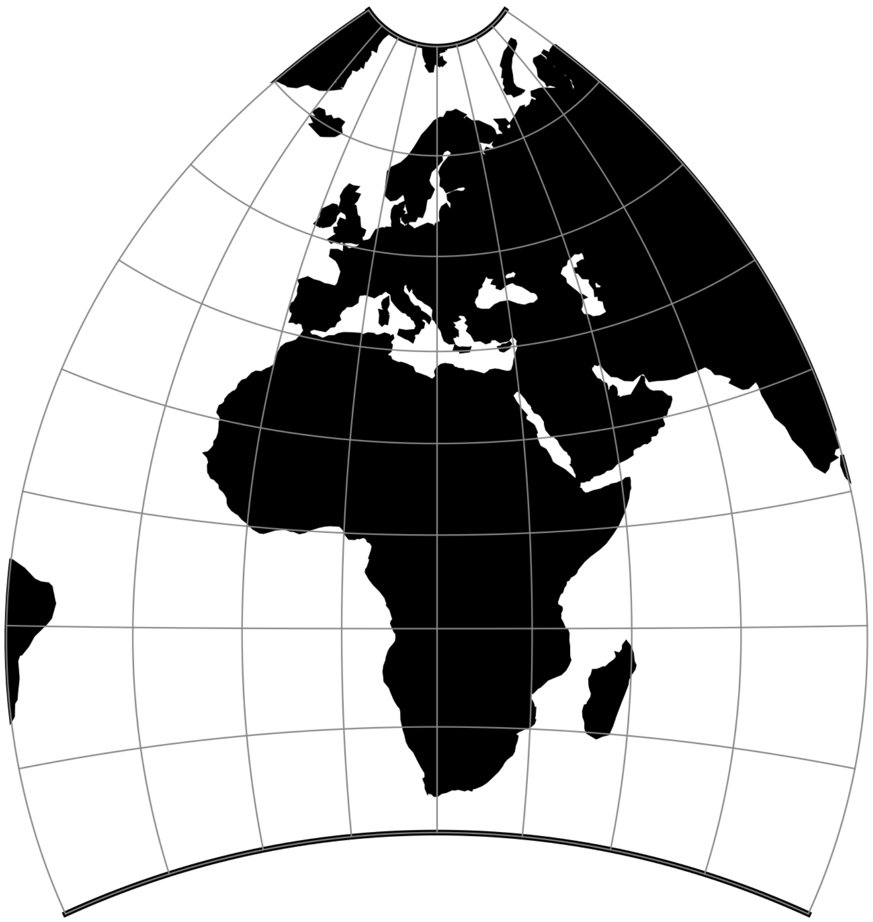


Fig. 67: proj-string: `+proj=mil_os`

Parameters

Note: All parameters are optional for the projection.

+ellps=<value>
See proj -le for a list of available ellipsoids.

Defaults to "WGS84".

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.71 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

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
```

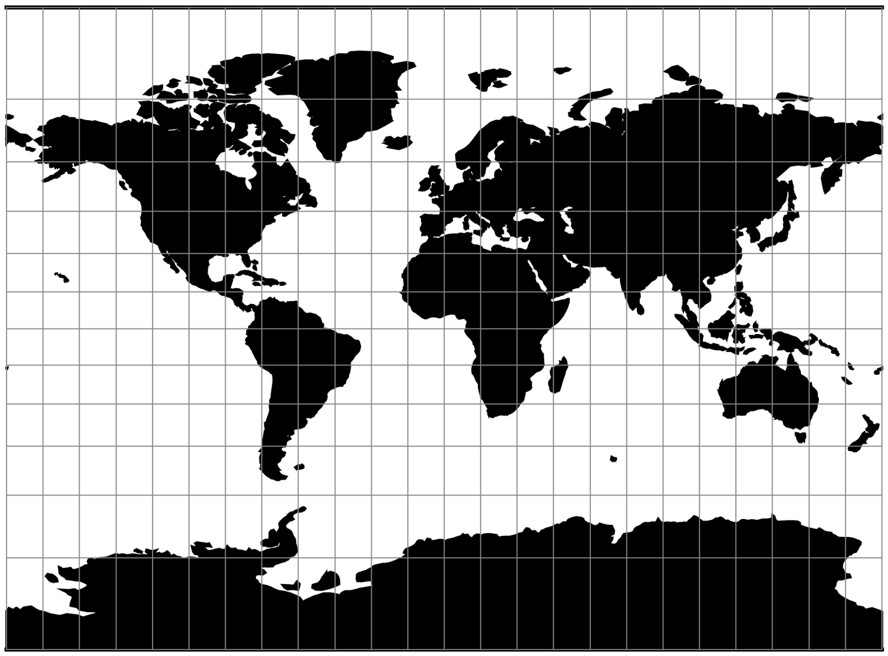


Fig. 68: proj-string: `+proj=mill`

Parameters

Note: All parameters for the projection are optional.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

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

$$\lambda = x$$

$$\phi = 2.5 * (\arctan\left[e^{0.8*y}\right] - \frac{\pi}{4})$$

Further reading

1. Wikipedia

7.1.72 Mollweide

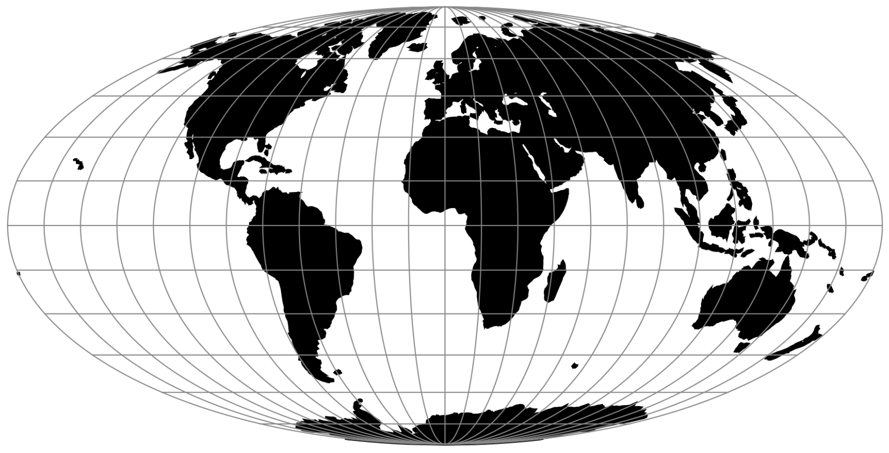


Fig. 69: proj-string: `+proj=moll`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.73 Murdoch I

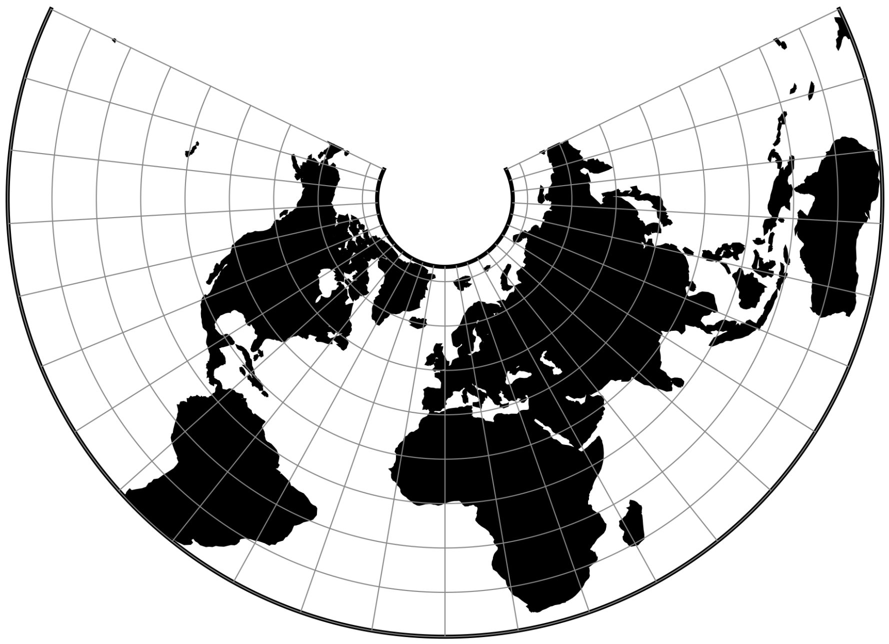


Fig. 70: proj-string: `+proj=murd1 +lat_1=30 +lat_2=50`

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

Optional

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.74 Murdoch II

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

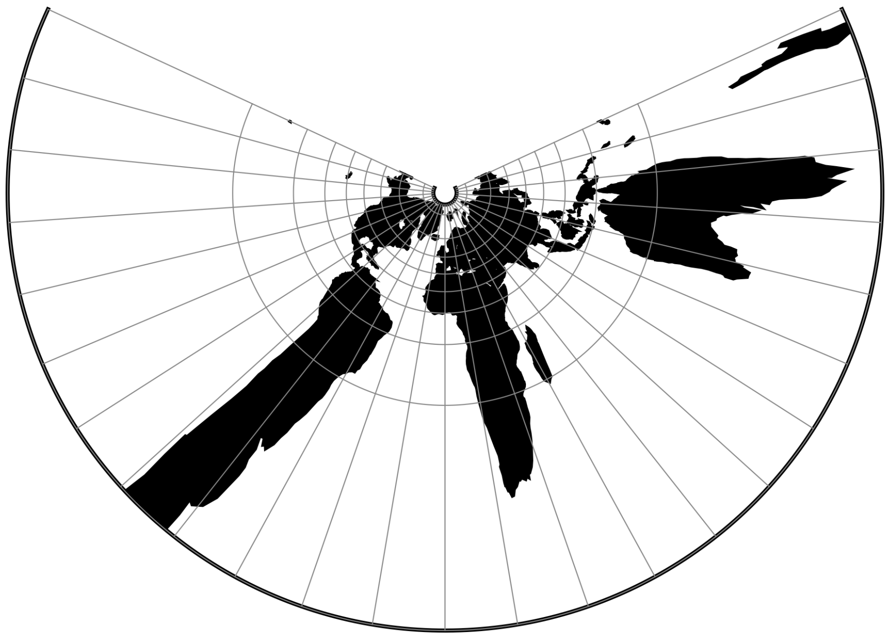


Fig. 71: proj-string: `+proj=murd2 +lat_1=30 +lat_2=50`

Optional

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.75 Murdoch III

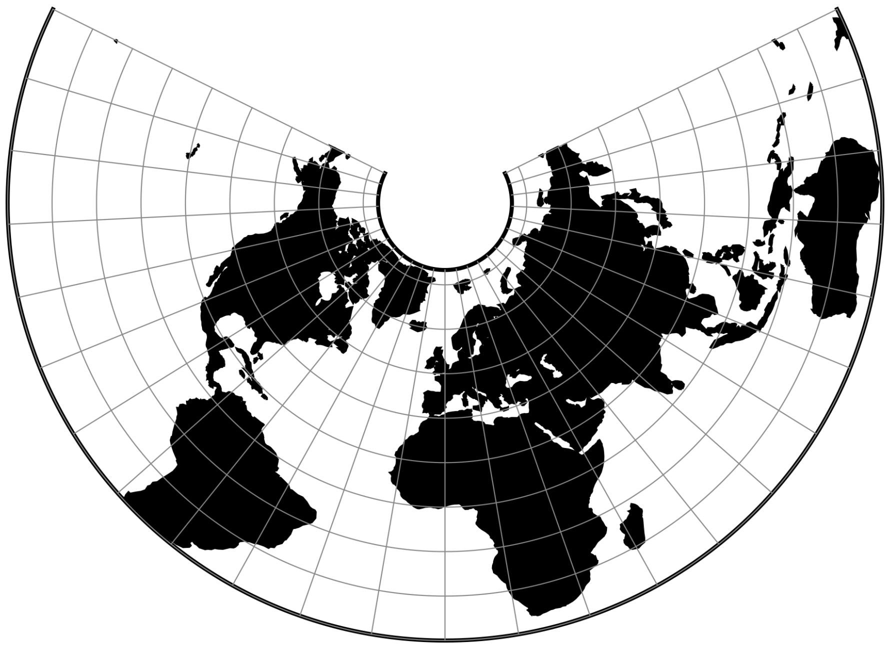


Fig. 72: proj-string: `+proj=murd3 +lat_1=30 +lat_2=50`

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

Optional

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.76 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

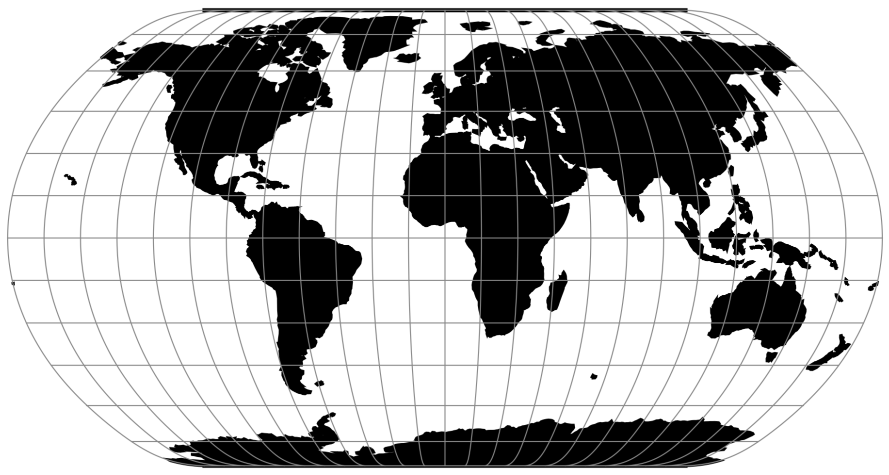


Fig. 73: proj-string: `+proj=natearth`

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.

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
```

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with +ellps +R takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

Further reading

1. Wikipedia

7.1.77 Nell

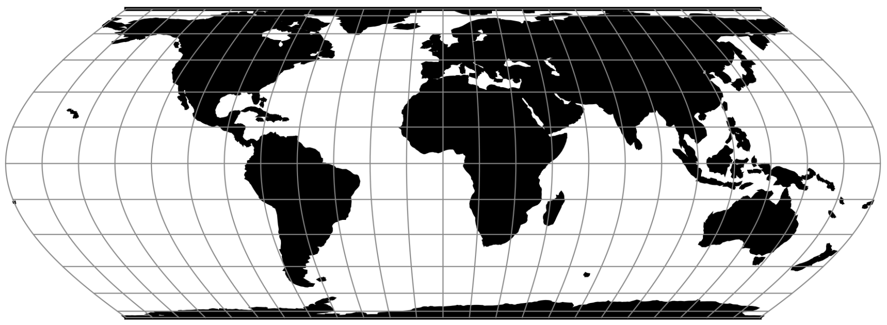


Fig. 74: proj-string: +proj=nell

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.78 Nell-Hammer

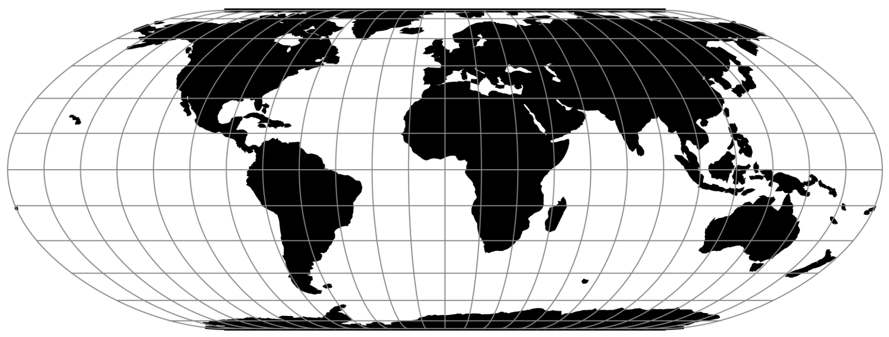


Fig. 75: proj-string: `+proj=nell_h`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.79 Nicolosi Globular

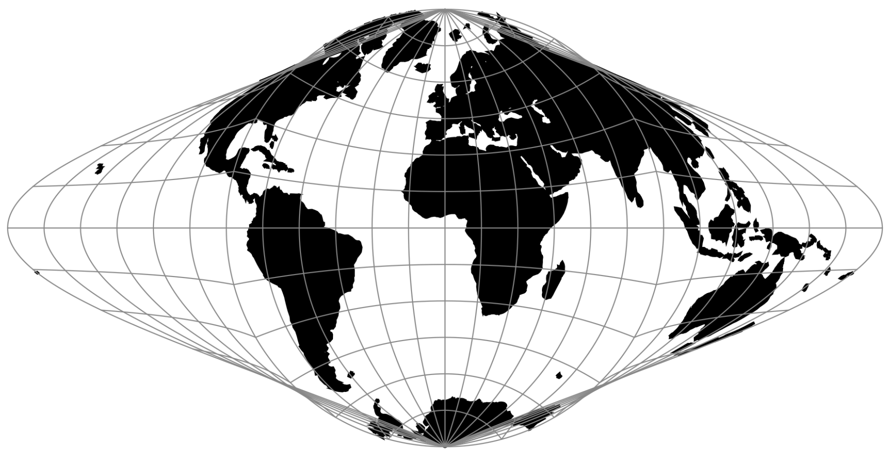


Fig. 76: proj-string: `+proj=nicol`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.80 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

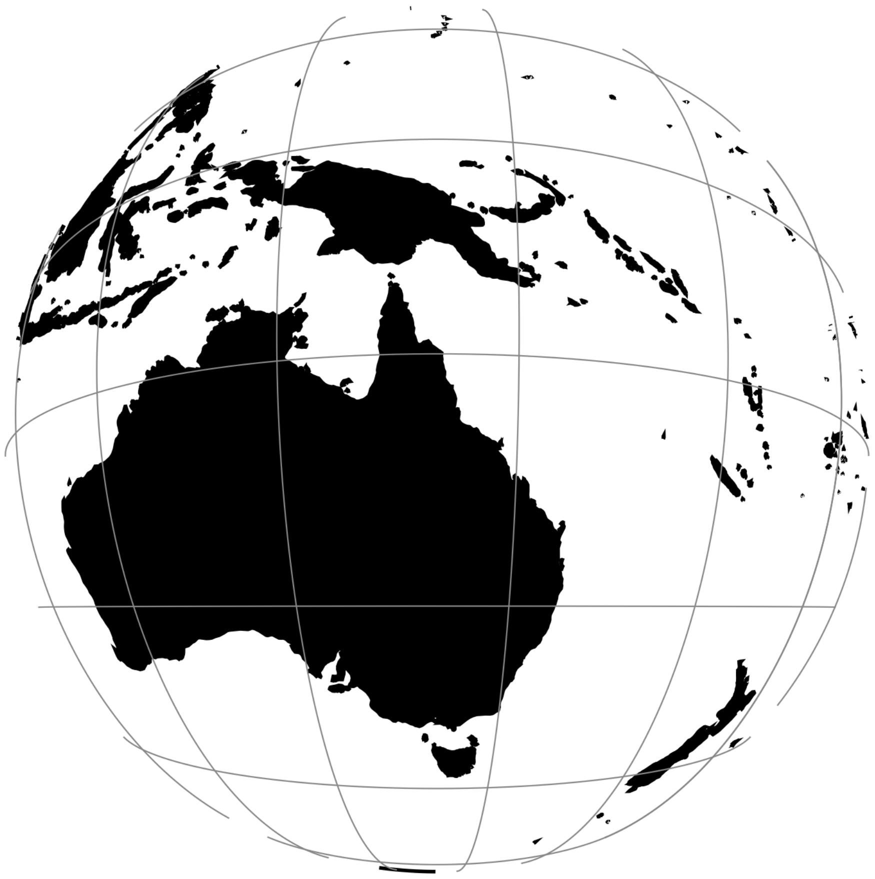


Fig. 77: proj-string: `+proj=nsper +h=3000000 +lat_0=-20 +lon_0=145`

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>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.81 New Zealand Map Grid

Parameters

Note: All standard projection parameters are hard-coded for this projection

7.1.82 General Oblique Transformation

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=obt_ran`, 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 pole of the new coordinates, α and β respectively, are to be placed at 45°N and 90°W and use the *Mollweide* projection. Because the central meridian of the translated coordinates will follow the β 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
```

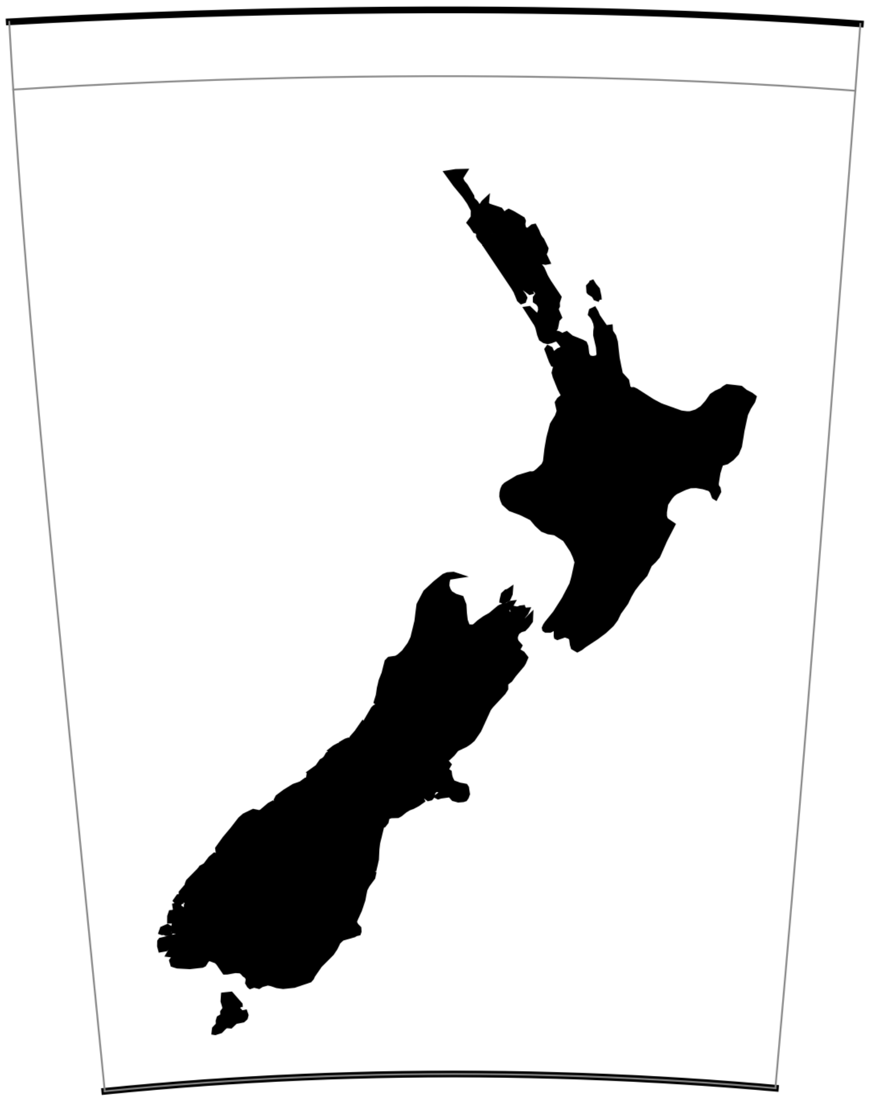


Fig. 78: proj-string: `+proj=nzmg`

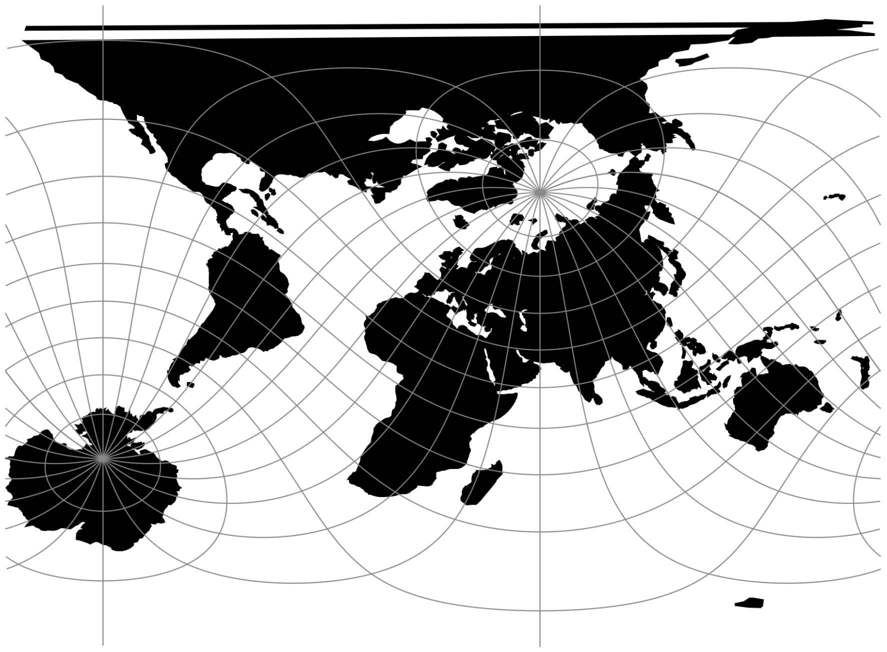


Fig. 79: proj-string: `+proj=ob_tran +o_proj=mill +o_lon_p=40 +o_lat_p=50 +lon_0=60`

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 new pole for oblique projection.

+o_lon_p=<longitude>
    Longitude of new pole for oblique projection.

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

+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>
    Longitude of projection center.

    Defaults to 0.0.

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
    False easting.

    Defaults to 0.0.

+y_0=<value>
    False northing.

    Defaults to 0.0.

7.1.83 Oblique Cylindrical Equal Area

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 to points that make up the pole.

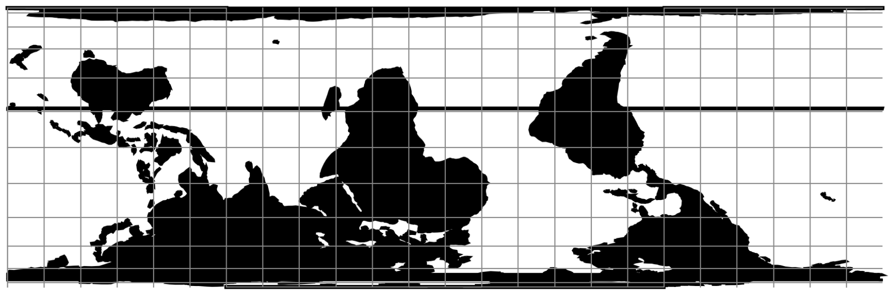


Fig. 80: proj-string: `+proj=ocea`

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>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+k_0=<value>
Scale factor. Determines scale factor used in the projection.

Defaults to 1.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.84 Oblated Equal Area

Described in [Snyder1988].

Parameters

Required

+m=<value>

+n=<value>

Optional

+theta=<value>

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.85 Oblique Mercator

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*.

+lonc=<value>
: Longitude of the central point.

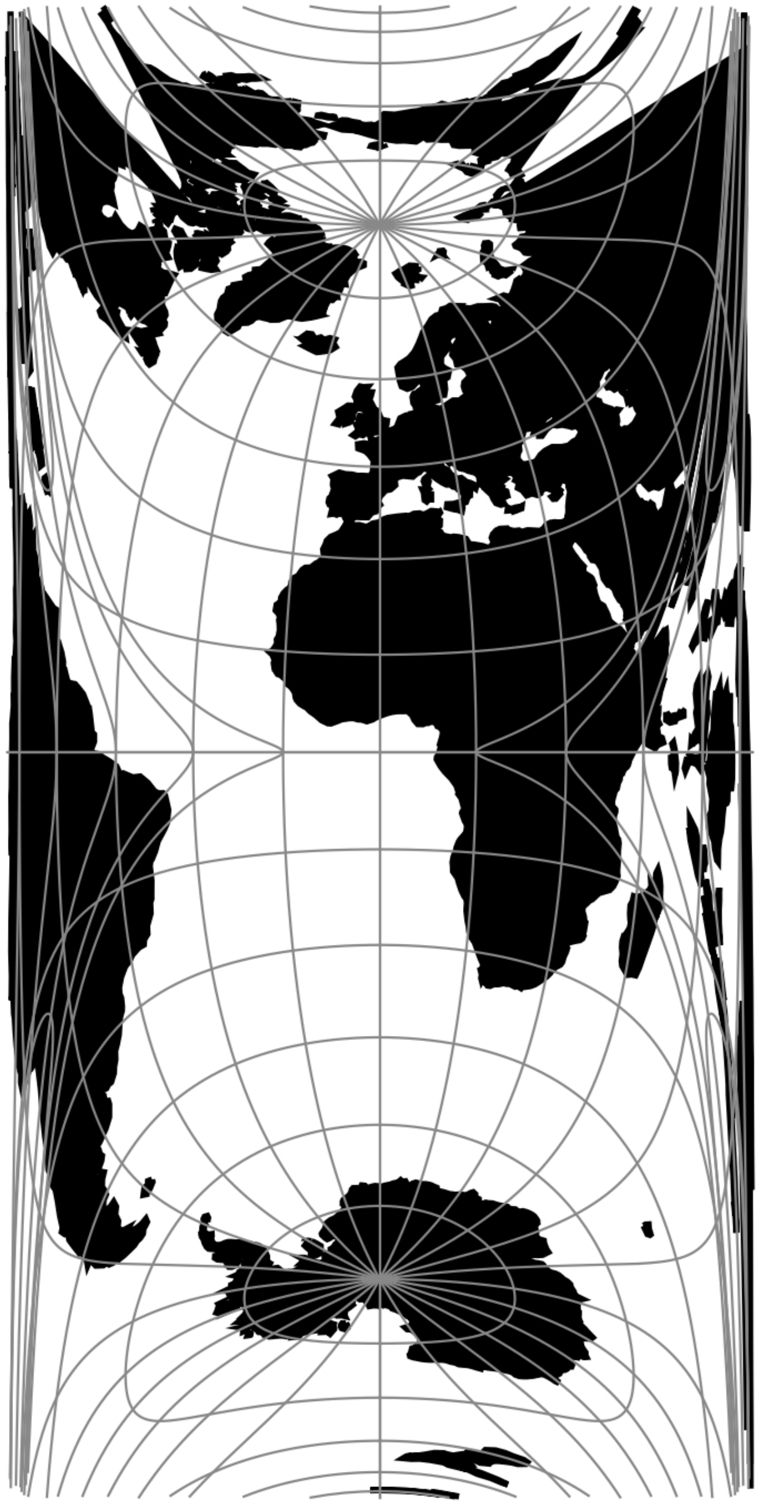


Fig. 81: proj-string: `+proj=oea +m=1 +n=2`

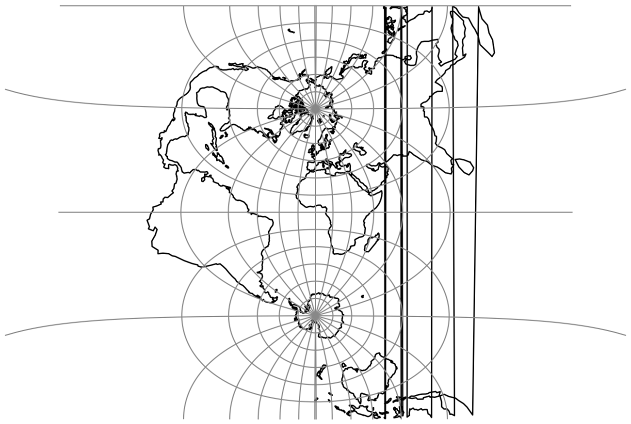


Fig. 82: proj-string: `+proj=omerc +lat_1=45 +lat_2=55`

+lat_0=<value>
Latitude of the central point.

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
Do not rotate axis.

+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.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.86 Ortelius Oval

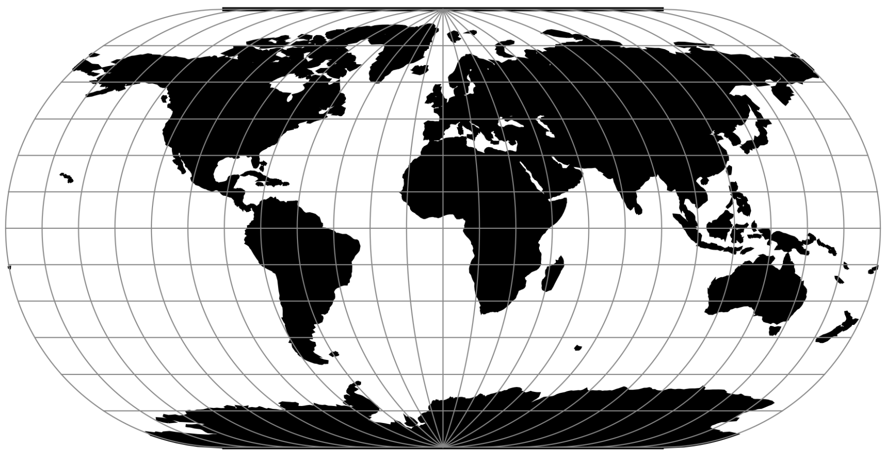


Fig. 83: proj-string: `+proj=ortel`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.87 Orthographic

The orthographic projection is a perspective azimuthal projection centered around a given latitude and longitude.

Classification	Azimuthal
Available forms	Forward and inverse, spherical projection
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

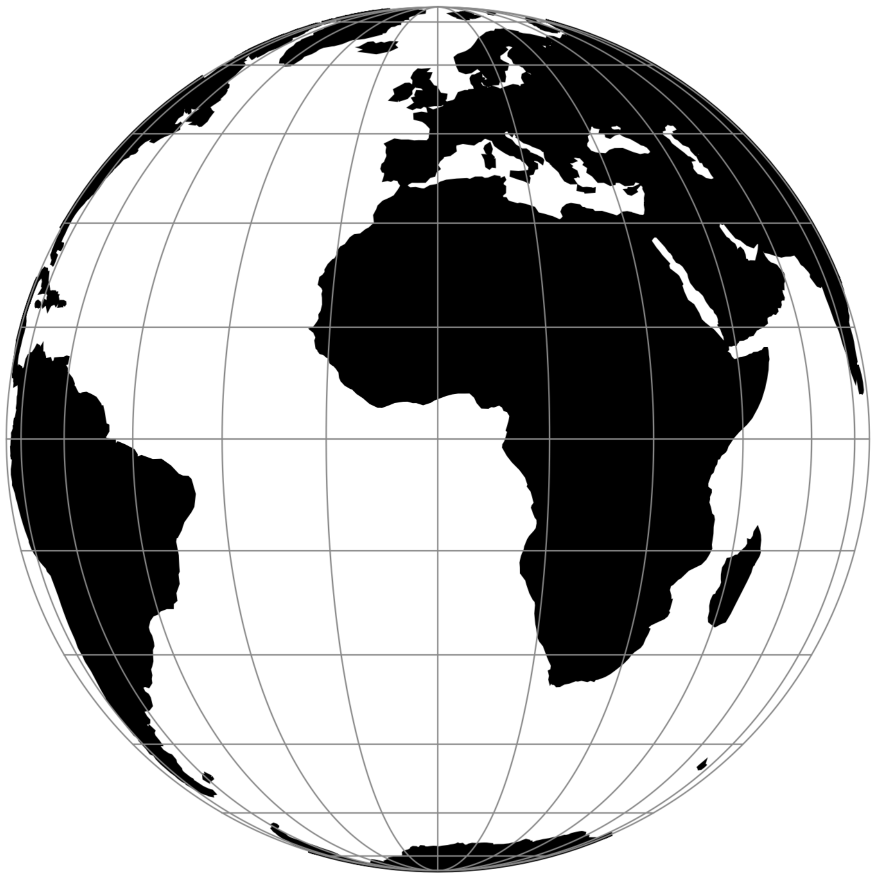


Fig. 84: proj-string: `+proj=ortho`

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.88 Perspective Conic

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

Optional

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

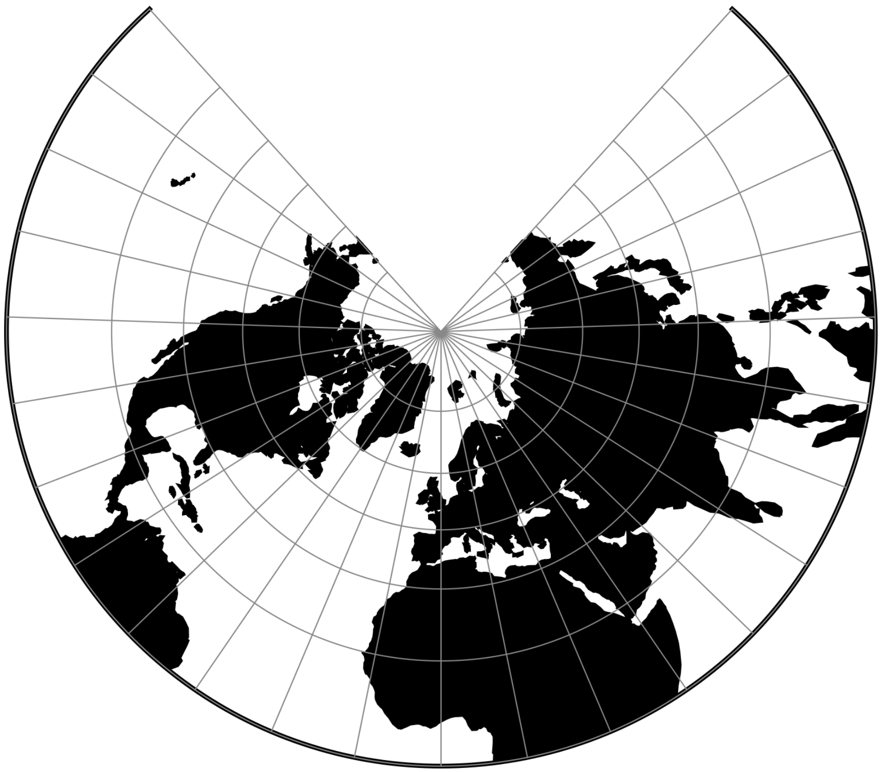


Fig. 85: proj-string: `+proj=pconic +lat_1=25 +lat_2=75`

+y_0=<value>
 False northing.

 Defaults to 0.0.

7.1.89 Polyconic (American)

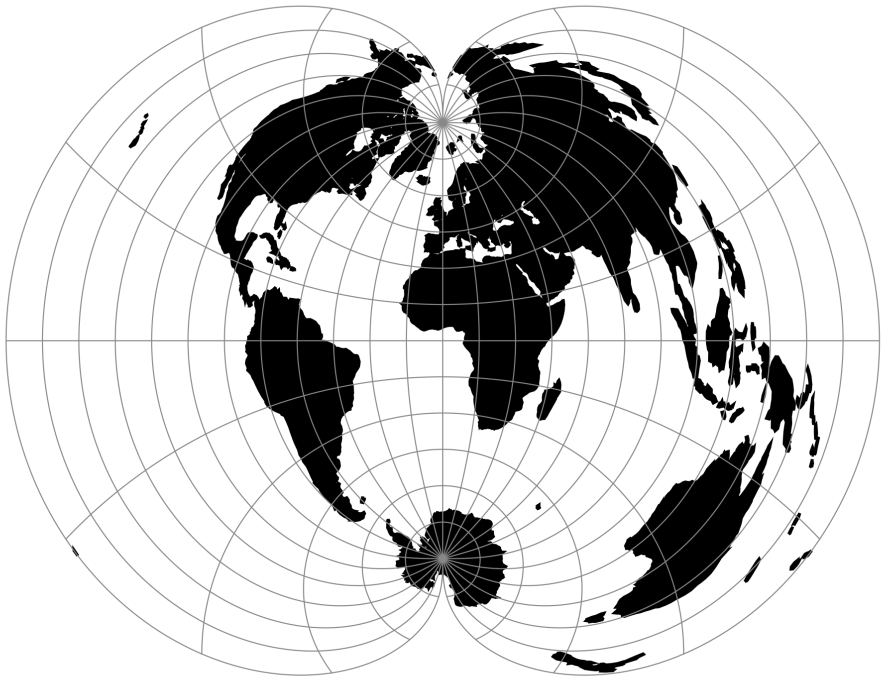


Fig. 86: proj-string: `+proj=poly`

Parameters

Note: All parameters are optional for projection.

+lon_0=<value>
 Longitude of projection center.

 Defaults to 0.0.

+ellps=<value>
 See `proj -le` for a list of available ellipsoids.

 Defaults to "WGS84".

+R=<value>
 Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
 False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.90 Putnins P1

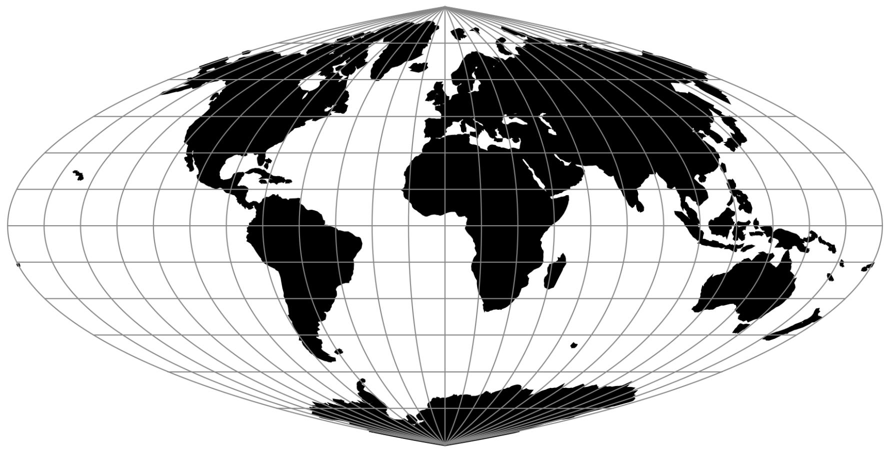


Fig. 87: proj-string: `+proj=putp1`

Parameters

Note: All parameters are optional for the Putnins P1 projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.91 Putnins P2

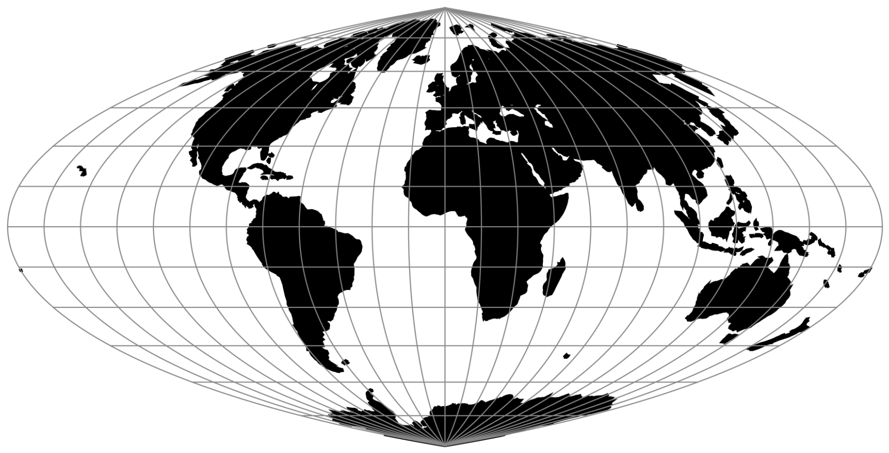


Fig. 88: proj-string: `+proj=putp2`

Parameters

Note: All parameters are optional for the projection.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.92 Putnins P3

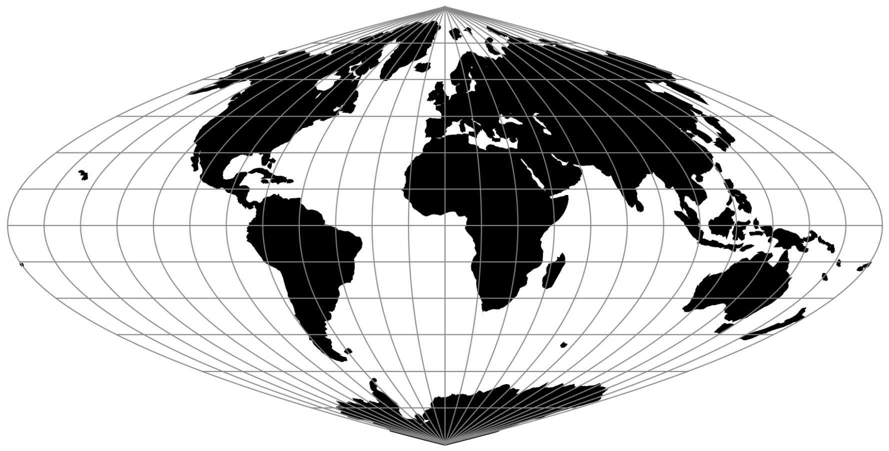


Fig. 89: proj-string: `+proj=putp3`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
    Longitude of projection center.

    Defaults to 0.0.

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
    False easting.

    Defaults to 0.0.

+y_0=<value>
    False northing.

    Defaults to 0.0.

7.1.93 Putnins P3'

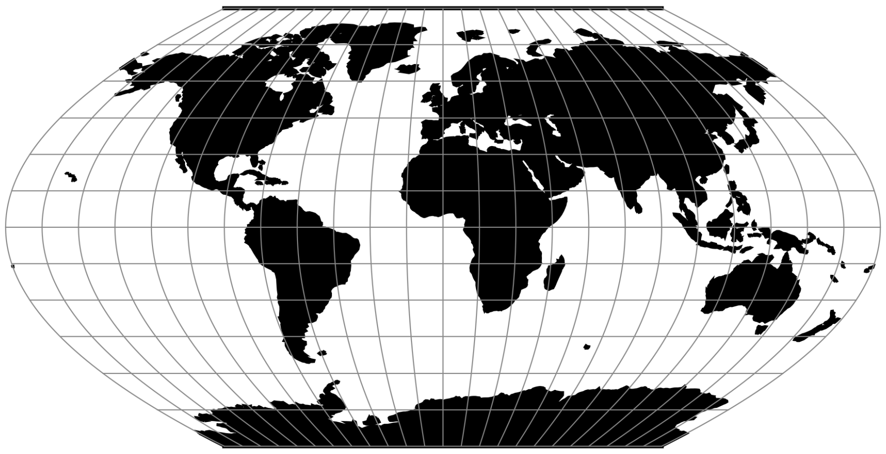


Fig. 90: proj-string: `+proj=putp3p`

Parameters

Note: All parameters are optional for the projection.

`+lon_0`=`<value>`
Longitude of projection center.

Defaults to 0.0.

`+R`=`<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0`=`<value>`
False easting.

Defaults to 0.0.

`+y_0`=`<value>`
False northing.

Defaults to 0.0.

7.1.94 Putnins P4’

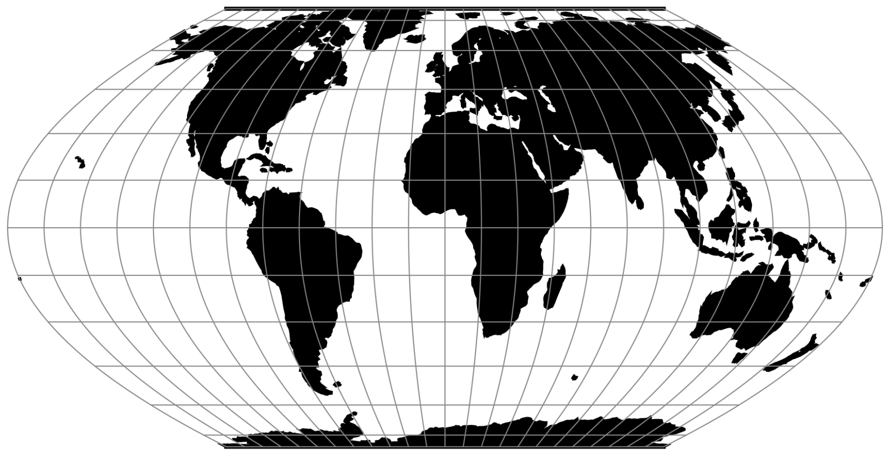


Fig. 91: proj-string: `+proj=putp4p`

Parameters

Note: All parameters are optional for the projection.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.95 Putnins P5

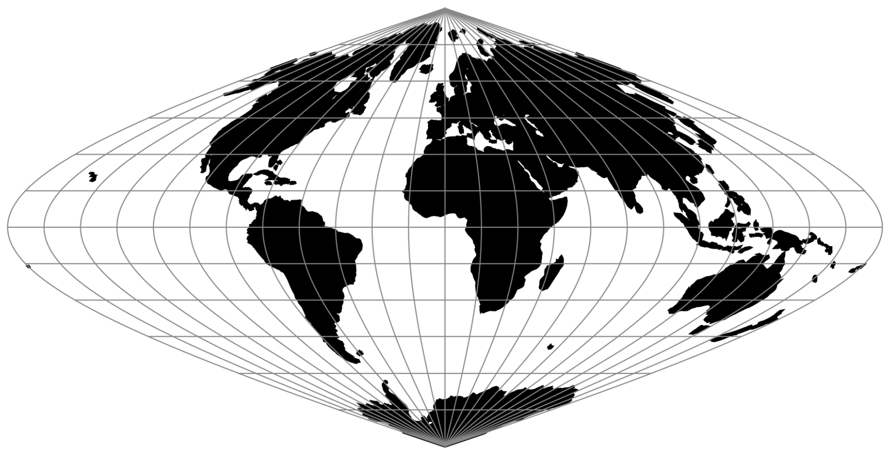


Fig. 92: proj-string: `+proj=putp5`

Parameters

Note: All parameters are optional for the projection.

`+lon_0``=<value>`

Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`

Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`

False easting.

Defaults to 0.0.

`+y_0``=<value>`

False northing.

Defaults to 0.0.

7.1.96 Putnins P5'

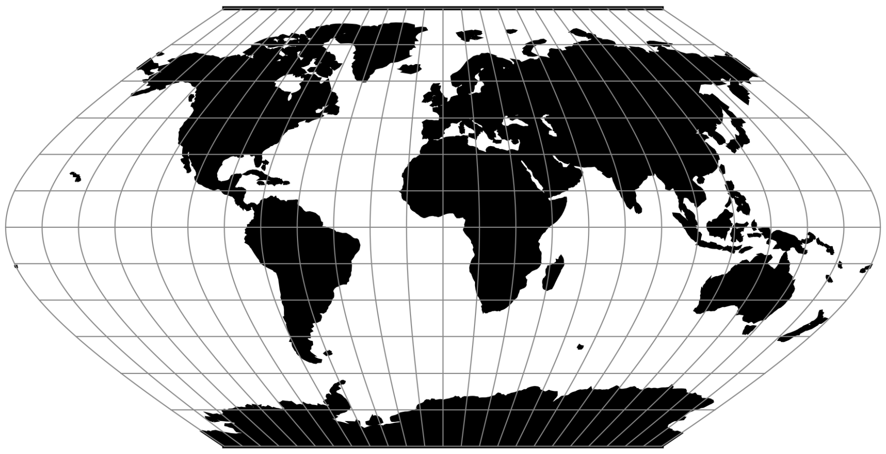


Fig. 93: proj-string: `+proj=putp5p`

Parameters

Note: All parameters are optional for the projection.

`+lon_0``=<value>`
  Longitude of projection center.

  Defaults to 0.0.

`+R``=<value>`
  Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
  False easting.

  Defaults to 0.0.

`+y_0``=<value>`
  False northing.

  Defaults to 0.0.

7.1.97 Putnins P6

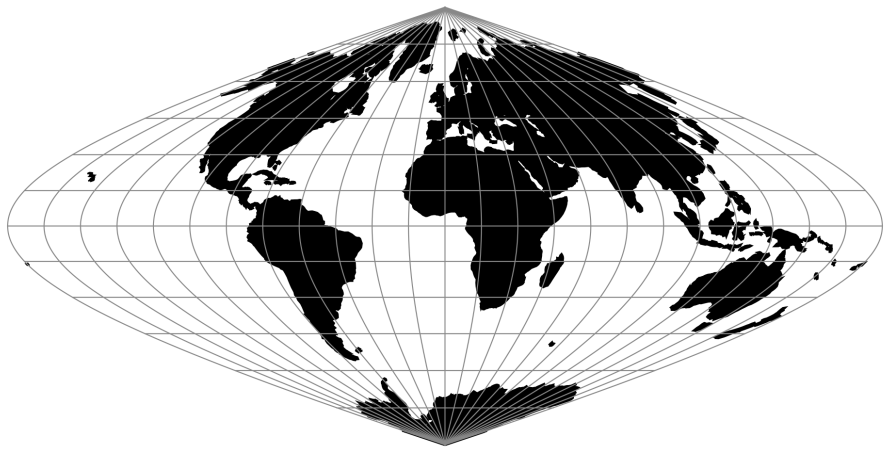


Fig. 94: proj-string: `+proj=putp6`

Parameters

Note: All parameters are optional for the projection.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.98 Putnins P6'

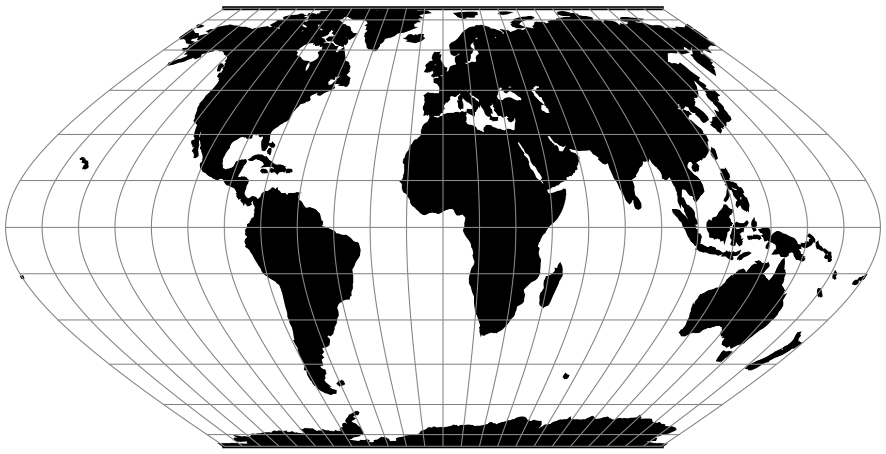


Fig. 95: proj-string: `+proj=putp6p`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
    Longitude of projection center.

    Defaults to 0.0.

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
    False easting.

    Defaults to 0.0.

+y_0=<value>
    False northing.

    Defaults to 0.0.

7.1.99 Quartic Authalic

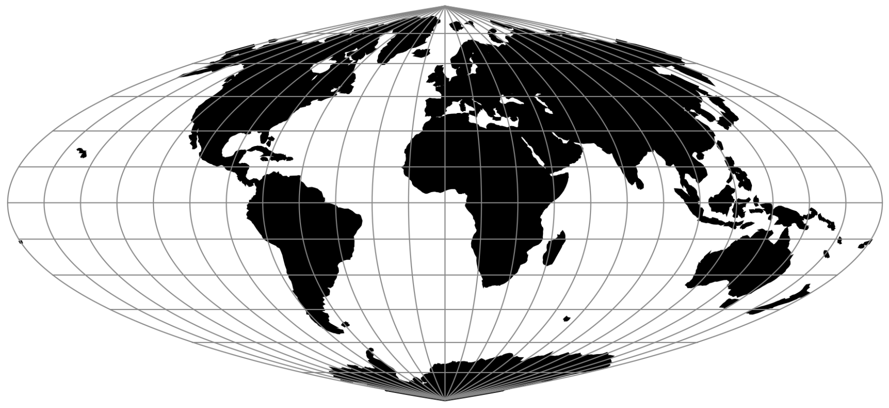


Fig. 96: proj-string: `+proj=qua_aut`

Parameters

Note: All parameters are optional for the Quartic Authalic projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
: False easting.

  Defaults to 0.0.

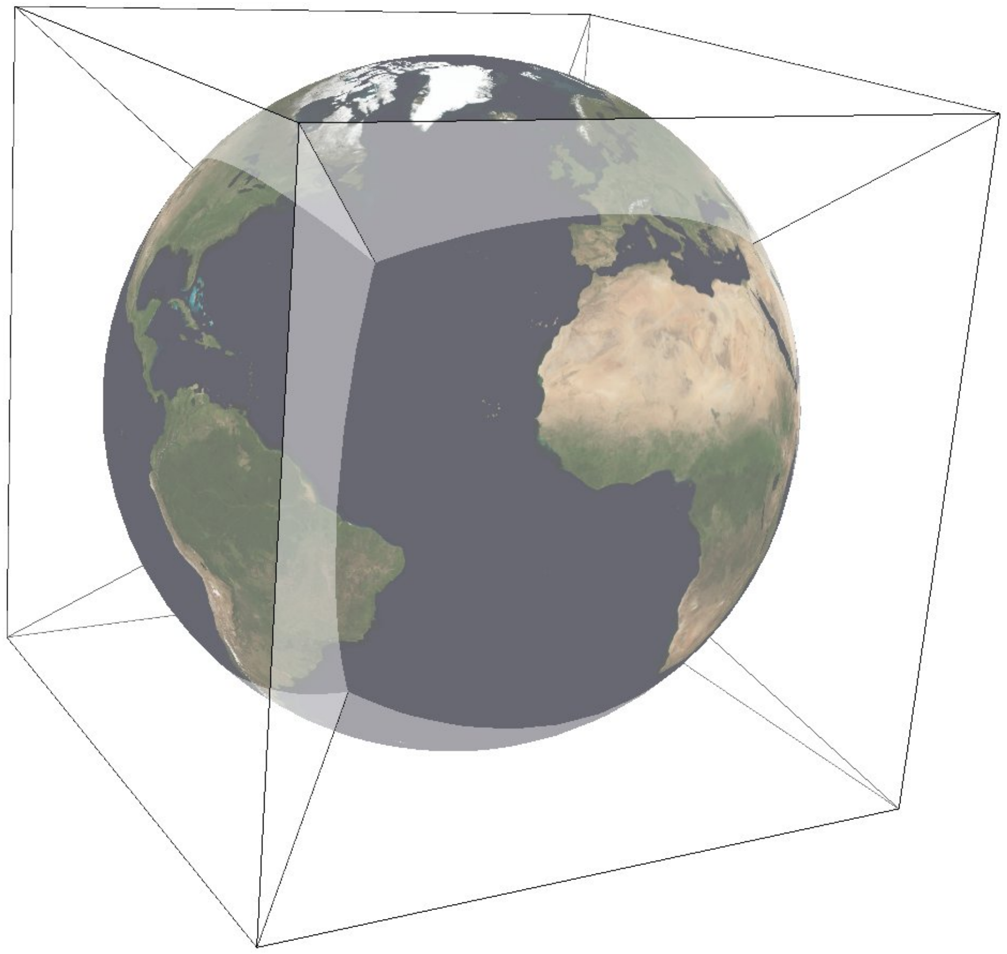
+y_0=<value>
: False northing.

  Defaults to 0.0.

7.1.100 Quadrilateralized Spherical Cube

Classification	Azimuthal
Available forms	Forward and inverse, elliptical projection
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.

Usage

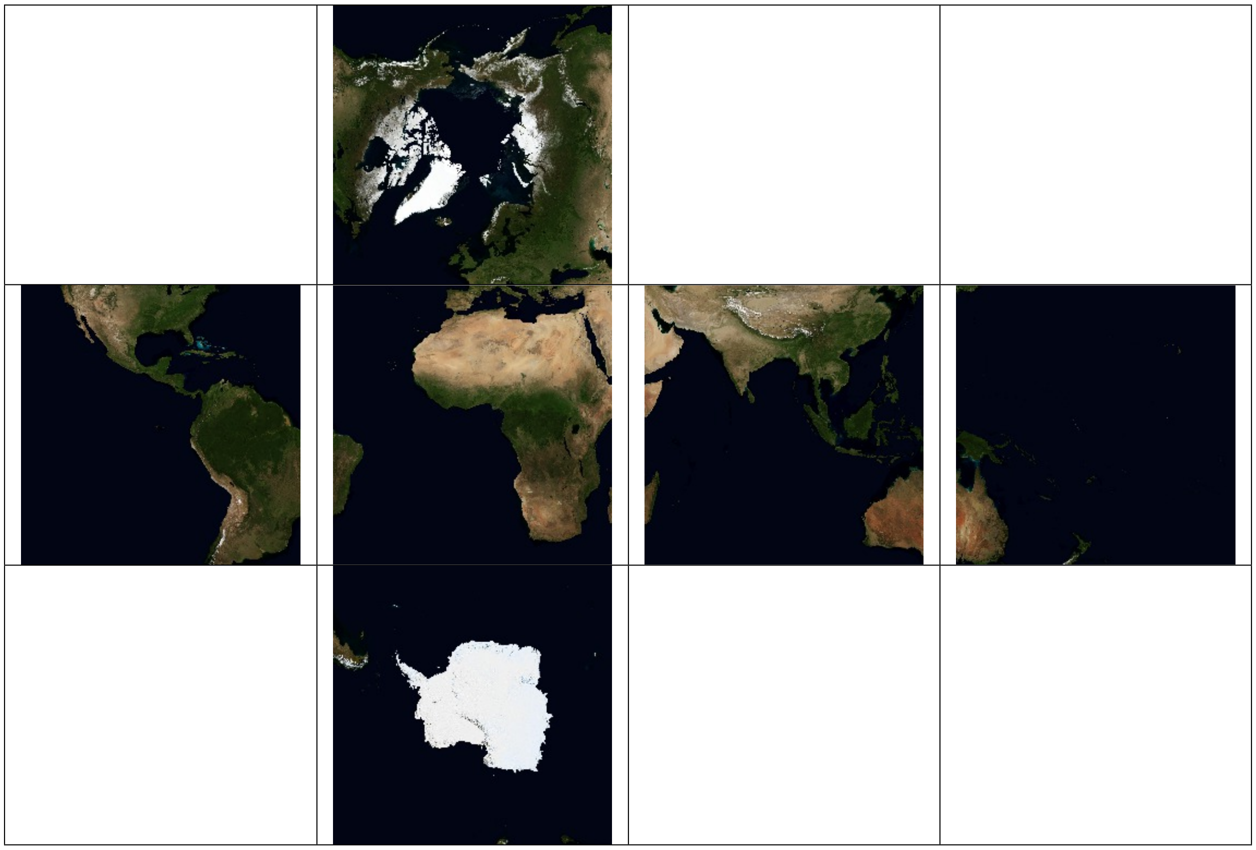
The following example uses QSC projection via GDAL to create the six cube side maps from a world map for the WGS84 ellipsoid:

```
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.



Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

Further reading

1. Wikipedia
2. NASA

7.1.101 Robinson

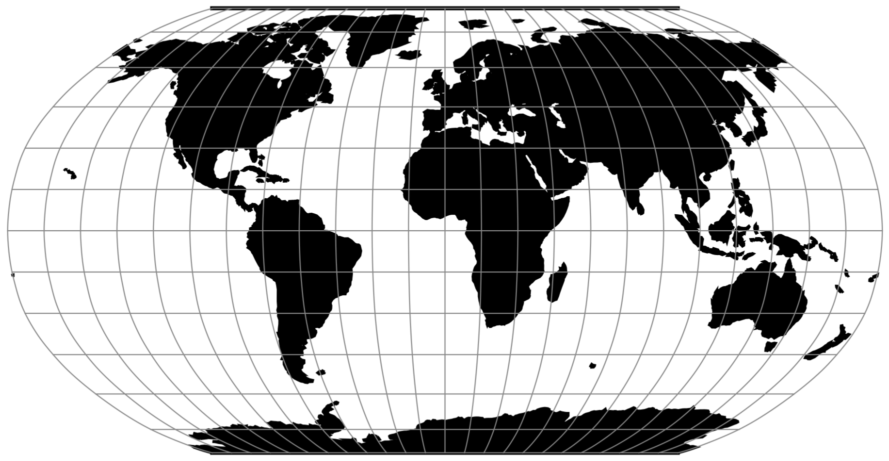


Fig. 97: proj-string: `+proj=robin`

Parameters

Note: All parameters are optional for the projection.

`+lon_0``=<value>`
  Longitude of projection center.

  Defaults to 0.0.

`+R``=<value>`
  Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
  False easting.

  Defaults to 0.0.

`+y_0``=<value>`
  False northing.

  Defaults to 0.0.

7.1.102 Roussilhe Stereographic

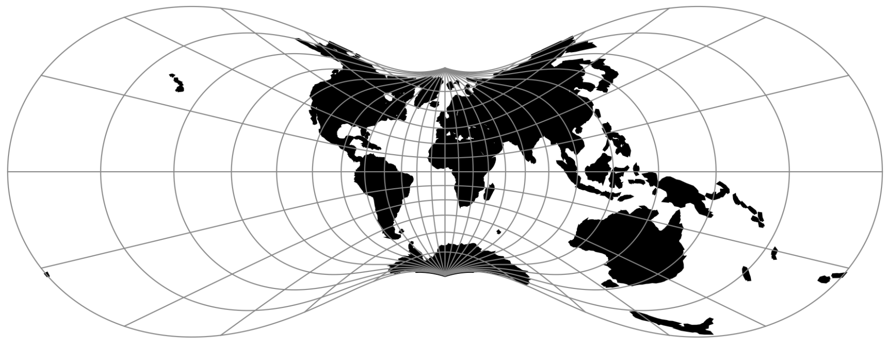


Fig. 98: proj-string: `+proj=rouss`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.103 Rectangular Polyconic

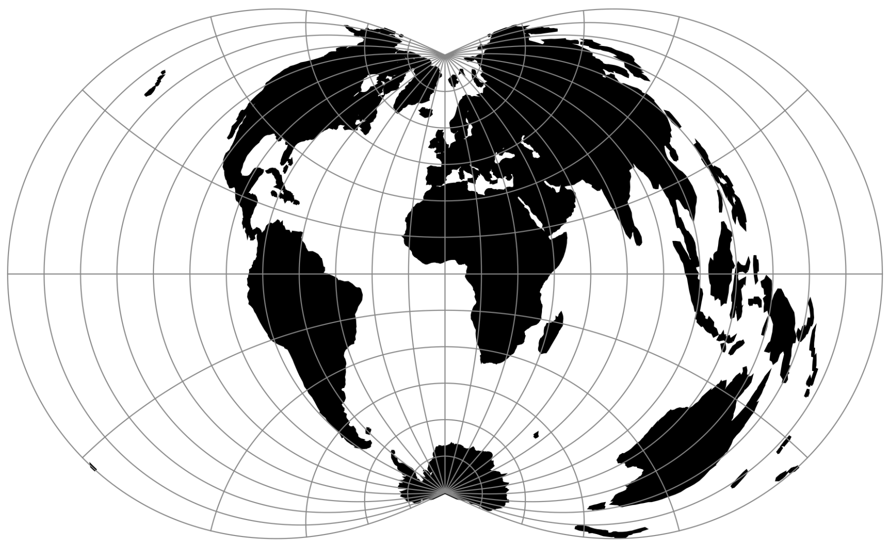


Fig. 99: proj-string: `+proj=rpoly`

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.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.104 Sinusoidal (Sanson-Flamsteed)

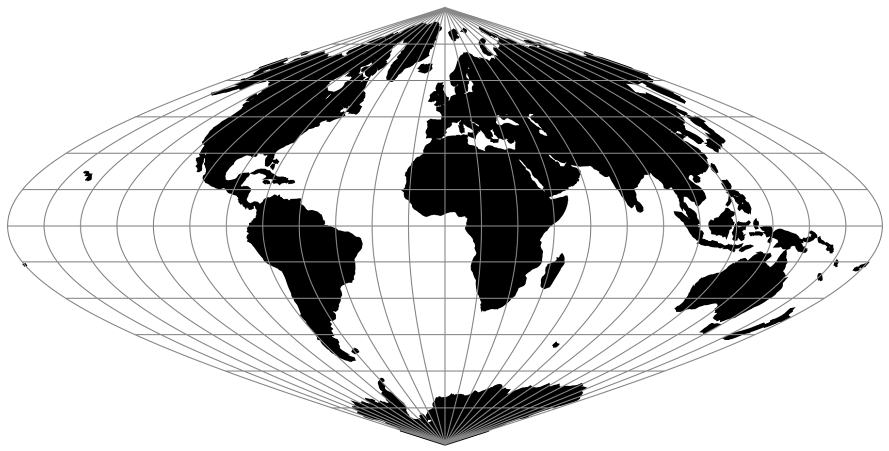


Fig. 100: proj-string: `+proj=sinu`

MacBryde and Thomas developed generalized formulas for several of the pseudocylindricals with sinusoidal meridians:

$$x = C\lambda(m + cos\theta)/(m + 1)$$

$$y = C\theta$$

$$C = \sqrt{(m + 1)/n}$$

Parameters

Note: All parameters are optional for the Sinusoidal projection.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.105 Swiss Oblique Mercator

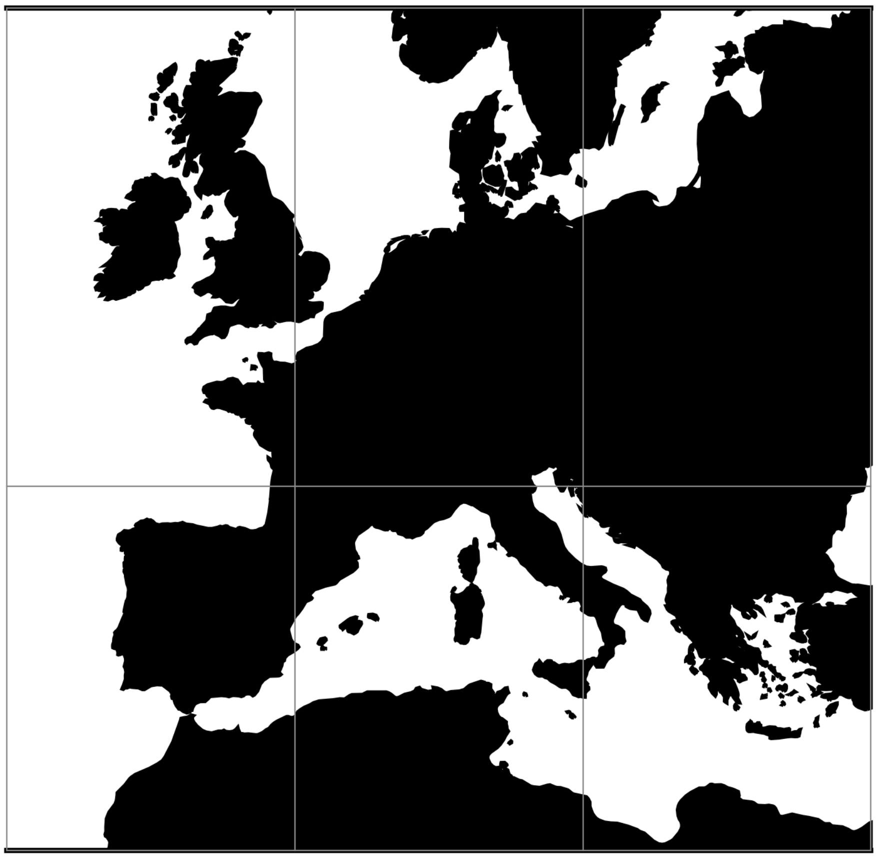


Fig. 101: proj-string: `+proj=somerc`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
: Longitude of projection center.

  Defaults to 0.0.

+ellps=<value>
: See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
: Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+k_0=<value>
Scale factor. Determines scale factor used in the projection.

Defaults to 1.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.106 Stereographic

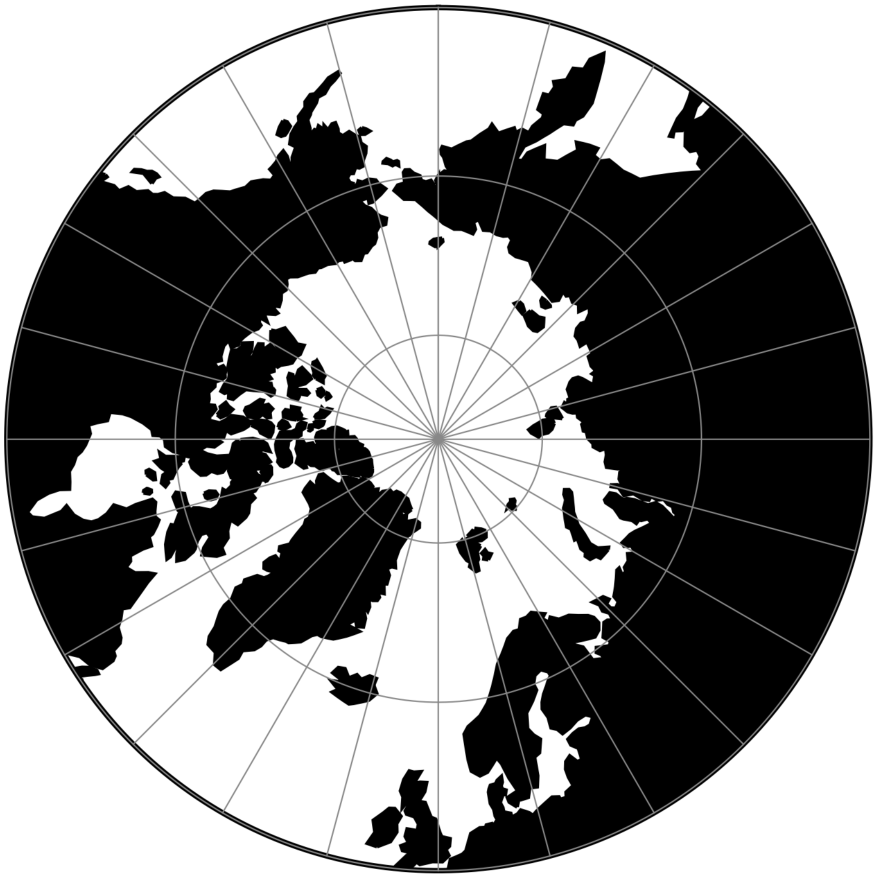


Fig. 102: proj-string: `+proj=stere +lat_0=90 +lat_ts=75`

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.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.107 Oblique Stereographic Alternative

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

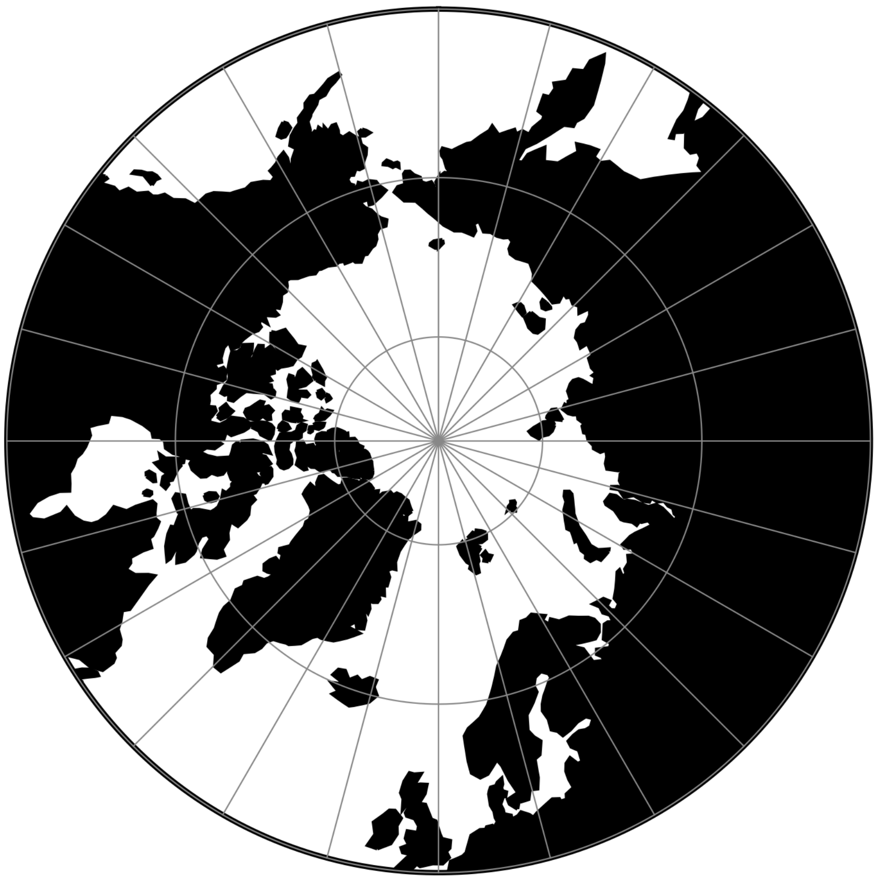


Fig. 103: proj-string: `+proj=sterea +lat_0=90`

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.108 Gauss-Schreiber Transverse Mercator (aka Gauss-Laborde Reunion)

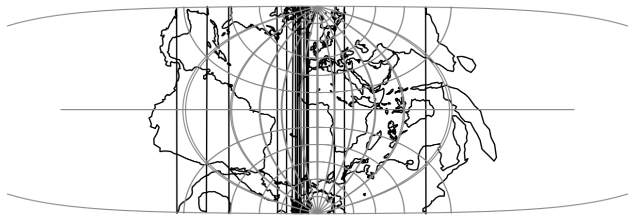


Fig. 104: proj-string: `+proj=gstmerc`

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>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.109 Transverse Central Cylindrical

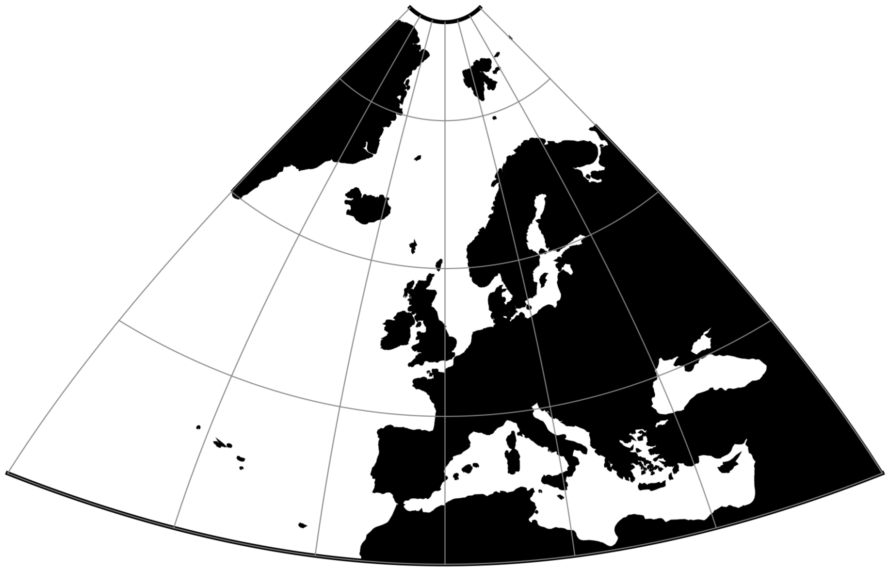


Fig. 105: proj-string: `+proj=tcc`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.110 Transverse Cylindrical Equal Area

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+k_0=<value>
Scale factor. Determines scale factor used in the projection.

Defaults to 1.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.111 Tissot

Parameters

Required

+lat_1=<value>
First standard parallel.

Defaults to 0.0.

+lat_2=<value>
Second standard parallel.

Defaults to 0.0.

Optional

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

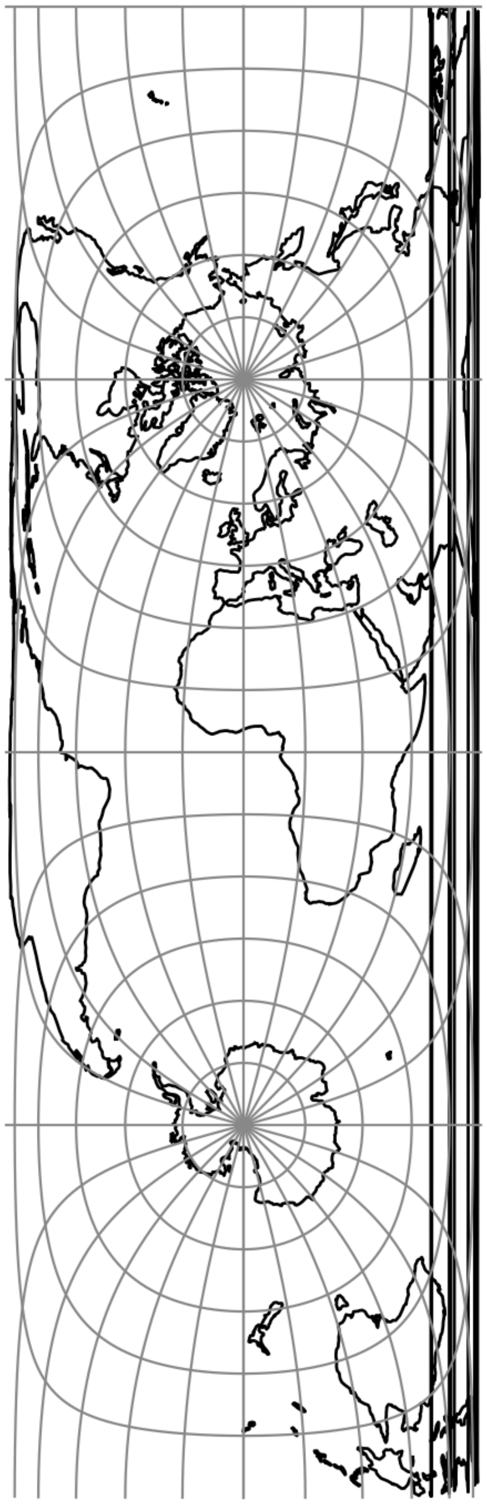


Fig. 106: proj-string: `+proj=tcea`

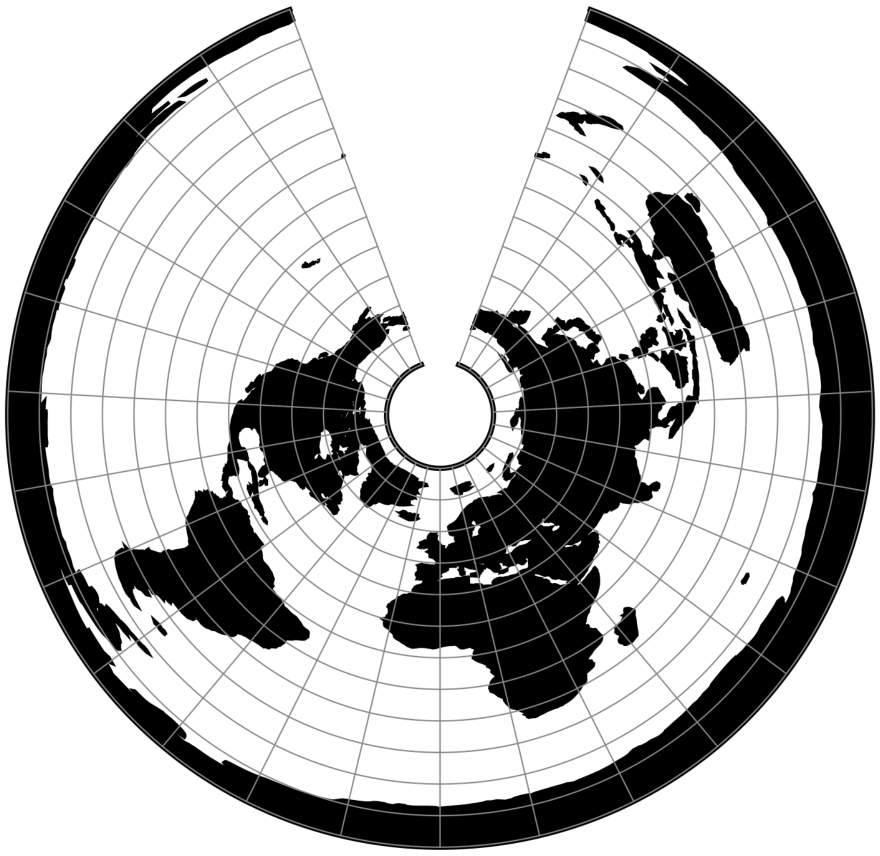


Fig. 107: proj-string: `+proj=tissot +lat_1=60 +lat_2=65`

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.112 Transverse Mercator

The transverse Mercator projection in its various forms is the most widely used projected coordinate system for world topographical and offshore mapping.

Classification	Transverse and oblique cylindrical
Available forms	Forward and inverse, Spherical and Elliptical
Defined area	Global, but reasonably accurate only within 15 degrees of the central meridian
Alias	tmerc
Domain	2D
Input type	Geodetic coordinates
Output type	Projected coordinates

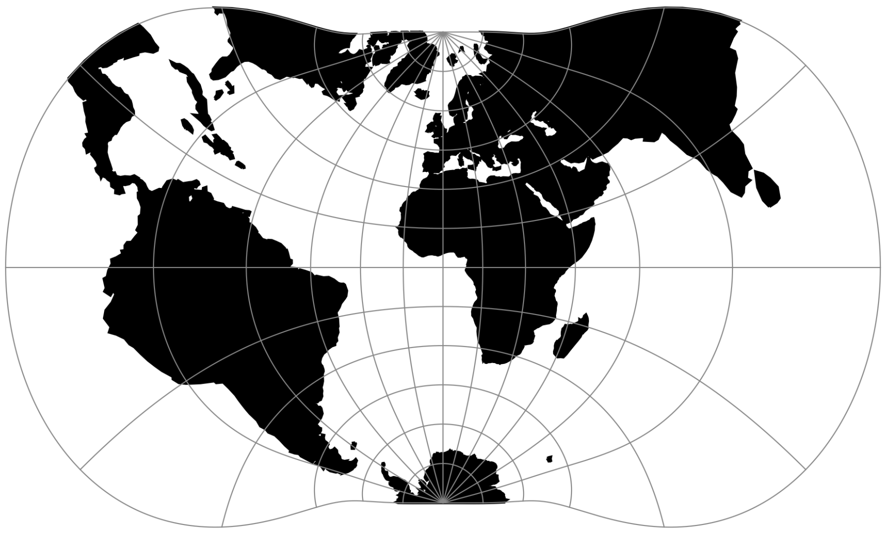


Fig. 108: proj-string: `+proj=tmerc`

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 6°	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	Always 6°	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 +lat_0=0 +lon_0=9 +k_0=1 +x_0=3500000 +y_0=0␣
↪+ellps=bessel +datum=potsdam +units=m +no_defs
3500000.00  5651505.56
```

Example using Gauss Boaga on Italy area (EPSG:3004)

```
$ echo 15 42 | proj +proj=tmerc +lat_0=0 +lon_0=15 +k_0=0.9996 +x_0=2520000 +y_0=0␣
↪+ellps=intl +units=m +no_defs
2520000.00  4649858.60
```

Parameters

Note: All parameters for the projection are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+k_0=<value>
Scale factor. Determines scale factor used in the projection.

Defaults to 1.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

Mathematical definition

The formulas describing the Transverse Mercator are all taken from [Evenden2005].

ϕ_0 is the latitude of origin that match the center of the map. It can be set with `+lat_0`.

k_0 is the scale factor at the natural origin (on the central meridian). It can be set with `+k_0`.

$M(\phi)$ is the meridional distance.

Spherical form

Forward projection

$$B = \cos\phi \sin\lambda$$

$$x = \frac{k_0}{2} \ln(\frac{1+B}{1-B})$$

$$y = k_0(\arctan(\frac{\tan(\phi)}{\cos\lambda}) - \phi_0)$$

Inverse projection

$$D = \frac{y}{k_0} + \phi_0$$

$$x' = \frac{x}{k_0}$$

$$\phi = \arcsin(\frac{\sin D}{\cosh x'})$$

$$\lambda = \arctan(\frac{\sinh x'}{\cos D})$$

Elliptical form

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

$$\phi_1 = M^-1(y)$$

$$N_1 = \frac{k_0}{1 - e^2 \sin^2 \phi_1)^{1/2}}$$

$$R_1 = \frac{k_0(1 - e^2)}{(1 - e^2 \sin^2 \phi_1)^{3/2}}$$

$$t_1 = \tan(\phi_1)$$

$$\eta_1 = \frac{e^2}{1 - e^2} cos^2 \phi_1$$

$$\begin{aligned}
\phi = \phi_1 \\
&- \frac{t_1 x^2}{2! R_1 N_1} \\
&+ \frac{t_1 x^4}{4! R_1 N_1^3}(5 + 3t_1^2 + \eta_1^2 - 4\eta_1^4 - 9\eta_1^2 t_1^2) \\
&- \frac{t_1 x^6}{6! R_1 N_1^5}(61 + 90t_1^2 + 46\eta_1^2 + 45t_1^4 - 252t_1^2\eta_1^2) \\
&+ \frac{t_1 x^8}{8! R_1 N_1^7}(1385 + 3633t_1^2 + 4095t_1^4 + 1575t_1^6)
\end{aligned}$$

$$\begin{aligned}
\lambda = \frac{x}{\cos\phi N_1} \\
&- \frac{x^3}{3! \cos\phi N_1^3}(1 + 2t_1^2 + \eta_1^2) \\
&+ \frac{x^5}{5! \cos\phi N_1^5}(5 + 6\eta_1^2 + 28t_1^2 - 3\eta_1^2 + 8t_1^2\eta_1^2) \\
&- \frac{x^7}{7! \cos\phi N_1^7}(61 + 662t_1^2 + 1320t_1^4 + 720t_1^6)
\end{aligned}$$

Further reading

1. Wikipedia
2. EPSG, POSC literature pertaining to Coordinate Conversions and Transformations including Formulas

7.1.113 Two Point Equidistant

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.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

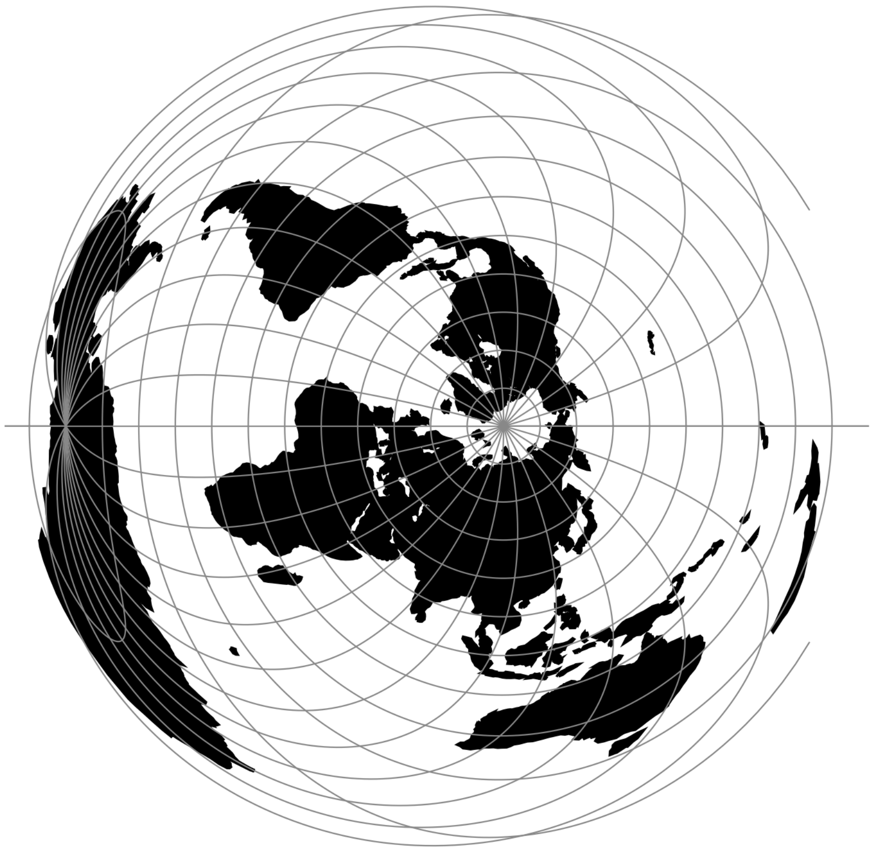


Fig. 109: proj-string: `+proj=tpeqd +lat_1=60 +lat_2=65`

Defaults to 0.0.

7.1.114 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

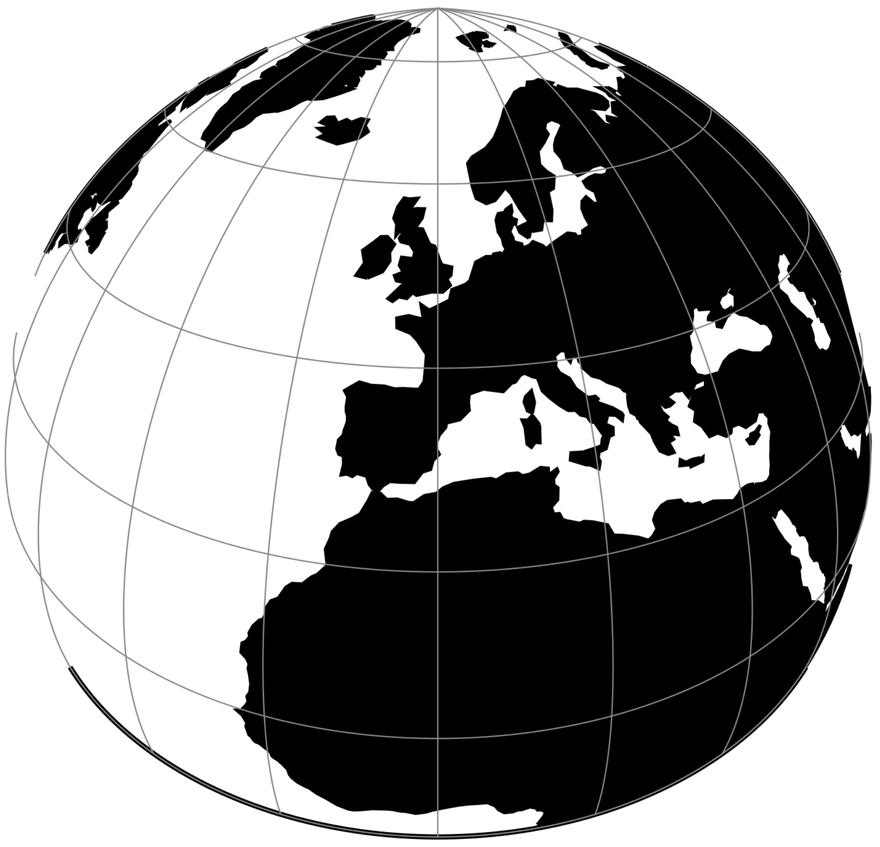


Fig. 110: proj-string: `+proj=tpers +h=5500000 +lat_0=40`

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.

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

+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>
Longitude of projection center.

Defaults to 0.0.

+lat_0=<value>
Latitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.115 Universal Polar Stereographic

Parameters

Note: All parameters are optional for the projection.

+south
South polar aspect.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

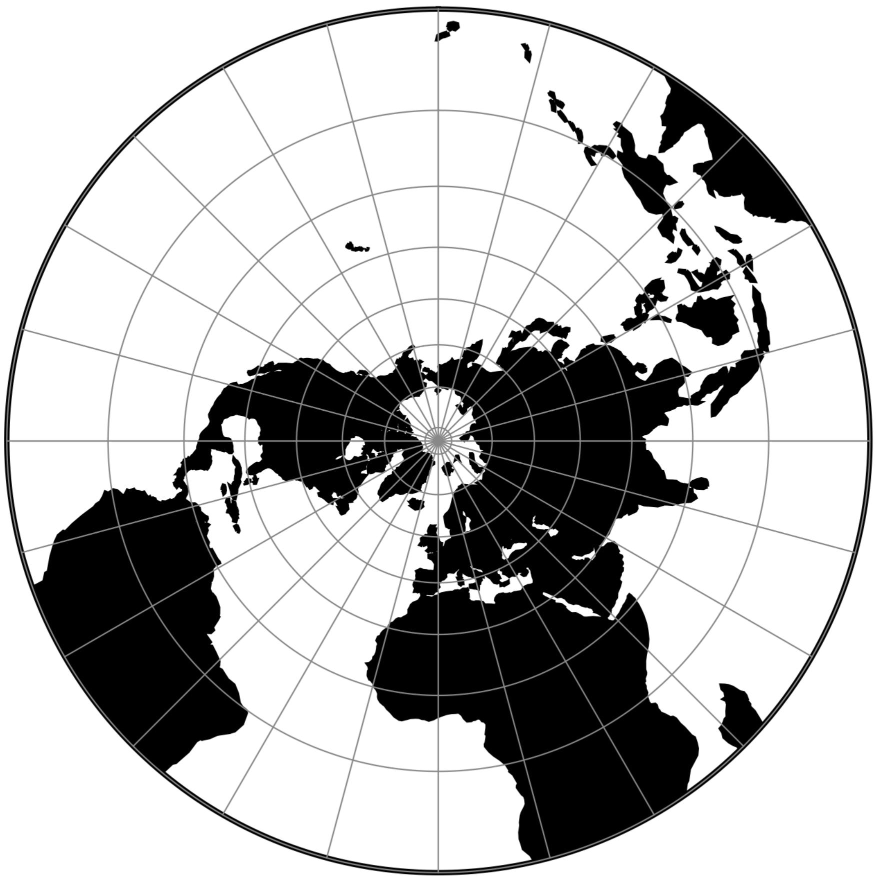


Fig. 111: proj-string: `+proj=ups`

+ellps=<value>
  See `proj -le` for a list of available ellipsoids.

  Defaults to "WGS84".

+R=<value>
  Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
  False easting.

  Defaults to 0.0.

+y_0=<value>
  False northing.

  Defaults to 0.0.

7.1.116 Urmaev V

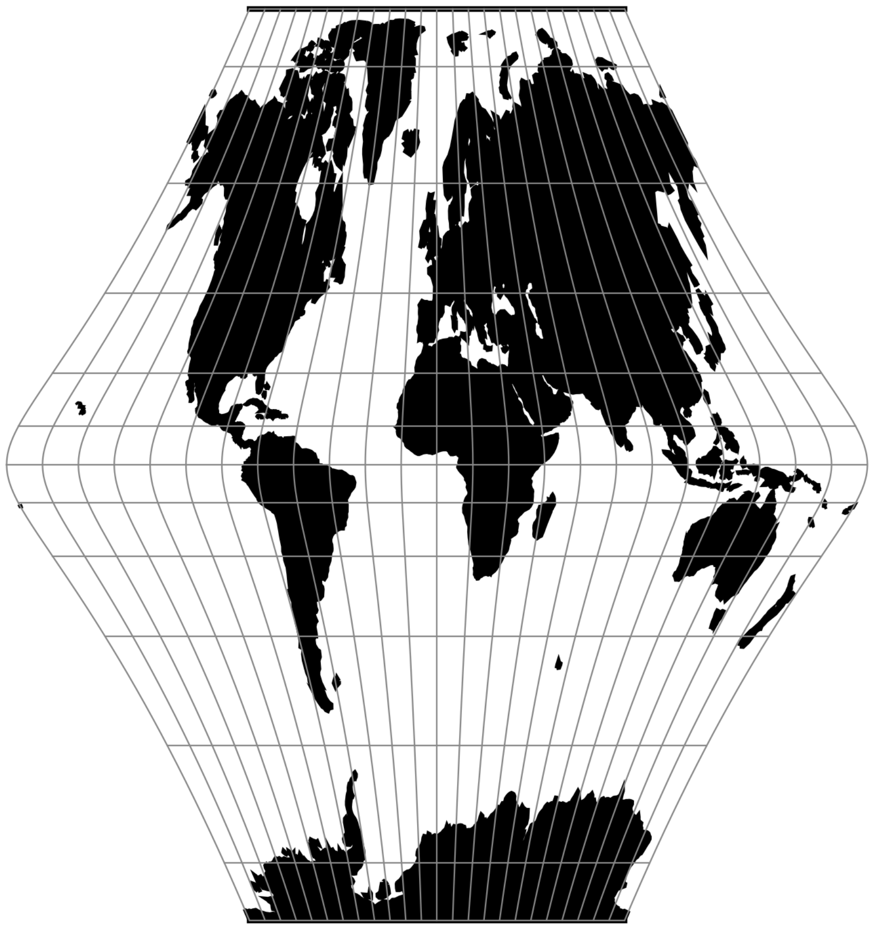


Fig. 112: proj-string: `+proj=urm5 +n=0.9 +alpha=2 +q=4`

Parameters

Note: All parameters are optional for the projection.

+n=<value>
Set the n constant. Value between 0 and 1.

+q=<value>
Set the q constant.

+alpha=<value>
Set the α constant.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.117 Urmaev Flat-Polar Sinusoidal

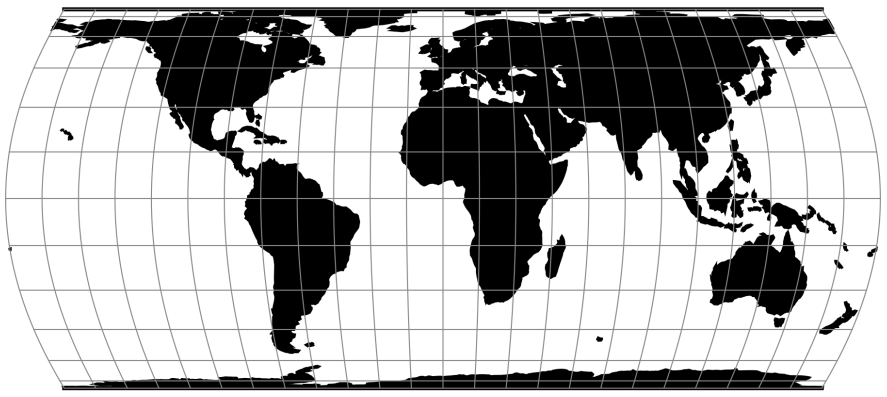


Fig. 113: proj-string: `+proj=urmfps +n=0.5`

Parameters

Note: All parameters are optional for the projection.

+n=<value>
Set the n constant. Value between 0 and 1.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.118 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 longigude.

Classification	Transverse cylindrical, conformal
Available forms	Forward and inverse, Spherical and Elliptical
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

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].

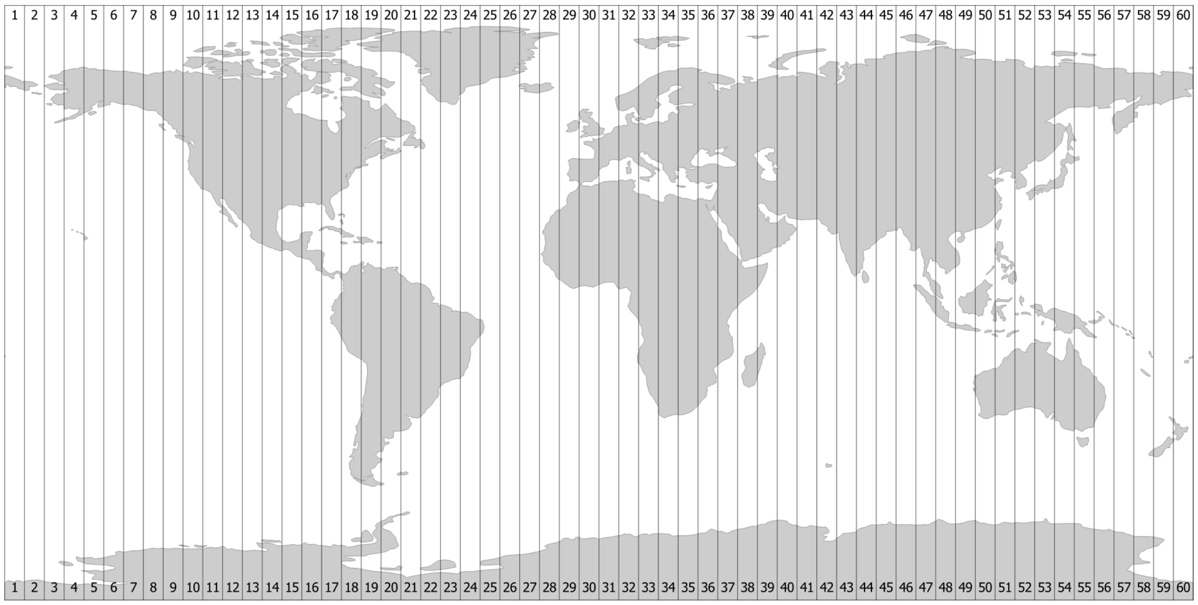


Fig. 114: UTM zones.

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 souther hemisphere:

```
$ echo 174 -44 | proj +proj=utm +zone=59 +south
740526.32       5123750.87
```

Parameters

Required

+zone=<value>
    Select which UTM zone to use. Can be a value between 1-60.

+south
    Add this flag when using the UTM on the southern hemisphere.

Optional

+ellps=<value>
See *proj -le* for a list of available ellipsoids.

Defaults to "WGS84".

Further reading

1. Wikipedia

7.1.119 van der Grinten (I)

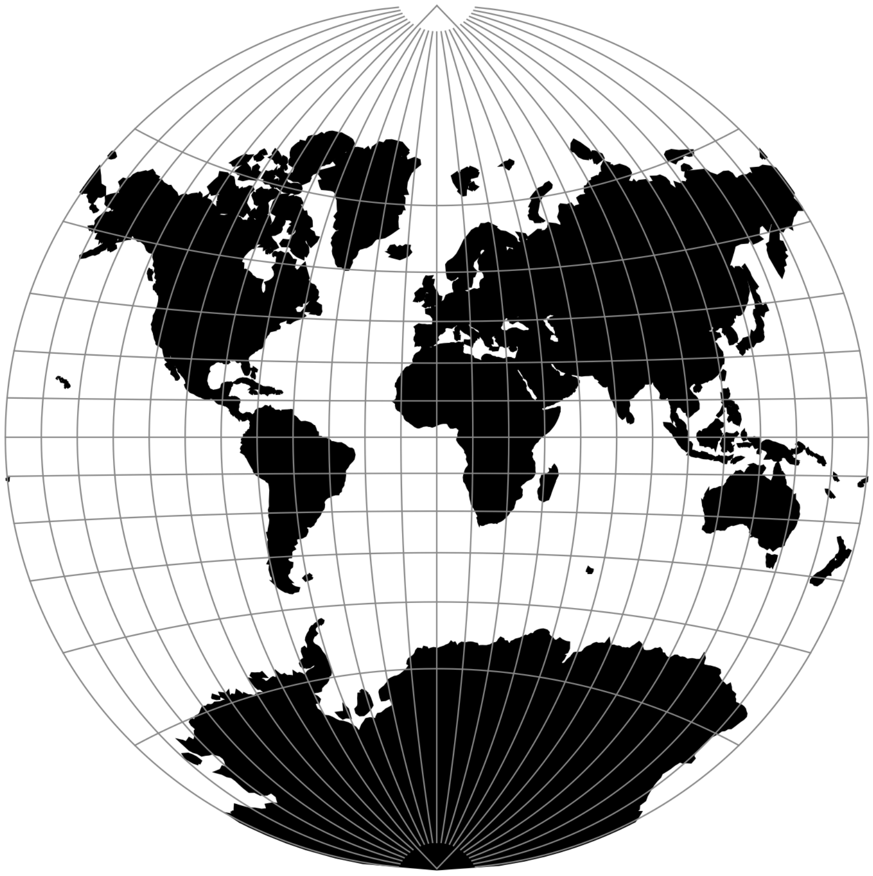


Fig. 115: proj-string: `+proj=vandg`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.120 van der Grinten II

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

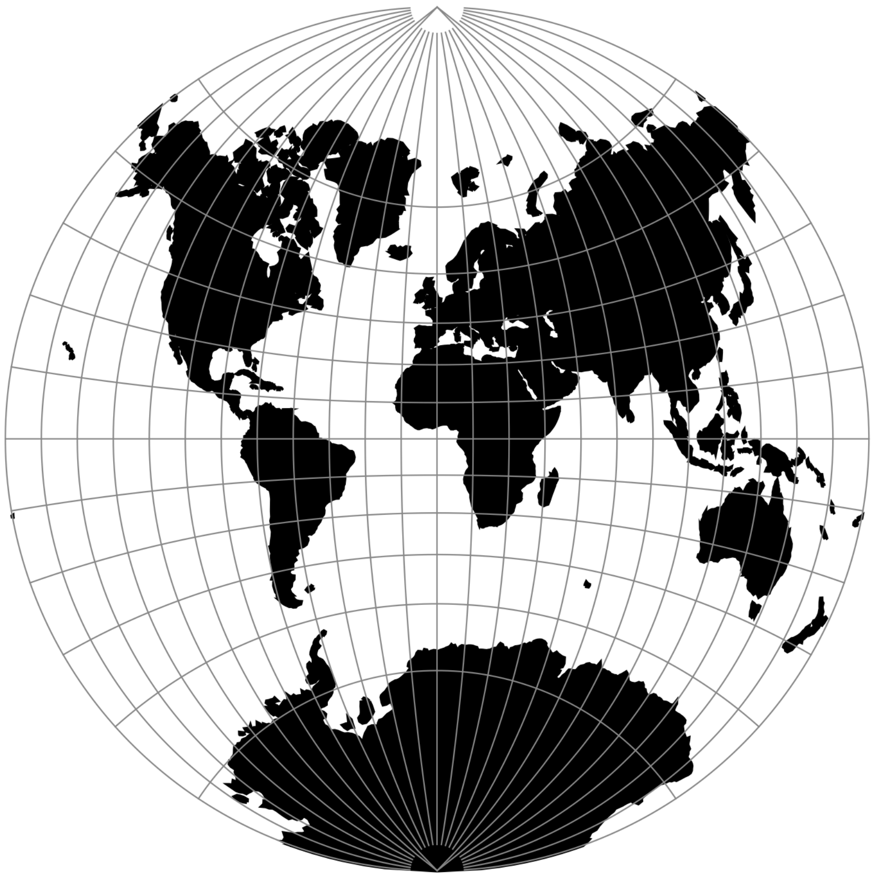


Fig. 116: proj-string: `+proj=vandg2`

7.1.121 van der Grinten III

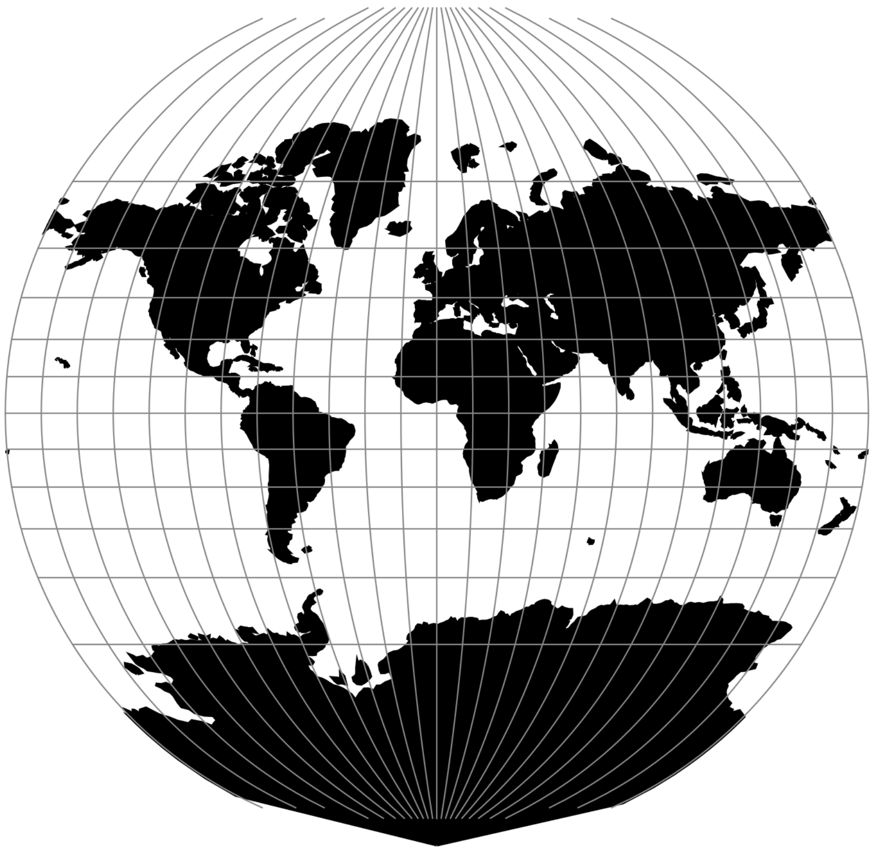


Fig. 117: proj-string: `+proj=vandg3`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
    Longitude of projection center.

    Defaults to 0.0.

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
    False easting.

    Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.122 van der Grinten IV

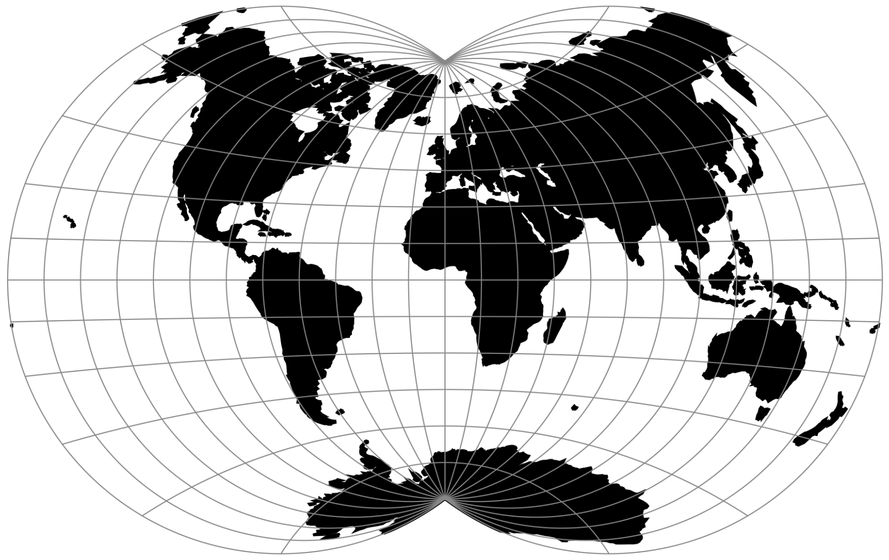


Fig. 118: proj-string: `+proj=vandg4`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.123 Vitkovsky I

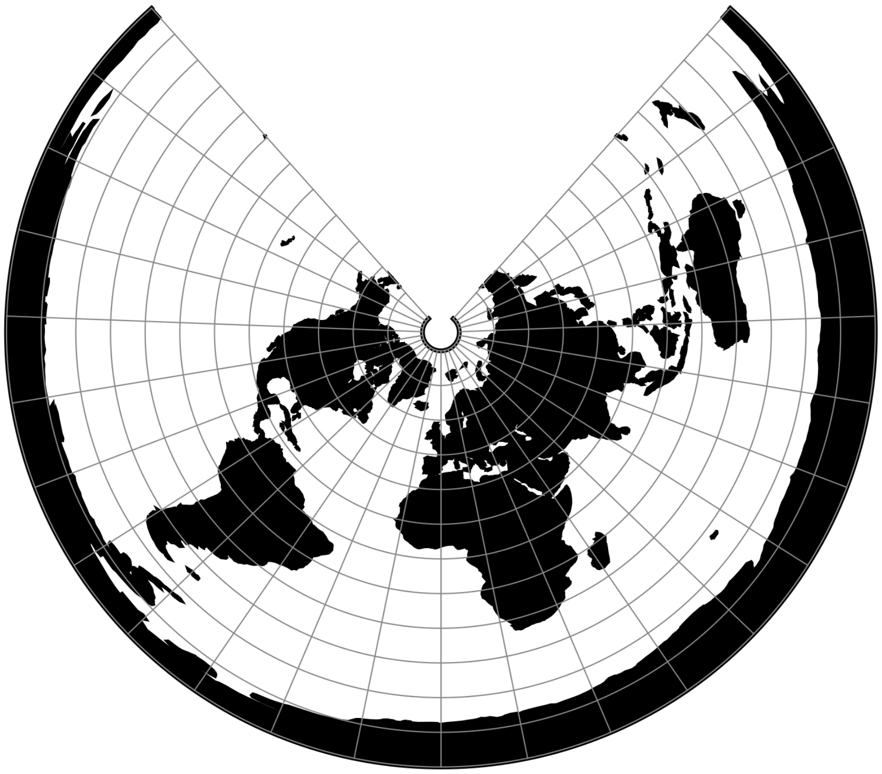


Fig. 119: proj-string: `+proj=vitk1 +lat_1=45 +lat_2=55`

Parameters

Required

+lat_1=<value>
    First standard parallel.

    Defaults to 0.0.

+lat_2=<value>
    Second standard parallel.

    Defaults to 0.0.

Optional

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.124 Wagner I (Kavraisky VI)

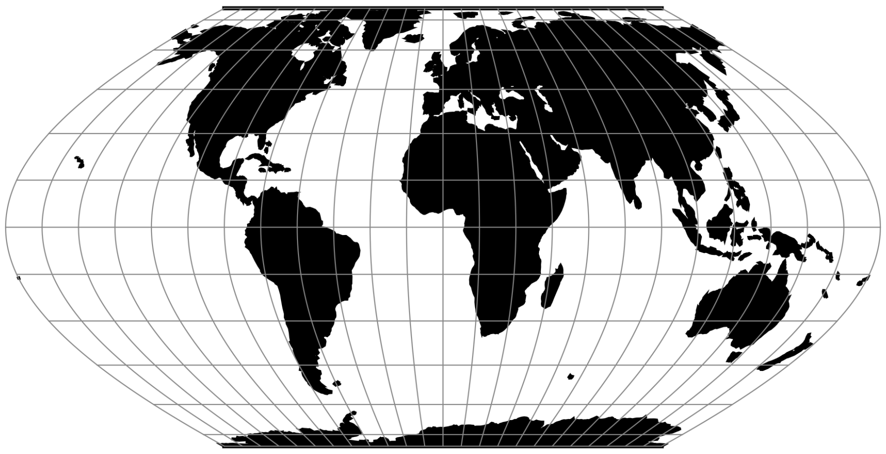


Fig. 120: proj-string: `+proj=wag1`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.125 Wagner II

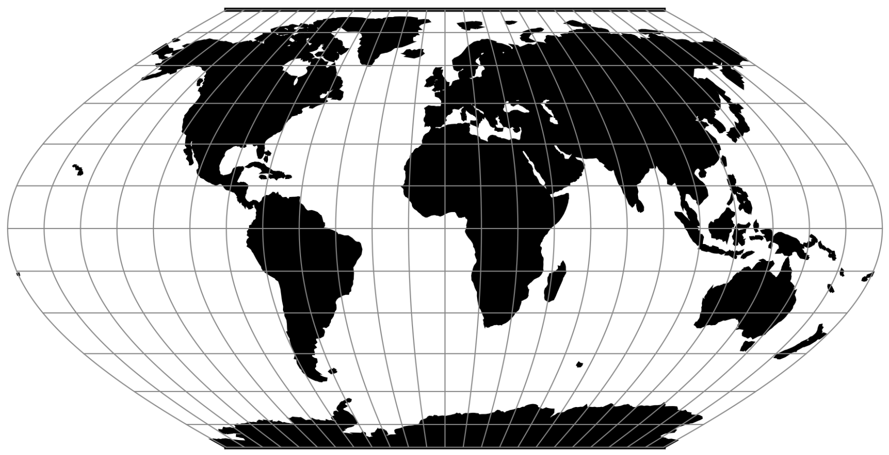


Fig. 121: proj-string: `+proj=wag2`

$$x = 0.92483\lambda \cos\theta$$
$$y = 1.38725\theta$$
$$\sin\theta = 0.88022\sin(0.8855\phi)$$

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.126 Wagner III

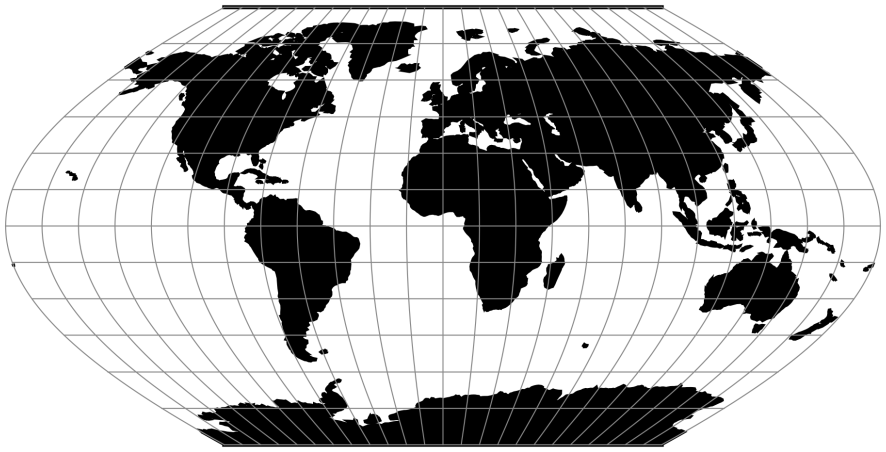


Fig. 122: proj-string: `+proj=wag3`

$$x = [\cos \phi_{ts} / \cos(2\phi_{ts}/3)] \lambda \cos(2\phi/3)$$
$$y = \phi$$

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.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.127 Wagner IV

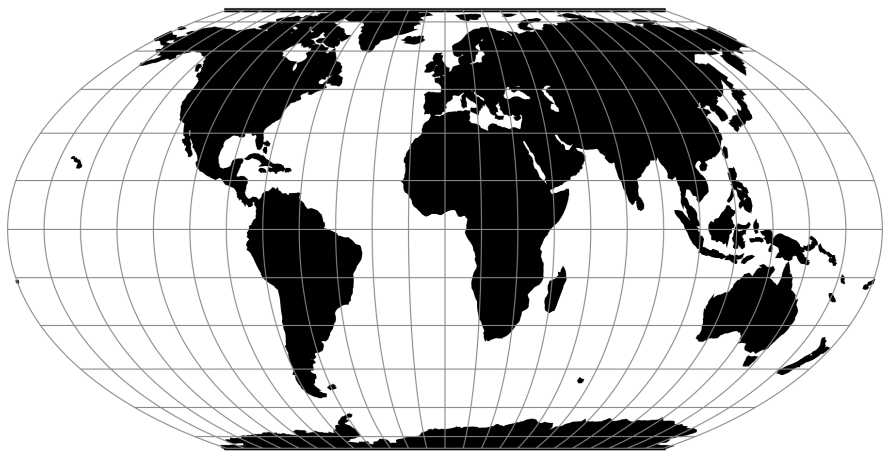


Fig. 123: proj-string: `+proj=wag4`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.128 Wagner V

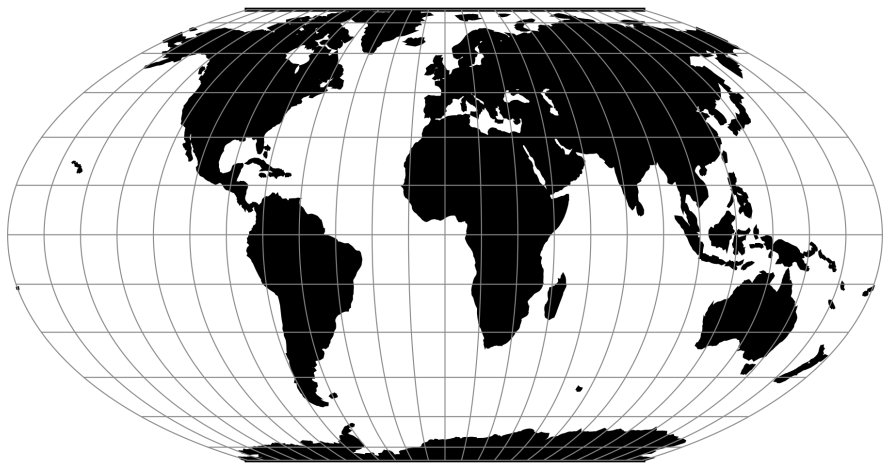


Fig. 124: proj-string: `+proj=wag5`

Parameters

Note: All parameters are optional.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.129 Wagner VI

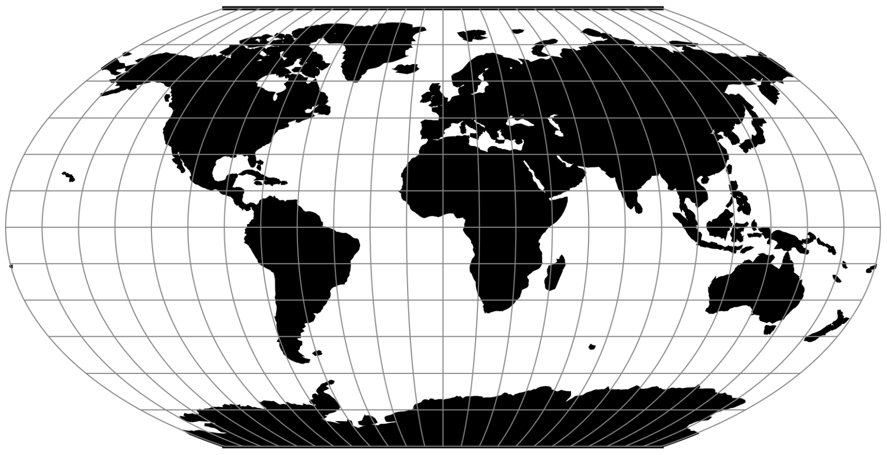


Fig. 125: proj-string: `+proj=wag6`

Parameters

Note: All parameters are optional for the Wagner VI projection.

`+lon_0``=<value>`
Longitude of projection center.

Defaults to 0.0.

`+R``=<value>`
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

`+x_0``=<value>`
False easting.

Defaults to 0.0.

`+y_0``=<value>`
False northing.

Defaults to 0.0.

7.1.130 Wagner VII

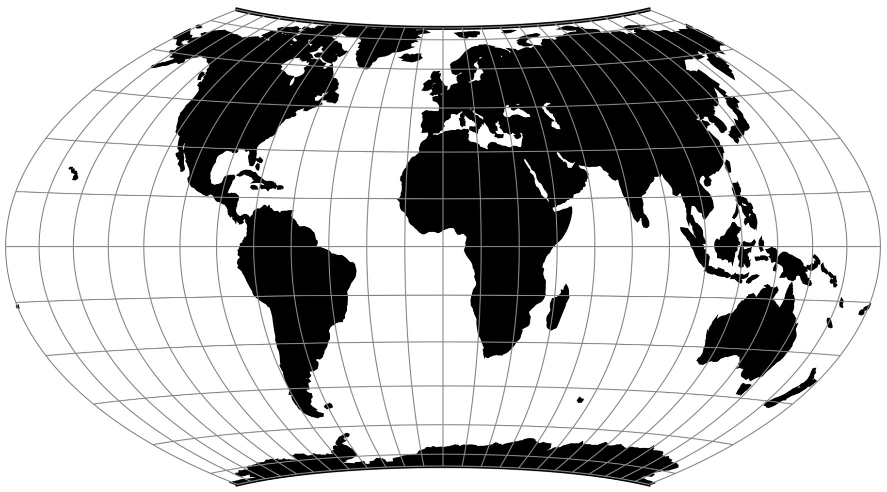


Fig. 126: proj-string: `+proj=wag7`

7.1.131 Web Mercator / Pseudo Mercator

New 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

Usage

Example:

```
$ echo 2 49 | proj +proj=webmerc +datum=WGS84
222638.98       6274861.39
```

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>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

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

$$\lambda = x$$

$$\phi = \frac{\pi}{2} - 2\arctan\left[e^{-y}\right]$$

Further reading

1. Wikipedia

7.1.132 Werenskiold I

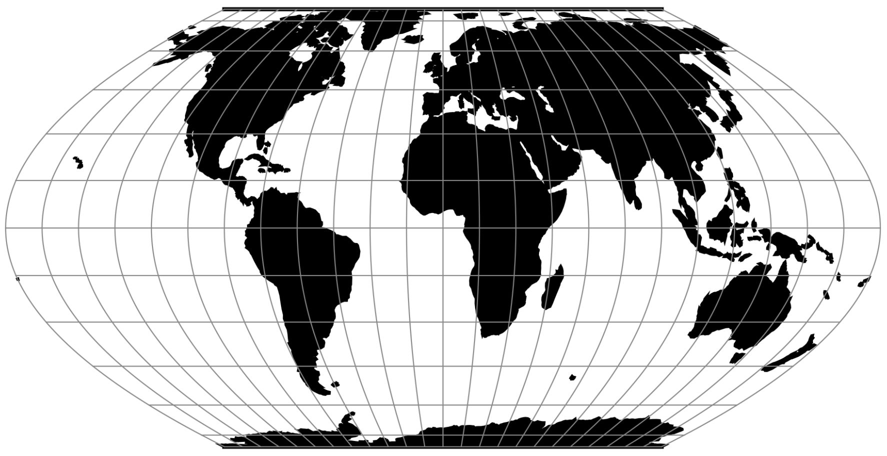


Fig. 127: proj-string: `+proj=weren`

Parameters

Note: All parameters are optional for the projection.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.133 Winkel I

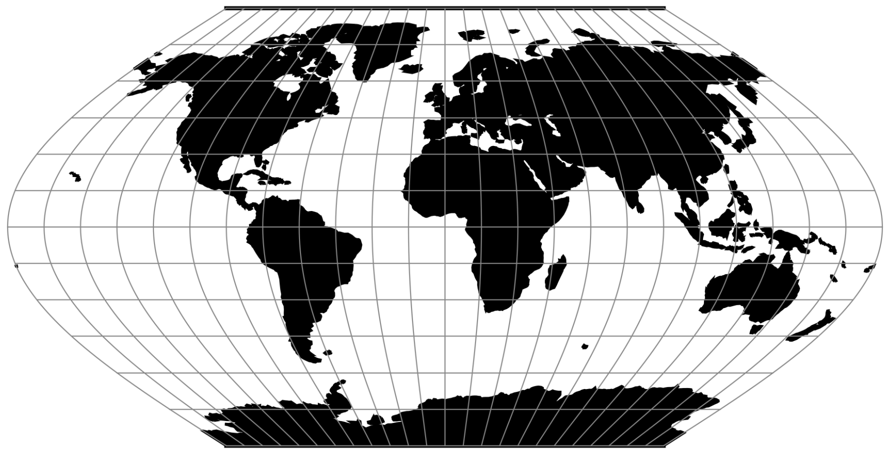


Fig. 128: proj-string: `+proj=wink1`

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.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.134 Winkel II

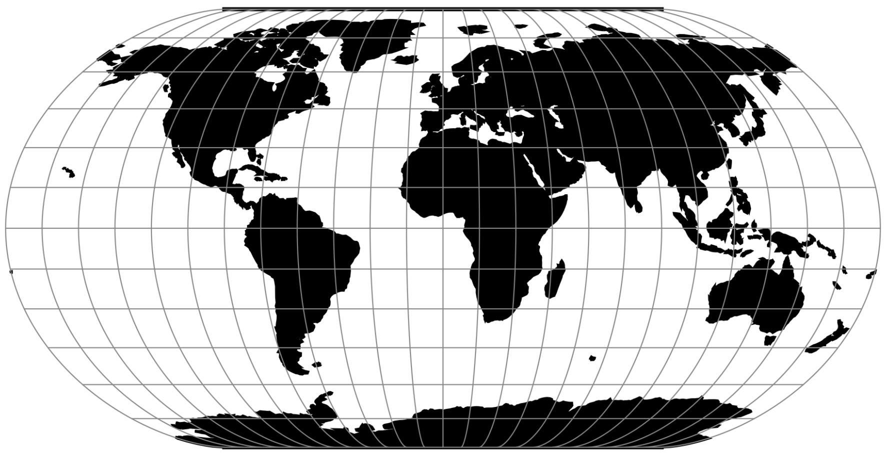


Fig. 129: proj-string: `+proj=wink2`

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.

+lon_0=<value>
Longitude of projection center.

Defaults to 0.0.

+R=<value>
Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
False easting.

Defaults to 0.0.

+y_0=<value>
False northing.

Defaults to 0.0.

7.1.135 Winkel Tripel

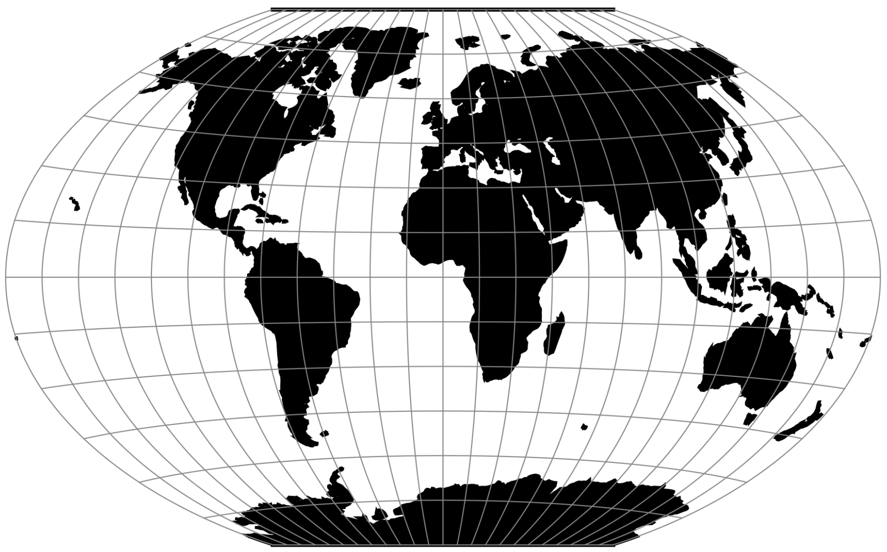


Fig. 130: proj-string: `+proj=wintri`

Parameters

Note: All parameters are optional for the projection.

+lat_1=<value>
    First standard parallel.

    Defaults to 0.0.

+lon_0=<value>
    Longitude of projection center.

    Defaults to 0.0.

+R=<value>
    Radius of the sphere given in meters. If used in conjunction with `+ellps` `+R` takes precedence.

+x_0=<value>
    False easting.

    Defaults to 0.0.

+y_0=<value>
    False northing.

    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

New 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.

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
```

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

New 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 centrer to the point of longitude=90, latitude=0 and the third axis (Z) points to the North pole.

Usage

Convert geodetic coordinates to GRS80 cartesian coordinates:

```
echo 17.7562015132 45.3935192042 133.12 2017.8 | cct +proj=cart +ellps=GRS80
4272922.1553   1368283.0597  4518261.3501     2017.8000
```

Parameters

+ellps=<value>

See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

7.2.3 Geocentric Latitude

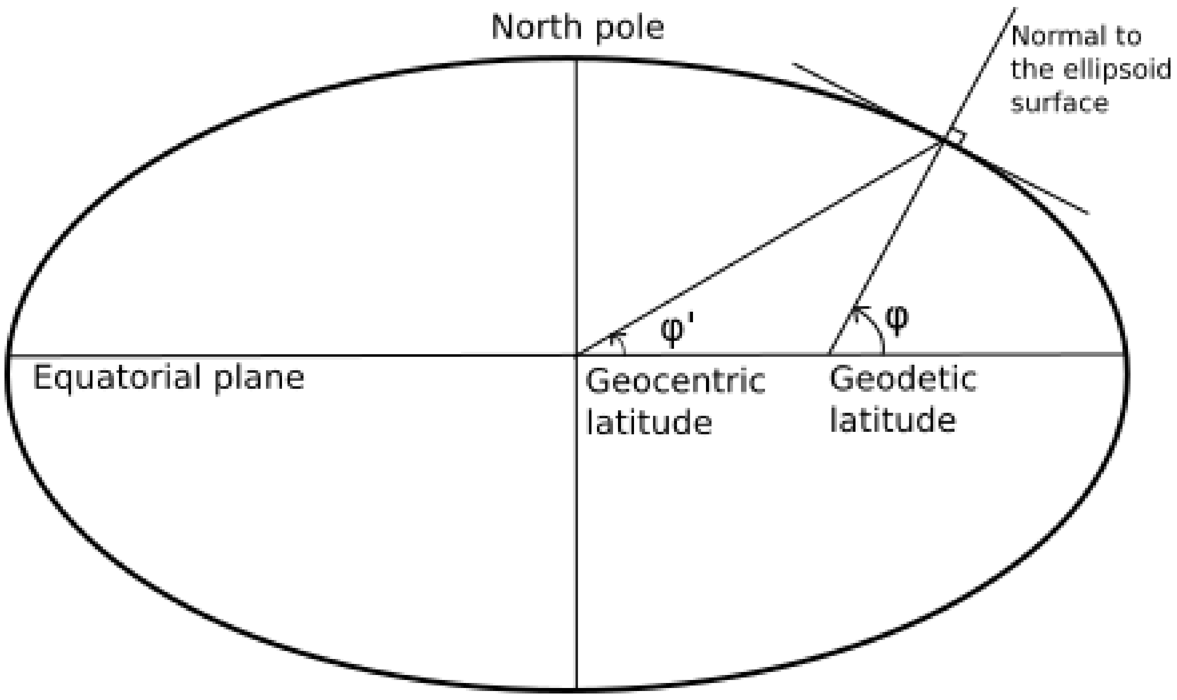
New in version 5.0.0.

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 fom the *Geodetic to cartesian conversion* which converts geodetic coordinates to geocentric coordinates in the cartesian domain.

Mathematical definition

The formulas describing the conversion are taken from [Snyder1987] (equation 3-28)

Let ϕ' to be the geocentric latitude and ϕ 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.

Usage

Converting from geodetic latitude to geocentric latitude:

```
+proj=geoc +ellps=GRS80
```

Converting from geocentric latitude to geodetic latitude:

```
+proj=pipeline +step +proj=geoc +inv +ellps=GRS80
```

Parameters

+ellps=<value>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

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.

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>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+datum=<value>
Declare the datum used with the coordinates. See `cs2cs -l` for a list of available datums.

+towgs84=<list>
A list of three or seven *Helmert* parameters that maps the input coordinates to the WGS84 datum.

7.2.5 Unit conversion

New 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 +y_
↪0=0
+step +proj=unitconvert +xy_in=m +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 ...
```

Parameters

+xy_in=<unit> or <conversion_factor>
: Horizontal input 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.

+xy_out=<unit> or <conversion_factor>
: Horizontal 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.

+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.

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

Angular units

New 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

Time units

In the table below all time units supported by PROJ are listed.

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 Kinematic datum shifting utilizing a deformation model

New in version 5.0.0.

Perform datum shifts means of a deformation/velocity model.

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.

The horizontal grid is stored in CTable2 format and the vertical grid is stored in the GTX format. Both grids are expected to contain grid-values in units of mm/year. Details about the formats can be found in the GDAL documentation. GDAL both reads and writes both file formats. Using GDAL for construction of new grids is recommended.

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
        xy_grids = ./nkgrf03vel_realigned_xy.ct2
        z_grids  = ./nkgrf03vel_realigned_z.gtx
        # 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
```

(continues on next page)

(continued from previous page)

```
step    proj = deformation t_epoch = 1994.704 t_obs = 2000.0
        xy_grids = ./nkgrf03vel_realigned_xy.ct2
        z_grids  = ./nkgrf03vel_realigned_z.gtx
```

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
xy_grids = ./nkgrf03vel_realigned_xy.ct2
z_grids  = ./nkgrf03vel_realigned_z.gtx
```

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 two grids, specified with *+xy_grids* and *+z_grids*. The first is the horizontal part of the model and the second is the vertical component.

Parameters

Required

+xy_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 for the horizontla component of a deformation model is expected to be in CTable2 format.

+z_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 for the vertical component of a deformation model is expected to be in either GTX format.

+t_epoch=<value>
Central epoch of transformation given in decimalyears.

Optional

+t_obs=<value>
Observation time of coordinate(s) given in decimalyears. If not specified, the observation time from the temporal component of 4D input points is used.

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_c and t_{obs} determine the magnitude of the coordinate correcton as seen in eq. (7.1) below.

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_B = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}_A + (t_c - t_{obs}) \begin{pmatrix} V_X \\ V_Y \\ V_Z \end{pmatrix} \tag{7.1}$$

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 ϕ and λ 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.

7.3.2 Helmert transform

New in version 5.0.0.

The Helmert transformation changes coordinates from one reference frame to anoether 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 it's 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 behaviour 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 can either by stated as part of the coordinate when using PROJ's 4D-functionality or it can be controlled in the transformation setup by the parameter *t_obs*. When *t_obs* is specified, all transformed coordinates are treated as if they have the same observation time.

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@2017.0* to *ITRF93@2017.0* 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 t_obs=2017.0
```

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

New in version 5.2.0.

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 x-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.

+t_obs=<value>
Observation time of coordinate(s) given in decimalyear. Mostly useful in 2D and 3D transformations where the observation time is not passed as part of the input coordinate. Can be used to override the observation time from the input coordinate.

+exact
Use exact transformation equations.

See (7.6)

+transpose
Deprecated since version 5.2.0: (removed)

Transpose rotation matrix and follow the **Position Vector** rotation convention. If *+transpose* is not added the **Coordinate Frame** rotation convention is used.

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

$$\dot{P} = P + \hat{P}\left(t - t_{central}\right) \tag{7.1}$$

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 scale. 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 $R_x = radians\left(rx\right)$, $R_z = radians\left(ry\right)$ and $R_z = radians\left(rz\right)$

In the *Coordinate Frame* convention, $R_x = -radians\left(rx\right)$, $R_z = -radians\left(ry\right)$ and $R_z = -radians\left(rz\right)$

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:

$$\mathbf{R} = \mathbf{R_X R_Y R_Y} \tag{7.5}$$

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 approxition 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:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} \dot{T_x} \\ \dot{T_y} \\ \dot{T_z} \end{bmatrix} + \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{7.7}$$

7.3.3 Horner polynomial evaluation

New in version 5.0.0.

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 inhomogenous 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 [EPSGGuidanceNumber7Part2].

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 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.

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).

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
```

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>

See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

+deg=<value>
Degree of polynomial

+fwd_origin=<northing,easting>
Coordinate of origin for the forward mapping

+inv_origin=<northing,easting>
Coordinate of origin for the inverse 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_11,u_12,...,u_ij,..,u_mn>
Coefficients for the forward transformation i.e. latitude to northing as described in (7.7).

+fwd_v=<v_11,v_12,...,v_ij,..,v_mn>
Coefficients for the forward transformation i.e. longitude to easting as described in (7.7).

+inv_u=<u_11,u_12,...,u_ij,..,u_mn>
Coefficients for the inverse transformation i.e. latitude to northing as described in (7.7).

+inv_v=<v_11,v_12,...,v_ij,..,v_mn>
Coefficients for the inverse 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).

+inv_c=<c_1,c_2,...,c_N>
Coefficients for the complex inverse transformation as described in (7.10).

Optional

+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.

Further reading

1. Wikipedia

7.3.4 Molodensky transform

New 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 parameterize, 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 (ϕ_1, λ_1, h_1) to (ϕ_2, λ_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 (δa) and flattening (δf), 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].

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
```

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>
See `proj -le` for a list of available ellipsoids.

Defaults to "WGS84".

Optional

+abridged
Use the abridged version of the Molodensky transform.

7.3.5 Horizontal grid shift

New in version 5.0.0.

Change of horizontal datum by grid shift.

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 CTable2, NTv1 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.

Temporal gridshifting

New 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.

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 CTable2, NTv1 or NTv2 format.

Optional

+t_epoch=<time>
Central epoch of the transformation.

New 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.

New in version 5.1.0.

7.3.6 Vertical grid shift

New in version 5.0.0.

Change Vertical datum change by grid shift

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_16.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_16.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 GTX file format for vertical grid corrections. Details about all the format can be found in the GDAL documentation. GDAL both reads and writes the format. Using GDAL for construction of new grids is recommended.

Temporal gridshifting

New 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.

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 GTX format.

Optional

+t_epoch=<time>
Central epoch of the transformation.

New 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.

New in version 5.1.0.

+multiplier=<value>
Specify the multiplier to apply to the grid value in the forward transformation direction, such that:

$$Z_{target} = Z_{source} + multiplier \times gridvalue \tag{7.11}$$

The multiplier can be used to control whether the gridvalue should be added or sustracted, 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.

New in version 5.2.0.

7.4 The pipeline operator

New 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.

New 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

Required

`+step`
Separate each step in a pipeline.

Optional

`+inv`
Invert a step in a pipeline.

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 contains 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.4 project also distribute a number of packages containing transformation grids and additional init-files not included in the main PROJ package.

8.1 External resources

For a functioning PROJ 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.

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

Click the links to jump to the relevant README files for each package. Details on the content of the packages maintained there.

Links to the resource packages can be found in the *download section*.

8.2 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.

Below is a given a list of grid resources for various countries which are not included in the grid distributions mentioned above.

8.2.1 Free grids

Below is a list of grids distributed under a free and open license.

Switzerland

Background in ticket #145

We basically have two shift grids available. An official here:

Swiss CHENyx06 dataset in NTv2 format

And a derived in a temporary location which is probably going to disappear soon.

Main problem seems to be there's no mention of distributivity of the grid from the official website. It just tells: "you can use freely". The "contact" link is also broken, but maybe someone could make a phone call to ask for rephrasing that.

Hungary

Hungarian grid ETRS89 - HD72/EOV (epsg:23700), both horizontal and elevation grids

8.2.2 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.

Austria

Austrian Grid for MGI

Brazil

Brazilian grids for datums Corrego Alegre 1961, Corrego Alegre 1970-72, SAD69 and SAD69(96)

Netherlands

Dutch grid (Registration required before download)

Portugal

Portuguese grids for ED50, Lisbon 1890, Lisbon 1937 and Datum 73

South Africa

South African grid (Cape to Hartebeesthoek94 or WGS84)

Spain

Spanish grids for ED50.

8.2.3 HARN

With the support of i-cubed, Frank Warmerdam has written tools to translate the HPGN grids from NOAA/NGS from `.los/.las` format into NTv2 format for convenient use with PROJ. This project included implementing a .los/.las reader for GDAL, and an NTv2 reader/writer. Also, a script to do the bulk translation was implemented in https://github.com/OSGeo/gdal/tree/trunk/gdal/swig/python/samples/loslas2ntv2.py. The command to do the translation was:

```
loslas2ntv2.py -auto *hpgn.los
```

As GDAL uses NAD83/WGS84 as a pivot datum, the sense of the HPGN datum shift offsets were negated to map from HPGN to NAD83 instead of the other way. The files can be used with PROJ like this:

```
cs2cs +proj=latlong +datum=NAD83
      +to +proj=latlong +nadgrids=./azhpgn.gsb +ellps=GRS80
```

```
# input:
-112 34
```

```
# output:
111d59'59.996"W 34d0'0.006"N -0.000
```

This was confirmed against the NGS HPGN calculator.

The grids are available at http://download.osgeo.org/proj/hpgn_ntv2.zip

8.2.4 HTDP

This page documents 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).

Getting and building HTDP

The HTDP modelling program is in written 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
```

Getting crs2crs2grid.py

The *crs2crs2grid.py* script can be found at https://github.com/OSGeo/gdal/tree/trunk/gdal/swig/python/samples/crs2crs2grid.py

It depends on having the GDAL Python bindings operational. If they are not

```
Traceback (most recent call last):
  File "./crs2crs2grid.py", line 37, in <module>
    from osgeo import gdal, gdal_array, osr
ImportError: No module named osgeo
```

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 the "htdp" program 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 the *htdp* program will be 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)
```

(continues on next page)

(continued from previous page)

```
 31...NAD_83(PACP00)  (NAD 83(PA11) will be used)
 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
```

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.

8.3 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 transformations 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_LIB` environment variable.

The format of init files made up of a 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 an observation epoch when transforming from ITRF2000 to ITRF2005:

```
+init=ITRF2000:ITRF2005 +t_obs=2010.5
```

which then expands to

```
+proj=helmert +x=-0.0001 +y=0.0008 +z=0.0058 +s=-0.0004
+dx=0.0002 +dy=-0.0001 +dz=0.0018 +ds=-0.000008
+t_epoch=2000.0 +convention=position_vector
+t_obs=2010.5
```

Below is a list of the init files that are packaged with PROJ.

Name	Description
esri	Auto-generated mapping from Esri projection index. Not maintained any more
epsg	EPSG database
GL27	Great Lakes Grids
IGNF	French coordinate systems supplied by the IGNF
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

8.4 The defaults file

The `proj_def.dat` file supplies default parameters for PROJ. It uses the same syntax as the init files described above. The identifiers in the defaults file describe to what the parameters should apply. If the `<general>` identifier is used, then all parameters in that section applies for all proj-strings. Otherwise the identifier is connected to a specific projection. With the defaults file supplied with PROJ the default ellipsoid is set to WGS84 (for all proj-strings). Apart from that only the Albers Equal Area, *Lambert Conic Conformal* and the *Lagrange* projections have default parameters. Defaults can be ignored by adding the `+no_def` parameter to a proj-string.

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 ϕ and longitude λ. (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 (ϕ_1, λ_1) and (ϕ_2, λ_2) is called the geodesic. Its length is s_{12} and the geodesic from point 1 to point 2 has forward azimuths α_1 and α_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 ϕ_1, λ_1, α_1, s_{12}, determine ϕ_2, λ_2, α_2.
- the inverse problem — given ϕ_1, λ_1, ϕ_2, λ_2, determine s_{12}, α_1, α_2.

PROJ incorporates C library for Geodesics from GeographicLib. This library provides routines to solve the direct and inverse geodesic problems. Full double precision accuracy is maintained provided that $|f| < \frac{1}{50}$. Refer to the

application programming interface

for full documentation. A brief summary of the routines is given in geodesic(3).

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 (ϕ_1, λ_1), $(0, \lambda_1)$, $(0, \lambda_2)$, and (ϕ_2, λ_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 :math`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 $M_{12} = M_{21} = 1$.
- σ_{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 $180°$.

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 $\pm 180°$ for oblate ellipsoids.)
- $\lambda_2 = \lambda_1 \pm 180°$ (with neither point at a pole). If $\alpha_1 = 0°$ or $\pm 180°$, 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 ϕ_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 δ. (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 δ.

9.5 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 then 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 Quick start

This is a short introduction to the PROJ API. In the following section we create a simple program that transforms a geodetic coordinate to UTM and back again. The program is explained a few lines at a time. The complete program can be seen at the end of the section.

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 <stdio.h>
#include <proj.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_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 it created here. The section on *threads* discusses this in detail.

```
/* use PJ objects from only one thread                          */
```

Next we create the `PJ` transformation object `P` with the function `proj_create`. `proj_create` takes the threading context `C` created above, and a proj-string that defines the desired transformation. Here we transform from geodetic coordinate to UTM zone 32N. 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. In case the creation of the `PJ` object fails an error message is displayed and the program returns. See *Error handling* for further details.

```

P = proj_create (C, "+proj=utm +zone=32 +ellps=GRS80");
if (0==P)
```

PROJ uses it's own data structures for handling coordinates. Here we use a `PJ_COORD` which is easily assigned with the function `proj_coord`. Note that the input values are converted to radians with `proj_torad`. This is necessary since PROJ is using radians internally. See *Transformations* for further details.

```
/* note: PROJ.4 works in radians, hence the proj_torad() calls */
```

The coordinate defined above is transformed with `proj_trans_coord`. For this a `PJ` object, a transformation direction (either forward or inverse) and the coordinate is needed. The transformed coordinate is returned in `b`. Here the forward (`PJ_FWD`) transformation from geodetic to UTM is made.

```
/* transform to UTM zone 32, then back to geographical */
b = proj_trans (P, PJ_FWD, a);
```

The inverse transformation (UTM to geodetic) is done similar to above, this time using `PJ_INV` as the direction.

```
printf ("easting: %g, northing: %g\n", b.enu.e, b.enu.n);
b = proj_trans (P, PJ_INV, b);
```

Before ending the program the allocated memory needs to be released again:

```
/* Clean up */
proj_destroy (P);
```

A complete compilable version of the above can be seen here:

```
#include <stdio.h>
#include <proj.h>

int main (void) {
    PJ_CONTEXT *C;
    PJ *P;
    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                          */
    C = proj_context_create();

    P = proj_create (C, "+proj=utm +zone=32 +ellps=GRS80");
    if (0==P)
        return puts ("Oops"), 0;

    /* a coordinate union representing Copenhagen: 55d N, 12d E    */
    /* note: PROJ.4 works in radians, hence the proj_torad() calls */
    a = proj_coord (proj_torad(12), proj_torad(55), 0, 0);

    /* transform to UTM zone 32, then back to geographical */
    b = proj_trans (P, PJ_FWD, a);
    printf ("easting: %g, northing: %g\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;
}
```

10.2 Transformations

10.3 Error handling

10.4 Threads

This page is about efforts to make PROJ thread safe.

10.4.1 Key Thread Safety Issues

- the global pj_errno variable is shared between threads and makes it essentially impossible to handle errors safely. Being addressed with the introduction of the projCtx execution context.
- the datum shift using grid files uses globally shared lists of loaded grid information. Access to this has been made safe in 4.7.0 with the introduction of a PROJ mutex used to protect access to these memory structures (see pj_mutex.c).

10.4.2 projCtx

Primarily in order to avoid having pj_errno as a global variable, a "thread context" structure has been introduced into a variation of the PROJ API for the 4.8.0 release. The pj_init() and pj_init_plus() functions now have context variations called pj_init_ctx() and pj_init_plus_ctx() which take a projections context.

The projections context can be created with pj_ctx_alloc(), and there is a global default context used when one is not provided by the application. There is a pj_ctx_ set of functions to create, manipulate, query, and destroy contexts. The contexts are also used now to handle setting debugging mode, and to hold an error reporting function for textual error and debug messages. The API looks like:

```
projPJ pj_init_ctx( projCtx, int, char ** );
projPJ pj_init_plus_ctx( projCtx, const char * );

projCtx pj_get_default_ctx(void);
projCtx pj_get_ctx( projPJ );
void pj_set_ctx( projPJ, projCtx );
projCtx pj_ctx_alloc(void);
void    pj_ctx_free( projCtx );
int pj_ctx_get_errno( projCtx );
void pj_ctx_set_errno( projCtx, int );
void pj_ctx_set_debug( projCtx, int );
void pj_ctx_set_logger( projCtx, void (*)(void *, int, const char *) );
void pj_ctx_set_app_data( projCtx, void * );
void *pj_ctx_get_app_data( projCtx );
```

Multithreaded applications are now expected to create a projCtx per thread using pj_ctx_alloc(). The context's error handlers, and app data may be modified if desired, but at the very least each context has an internal error value accessed with pj_ctx_get_errno() as opposed to looking at pj_errno.

Note that pj_errno continues to exist, and it is set by pj_ctx_set_errno() (as well as setting the context specific error number), but pj_errno still suffers from the global shared problem between threads and should not be used by multithreaded applications.

Note that pj_init_ctx(), and pj_init_plus_ctx() will assign the projCtx to the created projPJ object. Functions like pj_transform(), pj_fwd() and pj_inv() will use the context of the projPJ for error reporting.

10.4.3 src/multistresstest.c

A small multi-threaded test program has been written (src/multistresstest.c) for testing multithreaded use of PROJ. It performs a series of reprojections to setup a table expected results, and then it does them many times in several threads to confirm that the results are consistent. At this time this program is not part of the builds but it can be built on linux like:

```
gcc -g multistresstest.c .libs/libproj.so -lpthread -o multistresstest
./multistresstest
```

10.5 Reference

10.5.1 Data types

This section describes the numerous data types in use in PROJ.4. As a rule of thumb PROJ.4 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.4 when the `PJ_` prefix was not consistently used.

Transformation objects

PJ

Object containing everything related to a given projection or transformation. As a user of the PROJ.4 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()*.

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;
```

PJ_FWD

Perform transformation in the forward direction.

PJ_IDENT

Identity. Do nothing.

PJ_INV

Perform transformation in the inverse direction.

PJ_CONTEXT

Context objects enable safe multi-threaded usage of PROJ.4. 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()*.

PJ_AREA

Opaque object describing an area in which a transformation is performed.

Note: This object is not fully implemented yet. It is to be used with *proj_create_crs_to_crs()* to select the best transformation between the two input coordinate reference systems.

2 dimensional coordinates

Various 2-dimensional coordinate data types.

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.

PJ_XY

2-dimensional cartesian coordinate.

```
typedef struct { double x, y; } PJ_XY;
```

double **PJ_XY.x**

Easting.

double **PJ_XY.y**

Northing.

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.

3 dimensional coordinates

The following data types are the 3-dimensional equivalents to the data types above.

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.

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.

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.

Spatiotemporal coordinate types

The following data types are extensions of the triplets above into the time domain.

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.

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.

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.

Ancillary types for geodetic computations

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.

Complex coordinate types

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_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_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.

Projection derivatives

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;
    double dy_dlam;
    double dy_dphi;
} PJ_FACTORS;
```

double **PJ_FACTORS.meridional_scale**

Meridional scale at coordinate (λ, ϕ).

double **PJ_FACTORS.parallel_scale**

Parallel scale at coordinate (λ, ϕ).

double **PJ_FACTORS.areal_scale**

Areal scale factor at coordinate (λ, ϕ).

double **PJ_FACTORS.angular_distortion**

Angular distortion at coordinate (λ, ϕ).

double **PJ_FACTORS.meridian_parallel_angle**

Meridian/parallel angle, θ', at coordinate (λ, ϕ).

double **PJ_FACTORS.meridian_convergence**

Meridian convergence at coordinate (λ, ϕ). 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 (λ, ϕ).

double **PJ_FACTORS.dy_dlam**

Partial derivative $\frac{\partial y}{\partial \lambda}$ of coordinate (λ, ϕ).

double **PJ_FACTORS.dx_dphi**

Partial derivative $\frac{\partial x}{\partial \phi}$ of coordinate (λ, ϕ).

double **PJ_FACTORS.dy_dphi**

Partial derivative $\frac{\partial y}{\partial \phi}$ of coordinate (λ, ϕ).

List structures

PJ_OPERATIONS

Description a PROJ.4 operation

```
struct PJ_OPERATIONS {
    char    *id;                /* operation keyword */
    PJ *(*proj)(PJ *);          /* operation  entry point */
    char    * const *descr;     /* description text */
};
```

char ***id**

Operation keyword.

PJ ***(*op)** (*PJ* *)

Operation entry point.

char * const *

Description of operation.

PJ_ELLPS

Description of ellipsoids defined in PROJ.4

```
struct PJ_ELLPS {
    char    *id;
    char    *major;
    char    *ell;
    char    *name;
};
```

char ***id**

Keyword name of the ellipsoid.

char ***major**

Semi-major axis of the ellipsoid, or radius in case of a sphere.

char ***ell**

Elliptical parameter, e.g. *rf=298.257* or *b=6356772.2*.

char ***name**

Name of the ellipsoid

PJ_UNITS

Distance units defined in PROJ.

```
struct PJ_UNITS {
    char    *id;          /* units keyword */
    char    *to_meter;    /* multiply by value to get meters */
    char    *name;        /* comments */
    double   factor;      /* to_meter factor in actual numbers */
};
```

char ***id**

Keyword for the unit.

char ***to_meter**

Text representation of the factor that converts a given unit to meters

char ***name**

Name of the unit.

double **factor**
Conversion factor that converts the unit to meters.

PJ_PRIME_MERIDIANS
Prime meridians defined in PROJ.

```
struct PJ_PRIME_MERIDIANS {
    char    *id;
    char    *defn;
};
```

char ***id**
Keyword for the prime meridian

char ***def**
Offset from Greenwich in DMS format.

Info structures

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.

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;
```

(continues on next page)

(continued from previous page)

```
    const char  *description;
    const char  *definition;
    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 afther 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.

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**
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 latitudianl 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.

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.

Logging

PJ_LOG_LEVEL
Enum of logging levels in PROJ. Used to set the logging level in PROJ. Usually using `proj_log_level()`.

PJ_LOG_NONE
Don't log anything.

PJ_LOG_ERROR
Log only errors.

PJ_LOG_DEBUG
Log errors and additional debug information.

PJ_LOG_TRACE
Highest logging level. Log everything including very detailed debug information.

PJ_LOG_TELL
Special logging level that when used in `proj_log_level()` will return the current logging level set in PROJ.

New in version 5.1.0.

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 `void` pointer references a data structure used by the calling application, the `int` is used to set the logging level and the `const char` pointer is the string that will be logged by the function.

New in version 5.1.0.

10.5.2 Functions

Threading contexts

*PJ_CONTEXT** **proj_context_create** (void)

Create a new threading-context.

Returns *PJ_CONTEXT**

void **proj_context_destroy** (*PJ_CONTEXT* **ctx*)

Deallocate a threading-context.

Parameters

- **ctx** (PJ_CONTEXT*) – Threading context.

Transformation setup

*PJ** **proj_create** (*PJ_CONTEXT* **ctx*, const char **definition*)

Create a transformation object from a proj-string.

Example call:

```
PJ *P = proj_create(0, "+proj=etmerc +lat_0=38 +lon_0=125 +ellps=bessel");
```

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 transformation 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 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***) – Vector of strings with proj-string parameters, e.g. `+proj=merc`

Returns *PJ**

*PJ** **proj_create_crs_to_crs**(*PJ_CONTEXT *ctx*, const char **srid_from*, const char **srid_to*, *PJ_AREA *area*)

Create a transformation object that is a pipeline between two known coordinate reference systems.

`srid_from` and `srid_to` should be the value part of a `+init=...` parameter set, i.e. "epsg:25833" or "IGNF:AMST63". Any projection definition that can be found in a init-file in `PROJ_LIB` is a valid input to this function.

For now the function mimics the cs2cs app: An input and an output CRS is given and coordinates are transformed via a hub datum (WGS84). This transformation strategy is referred to as "early-binding" by the EPSG. The function can be extended to support "late-binding" transformations in the future without affecting users of the function. When the function is extended to the late-binding approach the `area` argument will be used. For now it is just a place-holder for a future improved implementation.

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.
- **srid_from** (*const char**) – Source SRID.
- **srid_to** (*const char**) – Destination SRID.
- **area** (`PJ_AREA`) – Descriptor of the desired area for the transformation.

Returns *PJ**

*PJ** **proj_destroy**(*PJ *P*)

Deallocate a `PJ` transformation object.

Parameters

- **P** (`PJ*`) –

Returns *PJ**

Coordinate transformation

PJ_COORD **proj_trans**(*PJ *P*, *PJ_DIRECTION direction*, *PJ_COORD coord*)

Transform a single `PJ_COORD` coordinate.

Parameters

- **P** (`PJ*`) –
- **direction** (`PJ_DIRECTION`) – Transformation direction.
- **coord** (`PJ_COORD`) – Coordinate that will be transformed.

Returns *PJ_COORD*

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

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_transform()` 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, sizeof(XYQS),
    &(survey[0].x), stride, 345,  /*  We have 345 eastings  */
    &(survey[0].y), stride, 345,  /*  ...and 345 northings. */
    &height, 1,                   /*  The height is the constant  23.45 m */
    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_transform()` 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`** – Transformation direction
- **`x`** (*`double*`*) – Array of x-coordinates
- **`y`** (*`double*`*) – Array of y-coordinates
- **`z`** (*`double*`*) – Array of z-coordinates
- **`t`** (*`double*`*) – Array of t-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

- **sy** (*size_t*) – Step length, in bytes, between consecutive elements of the corresponding array
- **nv** (*size_t*) – Number of elements in the corresponding array
- **sz** (*size_t*) – Step length, in bytes, between consecutive elements of the corresponding array
- **nz** (*size_t*) – Number of elements in the corresponding array
- **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

size_t **proj_trans_array** (*PJ *P*, *PJ_DIRECTION direction*, size_t *n*, *PJ_COORD *coord*)

Batch transform an array of `PJ_COORD`.

Parameters

- **P** (`PJ*`) –
- **direction** (`PJ_DIRECTION`) – Transformation direction
- **n** (*size_t*) – Number of coordinates in `coord`

Returns `size_t` 0 if all observations are transformed without error, otherwise returns error number

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()`.

Param PJ* P: 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()`.

Param PJ_CONTEXT* ctx: threading context.

Returns `int`

void **proj_errno_set** (*PJ *P*, int *err*)

Change the error-state of `P` to *err*.

param PJ* P Transformation object.

param int err 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;
}
```

Param PJ* P: 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 code.

const char* **`proj_errno_string`** (int *err*)

New in version 5.1.0.

Get a text representation of an error number.

Parameters

- **`err`** (*`int`*) – Error number.

Returns `const char*` String with description of error.

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`*

New 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.

New in version 5.1.0.

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*

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 *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 `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.

Returns `PJ_UNITS*`

const *PJ_PRIME_MERIDIANS** **proj_list_prime_meridians**(void)
Get a pointer to an array of prime meridians 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 `PJ_PRIME_MERIDIANS*`

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.

Parameters

- **P** (`PJ*`) – Transformation 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.

Parameters

- **P** (`PJ*`) – Transformation 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.

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.

Parameters

- **P** (*const PJ**) –
- **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.

The function also calculates the partial derivatives of the given coordinate.

Parameters

- **P** (*const PJ**) – Transformation object

- **lp** (*const 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.

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`)

PJ_COORD **proj_geocentric_latitude**(const *PJ* **P*, *PJ_DIRECTION direction*, *PJ_COORD coord*)
Convert from geographical latitude to geocentric latitude.

Parameters

- **P** (*const PJ**) – Transformation object
- **direction** (`PJ_DIRECTION`) – Starting direction of transformation
- **coord** (`PJ_COORD`) – Coordinate

Returns *PJ_COORD* Converted coordinate

int **proj_angular_input**(*PJ* **P*, enum *PJ_DIRECTION dir*)
Check if a operation expects angular input.

Parameters

- **P** (*const PJ**) – Transformation object

- **`direction`** (`PJ_DIRECTION`) – Starting direction of transformation

Returns `int` 1 if angular input is expected, otherwise 0

int **`proj_angular_output`** (*PJ *P*, enum *PJ_DIRECTION dir*)

Check if an operation returns angular output.

Parameters

- **`P`** (*`const PJ*`*) – Transformation object
- **`direction`** (`PJ_DIRECTION`) – Starting direction of transformation

Returns `int` 1 if angular output is returned, otherwise 0

10.5.3 Deprecated API

Contents

Introduction

Procedure `pj_init()` selects and initializes a cartographic projection with its argument control parameters. `argc` is the number of elements in the array of control strings argv that each contain individual cartographic control keyword assignments (+ proj arguments). The list must contain at least the proj=projection and Earth's radius or elliptical parameters. If the initialization of the projection is successful a valid address is returned otherwise a NULL value.

The `pj_init_plus()` function operates similarly to `pj_init()` but takes a single string containing the definition, with each parameter prefixed with a plus sign. For example `+proj=utm +zone=11 +ellps=WGS84`.

Once initialization is performed either forward or inverse projections can be performed with the returned value of `pj_init()` used as the argument proj. The argument structure projUV values u and v contain respective longitude and latitude or x and y. Latitude and longitude are in radians. If a projection operation fails, both elements of projUV are set to `HUGE_VAL` (defined in `math.h`).

Note: all projections have a forward mode, but some do not have an inverse projection. If the projection does not have an inverse the projPJ structure element inv will be `NULL`.

The `pj_transform` function may be used to transform points between the two provided coordinate systems. In addition to converting between cartographic projection coordinates and geographic coordinates, this function also takes care of datum shifts if possible between the source and destination coordinate system. Unlike `pj_fwd()` and `pj_inv()` it is also allowable for the coordinate system definitions `(projPJ *)` to be geographic coordinate systems (defined as `+proj=latlong`). The x, y and z arrays contain the input values of the points, and are replaced with the output values. The function returns zero on success, or the error number (also in `pj_errno`) on failure.

Memory associated with the projection may be freed with `pj_free()`.

Example

The following program reads latitude and longitude values in decimal degrees, performs Mercator projection with a Clarke 1866 ellipsoid and a 33° latitude of true scale and prints the projected cartesian values in meters:

```
#include <proj_api.h>

main(int argc, char **argv) {
     projPJ pj_merc, pj_latlong;
     double x, y;

     if (!(pj_merc = pj_init_plus("+proj=merc +ellps=clrk66 +lat_ts=33")) )
        exit(1);
     if (!(pj_latlong = pj_init_plus("+proj=latlong +ellps=clrk66")) )
        exit(1);
     while (scanf("%lf %lf", &x, &y) == 2) {
        x *= DEG_TO_RAD;
        y *= DEG_TO_RAD;
        p = pj_transform(pj_latlong, pj_merc, 1, 1, &x, &y, NULL );
        printf("%.2f\t%.2f\n", x, y);
     }
     exit(0);
}
```

For this program, an input of `-16 20.25` would give a result of `-1495284.21 1920596.79`.

API Functions

pj_transform

```
int pj_transform( projPJ srcdefn,
                  projPJ dstdefn,
                  long point_count,
                  int point_offset,
                  double *x,
                  double *y,
                  double *z );
```

Transform the x/y/z points from the source coordinate system to the destination coordinate system.

`srcdefn`: source (input) coordinate system.

`dstdefn`: destination (output) coordinate system.

`point_count`: the number of points to be processed (the size of the x/y/z arrays).

`point_offset`: the step size from value to value (measured in doubles) within the x/y/z arrays - normally 1 for a packed array. May be used to operate on xyz interleaved point arrays.

`x/y/z`: The array of X, Y and Z coordinate values passed as input, and modified in place for output. The Z may optionally be NULL.

`return`: The return is zero on success, or a PROJ.4 error code.

The `pj_transform()` function transforms the passed in list of points from the source coordinate system to the destination coordinate system. Note that geographic locations need to be passed in radians, not decimal degrees, and will be returned similarly. The `z` array may be passed as NULL if Z values are not available.

If there is an overall failure, an error code will be returned from the function. If individual points fail to transform - for instance due to being over the horizon - then those x/y/z values will be set to `HUGE_VAL` on return. Input values that are `HUGE_VAL` will not be transformed.

pj_init_plus

```
projPJ pj_init_plus(const char *definition);
```

This function converts a string representation of a coordinate system definition into a projPJ object suitable for use with other API functions. On failure the function will return NULL and set pj_errno. The definition should be of the general form `+proj=tmerc +lon_0 +datum=WGS84`. Refer to PROJ.4 documentation and the *Geodetic transformation* notes for additional detail.

Coordinate system objects allocated with `pj_init_plus()` should be deallocated with `pj_free()`.

pj_free

```
void pj_free( projPJ pj );
```

Frees all resources associated with pj.

pj_is_latlong

```
int pj_is_latlong( projPJ pj );
```

Returns TRUE if the passed coordinate system is geographic (`proj=latlong`).

pj_is_geocent

```
int pj_is_geocent( projPJ pj );``
```

Returns TRUE if the coordinate system is geocentric (`proj=geocent`).

pj_get_def

```
char *pj_get_def( projPJ pj, int options);``
```

Returns the PROJ.4 initialization string suitable for use with `pj_init_plus()` that would produce this coordinate system, but with the definition expanded as much as possible (for instance `+init=` and `+datum=` definitions).

pj_latlong_from_proj

```
projPJ pj_latlong_from_proj( projPJ pj_in );``
```

Returns a new coordinate system definition which is the geographic coordinate (lat/long) system underlying `pj_in`.

pj_set_finder

```
void pj_set_finder( const char *(*new_finder)(const char *) );``
```

Install a custom function for finding init and grid shift files.

pj_set_searchpath

```
void pj_set_searchpath ( int count, const char **path );``
```

Set a list of directories to search for init and grid shift files.

pj_deallocate_grids

```
void pj_deallocate_grids( void );``
```

Frees all resources associated with loaded and cached datum shift grids.

pj_strerrno

```
char *pj_strerrno( int );``
```

Returns the error text associated with the passed in error code.

pj_get_errno_ref

```
int *pj_get_errno_ref( void );``
```

Returns a pointer to the global pj_errno error variable.

pj_get_release

```
const char *pj_get_release( void );``
```

Returns an internal string describing the release version.

Obsolete Functions

`XY pj_fwd( LP lp, PJ *P );`

`LP pj_inv( XY xy, PJ *P );`

`projPJ pj_init(int argc, char **argv);`

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 `${PROJ4_LIBRARIES}` provided by the CMake configuration which comes with the library. Typical usage is:

```
find_package(PROJ4)

target_link_libraries(MyApp ${PROJ4_LIBRARIES})
```

By adding the imported library target `${PROJ4_LIBRARIES}` to the target link libraries, CMake will also pass the include directories to the compiler. This requires that you use CMake version 2.8.11 or later. If you are using an older version of CMake, then add

```
include_directories(${PROJ4_INCLUDE_DIRS})
```

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 `PROJ4_DIR`.

10.7 Language bindings

PROJ bindings are available for a number of different development platforms.

10.7.1 Python

pyproj: Python interface (wrapper for PROJ)

10.7.2 Ruby

proj4rb: Bindings for PROJ in ruby

10.7.3 TCL

proj4tcl: Bindings for PROJ in tcl (critcl source)

10.7.4 MySQL

fProj4: Bindings for PROJ in MySQL

10.7.5 Excel

proj.xll: Excel add-in for PROJ map projections

10.7.6 Visual Basic

PROJ VB Wrappers: By Eric G. Miller.

10.7.7 Fortran

Fortran-Proj: Bindings for PROJ in Fortran (By João Macedo @likeno)

10.8 Version 4 to 5 API Migration

This is a transition guide for developers wanting to migrate their code to use PROJ version 5.

10.8.1 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.

10.8.2 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 latitude and longitude from the command line and convert them to projected coordinates with the Mercator projection.

We start by writing the program for PROJ v. 4:

```c
#include <proj_api.h>

main(int argc, char **argv) {
    projPJ pj_merc, pj_latlong;
    double x, y;

    if (!(pj_merc = pj_init_plus("+proj=merc +ellps=clrk66 +lat_ts=33")) )
        return 1;
    if (!(pj_latlong = pj_init_plus("+proj=latlong +ellps=clrk66")) )
        return 1;

    while (scanf("%lf %lf", &x, &y) == 2) {
        x *= DEG_TO_RAD;
        y *= DEG_TO_RAD;
        p = pj_transform(pj_latlong, pj_merc, 1, 1, &x, &y, NULL );
        printf("%.2f\t%.2f\n", x, y);
    }

    return 0;
}
```

The same program implemented using PROJ v. 5:

```c
#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);
    }

}
```

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.

10.8.3 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_outout
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

Attention: The `projects.h` header and the functions related to it is considered deprecated from version 5.0.0 and onwards. The header will be removed from PROJ in version 6.0.0 scheduled for release February 1st 2019.

Attention: The nmake build system on Windows will not be supported from 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 will be removed from PROJ in version 7.0.0 scheduled for release February 1st 2020.

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

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.

11.1 Communication channels

11.1.1 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

11.1.2 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.4

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.

11.1.3 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

11.2 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.

11.2.1 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.

11.2.2 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.

11.2.3 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.

11.2.4 Write documentation

PROJ is in dire need of better documentation. Any contributions of documentation are greatly appreciated. The PROJ documentation is available on proj4.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.

11.2.5 Code contributions

See *Code contributions*

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.

11.2.6 Additional Resources

- General GitHub documentation
- GitHub pull request documentation

11.2.7 Acknowledgements

The *code contribution* section of this CONTRIBUTING file is inspired by PDAL's and the *legalese* section is modified from GDAL committer guidelines

11.3 Guidelines for PROJ code contributors

This is a guide for PROJ, casual or regular, code contributors.

11.3.1 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.

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

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.

Coding conventions

Programming language

PROJ is developed strictly in ANSI C 89.

Coding style

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.

File names

Files in which projections are implemented are prefixed with an upper-case `PJ_` and most other files are prefixed with lower-case `pj_`. Some file deviate from this pattern, most of them dates back to the very early releases of PROJ. New contributions should follow the pj-prefix pattern. Unless there are obvious reasons not to.

11.3.2 Tools

cppcheck static analyzer

You can run locally `scripts/cppcheck.sh` that is a wrapper script around the cppcheck utility. It is known to work with cppcheck 1.61 of Ubuntu Trusty 14.0, since this is what is currently used on Travis-CI (`travis/linux_gcc/before_install.sh`). At the time of writing, this also works with cppcheck 1.72 of Ubuntu Xenial 16.04, and latest cppcheck master.

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.

CLang Static Analyzer (CSA)

CSA is run by the `travis/csa` build configuration. You may also run it locally.

Preliminary step: install clang. For example:

```
wget http://releases.llvm.org/6.0.0/clang+llvm-6.0.0-x86_64-linux-gnu-ubuntu-14.04.
↪tar.xz
tar xJf clang+llvm-6.0.0-x86_64-linux-gnu-ubuntu-14.04.tar.xz
```

Run configure under the scan-build utility of clang:

```
./clang+llvm-6.0.0-x86_64-linux-gnu-ubuntu-14.04/bin/scan-build ./configure
```

Build under scan-build:

```
./clang+llvm-6.0.0-x86_64-linux-gnu-ubuntu-14.04/bin/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 someling like

```
./clang+llvm-6.0.0-x86_64-linux-gnu-ubuntu-14.04/bin/scan-view /tmp/scan-build-2018-
↪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 !

Typo detection and fixes

Run `scripts/fix_typos.sh`

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

11.4 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.

11.4.1 PROJ RFC 1: Project Committee Guidelines

Author Frank Warmerdam, Howard Butler

Contact howard@hobu.co

Status Passed

Last Updated 2018-06-08

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.

List of PSC Members

(up-to-date as of 2018-06)

- 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)
- Frank Warmerdam @warmerdam (USA) **Emeritus**

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.

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.

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.

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.

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.

Updates

June 2018

RFC 1 was ratified by the following members

11.4.2 PROJ RFC 2: Initial integration of "GDAL SRS barn" work

Author Even Rouault

Contact even.rouault at spatialys.com

Status Adopted (not yet merged into master)

Initial version 2018-10-09

Last Updated 2018-10-31

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

Related standards

Consult Applicable standards

(They will be linked from the PROJ documentation)

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 intrinsinc 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 appyling 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 susbstituted 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
```

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

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 mentionned 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

- **Commmon:**
    - *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.

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]]]],
                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",
```

(continues on next page)

(continued from previous page)

```
                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]],
        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␣
↪+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
```

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

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.)

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 the Sphinx extension Breathe, producing:

- One section with the C++ API
- One section with the C API

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.

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.

Runtime requirements

- libc++/libstdc++/MSVC runtime consistent with the compiler used
- libsqlite3 >= 3.7

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

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

Adoption status

The RFC has been adopted with support from PSC members Kurt Schwehr, Kristian Evers, Howard Butler and Even Rouault.

11.4.3 PROJ RFC 3: Dependency management

Author Kristian Evers

Contact kreve@sdfe.dk

Status Adopted

Last Updated 2019-01-16

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)

A change in programming language standard can only be introduced with a new major version release of PROJ. Changes for 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 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.

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. :ref:RFC2 <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.

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 aknowledges 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. Adaptation of new programming language standards should be coordinated with a major version release of PROJ.

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 requried 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.

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 acception 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)

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

CHAPTER

TWELVE

FAQ

12.1 Where can I find the list of projections and their arguments?

Here.

Additinoally, the `proj` command itself can report the list of projections using the `-lp` option, the list of ellipsoids with the `-le` option, the list of units with the `-lu` option, and the list of built-in datums with the `-ld` option.

12.2 How do I do datum shifts between NAD27 and NAD83?

Using the `cs2cs` application. The following example demonstrates using the default shift parameters for NAD27 to NAD83:

```
% cs2cs +proj=latlong +datum=NAD27 +to +proj=latlong +datum=NAD83 -117 30
```

producing:

```
117d0'2.901"W   30d0'0.407"N 0.000
```

In order for datum shifting to work properly the various grid shift files must be available. See below. More details are available in the section on *resource files*.

12.3 How do I build/configure PROJ to support datum shifting?

After downloading and unpacking the PROJ source, also download and unpack the set of datum shift files. See *Download* for instructions how to fetch and install these files

On Windows the extra nadshift target must be used. For instance `nmake /f makefile.vc nadshift` in the `proj/src` directory.

A default build and install on Unix will normally build knowledge of the directory where the grid shift files are installed into the PROJ library (usually `/usr/local/share/proj`). On Windows the library is normally built thinking that C:PROJNAD is the installed directory for the grid shift files. If the built in concept of the PROJ data directory is incorrect, the `PROJ_LIB` environment can be defined with the correct directory.

12.4 How do I debug problems with NAD27/NAD83 datum shifting?

1. Verify that you have the binary files (eg. `/usr/local/share/proj/conus`) installed on your system. If not, see the previous question.
2. Try a datum shifting operation in relative isolation, such as with the **cs2cs** command listed above. Do you get reasonable results? If not it is likely the grid shift files aren't being found. Perhaps you need to define *PROJ_LIB*?
3. The **cs2cs** command and the underlying `pj_transform()` API know how to do a grid shift as part of a more complex coordinate transformation; however, it is imperative that both the source and destination coordinate system be defined with appropriate datum information. That means that implicitly or explicitly there must be a `+datum=` clause, a `+nadgrids=` clause or a `+towgs84=` clause. For instance `cs2cs +proj=latlong +datum=NAD27 +to +proj=latlong +ellps=WGS84` won't work because defining the output coordinate system as using the ellipse WGS84 isn't the same as defining it to use the datum WGS84 (use `+datum=WGS84`). If either the input or output are not identified as having a datum, the datum shifting (and ellipsoid change) step is just quietly skipped!
4. The *PROJ_DEBUG* environment can be defined (any value) to force extra output from the PROJ library to stderr (the text console normally) with information on what data files are being opened and in some cases why a transformation fails.

```
export PROJ_DEBUG=ON
cs2cs ...
```

Note: `PROJ_DEBUG` support is not yet very mature in the PROJ library.

5. The *-v* flag to **cs2cs** can be useful in establishing more detail on what parameters being used internally for a coordinate system. This will include expanding the definition of `+datum` clause.

12.5 How do I use EPSG coordinate system codes with PROJ?

There is somewhat imperfect translation between 2D geographic and projected coordinate system codes and PROJ descriptions of the coordinate system available in the `epsg` definition file that normally lives in the `proj/nad` directory. If installed (it is installed by default on Unix), it is possible to use EPSG numbers like this:

```
% cs2cs -v +init=epsg:26711
# ---- From Coordinate System ----
#Universal Transverse Mercator (UTM)
#       Cyl, Sph
#       zone= south
# +init=epsg:26711 +proj=utm +zone=11 +ellps=clrk66 +datum=NAD27 +units=m
# +no_defs +nadgrids=conus,ntv1_can.dat
#--- following specified but NOT used
# +ellps=clrk66
# ---- To Coordinate System ----
#Lat/long (Geodetic)
#
# +proj=latlong +datum=NAD27 +ellps=clrk66 +nadgrids=conus,ntv1_can.dat
```

The proj/nad/epsg file can be browsed and searched in a text editor for coordinate systems. There are known to be problems with some coordinate systems, and any coordinate systems with odd axes, a non-greenwich prime meridian or other quirkiness are unlikely to work properly. Caveat Emptor!

12.6 How do I use 3 parameter and 7 parameter datum shifting

Datum shifts can be approximated with 3 and 7 parameter transformations. Their use with **cs2cs** is more fully described in the *geodetic tranformation* section.

More generically, the *Helmert transform* can be used with **cct**.

12.7 Does PROJ work in different international numeric locales?

No. PROJ makes extensive use of the `sprintf()` and `atof()` C functions internally to translate numeric values. If a locale is in effect that modifies formatting of numbers, altering the role of commas and periods in numbers, then PROJ will not work. This problem is common in some European locales.

On UNIX-like platforms, this problem can be avoided by forcing the use of the default numeric locale by setting the `LC_NUMERIC` environment variable to C.

```
$ export LC_NUMERIC=C
$ proj ...
```

Note: NOTE: Per ticket #49, in PROJ 4.7.0 and later pj_init() operates with locale overridden to "C" to avoid most locale specific processing for applications using the API. Command line tools may still have issues.

12.8 Changing Ellipsoid / Why can't I convert from WGS84 to Google Earth / Virtual Globe Mercator?

The coordinate system definition for Google Earth, and Virtual Globe Mercator is as follows, which uses a sphere as the earth model for the Mercator projection.

```
+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0
     +x_0=0.0 +y_0=0 +k=1.0 +units=m +no_defs
```

But, if you do something like:

```
cs2cs +proj=latlong +datum=WGS84
    +to +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0
                   +x_0=0.0 +y_0=0 +k=1.0 +units=m +no_defs
```

to convert between WGS84 and mercator on the sphere there will be substantial shifts in the Y mercator coordinates. This is because internally **cs2cs** is having to adjust the lat/long coordinates from being on the sphere to being on the WGS84 datum which has a quite differently shaped ellipsoid.

In this case, and many other cases using spherical projections, the desired approach is to actually treat the lat/long locations on the sphere as if they were on WGS84 without any adjustments when using them for converting to other coordinate systems. The solution is to "trick" PROJ into applying no change to the lat/long values when going to (and through) WGS84. This can be accomplished by asking PROJ to use a null grid shift file for switching from your spherical lat/long coordinates to WGS84.

```
cs2cs +proj=latlong +datum=WGS84 \
    +to +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 \
    +x_0=0.0 +y_0=0 +k=1.0 +units=m +nadgrids=@null +no_defs
```

Note the strategic addition of `+nadgrids=@null` to the spherical projection definition.

Similar issues apply with many other datasets distributed with projections based on a spherical earth model - such as many NASA datasets. This coordinate system is now known by the EPSG code 3857 and has in the past been known as EPSG:3785 and EPSG:900913. When using this coordinate system with GDAL/OGR it is helpful to include the +wktext so the exact PROJ string will be preserved in the WKT representation (otherwise key parameters like `+nadgrids=@null` will be dropped):

```
+proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0
            +units=m +nadgrids=@null +wktext  +no_defs
```

12.9 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*.

12.10 What options does PROJ allow for the shape of the Earth (geodesy)?

See https://github.com/OSGeo/proj.4/blob/5.2/src/pj_ellps.c for possible ellipse options. For example, putting `+ellps=WGS84` uses the `WGS84` Earth shape.

12.11 What if I want a spherical Earth?

Use `+ellps=sphere`. See https://github.com/OSGeo/proj.4/blob/5.2/src/pj_ellps.c for the radius used in this case.

12.12 How do I change the radius of the Earth? How do I use PROJ for work on Mars?

You can supply explicit values for the semi minor and semi major axes instead of using the symbolic "sphere" value. Eg, if the radius were 2000000m:

```
+proj=laea +lon_0=-40.000000 +lat_0=74.000000 +x_0=1000000 +y_0=1700000 +a=2000000␣
↪+b=2000000"
```

12.13 How do I do False Eastings and False Northings?

Use `+x_0` and `+y_0` in the projection string.

CHAPTER

THIRTEEN

GLOSSARY

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 pseudo-cylindrical 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

[EPSGGuidanceNumber7Part2] *Geomatics Guidance Note 7, part 2 - Coordinate Conversions & Transformations including Formulas*. 2018. URL: https://www.iogp.org/bookstore/product/coordinate-conversions-and-transformation-including-formulas/.

[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. URL: https://arxiv.org/abs/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, 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.

[Evenden1995] Evenden, G. I. *Cartograpic Projection Procedures for the UNIX Environment — A User's Manual*. 1995. URL: https://github.com/OSGeo/proj.4/blob/5.2/docs/old/of90-284.pdf.

[Evenden2005] Evenden, G. I. *libproj4: A Comprehensive Library of Cartographic Projection Functions (Preliminary Draft)*. 2005. URL: https://github.com/OSGeo/proj.4/blob/5.2/docs/old/libproj.pdf.

[EversKnudsen2017] Evers, K. and Knudsen, T. Transformation pipelines for PROJ.4. In *FIG Working Week 2017 Proceedings*. 2017. URL: http://www.fig.net/resources/proceedings/fig_proceedings/fig2017/papers/iss6b/ISS6B_evers_knudsen_9156.pdf.

[Helmert1880] Helmert, F. R. *Mathematical and Physical Theories of Higher Geodesy*. Volume 1. Teubner, Leipzig, 1880. doi:10.5281/zenodo.32050.

[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.

[Karney2011] Karney, C. F. F. Geodesics on an ellipsoid of revolution. *ArXiv e-prints*, 2011. arXiv:1102.1215.

[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.

[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.

[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. URL: https://pubs.er.usgs.gov/publication/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.

[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.

[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.

[Savric2018] Šavrič, B., Patterson, T., and Jenny, B. The equal earth map projection. *International Journal of Geographical Information Science*, 2018. URL: https://www.researchgate.net/profile/Bojan_Savric2/publication/326879978_The_Equal_Earth_map_projection/links/5b69d0ae299bf14c6d951b77/The-Equal-Earth-map-projection.pdf, doi:10.1080/13658816.2018.1504949.

INDEX