PROJ.4 Documentation

Release 5.0.0

Gerald Evenden

Dec 07, 2017

CONTENTS

PROJ.4 is a standard UNIX filter function which converts geographic longitude and latitude coordinates into cartesian coordinates (and vice versa), and it is a C API for software developers to include coordinate transformation in their own software. PROJ.4 is maintained on GitHub.

Platform	Test Status and Coverage
Travis	
AppVeyor	
Coverage	

Full documentation is available as a single PDF at https://raw.githubusercontent.com/OSGeo/proj.4/gh-pages/proj4.pdf

CHAPTER

ONE

DOCUMENTATION

Using PROJ.4

The main purpose of PROJ.4 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.

Quick start

Coordinate transformations are defined by, what in PROJ.4 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.4 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 responce 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.4 also comes bundled with the `cs2cs` utility which is used to transform from onecoordinate 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 seperates the source and destination projection definitions.

If you happen to know the EPSG identifiers for the two cordinates 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

Applications

Bundled with PROJ.4 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.

proj

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

Synopsis

```
proj [ -bcCeEfiIlmorsStTvVwW [ args ] ] [ +args ] file[s]
invproj [ -bcCeEfiIlmorsStTwW [ args ] ] [ +args ] file[s]
```

Description

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 son process and allows␣
↪bypassing
        formatting operations.

-i      Selects binary input only (see -b option).

-C      Check. Invoke all built in self tests and report. Get more verbose report by
        preceding with the -V option).

-I      alternate method to specify inverse projection. Redundant when used with␣
↪invproj.

-o      Selects binary output only (see -b option).

-ta     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
       String is an arbitrary string to be output if an error is detected during data
       transformations. The default value is: *\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[p|P|=|e|u|d]id
       List projection identifiers with -l, -lp or -lP (expanded) that can be selected
       with +proj. -l=id gives expanded description of projection id. List
       ellipsoid identifiers with -le, that can be selected with +ellps, -lu list of
       cartesian to meter conversion factors that can be selected with +units or -ld
       list of datums that can be selected with +datum.

-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 meridinal 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 meridinal 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
       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|W]n
       N is the number of significant fractional digits to employ for seconds output␣
↪(when
       the  option  is  not  specified, -w3 is assumed). When -W is employed the␣
↪fields
       will be constant width and  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. Res is an integer number specifying the power of 10 precision of
       the approximation. For example, a res of -3 specifies an approximation with an
       accuracy better than .001. Umax, and vmax specify maximum degree of the␣
↪polynomials
       (default: 15).
```

The `+args` run-line arguments are associated with cartographic parameters. Usage varies with projection and for a complete description consult the projection pages

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 trans- formed. A - will specify the location of pro- cessing standard input. If no files are speci- fied, 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 geo- graphic 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.

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 stan- dard 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
```

cs2cs

`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.

Synopsis

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

Description

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.

-ta     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
        String is an arbitrary string to be output if an error is detected  during ␣
↪data
        transformations. The  default value is: *\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[p|P|=|e|u|d]id
        List projection identifiers with -l, -lp or -lP (expanded) that can be selected
        with +proj. -l=id gives expanded description of projection id. List
        ellipsoid identifiers with -le, that can be selected with +ellps,-lu list  of
        cartesian to meter conversion factors that can be selected with +units or -ld
        list of datums that can be selected with +datum.

-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
        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␣
↪deci-
        mal  degrees. The default format is "%.2f" for forward projection and DMS for
        inverse.
```

```
-[w|W]n
       N is the number of significant fractional digits to employ for seconds output␣
↪(when
       the option is not specified, -w3 is assumed). When -W is employed the fields
       will be constant width and 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 `+args` run-line arguments are associated with cartographic parameters. Usage varies with projection and for a complete description consult the projection pages

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.

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:

```
1402285.99      5076292.42 0.000
```

geod

`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 call geod with the -I option.

Synopsis

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

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

Description

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.

-ta     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|F] format
        Format is a printf format string to control the output
        form of the geographic coordinate values (f) or distance
        value (F). The default mode is DMS for geographic
        coordinates and "%.3f" for distance.

-[w|W]n
        N is the number of significant fractional digits to
        employ for seconds output (when the option is not
        specified, -w3 is assumed). 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. See `proj` documentation for full list of these parameters and 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.

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 loca- tion 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.

Further reading

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

Cartographic projection

The foundation of PROJ.4 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.4 library.

Below is a list of PROJ.4 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.

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.

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.

Longitude Wrapping

By default PROJ.4 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`.

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 clared 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                             <i>(input)</i>
3d41'16.48"E    0dN 0.000       <i>(output)</i>
```

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.

Geodetic transformation

PROJ.4 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.4 has over time evolved into a powerfull 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 geodetec transformations of varying complexity can be performed.

In PROJ.4, 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.4 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.

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.4 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 itroduction). In PROJ.4 it can be implemented as

```
proj=pipeline
step proj=cart ellps=intl
step proj=helmert
     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
      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
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.4 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 *[AltamimiEtAl2002]*, except the epoch which is the epoch of the transformation. The default setting is to use "coordinate frame" convention of the Helmert transform, but "position vector" convention can also be used. 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
     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
     epoch=1988.0
step proj=unitconvert t_in=decimalyear t_out=gps_week
```

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-lineaer planar correction derived from interpolation in a correction grid can be applied. Originally this was implemented as a means to transform coordinates between the 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
```

Grid Based Datum Adjustments

In many places (notably North America and Austrialia) 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.4 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.4 `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 (see *Non-Free Grids*), 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 *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.

Resource files

A number of files containing preconfigured transformations and default parameters for certain projections are bundled with the PROJ.4 distribution. Init files contains preconfigured proj-strings for various coordinate reference systems and the defaults file contains default values for parameters of select projections.

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 necesarily 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.4 but it is also possible to add your own custom init files. PROJ.4 looks for the init files in the directoty 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 obervation epoch when transforming from ITRF2000 to ITRF2005:

```
+init=ITRF2000:ITRF2005 +tobs=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
+epoch=2000.0 +transpose
+tobs=2010.5
```

Below is a list of the init files that are packaged with PROJ.4.

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

Defaults file

The `proj_def.dat` file supplies default parameters for PROJ.4. 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.4 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.

Environment variables

PROJ.4 can be crontrolled 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.4 *resource files*. It is only possible to specify a single library in *PROJ_LIB*; e.g. it does

not behave like PATH. PROJ.4 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.4. The default debug level is zero, which results in no debug output when using PROJ.4. A number from 1-3, whit 3 being the most verbose setting.

Coordinate operations

Coordinate operations in PROJ.4 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.4 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.

Projections

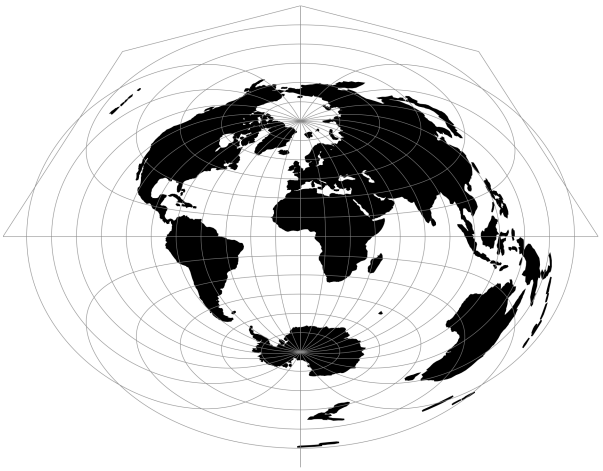
Projections are coordinate operations that are technically conversions but since projections are so fundamental to PROJ.4 we differentiate them from conversions.

Projections map the spherical 3D space to a flat 2D space.

Azimuthal Equidistant

Classification	Azimuthal
Available forms	Forward and inverse, spherical and elliptical projection
Implemented by	Gerald I. Evenden
Options	
+guam	Use Guam elliptical formulas. Defaults to false.

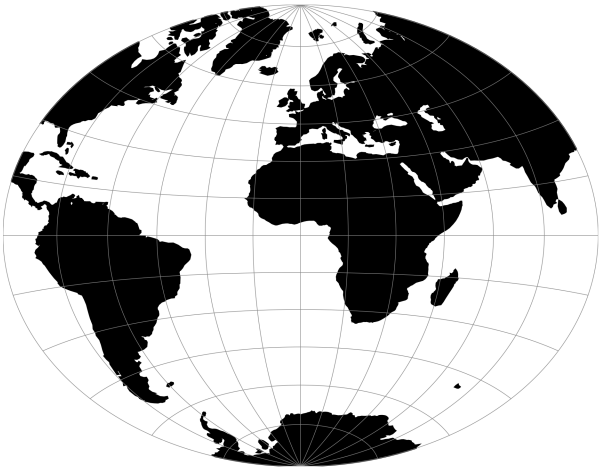
+proj=aeqd



Airy

Classification	
Available forms	Forward spherical projection
Defined area	Global
Implemented by	Gerald I. Evenden
Options	
+lat_b	
+no_cut	Do not cut at hemisphere limit

+proj=airy

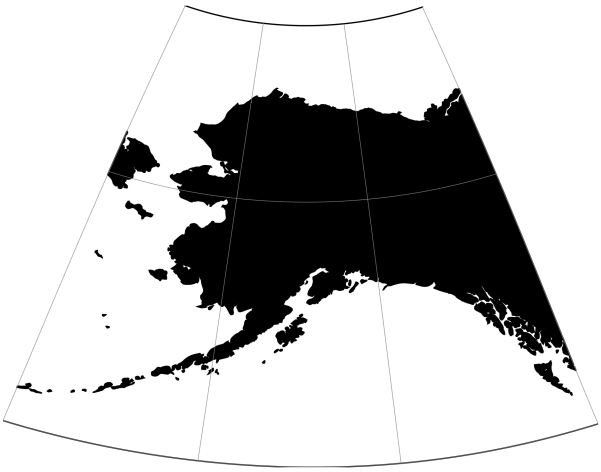


Aitoff

+proj=aitoff

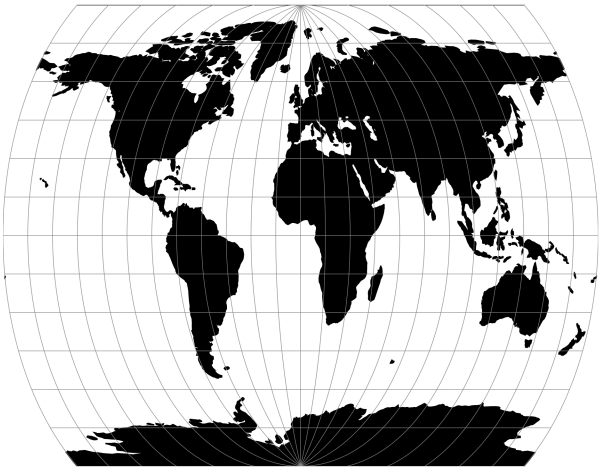
Mod. Stererographics of Alaska

+proj=alsk



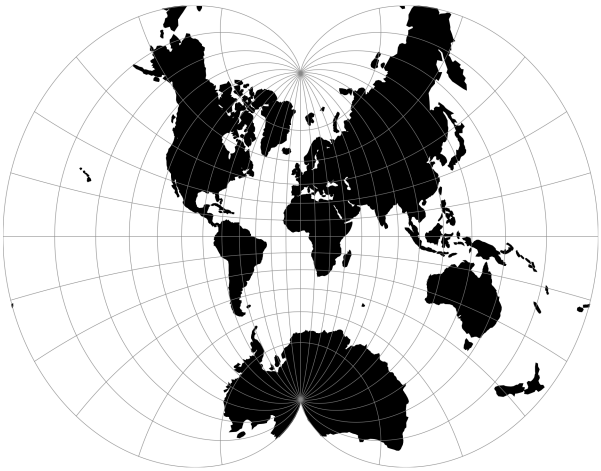
Apian Globular I

+proj=apian



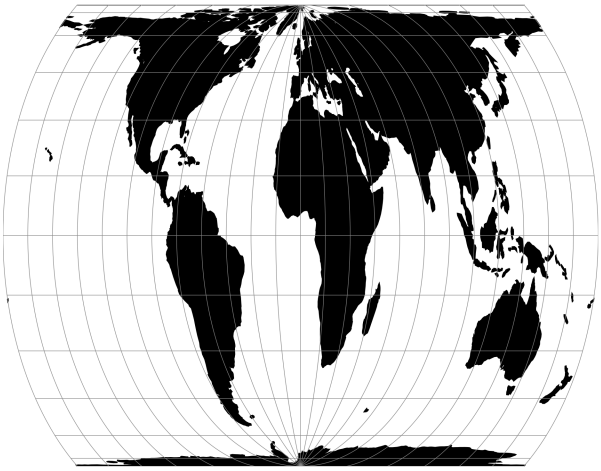
August Epicycloidal

+proj=august



Bacon Globular

+proj=bacon



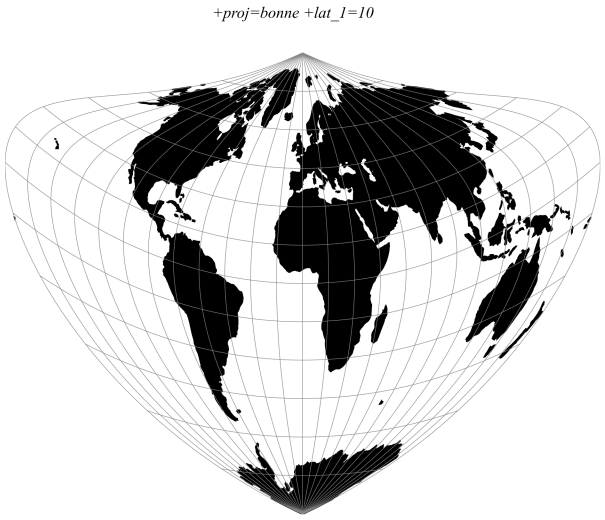
Bipolar conic of western hemisphere

+proj=bipc +ns

Boggs Eumorphic

+proj=boggs

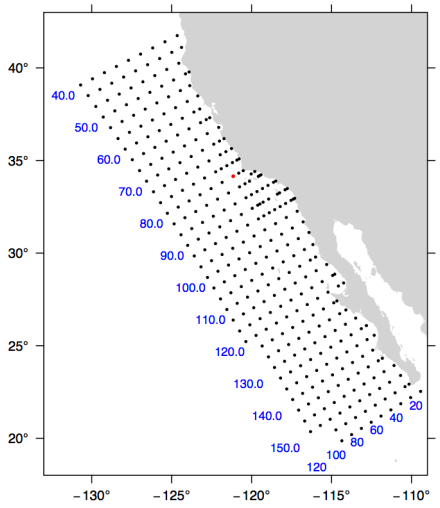
Bonne (Werner lat_1=90)



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
Implemented by	Frank Warmerdam
Options	
No special options for this projection	



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
```

Mathematical definition

The algorithm used to make conversions is described in *[EberHewitt1979]* with a few corrections reported in *[WeberMoore2013]*.

Further reading

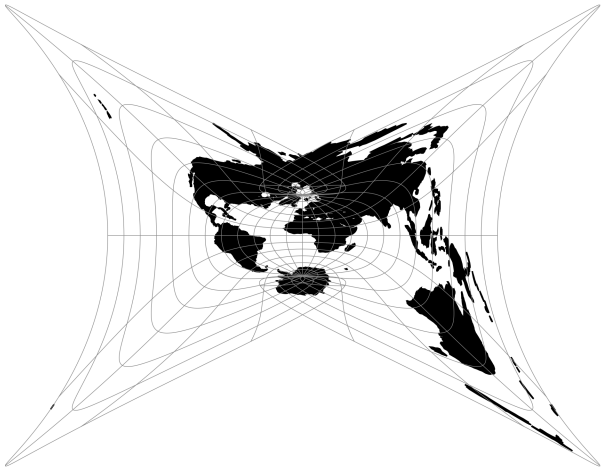
1. General information about the CalCOFI program
2. The Investigaciones Mexicanas de la Corriente de California

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
Implemented by	Gerald I. Evenden
Options	
+lat_0	Latitude of origin (Default to 0)

+proj=cass



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
```

Mathematical definition

The formulas describing the Cassini projection are taken from Snyder's *[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 meridionial 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

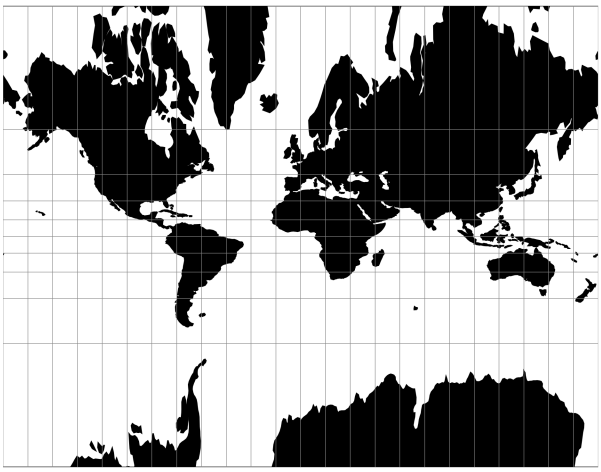
$$R = (1 - e^2)(1 - e^2 sin^2\phi')^{-3/2}$$
$$D = x/N$$
$$\phi = \phi' - \tan\phi'\frac{N}{R}(\frac{D^2}{2} - (1 + 3T)\frac{D^4}{24})$$
$$\lambda = \frac{(D - T\frac{D^3}{3} + (1 + 3T)T\frac{D^5}{15})}{\cos\phi'}$$

Further reading

1. Wikipedia
2. *[Snyder1987]*
3. EPSG, POSC literature pertaining to Coordinate Conversions and Transformations including Formulas

Central Cylindrical

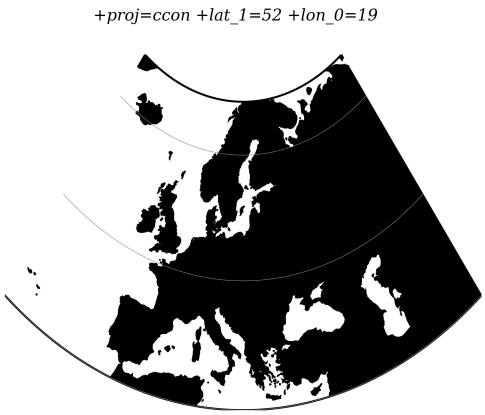
+proj=cc



Central Conic

This is central (centrographic) projection on cone tangent at `lat_0` 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
Implemented by	Lukasz Komsta
Options	
+lat_1	Latitude of standard parallel.



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 +lat_0=52 +lon_0=19 +axis=esu +a=6390000 +x_0=330000 +y_0=-350000
```

For more information see *[Komsta2016]* and *[Verey2017]*.

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:

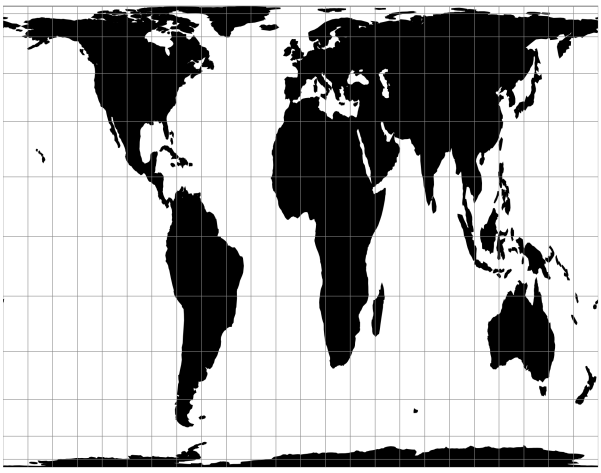
```
cat << EOF  | src/cs2cs -v -f "%E" +proj=longlat +datum=WGS84 +no_defs +to +proj=ccon␣
↪+lat_1=52 +lat_0=52 +lon_0=19 +axis=esu +a=6390000 +x_0=330000 +y_0=-350000
24 55
15 49
24 49
19 52
EOF
```

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
```

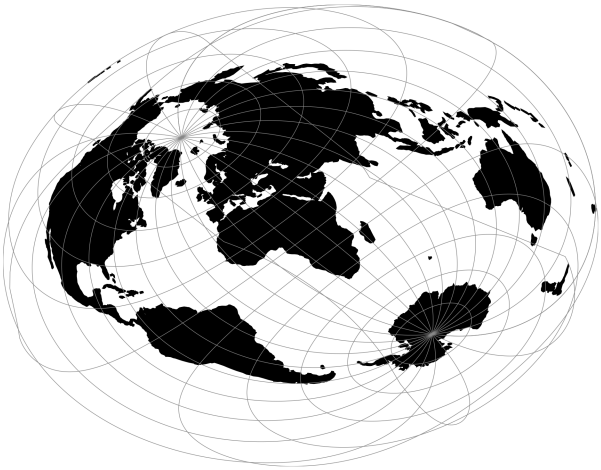
Equal Area Cylindrical

+proj=cea



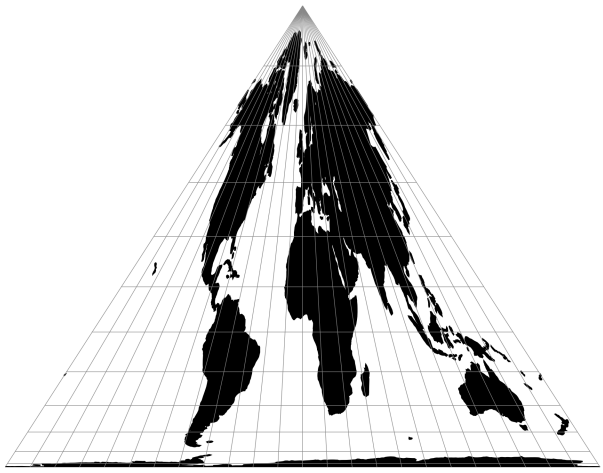
Chamberlin Trimetric

+proj=chamb +lat_1=10 +lon_1=30 +lon_2=40



Collignon

+proj=collg

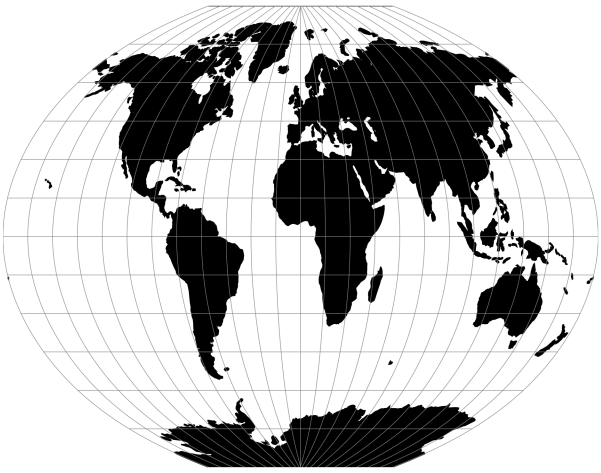


Craster Parabolic (Putnins P4)

+proj=crast

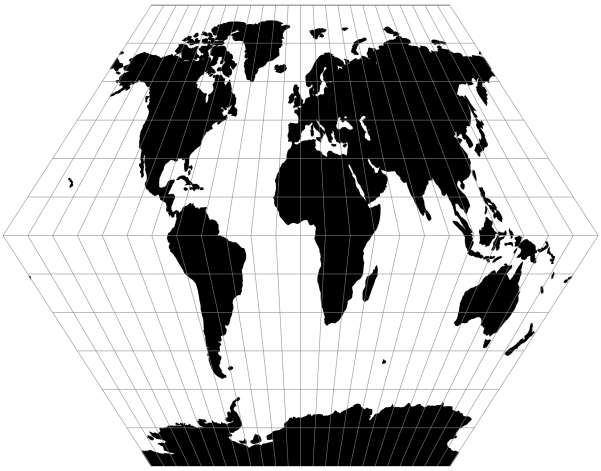
Denoyer Semi-Elliptical

+proj=denoy



Eckert I

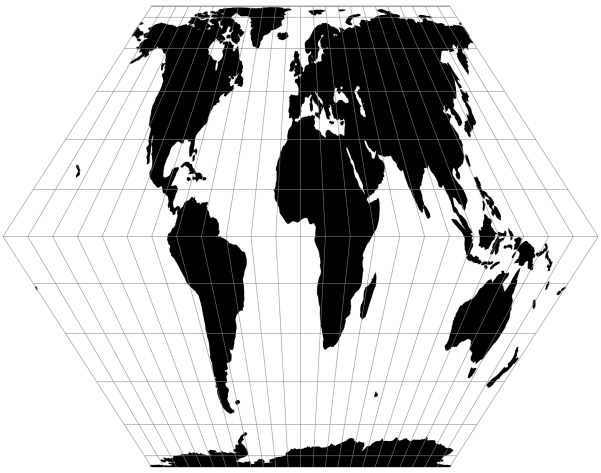
+proj=eck1



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

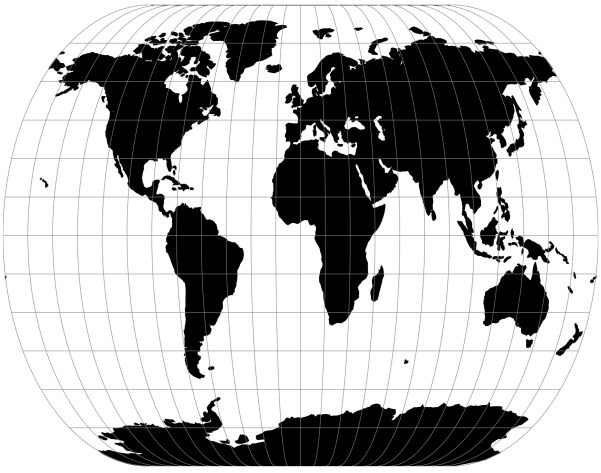
Eckert II

+proj=eck2



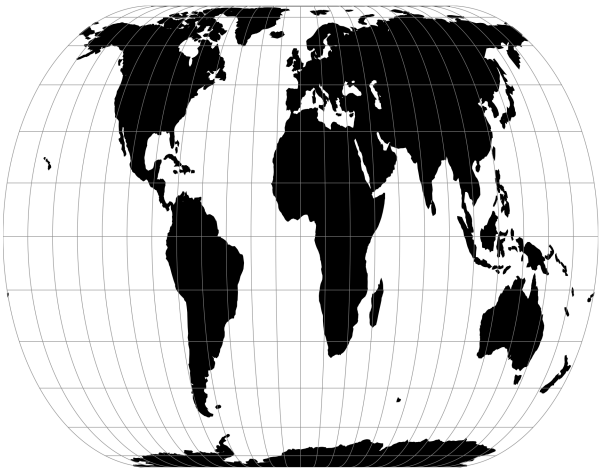
Eckert III

+proj=eck3



Eckert IV

+proj=eck4

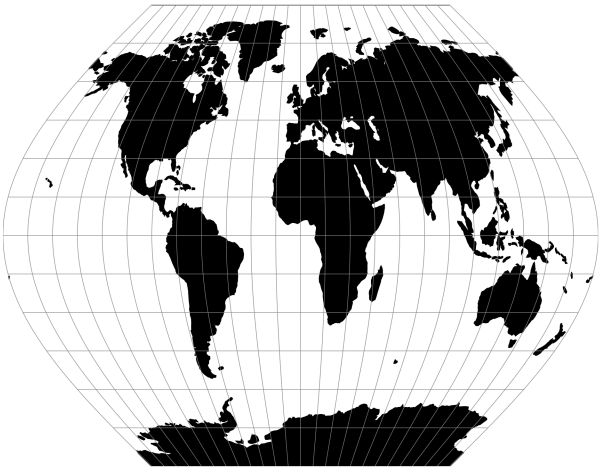


$$x = \lambda(1 + cos\phi)/\sqrt{2 + \pi}$$

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

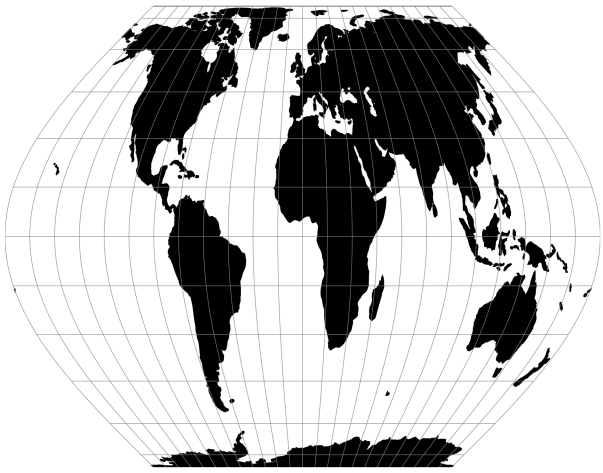
Eckert V

+proj=eck5



Eckert VI

+proj=eck6

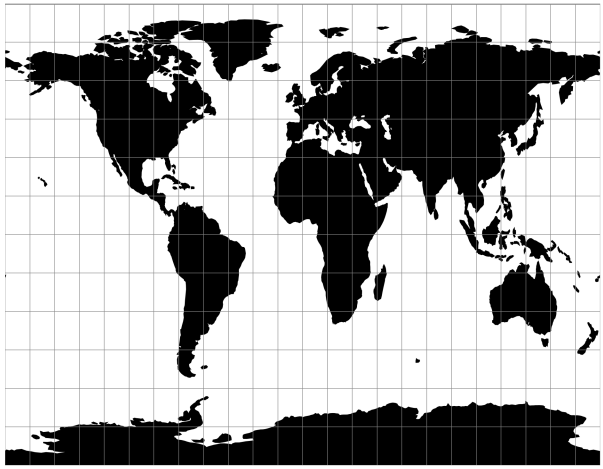


Equidistant Cylindrical (Plate Carrée)

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
Implemented by	Gerald I. Evenden
Options	
+lat_ts	Latitude of true scale. Defaults to 0.0
+lat_0	Center of the map : latitude of origin

+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 +lat_0=90w
-8483684.61     13358338.90
```

Mathematical definition

The formulas describing the Equidistant Cylindrical projection are all taken from Snyder's *[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

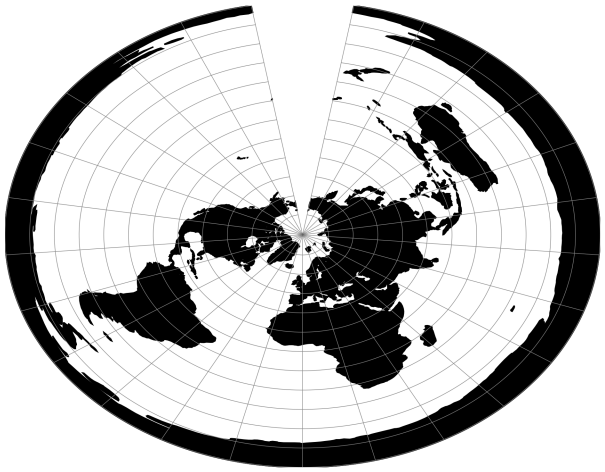
1. Wikipedia
2. Wolfram Mathworld

Equidistant Conic

+proj=eqdc +lat_1=55 +lat_2=60

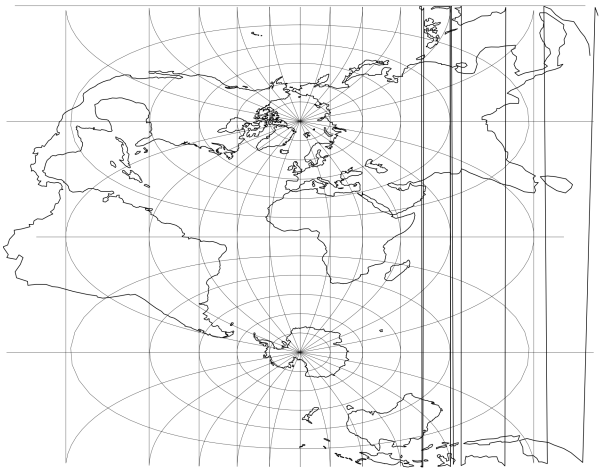
Euler

+proj=euler +lat_1=67 +lat_2=75



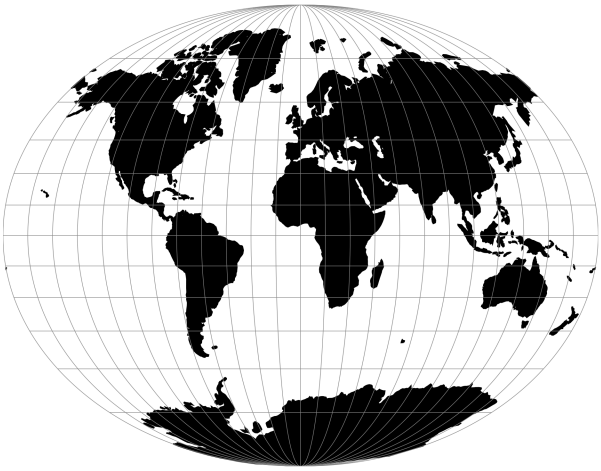
Extended Transverse Mercator

+proj=etmerc

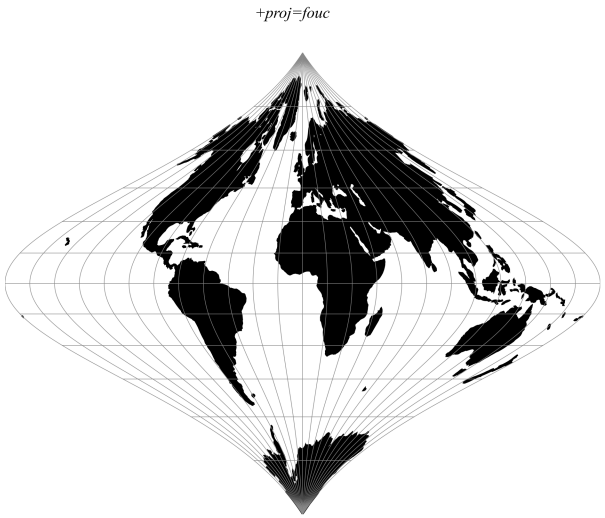


Fahey

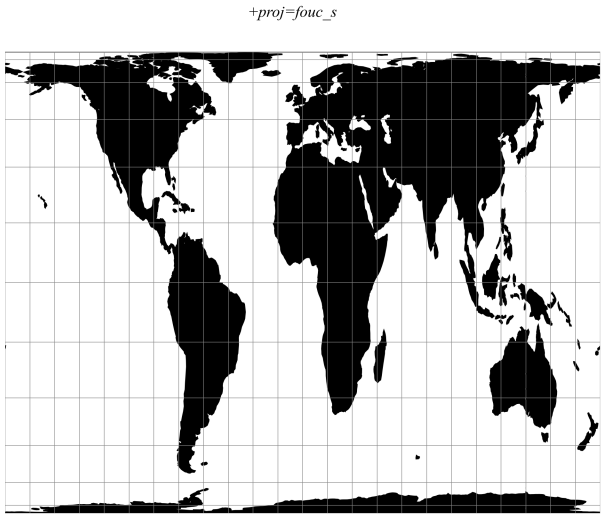
+proj=fahey



Foucaut



Foucaut Sinusoidal



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 $0 <= n <= 1$.

$$x = \lambda \cos\phi / (n + (1 - n)\ cos\phi)$$
$$y = n\phi + (1 - n) \sin\phi$$

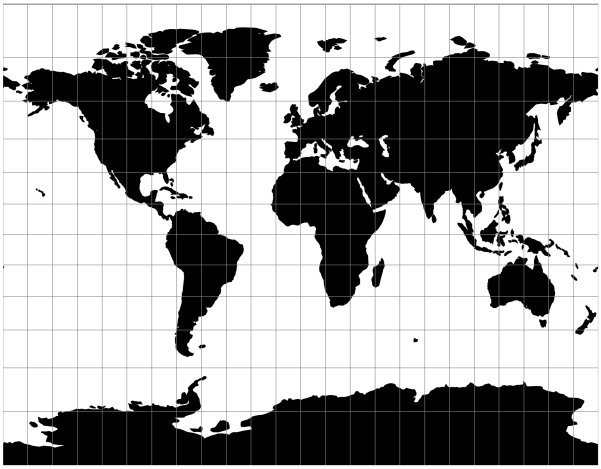
For the inverse, the Newton-Raphson method can be used to determine ϕ from the equation for y above. As $n \to 0$ and $\phi \to \pi/2$, convergence is slow but for $n = 0$, $\phi = \sin^1 y$

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
Implemented by	Gerald I. Evenden
Options	No special options for this projection

+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
```

Mathematical definition

The formulas describing the Gall Stereographical are all taken from Snyder's *[Snyder1993]*.

Spherical form

Forward projection

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

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

Inverse projection

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

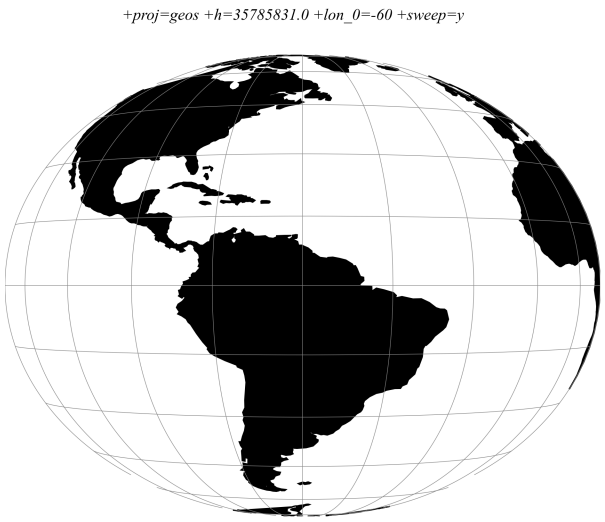
$$\lambda = \sqrt{2}x$$

Further reading

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

Geostationary Satellite View

Classification	Azimuthal
Available forms	Forward and inverse, spherical and elliptical projection
Defined area	Global
Implemented by	Gerald I. Evenden and Martin Raspaud$
Options	
+h	Satellite height above earth. Required.
+sweep	Sweep angle axis of the viewing instrument. Valid options are `x` and `y`. Defaults to `y`.
+lon_0	Subsatellite longitude point.



The geos projection pictures how a geostationary satellite scans the earth at regular scanning angle intervals.

Usage

In order to project using the geos projection you can do the following:

```
proj +proj=geos +h=35785831.0
```

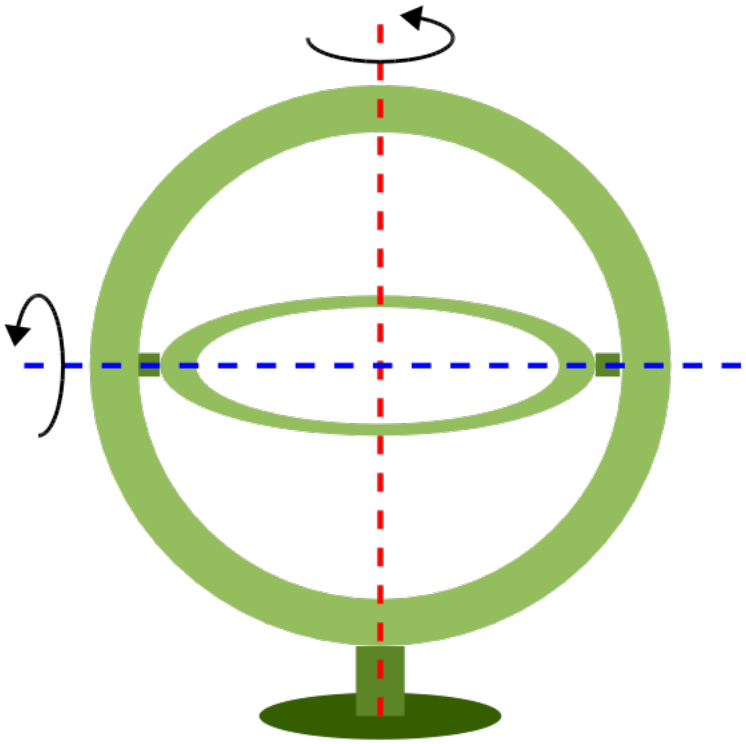
The required argument `h` is the viewing point (satellite position) height above the earth.

The projection coordinate relate to the scanning angle by the following simple relation:

```
scanning_angle (radians) = projection_coordinate / h
```

Note on sweep angle

The viewing instrument on-board geostationary satellites described by this projection have a two-axis gimbal viewing geometry. This means that the different scanning positions are obtained by rotating the gimbal along a N/S axis (or `y`) and a E/W axis (or `x`).



In the image above, the outer-gimbal axis, or sweep-angle axis, is the N/S axis (`y`) while the inner-gimbal axis, or fixed-angle axis, is the E/W axis (`x`).

This example represents the scanning geometry of the Meteosat series satellite. However, the GOES satellite series use the opposite scanning geometry, with the E/W axis (`x`) as the sweep-angle axis, and the N/S (`y`) as the fixed-angle axis.

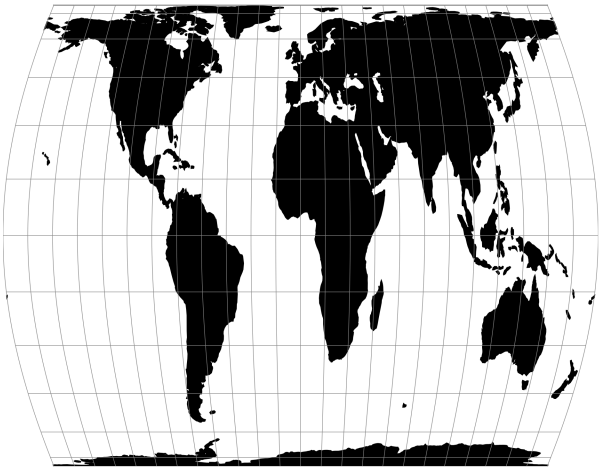
The sweep argument is used to tell PROJ.4 which on which axis the outer-gimbal is rotating. The possible values are x or y, y being the default. Thus, the scanning geometry of the Meteosat series satellite should take sweep as x, and GOES should take sweep as y.

Ginsburg VIII (TsNIIGAiK)

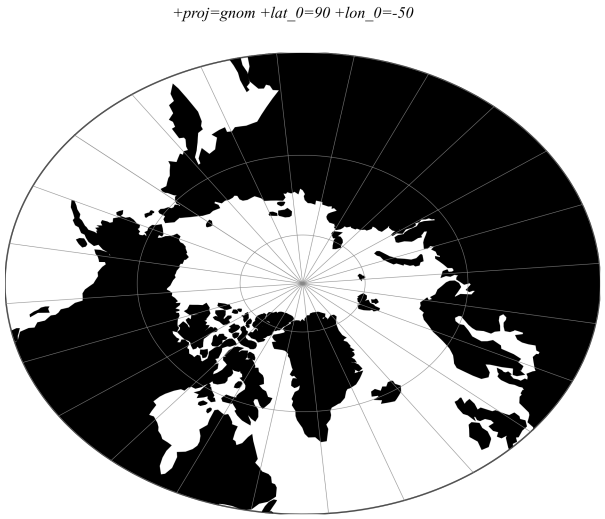
+proj=gins8

General Sinusoidal Series

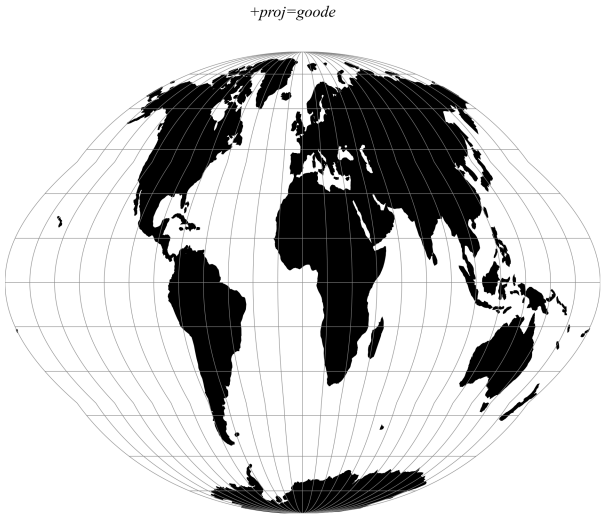
+proj=gn_sinu +m=2 +n=3



Gnomonic

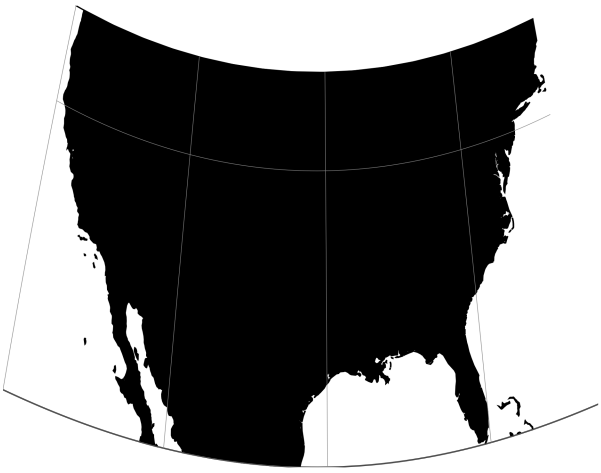


Goode Homolosine



Mod. Stererographics of 48 U.S.

+proj=gs48

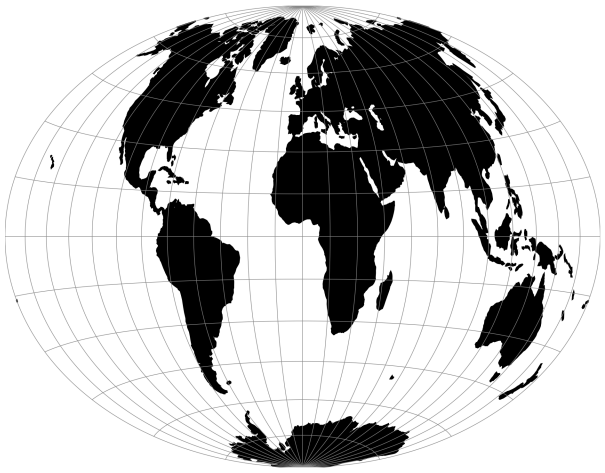


Mod. Stererographics of 50 U.S.

+proj=gs50

Hammer & Eckert-Greifendorff

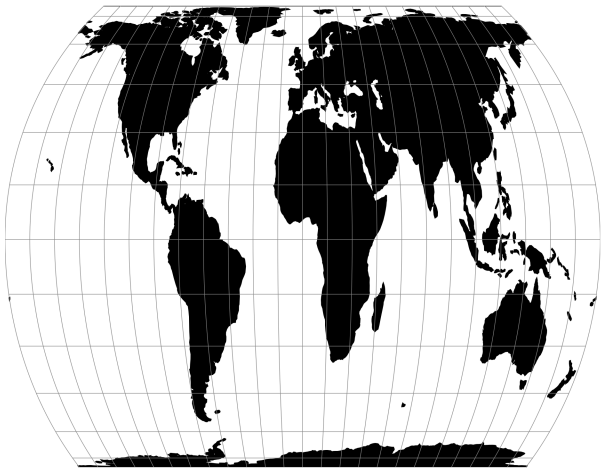
+proj=hammer



Hatano Asymmetrical Equal Area

Classification	*Pseudocylindrical Projection*
Available forms	Forward and inverse, spherical projection
Defined area	Global, but best between standard parallels
Implemented by	Gerald I. Evenden
Options	
+lat_1	Standard Parallel 1
+lat_2	Standard Parallel 2
+sym	Symmetric form used instead of asymmetric

+proj=hatano



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_p	C_p
if `+sym` or $\phi > 0$	1.75859	2.67595
if not `+sym` and $\phi < 0$	1.93052	2.43763

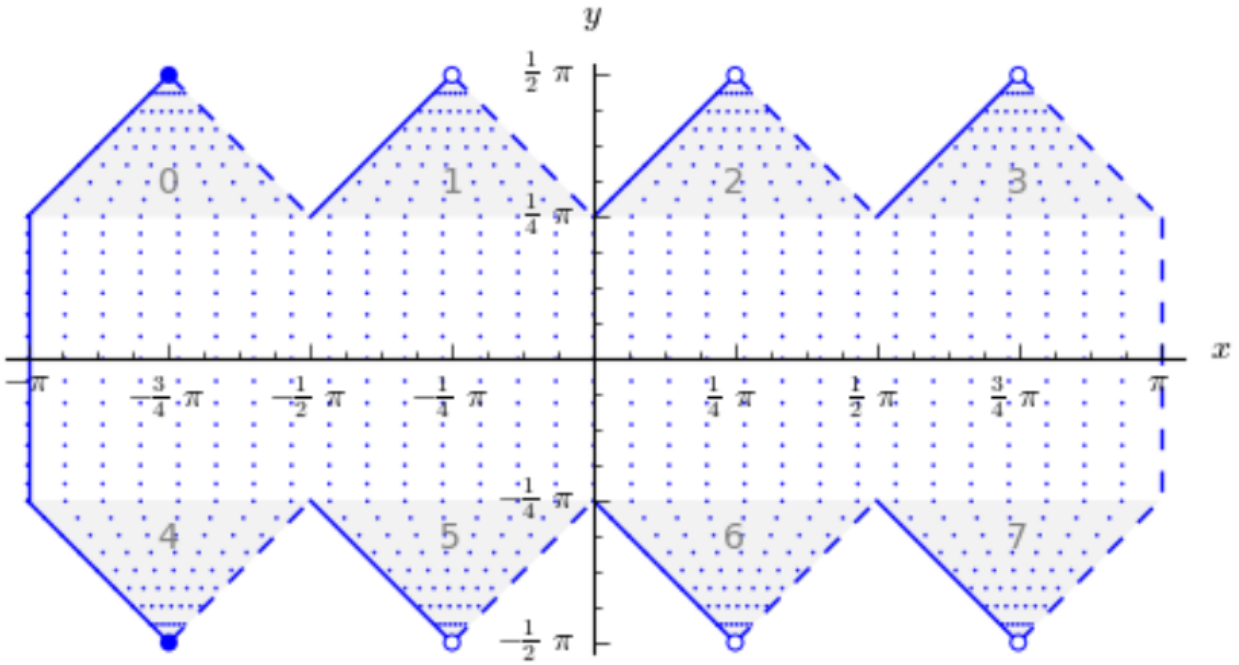
For $\phi = 0$, $y \leftarrow 0$, and $x \leftarrow 0.85\lambda$.

Further reading

1. Compare Map Projections
2. Mathworks

HEALPix

Classification	Mixed
Available forms	Forward and inverse, spherical and elliptical projection
Defined area	Global
Implemented by	Alex Raichev and Michael Speth
Options	
No special options for this projection	



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:

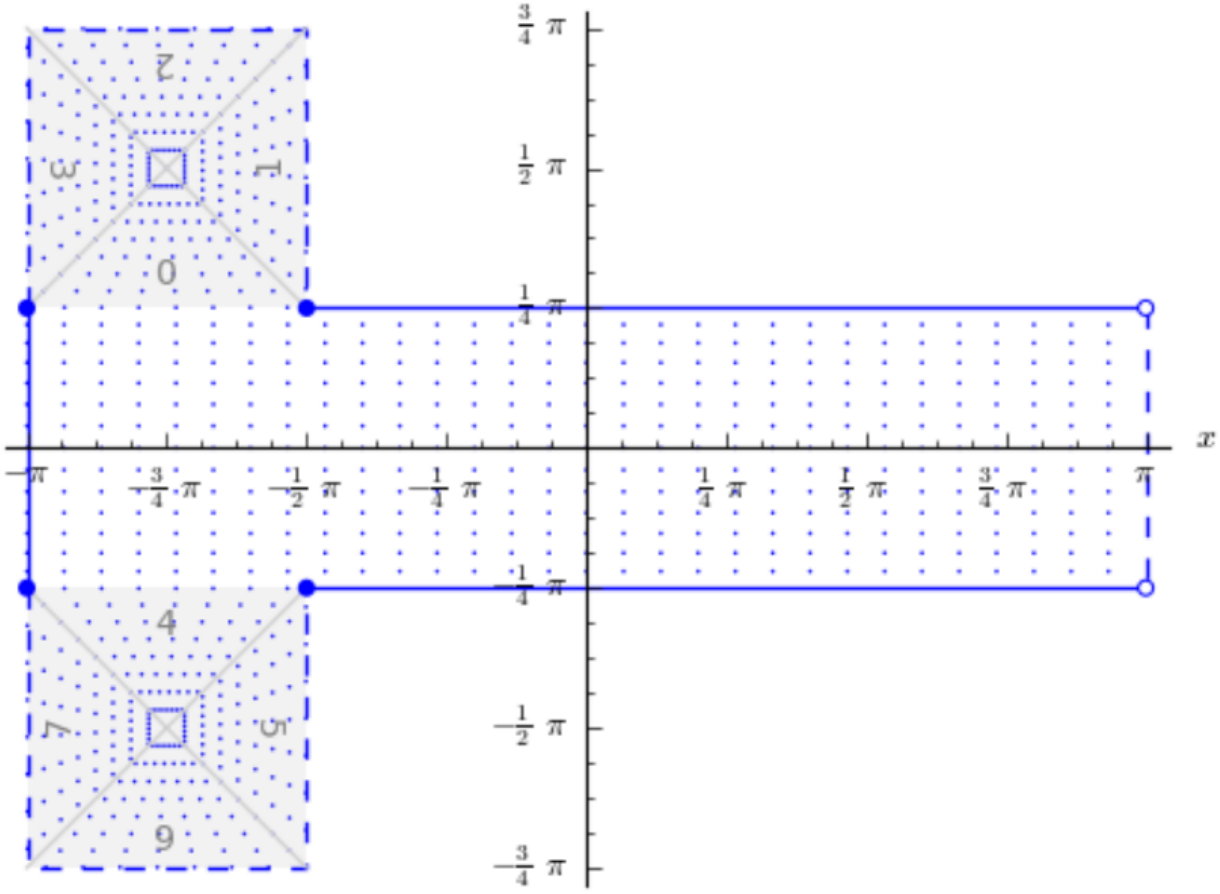
```
proj +proj=healpix +lon_0=0 +a=1 -E <<EOF
0 0
EOF
# output
0 0 0.00 0.00
```

Further reading

1. NASA
2. Wikipedia

rHEALPix

Classification	Mixed
Available forms	Forward and inverse, spherical and elliptical projection
Defined area	Global
Implemented by	Alex Raichev and Michael Speth
Options	
+north_square	Position of the north polar square. Valid inputs are 0–3. Defaults to 0.
+south_square	Position of the south polar square. Valid inputs are 0–3. Defaults to 0.



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 noval approach for the rHEALPix projection. The initial intention of using rHEALPix in the Spatial Computation Engine Science Collaboration Environment (SCENZGrid).

Usage

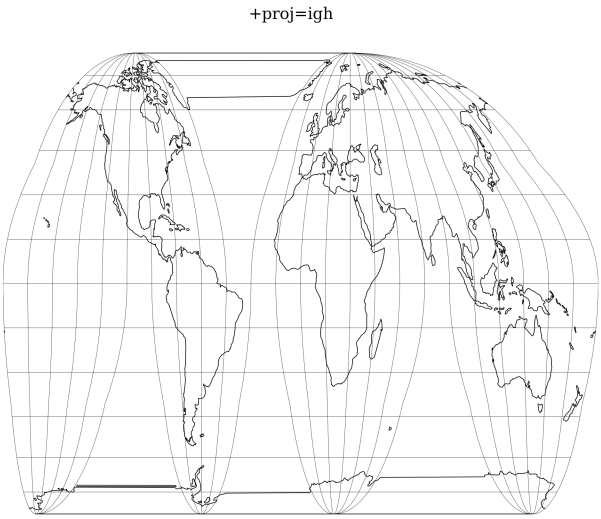
To run a rHEALPix projection on a WGS84 ellipsoidal model, use the following command:

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

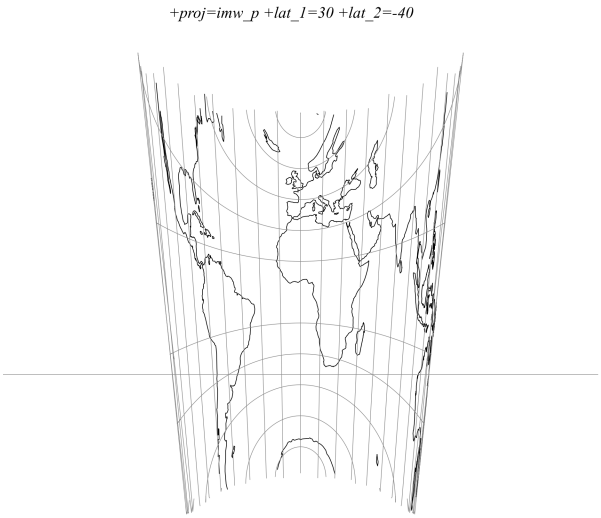
Further reading

1. NASA
2. Wikipedia

Interrupted Goode Homolosine

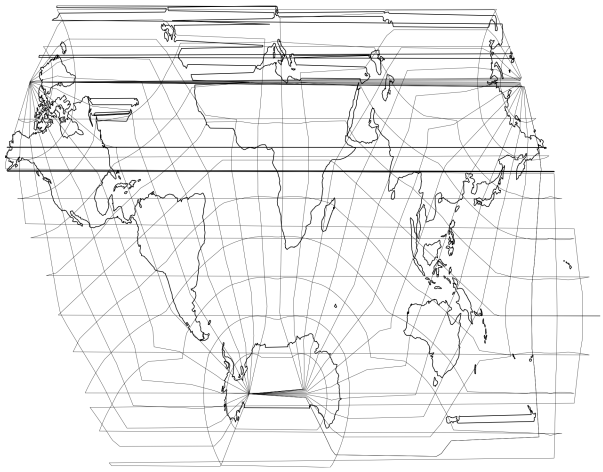


International Map of the World Polyconic



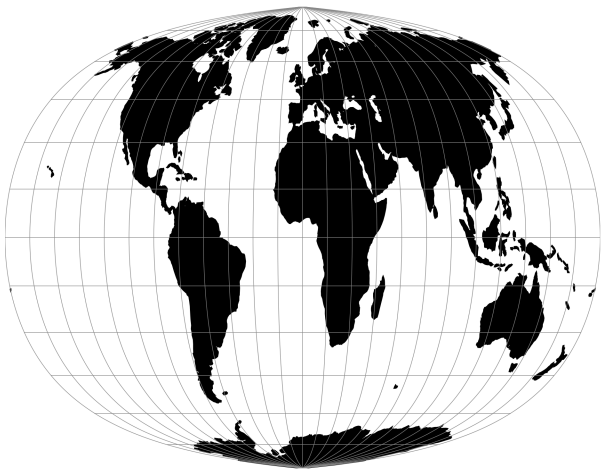
Icosahedral Snyder Equal Area

+proj=isea



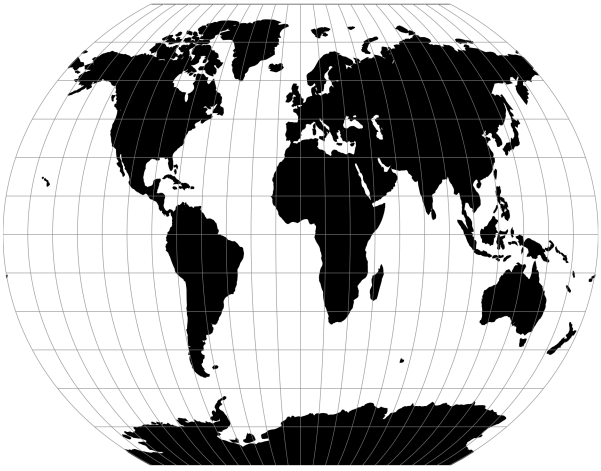
Kavraisky V

+proj=kav5



Kavraisky VII

+proj=kav7

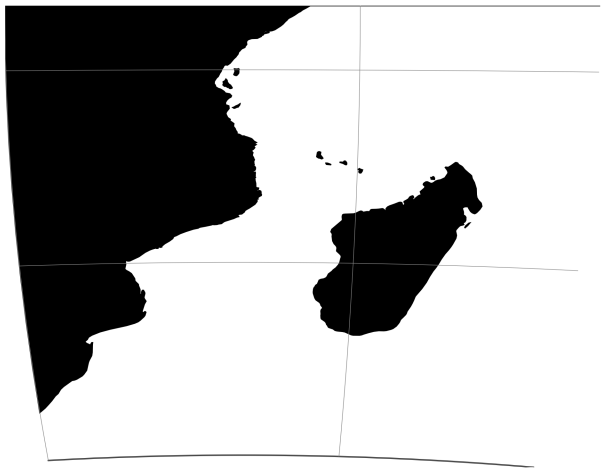


Krovak

+proj=krovak

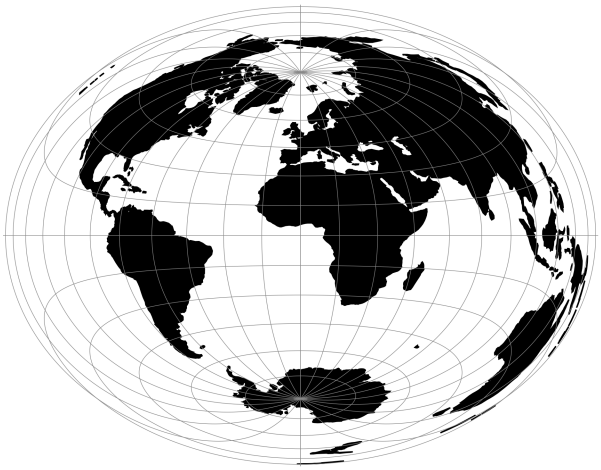
Laborde

+proj=labrd +lon_0=40 +lat_0=-10



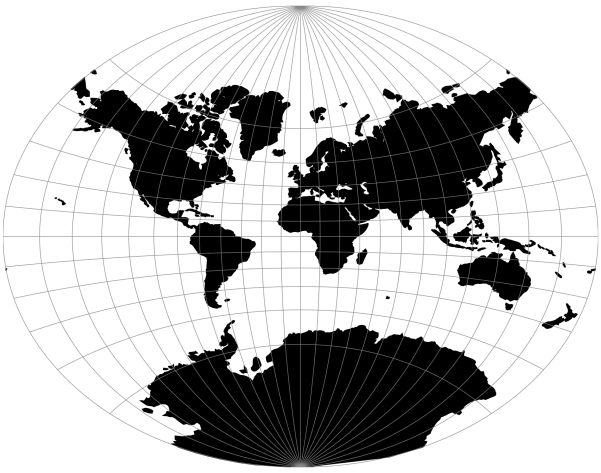
Lambert Azimuthal Equal Area

+proj=laea



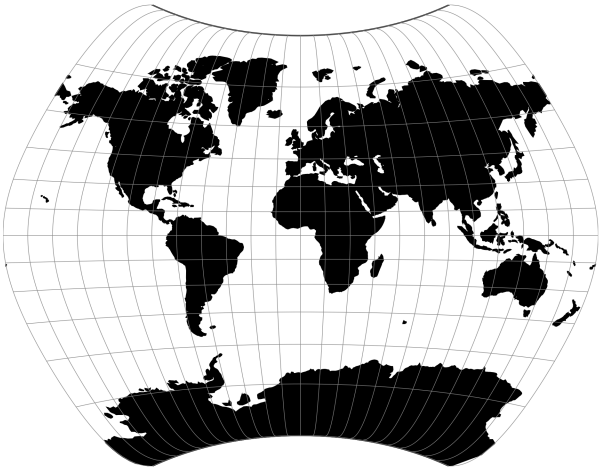
Lagrange

+proj=lagrng



Larrivee

+proj=larr



Laskowski

+proj=lask

Lambert Conformal Conic

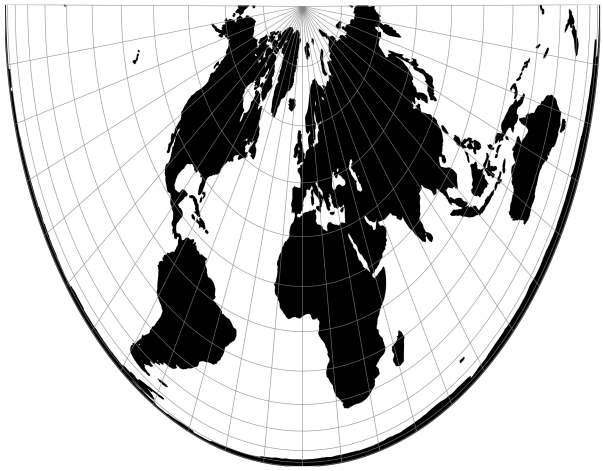
+proj=lcc +lon_0=-90

Lambert Conformal Conic Alternative

+proj=lcca +lat_0=35

Lambert Equal Area Conic

+proj=leac

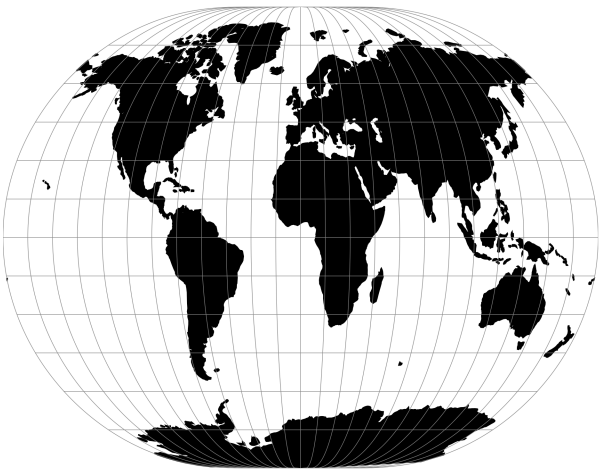


Lee Oblated Stereographic

+proj=lee_os

Loximuthal

+proj=loxim

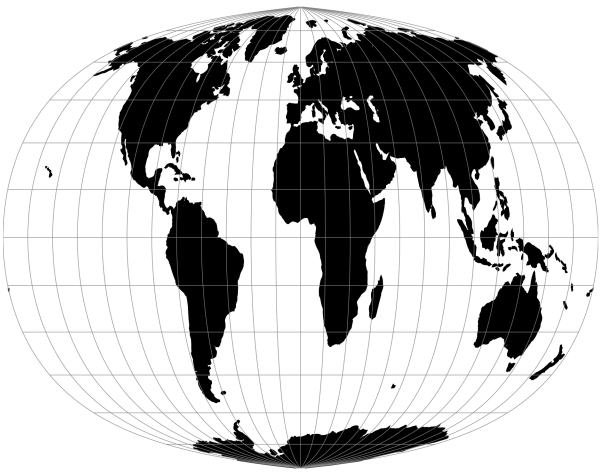


Space oblique for LANDSAT

+proj=lsat +path=2 +lsat=1

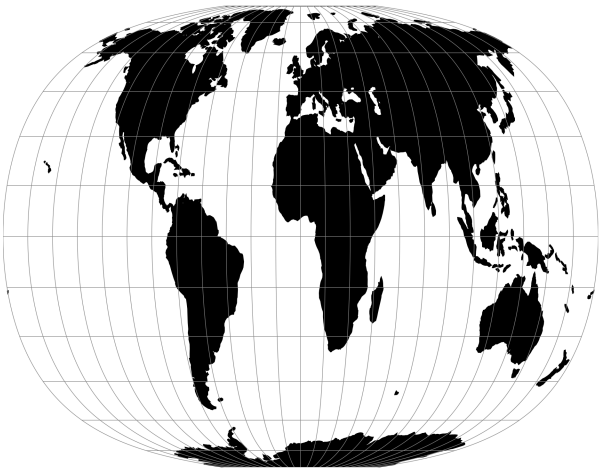
McBryde-Thomas Flat-Polar Sine (No. 1)

+proj=mbt_s



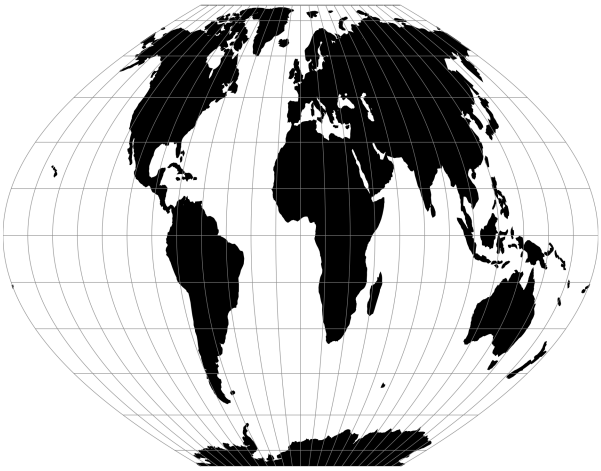
McBryde-Thomas Flat-Pole Sine (No. 2)

+proj=mbt_fps

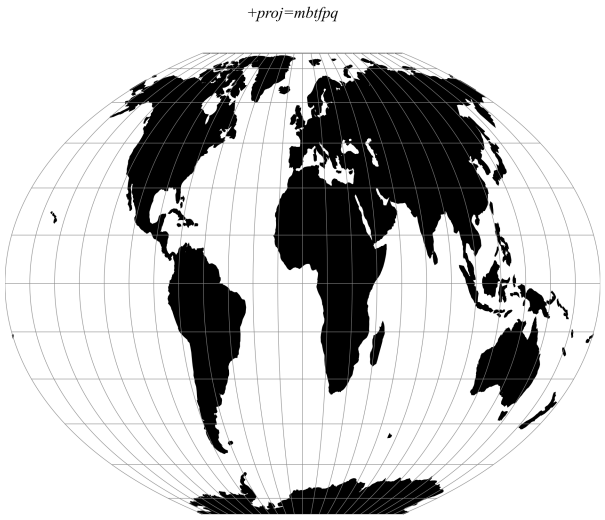


McBride-Thomas Flat-Polar Parabolic

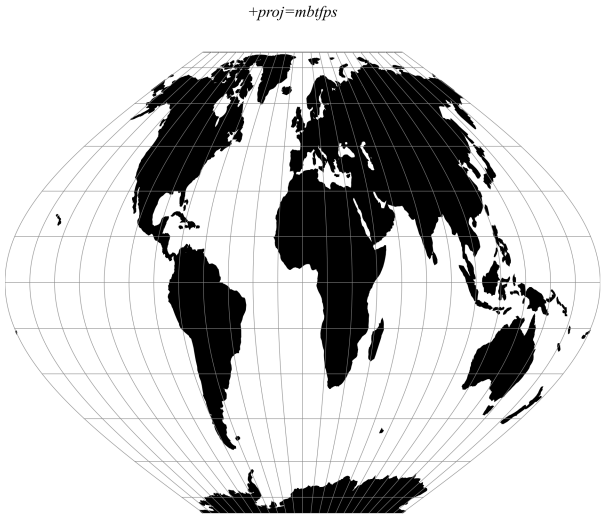
+proj=mbtfpp



McBryde-Thomas Flat-Polar Quartic



McBryde-Thomas Flat-Polar Sinusoidal

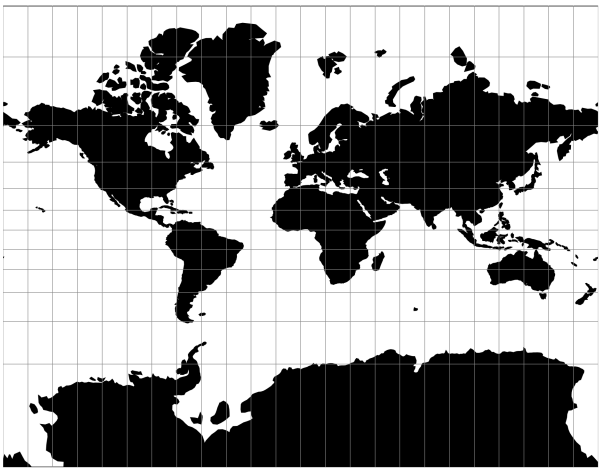


Mercator

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
Implemented by	Gerald I. Evenden
Options	
+lat_ts	Latitude of true scale. Defaults to 0.0
+k_0	Scaling factor. Defaults to 1.0

+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 excusive. If used together, `+lat_ts` takes precedence over `+k_0`.

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:

$$k_0 = m \left(\phi_t s \right)$$

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

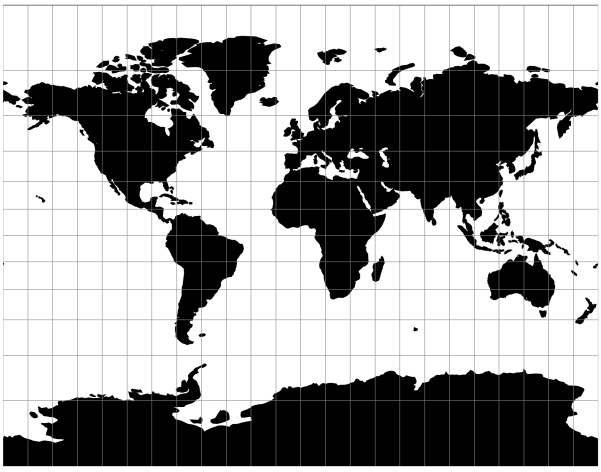
Miller Oblated Stereographic

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
Implemented by	Gerald I. Evenden
Options	
+lat_0	Latitude of origin (Default to 0)

+proj=mill



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
```

Mathematical definition

The formulas describing the Miller projection are all taken from Snyder's manuals *[Snyder1987]*.

Forward projection

$$x = \lambda$$

$$y = 1.25 * \ln\left[\tan\left(\frac{\pi}{4} + 0.4 * \phi\right)\right]$$

Inverse projection

$$\lambda = x$$

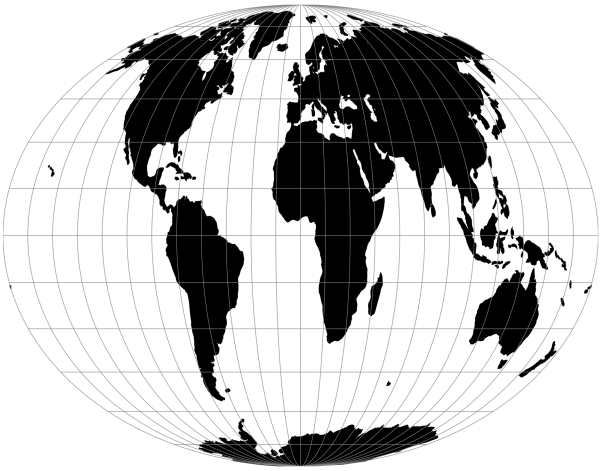
$$\phi = 2.5 * (\arctan\left[e^{0.8*y}\right] - \frac{\pi}{4})$$

Further reading

1. Wikipedia

Mollweide

+proj=moll

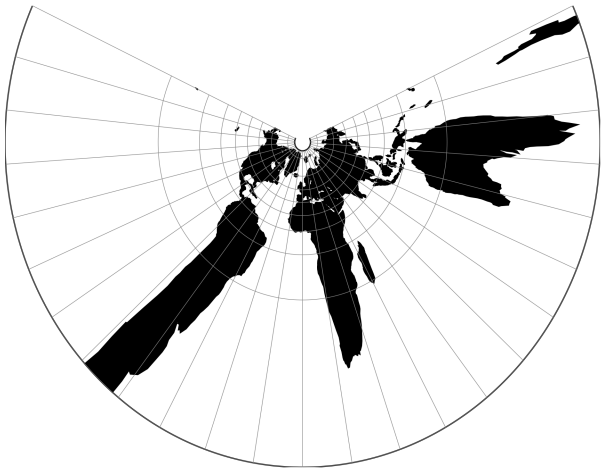


Murdoch I

+proj=murd1 +lat_1=30 +lat_2=50

Murdoch II

+proj=murd2 +lat_1=30 +lat_2=50

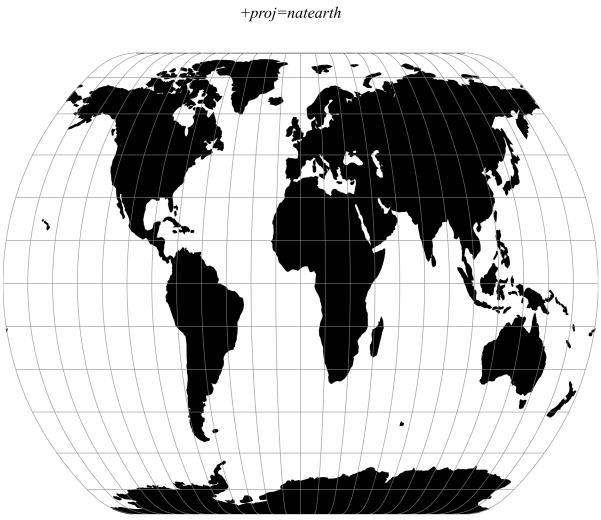


Murdoch III

+proj=murd3 +lat_1=30 +lat_2=50

Natural Earth

Classification	Pseudo cylindrical
Available forms	Forward and inverse, spherical projection
Defined area	Global
Implemented by	Bernhard Jenny
Options	
No special options for this projection	



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:

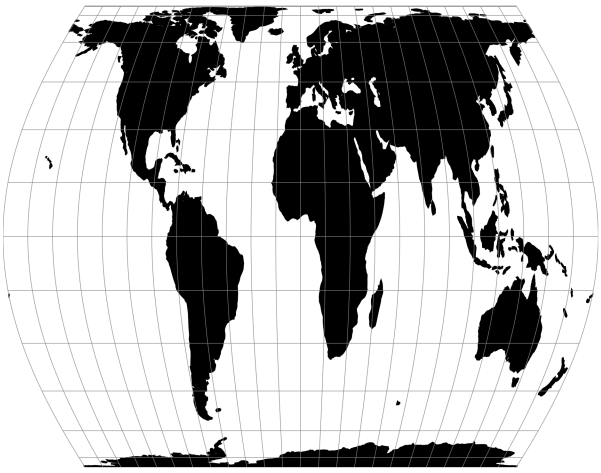
```
$ echo 3500 -8000 | proj -I +proj=natearth +a=7500
37d54'6.091"E  61d23'4.582"S
```

Further reading

1. Wikipedia

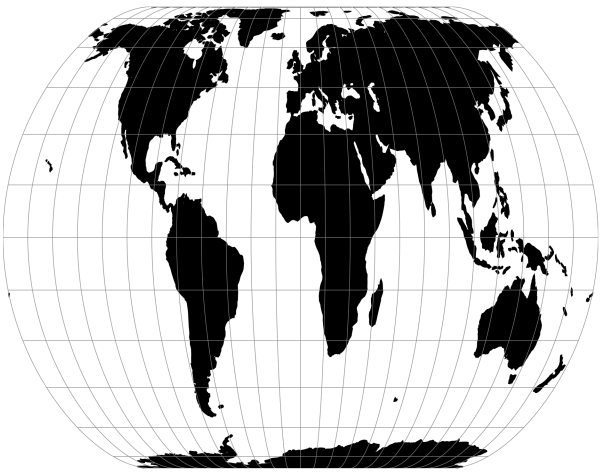
Nell

+proj=nell



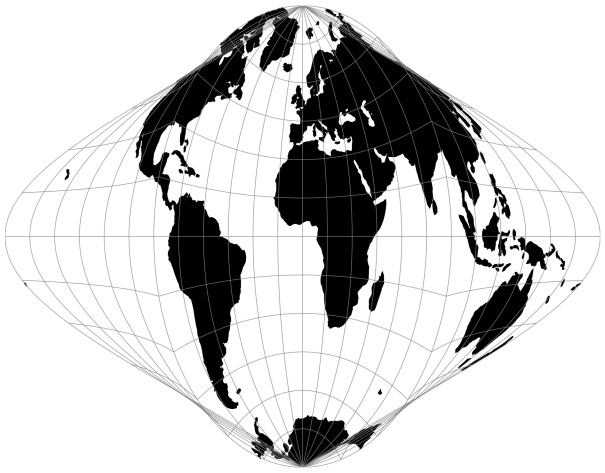
Nell-Hammer

+proj=nell_h



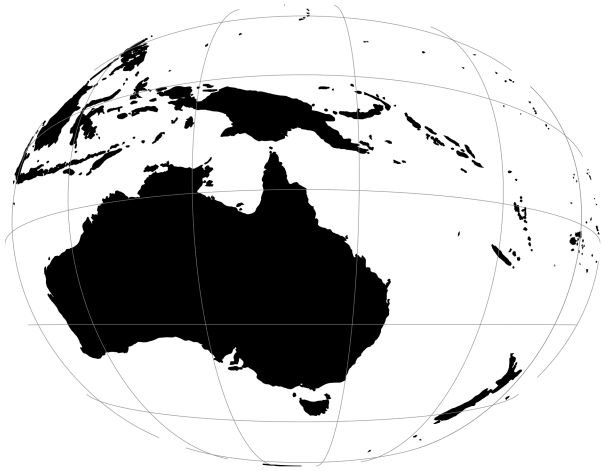
Nicolosi Globular

+proj=nicol



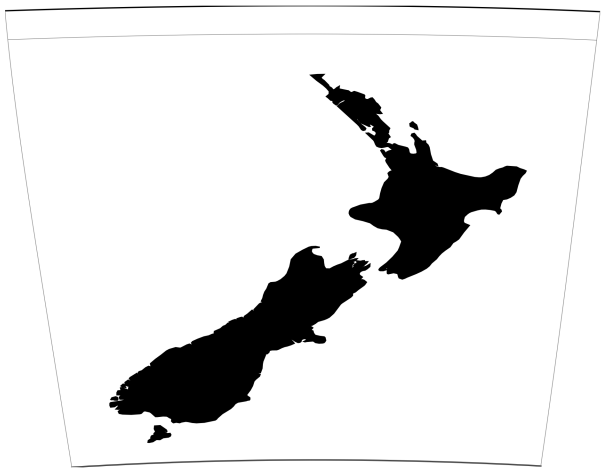
Near-sided perspective

+proj=nsper +h=3000000 +lat_0=-20 +lon_0=145



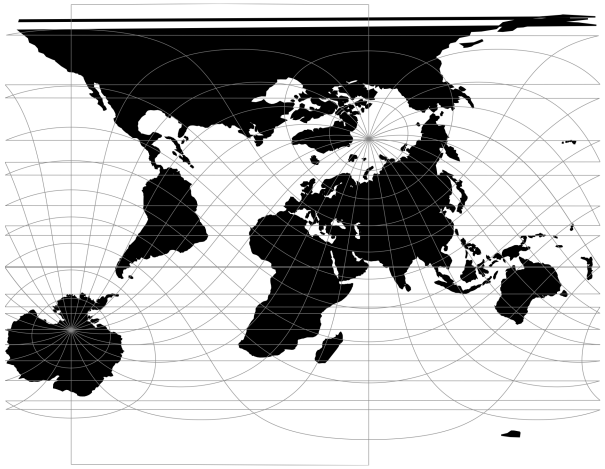
New Zealand Map Grid

+proj=nzmg



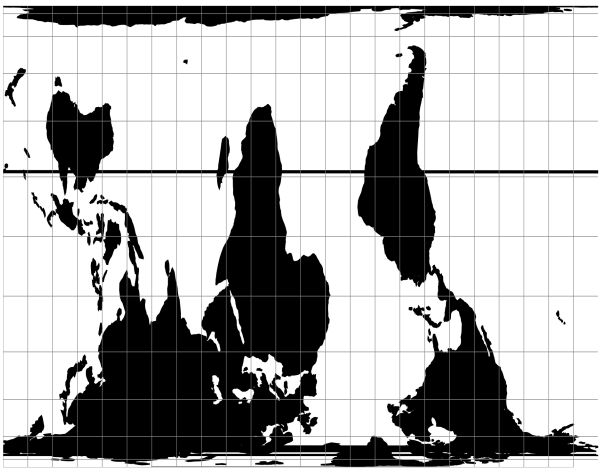
General Oblique Transformation

+proj=ob_tran +o_proj=mill +o_lon_p=40 +o_lat_p=50 +lon_0=60



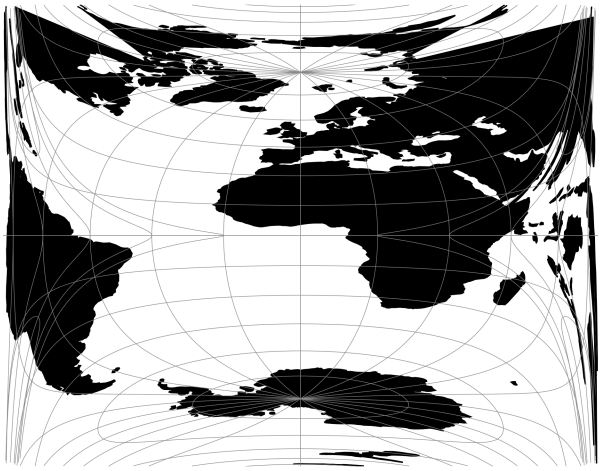
Oblique Cylindrical Equal Area

+proj=ocea



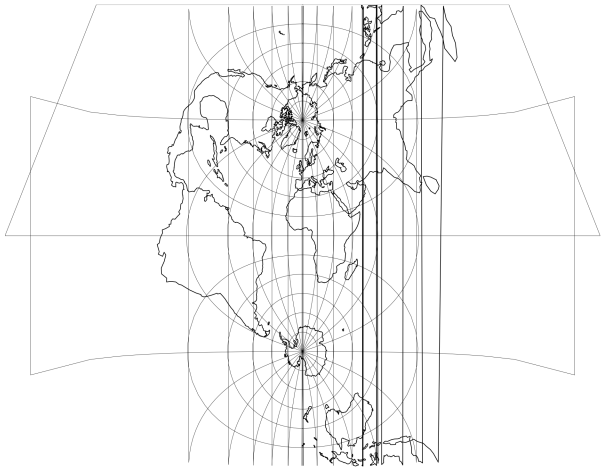
Oblated Equal Area

+proj=oea +m=1 +n=2



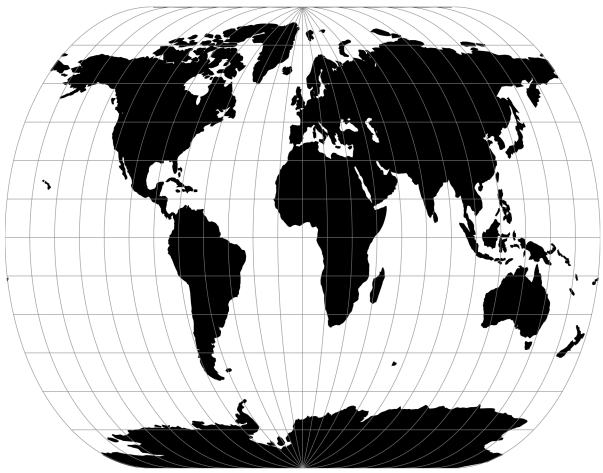
Oblique Mercator

+proj=omerc +lat_1=45 +lat_2=55



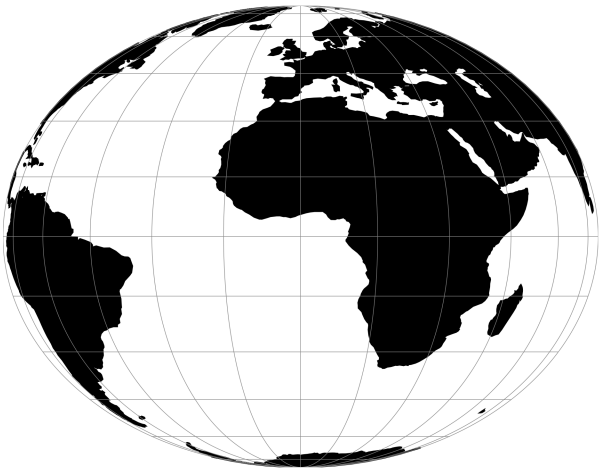
Ortelius Oval

+proj=ortel



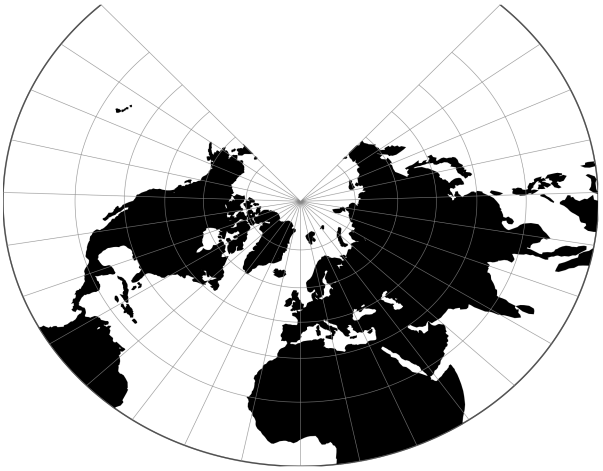
Orthographic

+proj=ortho



Perspective Conic

+proj=pconic +lat_1=25 +lat_2=75



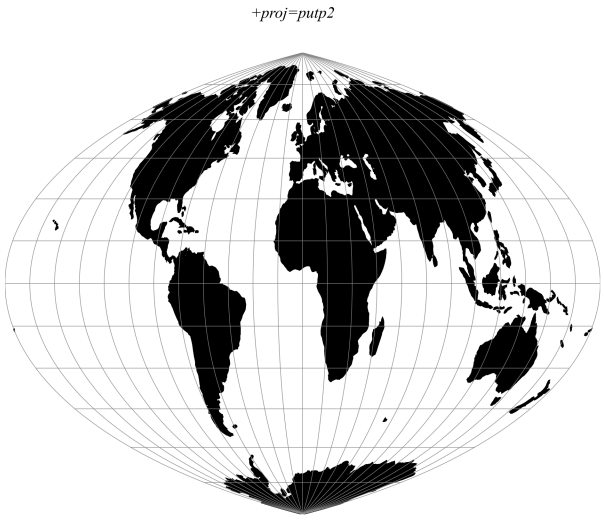
Polyconic (American)

+proj=poly

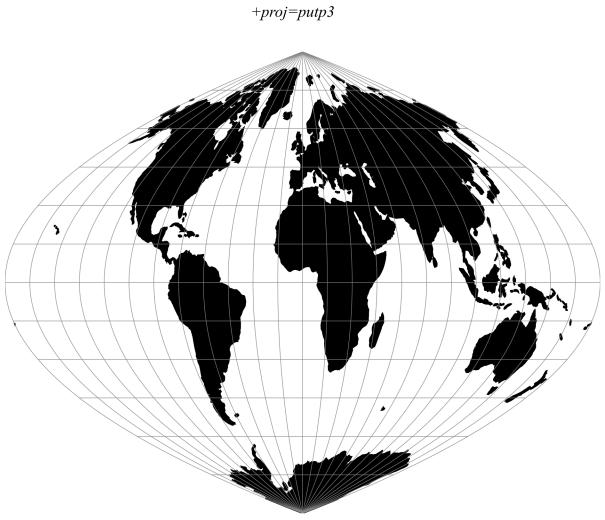
Putnins P1

+proj=putp1

Putnins P2

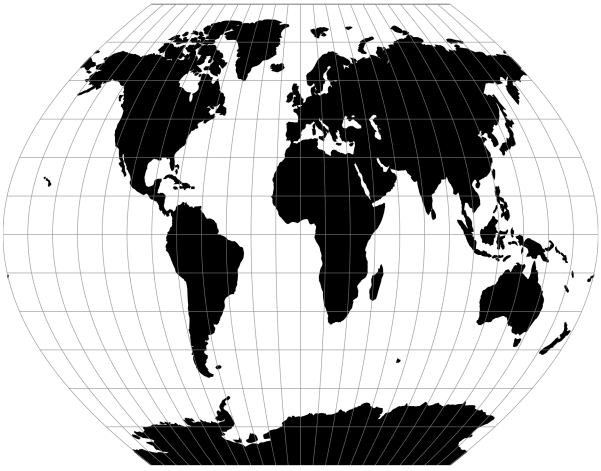


Putnins P3



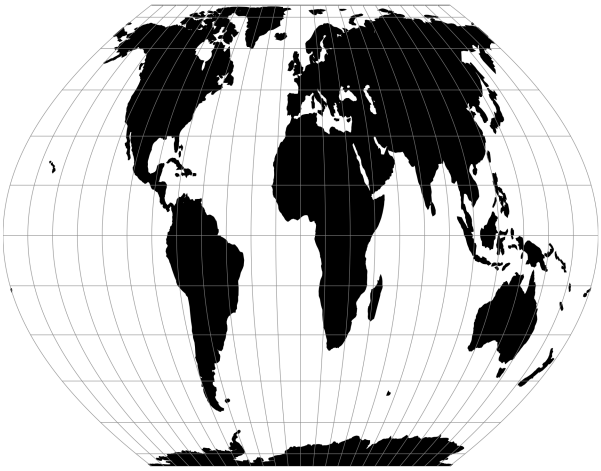
Putnins P3'

+proj=putp3p



Putnins P4'

+proj=putp4p

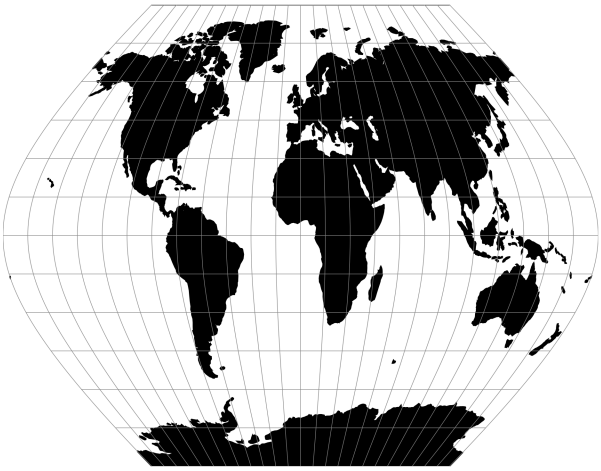


Putnins P5

+proj=putp5

Putnins P5'

+proj=putp5p

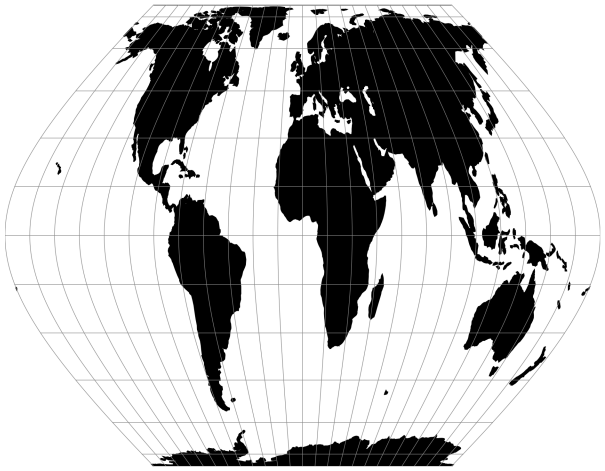


Putnins P6

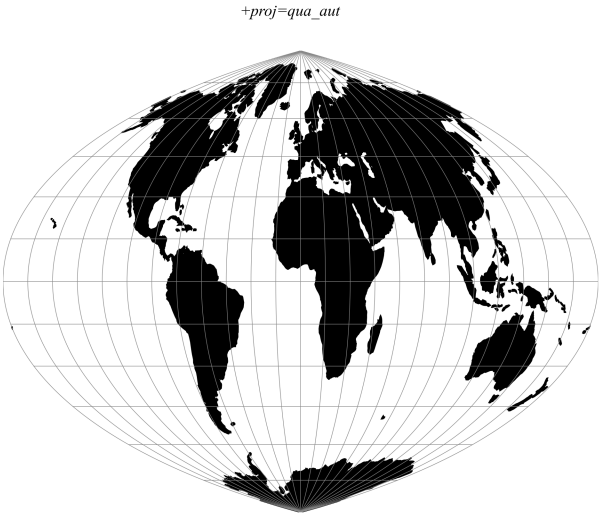
+proj=putp6

Putnins P6'

+proj=putp6p



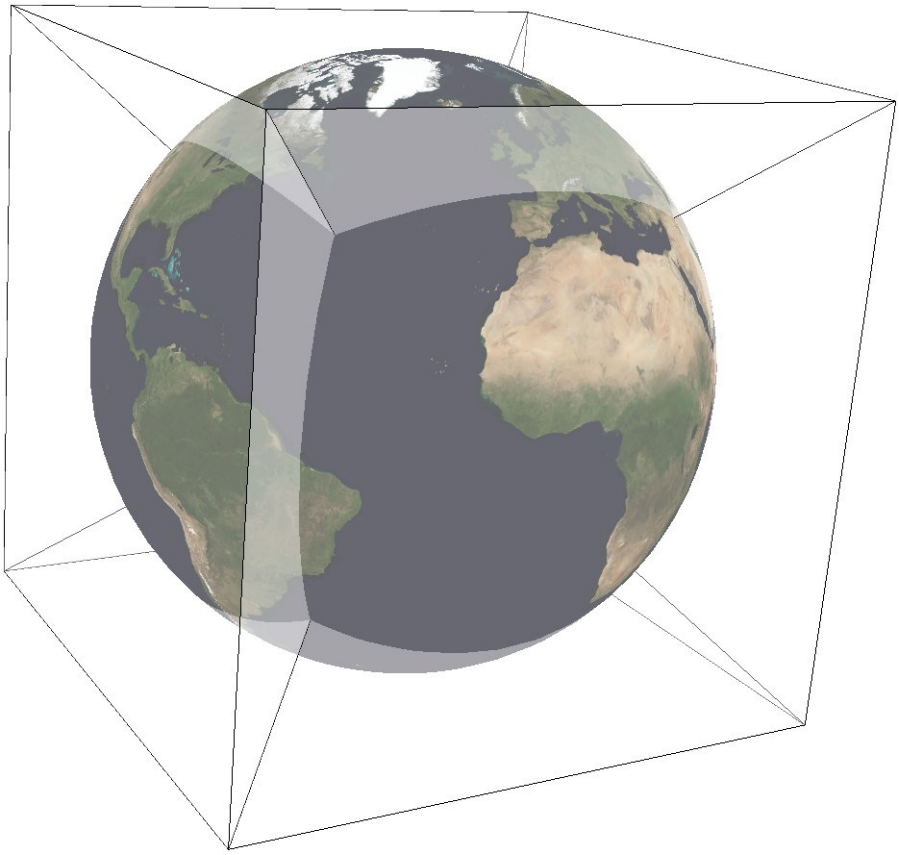
Quartic Authalic



Quadrilateralized Spherical Cube

Classification	Azimuthal
Available forms	Forward and inverse, elliptical projection
Defined area	Global
Implemented by	Martin Lambers
Options	
+lat_0	Latitude (in degrees) of the view position.
+lon_0	Longitude (in degrees) of the view position.

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

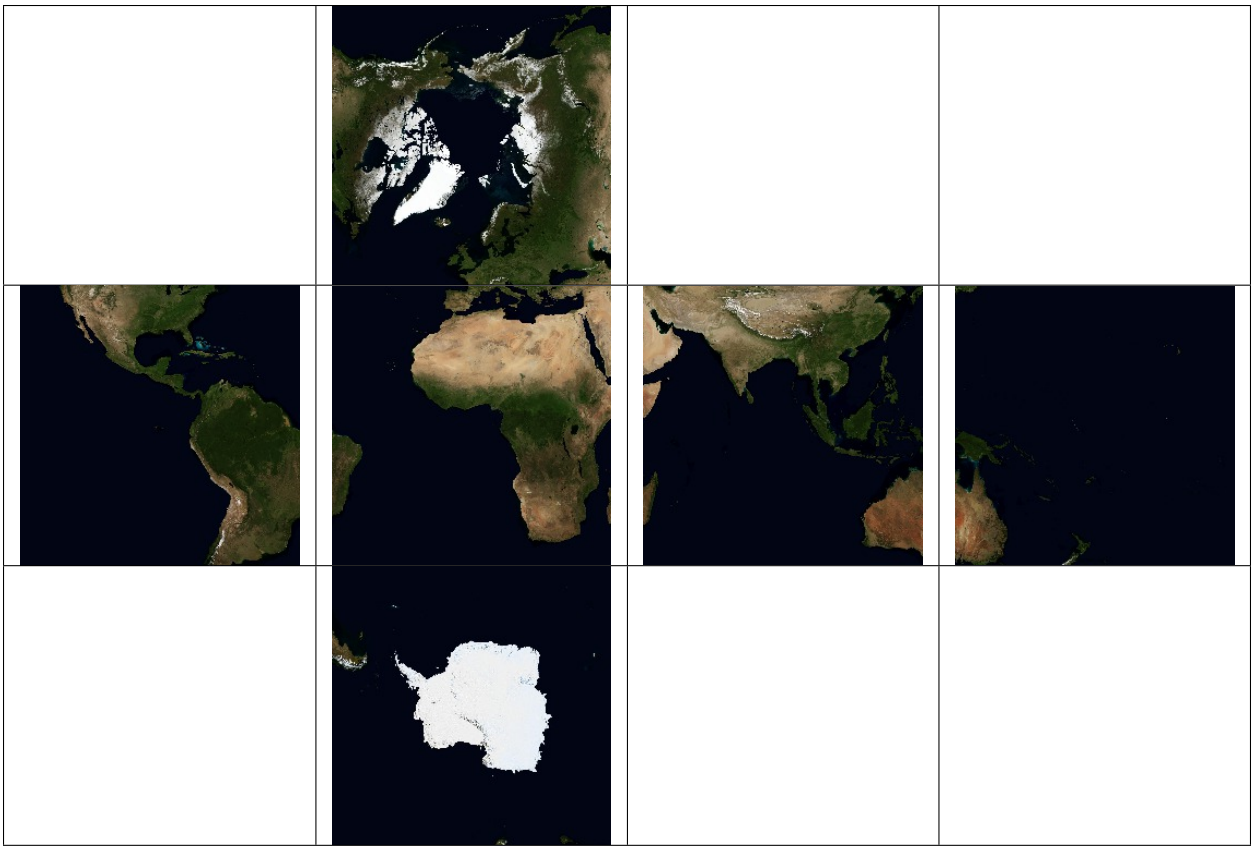
The following example uses QSC projection via GDAL to create the six cube side maps from a world map for the WGS84 ellipsoid:

```
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.

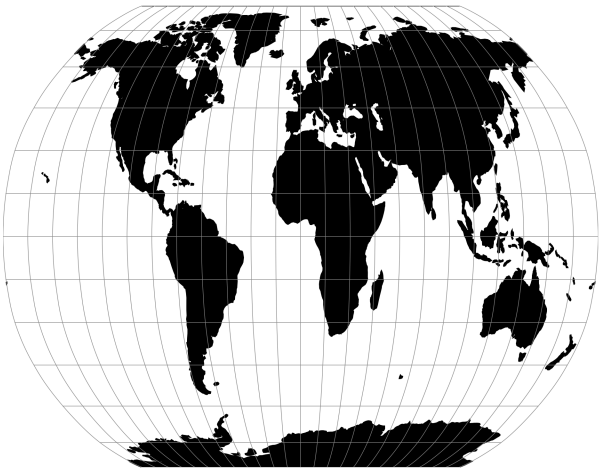


Further reading

1. Wikipedia
2. NASA

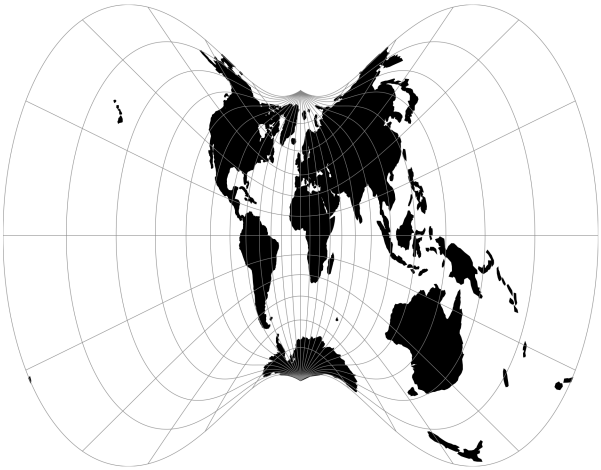
Robinson

+proj=robin

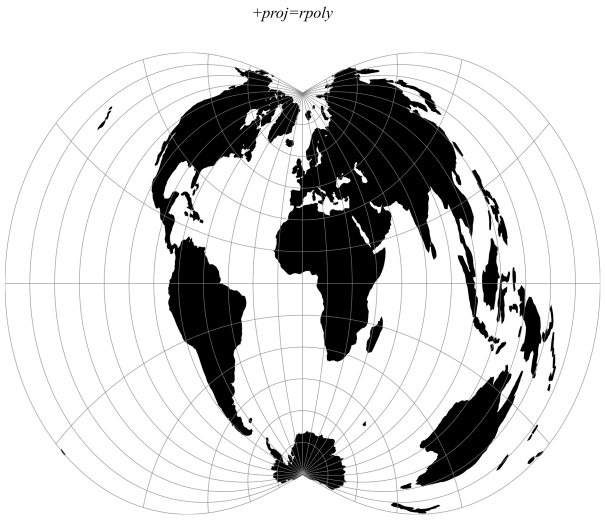


Roussilhe Stereographic

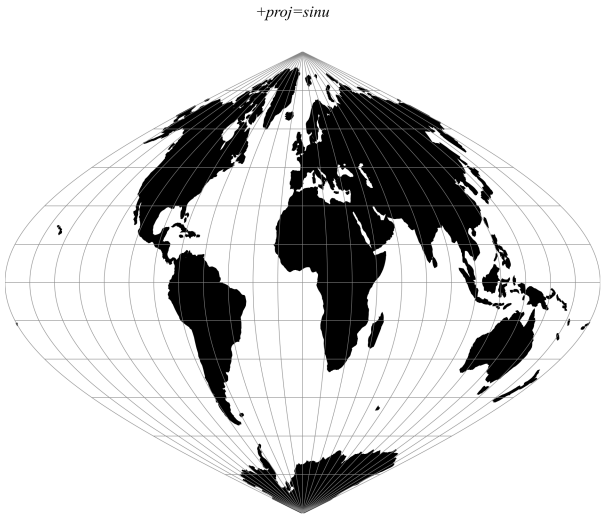
+proj=rouss



Rectangular Polyconic



Sinusoidal (Sanson-Flamsteed)



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

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

$$y = C\theta$$

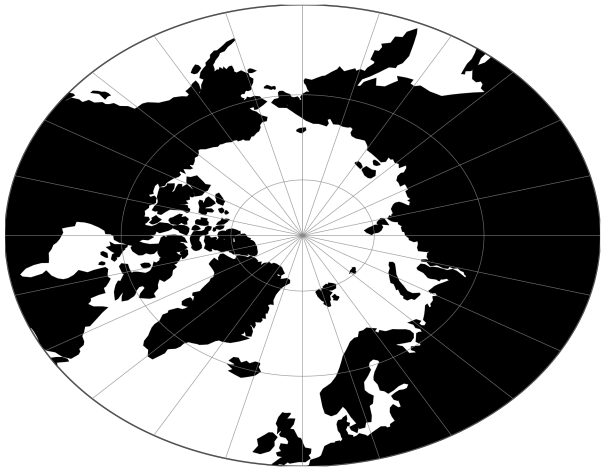
$$C = \sqrt{(m + 1)/n}$$

Swiss. Obl. Mercator

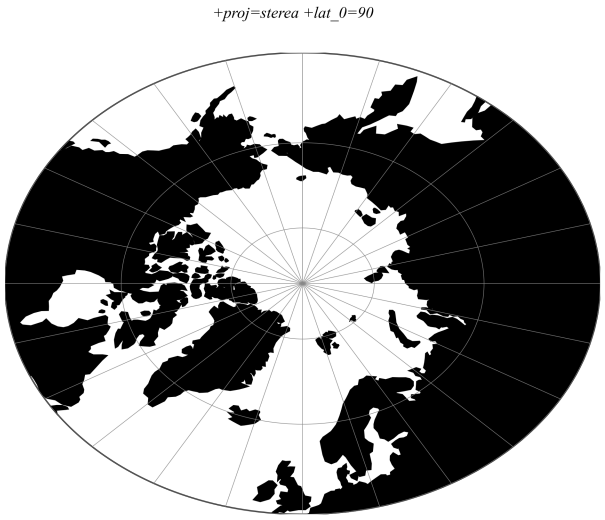
+proj=somerc

Stereographic

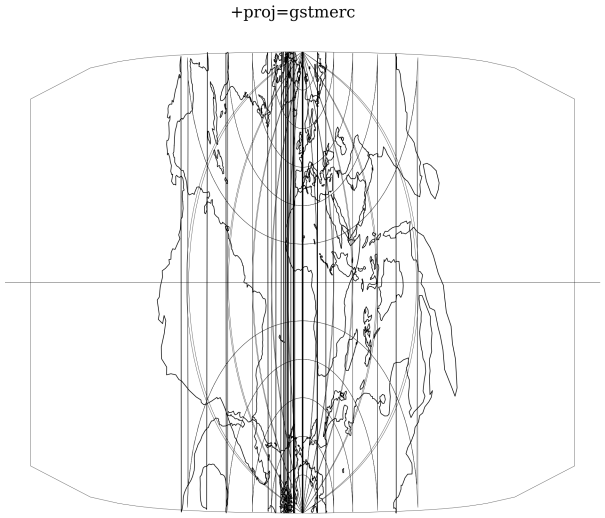
+proj=stere +lat_0=90 +lat_ts=75



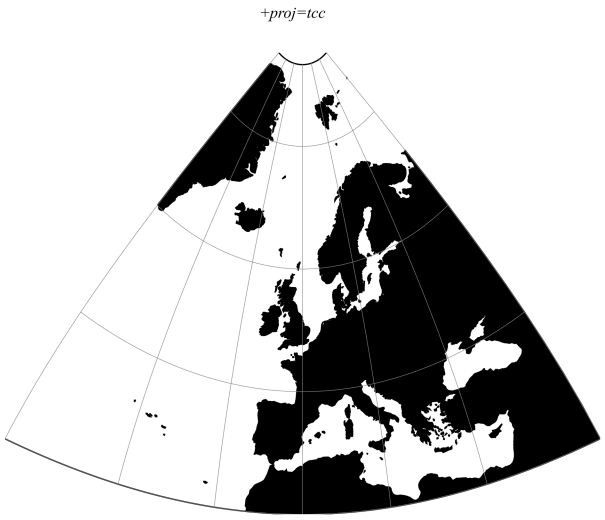
Oblique Stereographic Alternative



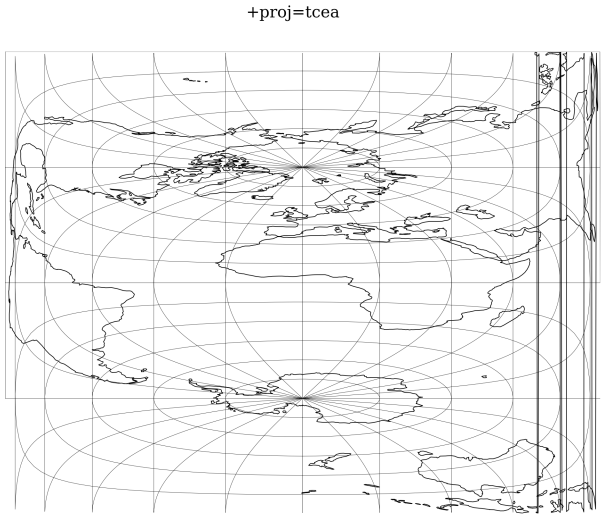
Gauss-Schreiber Transverse Mercator (aka Gauss-Laborde Reunion)



Transverse Central Cylindrical



Transverse Cylindrical Equal Area



Tissot

+proj=tissot +lat_1=60 +lat_2=65

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
Implemented by	Gerald I. Evenden
Options	
+lat_0	Latitude of origin (Default to 0)
+k0	Scale factor at natural origin (Default to 1)

+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=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=0.9996 +x_0=2520000 +y_0=0␣
↪+ellps=intl +units=m +no_defs
2520000.00  4649858.60
```

Mathematical definition

The formulas describing the Transverse Mercator are all taken from Evenden's *[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 meridianal 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

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

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

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

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

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

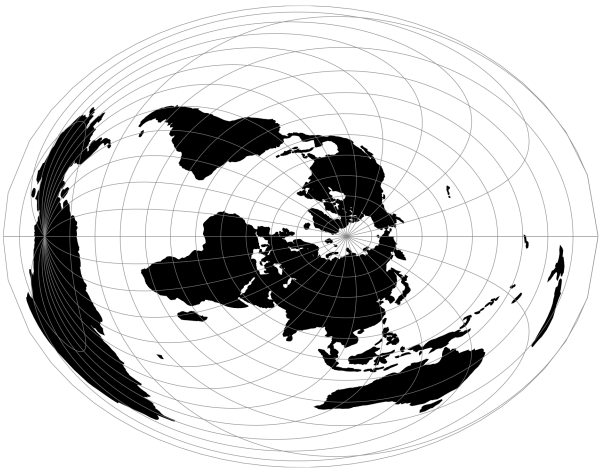
$$\begin{aligned}
\phi = \phi_1 & \\
& - \frac{t_1 x^2}{2! R_1 N_1} \\
& + \frac{t_1 x^4}{4! R_1 N_1^3}(5 + 3t_1^2 + \eta_1^2 - 4\eta_1^4 - 9\eta_1^2 t_1^2) \\
& - \frac{t_1 x^6}{6! R_1 N_1^5}(61 + 90t_1^2 + 46\eta_1^2 + 45t_1^4 - 252t_1^2\eta_1^2) \\
& + \frac{t_1 x^8}{8! R_1 N_1^7}(1385 + 3633t_1^2 + 4095t_1^4 + 1575t_1^6) \\
\lambda = & \frac{x}{\cos \phi N_1} \\
& - \frac{x^3}{3! \cos \phi N_1^3}(1 + 2t_1^2 + \eta_1^2) \\
& + \frac{x^5}{5! \cos \phi N_1^5}(5 + 6\eta_1^2 + 28t_1^2 - 3\eta_1^2 + 8t_1^2\eta_1^2) \\
& - \frac{x^7}{7! \cos \phi N_1^7}(61 + 662t_1^2 + 1320t_1^4 + 720t_1^6)
\end{aligned}$$

Further reading

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

Two Point Equidistant

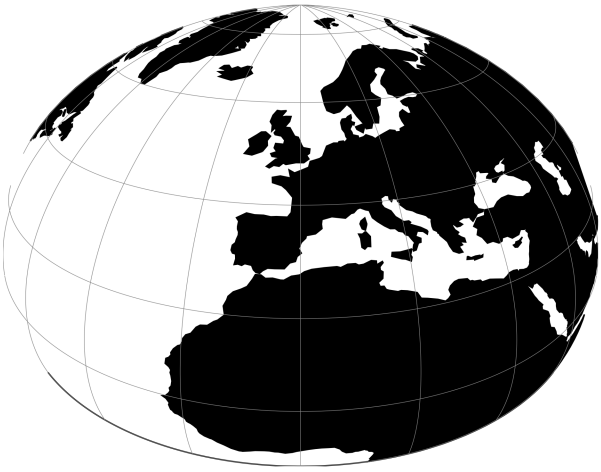
+proj=tpeqd +lat_1=60 +lat_2=65



Tilted perspective

Classification	Azimuthal
Available forms	Forward and inverse, spherical projection
Defined area	Global
Implemented by	Gerald I. Evenden
Options	
+h	Height (in meters) above the surface. Required.
+azi	Bearing (in degrees) from due north.
+tilt	Angle (in degrees) away from nadir.
+lat_0	Latitude (in degrees) of the view position.
+lon_0	Longitude (in degrees) of the view position.

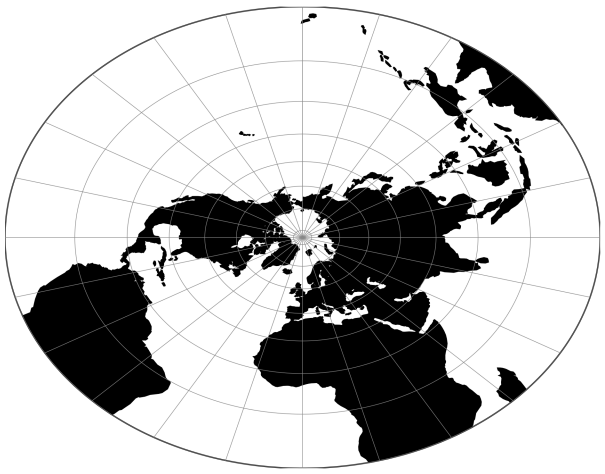
+proj=tpers +h=5500000 +lat_0=40



Tilted Perspective is similar to *Near-sided perspective* (`nsper`) in that it simulates a perspective view from a hight. Where `nsper` projects onto a plane tangent to the surface, Tilted Perspective orients the plane towards the direction of the view. Thus, extra parameters azi and tilt are required beyond *nsper*‘‘s `h`. As with `nsper`, `lat_0` & `lon_0` are also required for satellite position.

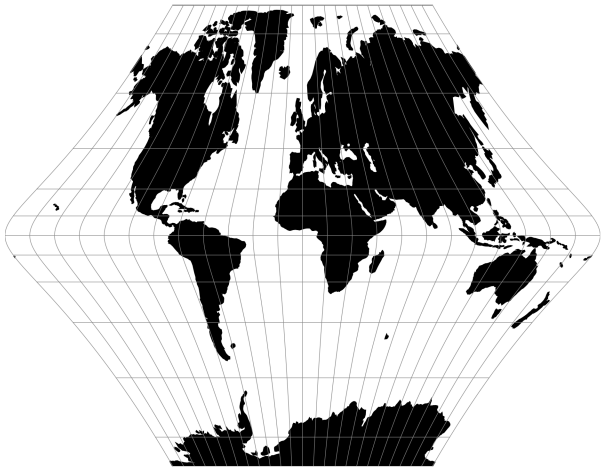
Universal Polar Stereographic

+proj=ups



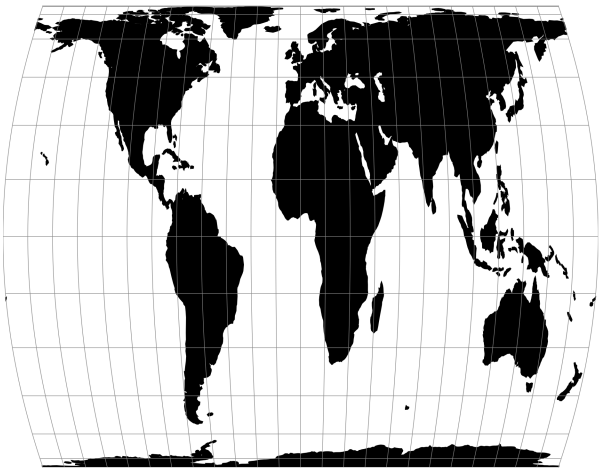
Urmaev V

+proj=urm5 +n=0.9 +alpha=2 +q=4



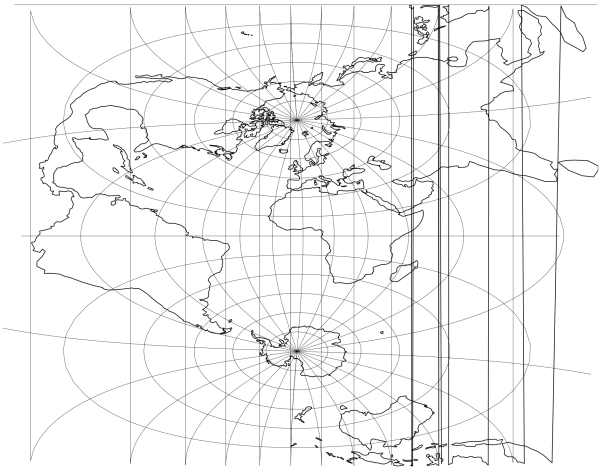
Urmaev Flat-Polar Sinusoidal

+proj=urmfps +n=0.5



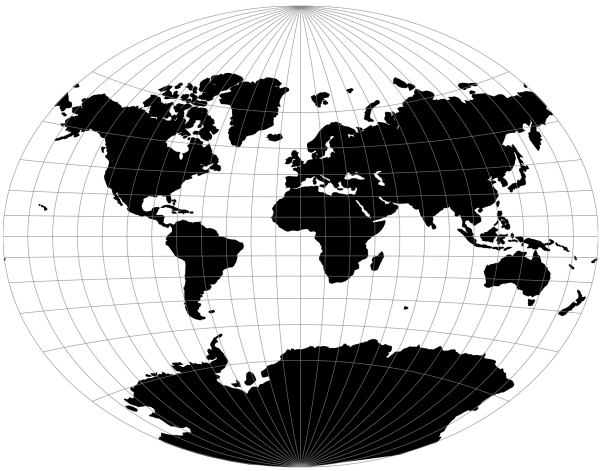
Universal Transverse Mercator (UTM)

+proj=utm



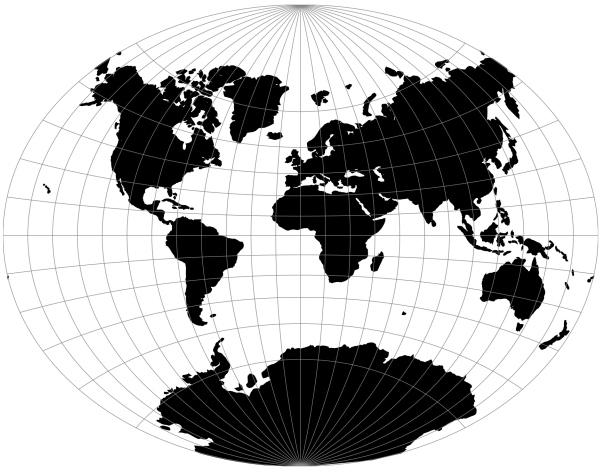
van der Grinten (I)

+proj=vandg



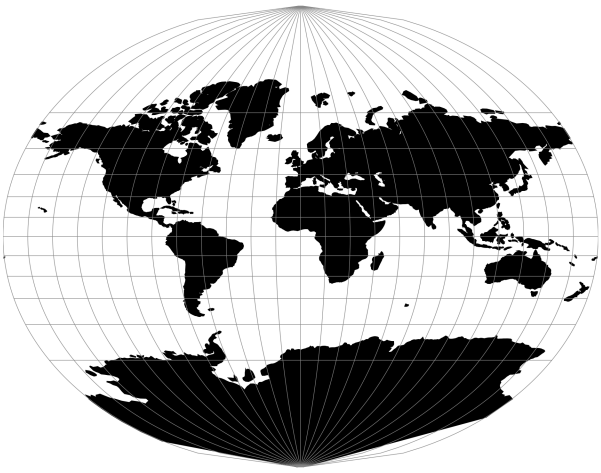
van der Grinten II

+proj=vandg2



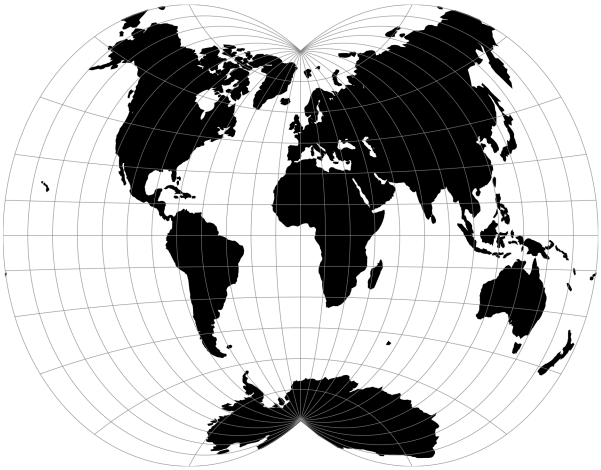
van der Grinten III

+proj=vandg3



van der Grinten IV

+proj=vandg4

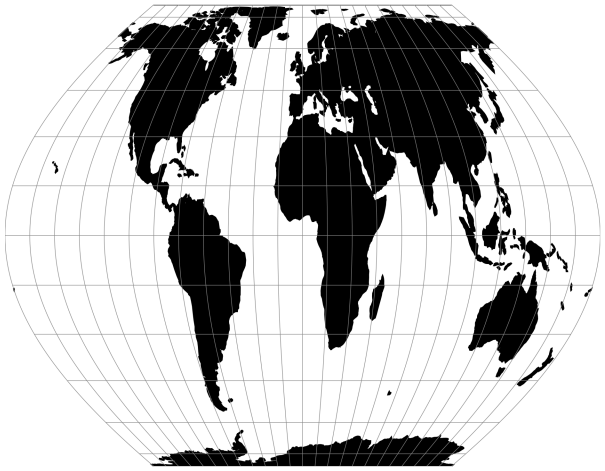


Vitkovsky I

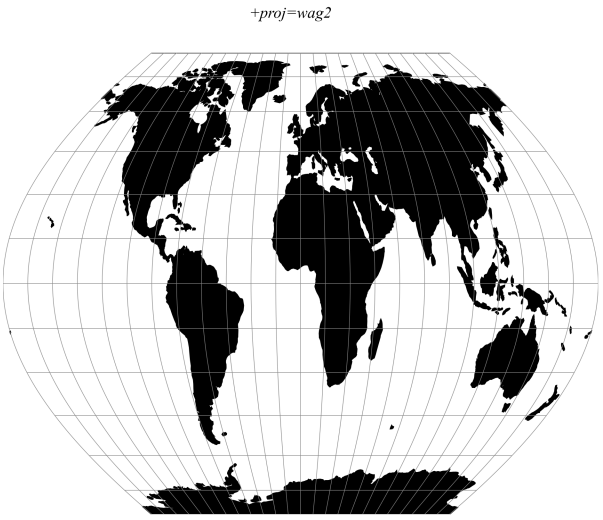
+proj=vitk1 +lat_1=45 +lat_2=55

Wagner I (Kavraisky VI)

+proj=wag1

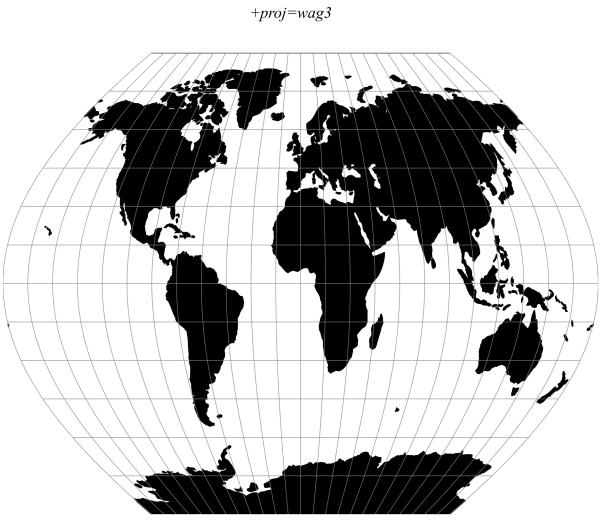


Wagner II



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

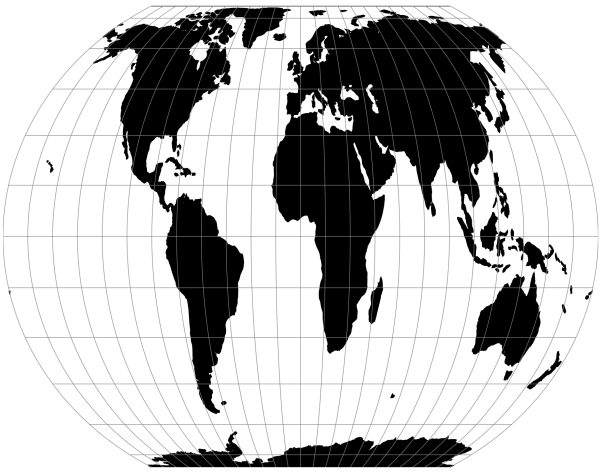
Wagner III



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

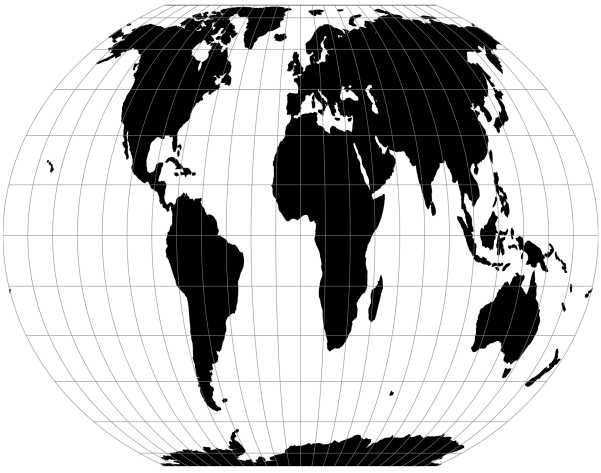
Wagner IV

+proj=wag4



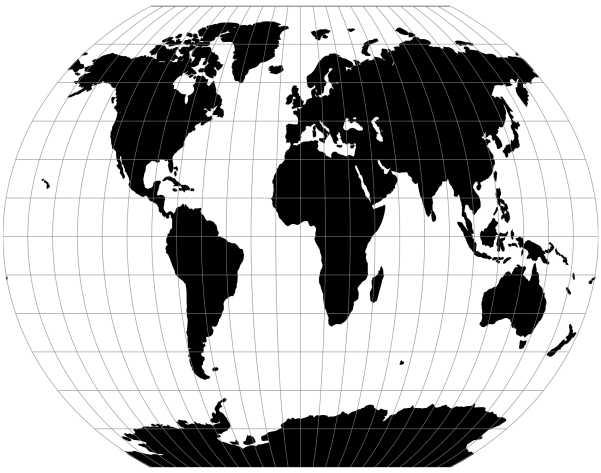
Wagner V

+proj=wag5



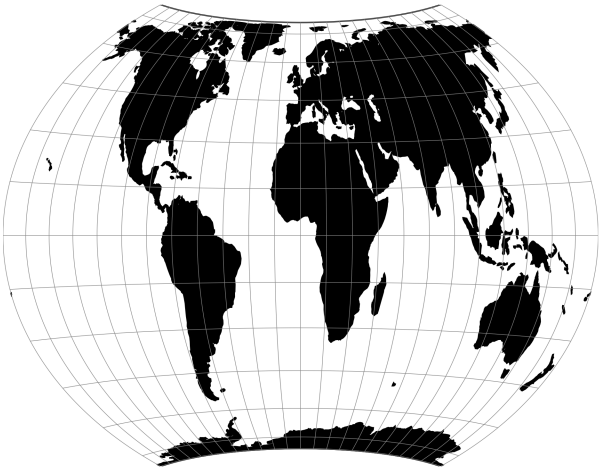
Wagner VI

+proj=wag6



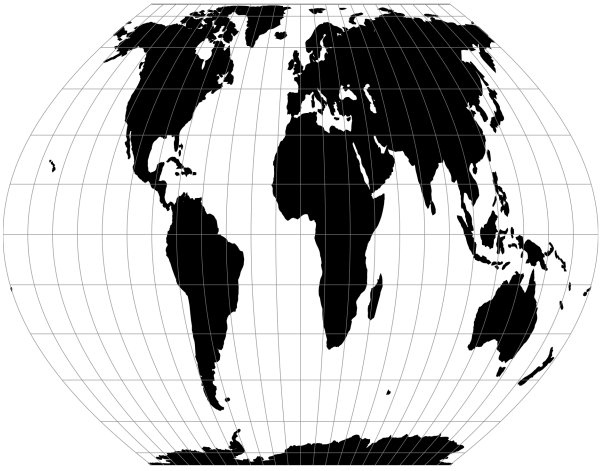
Wagner VII

+proj=wag7



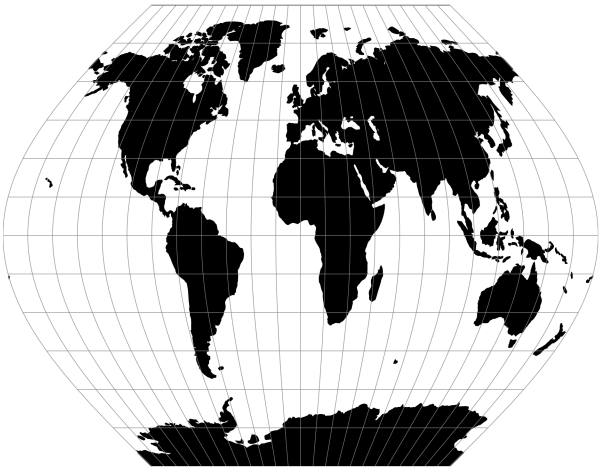
Werenskiold I

+proj=weren



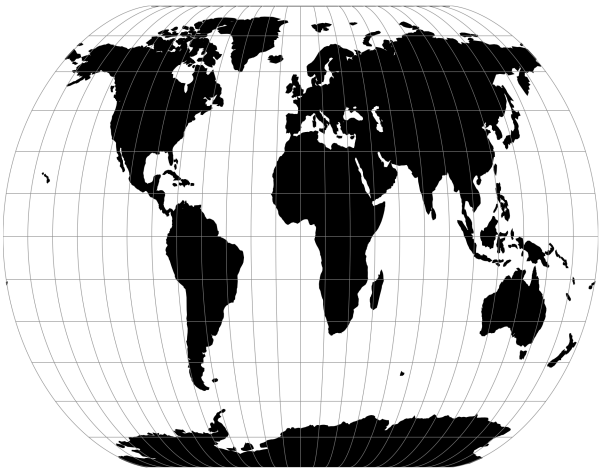
Winkel I

+proj=wink1



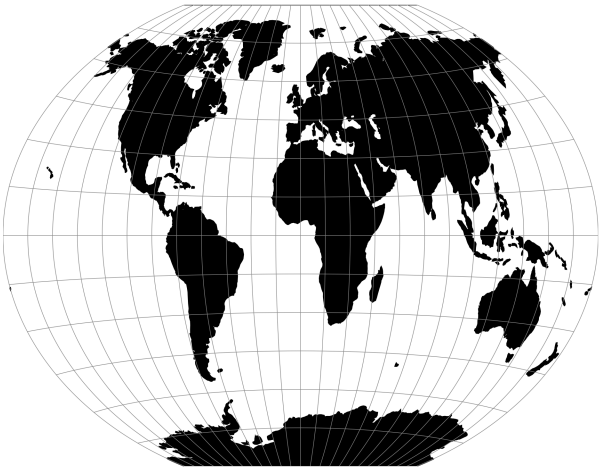
Winkel II

+proj=wink2



Winkel Tripel

+proj=wintri



Conversions

Conversions are coordinate operations in which both coordinate reference systems are based on the same datum. In PROJ.4 projections are differentiated from conversions.

Axis swap

Change the order and sign of 2,3 or 4 axes.

Options	
+order	Ordered comma-separated list of axis, e.g. *+order=2,1,3,4*

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.

Examples

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
```

Cartesian to geodetic conversion

Convert geodetic coordinates to cartesian coordinates.

Options	
+ellps	Ellipsoid of the input coordinates. If used together with the ellipsoid parameters below, `+ellps` is overwritten.
+a	Semi-major radius of ellipsoid axis.
+b	Semi-minor radius of ellipsoid axis.
+es	Eccentricity of ellipsoid.
+f	Flattening of ellipsoid.

Lat/long (Geodetic)

Lat/long (Geodetic alias)

Unit conversion

Convert between various distance and time units.

Options	
+xy_in	Input unit of the horizontal components.
+xy_out	Output unit of the horizontal components.
+z_in	Input unit of the vertical component.
+z_out	Output unit of the vertical component.
+t_in	Input unit of the time component.
+t_out	Output unit of the time component.

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=0.999964286 +x_0=500000.00001016 +y_0=0
+step +proj=unitconvert +xy_in=m +xy_out=us-ft
```

Often when working with GNNS 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=gpsweek +t_out=decimalyear
+step +proj=helmert +epoch=2000.0 +t_obs=2017.5 ...
```

Distance units

In the table below all distance units supported by PROJ.4 is 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

Time units

In the table below all time units supported by PROJ.4 is listed.

mjd	Modfied Julian date
decimalyear	Decimal year
gps_week	GPS Week

Transformations

Transformations coordinate operation in which the two coordinate reference systems are based on different datums.

Helmert transform

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.

Input type	Cartesian coordinates (spatial), decimalyears (temporal).
Output type	Cartesian coordinates (spatial), decimalyears (temporal).
Options	
x	Translation of the x-axis [m]. *Optional*.
y	Translation of the y-axis [m]. *Optional*.
z	Translation of the z-axis. [m]. *Optional*.
s	Scale factor [ppm]. *Optional*.
rx	X-axis rotation in the 3D Helmert [arc seconds]. *Optional*.
ry	Y-axis rotatoin in the 3D Helmert [arc seconds]. *Optional*.
rz	Z-axis rotation in the 3D Helmert [arc seconds]. *Optional*.
theta	Rotation angle in the 2D Helmert. [arc seconds]. *Optional*.
dx	Translation rate of the x-axis. [m/year]. *Optional*.
dy	Translation rate of the y-axis. [m/year]. *Optional*.
dz	Translation rate of the z-axis. [m/year]. *Optional*.
ds	Scale rate factor [ppm/year]. *Optional*.
drx	Rotation rate of the x-axis [arc seconds/year]. *Optional*.
dry	Rotation rate of the y-axis [arc seconds/year]. *Optional*.
drz	Rotation rate of the y-axis [arc seconds/year]. *Optional*.
t_epoch	Central epoch of transformation. [decimalyear]. Only used in spatiotemporal transformations. *Optional*.
t_obs	Observation time of coordinate(s). Mostly useful in 2D and 3D transformations. [decimalyear]. *Optional*.
exact	Use exact transformation equations. *Optional*.
transpose	Transpose rotation matrix. *Optional*.

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 (2D Helmert), transform 3D cartesian coordinates from one static reference frame to another or it can be used to do fully kinematic transformations from global reference frame 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 paramater is specified a kinematic version of the transformation is used.

Examples

Transforming coordinates from NAD72 to NAD83 using the 4 parameter 2D Helmert:

```
proj=helmert ellps=GRS80 x=-9597.3572 y=.6112
             s=0.304794780637 theta=-1.244048
```

Simplified transformations from ITRF2008/IGS08 to ETRS89 using 7 parameters:

```
proj=helmert ellps=GRS80 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 ellps=GRS80
     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
     epoch=1988.0 tobs=2017.0    transpose
```

Mathematical description

In the notation used below, $\hat{P}$ is the rate of change of a given transformation parameter P. $\dot{P}$ is the kinematically adjusted version of P, described by

$$\dot{P} = P + \hat{P}\,(t - t_{central}) \tag{1.1}$$

where t is the observation time of the coordinate and $t_{central}$ is the central epoch of the transformation. Equation (1.1) 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* paramaters.

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:

$$\begin{bmatrix} X \\ Y \end{bmatrix}^B = \begin{bmatrix} T_x \\ T_y \end{bmatrix} + s \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}^A \tag{1.2}$$

(1.2) can be extended to a time-varying kinematic version by adjusting the parameters with (1.1) to (1.2), which yields the kinematic 2D Helmert transform:

$$\begin{bmatrix} X \\ Y \end{bmatrix}^B = \begin{bmatrix} \dot{T}_x \\ \dot{T}_y \end{bmatrix} + s(t) \begin{bmatrix} \cos\dot{\theta} & \sin\dot{\theta} \\ -\sin\ \dot{\theta} & \cos\dot{\theta} \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}^A \tag{1.2}$$

All parameters in (1.2) are determined by the use of (1.1), 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

$$V^B = T + \left(1 + s \times 10^{-6}\right) \mathbf{R} V^A \tag{1.2}$$

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.

The rotation matrix is composed of three rotation matrices, one for each axis:

$$\mathbf{R}_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \tag{1.2}$$

$$\mathbf{R}_Y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{1.3}$$

$$\mathbf{R}_Z = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

The three rotation matrices can be combined in one:

$$\mathbf{R} = \mathbf{R_X R_Y R_Y} \tag{1.5}$$

For $\mathbf{R}$, this yields:

$$\begin{bmatrix} \cos\theta\cos\psi & -\cos\phi\sin\psi + \sin\phi\sin\theta\cos\psi & \sin\phi\sin\psi + \cos\phi\sin\theta\cos\psi \\ \cos\theta\sin\psi & \cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi & -\sin\phi\cos\psi + \cos\phi\sin\theta\sin\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \tag{1.6}$$

Using the small angle approxition the rotation matrix can be simplified to

$$\mathbf{R} = \begin{bmatrix} 1 & -R_z & R_y \\ Rz & 1 & -R_x \\ -Ry & R_x & 1 \end{bmatrix} \tag{1.7}$$

Which allow us to express the most common version of the Helmert transform, using the approximated rotation matrix:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} + \left(1 + s \times 10^{-6}\right) \begin{bmatrix} 1 & -R_z & R_y \\ Rz & 1 & -R_x \\ -Ry & R_x & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{1.7}$$

Applying (1.1) we get the kinematic version of the approximated 3D Helmert:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} \dot{T_x} \\ \dot{T_y} \\ \dot{T_z} \end{bmatrix} + \left(1 + \dot{s} \times 10^{-6}\right) \begin{bmatrix} 1 & -\dot{R}_z & \dot{R}_y \\ \dot{R}_z & 1 & -\dot{R}_x \\ -\dot{R}_y & \dot{R}_x & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{1.7}$$

The Helmert transformation can be applied without using the rotation parameters, in which case it becomes a simlpe translation of the origin of the coordinate system. When using the Helmert in this version equation (1.2) simplies to:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} + \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{1.7}$$

That after application of (1.1) has the following kinematic countpart:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^B = \begin{bmatrix} \dot{T_x} \\ \dot{T_y} \\ \dot{T_z} \end{bmatrix} + \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^A \tag{1.7}$$

Molodensky transform

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]*.

Input type	Geodetic coordinates.
Output type	Geodetic coordinates.
Options	
+da	Difference in semimajor axis of the defining ellipsoids.
+df	Difference in flattening of the defining ellipsoids.
+dx	Offset of the X-axes of the defining ellipsoids.
+dy	Offset of the Y-axes of the defining ellipsoids.
+dz	Offset of the Z-axes of the defining ellipsoids.
+ellps	Ellipsoid definition of source coordinates. Any specification can be used (e.g. *+a*, *+rf*, etc). If not specified, default ellipsoid is used.
+abridged	Use the abridged version of the Molodensky transform. Optional.

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
```

Horizontal grid shift

Change of horizontal datum by grid shift.

Input type	Geodetic coordinates.
Output type	Geodetic coordinates.
Options	
+grids	Comma-separated list of grids to load.

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:

```
+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:

```
+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.4 search path. The list of grids is prioritized so that grids in the start of the list takes presedence over the grids in the back of the list.

Vertical grid shift

Change Vertical datum change by grid shift

Input type	Geodetic coordinates (horizontal), meters (vertical).
Output type	Geodetic coordinates (horizontal), meters (vertical).
Options	
+grids	Comma-separated list of grids to load.

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:

```
+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 presedence over the global model:

```
+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.4 search path. The list of grids is prioritized so that grids in the start of the list takes presedence over the grids in the back of the list.

Geodesic calculations

Contents

Introduction

Consider a 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.

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.

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:

- $s_{13} = s_{12} + s_{23}$,
- $\sigma_{13} = \sigma_{12} + \sigma_{23}$,
- $S_{13} = S_{12} + S_{23}$,
- $m_{13} = m_{12}M_{23} + m_{23}M_{21}$,
- $M_{13} = M_{12}M_{23} - (1 - M_{12}M_{21})m_{23}/m_{12}$,
- $M_{31} = M_{32}M_{21} - (1 - M_{23}M_{32})m_{12}/m_{23}$.

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.

Background

The algorithms implemented by this package are given in *[Karney2013]* 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 *[GeodesicBib]*, *[GeodesicWiki]*, and *[Karney2011]*.

Grids

Contents

Grid files are important for shifting and transforming between datums

US, Canadian, French and New Zealand

- http://download.osgeo.org/proj/proj-datumgrid-1.6.zip: US, Canadian, French and New Zealand datum shift grids - unzip in the *nad* directory before configuring to add NAD27/NAD83 and NZGD49 datum conversion

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.

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.4. 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.4 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

See also:

HTPD describes similar grid shifting

HTDP

HTPD describes the situation with HTDP grids based on NOAA/NGS HTDP Model.

Hungary

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

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.

Canada NTv2.0

Although NTv1 grid shifts are provided freely with PROJ.4, the higher-quality NTv2.0 file needs to be downloaded from Natural Resources Canada. More info: http://www.geod.nrcan.gc.ca/tools-outils/ntv2_e.php.

Procedure:

1. Visit the NTv2, and register/login
2. Follow the Download NTv2 link near the bottom of the page.
3. Unzip *ntv2_100325.zip* (or similar), and move the grid shift file *NTV2_0.GSB* to the proj directory (be sure to change the name to lowercase for consistency) * e.g.: *mv NTV2_0.GSB /usr/local/share/proj/ntv2_0.gsb*

4. **Test it using:**

```
cs2cs +proj=latlong +ellps=clrk66 +nadgrids=@ntv2_0.gsb +to +proj=latlong␣
↪+ellps=GRS80 +datum=NAD83
-111 50
```

```
111d0'3.006"W   50d0'0.103"N 0.000  # correct answer
```

Australia

Geocentric Datum of Australia AGD66/AGD84

Canada

Canadian NTv2 grid shift binary for NAD27 <=> NAD83.

Germany

German BeTA2007 DHDN GK3 => ETRS89/UTM

Great Britain

Great Britain's OSTN15_NTv2: OSGB 1936 => ETRS89

Great Britain's OSTN02_NTv2: OSGB 1936 => ETRS89

Austria

Austrian Grid for MGI

Spain

Spanish grids for ED50.

Portugal

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

Brazil

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

South Africa

South African grid (Cape to Hartebeesthoek94 or WGS84)

Netherlands

Dutch grid (Registration required before download)

HTPD

Contents

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.4 compatible grid shift files for fine grade conversions between various NAD83 epochs and WGS84. Traditionally PROJ.4 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 available you will get an error something like the following:

```
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)
 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 (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.

Development

These pages are primarily focused towards developers either contributing to the PROJ.4 project or using the library in their own software.

Quick start

This is a short introduction to the PROJ.4 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.4 API or consult the *API reference* for specifics.

Before the PROJ.4 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.

```
C = proj_context_create();
```

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)
    return puts ("Oops"), 0;
```

PROJ.4 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.4 is using radians internally. See *Transformations* for further details.

```
a = proj_coord (proj_torad(12), proj_torad(55), 0, 0);
```

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.

```
b = proj_trans_coord (P, PJ_FWD, a);
printf ("easting: %g, northing: %g\n", b.en.e, b.en.n);
```

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

```
b = proj_trans_coord (P, PJ_INV, b);
printf ("longitude: %g, latitude: %g\n", b.lp.lam, b.lp.phi);
```

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

```
proj_destroy (P);
proj_context_destroy (C); /* may be omitted in the single threaded case */
```

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=0 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_coord (P, PJ_FWD, a);
    printf ("easting: %g, northing: %g\n", b.en.e, b.en.n);
    b = proj_trans_coord (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;
}
```

Transformations

Error handling

Threads

This page is about efforts to make PROJ.4 thread safe.

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.4 mutex used to protect access to these memory structures (see pj_mutex.c).

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.4 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.

src/multistresstest.c

A small multi-threaded test program has been written (src/multistresstest.c) for testing multithreaded use of PROJ.4. 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
```

Reference

Data types

This section describe the multiplude of 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 consistenly used.

Transformation objects

PJ
: Object containing everything related to a given projection or transformation. As a user of the PROJ.4 library your are only exposed to pointers to this object and the contents are hidden in 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 enables 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 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.

LP

Geodetic coordinate, latitude and longitude. Usually in radians.

```
typedef struct { double lam, phi; } LP;
```

double **LP.lam**
Longitude. Lambda.

double **LP.phi**
Latitude. Phi.

XY

2-dimensional cartesian coordinate.

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

double **XY.lam**
Easting.

double **XY.phi**
Northing.

UV

2-dimensional generic coordinate. Usually used when contents can be either a *XY* or *UV*.

```
typedef struct {double u, v; } UV;
```

double **UV.u**
Longitude or easting, depending on use.

double **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.

LPZ

3-dimensional version of *LP*. Holds longitude, latitude and vertical component.

```
typedef struct { double lam, phi, z; } LPZ;
```

double **LPZ.lam**
Longitude. Lambda.

double **LPZ.phi**
Latitude. Phi.

double **LPZ.z**
Vertical component.

XYZ
Cartesian coordinate in 3 dimensions. Extension of *XY*.

```
typedef struct { double x, y, z; } XYZ;
```

double **XYZ.x**
Easting.

double **XYZ.y**
Northing.

double **XYZ.z**
Vertical component.

UVW
3-dimensional extension of *UV*.

```
typedef struct {double u, v, w; } UVW;
```

double **UVW.u**
Longitude or easting, depending on use.

double **UVW.v**
Latitude or northing, depending on use.

double **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 *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. Usefull 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.

double **PJ_XYZT.y**
: Northing.

double **PJ_XYZT.z**
: Vertical component.

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 usefull in two, three and four dimensions.

```
typedef union {
    double v[4];
    PJ_XYZT xyzt;
    PJ_UVWT uvwt;
    PJ_LPZT lpzt;
    XYZ  xyz;
    UVW  uvw;
    LPZ  lpz;
    XY   xy;
    UV   uv;
    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.

XYZ **PJ_COORD.xyz**
3-dimensional cartesian coordinate.

UVW **PJ_COORD.uvw**
3-dimensional generic coordinate.

LPZ **PJ_COORD.lpz**
Longitude, latitude and vertical component.

XY **PJ_COORD.xy**
2-dimensional cartesian coordinate.

UV **PJ_COORD.uv**
2-dimensional generic coordinate.

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()*. Depending on the underlying projection, values can be calculated either numerically or analytically.

```
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;
} 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 error.

double **PJ_FACTORS.tissot_semiminor**
Minimum scale error.

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.4.

```
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.4.

```
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.4. Struct is populated by *proj_info()*.

```
typedef struct {
    char        release[64];
    char        version[64];
    int         major;
    int         minor;
    int         patch;
```

```
    char        searchpath[512];
} PJ_INFO;
```

char PJ_INFO.release[64]
Release info. Version number and release date, e.g. "Rel. 4.9.3, 15 August 2016".

char PJ_INFO.version[64]
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.

char PJ_INFO.searchpath[512]
Search path for PROJ.4. List of directories separated by semicolons, e.g. "C:Usersdoctorwho;C:OSGeo4W64\shareproj". 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()*.

```
typedef struct {
    char        id[16];
    char        description[128];
    char        definition[512];
    int         has_inverse;
    double      accuracy;
} PJ_PROJ_INFO;
```

char PJ_PROJ_INFO.id[16]
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*".

char PJ_PROJ_INFO.description[128]
Long describes of the operation the *PJ* object is based on, e.g. "*Mercator Cyl, Sph&Ell lat_ts=*".

char PJ_PROJ_INFO.definition[512]
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.4. 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.4. 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.

Functions

Threading contexts

*PJ_CONTEXT** **proj_context_create**(void)
Create a new threading-context.

Returns *PJ_CONTEXT**

void **proj_context_destroy**(*PJ_CONTEXT* **ctx*)
Deallacote 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");
```

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);
```

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);
```

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

- **nz** (*size_t*) – Number of elements in the corresponding array
- **st** (*size_t*) – Step lenght, in bytes, between consecutive elements of the corresponding array
- **nt** (*size_t*) – Number of elements in the corresponding array

Returns Number of transformations succesfully 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 cordinates in coord

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

Initializers

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*

PJ_OBS **proj_obs**(double *x*, double *y*, double *z*, double *t*, double *o*, double *p*, double *k*, int *id*, unsigned int *flags*)

Initializer for the PJ_OBS union. The function is shorthand for the otherwise convoluted assignment. Equivalent to

```
PJ_OBS c = {{{1.0, 2.0, 3.0, 4.0}}, {{5.0, 6.0, 7.0}}, 8, 9};
```

or

```
PJ_OBS c;
// Assign using the PJ_COORD part of the struct in the union
o.coo.v[0] = 1.0;
o.coo.v[1] = 2.0;
o.coo.v[2] = 3.0;
o.coo.v[3] = 4.0;
o.anc.v[0] = 5.0;
o.anc.v[1] = 6.0;
o.anc.v[2] = 7.0;
o.id       = 8;
o.flags    = 9;
```

which is a bit too verbose in most practical applications.

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*
- **o** (*double*) – 1st component in a PJ_TRIPLET
- **p** (*double*) – 2nd component in a PJ_TRIPLET
- **k** (*double*) – 3rd component in a PJ_TRIPLET
- **id** (*int*) – Ancillary data, e.g. an ID
- **flags** (*unsigned int*) – Flags

Returns PJ_OBS

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.

Param PJ* P: Transformation object.

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.

Info functions

PJ_INFO **proj_info**(void)

Get information about the current instance of the PROJ.4 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.4 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.4 searchpath

Returns *PJ_INIT_INFO*

Lists

const *PJ_OPERATIONS** **proj_list_operations** (void)
Get a pointer to an array of all operations in PROJ.4. 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.4:

```
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.4. 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.4. 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.4. 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, *LP* *a*, *LP* *b*)
Calculate geodesic distance between two points in geodetic coordinates.

Parameters

- **P** (PJ*) – Transformation object
- **a** (LP) – Coordinate of first point
- **b** (LP) – Coordinate of second point

Returns double Distance between a and b in meters.

double **proj_lp_dist** (const *PJ* *P, *LPZ* *a*, *LPZ* *b*)
Calculate geodesic distance between two points in geodetic coordinates.

Parameters

- **P** (PJ*) – Transformation object
- **a** (LPZ) – Coordinate of first point
- **b** (LPZ) – Coordinate of second point

Returns double Distance between a and b in meters.

double **proj_xy_dist** (*XY* *a*, *XY*, b)
Calculate 2-dimensional euclidean between two projected coordinates.

Parameters

- **a** (XY) – First coordinate
- **b** (XY) – Second coordinate

Returns double Distance between a and b in meters.

double **proj_xyz_dist** (*XYZ a*, *XYZ b*)
Calculate 3-dimensional euclidean between two projected coordinates.

Parameters

- **a** (XYZ) – First coordinate
- **b** (XYZ) – Second coordinate

Returns double Distance between a and b in meters.

Various

double **proj_roundtrip** (*PJ *P*, *PJ_DIRECTION direction*, int *n*, *PJ_COORD *coo*)
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
- **obs** (*PJ_OBS*) – Input coordinate

Returns double Distance between original coordinate and the resulting coordinate after n transformation iterations.

PJ_FACTORS **proj_factors** (*PJ *P*, *LP 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 LP*) – 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_geoc_lat**(const *PJ* *_P_, *PJ_DIRECTION* _direction_, *PJ_COORD* _coo_)

Convert geographical to geocentric latitude.

Parameters

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

param *P* Transformation object

type *P* const PJ*

param *direction* Starting direction of transformation

type *direction* PJ_DIRECTION

returns int 1 if angular output is returned, otherwise 0

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 degress, 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);
```

Language bindings

PROJ.4 bindings are available for a number of different development platforms.

Python

pyproj: Python interface (wrapper for PROJ.4)

Ruby

proj4rb: Bindings for PROJ.4 in ruby

TCL

proj4tcl: Bindings for PROJ.4 in tcl (critcl source)

MySQL

fProj4: Bindings for PROJ.4 in MySQL

Excel

proj.xll: Excel add-in for PROJ.4 map projections

Visual Basic

PROJ.4 VB Wrappers: By Eric G. Miller.

Version 4 to 5 API Migration

This is a transition guide for developers wanting to migrate their code to use PROJ version 5.

The difference between the old and new API is best shown with 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 progran for PROJ v. 4:

```
#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:

```
#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_todeg(c.lp.lam);
        c.lp.phi = proj_todeg(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 superflous. 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.

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

FAQ

Contents

Where can I find the list of projections and their arguments?

There is no simple single location to find all the required information. The !PostScript/PDF documents listed on the [http://trac.osgeo.org/proj/wiki main] PROJ.4 page under documentation are the authoritative source but projections and options are spread over several documents in a form more related to their order of implementation than anything else.

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.

The [http://www.remotesensing.org/geotiff/proj_list/ GeoTIFF Projections Pages] include most of the common PROJ.4 projections, and a definition of the projection specific options for each.

- How do I do datum shifts between NAD27 and NAD83?

While the ‘‘‘nad2nad’‘’ program can be used in some cases, the ‘‘‘cs2cs’‘’ is now the preferred mechanism. 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 [wiki:GenParms#nadgrids-GridBasedDatumAdjustments General Parameters] document.

How do I build/configure PROJ.4 to support datum shifting?

After downloading and unpacking the PROJ.4 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.4 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.4 data directory is incorrect, the `PROJ_LIB` environment can be defined with the correct directory.

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.4 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.4 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.

How do I use EPSG coordinate system codes with PROJ.4?

There is somewhat imperfect translation between 2d geographic and projected coordinate system codes and PROJ.4 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 quirkyness are unlikely to work properly. Caveat Emptor!

How do I use 3 parameter and 7 parameter datum shifting

Datum shifts can be approximated with 3 and 7 parameter transformations. Their use is more fully described in the [wiki:GenParms#towgs84-DatumtransformationtoWGS84 towgs84] parameter discussion.

Does PROJ.4 work in different international numeric locales?

No. PROJ.4 makes extensive use of sprintf() and atof() 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.4 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.

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.4 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.4 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
```

Why do I get different results with 4.5.0 and 4.6.0?

The default datum application behavior changed with the 4.6.0 release. PROJ.4 will now only apply a datum shift if both the source and destination coordinate system have valid datum shift information.

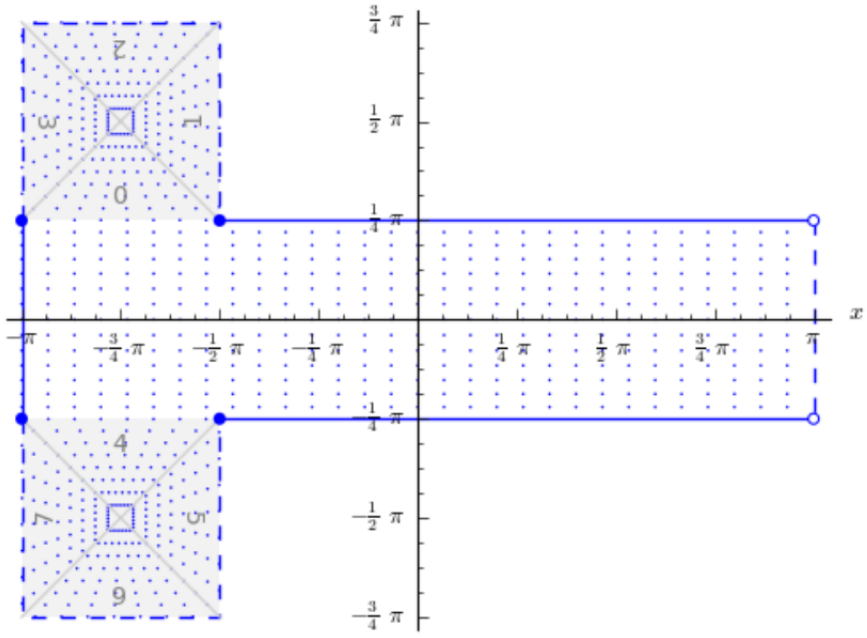
From the PROJ.4 4.6.0 Release Notes (in NEWS):

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

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*.

What is the HEALPix projection and how can I use it?



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. ||

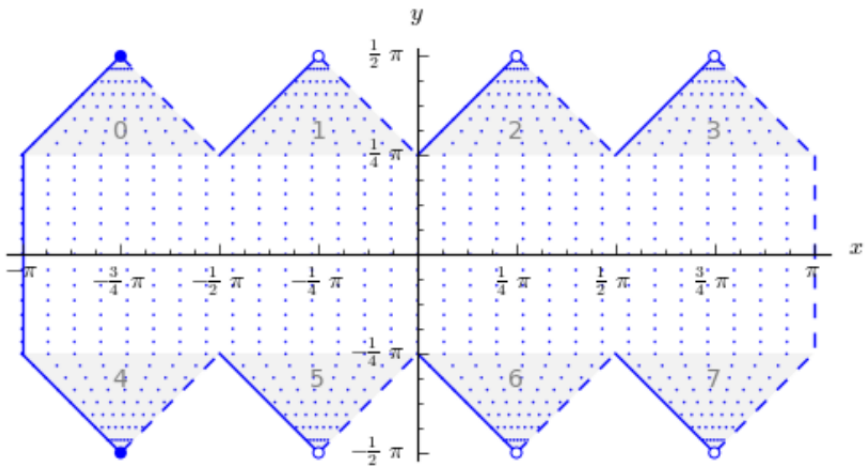
To run a forward HEALPix projection on a unit sphere model, use the following command:

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

Output of the above command.

```
0 0 0.00 0.00
```

What is the rHEALPix projection and how can I use it?



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 no-val approach for the rHEALPix projection. The initial intention of using rHEALPix in the Spatial Computation Engine Science Collaboration Environment (SCENZGrid).

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

```
proj +proj=rhealpix -f '%.2f' -I +lon_0=0 +a=1 +ellps=WGS84 +npole=0 +spole=0 -E <<EOF
0 0.7853981633974483
EOF
```

Where spole and npole are integers from the range of 0 to 3 inclusive and represent the positions of the north polar and south polar squares.

Output of above command:

```
0 0.7853981633974483 0.00 41.94
```

What options does PROJ.4 allow for the shape of the Earth (geodesy)?

See https://github.com/OSGeo/proj.4/blob/master/src/pj_ellps.c for possible ellipse options. For example, putting `+ellps=WGS84` uses the `WGS84` Earth shape.

What if I want a spherical Earth?

Use `+ellps=sphere`. See https://github.com/OSGeo/proj.4/blob/master/src/pj_ellps.c for the radius used in this case.

How do I change the radius of the Earth? How do I use PROJ.4 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"
```

How do I do False Eastings and False Northings?

Use `+x_0` and `+y_0` in the projection string.

Contributing

PROJ.4 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.4. Here's a few suggestion on how:

- Help PROJ.4-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.4 is managed on GitHub most contributions require that you have a GitHub account. Familiarity with issues and the GitHub Flow is an advantage.

Help a fellow PROJ.4 user

The main forum for support for PROJ.4 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.4 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.

Adding bug reports

Bug reports are handled in the issue tracker on PROJ.4'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.4 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.4. If the issue materializes in software that uses PROJ.4 it is likely a problem with that particular software. Make sure that it actually is a PROJ.4 problem before you submit an issue. If you can reproduce the problem only by using tools from PROJ.4 it is definitely a problem with PROJ.4.

Feature requests

Got an idea for a new feature in PROJ.4? 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.

Write documentation

PROJ.4 is in dire need of better documentation. Any contributiions of documentation are greatly appreaciated. The PROJ.4 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.4's supported projections please use the Mercator projection as a template.

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.4 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.4 developers will look at your patch and take an appropriate action.

Coding conventions

Programming language

PROJ.4 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.4 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 evaulating 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.4. New contributions should follow the pj-prefix pattern. Unless there are obvious reasons not to.

Legalese

Commiters are the front line gatekeepers to keep the code base clear of improperly contributed code. It is important to the PROJ.4 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 commiter themselves understanding about the license is hopefully clear. For other contributors, the commiter should verify the understanding unless the commiter 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.

Commiters 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.

Additional Resources

- General GitHub documentation
- GitHub pull request documentation

Acknowledgements

The *code contribution* section of this CONTRIBUTING file is inspired by PDAL's and the *legalese* section is modified from GDAL contributer guidelines

Download

Contents

Release Notes

- NEWS

Current Release

- **2016-09-02** proj-4.9.3.tar.gz (md5)
- **2016-09-11** proj-datumgrid-1.6.zip

Past Releases

- **2015-09-13** proj-4.9.2.tar.gz
- **2015-03-04** proj-4.9.1.tar.gz

Binaries

Linux

- RedHat RPMs
- SUSE
- Debian
- pkgsrc
- Delphi

Docker

A Docker image with just PROJ.4 binaries and a full compliment of grid shift files is available on DockerHub:

Windows

- OSGeo4W contains 32-bit and 64-bit Windows binaries, including support for many *grids*.

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 pseudocylindrical projections are conformal but many are equal area.

To further reduce distortion, pseudocylindrical are often presented in interrupted form that are made by joining several regions with appropriate central meridians and false easting and clipping boundaries. Interrupted Homolosine constructions are suited for showing respective global land and oceanic regions, for example. To reduce the lateral size of the map, some uses remove an irregular, North-South strip of the mid-Atlantic region so that the western tip of Africa is plotted north of the eastern tip of South America.

License

Author Frank Warmerdam

Contact warmerdam@pobox.com

Date 2001

PROJ.4 has been placed under an MIT license. I believe this to be as close as possible to public domain while satisfying those who say that a copyright notice is required in some countries. The COPYING file read as follows:

All source, data files and other contents of the PROJ.4 package are available under the following terms. Note that the PROJ 4.3 and earlier was "public domain" as is common with US government work, but apparently this is not a well defined legal term in many countries. I am placing everything under the following MIT style license because I believe it is effectively the same as public domain, allowing anyone to use the code as they wish, including making proprietary derivatives.

Though I have put my own name as copyright holder, I don't mean to imply I did the work. Essentially all work was done by Gerald Evenden.

```
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 Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

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

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

References

CHAPTER

TWO

MAILING LIST

The PROJ.4 mailing list can be found at http://lists.maptools.org/mailman/listinfo/proj

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[AltamimiEtAl2002] Altamimi, Z., P. Sillard, and C. Boucher (2002), ITRF2000: A new release of the International Terrestrial Reference Frame for earth science applications, J. Geophys. Res., 107(B10), 2214, doi:10.1029/2001JB000561.

[Bessel1825] F. W. Bessel, 1825, The calculation of longitude and latitude from geodesic measurements, Astron. Nachr. **331**(8), 852–861 (2010), translated by C. F. F. Karney and R. E. Deakin.

[CalabrettaGreisen2002] M. Calabretta and E. Greisen, 2002, "Representations of celestial coordinates in FITS". Astronomy & Astrophysics 395, 3, 1077–1122.

[ChanONeil1975] F. Chan and E. M. O'Neill, 1975, "Feasibility Study of a Quadrilateralized Spherical Cube Earth Data Base", Tech. Rep. EPRF 2-75 (CSC), Environmental Prediction Research Facility.

[Danielsen1989] J. Danielsen, 1989, The area under the geodesic, Survey Review **30**(232), 61–66.

[Deakin2004] R.E. Deakin, 2004, The Standard and Abridged Molodensky Coordinate Transformation Formulae.

[EberHewitt1979] L. E. Eber and R.P. Hewitt, 1979, Conversion algorithms for the CALCOFI station grid, California Cooperative Oceanic Fisheries Investigations Reports 20:135-137.

[Evenden1995] G. I. Evenden, 1995, Cartograpic Projection Procedures for the UNIX Environment - A User's Manual.

[Evenden2005] G. I. Evenden, 2005, libproj4: A Comprehensive Library of Cartographic Projection Functions (Preliminary Draft).

[EversKnudsen2017] K. Evers and T. Knudsen, 2017, Transformation pipelines for PROJ.4, FIG Working Week 2017 Proceedings.

[GeodesicBib] A geodesic bibliography.

[GeodesicWiki] The wikipedia page, Geodesics on an ellipsoid.

[Helmert1880] F. R. Helmert, 1880, Mathematical and Physical Theories of Higher Geodesy, Vol 1, (Teubner, Leipzig), Chaps. 5–7.

[Karney2011] C. F. F. Karney, 2011, Geodesics on an ellipsoid of revolution; errata.

[Karney2013] C. F. F. Karney, 2013, Algorithms for geodesics, J. Geodesy **87**(1) 43–55; addenda.

[Komsta2016] L. Komsta, 2016, ATPOL geobotanical grid revisited - a proposal of coordinate conversion algorithms, Annales UMCS Sectio E Agricultura 71(1), 31–37.

[LambersKolb2012] M. Lambers and A. Kolb, 2012, "Ellipsoidal Cube Maps for Accurate Rendering of Planetary-Scale Terrain Data", Proc. Pacfic Graphics (Short Papers).

[ONeilLaubscher1976] E. M. O'Neill and R. E. Laubscher, 1976, "Extended Studies of a Quadrilateralized Spherical Cube Earth Data Base", Tech. Rep. NEPRF 3-76 (CSC), Naval Environmental Prediction Research Facility.

[Snyder1987] J. P. Snyder, 1987, Map Projections - A Working Manual. U.S. Geological Survey professional paper 1395.

[Snyder1993] J. P. Snyder, 1993, Flattening the Earth, Chicago and London, The University of Chicago press.

[Steers1970] J. A. Steers, 1970, An introduction to the study of map projections (15th ed.), London Univ. Press, p. 229.

[Verey2017] M. Verey, 2017, Theoretical analysis and practical consequences of adopting an ATPOL grid model as a conical projection, defining the conversion of plane coordinates to the WGS-84 ellipsoid, Fragmenta Floristica et Geobotanica Polonica (preprint submitted).

[Vincenty1975] T. Vincenty, 1975, Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations, Survey Review **23**(176), 88–93.

[WeberMoore2013] E. D. Weber and T.J. Moore, 2013, Corrected Conversion Algorithms For The Calcofi Station Grid And Their Implementation In Several Computer Languages, California Cooperative Oceanic Fisheries Investigations Reports 54.

[Zajac1978] A. Zajac, 1978, "Atlas of distribution of vascular plants in Poland (ATPOL)", Taxon 27(5/6), 481–484.

INDEX

U

X