Contents

Introduction to Geometrical Geodesy in *Mathematica*

Geodesy is the science that studies the macroscopic shape of the Earth, including its gravitational and magnetic fields. A basic function of geodesy is to create meaningful coordinate systems for the purpose of surveying, mapping, navigation, and any other application of spatial data. In particular, geodesy is what gives meaning to latitude, longitude, and height. Geodesy is the cornerstone of all spatial sciences and has applications in geology, geophysics, and astronomy, to name a few.

Geodesy can broadly subdivided into physical geodesy and geometrical geodesy. Physical geodesy is concerned with the phenomena that give rise to the **geoid**, being the Earth's gravitational equipotential surface that is associated with mean sea level, and the physics that describe it for the purpose of creating vertical datums. Geometrical geodesy is used to answer geometric questions such as, "what is the distance between two places" or "what direction is point **A** from point **B.**" The material in this package is concerned solely with geometrical geodesy.

Organization of the package

This package subdivides geometrical geodesy into several sections. The Units section defines many of the linear and angular units used in surveying, mapping, and geodesy. The Reference Ellipsoid section enumerates the parameters that define many of the geodetic reference ellipsoids that are used in surveying, mapping, and geodesy. The Datums section defines many of the geodetic reference ellipsoids that are used in surveying, mapping, and geodesy. The Datum Transformation section implements the seven-parameter Helmert transformation. The Coordinate Systems section implements the geodetic curvilinear coordinate system (geodetic latitude and longitude, and ellipsoid height) and its associated three-dimensional Cartesian coordinate system (XYZ), and the local three-dimensional Cartesian coordinate system (East-North-Up). The Projection section implements the Mercator, Transverse Mercator, Oblique Mercator, Lambert Conformal Conic, Stereographic and Orthometric projections. These are sufficient that they could be used to implement the Universal Transverse Mercator (UTM) and State Plane Coordinate System (SPCS) mapping systems. The Geodetic Computations section implements many formulae for many common problems in geometrical geodesy, such as the **forward** and **inverse** problems.

Units Overview

This package defines both linear and angular units of measure. All computations in this package are performed in meters and with radians. Therefore, this package provides functions that convert degrees/minutes/seconds and decimal degrees into radians and various conversion factors to and from meters. Functions are also provided to display radians in decimal degrees and degrees/minutes/seconds.

Reference Ellipsoids Overview

Reference ellipsoids, generically speaking, are either models of the Earth's macroscopic shape or, in some sense, first-order approximations of the geoid. A sphere is the simplest approximation of the Earth shape, but the Earth's rotational axis is slightly shorter than its equatorial axis. Consequently, the Earth is better modeled by an ellipse of rotation, or more simply, an ellipsoid. These ellipsoids constitute the reference surface upon which coordinate systems are developed and upon which computations are performed. Therefore, they are known as reference ellipsoids.

Datums Overview

A geodetic datum is a system by which coordinates, such as latitude and longitude, can be meaningfully assigned to geographic locations. Hundreds of geodetic datums have been created, mostly with local or regional extent. This package defines several of these datums and associates them with the reference ellipsoid upon which they were defined.

Datum Transformations Overview

Recently, because of advances in satellite geodesy and technologies such as Very Long Baseline Interferometry, globally-applicable datums have been created. These include the International Terrestrial Reference System, the World Geodetic System of 1984, the Geodetic Datum of Austrialia of 1994, European Terrestrial Reference System of 1989, the South American Geocentric Reference System, and the North American Datum of 1983. These global geodetic datums largely agree on the location, scale, and orientation of their coordinate systems. Consequently, there exist relatively simple, rigorous mathematical transformations between them. This package implements the seven-parameter Helmert transformation.

Coordinate Systems Overview

Geometrical geodesy is performed in a wide variety of coordinate systems, including geocentric Cartesian (XYZ), local Cartesian (ENU), geographic (ϕ, λ, h), and cartographic grids. This package implements all of these systems and provides transformations between them and operations upon them. For example, XYZ points can be thought of as vectors, being a vector from the origin of the coordinate system to the point. Therefore, they can be treated as vectors and this package provides a full set of vector algebra operations for them, including addition, scalar multiplication, inner multiplication (dot product), outer multiplication (cross product), and matrix multiplication.

Positions in this package are objects of one of grid types, namely *blh*, *xyz*, *enu*, or *grid*. Positions consist of three coordinates and the datum within which the coordinates are defined.

Geodetic Computations Overview

This package implements the major computations performed in geometric geodesy. For the most part, these computations pertain to geodetic traverses, the *direct* and *inverse* problems performed on the surface of a reference ellipsoid.

Projections Overview

This package implements the major conformal projections performed in geometric geodesy. These include Mercator, Lambert conformal conic, and stereographic.

Miscellaneous Functions Overview

This package provides a few, general purpose functions that are strictly limited to geometrical geodesy. These functions include a generic function for printing/displaying numbers with a great many significant digits, and a vector norm (magnitude) operator.

Linear Units

The following is a list of the linear units defined in GeometricalGeodesy`Units`. All computations in this package are performed with linear units of meters, so the values in the following table provide the factor to convert the unit into meters. Whenever possible, the exact conversion is given and retained as a rational number, thereby providing it into computations exactly.

Unit	Value
Meter	1
Kilometer	1000 Meter
KM	abbreviation for Kilometer
Decimeter	1/10 Meter
dm	abbreviation for Decimeter
Centimeter	1/100 Meter
cm	abbreviation for Centimeter
Millimeter	1/1000 Meter
mm	abbreviation for Millimeter
USSurveyFoot	1200/3937 Meter
USFt	abbreviation for USSurveyFoot
USSurveyYard	400/3937 Meter
USYard	abbreviation for USSurveyYard
USSurveyInch	100/3937 Meter
USIn	abbreviation for USSurveyInch
StatuteMile	5280 USFt
USMile	abbreviation for StatuteMile
GuntersChain	66 USFt
Link	1/100 GuntersChain
Rod	1/320 StatuteMile
InternationalYard	9144/10000 Meter
IntYard	abbreviation for InternationalYard
InternationalFoot	3048/10000 Meter
IntFt	abbreviation for InternationalFoot
InternationalInch	254/10000 Meter
IntIn	abbreviation for InternationalInch
NauticalMile	1852 Meter
NM	abbreviation for NauticalMile

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Converting a value to meters is done simply by giving the value followed by the value's original units. A space character is required between the numeric quantity and the conversion factor, the space being a multiplication operator. Convert one nautical mile to meters

```
1 NauticalMile
```

```
1852
```

Convert 125 International inches to meters. Notice the exact conversion left as a rational number.

```
125 IntIn
```

$$\frac{127}{40}$$

Converting a value from meters to another unit is done simply by dividing the value by the desired units. Convert 32.5 meters to US Survey Feet

```
32.5 Meter / USFt
```

```
106.627
```

Convert 6378137 meters to nautical miles. Notice that it is not necessary to explicitly include Meter for the units of any quantity that is, in fact, expressed in meters. All linear quantities are assumed to be in meters.

```
6378137 / NM
```

$$\frac{6378137}{1852}$$

One can always convert the exact rational quantity to a numeric approximation with the built-in function **N**.

```
6378137 / NM // N
```

```
3443.92
```

One meter is 39.37 English inches.

```
1 Meter / USInch // N
```

```
39.37
```

Units can be converted freely simply by given the value followed by its original units divided by the desired units. Convert one nautical mile to US Survey Feet.

```
1 NauticalMile / USSurveyFoot // N
```

```
6076.1
```

```
1 StatuteMile / USFt
```

```
5280
```

Disparate units can be easily combined in expressions

```
(2430.15 USFt + 320.432 Meter + 5 Rod + 41 GuntersChain - 336 Link) / StatuteMile
```

```
1.14549
```

decimal degrees

Mathematica comes with a constant that converts angles expressed in decimal degrees into radians, namely, **`Degree.`**

Convert from DD to radians:

```
30 Degree
```

```
30 °
```

```
Sin[30 °]
```

$\frac{1}{2}$

To convert from radians to decimal degrees, divide by `Degree`.

```
0.5 / °
```

```
28.6479
```

Printing/displaying decimal degrees: PDD

PDD is a function that converts angles expressed in radians to decimal degrees. This function is listable and slightly more attractive than dividing by Degree. Otherwise, it does the same thing.

`PDD[α]`	convert an angle α radians into a number given in decimal degrees.
`PDD[{α ..}]`	convert a list of angles given in radians into a number given in decimal degrees

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Convert an angle: 0.123 radians is equivalent to 7.04738 degrees, approximately.

```
PDD[0.123]
```

```
7.04738
```

```
0.123 / °
```

```
7.04738
```

```
PDD[Table[α, {α, 0, 0.5, 0.1}]]
```

```
{0, 5.72958, 11.4592, 17.1887, 22.9183, 28.6479}
```

Converting radians to arc-degrees, minutes, and seconds: DMS

Four constants are defined pertaining to the cardinal directions. These constants are used as parameters for **DMS** to specify the hemisphere of the angle. The constants have values of the text strings of their names. For example, **North** has the value **"North"**.

`North`	the direction along a meridean towards the geodetic North Pole
`South`	the direction along a meridean towards the geodetic South Pole
`East`	the direction along a parallel in increasing longitude
`West`	the direction along a parallel in decreasing longitude

DMS is a function that converts inputs of arc-degrees/minutes/seconds into radians.

`DMS[`*degrees*`,` *minutes*`,` *seconds*`,` *hemisphere*`]`	convert an angle expressed as *degrees*°, *minutes*', *seconds*" into radians, with *hemisphere* being optional.
`DMS[{`*degrees*`,` *minutes*`,` *seconds*`,` *hemisphere*`}]`	convert a list of the three inputs into radians
`DMS[{{`*degrees*`,` *minutes*`,` *seconds*`,` *hemisphere*`} ..}]`	convert a list of one or more lists of the three inputs into a list of radians
hemisphere	one of North, South, East, West, "N", "S", "E", "W", "n", "s", "e", "w". The quotations are optional; see examples.

The arguments for **DMS** are type-checked: *degrees* and *minutes* must be integers, although *seconds* can be any type of number. No range checking is done on the arguments but nonsensical results might be produced if the inputs fall outside the normal bounds (e.g., *minutes* greater than 59). In any case, negative angles should be created by negating the entire expression rather than providing negative arguments (see below).

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

These examples illustrate how to use **DMS** without using the optional *hemisphere* argument. These functions would be used, for example, to convert horizontal angles measured with a transit to radians. Convert 20° 30' 5" to radians.

```
DMS[20, 30, 5] // N
```

```
0.357817
```

The list versions. These versions are often useful for processing lists of angles.

```
DMS[{20, 30, 5}] // N
```

```
0.357817
```

```
Sin[DMS[Table[{d, 30, 5}, {d, 0, 70, 10}]]] // N
```

```
{0.00875078, 0.182259, 0.35023, 0.507559, 0.649466, 0.77164, 0.870368, 0.94265}
```

A negative angle.

```
-DMS[20, 30, 5] // N
```

```
-0.357817
```

These examples illustrate how to use `DMS` using the optional *hemisphere* argument. These functions might be used, for example, to specify azimuths. **`South`** is a constant defined in this package.

```
DMS[20, 30, 5, South] // N
```

```
-0.357817
```

The hemisphere can be given either with a symbol...

```
DMS[{15, 0, 5, e}] // N
```

```
0.261824
```

or with a text string.

```
DMS[{15, 0, 5, "west"}] // N
```

```
-0.261824
```

```
Tan[DMS[Table[{d, 30, 5, w}, {d, 0, 70, 10}]]] // N
```

```
{-0.00875111, -0.185364, -0.373912, -0.589078, -0.854123, -1.21316, -1.76759, -2.82413}
```

Displaying arc-degrees, minutes, seconds: PDMS

`PDMS` (Print in Degrees Minutes Seconds) is a function that converts an angle given in radians into a list of three numbers corresponding to that angle in degrees, minutes, and seconds. **`PPDMS`** (Pretty Print) is a function that converts an angle given in radians into a text string corresponding to that angle in degrees, minutes, and seconds.

`PDMS[α]`	convert an angle α radians into a list {*degrees, minutes, seconds*}.
`PDMS[{α ..}]`	convert one or more lists of angles given in radians into {*degree, minute, seconds*} lists.
`PPDMS[α]`	Returns a String of the form DD°MM' SS.SSS"
`PPDMSϕ[α]`	Returns a String of the form DD°MM' SS.SSS" N or DD°MM' SS.SSS" S
`PPDMSλ[α]`	Returns a String of the form DD°MM' SS.SSS" E or DD°MM' SS.SSS" W

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Convert 0.123 radians into a DMS angle.

```
PDMS[0.123]
PPDMS[0.123]
```

```
{7, 2, 50.5712}
```

```
7°2'50.5712"
```

PDMS is the inverse function for DMS and vice versa

```
PDMS[DMS[1, 20, 39.6, w]]
```

```
{-1, -20, -39.6}
```

```
DMS[PDMS[0.123]]
```

```
0.123
```

PDMS is listable

```
PDMS[{0.123}]
PPDMS[{0.123, 0.123, 0.123}]
```

```
{{7, 2, 50.5712}}
```

```
{7°2'50.5712", 7°2'50.5712", 7°2'50.5712"}
```

Notice that using exact irrational numbers (a) works and (b) produces exact results.

```
PDMS[Table[α, {α, 0, π, π / 10}]]
PPDMS[Table[α, {α, 0, π, π / 10}]]
```

```
{{0, 0, 0}, {18, 0, 0}, {36, 0, 0}, {54, 0, 0}, {72, 0, 0},
 {90, 0, 0}, {108, 0, 0}, {126, 0, 0}, {144, 0, 0}, {162, 0, 0}, {180, 0, 0}}
```

```
{0°0'0", 18°0'0", 36°0'0", 54°0'0", 72°0'0",
 90° 0' 0", 108°0'0", 126°0'0", 144°0'0", 162°0'0", 180°0'0"}
```

PPDMSϕ is used to display latitude angles

```
PPDMSϕ[DMS[41, 24, 5.0775, S]]
```

```
41°24'5.0775"S
```

```
PPDMSϕ[DMS[41, 24, 5.0775, N]]
```

```
41°24'5.0775"N
```

PPDMSλ is used to display longitude angles

```
PPDMSλ[DMS[41, 24, 5.0775, E]]
```

```
41°24'5.0775"E
```

```
PPDMSλ[DMS[41, 24, 5.0775, W]]
```

```
41°24'5.0775"W
```

gon, centigon, milligon

The gon is an angular unit subdividing the circle into 400 parts. Therefore, 400 gon = 2π radians. Gons are often used as centigons (100 centigon = 1 gon) and milligons (1000 milligons = 1 gon)

`gon`	convert an angle expressed in gons into radians
`centigon`	convert an angle expressed in centigons into radians
`milligon`	convert an angle expressed in milligons into radians

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The relationship between gons and radians

```
400 gon
```

```
2 π
```

The relationship between gons and degrees

```
1 gon * 180 / π
```

$$\frac{9}{10}$$

The relationship between centigons and milligons as arc-seconds

```
PDMS[{1 centigon, 1 milligon}]
```

```
{{0, 0, 32.4}, {0, 0, 3.24}}
```

The relationship between various degrees in gons

```
Transpose[Table[{θ "°", θ ° / gon // N }, {θ, 0, 90, 15}]] // TableForm
```

```
0       15 °          30 °          45 °      60 °          75 °          90 °
0.      16.6667       33.3333       50.       66.6667       83.3333       100.
```

Example: Suppose a total station measures a zenith angle of 98.7912037 gon and a slant distance of 109.72 USFt. What is the vertical distance in meters?

```
109.72 USFt * Cos[98.7912037 gon]
```

```
0.634963
```

mil

The mil is a military unit of measure with different definitions for different military organizations

`milNATO`	6400 milNATO == 2π radians
`milUK`	1000 milUK == 1 radian
`milUS`	4000 milUS == 2π radians
`milUSSR`	6300 milUSSR == 2π radians

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The relationship between milNATO gons and radians

```
6400 milNATO
```

```
2 π
```

The relationship between milUK gons and radians

```
1000 milUK
```

```
1
```

The relationship between milUS gons and radians

```
4000 milUS
```

```
2 π
```

The relationship between milUSSR gons and radians

```
6300 milUSSR
```

```
2 π
```

The relationship between mils and degrees/minutes/seconds

```
PDMS /@ {milNATO, milUK, milUS, milUSSR}
```

```
{{0, 3, 22.5}, {0, 3, 26.2648}, {0, 5, 24.}, {0, 3, 25.7143}}
```

The relationship between various degrees in various mils

```
TableForm[
 Transpose[Table[{θ "°", θ°/milNATO // N , θ°/milUK // N , θ°/milUS // N , θ°/milUSSR // N},
   {θ, 15, 90, 15}]], TableHeadings → {{"", "NATO", "UK", "US", "USSR"}, None}
]
```

```
         15°        30°        45°        60°        75°        90°
NATO     266.667    533.333    800.       1066.67    1333.33    1600.
UK       261.799    523.599    785.398    1047.2     1309.      1570.8
US       166.667    333.333    500.       666.667    833.333    1000.
USSR     262.5      525.       787.5      1050.      1312.5     1575.
```

circle

The circle is an angular unit of measure that subdivides the circle into one part.

`circle` 1 circle == 2π radians

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

circles and radians

```
1 circle
```

```
2 π
```

milli-arcseconds (mas)

The mas is one milli-arcsecond. This unit appears in astronomy and geodesy.

`mas` 1 mas == $4.84813681 \times 10^{-9}$ radians

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

mas and radians

```
1 mas
```

```
4.84814 × 10^-9
```

There are 3600×360×1000 = 1296000000 mas in 2π radians

```
3600 * 360 * 1000 mas / (2 π)

1.
```

Reference Ellipsoid names

NAME	SEMIMAJOR	SEMIMINOR or FLATTENING^{-1}
Airy1830	6377563.396	299.3249646
AustralianNational	6378160	298.25
Bessel1841	6377397.155	299.1528128
Bessel1841Namibia	6377483.865	299.1528128
Clarke1866	6378206.4	6356583.8
Clarke1880	6378249.145	293.465
Everest	6377298.556	300.8017
Everest1830	6377276.345	300.8017
Everest1956	6377301.243	300.8017
EverestPakistan	6377309.613	300.8017
Everest1948	6377304.063	300.8017
Everest1969	6377295.664	300.8017
GRS80	6378137	6356752.3141
		see Moritz for definition.
Hayford1909	6378388	297
Helmert1906	6378200	298.3
Hough1960	6378270	297
Indonesian1974	6378160	298.247
International1924	6378388	297
Krassovsky	6378245	298.3
ModifiedAiry	6377340.189	299.3249646
ModifiedFischer1960	6378155	298.3
SouthAmerican1969	6378160	298.25
WGS72	6178135	298.26
WGS84	6378137	298.257223563
		see NIMA for definition

The ellipsoid names have values of the text strings of their names. For example, `Krassovsky` has the value `"Krassovsky"`.

For definitions of the local geodetic datums, see NIMA-A1

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

`$ellipsoidNames`	A global variable whose value is a list of the ellipsoid names.

```
$ellipsoidNames
```

AddEllipsoid

Users can add custom ellipsoids if they so desire.

`AddEllipsoid`[*name*,*a*,*p*]	Add a new ellipsoid named *name* with semimajor axis length *a* and either semiminor axis or flattening given by *p*.
name	*name* is added to `$ellipsoidNames.` *name* can be either a symbol or a text string and cannot be one of the package−defined ellipsoids.
a	*a* is the length of the semimajor axis in meters. For units conversions,see Units .
p	*p* is a rule of the form *Parameter*→*value* ,where *Parameter* can be either **`SemiminorAxis`** or **`InverseFlattening`** and *value* is a numeric quantity.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The name of the new ellipsoid is a text string. The semimajor axis is always provided but the second parameter can be either the semiminor axis or the inverse flattening. In this example, the length of the semiminor axis is provided.

```
AddEllipsoid[aNewEllipsoid = "foo", 6378137, SemiminorAxis → 6378000]
```

Check to see if it was, in fact, added to the known ellipsoid list.

```
$ellipsoidNames
```

Check its parameters. Note that the computed parameters function properly.

```
semiMajor[aNewEllipsoid]

semiMinor[aNewEllipsoid]

1 / flattening[aNewEllipsoid]

ρ[35 °, aNewEllipsoid] // P

Remove[aNewEllipsoid]
```

In this example, the inverse flattening is provided.

```
AddEllipsoid[anotherNewEllipsoid = "foobar", 6378137, InverseFlattening → 400]
```

```
$ellipsoidNames
```

```
semiMajor[anotherNewEllipsoid]
```

```
semiMinor[anotherNewEllipsoid]
```

```
flattening[anotherNewEllipsoid]
```

```
Remove[anotherNewEllipsoid]
```

You cannot change a system-defined ellipsoid

```
AddEllipsoid[GRS80, 6378000, InverseFlattening → 400];
semiMajor[GRS80]
```

defaultEllipsoid

It is often the case that one and only one ellipsoid will be used throughout a notebook or project. The package has a default ellipsoid that is set to GRS80 when the package is initialized

`defaultEllipsoid`	Evaluates to the name of the current default. Is initialized to GRS80.
`SetDefaultEllipsoid[`*name*`]`	Sets the default to a new value. *name* must be a member of `$ellipsoidNames`

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The default is GRS80 at initialization.

```
originalDefaultEllipsoid = defaultEllipsoid
```

Set the default to a new value.

```
SetDefaultEllipsoid[WGS84]
```

Check that the default was changed.

```
defaultEllipsoid
```

You are not allowed to change the default to anything except a recognized ellipsoid.

```
SetDefaultEllipsoid[notAnEllipsoid]
```

```
SetDefaultEllipsoid["notAnEllipsoid"]
```

```
defaultEllipsoid
```

```
SetDefaultEllipsoid[originalDefaultEllipsoid]
Remove[originalDefaultEllipsoid]
```

Reference Ellipsoid Parameters

`semiMajor[`*ellipsoidName*`]`	returns the length of the length of the semimajor axis of *ellipsoidName*. If *ellipsoidName* is omitted, the length of the default ellipsoid is returned.
`semiMinor[`*ellipsoidName*`]`	returns the length of the length of the semiminor axis of *ellipsoidName*. If *ellipsoidName* is omitted, the length of the default ellipsoid is returned.
`flattening[`*ellipsoidName*`]`	returns the flattening of *ellipsoidName*. Flattening is defined as $\frac{(a-b)}{a}$, where a is semimajor and b is semiminor. If *ellipsoidName* is omitted, the flattening of the default ellipsoid is returned.
`eSq[`*ellipsoidName*`]`	returns the (first) eccentricity squared of *ellipsoidName*. If *ellipsoidName* is omitted, the value of the default ellipsoid is returned. $e^2=2f-f^2=\frac{a^2-b^2}{a^2}=1-b^2/a^2$.
`ϵSq[`*ellipsoidName*`]`	ESCeESCSq returns the (second) eccentricity squared of *ellipsoidName*. *N.B.*: the first letter of this function is an epsilon, note an 'e'. If *ellipsoidName* is omitted, the value of the default ellipsoid is returned. $\epsilon^2=e^2/(1-e^2)$.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The values of the default ellipsoid's semimajor axis. N.B. the second eccentricity squared is spelled with an epsilon.

```
{semiMajor[], semiMinor[], flattening[], eSq[], ϵSq[]} // P
```

```
{6378137., 6356752.314, 0.003352810688, 0.006694380023, 0.006739496775}
```

You can get the info on a specific ellipsoid,

```
{semiMajor[Everest], semiMinor[Everest],
  flattening[Everest], eSq[Everest], ϵSq[Everest]} // P
```

```
{6377298.556, 6356097.55, 0.003324449297, 0.00663784663, 0.006682202063}
```

but not on one that's unknown. Note that the error messages for flattening and the eccentricities come from semiMajor or semiMinor. This is because they are functions that call semiMajor or semiMinor and abort before returning a value.

```
{semiMajor[Foo], semiMinor[Bar], flattening[Baz], eSq[Bling], ϵSq[Blang]}
```

```
General::spell : Possible spelling error: new symbol
   name "Blang" is similar to existing symbols {Blank, Bling}. More…
```

```
{semiMajor[Foo], semiMinor[Bar], flattening[Baz], eSq[Bling], ϵSq[Blang]}
```

This is a list of the lengths of all the ellipsoids' parameters.

```
{#, semiMajor[#], semiMinor[#], flattening[#], eSq[#], ϵSq[#]} & /@ $ellipsoidNames // P //
 TableForm
```

```
Airy1830                 6377563.396     6356256.909     0.003340850641     0.00667054
AustralianNational       6378160.        6356774.719     0.003352891869     0.00669454
Bessel1841               6377397.155     6356079.        0.003342773182     0.00667437
Bessel1841Namibia        6377483.865     6356165.383     0.003342773182     0.00667437
Clarke1866               6378206.4       6356583.8       0.003390075304     0.00676865
Clarke1880               6378249.145     6356514.87      0.003407561379     0.00680351
Everest                  6377298.556     6356097.55      0.003324449297     0.00663784
Everest1830              6377276.345     6356075.413     0.003324449297     0.00663784
Everest1956              6377301.243     6356100.228     0.003324449297     0.00663784
EverestPakistan          6377309.613     6356108.571     0.003324449297     0.00663784
Everest1948              6377304.063     6356103.039     0.003324449297     0.00663784
Everest1969              6377295.664     6356094.668     0.003324449297     0.00663784
GRS80                    6378137.        6356752.314     0.003352810688     0.00669438
Hayford1909              6378388.        6356890.231     0.003370407819     0.00672945
Helmert1906              6378200.        6356818.17      0.003352329869     0.00669342
Hough1960                6378270.        6356794.343     0.003367003367     0.00672267
Indonesian1974           6378160.        6356774.504     0.003352925595     0.00669460
International1924        6378388.        6356911.946     0.003367003367     0.00672267
Krassovsky               6378245.        6356863.019     0.003352329869     0.00669342
ModifiedAiry             6377340.189     6356034.448     0.003340850641     0.00667054
ModifiedFischer1960      6378155.        6356773.32      0.003352329869     0.00669342
SouthAmerican1969        6378160.        6356774.719     0.003352891869     0.00669454
WGS72                    6178135.        6157421.076     0.003352779454     0.00669431
WGS84                    6378137.        6356752.314     0.003352810665     0.00669437
```

`MeridionalArc[ϕ1,ϕ2,`*`ellipsoidName`*`]`	The length of the meridian between two latitudes is the integral of ρ from $\phi1$ to $\phi2$. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

Distance along the meridian of the default ellipsoid from 30° to 45°

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

```
MeridionalArc[30 °, 45 °] // P
```

```
1664830.98
```

Distance along the meridian from 30° to 45° for all the various ellipsoids

```
{#, MeridionalArc[30 °, 45 °, #] // P} & /@ $ellipsoidNames // TableForm
```

```
Airy1830                 1664699.001
AustralianNational       1664836.863
Bessel1841               1664652.756
Bessel1841Namibia        1664675.389
Clarke1866               1664793.801
Clarke1880               1664779.01
Everest                  1664654.203
Everest1830              1664648.405
Everest1956              1664654.904
EverestPakistan          1664657.089
Everest1948              1664655.641
Everest1969              1664653.448
GRS80                    1664830.98
Hayford1909              1664870.385
Helmert1906              1664848.138
Hough1960                1664844.637
Indonesian1974           1664836.813
International1924        1664875.437
Krassovsky               1664859.884
ModifiedAiry             1664640.739
ModifiedFischer1960      1664836.392
SouthAmerican1969        1664836.863
WGS72                    1612626.2
WGS84                    1664830.98
```

`MeridianQuadrant[`*`ellipsoidName`*`]`	The length of the meridian from the equator to the pole. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

Distance along the meridian from the equator to the pole of the default ellipsoid

---

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

```
MeridianQuadrant[] // P
```

```
10001965.73
```

Distance along the meridian from the equator to the pole for all the various ellipsoids

```
(MeridianQuadrant[#] // P) & /@ $ellipsoidNames
```

```
{10001126.08, 10002001.39, 10000855.76, 10000991.74, 10001888.04, 10001867.55,
 10000792.85, 10000758.02, 10000797.06, 10000810.19, 10000801.48, 10000788.31,
 10001965.73, 10002271.26, 10002066.93, 10002103.26, 10002001.22, 10002288.3,
 10002137.5, 10000776.05, 10001996.36, 10002001.39, 9688329.916, 10001965.73}
```

`MeanRadius[MeanRadiusGeometric, ellipsoidName]`	$(2a+b)/3$, where a is the semimajor axis and b is the length of the semiminor axis. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.
`MeanRadius[MeanRadiusMass,ellipsoidName]`	$\sqrt[3]{a^2 b}$, where a is the semimajor axis and b is the length of the semiminor axis. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.
`MeanRadius[ellipsoidName]`	$(2a+b)/3$, where a is the semimajor axis and b is the length of the semiminor axis. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Mean radius can be computed either geometrically or as the radius of a sphere with the same mass as the ellipsoid. The former is "geometric" and the latter is "mass." Both varieties are provided.

If MeanRadius[] is called without arguments, the default are geometric and to compute on the default ellipsoid.

```
MeanRadius[] // P
```

```
6371008.771
```

The first argument can be supplied explicitly controlling whether to compute geometric or mass. If the second argument is omitted, the computation is performed on the default ellipsoid.

```
MeanRadius[MeanRadiusGeometric] // P
```

```
6371008.771
```

```
MeanRadius[MeanRadiusMass] // P
```

```
6371000.79
```

If both arguments are provided, then the ellipsoid can be stipulated explicitly. The following are the mean mass radii of all the reference ellipsoids given in US Survey Feet

```
({#, MeanRadius[MeanRadiusMass, #] / USFt} & /@ $ellipsoidNames) // P
```

```
{{Airy1830, 20900395.57}, {AustralianNational, 20902266.57},
 {Bessel1841, 20899837.37}, {Bessel1841Namibia, 20900121.49}, {Clarke1866, 20902158.68},
 {Clarke1880, 20902176.51}, {Everest, 20899642.28}, {Everest1830, 20899569.49},
 {Everest1956, 20899651.09}, {EverestPakistan, 20899678.52}, {Everest1948, 20899660.33},
 {Everest1969, 20899632.81}, {GRS80, 20902191.76}, {Hayford1909, 20902891.3},
 {Helmert1906, 20902401.58}, {Hough1960, 20902528.4}, {Indonesian1974, 20902266.33},
 {International1924, 20902915.1}, {Krassovsky, 20902549.05},
 {ModifiedAiry, 20899664.08}, {ModifiedFischer1960, 20902254.11},
 {SouthAmerican1969, 20902266.57}, {WGS72, 20246752.9}, {WGS84, 20902191.76}}
```

`AuthalicSphereRadius[`*`ellipsoidName`*`]`	The radius of a sphere whose area is equal to that of a specified ellipsoid. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

A sphere whose area is equal to that of a specified ellipsoid.

```
AuthalicSphereRadius[] // P
```

```
6371007.181
```

The names of the ellipsoids and their authalic spherical radii in miles, sorted smallest to largest.

```
Sort[{#, AuthalicSphereRadius[#] / USMile // N} & /@ $ellipsoidNames,
  OrderedQ[{#1[[2]], #2[[2]]}] &] // TableForm
```

```
WGS72                    3834.62
Everest1830              3958.26
Everest1969              3958.27
Everest                  3958.27
Everest1956              3958.27
Everest1948              3958.27
ModifiedAiry             3958.27
EverestPakistan          3958.28
Bessel1841               3958.31
Bessel1841Namibia        3958.36
Airy1830                 3958.41
Clarke1866               3958.75
Clarke1880               3958.75
GRS80                    3958.75
WGS84                    3958.75
ModifiedFischer1960      3958.76
Indonesian1974           3958.77
AustralianNational       3958.77
SouthAmerican1969        3958.77
Helmert1906              3958.79
Hough1960                3958.82
Krassovsky               3958.82
Hayford1909              3958.88
International1924        3958.89
```

Radius of Curvature in the Meridian

$\rho[\phi,$*ellipsoidName*$]$	ESC r ESC the radius of curvature in the meridian at geodetic latitude ϕ on *ellipsoidName* . *ellipsoidName* is optional and, if omitted, `defaultEllipsoid` is used. $\rho = \frac{a(1-e^2)}{(1-e^2\sin^2\phi)^{3/2}}$

The radius of curvature in the meridian is the radius of an osculating circle tangent to and in the plane of a meridian at a particular geodetic latitude.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The radius of curvature in the meridian is denoted by the greek letter rho (ESC r ESC)

```
ρ[30 °] // P
```

```
6351377.104
```

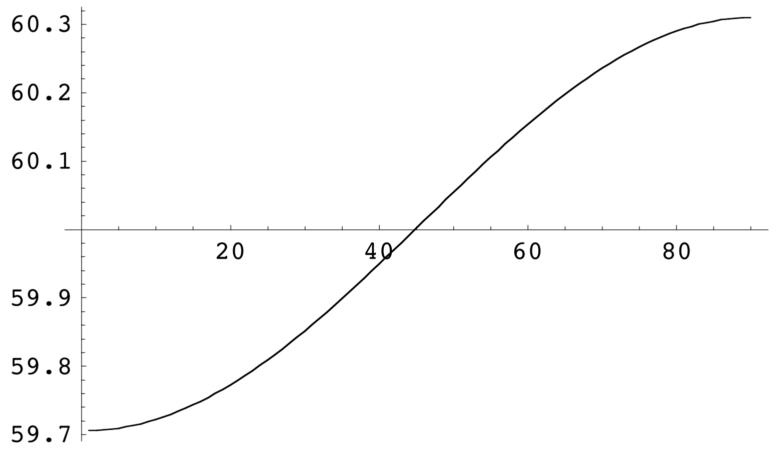
If you supply an ellipsoid name, you'll get different answers because each ellipsoid has a different shape. The following example picks a name at random from the list of all the names, then prints that name along with that ellipsoid's ρ at 59°14'5"N.

```
With[{n = $ellipsoidNames⟦Random[Integer, {1, Length[$ellipsoidNames]}]⟧},
 {n, ρ[DMS[59, 14, 5, N], n] // P}]
```

```
{Everest, 6381825.687}
```

It is well-known that one degree of latitude nominally spans sixty nautical miles. The following example computes the exact distance between each degree of latitude on the GRS 80 ellipsoid.

```
ListPlot[Table[NIntegrate[ρ[ϕ, GRS80], {ϕ, ϕ0 °, (ϕ0 + 1) °}] / NM, {ϕ0, 0, 89, 1}],
  PlotJoined → True];
```



Radius of Curvature in the Prime Vertical

ν[ϕ,*ellipsoidName*]	ESC n ESC the radius of curvature in the prime vertical at geodetic latitude ϕ on *ellipsoidName* . *ellipsoidName* is optional and, if omitted, `defaultEllipsoid` is used. $\nu = \frac{a}{(1-e^2 \sin^2 \phi)^{1/2}}$

The radius of curvature in the prime vertical (RCPV) is the radius of an osculating circle tangent to a parallel of latitude at that latitude. However, unlike the radius of curvature in the meridian, the RCPV is *not* in the plane of the parallel. Rather, the RCPV is normal to the ellipsoid, being the length of a line segment that begins on the surface of the ellipsoid and is produced inwards to the semiminor axis. The osculating circle of the RCPV is perpendicular to the osculating circle of the radius of curvature in the meridian. The radius of curvature in the prime vertical is given by

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

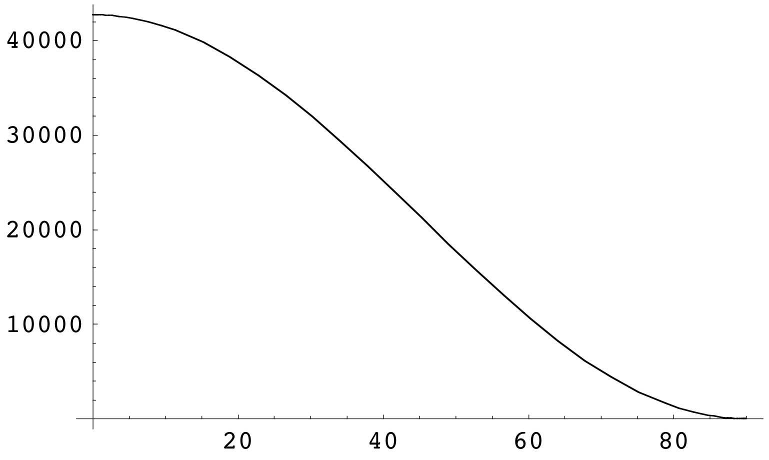
The radius of curvature in the prime vertical is denoted by the greek letter nu (ESC n ESC).

```
ν[30 °] // P
```

```
6383480.918
```

$\nu > \rho$ for $\phi < 90°$.

```
Plot[ν[ϕ °] - ρ[ϕ °], {ϕ, 0, 90}];
```



Radius of Curvature in the Normal Section

η[ϕ,α,*ellipsoidName*]	⎋h⎋ the radius of curvature in the normal section at geodetic latitude ϕ in the direction azimuth α on *ellipsoidName* . *ellipsoidName* is optional and, if omitted, `defaultEllipsoid` is used. $\frac{1}{\eta} = \frac{\cos^2 \alpha}{\rho} + \frac{\sin^2 \alpha}{\nu}$

The radius of curvature in the normal section (RCNS) is the radius of an osculating circle tangent to the ellipsoid in the plane of a normal section oriented at azimuth α and at geodetic latitude ϕ. The RCNS is a blending of ρ and ν, being equal to ρ for $\alpha = 0°/180°$ and being equal to ν for $\alpha = 90°/270°$.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

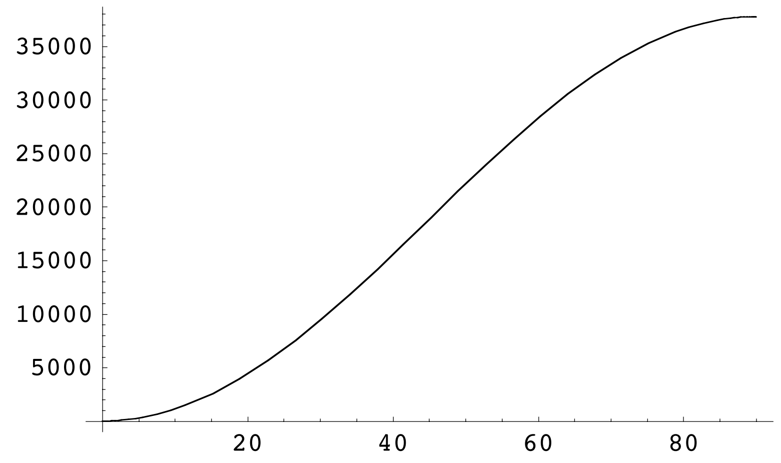
The radius of curvature in the normal section is denoted by the greek letter eta (⎋ *h* ⎋).

```
η[30 °, DMS[55, 0, 0]] // P
```

```
6372883.323
```

The following plot shows how RCNS varies from ρ to ν as α varies from 0° to 90°.

```
Plot[η[-20°, α°] - ρ[-20°], {α, 0, 90}];
```



Geodetic Datum Names

Symbol Name	Datum Name	Reference Ellipsoid
ETRS89	European Terrestrial Reference System of 1989	GRS 80
GDA94	Geodetic Datum of Australia of 1994	GRS 80
SIRGAS	South American Geocentric Reference System	GRS 80
ITRF92	International Terrestrial Reference Frame, 1992 Realization	GRS 80
ITRF93	International Terrestrial Reference Frame, 1993 Realization	GRS 80
ITRF94	International Terrestrial Reference Frame, 1994 Realization	GRS 80
ITRF96	International Terrestrial Reference Frame, 1996 Realization	GRS 80
ITRF97	International Terrestrial Reference Frame, 1997 Realization	GRS 80
ITRF00	International Terrestrial Reference Frame, 2000 Realization	GRS 80
NAD27	North American Datum of 1927	Clarke 1866
NAD831986	North American Datum of 1983 (1986)	GRS 80
NAD83HARN	North American Datum of 1983 (HARN)	GRS 80
NAD83CORS93	North American Datum of 1983 (CORS93)	GRS 80
NAD83CORS94	North American Datum of 1983 (CORS94)	GRS 80
NAD83CORS96	North American Datum of 1983 (CORS96)	GRS 80
WGS84Original	World Geodetic System of 1984, original adjustment. 1 Jan 87 – 1 Jan 94	WGS 84
WGS84G730	World Geodetic System of 1984, first upgrade. 2 Jan 94 – 28 Sept 96	WGS 84
WGS84G873	World Geodetic System of 1984, second upgrade, closely aligned with ITRF94. 29 Sept 96 onwards	WGS 84

The datum symbols have values of the text strings of their names. For example, `NAD27` has the value `"NAD 27 North American Datum of 1927"`.

For definitions of the local geodetic datums, see NIMA-A1

`$datumNames`	A global variable whose value is a list of the datum names.

This loads the package.

```
Needs["GeometricalGeodesy`"]

$datumNames

{ETRS89, GDA94, SIRGAS, ITRF92, ITRF93, ITRF94, ITRF96, ITRF97,
 ITRF00, NAD 27, NAD 83 (1986), NAD 83 (HARN), NAD 83 (CORS93),
 NAD 83 (CORS94), NAD 83 (CORS96), WGS 84, WGS 84 (G730), WGS 84 (G873)}
```

`referenceEllipsoid[`*`datumName`*`]`	A function that returns the name of the reference ellipsoid associated with a particular datum.

This loads the package.

```
Needs["GeometricalGeodesy`"]

{#, referenceEllipsoid[#]} & /@ $datumNames // TableForm

ETRS89              GRS80
GDA94               GRS80
SIRGAS              GRS80
ITRF92              GRS80
ITRF93              GRS80
ITRF94              GRS80
ITRF96              GRS80
ITRF97              GRS80
ITRF00              GRS80
NAD 27              Clarke1866
NAD 83 (1986)       GRS80
NAD 83 (HARN)       GRS80
NAD 83 (CORS93)     GRS80
NAD 83 (CORS94)     GRS80
NAD 83 (CORS96)     GRS80
WGS 84              WGS84
WGS 84 (G730)       WGS84
WGS 84 (G873)       WGS84
```

`defaultDatum`	A variable that returns the current default datum.

```
originalDefaultDatum = defaultDatum

NAD 83 (CORS96)
```

`SetDefaultDatum`	A function that sets the current default datum.

```
SetDefaultDatum[ITRF00]

ITRF00

defaultDatum

ITRF00

SetDefaultDatum[originalDefaultDatum]
Remove[originalDefaultDatum]
```

`AddDatum[`*`datumName, ellipsoidName`*`]`	Adds a new datum name to the list of recognized datums.

The user is allowed to add datums to the list that the system knows about. All that is required is the name of the datum (a text string) and the name of the reference ellipsoid associated with that datum. The name of the reference ellipsoid must already be on the list of recognized ellipsoids (see $ellipsoidNames)

```
$ellipsoidNames

{Airy1830, AustralianNational, Bessel1841, Bessel1841Namibia, Clarke1866, Clarke1880,
 Everest, Everest1830, Everest1956, EverestPakistan, Everest1948, Everest1969,
 GRS80, Hayford1909, Helmert1906, Hough1960, Indonesian1974, International1924,
 Krassovsky, ModifiedAiry, ModifiedFischer1960, SouthAmerican1969, WGS72, WGS84}

AddDatum["Bogota Observatory", International1924]

Bogota Observatory

$datumNames

{ETRS89, GDA94, SIRGAS, ITRF92, ITRF93, ITRF94, ITRF96, ITRF97, ITRF00,
 NAD 27, NAD 83 (1986), NAD 83 (HARN), NAD 83 (CORS93), NAD 83 (CORS94),
 NAD 83 (CORS96), WGS 84, WGS 84 (G730), WGS 84 (G873), Bogota Observatory}
```

You are not allowed to change a system-established datum.

```
AddDatum[NAD27, International1924]

AddDatum::noChange : you are not allowed to change a system datum (NAD 27)
```

You must use a reference ellipsoid known to the system.

```
AddDatum["Dabola", "Clarke18880"]

AddDatum::unknownEllipsoid : you must provide a known ellipsoid (Clarke18880)
```

Helmert 7 Parameter Datum Transformation

`χHelmert[ϵ,T,s,p,targetDatum]`	returns an **`xyz`** object in the **`targetDatum`** datum, which is a seven−parameter Helmert transformation of **`p`** given the differential rotation matrix **`ϵ`**, the translation vector **`T`**, and the scale factor **`s`**. **`ϵ`** is a vector **`{ϵx_?NumericQ,ϵy_?NumericQ,ϵz_?NumericQ}`**. **`T`** is a vector **`{Δx_?NumericQ,Δy_?NumericQ,Δz_?NumericQ}`**. **`s`** is a positive scalar and **`p`** is an **`xyz`** object.

Positions can be transformed into the equivalent coordinates in a different geodetic datums by many different methods. This package implements the seven-parameter Helmert method. This method is not the most rigorous available; it does not consider the time-varying aspect of the transformation parameters.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Create a point to play with

```
LX3030 = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, ITRF00];
LX3030 // PPDMS
```

```
41°48'44.7844"N, 72°15'2.04232"W, 157.303, ITRF00
```

Here is that point as an **`xyz`** object

```
LX3030 ⇒ xyz // P
```

```
{1451423.342, -4534399.864, 4230204.318, ITRF00, GRS80}
```

Move it to NAD 83(CORS96) 2002.5

```
x = χHelmert[{25.915 mas, 9.426 mas, 11.599 mas},
  {0, 0, 0}, 0.62 × 10^-9, LX3030 ⇒ xyz, NAD83CORS96]; x // P
```

```
{1451422.895, -4534399.417, 4230204.957, NAD 83 (CORS96), GRS80}
```

Here's the change. Note the use of the ⊖ operator because the two objects are in different datums.

```
(LX3030 ⇒ xyz) ⊖ x // P
```

```
{0.4473996195, -0.4470506106, -0.6386502264, None, None}
```

```
Remove[LX3030, x]
```

MakeBLH

This package implements an object-oriented representation of points in various coordinate systems. For example, geodetic coordinates are of type **`blh`** and have the form **`blh[ϕ,λ,h,datumName]`**. Similarly, geocentric Cartesian coordinates are of type **`xyz`** and have the form **`xyz[x,y,z,datumName]`**. This approach was adopted because it is common in geodesy to have many coordinates for a single place. This implementation makes it very simple to disambiguate the coordinates -- it's easy to know what coordinate system and datum a set of coordinates is in.

`blh`	a data type indicating a point in a geodetic coordinate system.
`MakeBLH[ϕ,λ,h,`*`datumName`*`]`	Creates a new point in a geodetic coordinate system with coordinates `ϕ,λ,h`. *`datumName`* is optional and, if omitted,is supplied with the value *defaultDatum*.
`MakeBLH[{ϕ,λ,h,`*`datumName`*`}]`	Creates a new point in a geodetic coordinate system from a list of coordinates `{ϕ,λ,h}`. The datum is supplied explicitly.
`MakeBLH[{ϕ,λ,h},`*`datumName`*`]`	Creates a new point in a geodetic coordinate system from a list of coordinates `{ϕ,λ,h}`. *`datumName`* is optional and, if omitted, is supplied with the value *defaultDatum*.
`MakeBLH[{{ϕ,λ,h,`*`datumName`*`}..}]`	Creates a list of new points in a geodetic coordinate system from a list of coordinates `{ϕ,λ,h}`. *`datumName`* is optional and, if omitted, is supplied with the value *defaultDatum*.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Create a new point in a geodetic coordinate system. If you leave off the ellipsoid, the default is used.

```
ABQ1[blh, NAD83CORS96] =
 MakeBLH[DMS[34, 57, 26.54647, N], DMS[106, 29, 40.03825, W], 1720.503]
```

```
blh[0.610121, -1.85868, 1720.5, NAD 83 (CORS96)]
```

You can specify a particular datum.

```
ABQ1[blh, ITRF00] =
 MakeBLH[DMS[34, 57, 26.56675, N], DMS[106, 29, 40.07414, W], 1719.534, ITRF00]
```

```
blh[0.610121, -1.85868, 1719.53, ITRF00]
```

```
Remove[ABQ1]
```

The datum must be one of the ones in *$datumNames*

```
MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, "fooBar"]

MakeBLH::badDatum : attempt to use an unrecognized
   datum fooBar. See $datumNames for a list of recognized datums.

MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, notARealDatumName]

MakeBLH::badDatum : attempt to use an unrecognized datum
   notARealDatumName. See $datumNames for a list of recognized datums.
```

Sometimes coordinates come in lists, so you can use them that way without explicitly pulling them out.

```
MakeBLH[{DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303}]

blh[0.729765, -1.26101, 157.303, NAD 83 (CORS96)]
```

You can create lists of points at once, either the same place in different datums or different places, or both.

```
MakeBLH[
   {
    {DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303},
    {DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, NAD27},
    {DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, ITRF00}}] // PPDMS //
 TableForm

41°48'44.7844"N, 72°15'2.04232"W, 157.303, NAD 83 (CORS96)
41°48'44.7844"N, 72°15'2.04232"W, 157.303, NAD 27
41°48'44.7844"N, 72°15'2.04232"W, 157.303, ITRF00
```

BLH Extraction Functions

The following group of functions extract pieces from the blh data structure.

`blhE[`*list*`]`	extracts any item in *list* that is of type `blh`.
`blhQ[`*p*`]`	returns `True` iff *p* is of type `blh`.
`latE[`*p*`]`	extract the geodetic latitude of *p e blh*.
`lonE[`*p*`]`	extract the geodetic longitude of *p e blh*.
`hgtE[`*p*`]`	extract the ellipsoid height of *p e blh*.
`datumE[`*p*`]`	extract the datum *p e blh* is defined on.
`ellipE[`*p*`]`	extract the ellipsoid *p e blh* is defined on. This is a shorthand for **`referenceEllipsoid[datumE[p]]`**.
β`E[`*p*`]`	extract the reduced latitude from *p*. Reduced latitude has the form $\tan[\beta]=\sqrt{1-e^2}\,\tan[\phi]$
ψ`E[p]`	extract the geocentric latitude from *p*. Geocentric latitude has the form $\tan[\psi]=(1-e^2)\tan[\phi]$

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Create a new point in a geodetic coordinate system. If you leave off the ellipsoid, the default is used.

```
lx3030 = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, ITRF00]
```

```
blh[0.729765, -1.26101, 157.303, ITRF00]
```

You can test if a point is in geodetic coordinates.

```
blhQ[lx3030]
```

```
True
```

```
blhQ["Hello, World!"]
```

```
False
```

Extraction functions pull the coordinates and ellipsoid information out of the data structure

```
{latE[lx3030] // PDMS, lonE[lx3030] // PDMS, hgtE[lx3030], datumE[lx3030], ellipE[lx3030]}
```

```
{{41, 48, 44.7844}, {-72, -15, -2.04232}, 157.303, ITRF00, GRS80}
```

You can display the coordinates as pretty-printed strings:

```
lx3030 // PPDMS
```

```
41°48'44.7844"N, 72°15'2.04232"W, 157.303, ITRF00
```

You can display the coordinates as lists of numbers:

```
lx3030 // P
```

```
{{41, 48, 44.7844}, {-72, -15, -2.04232}, 157.303, ITRF00, GRS80}
```

You can display the coordinates in decimal degrees:

```
lx3030 // PDD // P
```

```
{41.81244011, -72.25056731, 157.303, ITRF00}
```

It can be convenient to keep all the coordinates for a point in a list:

```
LX3030 = {lx3030, lx3030 ⇒ xyz, lx3030 ⇒ xyz ⇒ ITRF00}
```

```
{blh[0.729765, -1.26101, 157.303, NAD 83 (CORS96)],
 xyz[1.45142 × 10^6, -4.5344 × 10^6, 4.2302 × 10^6, NAD 83 (CORS96)],
 xyz[1.45142 × 10^6, -4.5344 × 10^6, 4.2302 × 10^6, NAD 83 (CORS96)] ⇒ ITRF00}
```

blhE pull all the BLH points out of a list:

```
LX3030 // blhE
```

```
{blh[0.729765, -1.26101, 157.303, NAD 83 (CORS96)]}
```

Reference Ellipsoid Parameters

`semiMajor[ellipsoidName]`	returns the length of the length of the semimajor axis of *ellipsoidName*. If *ellipsoidName* is omitted, the length of the default ellipsoid is returned.
`semiMinor[ellipsoidName]`	returns the length of the length of the semiminor axis of *ellipsoidName*. If *ellipsoidName* is omitted, the length of the default ellipsoid is returned.
`flattening[ellipsoidName]`	returns the flattening of *ellipsoidName*. Flattening is defined as $\frac{(a-b)}{a}$, where a is semimajor and b is semiminor. If *ellipsoidName* is omitted, the flattening of the default ellipsoid is returned.
`eSq[ellipsoidName]`	returns the (first) eccentricity squared of *ellipsoidName*. If *ellipsoidName* is omitted, the value of the default ellipsoid is returned. $e^2 = 2f - f^2 = \frac{a^2-b^2}{a^2} = 1 - b^2/a^2$.
`ϵSq[ellipsoidName]`	ESCeESCSq returns the (second) eccentricity squared of *ellipsoidName*. *N.B.*: the first letter of this function is an epsilon, note an '*e*'. If *ellipsoidName* is omitted, the value of the default ellipsoid is returned. $\epsilon^2 = e^2/(1-e^2)$.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The values of the default ellipsoid's semimajor axis. N.B. the second eccentricity squared is spelled with an epsilon.

```
{semiMajor[], semiMinor[], flattening[], eSq[], ϵSq[]} // P
```

```
{6378137., 6356752.314, 0.003352810688, 0.006694380023, 0.006739496775}
```

You can get the info on a specific ellipsoid,

```
{semiMajor[Everest], semiMinor[Everest],
  flattening[Everest], eSq[Everest], ϵSq[Everest]} // P
```

```
{6377298.556, 6356097.55, 0.003324449297, 0.00663784663, 0.006682202063}
```

but not on one that's unknown. Note that the error messages for flattening and the eccentricities come from semiMajor or semiMinor. This is because they are functions that call semiMajor or semiMinor and abort before returning a value.

```
{semiMajor[Foo], semiMinor[Bar], flattening[Baz], eSq[Bling], ϵSq[Blang]}
```

```
General::spell : Possible spelling error: new symbol
   name "Blang" is similar to existing symbols {Blank, Bling}. More…
```

```
{semiMajor[Foo], semiMinor[Bar], flattening[Baz], eSq[Bling], ϵSq[Blang]}
```

This is a list of the lengths of all the ellipsoids' parameters.

```
{#, semiMajor[#], semiMinor[#], flattening[#], eSq[#], ϵSq[#]} & /@ $ellipsoidNames // P //
 TableForm
```

```
Airy1830                 6377563.396     6356256.909     0.003340850641     0.00667054
AustralianNational       6378160.        6356774.719     0.003352891869     0.00669454
Bessel1841               6377397.155     6356079.        0.003342773182     0.00667437
Bessel1841Namibia        6377483.865     6356165.383     0.003342773182     0.00667437
Clarke1866               6378206.4       6356583.8       0.003390075304     0.00676865
Clarke1880               6378249.145     6356514.87      0.003407561379     0.00680351
Everest                  6377298.556     6356097.55      0.003324449297     0.00663784
Everest1830              6377276.345     6356075.413     0.003324449297     0.00663784
Everest1956              6377301.243     6356100.228     0.003324449297     0.00663784
EverestPakistan          6377309.613     6356108.571     0.003324449297     0.00663784
Everest1948              6377304.063     6356103.039     0.003324449297     0.00663784
Everest1969              6377295.664     6356094.668     0.003324449297     0.00663784
GRS80                    6378137.        6356752.314     0.003352810688     0.00669438
Hayford1909              6378388.        6356890.231     0.003370407819     0.00672945
Helmert1906              6378200.        6356818.17      0.003352329869     0.00669342
Hough1960                6378270.        6356794.343     0.003367003367     0.00672267
Indonesian1974           6378160.        6356774.504     0.003352925595     0.00669460
International1924        6378388.        6356911.946     0.003367003367     0.00672267
Krassovsky               6378245.        6356863.019     0.003352329869     0.00669342
ModifiedAiry             6377340.189     6356034.448     0.003340850641     0.00667054
ModifiedFischer1960      6378155.        6356773.32      0.003352329869     0.00669342
SouthAmerican1969        6378160.        6356774.719     0.003352891869     0.00669454
WGS72                    6178135.        6157421.076     0.003352779454     0.00669431
WGS84                    6378137.        6356752.314     0.003352810665     0.00669437
```

`MeridionalArc[ϕ1,ϕ2,`*`ellipsoidName`*`]`	The length of the meridian between two latitudes is the integral of ρ from $\phi 1$ to $\phi 2$. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

Distance along the meridian of the default ellipsoid from 30° to 45°

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

```
MeridionalArc[30 °, 45 °] // P
```

```
1664830.98
```

Distance along the meridian from 30° to 45° for all the various ellipsoids

```
{#, MeridionalArc[30 °, 45 °, #] // P} & /@ $ellipsoidNames // TableForm
```

```
Airy1830                1664699.001
AustralianNational      1664836.863
Bessel1841              1664652.756
Bessel1841Namibia       1664675.389
Clarke1866              1664793.801
Clarke1880              1664779.01
Everest                 1664654.203
Everest1830             1664648.405
Everest1956             1664654.904
EverestPakistan         1664657.089
Everest1948             1664655.641
Everest1969             1664653.448
GRS80                   1664830.98
Hayford1909             1664870.385
Helmert1906             1664848.138
Hough1960               1664844.637
Indonesian1974          1664836.813
International1924       1664875.437
Krassovsky              1664859.884
ModifiedAiry            1664640.739
ModifiedFischer1960     1664836.392
SouthAmerican1969       1664836.863
WGS72                   1612626.2
WGS84                   1664830.98
```

`MeridianQuadrant[`*ellipsoidName*`]`	The length of the meridian from the equator to the pole. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

Distance along the meridian from the equator to the pole of the default ellipsoid

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

```
MeridianQuadrant[] // P
```

```
10001965.73
```

Distance along the meridian from the equator to the pole for all the various ellipsoids

```
(MeridianQuadrant[#] // P) & /@ $ellipsoidNames
```

```
{10001126.08, 10002001.39, 10000855.76, 10000991.74, 10001888.04, 10001867.55,
 10000792.85, 10000758.02, 10000797.06, 10000810.19, 10000801.48, 10000788.31,
 10001965.73, 10002271.26, 10002066.93, 10002103.26, 10002001.22, 10002288.3,
 10002137.5, 10000776.05, 10001996.36, 10002001.39, 9688329.916, 10001965.73}
```

`MeanRadius[MeanRadiusGeometric, ellipsoidName]`	$(2a+b)/3$, where a is the semimajor axis and b is the length of the semiminor axis. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.
`MeanRadius[MeanRadiusMass,ellipsoidName]`	$\sqrt[3]{a^2 b}$, where a is the semimajor axis and b is the length of the semiminor axis. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.
`MeanRadius[ellipsoidName]`	$(2a+b)/3$, where a is the semimajor axis and b is the length of the semiminor axis. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Mean radius can be computed either geometrically or as the radius of a sphere with the same mass as the ellipsoid. The former is "geometric" and the latter is "mass." Both varieties are provided.

If MeanRadius[] is called without arguments, the default are geometric and to compute on the default ellipsoid.

```
MeanRadius[] // P
```

```
6371008.771
```

The first argument can be supplied explicitly controlling whether to compute geometric or mass. If the second argument is omitted, the computation is performed on the default ellipsoid.

```
MeanRadius[MeanRadiusGeometric] // P
```

```
6371008.771
```

```
MeanRadius[MeanRadiusMass] // P
```

```
6371000.79
```

If both arguments are provided, then the ellipsoid can be stipulated explicitly. The following are the mean mass radii of all the reference ellipsoids given in US Survey Feet

```
({#, MeanRadius[MeanRadiusMass, #] / USFt} & /@ $ellipsoidNames) // P
```

```
{{Airy1830, 20900395.57}, {AustralianNational, 20902266.57},
 {Bessel1841, 20899837.37}, {Bessel1841Namibia, 20900121.49}, {Clarke1866, 20902158.68},
 {Clarke1880, 20902176.51}, {Everest, 20899642.28}, {Everest1830, 20899569.49},
 {Everest1956, 20899651.09}, {EverestPakistan, 20899678.52}, {Everest1948, 20899660.33},
 {Everest1969, 20899632.81}, {GRS80, 20902191.76}, {Hayford1909, 20902891.3},
 {Helmert1906, 20902401.58}, {Hough1960, 20902528.4}, {Indonesian1974, 20902266.33},
 {International1924, 20902915.1}, {Krassovsky, 20902549.05},
 {ModifiedAiry, 20899664.08}, {ModifiedFischer1960, 20902254.11},
 {SouthAmerican1969, 20902266.57}, {WGS72, 20246752.9}, {WGS84, 20902191.76}}
```

`AuthalicSphereRadius[` *`ellipsoidName`*`]`	The radius of a sphere whose area is equal to that of a specified ellipsoid. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

A sphere whose area is equal to that of a specified ellipsoid.

```
AuthalicSphereRadius[] // P
```

```
6371007.181
```

The names of the ellipsoids and their authalic spherical radii in miles, sorted smallest to largest.

```
Sort[{#, AuthalicSphereRadius[#] / USMile // N} & /@ $ellipsoidNames,
  OrderedQ[{#1[[2]], #2[[2]]}] &] // TableForm
```

```
WGS72                    3834.62
Everest1830              3958.26
Everest1969              3958.27
Everest                  3958.27
Everest1956              3958.27
Everest1948              3958.27
ModifiedAiry             3958.27
EverestPakistan          3958.28
Bessel1841               3958.31
Bessel1841Namibia        3958.36
Airy1830                 3958.41
Clarke1866               3958.75
Clarke1880               3958.75
GRS80                    3958.75
WGS84                    3958.75
ModifiedFischer1960      3958.76
Indonesian1974           3958.77
AustralianNational       3958.77
SouthAmerican1969        3958.77
Helmert1906              3958.79
Hough1960                3958.82
Krassovsky               3958.82
Hayford1909              3958.88
International1924        3958.89
```

`ReducedLatitude[`ϕ`,`*ellipsoidName*`]`	An equivalent latitude for a point on a sphere, $\tan[\beta] = (1\text{-f})\text{Tan}[\phi]$ for the specified ellipsoid. If *ellipsoidName* is omitted,the computation is performed on the default ellipsoid.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The reduced latitude of 45° with respect to the default datum and reference ellipsoid

```
ReducedLatitude[45 °] // PDMS
```

```
{44, 54, 13.6363}
```

An ellipsoid can be stipulated explicitly: a table of reduced latitudes for the Indonesian1974 ellipsoid

```
Table[ReducedLatitude[ϕ °, Indonesian1974], {ϕ, -90, 90, 15}] // PDMS // MatrixForm
```

```
⎛-90  0    0        ⎞
⎜-74  -57  -6.55983 ⎟
⎜-59  -54  -59.7778 ⎟
⎜-44  -54  -13.6244 ⎟
⎜-29  -55  -0.281485⎟
⎜-14  -57  -7.06356 ⎟
⎜0    0    0        ⎟
⎜14   57   7.06356  ⎟
⎜29   55   0.281485 ⎟
⎜44   54   13.6244  ⎟
⎜59   54   59.7778  ⎟
⎜74   57   6.55983  ⎟
⎝90   0    0        ⎠
```

A datum can be stipulated explicitly: a table of reduced latitudes for the ITRF96 datum

```
Table[ReducedLatitude[ϕ °, ITRF96], {ϕ, -90, 90, 15}] // PDMS // MatrixForm
```

```
⎛-90  0    0        ⎞
⎜-74  -57  -6.56579 ⎟
⎜-59  -54  -59.7881 ⎟
⎜-44  -54  -13.6363 ⎟
⎜-29  -55  -0.291766⎟
⎜-14  -57  -7.06949 ⎟
⎜0    0    0        ⎟
⎜14   57   7.06949  ⎟
⎜29   55   0.291766 ⎟
⎜44   54   13.6363  ⎟
⎜59   54   59.7881  ⎟
⎜74   57   6.56579  ⎟
⎝90   0    0        ⎠
```

Reduced latitude can be computed from a BLH object, or a table of them

```
ReducedLatitude[MakeBLH[45 °, 0, 0, "NAD 83 (HARN)"]] // PDMS
```

```
{44, 54, 13.6363}
```

```
ReducedLatitude[Table[MakeBLH[Random[Real, {-π / 2, π / 2}], 0, 0], {4}]] // PPDMS
```

```
PPDMS[ReducedLatitude[
  {blh[-1.49261, 0, 0, NAD 83 (CORS96)], blh[-1.5632, 0, 0, NAD 83 (CORS96)],
   blh[-0.10351, 0, 0, NAD 83 (CORS96)], blh[-1.40356, 0, 0, NAD 83 (CORS96)]}]]
```

β is a common notational shorthand for reduced latitude. βE extracts the reduced latitude from a BLH object.

```
βE[MakeBLH[35 °, 0]] // PPDMS
βE[Table[MakeBLH[Random[Real, {-π / 2, π / 2}], 0, 0], {4}]] // PPDMS
```

```
34°54'34.7113"

{-83°34'43.86", 68°26'23.4522", -67°29'59.7257", -74°1'27.7717"}
```

`GeocentricLatitude[ϕ,ellipsoidName]`	Geocentric latitude is required for things such as the Somali normal gravity formula: tan[ψ]= $(1-e^2)$ tan[ϕ]. If ellipsoidName is omitted,the computation is performed on the default ellipsoid.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

The geocentric latitude of 45° with respect to the default datum and reference ellipsoid

```
GeocentricLatitude[45 °] // PDMS
```

```
{44, 48, 27.2764}
```

An ellipsoid can be stipulated explicitly: a table of geocentric latitudes for the Indonesian1974 ellipsoid

```
Table[GeocentricLatitude[ϕ °, Indonesian1974], {ϕ, -90, 90, 15}] // PDMS // MatrixForm
```

```
-90  0    0
-74  -54  -12.6149
-59  -49  -59.0535
-44  -48  -27.2526
-29  -50  -1.06839
-14  -54  -14.6299
0    0    0
14   54   14.6299
29   50   1.06839
44   48   27.2526
59   49   59.0535
74   54   12.6149
90   0    0
```

A datum can be stipulated explicitly: a table of geocentric latitudes for the ITRF96 datum

```
Table[GeocentricLatitude[ϕ °, ITRF96], {ϕ, -90, 90, 15}] // PDMS // MatrixForm
```

```
-90  0    0
-74  -54  -12.6269
-59  -49  -59.0741
-44  -48  -27.2764
-29  -50  -1.08891
-14  -54  -14.6417
0    0    0
14   54   14.6417
29   50   1.08891
44   48   27.2764
59   49   59.0741
74   54   12.6269
90   0    0
```

Geocentric latitude can be computed from a BLH object, or a table of them

```
GeocentricLatitude[MakeBLH[45 °, 0, 0, "NAD 83 (HARN)"]] // PDMS
```

```
{44, 48, 27.2764}
```

```
GeocentricLatitude[Table[MakeBLH[Random[Real, {-π / 2, π / 2}], 0, 0], {4}]] // PPDMS
```

```
{-57°45'10.3861", -16°5'33.8431", 75°47'43.5884", 40°31'26.3778"}
```

ψ is a common notational shorthand for reduced latitude. ψE extracts the geocentric latitude from a BLH object.

```
ψE[MakeBLH[35 °, 0]] // PPDMS
ψE[Table[MakeBLH[Random[Real, {-π / 2, π / 2}], 0, 0], {4}]] // PPDMS
```

```
34°49'9.79933"
```

```
{3°41'48.789", -85°5'1.45687", -11°10'17.5455", -37°10'11.0751"}
```

Authalic Latitude

"The authalic latitude on a sphere having the same surface area as the ellipsoid provides a sphere which is truly equal-area (authalic) relative to the ellipsoid." (Snyder,J.P, 1994).

`AuthalicLat[`*f*`, `*ellipsoidName*`|`*datumName*`]`	Converts geodetic latitude ϕ into authalic latitude. The *ellipsoidName* and *datumName* are optional and, if omitted, *defaultDatum* is used.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

AuthalicLat can be used without specifying an ellipsoid or a datum

```
AuthalicLat[45 °] // PDMS
```

```
{44, 52, 18.1303}
```

An ellipsoid can be stipulated explicitly: a table of authalic latitudes for the Clarke1866 ellipsoid

```
Table[AuthalicLat[ϕ°, Clarke1866], {ϕ, -90, 90, 15}] // PDMS // MatrixForm
```

```
-90  0    0
-74  -56  -6.10384
-59  -53  -15.1658
-44  -52  -12.9876
-29  -53  -15.9441
-14  -56  -6.88216
0    0    0
14   56   6.88216
29   53   15.9441
44   52   12.9876
59   53   15.1658
74   56   6.10384
90   0    0
```

A datum can be stipulated explicitly: a table of authalic latitudes for the SIRGAS datum

```
Table[AuthalicLat[ϕ°, SIRGAS], {ϕ, -90, 90, 15}] // PDMS // MatrixForm
```

```
-90  0    0
-74  -56  -8.68374
-59  -53  -19.628
-44  -52  -18.1303
-29  -53  -20.3893
-14  -56  -9.44502
0    0    0
14   56   9.44502
29   53   20.3893
44   52   18.1303
59   53   19.628
74   56   8.68374
90   0    0
```

Authalic latitude can be computed from a BLH object, or a table of them

```
AuthalicLat[MakeBLH[45°, 0, 0, SIRGAS]] // PDMS
```

```
{44, 52, 18.1303}
```

```
AuthalicLat[Table[MakeBLH[Random[Real, {-π/2, π/2}], 0, 0], {4}]] // PPDMS
```

```
{4°10'3.18268", -36°6'57.0516", 47°50'4.95868", -51°25'8.52613"}
```

Polar Tests

Several tests are provided to check if points are at the poles or are antipodal to one another.

`northPoleQ[p]`	returns True iff $p\ e\ blh$ has latitude 90 °N
`southPoleQ[p]`	returns True iff $p\ e\ blh$ has latitude 90 °S
`antipodalQ[p,q]`	returns True iff p is antipodal to q, $p \in \mathtt{blh} \wedge q \in \mathtt{blh}$

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

It can be handy to check if a point is the north or south pole

```
lx3030 = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, ITRF00]

{northPoleQ[lx3030], northPoleQ[MakeBLH[90°, 0, 0]], southPoleQ[MakeBLH[-π/2, 0, 0]]}

{False, True, True}
```

It can also be handy to check if two points are antipodal

```
antipodalQ[MakeBLH[90°, 0, 0], MakeBLH[-π/2, 0, 0]]

True

antipodalQ[MakeBLH[-90°, 0, 0], MakeBLH[π/2, 0, 0]]

True

antipodalQ[MakeBLH[0, 0, 0], MakeBLH[0, π, 0]]

True

antipodalQ[lx3030, MakeBLH[0, π, 0]]

False

Remove[lx3030]
```

MakeXYZ

This package implements an object-oriented representation of points in various coordinate systems. For example, geodetic coordinates are of type **blh** and have the form **blh[ϕ,λ,h,datumName]**. Similarly, geocentric Cartesian coordinates are of type **xyz** and have the form **xyz[x,y,z,datumName]**. This approach was adopted because it is common in geodesy to have many coordinates for a single place. This implementation makes it very simple to disambiguate the coordinates -- it's easy to know what coordinate system and datum a set of coordinates is in.

The Cartesian coordinate system implemented in this package has its origin at the Earth's center of mass, has its *z*-axis coincident with the Earth's rotational axis, has its *x* and *y*-axes in the equatorial plane with the *x*-axis on the Prime Meridian. This coordinate system rotates with the Earth and, therefore, is commonly described as Earth-Centered, Earth-Fixed (ECEF).

`xyz`	a data type indicating a point in a geodetic Cartesian coordinate system.
`MakeXYZ[x,y,z,`*`datumName`*`]`	Create a new point in an XYZ coordinate system with coordinates `x,y,z`. *`datumName`* is optional and, if omitted, is supplied with the value `DefaultDatum`.
`MakeXYZ[{x,y,z,`*`datumName`*`}]`	Create a new point in an XYZ coordinate system from a list of coordinates `{x,y,z}`. *`datumName`* is optional and, if omitted,is supplied with the value `DefaultDatum`.
`MakeXYZ[{{x,y,z,`*`datumName`*`}..}]`	Create a list of new points in an XYZ coordinate system from a list of coordinates `{x,y,z}`. *`datumName`* is optional and, if omitted,is supplied with the value `DefaultDatum`.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Create a new point in an XYZ coordinate system.

```
MakeXYZ[1727667.027, -4500726.478, 4162718.427]
```

```
xyz[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, NAD 83 (CORS96)]
```

You can specify a particular datum.

```
MakeXYZ[1727667.027, -4500726.478, 4162718.427, SIRGAS]
```

```
xyz[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, SIRGAS]
```

You can create lists of points.

```
MakeXYZ[{{1727667.027, -4500726.478, 4162718.427},
   {1727667.027, -4500726.478, 4162718.427, NAD27},
   {1727667.027, -4500726.478, 4162718.427, GDA94}}] // TableForm
```

```
xyz[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, NAD 83 (CORS96)]
xyz[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, NAD 27]
xyz[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, GDA94]
```

XYZ Extraction Functions

The following datum of functions extract pieces from the **`xyz`** data structure.

`xyzE[list]`	extracts any item in *list* that is of type **xyz**.
`xyzQ[p]`	returns `True` iff *p* is of type **xyz**.
`xE[p]`	extract the *x*–coordinate of *p* ∈ **xyz**.
`yE[p]`	extract the *y*–coordinate of *p* ∈ **xyz**.
`zE[p]`	extract the *z*–coordinate of *p* ∈ **xyz**.
`datumE[p]`	extract the datum *p* ∈ **xyz** is defined on.
`ellipE[p]`	extract the reference ellipsoid *p* ∈ **xyz** is defined on.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

xyzQ returns True if and only if its argument has type xyz.

```
p = MakeXYZ[1727667.027, -4500726.478, 4162718.427]
xyzQ[p]
xyzQ[MakeBLH[100, -200, 30]]

xyz[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, NAD 83 (CORS96)]

True

False
```

The various extraction functions pull out the pieces of the object.

```
xE[p]
yE[p]
zE[p]
datumE[p]
ellipE[p]

1.72767×10^6

-4.50073×10^6

4.16272×10^6

NAD 83 (CORS96)

GRS80
```

xyzE pull all the XYZ points out of a list:

```
MakeXYZ[{{100, -200, 30}, {100, -200, 30, Group1}, {100, -200, 30, Group2}}]~
  Join~{MakeXYZ[0, 0, 0], MakeBLH[0, 0, 0]} // xyzE

{xyz[100, -200, 30, NAD 83 (CORS96)], xyz[0, 0, 0, NAD 83 (CORS96)]}
```

```
Remove[p]
```

XYZ BracketingBar (|...|)

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that treat Cartesian points as vectors. The BracketingBar operator returns the magnitude of a point treated as a vector.

❘p, opts❘	A function that returns the magnitude of p, meaning the Euclidean two−norm. The operator is typed in the following way: ⋮l⋮ p, *opts* ⋮r⋮.
NormOrder	An option for **BracketingBar**, default value is ∞. If ∞, use the whole vector in the computation. If < ∞, use only the first elements indicated. E.g., *NormOrder*−>2 directs that only the first two elements should be used in the computation.
NormSq	An option for **BracketingBar**, default value is True. If True, take the square root of the result. Otherwise, return the magnitude squared.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating an XYZ point.

```
p = MakeXYZ[1727667.027, -4500726.478, 4162718.427]
```

```
xyz[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, NAD 83 (CORS96)]
```

The BracketingBar operator returns the magnitude of p interpreted as a vector. The magnitude is defined as the Euclidean two-norm, i.e., $\sqrt{\sum_{i=1}^{\text{Length}[p]} p[\![i]\!]^2}$

```
❘p❘
```

```
6.36943×10^6
```

The NormSq option indicate whether to perform the square root. If NormSq → True, don't take the square root:

```
❘p, NormSq → True❘
```

```
4.05696×10^13
```

The options can be used together

```
Table[❘p, NormOrder → o, NormSq → True❘, {o, 1, 3}]
```

```
{2.98483×10^12, 2.32414×10^13, 4.05696×10^13}
```

```
Remove[p]
```

XYZ ToVector, ToUnit

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that convert XYZ points into *Mathematica* vectors and to *Mathematica* unit vectors.

`ToVector[p]`	A function that effectively strips off the object–oriented information from *p* leaving a list of its coordinates; a Mathematica vector.
`ToUnit[p]`	A function that returns an XYZ object which is a scaled version of *p* such that the magnitude of the result is one, i.e., a unit vector.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating an XYZ point.

```
p = MakeXYZ[400.34, -452.422, 234.1]

xyz[400.34, -452.422, 234.1, NAD 83 (CORS96)]
```

Use ToVector to strip out all the object-oriented information leaving just a list of coordinates.

```
ToVector[p]

{400.34, -452.422, 234.1}
```

Use ToUnit to strip out all the object-oriented information leaving just a list of coordinates scaled so that the result is a unit vector.

```
ToUnit[p]

xyz[0.617914, -0.698301, 0.361327, NAD 83 (CORS96)]
```

Verify that ToUnit produces a unit vector:

```
|ToUnit[p]|

1.
```

XYZ Linear algebra operators: scalar multiplication, vector addition and subtraction, inner product, outer produce, matrix multiplication.

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that treat Cartesian points as vectors. The following functions provide the standard linear algebra operators for XYZ points treated as vectors.

$k\ p$ $k*p$	scalar multiplication returns an XYZ object
p_1+p_2 $p_1\oplus p_2$ p_1-p_2 $p_1\ominus p_2$	addition/subtraction of two XYZ vectors returns an XYZ object. The circle version forces the operation even if the operands are in different datums.
$p_1 \cdot p_2$ $p_1\odot p_2$	scalar/inner/dot product returns a scalar. The circle version forces the operation even if the operands are in different datums.
$p_1\times p_2$ $p_1\otimes p_2$	vector/outer/cross product returns an XYZ object. The circle version forces the operation even if the operands are in different datums.
$M.p$	matrix multiplication returns an XYZ object

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating two XYZ points.

```
p1 = MakeXYZ[100 Random[], 100 Random[], 100 Random[]]
p2 = MakeXYZ[100 Random[], 100 Random[], 100 Random[]]

xyz[54.2169, 91.5717, 24.0894, NAD 83 (CORS96)]

xyz[67.8553, 57.9743, 42.5104, NAD 83 (CORS96)]
```

Scalar multiplication can be expressed in all the usual ways

```
30 p1

xyz[1626.51, 2747.15, 722.681, NAD 83 (CORS96)]

30 * p1

xyz[440.162, 13.3102, 164.097, NAD 83 (CORS96)]

p1 * 30

xyz[440.162, 13.3102, 164.097, NAD 83 (CORS96)]

p1 30

xyz[440.162, 13.3102, 164.097, NAD 83 (CORS96)]
```

Vector addition and subtraction is expressed in the usual way. Addition is commutative, subtraction is not

```
p1 + p2
p2 + p1

xyz[45.8248, 68.4142, 54.4157, NAD 83 (CORS96)]

xyz[45.8248, 68.4142, 54.4157, NAD 83 (CORS96)]
```

```
p1 - p2
p2 - p1

xyz[-16.4807, -67.5269, -43.4759, NAD 83 (CORS96)]

xyz[16.4807, 67.5269, 43.4759, NAD 83 (CORS96)]
```

Vector addition and subtraction with *Mathematica* vectors does NOT work:

```
{1, 3, 4} + p1

{1 + xyz[14.6721, 0.443672, 5.46991, NAD 83 (CORS96)],
 3 + xyz[14.6721, 0.443672, 5.46991, NAD 83 (CORS96)],
 4 + xyz[14.6721, 0.443672, 5.46991, NAD 83 (CORS96)]}
```

The inner product is $\sum p_1[\![i]\!]\, p_2[\![i]\!]$. Also, $\mathrm{Cos}[\theta] = \frac{\mathrm{p1.p2}}{|\mathrm{p1}|\,|\mathrm{p2}|}$. See also BracketingBar, the norm operator.

```
p1.p2

754.961
```

$$\mathrm{ArcCos}\left[\frac{\mathrm{p1.p2}}{|\mathrm{p1}|\ |\mathrm{p2}|}\right] \mathbin{//} \mathrm{PPDMS}$$

```
57°21'50.5604"
```

The outer product is vector perpendicular to the other two. The outer product is typed ⋮cross⋮ and is also not commutative

```
p1 × p2
p2 × p1

xyz[-350.077, -547.734, 983.447, NAD 83 (CORS96)]

xyz[350.077, 547.734, -983.447, NAD 83 (CORS96)]
```

This shows that the cross product is perpendicular to its components because the cosine of 90° is zero:

```
Chop[p1 × p2.p1]
Chop[p1 × p2.p2]

0

0
```

An ENU system contains a rotation matrix:

```
M = MakeENUSystem[MakeBLH[41 °, -72 °, 0]][[1]]; M // MatrixForm
```

$$\begin{pmatrix} 0.951057 & 0.309017 & 0. \\ -0.202733 & 0.623949 & 0.75471 \\ 0.233218 & -0.717771 & 0.656059 \end{pmatrix}$$

Matrix multiplication can rotation vectors:

```
M.p1

xyz[14.0911, 1.4305, 6.69192, NAD 83 (CORS96)]
```

Of course, the operators can be mixed in the usual way:

```
p1.p2 ((4.50 p1) × (p2 - p1))
```

xyz[-1.18933×10^{6}, -1.86083×10^{6}, 3.34109×10^{6}, NAD 83 (CORS96)]

The system checks to make sure that operands are in the same datum. A vector is created in the European Terrestrial Reference System of 1989 and added to **p1**, which was created in NAD 83 (CORS96). The operation fails. To force the computation, use the **CirclePlus** operator ⊕, spelled ⎋c+⎋. Note that the datum of the result is **None.** A similar situation applies for **CircleMinus** ⊖, spelled ⎋c-⎋, **CircleDot** ⊙, spelled ⎋c.⎋, and **CircleTimes** ⊗, spelled ⎋c*⎋.

```
v3 = MakeXYZ[100 Random[], 100 Random[], 100 Random[], ETRS89]
p1 + v3
p1 ⊕ v3
p1 ⊖ v3
p1 ⊙ v3
p1 ⊗ v3

xyz[84.359, 94.7098, 5.31459, ETRS89]

xyz[14.6721, 0.443672, 5.46991, NAD 83 (CORS96)] + xyz[84.359, 94.7098, 5.31459, ETRS89]

xyz[99.0311, 95.1534, 10.7845, None]

xyz[-69.6869, -94.2661, 0.155314, None]

1308.81

xyz[-515.696, 383.46, 1352.16, None]

Remove[p1, p2, v3, M]
```

MakeGRD

Geodetic coordinates (blh) are typically projected into cartographic planimetric grids for mapping purposes. This package implements Cartesian grid coordinates with **grd** objects.

Type-checking is a key feature of this package so, in general, it is important that objects in different coordinate systems not be allowed to be intermingled in computations. To this end, position objects are annotated with the information needed to determine which other objects are safe to be used as co-operands. In most respects, **blh** objects, **enu** objects and **grd** objects are identical but their central distinction is the coordinate system in which they are defined. **blh** objects are defined within a datum; that is the only information needed to uniquely identify a **blh** object's coordinate system. **enu** objects exist in a coordinate system whose definition requires a datum and a local origin. **grd** objects exist in a coordinate system whose definition requires a datum, a projection (e.g., Transverse Mercator), standards, central meridian and parallels, false easting and northing for the origin, and possibly other information as well.

`grd`	a data type indicating a point in a cartographic planimetric grid coordinate system.
`MakeGRD[x,y,z,`*`csName`*`]`	Create a new point in an grid coordinate system with coordinates `x,y,z`. *`csName`* is optional and, if omitted, is supplied with the value `DefaultGroup`.
`MakeGRD[{x,y,z,`*`csName`*`}]`	Create a new point in an grid coordinate system from a list of coordinates `{x,y,z}`. *`csName`* is optional and, if omitted,is supplied with the value `DefaultGroup`.
`MakeGRD[{{x,y,z,`*`csName`*`}..}]`	Create a list of new points in an grid coordinate system from a list of coordinates `{x,y,z}`. *`csName`* is optional and, if omitted,is supplied with the value `DefaultGroup`.
`MakeGRD[{{x,y,z}..},`*`csName`*`]`	Create a list of new points in an grid coordinate system from a list of coordinates `{x,y,z}`. *`csName`* is optional and, if omitted,is supplied with the value `DefaultGroup`.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Create a new point in a grid coordinate system.

```
MakeGRD[500, 500]
MakeGRD[500, 500, 100]
```

```
grd[500, 500, 0, DefaultGroup]

grd[500, 500, 100, DefaultGroup]
```

You can specify a group

```
MakeGRD[500, 500, "ThisGroup"]
MakeGRD[500, 500, 100, "ThisGroup"]

grd[500, 500, 0, ThisGroup]

grd[500, 500, 100, ThisGroup]
```

This format arises when pulling specific points of interest out from a list

```
MakeGRD[{500, 500}]
MakeGRD[{500, 500, 200}]

grd[500, 500, 0, DefaultGroup]

grd[500, 500, 200, DefaultGroup]
```

All the forms all you to specify a group

```
MakeGRD[{500, 500}, "ThatGroup"]
MakeGRD[{500, 500, 20}, "ThatGroup"]

MakeGRD[{500, 500}, ThatGroup]

grd[500, 500, 20, ThatGroup]
```

MakeGRD is listable

```
MakeGRD[Table[{Random[], Random[]}, {3}]]

{grd[0.914944, 0.515165, 0, DefaultGroup],
 grd[0.912025, 0.349908, 0, DefaultGroup], grd[0.585328, 0.276907, 0, DefaultGroup]}

MakeGRD[Table[{Random[], Random[], Random[]}, {3}]]

{grd[0.623174, 0.909838, 0.690993, DefaultGroup],
 grd[0.622304, 0.804053, 0.369495, DefaultGroup],
 grd[0.917022, 0.260327, 0.906515, DefaultGroup]}
```

MakeGRD is listable; one group for all:

```
MakeGRD[Table[{Random[], Random[]}, {2}], "one group"]

{{MakeGRD[0.459191, one group], MakeGRD[0.956532, one group]},
 {MakeGRD[0.591154, one group], MakeGRD[0.412219, one group]}}

MakeGRD[Table[{Random[], Random[], Random[]}, {2}], "one group"]

{grd[0.65852, 0.656775, 0.331865, one group],
 grd[0.948097, 0.543035, 0.918413, one group]}
```

MakeGRD is listable; a different group for each:

```
MakeGRD[Table[{Random[], Random[], FromCharacterCode[Random[Integer, {110, 120}]]}, {2}]]

{grd[0.115425, 0.036735, 0, p], grd[0.887036, 0.247634, 0, q]}
```

```
MakeGRD[Table[
  {Random[], Random[], Random[], FromCharacterCode[Random[Integer, {110, 120}]]}, {2}]]
```

```
{grd[0.83383, 0.501189, 0.177239, x], grd[0.494353, 0.882012, 0.101262, v]}
```

GRD Extraction Functions

The following datum of functions extract pieces from the **grd** data structure.

`grdE[`*list*`]`	extracts any item in *list* that is of type **grd**.
`grdQ[`*p*`]`	returns `True` iff *p* is of type **grd**.
`xE[`*p*`]`	extract the *x*–coordinate of *p* e `grd`.
`yE[`*p*`]`	extract the *y*–coordinate of *p* e **grd**.
`zE[`*p*`]`	extract the *z*–coordinate of *p* e **grd**.
`datumE[`*p*`]`	extract the datum *p* e **grd** is defined on.
`ellipE[`*p*`]`	extract the reference ellipsoid *p* e **grd** is defined on.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

grdQ returns True if and only if its argument has type **grd**.

```
p = MakeGRD[1727667.027, -4500726.478, 4162718.427]
grdQ[p]
grdQ[MakeBLH[100, -200, 30]]
```

```
grd[1.72767×10⁶, -4.50073×10⁶, 4.16272×10⁶, DefaultGroup]

True

False
```

The various extraction functions pull out the pieces of the object.

```
xE[p]
yE[p]
zE[p]
datumE[p]
ellipE[p]
```

```
1.72767×10^6

-4.50073×10^6

4.16272×10^6

datumE[grd[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, DefaultGroup]]

ellipE[grd[1.72767×10^6, -4.50073×10^6, 4.16272×10^6, DefaultGroup]]
```

grdE pull all the **grd** points out of a list:

```
MakeGRD[{{100, -200, 30}, {100, -200, 30, "Group1"}, {100, -200, 30, "Group2"}}] ~
  Join ~ {MakeGRD[0, 0, 0], MakeBLH[0, 0, 0]} // grdE
```

```
{grd[100, -200, 30, DefaultGroup], grd[100, -200, 30, Group1],
 grd[100, -200, 30, Group2], grd[0, 0, 0, DefaultGroup]}
```

```
Remove[p]
```

GRD BracketingBar (|...|)

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that treat Cartesian points as vectors. The BracketingBar operator returns the magnitude of a point treated as a vector.

|*p*, *opts*|	A function that returns the magnitude of *p*, meaning the Euclidean two−norm. The operator is typed in the following way: ⋮l|⋮ *p*, *opts* ⋮r|⋮.
NormOrder	An option for **BracketingBar**, default value is ∞. If ∞, use the whole vector in the computation. If < ∞, use only the first elements indicated. E.g., *NormOrder*−>2 directs that only the first two elements should be used in the computation.
NormSq	An option for **BracketingBar**, default value is True. If True, take the square root of the result. Otherwise, return the magnitude squared.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating an **grd** point.

```
p = MakeGRD[1727667.027, -4500726.478, 4162718.427]
```

```
grd[1.72767 × 10^6, -4.50073 × 10^6, 4.16272 × 10^6, DefaultGroup]
```

The BracketingBar operator returns the magnitude of p interpreted as a vector. The magnitude is defined as the Euclidean two-norm, i.e., $\sqrt{\sum_{i=1}^{\text{Length}[p]} p[[i]]^2}$

```
|p|
```

$\sqrt{\texttt{xE[p]}^2 + \texttt{yE[p]}^2 + \texttt{zE[p]}^2}$

```
6.36943 × 10^6
```

```
6.36943 × 10^6
```

The NormSq option indicate whether to perform the square root. If NormSq → True, don't take the square root:

```
|p, NormSq → True|
```

```
4.05696 × 10^13
```

The options can be used together

```
Table[|p, NormOrder → o, NormSq → True|, {o, 1, 3}]
```

```
{2.98483 × 10^12, 2.32414 × 10^13, 4.05696 × 10^13}
```

```
Remove[p]
```

GRD ToVector, ToUnit

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that convert **grd** points into *Mathematica* vectors and to *Mathematica* unit vectors.

ToVector[*p*]	A function that effectively strips off the object–oriented information from *p* leaving a list of its coordinates; a Mathematica vector.
ToUnit[*p*]	A function that returns a **grd** object which is a scaled version of *p* such that the magnitude of the result is one, i.e., a unit vector.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating an GRD point.

```
p = MakeGRD[400.34, -452.422, 234.1]
```

```
grd[400.34, -452.422, 234.1, DefaultGroup]
```

Use ToVector to strip out all the object-oriented information leaving just a list of coordinates.

```
ToVector[p]
```

```
{400.34, -452.422, 234.1}
```

Use ToUnit to strip out all the object-oriented information leaving just a list of coordinates scaled so that the result is a unit vector.

```
ToUnit[p]
```

```
grd[0.617914, -0.698301, 0.361327, DefaultGroup]
```

Verify that ToUnit produces a unit vector:

```
|ToUnit[p]|
```

```
1.
```

GRD Linear algebra operators: scalar multiplication, vector addition and subtraction, inner product, outer produce, matrix multiplication.

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that treat Cartesian points as vectors. The following functions provide the standard linear algebra operators for GRD points treated as vectors.

$k\ p$ $k \star p$	scalar multiplication returns an GRD object
$p_1 + p_2$ $p_1 \oplus p_2$ $p_1 - p_2$ $p_1 \ominus p_2$	addition/subtraction of two GRD vectors returns an GRD object. The circle version forces the operation even if the operands are in different datums.
$p_1 \cdot p_2$ $p_1 \odot p_2$	scalar/inner/dot product returns a scalar. The circle version forces the operation even if the operands are in different datums.
$p_1 \times p_2$ $p_1 \otimes p_2$	vector/outer/cross product returns an GRD object. The circle version forces the operation even if the operands are in different datums.
$M.p$	matrix multiplication returns an GRD object

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating two GRD points.

```
p1 = MakeGRD[100 Random[], 100 Random[], 100 Random[]]
p2 = MakeGRD[100 Random[], 100 Random[], 100 Random[]]
```

```
grd[52.0627, 50.7649, 8.28201, DefaultGroup]

grd[67.0475, 77.5822, 27.2129, DefaultGroup]
```

Scalar multiplication can be expressed in all the usual ways

```
30 p1
```

```
grd[1561.88, 1522.95, 248.46, DefaultGroup]
```

```
30 * p1
```

```
grd[1561.88, 1522.95, 248.46, DefaultGroup]
```

```
p1 * 30
```

```
grd[1561.88, 1522.95, 248.46, DefaultGroup]
```

```
p1 30
```

```
grd[1561.88, 1522.95, 248.46, DefaultGroup]
```

Vector addition and subtraction is expressed in the usual way. Addition is commutative, subtraction is not

```
p1 + p2
p2 + p1
```

```
grd[119.11, 128.347, 35.495, DefaultGroup]

grd[119.11, 128.347, 35.495, DefaultGroup]
```

```
p1 - p2
p2 - p1
```

```
grd[-14.9848, -26.8173, -18.9309, DefaultGroup]

grd[14.9848, 26.8173, 18.9309, DefaultGroup]
```

Vector addition and subtraction with *Mathematica* vectors does NOT work:

```
{1, 3, 4} + p1
```

```
{1 + grd[52.0627, 50.7649, 8.28201, DefaultGroup],
 3 + grd[52.0627, 50.7649, 8.28201, DefaultGroup],
 4 + grd[52.0627, 50.7649, 8.28201, DefaultGroup]}
```

The inner product is $\sum p_1 [\![i]\!]\, p_2 [\![i]\!]$. Also, $\mathrm{Cos}[\theta] = \frac{\mathrm{p1.p2}}{|\mathrm{p1}|\,|\mathrm{p2}|}$. See also BracketingBar, the norm operator.

```
p1.p2
```

```
7654.5
```

```
ArcCos[p1.p2/(|p1| |p2|)] // PPDMS
```

```
9°38'40.1258"
```

The outer product is vector perpendicular to the other two. The outer product is typed ⋮cross⋮ and is also not commutative

```
p1 × p2
p2 × p1
```

```
grd[738.925, -861.492, 635.481, DefaultGroup]

grd[-738.925, 861.492, -635.481, DefaultGroup]
```

This shows that the cross product is perpendicular to its components because the cosine of 90° is zero:

```
Chop[p1 × p2.p1]
Chop[p1 × p2.p2]
```

```
0

0
```

An ENU system contains a rotation matrix:

```
M = MakeENUSystem[MakeBLH[41 °, -72 °, 0]][[1]]; M // MatrixForm
```

```
( 0.951057   0.309017   0.       )
( -0.202733  0.623949   0.75471  )
( 0.233218   -0.717771  0.656059 )
```

Matrix multiplication can rotation vectors:

```
M.p1
```

```
grd[65.2018, 27.3704, -18.8621, DefaultGroup]
```

Of course, the operators can be mixed in the usual way:

```
p1.p2 ((4.50 p1) × (p2 - p1))
```

```
grd[2.54525 × 10^7, -2.96743 × 10^7, 2.18893 × 10^7, DefaultGroup]
```

The system checks to make sure that operands are in the same datum. A vector is created in the European Terrestrial Reference System of 1989 and added to **p1**, which was created in NAD 83 (CORS96). The operation fails. To force the computation, use the **CirclePlus** operator ⊕, spelled ESCc+ESC. Note that the datum of the result is **None.** A similar situation applies for **CircleMinus** ⊖, spelled ESCc-ESC, **CircleDot** ⊙, spelled ESCc.ESC, and **CircleTimes** ⊗, spelled ESCc*ESC.

```
v3 = MakeGRD[100 Random[], 100 Random[], 100 Random[], "thisGroup"]
p1 + v3
p1 ⊕ v3
p1 ⊖ v3
p1 ⊙ v3
p1 ⊗ v3

grd[92.7097, 14.1474, 54.6176, thisGroup]

GeometricalGeodesy::datumMismatch :
 Cannot operate on objects in different datums: DefaultGroup vs
   thisGroup. Use the CirclePlus/CircleMinus operator instead.

grd[144.772, 64.9122, 62.8996, None]

grd[-40.647, 36.6175, -46.3356, None]

5997.25

grd[2655.49, -2075.72, -3969.85, None]

Remove[p1, p2, v3, M]
```

MakeENU

This package implements an object-oriented representation of points in various coordinate systems. For example, geodetic coordinates are of type **`blh`** and have the form **`blh[ϕ,λ,h,datumName]`**. Similarly, geocentric Cartesian coordinates are of type **`xyz`** and have the form **`xyz[x,y,z,datumName]`**. This approach was adopted because it is common in geodesy to have many coordinates for a single place. This implementation makes it very simple to disambiguate the coordinates -- it's easy to know what coordinate system and datum a set of coordinates is in.

The local Cartesian coordinate system with east/north/up coordinates is a three-dimensional Cartesian coordinate system whose origin is (typically) somewhere other than the Earth's center of mass. This coordinate system is created by translating the XYZ system to a new location, the origin of the local ENU system. Then, the axes are rotated so that y is aligned with the meridian (north axis), x is aligned with the parallel (east axis) and z completes a right-handed system pointing to the zenith (up axis).

`enu`	a data type indicating a point in a local geodetic Cartesian coordinate system.
`MakeENU[e,n,u,`*`groupName`*`]`	Create a new point in an ENU coordinate system with coordinates `e,n,u`. *`groupName`* is optional and, if omitted,is supplied with the value `DefaultGroup`. *`groupName`* is used to associate points in the same local system and can have any value.
`MakeENU[{e,n,u,`*`groupName`*`}]`	Create a new point in an ENU coordinate system from a list of coordinates `{e,n,u}`. *`groupName`* is optional and, if omitted, is supplied with the value `DefaultGroup`. *`groupName`* is used to associate points in the same local system.
`MakeENU[{{e,n,u,`*`groupName`*`}..}]`	Create a list of new points in an ENU coordinate system from a list of coordinates `{e,n,u}`. *`groupName`* is optional and, if omitted,is supplied with the value `DefaultGroup`. *`groupName`* is used to associate points in the same local system.

The grouping mechanism is used to associate ENU points with the transformation system that transforms them to and from the Cartesian reference frame. It is highly likely that there will be more than one group. For example, in photogrammetry, all points in a single stereo-pair are in a common ENU reference frame but each stereo-pair gives rise to a different frame. Therefore, photogrammetry work will have to cope with many points in many different local ENU systems. This package makes it easy to keep the points straight simply by grouping common points together via the name of their system. See conversions between systems below.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Create a new point in an ENU coordinate system.

```
MakeENU[100, -200, 30]
```

```
enu[100, -200, 30, DefaultGroup]
```

You can specify a particular group name.

```
MakeENU[100, -200, 30, Group1]
```

```
enu[100, -200, 30, Group1]
```

You can create lists of points.

```
MakeENU[{{100, -200, 30}, {100, -200, 30, Group1}, {100, -200, 30, Group2}}] // TableForm
```

```
enu[100, -200, 30, DefaultGroup]
enu[100, -200, 30, Group1]
enu[100, -200, 30, Group2]
```

ENU Extraction Functions

The following group of functions extract pieces from the enu data structure.

`enuE[`*list*`]`	extracts any item in *list* that is of type `enu`.
`enuQ[`*p*`]`	returns `True` iff *p* is of type `enu`.
`eE[`*p*`]`	extract the easting of *p* ∈ `enu`.
`nE[`*p*`]`	extract the northing of *p* ∈ *enu*.
`uE[`*p*`]`	extract the ellipsoid height of *p* ∈ *enu*.
`groupE[`*p*`]`	extract the group *p* ∈ *enu* is defined on.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

enuQ returns True if and only if its argument has type enu.

```
p = MakeENU[100, -200, 30]

enu[100, -200, 30, DefaultGroup]

enuQ[p]
enuQ[MakeXYZ[100, -200, 30]]

True

False
```

The various extraction functions pull out the pieces of the object.

```
eE[p]
nE[p]
uE[p]
groupE[p]

100

-200

30

DefaultGroup
```

enuE pull all the ENU points out of a list:

```
MakeENU[{{100, -200, 30}, {100, -200, 30, Group1}, {100, -200, 30, Group2}}] ~
  Join ~ {MakeXYZ[0, 0, 0], MakeBLH[0, 0, 0]} // enuE
```

```
{enu[100, -200, 30, DefaultGroup], enu[100, -200, 30, Group1], enu[100, -200, 30, Group2]}
```

ENU BracketingBar (|...|)

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that treat Cartesian points as vectors. The BracketingBar operator returns the magnitude of a point treated as a vector.

|*p, opts*|	A function that returns the magnitude of *p*, meaning the Euclidean two−norm. The operator is typed in the following way: ⋮l⋮ *p*, *opts* ⋮r⋮.
NormOrder	An option for **BracketingBar**, default value is ∞. If ∞, use the whole vector in the computation. If < ∞, use only the first elements indicated. E.g., *NormOrder*−>2 directs that only the first two elements should be used in the computation.
NormSq	An option for **BracketingBar**, default value is True. If True, take the square root of the result. Otherwise, return the magnitude squared.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating an ENU point.

```
p = MakeENU[400.34, -452.422, 234.1]
```

```
enu[400.34, -452.422, 234.1, DefaultGroup]
```

The BracketingBar operator returns the magnitude of p interpreted as a vector. The magnitude is defined as the Euclidean two-norm, i.e., $\sqrt{\sum_{i=1}^{\text{Length}[p]} p[\![i]\!]^2}$

```
|p|
```

```
647.889
```

The NormSq option indicate whether to perform the square root. If NormSq → True, don't take the square root:

```
|p, NormSq → True|
```

```
419761.
```

The options can be used together

```
Table[‖p, NormOrder → o, NormSq → True‖, {o, 1, 3}]

{160272., 364958., 419761.}

Remove[p]
```

ENU ToVector, ToUnit

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that convert ENU points into *Mathematica* vectors and to *Mathematica* unit vectors.

`ToVector[p]`	A function that effectively strips off the object–oriented information from *p* leaving a list of its coordinates; a Mathematica vector.
`ToUnit[p]`	A function that returns an ENU object which is a scaled version of *p* such that the magnitude of the result is one, i.e., a unit vector.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating an ENU point.

```
p = MakeENU[400.34, -452.422, 234.1]

enu[400.34, -452.422, 234.1, DefaultGroup]
```

Use ToVector to strip out all the object-oriented information leaving just a list of coordinates.

```
ToVector[p]

{400.34, -452.422, 234.1}
```

Use ToUnit to strip out all the object-oriented information leaving just a list of coordinates scaled so that the result is a unit vector.

```
ToUnit[p]

enu[0.617914, -0.698301, 0.361327, DefaultGroup]
```

Verify that ToUnit produces a unit vector:

```
‖ToUnit[p]‖

1.
```

ENU Linear algebra operators: scalar multiplication, vector addition and subtraction, inner product, outer produce, matrix multiplication.

There is a logical duality between points and vectors in Cartesian coordinate systems (although only in Cartesian coordinate systems). Therefore, this package provides functions that treat Cartesian points as vectors. The following functions provide the standard linear algebra operators for ENU points treated as vectors.

k p *k*p*	scalar multiplication returns an ENU object
p_1+p_2 p_1-p_2	addition/subtraction of two ENU vectors returns an ENU object
$p_1 . p_2$	scalar/inner/dot product returns a scalar
$p_1$âp$_2$	vector/outer/cross product returns an ENU object
M.p	matrix multiplication returns an ENU object

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating two ENU points in the same group.

```
p1 = MakeENU[100 Random[], 100 Random[], 100 Random[]]
p2 = MakeENU[100 Random[], 100 Random[], 100 Random[]]
```

```
enu[39.8629, 39.0277, 65.6337, DefaultGroup]

enu[23.9619, 58.4232, 65.1771, DefaultGroup]
```

Scalar multiplication can be expressed in all the usual ways

```
30 p1
```

```
enu[1195.89, 1170.83, 1969.01, DefaultGroup]
```

```
30 * p1
```

```
enu[1195.89, 1170.83, 1969.01, DefaultGroup]
```

```
p1 * 30
```

```
enu[1195.89, 1170.83, 1969.01, DefaultGroup]
```

```
p1 30
```

```
enu[1195.89, 1170.83, 1969.01, DefaultGroup]
```

Vector addition and subtraction is expressed in the usual way. Addition is commutative, subtraction is not

```
p1 + p2
p2 + p1
```

```
enu[63.8249, 97.4508, 130.811, DefaultGroup]

enu[63.8249, 97.4508, 130.811, DefaultGroup]
```

```
p1 - p2
p2 - p1
```

```
enu[15.901, -19.3955, 0.456538, DefaultGroup]

enu[-15.901, 19.3955, -0.456538, DefaultGroup]
```

Vector addition and subtraction with *Mathematica* vectors does NOT work:

```
{1, 3, 4} + p1
```

```
{1 + enu[39.8629, 39.0277, 65.6337, DefaultGroup],
 3 + enu[39.8629, 39.0277, 65.6337, DefaultGroup],
 4 + enu[39.8629, 39.0277, 65.6337, DefaultGroup]}
```

The inner product is $\sum p_1[\![i]\!]\, p_2[\![i]\!]$. Also, $\text{Cos}[\theta] = \frac{\text{p1.p2}}{|\text{p1}|\,|\text{p2}|}$. See also BracketingBar, the norm operator.

```
p1.p2
```

```
7513.13
```

$$\text{ArcCos}\left[\frac{\text{p1.p2}}{|\text{p1}|\ |\text{p2}|}\right] \text{ // PPDMS}$$

```
16°1'51.7325"
```

The outer product is vector perpendicular to the other two. The outer product is typed ⋮cross⋮ and is also not commutative

```
p1 × p2
p2 × p1
```

```
enu[-1290.82, -1025.44, 1393.74, DefaultGroup]

enu[1290.82, 1025.44, -1393.74, DefaultGroup]
```

This shows that the cross product is perpendicular to its components because the cosine of 90° is zero:

```
Chop[p1 × p2.p1]
Chop[p1 × p2.p2]
```

```
0

0
```

An ENU system contains a rotation matrix:

```
M = MakeENUSystem[MakeBLH[41 °, -72 °, 0]][[1]]; M // MatrixForm
```

$$\begin{pmatrix} 0.951057 & 0.309017 & 0. \\ -0.202733 & 0.623949 & 0.75471 \\ 0.233218 & -0.717771 & 0.656059 \end{pmatrix}$$

Matrix multiplication can rotation vectors:

```
M.p1

enu[49.9721, 65.8041, 24.3434, DefaultGroup]
```

Of course, the operators can be mixed in the usual way:

```
p1.p2 ((4.50 p1) × (p2 - p1))

enu[-4.36413 × 10^7, -3.46692 × 10^7, 4.7121 × 10^7, DefaultGroup]
```

The system checks to make sure that operands are in the same group. A vector is created in a different and added to **p1**, which was created in **DefaultGroup**. The operation fails. To force the computation, use the **CirclePlus** operator ⊕, spelled ESCc+ESC. Note that the group of the result is **None.** A similar situation applies for **CircleMinus** ⊖**,**spelled ESCc-ESC**, CircleDot ⊙,**spelled ESCc.ESC**,**and **CircleTimes ⊗**, spelled ESCc*ESC**.**

```
v3 = MakeENU[100 Random[], 100 Random[], 100 Random[], "OtherGroup"]
p1 + v3
p1 ⊕ v3
p1 ⊖ v3
p1 ⊙ v3
p1 ⊗ v3

enu[17.9123, 36.4928, 0.262342, OtherGroup]

enu[17.9123, 36.4928, 0.262342, OtherGroup] + enu[39.8629, 39.0277, 65.6337, DefaultGroup]

enu[57.7752, 75.5204, 65.896, None]

xyz[21.9506, 2.53488, 65.3713, None]

2155.48

enu[-2384.92, 1165.19, 755.634, None]

Remove[p1, p2, v3, M]
```

MakeENUSystem

The local Cartesian coordinate system with east/north/up coordinates is a three-dimensional Cartesian coordinate system whose origin is (typically) somewhere other than the Earth's center of mass. This coordinate system is created by translating the XYZ system to a new location, the origin of the local ENU system. Then, the axes are rotated so that *y* is aligned with the meridian (north axis), *x* is aligned with the parallel (east axis) and *z* completes a right-handed system pointing to the zenith (up axis).

`enuSys`	a data type indicating an ENU coordinate system.
`MakeENUSystem[`*p*`]`	Create the forward and inverse transformation matrices to map points to and from the ENU system. *p* is a geodetic point (`blh`) defining the origin of the local system. It is not necessary to stipulate an ellipsoid because *p* already contains one.
`MakeENUSystem[`*f*`,`*l*`,` *h*`,`*datumName*`,`*groupName*`]`	Same as `MakeENUSystem[`*p*`]` but all the parameters are provided explicitly. The group name is optional and, if omitted, will adopt the default group name.

Unlike XYZ and BLH, ENU requires the computation of a transformation matrix that maps coordinates from XYZ to the local system. Therefore, the first step in using ENU is to create this matrix. It happens that the transpose of this matrix is its inverse and that inverse matrix is the transformation from the local system back to XYZ. Therefore, the system constructor creates both matrices and stores them for future use.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating a BLH point to be the new local origin.

```
lx3030BLH = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303]
```

```
blh[0.729765, -1.26101, 157.303, NAD 83 (CORS96)]
```

Use that point to create the local system. The system has three pieces. 1) The rotation matrix to rotate the coordinate axes to the local system 2) an XYZ vector that establishes the coordinates of the origin of the local system 3) the inverse rotation matrix.

```
sys = MakeENUSystem[lx3030BLH]
```

```
enuSys[{{0.952399, 0.304855, 0},
   {-0.203245, 0.634959, 0.745331}, {0.227218, -0.709853, 0.666694}},
  xyz[1.45142×10^6, -4.5344×10^6, 4.2302×10^6, NAD 83 (CORS96)],
  {{0.952399, -0.203245, 0.227218},
   {0.304855, 0.634959, -0.709853}, {0, 0.745331, 0.666694}}, DefaultGroup]
```

The system can be given a name by supplying a second argument.

```
MakeENUSystem[lx3030BLH, Group1]
```

```
enuSys[{{0.952399, 0.304855, 0},
   {-0.203245, 0.634959, 0.745331}, {0.227218, -0.709853, 0.666694}},
  xyz[1.45142×10^6, -4.5344×10^6, 4.2302×10^6, NAD 83 (CORS96)],
  {{0.952399, -0.203245, 0.227218},
   {0.304855, 0.634959, -0.709853}, {0, 0.745331, 0.666694}}, Group1]
```

P[ ] works with ENU systems.

```
sys // P
```

$$\begin{pmatrix} 0.952399 & 0.304855 & 0 \\ -0.203245 & 0.634959 & 0.745331 \\ 0.227218 & -0.709853 & 0.666694 \end{pmatrix}$$

```
origin: {1451423.342, -4534399.864, 4230204.318, NAD 83 (CORS96), GRS80}

Group: DefaultGroup
```

We can define the parameters without a blh point.

```
MakeENUSystem[41 °, -72 °, 0, ITRF94, Group5]
```

```
enuSys[{{0.951057, 0.309017, 0.},
  {-0.202733, 0.623949, 0.75471}, {0.233218, -0.717771, 0.656059}},
 xyz[1.48964×10^6, -4.58465×10^6, 4.16242×10^6, ITRF94], {{0.951057, -0.202733, 0.233218},
  {0.309017, 0.623949, -0.717771}, {0., 0.75471, 0.656059}}, Group5]
```

Coordinate Systems Conversions

This package comes with conversions between all the coordinate systems. In some cases, these are exact (e.g., BLH→XYZ). In other cases, it is approximate (e.g., XYZ→BLH). Conversions can be done with two equivalent operators: χ or ⇒. The former, the Greek letter chi (ESCcESC) is the name of a function that does all the conversions. The latter (ESC␣=>ESC, note the space after the first ESC!) is an infix operator that does exactly the same thing as the χ function. However, either form can be more convenient syntactically in different situations.

BLH→XYZ

`χ[p,xyz]`	ESCcESC; Also note that `xyz` is `the` three literal letters –– see examples below. Returns the ECEF (xyz) point equivalent to the BLH point *p*. Note that it is not necessary to explicitly inform the system that *p* is a BLH point.
`p⇒xyz`	ESC␣=>ESC; Also note that `xyz` is `the` three literal letters –– see examples below. Returns the ECEF (xyz) point equivalent to the BLH point *p*.

The conversion from BLH to XYZ is accomplished by Helmert's projection and is exact.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating a BLH point.

```
1x3030BLH = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303]
```

```
blh[0.729765, -1.26101, 157.303, NAD 83 (CORS96)]
```

Convert it to ECEF.

```
χ[1x3030BLH, xyz] // P
```

```
{1451423.342, -4534399.864, 4230204.318, NAD 83 (CORS96), GRS80}
```

```
1x3030BLH ⇒ xyz // P
```

```
{1451423.342, -4534399.864, 4230204.318, NAD 83 (CORS96), GRS80}
```

```
Remove[1x3030BLH]
```

BLH→ENU

χ[*p*,enu,*enuSystem*]	ESC c ESC; Also note that enu is the three literal letters –– see examples below. Returns the ENU point equivalent to the BLH point *p*. Note that it is not necessary to explicitly inform the system that *p* is a BLH point. The third argument is the definition (basis) of the ENU coordinate system (see MakeENUSystem)
p⇒{enu,*enuSystem*}	ESC ␣=> ESC; Also note that enu is the three literal letters –– see examples below. Returns the ENU point equivalent to the BLH point *p*.

The conversion from BLH to ENU is accomplished by first converting BLH to XYZ, then XYZ to ENU. The former is done with Helmert's projection; it is exact. The latter is a vector translation and rotation; it is exact.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating a BLH point to serve as the origin of the ENU local coordinate system.

```
origin = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303];
sys = MakeENUSystem[origin];
```

Create another point to place into the ENU system.

```
p = MakeBLH[DMS[41, 48, 50, N], DMS[72, 15, 0, W], 180.];
```

Convert *p* to enu using both operations.

```
χ[p, enu, sys] // P
```

```
{47.14019573, 160.9196493, 22.69479156, DefaultGroup}
```

```
p ⇒ {enu, sys} // P
```

```
{47.14019573, 160.9196493, 22.69479156, DefaultGroup}
```

```
Remove[origin, sys, p]
```

XYZ→BLH

χ[*p*,blh]	ESCcESC; Also note that blh is the three literal letters –– see examples below. Returns the BLH point equivalent to the ECEF (xyz) point *p*. Note that it is not necessary to explicitly inform the system that *p* is an XYZ point.
p⇒blh	ESC␣=>ESC; Also note that blh is the three literal letters –– see examples below. Returns the ECEF (xyz) point equivalent to the BLH point *p*.

The conversion from XYZ to BLH is accomplished by Bowring's 1985 method and is approximate. However, the '85 method is an improvement to his original method by increasing the accuracy for positions from Earth, such as for GPS satellites. The method is iterative but provides at least millimeter accuracy without iteration.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating an XYZ point.

```
{1451423.342, -4534399.864, 4230204.318, NAD 83 (CORS96), GRS80}
```

```
p = MakeXYZ[1451423.342, -4534399.864, 4230204.318]
```

```
xyz[1.45142×10^6, -4.5344×10^6, 4.2302×10^6, NAD 83 (CORS96)]
```

Convert it to BLH.

```
χ[p, blh] // PPDMS
```

```
41°48'44.7844"N, 72°15'2.04234"W, 157.303, NAD 83 (CORS96)
```

```
p ⇒ blh
```

```
blh[0.729765, -1.26101, 157.303, NAD 83 (CORS96)]
```

The ⇒ operator can be used in serial:

```
p ⇒ blh ⇒ xyz // P
```

```
{1451423.342, -4534399.864, 4230204.318, NAD 83 (CORS96), GRS80}
```

```
Remove[p]
```

XYZ→ENU

`χ[p,enu,enuSystem]`	ESC c ESC; Also note that `enu` is `the` three literal letters -- see examples below. Returns the `ENU` point equivalent to the XYZ point *p*. Note that it is not necessary to explicitly inform the system that *p* is an XYZ point. The third argument is the definition (basis) of the ENU coordinate system (see MakeENUSystem)
`p⇒{enu,enuSystem}`	ESC ␣=> ESC; Also note that `enu` is `the` three literal letters -- see examples below. Returns the `ENU` point equivalent to the XYZ point *p*.

The conversion from XYZ to ENU is accomplished by a vector translation and rotation; it is exact.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating a BLH point to serve as the origin of the ENU local coordinate system.

```
origin = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303];
sys = MakeENUSystem[origin];
```

Create another point to place into the ENU system.

```
p = MakeXYZ[1451440.689, -4534299.425, 4230339.387];
```

Convert *p* to enu using both operations.

```
χ[p, enu, sys] // P
```

```
{47.14004493, 160.919681, 22.69440738, DefaultGroup}
```

```
p ⇒ {enu, sys} // P
```

```
{47.14004493, 160.919681, 22.69440738, DefaultGroup}
```

```
Remove[origin, sys, p]
```

ENU→BLH

χ[*p*,blh,*enuSystem*]	⎋c⎋; Also note that blh is the three literal letters -- see examples below. Returns the BLH point equivalent to the ENU point *p*. Note that it is not necessary to explicitly inform the system that *p* is an ENU point. The third argument is the definition (basis) of the ENU coordinate system (see MakeENUSystem)
p⇒{blh,*enuSystem*}	⎋␣=>⎋; Also note that blh is the three literal letters -- see examples below. Returns the BLH point equivalent to the ENU point *p*.

The conversion from BLH to BLH is accomplished first by converting to XYZ then to BLH. The former is done with a vector translation and rotation; it is exact. The latter is done with Bowring's 1985 method and it is iterative.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating a BLH point to serve as the origin of the ENU local coordinate system.

```
origin = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303];
sys = MakeENUSystem[origin];
```

Create another ENU point to convert to BLH.

```
p = MakeENU[100.0, -150.0, 30.0];
```

Convert *p* to blh using both operations.

```
χ[p, blh, sys] // PPDMS
```

```
41°48'39.9227"N, 72°14'57.7101"W, 187.306, NAD 83 (CORS96)
```

```
p ⇒ {blh, sys} // PPDMS
```

```
41°48'39.9227"N, 72°14'57.7101"W, 187.306, NAD 83 (CORS96)
```

```
(p ⇒ {blh, sys}) ⇒ xyz
```

xyz[1.45156×10^{6}, -4.53449×10^{6}, 4.23011×10^{6}, NAD 83 (CORS96)]

```
Remove[origin, sys, p]
```

ENU→XYZ

`χ[p,xyz,enuSystem]`	ESC c ESC; Also note that `xyz` is the three literal letters -- see examples below. Returns the `xyz` point equivalent to the ENU point *p*. Note that it is not necessary to explicitly inform the system that *p* is an ENU point. The third argument is the definition (basis) of the ENU coordinate system (see MakeENUSystem)
`p⇒{xyz,enuSystem}`	ESC ␣=> ESC; Also note that `xyz` is the three literal letters -- see examples below. Returns the `xyz` point equivalent to the ENU point *p*.

The conversion from ENU to XYZ is accomplished by a vector translation and rotation; it is exact.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Start by creating a BLH point to serve as the origin of the ENU local coordinate system.

```
origin = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303];
sys = MakeENUSystem[origin];
```

Create another ENU point to convert to XYZ.

```
p = MakeENU[100.0, -150.0, 30.0];
```

Convert *p* to xyz using both operations.

```
χ[p, xyz, sys] // P

{1451555.886, -4534485.917, 4230112.519, NAD 83 (CORS96), GRS80}

p ⇒ {xyz, sys} // P

{1451555.886, -4534485.917, 4230112.519, NAD 83 (CORS96), GRS80}

Remove[origin, sys, p]
```

The "Direct" Problem

This package implements the major computations performed in geometric geodesy. For the most part, these computations pertain to geodetic traverses, the *direct* and *inverse* problems performed on the surface of a geodetic ellipsoid. `Direct` computes the position and back azimuth of a point defined by a known point, an azimuth and the reduced distance to the new point. This is the fundamental computation needed for a geodetic traverse.

`Direct[`*`station,a,d,opts`*`]`	Compute the BLH coordinates defined by a starting *station* (BLH), a geodetic azimuth α, and a reduced distance d. The return value is a list {*newPoint*, *backAzimuth*}
`DirectMethod`	An option for `Direct`. Currently,only one value is allowed, namely `Vincenty75`. This option is included in case other methods are implemented in the future.
`EllipsoidHeight`	An option for `Direct`. Allows the user to provide an ellipsoid height for the result. Usage: `EllipsoidHeight→`*`v`*, where *v* is the ellipsoid height. Default value is zero.

There are many algorithms given in the literature for `Direct`. We chose to implement Vincenty's algorithm (1975) because it works well under all conditions. However, like all other geodetic computations, there are other ways to do it. Therefore, even though this implementation provides only one way, the code was written to easily permit the implementation of other methods, perhaps for comparison.

station is a geodetic (blh) point from which to begin. *station* cannot be either the North or South pole.

α is a geodetic azimuth (clockwise positive from North) defining the direction to produce from *station*.

d is the distance in meters (see *Units* for conversions) that the new point is to be from *station*.

`EllipsoidHeight` is an option that allows the user to provide an ellipsoid height for the result. This option has a default value of zero, meaning that, unless otherwise specified, the resulting point will have an ellipsoid height of zero. Otherwise, usage is: **`EllipsoidHeight→desiredValue`**

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

First, create a station to occupy

```
LX3030 = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303]; LX3030 // PPDMS
```

```
41°48'44.7844"N, 72°15'2.04232"W, 157.303, NAD 83 (CORS96)
```

Create a new point due north of LX3030. **`Direct`** returns the new point and the back azimuth as a list.

```
Direct[LX3030, 0 °, 1000.] // PPDMS
```

```
{41°49'17.1965"N, 72°15'2.04232"W, 0, NAD 83 (CORS96), 180°0'0"}
```

Create a series of new points by iterating over azimuth from north around the circle in 20° increments. Note that the back azimuths are **not** usually 180° different than the forward azimuth.

```
Table[Direct[LX3030, α°, 100000.], {α, 0, 360, 20}] // PPDMS // TableForm
```

```
42°42'45.7389"N, 72°15'2.04232"W, 0, NAD 83 (CORS96)       180°0'0"
42°39'27.5952"N, 71°50'0.39567"W, 0, NAD 83 (CORS96)       200°16'49.3973"
42°29'58.0255"N, 71°28'6.9327"W, 0, NAD 83 (CORS96)        220°31'29.4133"
42°15'28.1366"N, 71°12'3.67486"W, 0, NAD 83 (CORS96)       240°42'10.0007"
41°57'45.523"N, 71°3'45.2614"W, 0, NAD 83 (CORS96)         260°47'35.5407"
41°39'0.0132747"N, 71°4'6.0063"W, 0, NAD 83 (CORS96)       280°47'13.0346"
41°21'27.2623"N, 71°12'56.2101"W, 0, NAD 83 (CORS96)       300°41'13.0058"
41°7'12.4875"N, 71°29'6.68637"W, 0, NAD 83 (CORS96)        320°30'24.5882"
40°57'56.2124"N, 71°50'39.4045"W, 0, NAD 83 (CORS96)       340°16'7.07839"
40°54'43.3202"N, 72°15'2.04232"W, 0, NAD 83 (CORS96)       360°0'0"
40°57'56.2124"N, 72°39'24.6802"W, 0, NAD 83 (CORS96)       19°43'52.9216"
41°7'12.4875"N, 73°0'57.3983"W, 0, NAD 83 (CORS96)         39°29'35.4118"
41°21'27.2623"N, 73°17'7.87458"W, 0, NAD 83 (CORS96)       59°18'46.9942"
41°39'0.0132747"N, 73°25'58.0783"W, 0, NAD 83 (CORS96)     79°12'46.9654"
41°57'45.523"N, 73°26'18.8232"W, 0, NAD 83 (CORS96)        99°12'24.4593"
42°15'28.1366"N, 73°18'0.409785"W, 0, NAD 83 (CORS96)      119°17'49.9993"
42°29'58.0255"N, 73°1'57.1519"W, 0, NAD 83 (CORS96)        139°28'30.5867"
42°39'27.5952"N, 72°40'3.68897"W, 0, NAD 83 (CORS96)       159°43'10.6027"
42°42'45.7389"N, 72°15'2.04232"W, 0, NAD 83 (CORS96)       180°0'0"
```

```
Remove[LX3030]
```

InverseGeodetic

The geodetic INVERSE problem is: for two stations whose coordinates are given in latitude and longitude, compute the geodesic distance and the forward/backwards azimuths between them.

`InverseGeodetic[from, to, opts]`	Compute the geodesic distance and forward/backwards azimuths from station *from* to station *to*. The return value is a list {*geodesicDistance*, *fore Az*, *backAz*, *d*} where δ gives the difference between the last two iterates when the iteration stopped.
`InverseGeodetic[{from, to}, opts]`	This version accepts a list of arguments.
`InverseMethod`	An option for `InverseGeodetic`. There are three inverse algorithms that have been implemented, namely `ExtendedNewtonRaphson`, `Robbin61`, `Vincenty75`. The default is `Vincenty75`.
`InverseMaxIterations`	An option for `InverseGeodetic`. Maximum number of iterations. Default is 10.

There are many algorithms given in the literature for InverseGeodetic. We chose to implement Vincenty's algorithm (1975) because it works well under all conditions. However, like all other geodetic computations, there are other ways to do it. Therefore, even though this implementation provides only one way, the code was written to easily permit the implementation of other methods, perhaps for comparison.

from is a geodetic (blh) point from which to begin.

to is a geodetic (blh) point at which to end.

InverseMethod is an option that can take the following values: *ExtendedNewtonRaphson, Robbin61, Vincenty75*. *Extended-NewtonRaphson* is slowest but handles nearly antipodal stations properly. Based on Bowring 1985s method. *Robbin61* is the fastest and least accurate. Suitable only for fairly short baselines. *Vincenty75* is slower and more accurate than *Robbin61* but faster and less accurate than *ExtendedNewtonRaphson*. *Vincenty75* is the best choice for relatively long baselines unless stations are nearly antipodal.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

First, create a station to occupy

```
from = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303]; from // PPDMS
```

```
41°48'44.7844"N, 72°15'2.04232"W, 157.303, NAD 83 (CORS96)
```

Create a list of points to inverse with using **`Direct`** to create them. 10,000 KM base lines.

```
to = Table[Direct[from, α°, 10000000.], {α, 0, 360, 20}][[All, 1]];
to // PPDMS
```

```
{48°29'30.9262"N, 107°44'57.9577"E, 0, NAD 83 (CORS96),
 44°43'44.269"N, 78°56'48.4653"E, 0, NAD 83 (CORS96),
 35°0'44.5074"N, 55°58'51.8199"E, 0, NAD 83 (CORS96),
 21°58'58.2292"N, 38°34'9.79888"E, 0, NAD 83 (CORS96),
 7°26'25.3588"N, 24°15'19.301"E, 0, NAD 83 (CORS96),
 7°31'35.9402"S, 10°53'43.9112"E, 0, NAD 83 (CORS96),
 22°2'53.7331"S, 3°23'32.1989"W, 0, NAD 83 (CORS96),
 35°2'24.4204"S, 20°44'49.9907"W, 0, NAD 83 (CORS96),
 44°42'51.5388"S, 43°36'35.2035"W, 0, NAD 83 (CORS96),
 48°27'23.6483"S, 72°15'2.04232"W, 0, NAD 83 (CORS96),
 44°42'51.5388"S, 100°53'28.8812"W, 0, NAD 83 (CORS96),
 35°2'24.4204"S, 123°45'14.094"W, 0, NAD 83 (CORS96),
 22°2'53.7331"S, 141°6'31.8857"W, 0, NAD 83 (CORS96),
 7°31'35.9402"S, 155°23'47.9959"W, 0, NAD 83 (CORS96),
 7°26'25.3588"N, 168°45'23.3856"W, 0, NAD 83 (CORS96),
 21°58'58.2292"N, 183°4'13.8835"W, 0, NAD 83 (CORS96),
 35°0'44.5074"N, 200°28'55.9045"W, 0, NAD 83 (CORS96),
 44°43'44.269"N, 223°26'52.5499"W, 0, NAD 83 (CORS96),
 48°29'30.9262"N, 107°44'57.9577"E, 0, NAD 83 (CORS96)}
```

Compute the inverses. Notice that the forward inverse matches the values input to Direct exactly but that the backwards inverses generally do not. This is a consequence of the convergence of the Meridians.

```
With[{i = InverseGeodetic[from, #]}, {i⟦1⟧ // P, PPDMS[i⟦2⟧], PPDMS[i⟦3⟧], i⟦4⟧}] & /@ to //
  TableForm
```

```
10000000.     0°0'0"        360°0'0"            1.
10000000.     20°0'0"       338°58'34.6128"     -3.33067×10^-14
10000000.     40°0'0"       324°11'4.65362"     -5.77316×10^-15
10000000.     60°0'0"       315°49'46.5197"     -2.88658×10^-15
10000000.     80°0'0"       312°9'26.9305"      -1.31006×10^-14
10000000.     100°0'0"      312°8'42.1563"      -1.01918×10^-13
10000000.     120°0'0"      315°48'14.5057"     -4.44089×10^-16
10000000.     140°0'0"      324°10'14.3336"     -4.44089×10^-16
10000000.     160°0'0"      338°58'54.6122"     -6.87228×10^-14
10000000.     180°0'0"      360°0'0"            1.
10000000.     200°0'0"      21°1'5.38784"       6.87228×10^-14
10000000.     220°0'0"      35°49'45.6664"      4.44089×10^-16
10000000.     240°0'0"      44°11'45.4943"      4.44089×10^-16
10000000.     260°0'0"      47°51'17.8437"      1.01918×10^-13
10000000.     280°0'0"      47°50'33.0695"      1.31006×10^-14
10000000.     300°0'0"      44°10'13.4803"      2.88658×10^-15
10000000.     320°0'0"      35°48'55.3464"      5.77316×10^-15
10000000.     340°0'0"      21°1'25.3872"       3.33067×10^-14
10000000.     0°0'0"        360°0'0"            1.
```

Repeat with the Robbin61 method. Notice that none of the values are exact any more.

```
With[{i = InverseGeodetic[from, #, InverseMethod → Robbin61]},
    {i⟦1⟧ // P, PPDMS[i⟦2⟧], PPDMS[i⟦3⟧], i⟦4⟧}] & /@ to // TableForm
```

```
9990209.388     0°0'0"               360°0'0"               0
9990623.743     20°1'12.8096"        338°57'5.0608"         0
9991891.664     40°1'34.2491"        324°10'49.2721"        0
9994052.912     60°0'38.8971"        315°54'41.6709"        0
9997071.79      79°58'40.9265"       312°22'23.713"         0
10000735.21     99°56'29.0249"       312°29'29.2115"        0
10004603.11     119°55'4.77155"      316°13'0.589459"       0
10008059.73     139°55'14.0526"      324°32'33.5119"        0
10010461.32     159°57'4.51415"      339°12'9.34875"        0
10011321.97     180°0'0"             360°0'0"               0
10010461.32     200°2'55.4859"       20°47'50.6512"         0
10008059.73     220°4'45.9474"       35°27'26.4881"         0
10004603.11     240°4'55.2285"       43°46'59.4105"         0
10000735.21     260°3'30.9751"       47°30'30.7885"         0
9997071.79      280°1'19.0735"       47°37'36.287"          0
9994052.912     299°59'21.1029"      44°5'18.3291"          0
9991891.664     319°58'25.7509"      35°49'10.7279"         0
9990623.743     339°58'47.1904"      21°2'54.9392"          0
9990209.388     0°0'0"               360°0'0"               0
```

Repeat with the extended Newton-Raphson method. Notice that none of the values are exact any more, however the discrepancy lies in the Direct routine rather than with the Newton-Raphson method.

```
With[{i = InverseGeodetic[from, #, InverseMethod → ExtendedNewtonRaphson]},
    {i⟦1⟧ // P, PPDMS[i⟦2⟧], PPDMS[i⟦3⟧], i⟦4⟧}] & /@ to // TableForm
```

```
10000002.81    0°0'0"       -0°0'0"           1.
10000002.58    20°0'0"      -21°1'25.3872"    -6.90603×10^-12
10000001.6     40°0'0"      -35°48'55.3464"   -1.64984×10^-11
9999999.924    60°0'0"      -44°10'13.4803"   1.66434×10^-11
9999999.421    80°0'0"      -47°50'33.0695"   4.54294×10^-11
10000000.58    100°0'0"     -47°51'17.8437"   2.36546×10^-11
10000000.07    120°0'0"     -44°11'45.4943"   -5.93081×10^-13
9999998.4      140°0'0"     -35°49'45.6664"   -3.01892×10^-12
9999997.417    160°0'0"     -21°1'5.38784"    -4.89275×10^-13
9999997.186    180°0'0"     0°0'0"            1.
9999997.417    -160°0'0"    21°1'5.38784"     4.89275×10^-13
9999998.4      -140°0'0"    35°49'45.6664"    3.01892×10^-12
10000000.07    -120°0'0"    44°11'45.4943"    5.93081×10^-13
10000000.58    -100°0'0"    47°51'17.8437"    -2.36546×10^-11
9999999.421    -80°0'0"     47°50'33.0695"    -4.54294×10^-11
9999999.924    -60°0'0"     44°10'13.4803"    -1.66434×10^-11
10000001.6     -40°0'0"     35°48'55.3464"    1.64984×10^-11
10000002.58    -20°0'0"     21°1'25.3872"     6.90603×10^-12
10000002.81    0°0'0"       -0°0'0"           1.
```

```
Remove[from, to]
```

Great Circle Distance

`DistanceGreatCircle[`*`from`*`,`*`to`*`,`*`r`*`]`	Computes the great circle distance between two geodetic (`blh`) points. *r* is an optional argument whose default value is the semimajor axis of *from*'s ellipsoid. If *r* is supplied with a value, it is used as the radius of the sphere.
`DistanceGreatCircle[{{`*`from`*`,`*`to`*`}..},`*`r`*`]`	This version accepts a list of one or more stations. If *r* is supplied with a value, it is used as the radius of the sphere for all the station pairs.
`DistanceGreatCircle[{`*`from`*`,`*`to`*`},`*`r`*`]`	This version accepts a list of one station. It is mainly helpful for expressions like **`DistanceGreatCircle /@ listOfPositions`**.
`DistanceGreatCircle[{{`*`from`*`,`*`to`*`,`*`r`*`}..}]`	This version accepts a list of one or more stations. The radius of the sphere, *r*, is not optional in this version. This version allows each pair to have its own radius.

Great circle distances are the shortest distance between two points on the surface of a sphere. Although great circle distances are not commonly used in geodesy, they are included for completeness and for comparison with geodesic distances, such as are computed by **`Direct`**. Great circle distances in this package are computed with Sinnott's method.

This loads the package.

```
Needs["GeometricalGeodesy`"]
```

Create a list of points to compute great circle distances between. The ⟦All,1⟧ syntax extracts just the points (recall that **Direct** returns both the point and the back azimuth in a list).

```
from = MakeBLH[0, 0, 0];
toList = (Direct[from, 33 °, #] & /@ Range[0, 10000000, 500000])⟦All, 1⟧
```

```
{blh[0, 0, 0, NAD 83 (CORS96)], blh[0.0661676, 0.0427578, 0, NAD 83 (CORS96)],
 blh[0.132207, 0.0858897, 0, NAD 83 (CORS96)],
 blh[0.197986, 0.12978, 0, NAD 83 (CORS96)], blh[0.263364, 0.174835, 0, NAD 83 (CORS96)],
 blh[0.32819, 0.221491, 0, NAD 83 (CORS96)], blh[0.392292, 0.27023, 0, NAD 83 (CORS96)],
 blh[0.455476, 0.321592, 0, NAD 83 (CORS96)], blh[0.517516, 0.37619, 0, NAD 83 (CORS96)],
 blh[0.578141, 0.434719, 0, NAD 83 (CORS96)], blh[0.637031, 0.497978, 0, NAD 83 (CORS96)],
 blh[0.693795, 0.56687, 0, NAD 83 (CORS96)], blh[0.747959, 0.642397, 0, NAD 83 (CORS96)],
 blh[0.798946, 0.725641, 0, NAD 83 (CORS96)], blh[0.846061, 0.817685, 0, NAD 83 (CORS96)],
 blh[0.888483, 0.919492, 0, NAD 83 (CORS96)], blh[0.92527, 1.03169, 0, NAD 83 (CORS96)],
 blh[0.9554, 1.15427, 0, NAD 83 (CORS96)], blh[0.977847, 1.28627, 0, NAD 83 (CORS96)],
 blh[0.991712, 1.42554, 0, NAD 83 (CORS96)], blh[0.99637, 1.5688, 0, NAD 83 (CORS96)]}
```

Note that the great circle distances are longer than their geodesic counterparts.

```
DistanceGreatCircle[from, #] & /@ toList // P
```

```
{0., 502365.17, 1004686.389, 1506920.816, 2009027.797, 2510969.885,
 3012713.778, 3514231.137, 4015499.286, 4516501.753, 5017228.659,
 5517676.938, 6017850.384, 6517759.532, 7017421.377, 7516858.933,
 8016100.654, 8515179.731, 9014133.276, 9513001.418, 10011826.33}
```

Compare the results on the unit sphere.

```
DistanceGreatCircle[from, #, 1] & /@ toList // P
```

```
{0., 0.07876362172, 0.1575203526, 0.2362634757, 0.3149866171,
 0.3936839057, 0.4723501201, 0.5509808173, 0.6295724419, 0.7081224114,
 0.786629177, 0.8650922578, 0.9435122487, 1.021890802, 1.100230581,
 1.178535195, 1.256809105, 1.335057515, 1.413286243, 1.491501581, 1.569710141}
```

```
Remove[from, toList]
```

Geometric Geodesy Map Projections

Creating and Using a Projection System

This package implements the major conformal projections performed in geometric geodesy. These include Mercator, Lambert conformal conic, and stereographic. All projections work the same way. The user creates a projection system by invoking **`MakeProjectionSystem`**, which is an object holding the type of projection and its operational parameters. Many projection systems can be created in the same session; you do not have to only have one of them at a time.

A projection system is made active by invoking **`SetProjectionSystem`**. This copies the projection system into a private global variable so the projection mathematical software has access to the information it contains. Thereafter, all projections and inverse projections occur by invoking **`ProjectBLH`**.

`MakeProjectionSystem[`*`projection`*`, `*`opts`*`]`	Returns a system holding all the parameters needed to compute the projection

`projection`	The name of the desired projection. Values include **`Mercator, InverseMercator, ObliqueMercator, LambertConformalConic, Orthographic, Stereographic`**
`opts`	The options define the projection; see the following. If value is given in degrees, it must be enclosed in parentheses. Eg, **`CentralMeridian→(72°)`**.
`CentralMeridian`	an angle in radians; default value is 0 °.
`CentralParallel`	default value is 0 °
`Standards`	provides the location of the standards. If only one, then usage is **`Standards→`*`value`***. If more than one, then usage is **`Standards→{`*`value1,value2`*`}`**
`EarthModel`	either Spherical or a valid ellipsoid name
`SphericalRadius`	only used if EarthModel → Spherical. Then, this is the radius of the sphere to project upon.
`IncludeHeight`	a boolean value. If True, then the result is a 3 D grd point.
`IncludeScaleFactor`	a boolean value. If True, then the result is a list of the grd point and the scale factor of the projection at that location.
`GrdForm`	a boolean value. If True, then the result is a grd point. If False, then a Mathematica vector.

```
Needs["GeometricalGeodesy`"]
```

The easiest way to create a projection system is simply to provide the name of the projection. In this simplest form, all the defaults indicate a spherical Earth model, unit radius centered at the Prime Meridian and the equator without heights or the scale factor. The object returned holds all the information needed to actually perform the projection.

```
MakeProjectionSystem[Mercator]
```

```
{ProjectionName → Mercator, CentralMeridian → 0, CentralParallel → 0,
 Standards → Null, EarthModel → Spherical, SphericalRadius → 1, CentralScaleFactor → 1,
 CentralPoints → Null, OriginFalseEasting → 0, OriginFalseNorthing → 0}
```

`SetProjectionSystem[sys]`	Set the active projection system to `sys`. This function must be called before any projections can be performed. `sys` is the object returned by **`MakeProjectionSystem`**.
`$projectionSystem`	A global variable holding the current projection system.

```
Needs["GeometricalGeodesy`"]
```

This package is designed to accommodate points being represented by many positions in many coordinate systems. It is assumed that the user may want to project points into more than one cartographic grid. Therefore, since there may be more than one projection system, in order to use the projection, it must be set as the active one.

```
projSys = MakeProjectionSystem[Mercator];
SetProjectionSystem[projSys];
```

You do not need to assign the result of **`MakeProjectionSystem`** to a variable; it can be "fed" into **`SetProjection-System`** directly.

```
SetProjectionSystem[MakeProjectionSystem[Mercator]];
```

```
$projectionSystem
```

```
{ProjectionName → Mercator, CentralMeridian → 0, CentralParallel → 0,
 Standards → Null, EarthModel → Spherical, SphericalRadius → 1, CentralScaleFactor → 1,
 CentralPoints → Null, OriginFalseEasting → 0, OriginFalseNorthing → 0}
```

`ProjectBLH[ϕ, λ]`	Compute the easting and northing values associated with ϕ, λ in the projection system set with SetProjectionSystem. ϕ and λ are given in radians.
`ProjectBLH[{ϕ, λ}]`	This is the same as above except that the arguments are a list instead of individually
`ProjectBLH[ϕ, λ, h]`	same as above but a height argument is supplied. The height argument doesn' t do anything; it just goes along for the ride. It is included for convenience for users who are working with projected, 3 D points.
`ProjectBLH[{ϕ, λ, h}]`	same as above but the argument is a single list of coordinates, rather than giving them individually.
`ProjectBLH[p]`	Project point *p*, which is of type **`blh`**

```
Needs["GeometricalGeodesy`"]
```

All projections are done by calling the single function **`ProjectBLH`**, which determines which projection to use, along with all the parameters needed to define the projection, by consulting **`$projectionSystem`**. **`$projectionSystem`** is a read-only global variable that is set by **`SetProjectionSystem`**.

```
SetProjectionSystem[MakeProjectionSystem[Mercator]]; ProjectBLH[42. °, 280. °]
ProjectBLH[0. °, 0. °]
```

```
{4.88692, 0.809167}
```

```
{0., -1.11022 × 10^-16}
```

Changing the central meridian affects the values of the eastings. Notice in the following example that the value of the central meridian is enclosed within parentheses. This is necessary because the angle must be in radians and the substitution will not proceed correctly without them. Comparing with the previous example, the easting value changes but not the northing.

```
SetProjectionSystem[MakeProjectionSystem[Mercator, CentralMeridian → (270 °)]];
ProjectBLH[42. °, 280. °]
```

```
{0.174533, 0.809167}
```

Including the height in no way changes the projected coordinates. It simply adds in a third dimension.

```
ProjectBLH[42. °, 280. °, 900, IncludeHeight → True]
```

```
{0.174533, 0.809167, 900}
```

Most cartographic projections depict distances that are distorted from the normal section they represent. The ratio of the cartographic distance to the normal section distance is the *projection scale factor*. You can direct the system to include the projection scale factor in the result, which changes the result to a list with the first element being the projected point and the second being the scale factor at that point.

```
ProjectBLH[42. °, 280. °, IncludeScaleFactor → True]
```

```
{{0.174533, 0.809167}, 1.34563}
```

ProjectBLH can return **grd** objects instead of lists of coordinates.

```
ProjectBLH[42. °, 280. °, IncludeScaleFactor → True, GrdForm → True]
```

```
{grd[1.11319 × 10^6, 5.13238 × 10^6, 0, DefaultGroup], 1.34361}
```

The radius of a spherical Earth model is set with the **SphericalRadius** option.

```
SetProjectionSystem[MakeProjectionSystem[Mercator,
   CentralMeridian → (270 °), SphericalRadius → semiMajor[GRS80]]];
ProjectBLH[42. °, 280. °] //
 P
```

```
{1113194.908, 5160979.444}
```

The an ellipsoidal Earth model is specified with the **EarthModel** option. Supplying a value for this option fundamentally alters the behavior of ProjectBLH. If **EarthModel** is specified, the ellipsoidal versions of the equations are automatically used. Otherwise, the spherical versions are used.

```
SetProjectionSystem[
 MakeProjectionSystem[Mercator, CentralMeridian → (270 °), EarthModel → GRS80]];
ProjectBLH[42. °, 280. °] // P
```

```
{1113194.908, 5132380.528}
```

Inverse Projection

Cartographic projections map geodetic coordinates (latitude and longitude) to grid coordinates. All projections have inverse mappings that transform grid coordinates back to geodetic. Once the mapping system has been created using `SetProjectionSystem`, grid coordinates can be inverted to geodetic using the `InverseProjection` function.

`InverseProjection[x, y]`	Compute the geodetic latitude and longitude that x and y are the image of in the projection system set with SetProjectionSystem.
`InverseProjection[{x, y}]`	This is the same as above except that the arguments are a list instead of individually
`InverseProjection[x, y, h]`	same as above but a height argument is supplied. The height argument doesn' t do anything; it just goes along for the ride. It is included for convenience for users who are working with 3 D points.
`InverseProjection[{x, y, h}]`	same as above but the argument is a single list of coordinates, rather than giving them individually.
`InverseProjection[p]`	*p* is of type **grd**.

```
Needs["GeometricalGeodesy`"]
```

Project a pair of coordinates

```
SetProjectionSystem[
 MakeProjectionSystem[Mercator, CentralMeridian → (270 °), EarthModel → GRS80]];
p = ProjectBLH[42. °, 280. °];
p // P
InverseProjection[p] // PPDMS
```

```
{1113194.908, 5132380.528}

{42°0'0", 280°0'0"}
```

Project a blh object, with and without height

```
lx3030 = MakeBLH[DMS[41, 48, 44.78441, N], DMS[72, 15, 2.04232, W], 157.303, ITRF00];
p = ProjectBLH[lx3030];
Print["Projected without height: ", p // P];
Print["Inverse projection takes us back: ", InverseProjection[p]];
p = ProjectBLH[lx3030, IncludeHeight → True];
Print["Projected with height: ", p // P];
Print["Inverse projection takes us back: ",
  MakeBLH[InverseProjection[p, IncludeHeight → True]] // PPDMS];
```

```
   Projected without height: {-38099158.88, 5104430.577}

   Inverse projection takes us back: {0.729765, -1.26101}

   Projected with height: {-38099158.88, 5104430.577, 157.303}

   Inverse projection takes us back:
    41°48'44.7844"N, 72°15'2.04232"W, 157.303, NAD 83 (CORS96)
```

Mercator, Spherical

These examples construct a graticule for the Mercator projection and then project physical and political boundary points over the graticule to see how the Mercator projection depicts the Earth. This section uses a spherical Earth model of unit radius.

```
Needs["GeometricalGeodesy`"]
```

The following examples are taken from Snyder, John P. (1987) Map Projections -- A Working Manual. U.S. Geological Survey Professional Paper 1395, p. 266: Mercator, Spherical Earth Model

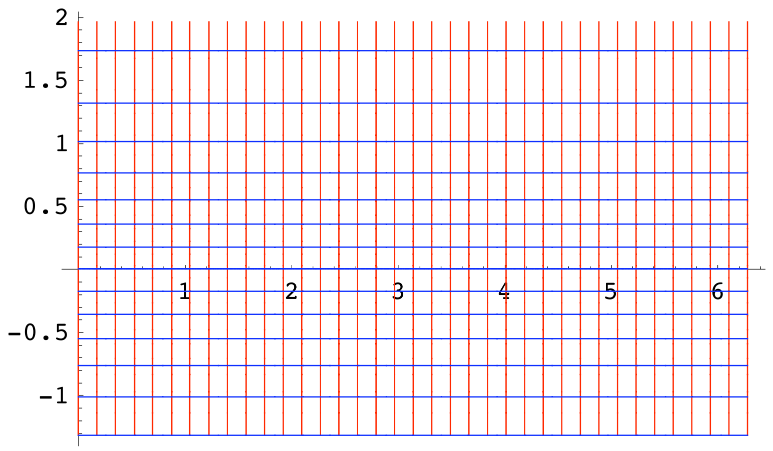
```
SetProjectionSystem[MakeProjectionSystem[Mercator, CentralMeridian → (-180 °)]];
{{e, n}, k} = ProjectBLH[35. °, -75. °, IncludeScaleFactor → True];
{e, n, k} // P
InverseProjection[e, n] // PPDMS
Remove[e, n, k]
```

```
{1.832595715, 0.6528365797, 1.220774589}
```

```
{35°0'0", -75°0'0"}
```

Here's the Mercator graticule. Build two lists of graphics. The first holds meridians (red); the second holds parallels (blue)

```
maxLat = 74;
minLat = -60;
meridians =
  (ParametricPlot[ProjectBLH[ϕ °, # °], {ϕ, minLat, maxLat}, PlotStyle → {RGBColor[1, 0, 0]},
     DisplayFunction → Identity] & /@ Range[-180, 180, 10]);
parallels = (ParametricPlot[ProjectBLH[# °, λ °], {λ, -180, 180}, PlotStyle →
       {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[minLat, maxLat, 10]);
Show[meridians, parallels, DisplayFunction → $DisplayFunction, PlotRange → All];
```

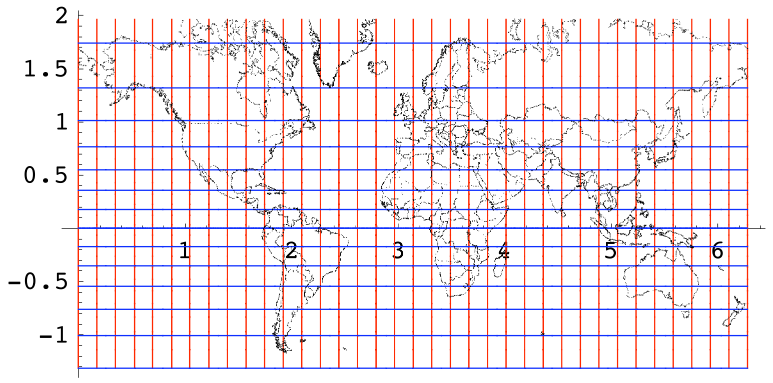


These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

Project and display them. Observe how the distortions become highly exaggerated far to the north or south of the central parallel. As is well-known, the Mercator projection depicts Greenland to be roughly the same size as South America when, in fact, is roughly 1/6 as large.

```
Show[
  meridians,
  parallels, ListPlot[
   (ProjectBLH[#⟦2⟧ °, #⟦3⟧ °]) & /@ Select[thinDataPoints, #⟦2⟧ < maxLat && #⟦2⟧ > minLat &],
   AspectRatio → Automatic, DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction,
  AspectRatio → Automatic];
```



```
Remove[maxLat, minLat, meridians, parallels]
```

Mercator, Ellipsoidal

This section illustrates the Mercator projection based upon an ellipsoidal world model; see Snyder p. 267. The usage of the interface is essentially the same as with the spherical model in that only the additional ellipsoidal Earth model parameters are provided. However, internally, the package is invoking an entirely different set of mathematics to perform the computations.

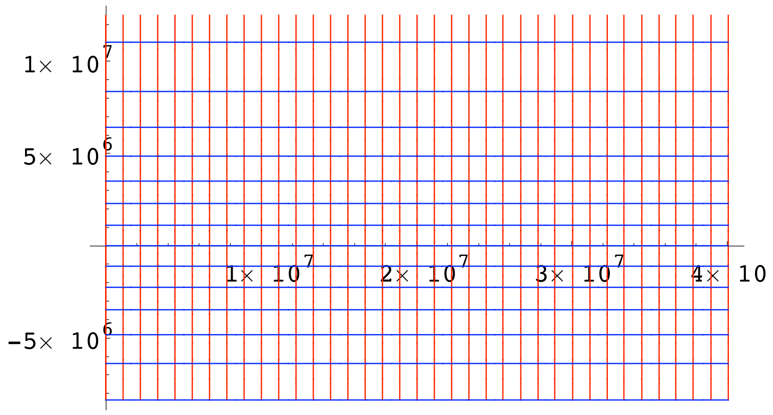
```
SetProjectionSystem[
  MakeProjectionSystem[Mercator, EarthModel → Clarke1866, CentralMeridian → (-180 °)]];
{{e, n}, k} = ProjectBLH[35. °, -75. °, IncludeScaleFactor → True];
{e, n, k} // P
InverseProjection[e, n] // PPDMS
Remove[e, n, k]
```

```
{11688673.72, 4139145.663, 1.219414608}
```

```
{35°0'0", -75°0'0"}
```

Next, find those points from the entire set that fall between 74° North and 60° South latitude. Create and display the graticule for this region. Being a standard cylindrical projection, the graticule is a rectangular grid whose line spacing varies with latitude but not longitude.

```
maxLat = 74;
minLat = -60;
meridians =
  (ParametricPlot[ProjectBLH[ϕ°, #°], {ϕ, minLat, maxLat}, PlotStyle → {RGBColor[1, 0, 0]},
     DisplayFunction → Identity] & /@ Range[-180, 180, 10]);
parallels = (ParametricPlot[ProjectBLH[#°, λ°], {λ, -180, 180}, PlotStyle →
      {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[minLat, maxLat, 10]);
Show[meridians, parallels, DisplayFunction → $DisplayFunction, PlotRange → All];
```

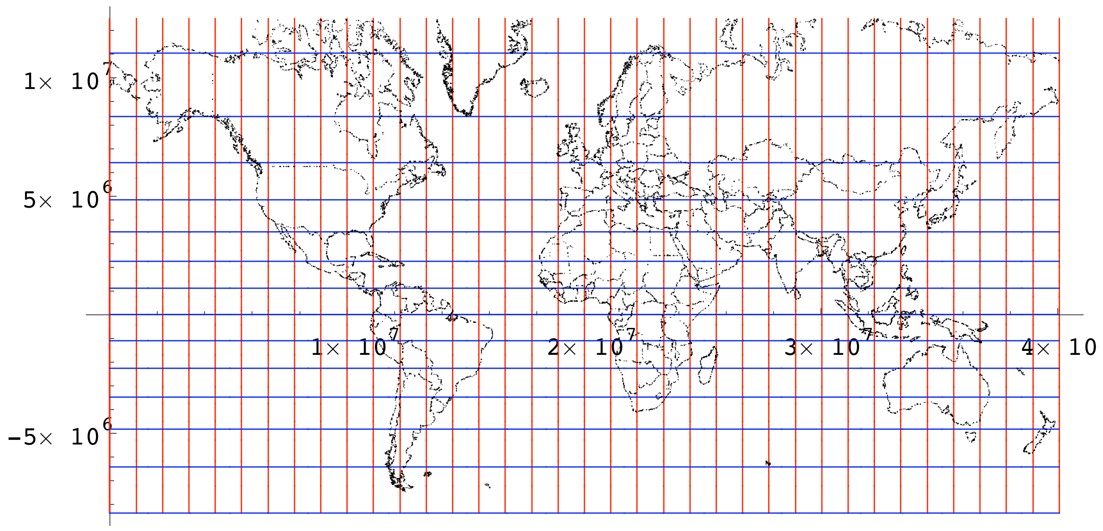


These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

Now, find those points from the entire set that fall between 74° North and 60° South latitude. Then project and display the results. Comparing the following graphic with the previous reveals not discernible difference. This is because the eccentricity of GRS80, as with all reference ellipsoids, is very small, meaning the Earth is, in fact, nearly spherical. However, note the difference in the axes. The figure below is based on a realistic Earth model whereas the figure above uses a spherical Earth model of unit radius. The easting and northings of the ellipsoidal model are usually considered more suitable for mapping purposes because the eastings and northings have units of meters: one meter of change on the map corresponds to one meter on the ellipsoid divided by the projection scale factor.

```
Show[
  meridians,
  parallels,
  ListPlot[
   (ProjectBLH[#[[2]] °, #[[3]] °]) & /@ Select[thinDataPoints, #[[2]] < maxLat && #[[2]] > minLat &],
   AspectRatio → Automatic, DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Remove[maxLat, minLat, meridians, parallels]
```

Transverse Mercator, Spherical

```
Needs["GeometricalGeodesy`"]
```

The Transverse Mercator projection is the primary projection of the Universal Transverse Mercator mapping system and is also the projection used in the State Plane Coordinate System for regions that run long North and South. These examples come from Snyder, p. 268

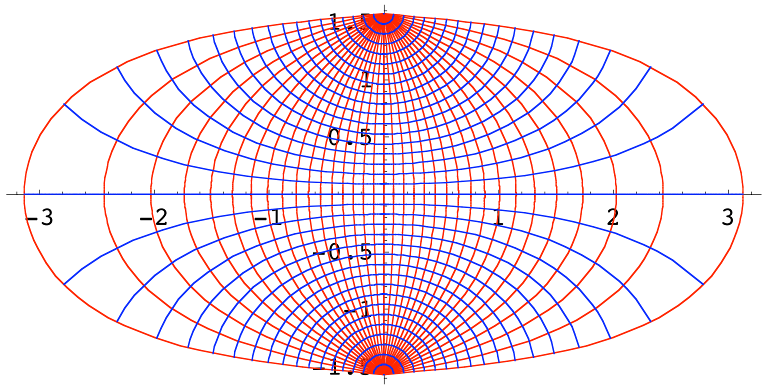
```
SetProjectionSystem[
  MakeProjectionSystem[TransverseMercator, CentralMeridian → (-75 °)]
 ];
pBLH = MakeBLH[DMS[40, 30, 0, N], DMS[73, 30, 0, W]];
p = {{e, n}, k} = ProjectBLH[pBLH, IncludeScaleFactor → True];
p // P
InverseProjection[p[[1]]] // PPDMS
Remove[pBLH, p, e, n, k]
```

```
{{0.01990773717, 0.7070276087}, 1.000198166}
```

```
{40°30'0", -73°30'0"}
```

Here's the Transverse Mercator graticule. TM cannot depict the entire Earth at once; only one hemisphere can be displayed.

```
minLon = -160;
maxLon = 10;
ΔLon = 5;
meridians =
  (ParametricPlot[ProjectBLH[ϕ°, #°], {ϕ, -90, 90}, PlotStyle → {RGBColor[1, 0, 0]},
      DisplayFunction → Identity] & /@ Range[minLon, maxLon, ΔLon]);
parallels = (ParametricPlot[ProjectBLH[#°, λ°], {λ, minLon, maxLon}, PlotStyle →
        {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[-90, 90, ΔLon]);
gratPlot = Show[meridians, parallels, DisplayFunction → $DisplayFunction,
   PlotRange → All, AspectRatio → Automatic];
```

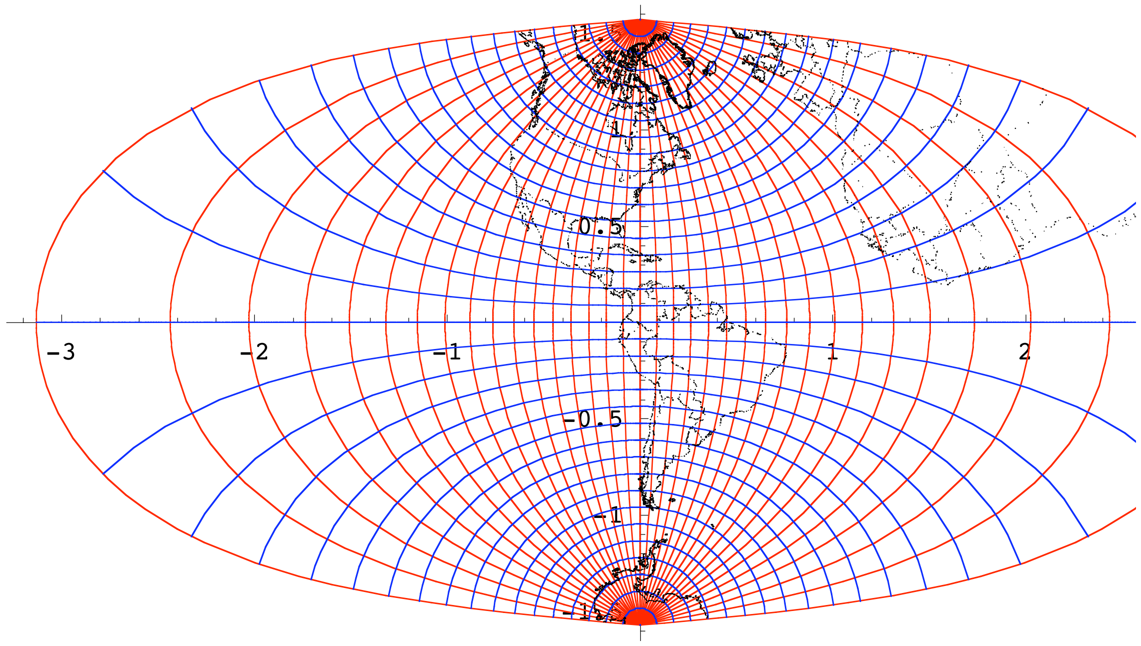


Now draw the world. Observe how the distortions become highly exaggerated far to the east or west of the central meridian. This is opposite that of the Mercator projection.

These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

```
Show[
  gratPlot,
  ListPlot[
   ProjectBLH[#⟦2⟧ °, #⟦3⟧ °] & /@ Select[thinDataPoints, #⟦3⟧ ≥ minLon && #⟦3⟧ ≤ maxLon &],
   DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Remove[maxLat, minLat, meridians, parallels]
```

Transverse Mercator, Ellipsoidal

```
Needs["GeometricalGeodesy`"]
```

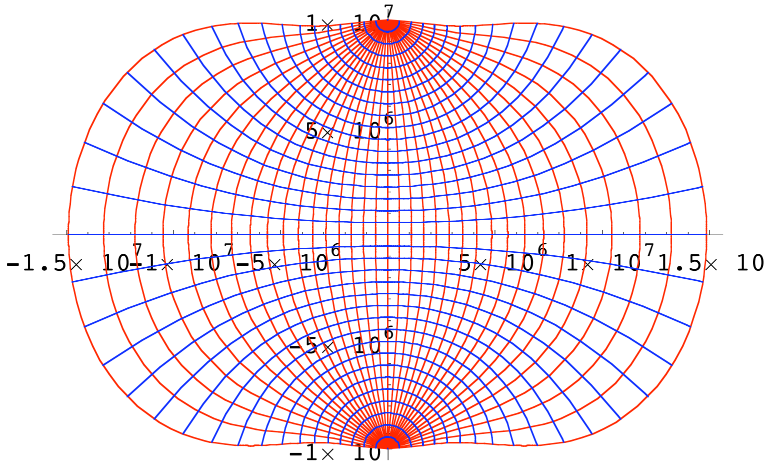
We'll create a projection system based on the Transverse Mercator projection using the Clarke 1866 reference ellipsoid, as would be the case when mapping in NAD 27. This example comes from Snyder, p. 269

```
SetProjectionSystem[
  MakeProjectionSystem[
   TransverseMercator, EarthModel → Clarke1866,
   CentralMeridian → (-75 °), CentralScaleFactor → 0.9996]];
{{e, n}, k} = ProjectBLH[DMS[40, 30, 0, N], DMS[73, 30, 0, W], IncludeScaleFactor → True];
{e, n} // P
{ϕ, λ} = InverseProjection[e, n];
{ϕ, λ} / ° // P
{ϕ // PPDMSϕ, λ // PPDMSλ}
Remove[e, n, ϕ, λ, k]
```

```
{127106.4674, 4484124.435}

{40.5, -73.5}

{40°30'0"N, 73°30'0"W}
```

Here's the Transverse Mercator graticule

```
minLon = -160;
maxLon = 10;
ΔLon = 5;
meridians =
  (ParametricPlot[ProjectBLH[ϕ°, #°], {ϕ, -90, 90}, PlotStyle → {RGBColor[1, 0, 0]},
     DisplayFunction → Identity] & /@ Range[minLon, maxLon, ΔLon]);
parallels = (ParametricPlot[ProjectBLH[#°, λ°], {λ, minLon, maxLon}, PlotStyle →
       {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[-90, 90, ΔLon]);
gratPlot = Show[meridians, parallels, DisplayFunction → $DisplayFunction,
   PlotRange → All, AspectRatio → Automatic];
```



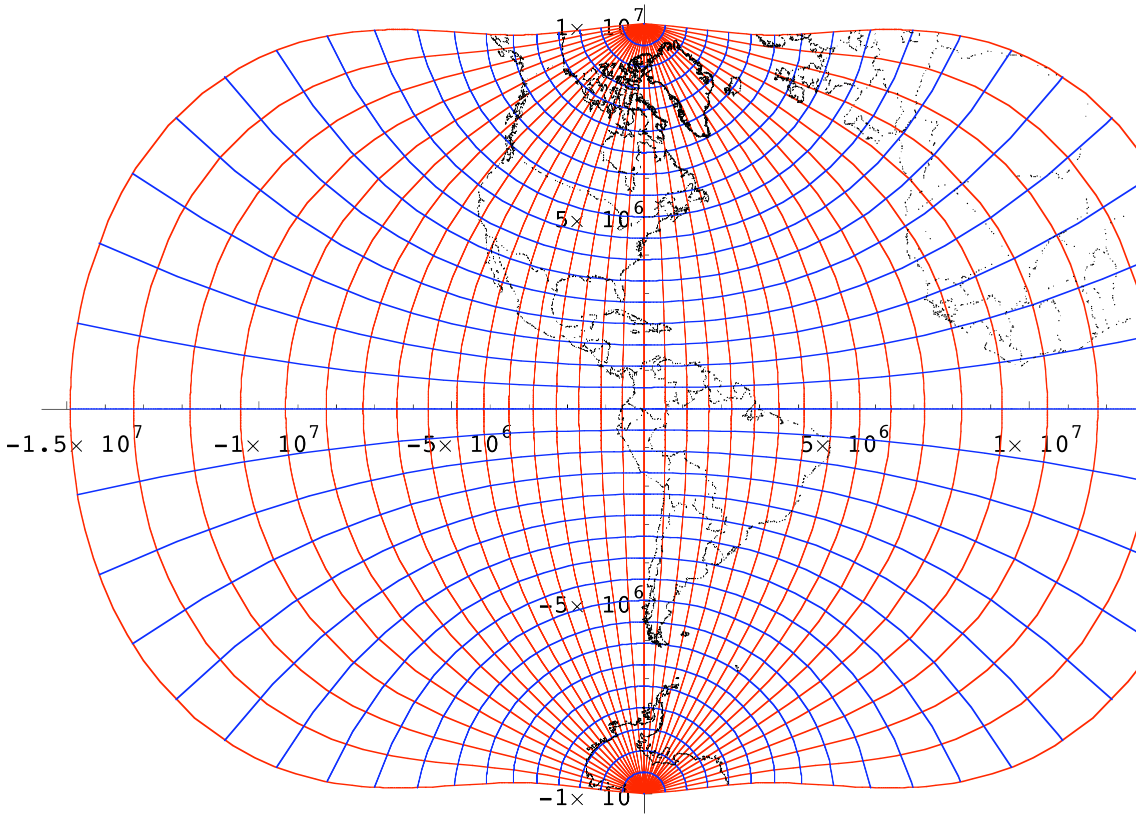
Note how the ellipsoidal graticule is visually different from the spherical model: there are inflection points near the poles. Now draw the hemisphere

---

These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

```
Show[
  gratPlot,
  ListPlot[
   ProjectBLH[#⟦2⟧ °, #⟦3⟧ °] & /@ Select[thinDataPoints, #⟦3⟧ ≥ minLon && #⟦3⟧ ≤ maxLon &],
   DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Remove[maxLat, minLat, meridians, parallels]
```

Oblique Mercator, Spherical

```
Needs["GeometricalGeodesy`"]
```

This example is Snyder, p. 272: Oblique Mercator, Spherical Earth Model

```
SetProjectionSystem[
  MakeProjectionSystem[ObliqueMercator, CentralPoints → {{45 °, 0 °}, {0 °, -90 °}}]];
{{e, n}, k} = ProjectBLH[DMS[30, 0, 0, S], DMS[120, 0, 0, E], IncludeScaleFactor → True];
{e, n, k} // P
{n, e} = InverseProjection[e, n];
{n // PPDMSϕ, e // PPDMSλ}
```

```
{-2.420133502, -0.04740264561, 1.001123716}
```

```
{30°0'0"S, 120°0'0"E}
```

The oblique Mercator projection is used for relatively small landforms, meaning those smaller than continents, whose shape is generally longer is one direction than the other, and whose principle axes do not generally fall perpendicular or parallel to North-South. For example, the "pan handle" of Alaska and Madagascar are often mapped with the oblique Mercator projection. For this example, we'll map New Zealand.

These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

Begin by extracting the points for New Zealand and finding their bounding box and extent.

```
Off[General::spell1];
NZ = Select[exDataPoints, #[[1]] == "New Zealand" &];
Length[NZ]
{{nzϕMin, nzλMin}, {nzϕMax, nzλMax}} =
 {{Min[NZ[[All, 2]]] °, Min[NZ[[All, 3]]] °}, {Max[NZ[[All, 2]]] °, Max[NZ[[All, 3]]] °}}
δϕ = nzϕMax - nzϕMin
δλ = nzλMax - nzλMin
On[General::spell1];
```

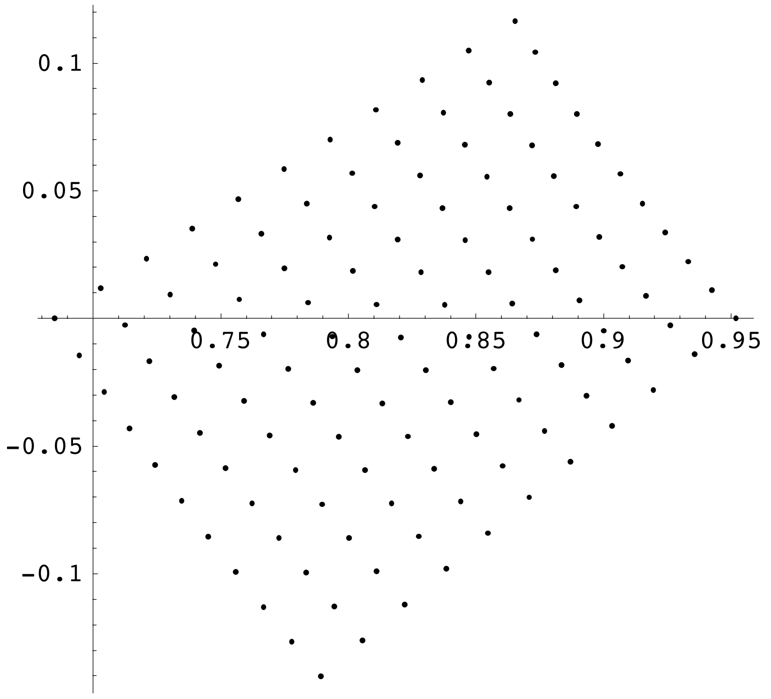
```
1401

{{-0.81472, 2.90528}, {-0.600568, 3.11646}}

0.214152

0.211185
```

Oblique Mercator projections require a pair of BLH coordinates to define the central axis along which the projection is defined.

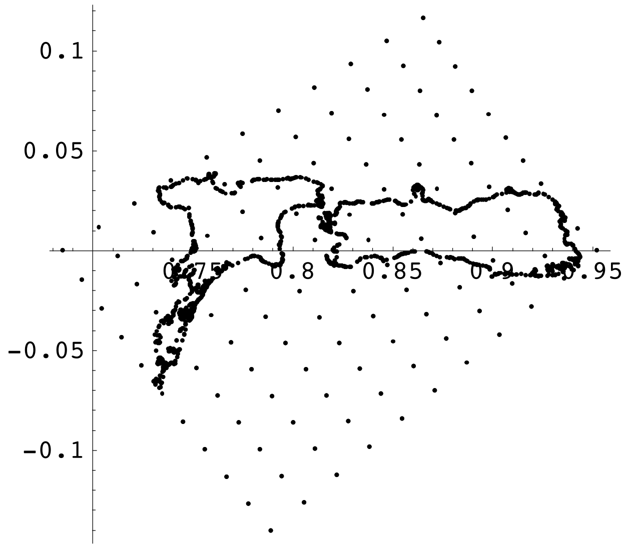
```
SetProjectionSystem[MakeProjectionSystem[ObliqueMercator,
  EarthModel → Spherical, CentralPoints → {{nzϕMin, nzλMin}, {nzϕMax, nzλMax}},
  GrdForm → False, IncludeScaleFactor → False]]; $projectionSystem // TableForm;
```

Here's the Oblique Mercator graticule and New Zealand

```
gratPlot = ListPlot[
   Select[
    Flatten[
     Table[ProjectBLH[ϕ , λ ], {ϕ, nzϕMin, nzϕMax, δϕ / 10}, {λ, nzλMin, nzλMax, δλ / 10}],
     1],
    #[[1]] ≠ ∞ && #[[2]] ≠ ∞ &], DisplayFunction → Identity];
Show[gratPlot, DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Show[
  gratPlot,
  ListPlot[
   ProjectBLH[#⟦2⟧ °, #⟦3⟧ °] & /@ NZ,
   DisplayFunction → Identity],
  gratPlot, DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



Oblique Mercator, Ellipsoidal

Snyder, p. 274-277: Oblique Mercator, Ellipsoidal Earth Model, Hotine Formulae

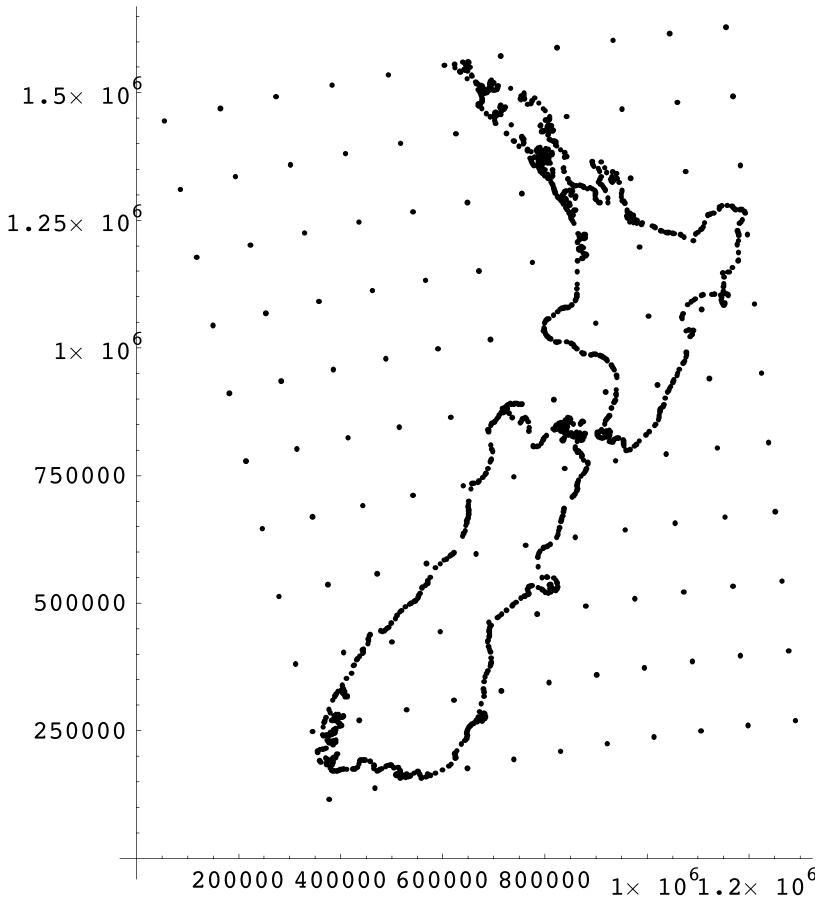
```
Seattle = MakeBLH[DMS[47, 30, 0, N], DMS[122, 18, 0, W]];
Miami = MakeBLH[DMS[25, 42, 0, N], DMS[80, 12, 0, W]];
NYC = MakeBLH[DMS[40, 48, 0, N], DMS[74, 00, 0, W]];
SetProjectionSystem[
  MakeProjectionSystem[
   ObliqueMercator,
   EarthModel → Clarke1866,
   CentralScaleFactor → 0.9996,
   CentralParallel → (40°),
   CentralPoints → {Seattle, Miami},
   OriginFalseEasting → 4000000.0,
   OriginFalseNorthing → 500000.0,
   Rectify → True
  ]
 ];
{{e, n}, k} = ProjectBLH[NYC, IncludeScaleFactor → True];
{e, n, k} // P
{ϕ, λ} = InverseProjection[e, n];
{ϕ, λ} /° // P
{ϕ // PPDMSϕ, λ // PPDMSλ}
```

```
{963436.0922, 4369142.81, 1.030755397}

{40.8, -74.}

{40°48'0"N, 74°0'0"W}
```

Now set up to project New Zealand. The ellipsoidal version of this projection allows the projected planimetric coordinates to be *rectified*, which rotates the resulting map so that north is at the top.

```
SetProjectionSystem[
 MakeProjectionSystem[
  ObliqueMercator,
  EarthModel → Clarke1866,
  CentralPoints → {{nzϕMax, nzλMax}, {nzϕMin, nzλMin}}, GrdForm → False,
  IncludeScaleFactor → False,
  OriginFalseEasting → 3000000,
  OriginFalseNorthing → 5500000,
  Rectify → True
 ]]; $projectionSystem // TableForm;
```

```
gratPlot = ListPlot[
  Select[
   Flatten[
    Table[ProjectBLH[ϕ , λ ], {ϕ, nzϕMin, nzϕMax, δϕ / 10}, {λ, nzλMin, nzλMax, δλ / 10}],
    1],
   #[[1]] ≠ ∞ && #[[2]] ≠ ∞ &], DisplayFunction → Identity]; Show[
 gratPlot,
 ListPlot[
  ProjectBLH[#[[2]] °, #[[3]] °] & /@ NZ,
  DisplayFunction → Identity],
 gratPlot, DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



Lambert Conformal Conic, Spherical

The following examples are taken from Snyder, John P. (1987) Map Projections -- A Working Manual. U.S. Geological Survey Professional Paper 1395, p. 295: Lambert Conformal Conic, Spherical Earth Model

---

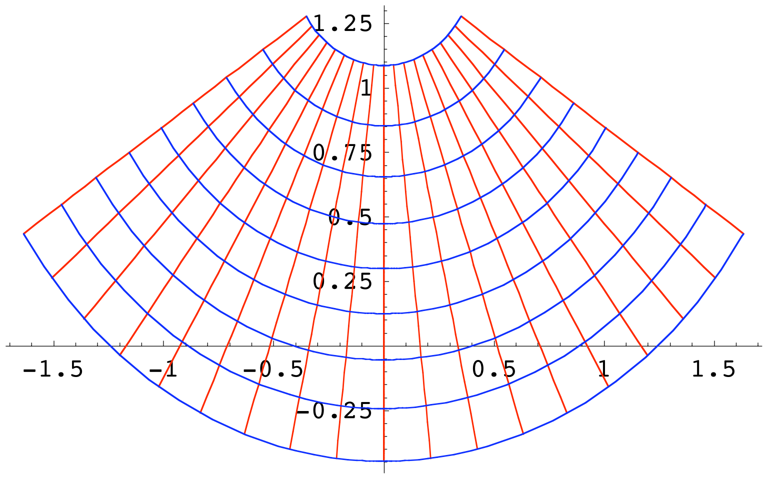
These examples construct a graticule for the Lambert conformal conic (LCC) projection and then project physical and political boundary points over the graticule to see how the LCC projection depicts the Earth. This section uses a spherical Earth model of unit radius. Notice that, unlike Mercator, LCC requires specifying the location of standards (either one or two)

```
Needs["GeometricalGeodesy`"]

SetProjectionSystem[
  MakeProjectionSystem[
   LambertConformalConic,
   Standards → {33 °, 45 °},
   CentralParallel → (23 °),
   CentralMeridian → (-96 °)]];
{{e, n}, k} = ProjectBLH[35. °, -75. °, IncludeScaleFactor → True];
{e, n, k} // P
InverseProjection[e, n] // PPDMS
Remove[e, n, k]
```

```
{0.2966784599, 0.2462112293, 0.997003959}

{35°0'0", -75°0'0"}
```

Here's the LCC graticule. Build two lists of graphics. The first holds meridians (red); the second holds parallels (blue)

```
maxLat = 80; maxLon = -6;
minLat = 0; minLon = -186;
meridians =
  (ParametricPlot[ProjectBLH[ϕ °, # °], {ϕ, minLat, maxLat}, PlotStyle → {RGBColor[1, 0, 0]},
      DisplayFunction → Identity] & /@ Range[minLon, maxLon, 10.]);
parallels = (ParametricPlot[ProjectBLH[# °, λ °], {λ, minLon, maxLon}, PlotStyle →
        {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[minLat, maxLat, 10]);
Show[meridians, parallels, DisplayFunction → $DisplayFunction, PlotRange → All];
```

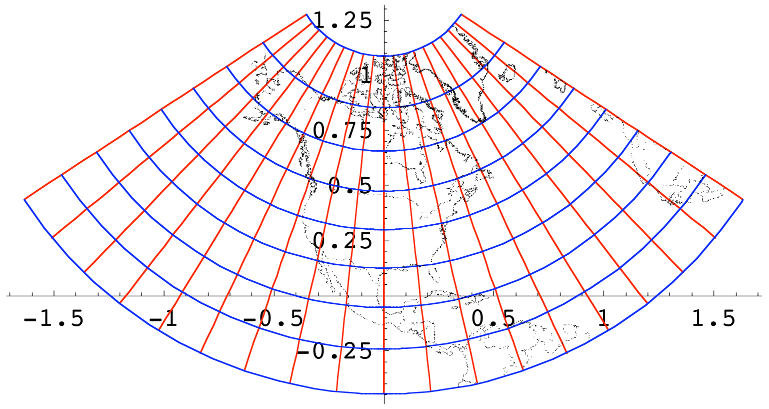


These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

Project and display them.

```
Show[
  meridians,
  parallels,
  ListPlot[(ProjectBLH[#[[2]] °, #[[3]] °]) & /@
    Select[thinDataPoints, #[[2]] < maxLat && #[[2]] > minLat && #[[3]] < maxLon && #[[3]] > minLon &],
   AspectRatio → Automatic, DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Remove[meridians, parallels, maxLat, maxLon, minLat, minLon];
```

Lambert Conformal Conic, Ellipsoidal

These examples construct a graticule for the Lambert conformal conic (LCC) projection and then project physical and political boundary points over the graticule to see how the LCC projection depicts the Earth. This section uses a spherical Earth model of unit radius. Notice that, unlike Mercator, LCC requires specifying the location of standards (either one or two)

```
Needs["GeometricalGeodesy`"]
```

The following examples are taken from Snyder, John P. (1987) Map Projections -- A Working Manual. U.S. Geological Survey Professional Paper 1395, p. 296: Lambert Conformal Conic, Ellipsoidal Earth Model

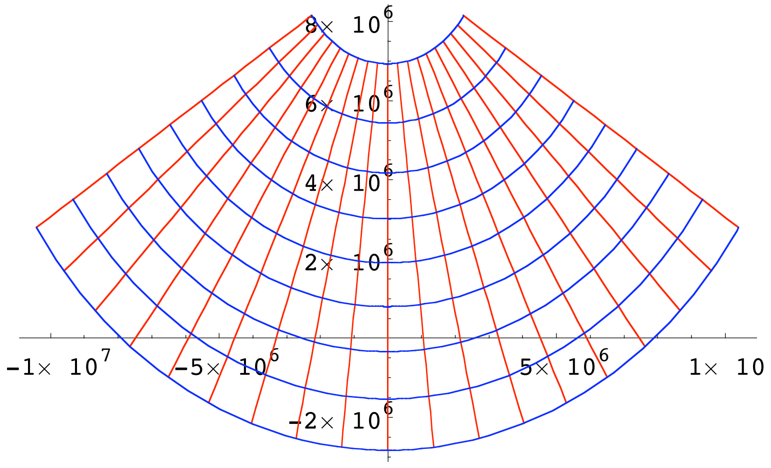
```
SetProjectionSystem[
  MakeProjectionSystem[
   LambertConformalConic,
   Standards → {33 °, 45 °},
   CentralParallel → (23 °),
   CentralMeridian → (-96 °),
   EarthModel → Clarke1866]];
{{e, n}, k} = ProjectBLH[35. °, -75. °, IncludeScaleFactor → True];
{e, n, k} // P
InverseProjection[e, n] // PPDMS
Remove[e, n, k]

{1894410.898, 1564649.478, 0.9970171418}

{35°0'0", -75°0'0"}
```

These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

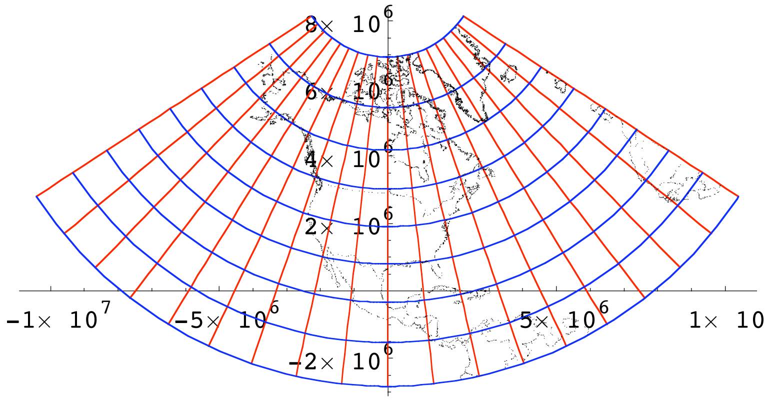
Here's the LCC graticule. Build two lists of graphics. The first holds meridians (red); the second holds parallels (blue)

```
maxLat = 80; maxLon = -6;
minLat = 0; minLon = -186;
meridians =
  (ParametricPlot[ProjectBLH[ϕ°, #°], {ϕ, minLat, maxLat}, PlotStyle → {RGBColor[1, 0, 0]},
     DisplayFunction → Identity] & /@ Range[minLon, maxLon, 10]);
parallels = (ParametricPlot[ProjectBLH[#°, λ°], {λ, minLon, maxLon}, PlotStyle →
       {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[minLat, maxLat, 10]);
Show[meridians, parallels, DisplayFunction → $DisplayFunction, PlotRange → All];
```



Project and display them.

```
Show[
  meridians,
  parallels,
  ListPlot[(ProjectBLH[#[[2]]°, #[[3]]°]) & /@
    Select[thinDataPoints, #[[2]] < maxLat && #[[2]] > minLat && #[[3]] < maxLon && #[[3]] > minLon &],
   AspectRatio → Automatic, DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Remove[meridians, parallels, maxLat, maxLon, minLat, minLon];
```

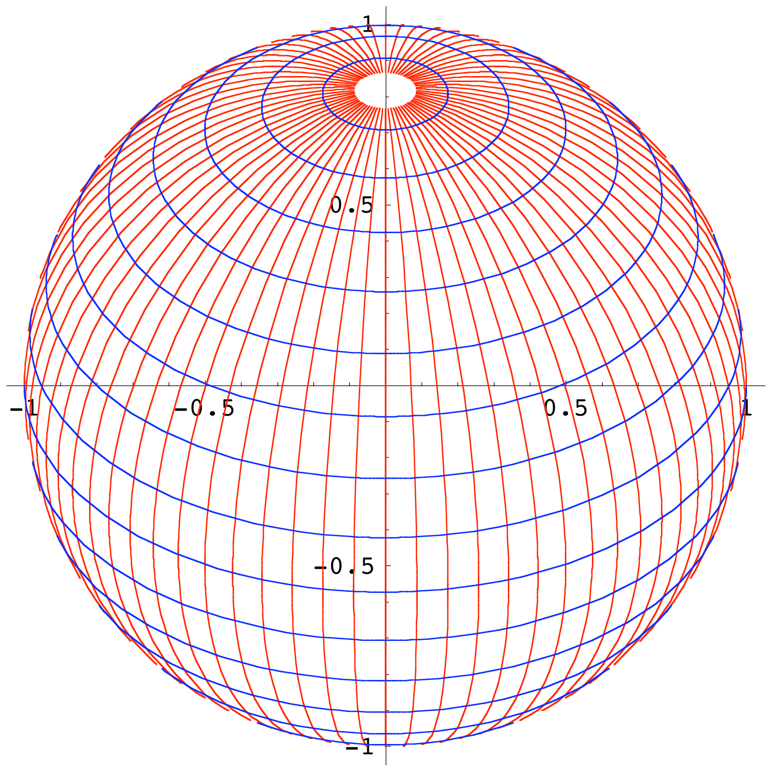
Orthographic

These examples construct a graticule for the orthographic projection and then project physical and political boundary points over the graticule to see how the orthographic projection depicts the Earth. This section uses a spherical Earth model of unit radius. The default projection uses a spherical Earth model with unity radius. The central meridian and parallel are both zero degrees by default.

```
Needs["GeometricalGeodesy`"]

SetProjectionSystem[MakeProjectionSystem[Orthographic, CentralParallel → 35.°]];
```

Here's the graticule. Build two lists of graphics. The first holds meridians (red); the second holds parallels (blue)

```
Off[ParametricPlot::pptr];
minLat = -90; maxLat = 85;
meridians = (ParametricPlot[ProjectBLH[ϕ°, #°], {ϕ, minLat, maxLat},
     PlotStyle → {RGBColor[1, 0, 0]}, DisplayFunction → Identity] & /@ Range[0, 360, 5]);
parallels = (ParametricPlot[ProjectBLH[#°, λ°], {λ, 0, 360}, PlotStyle →
       {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[minLat, maxLat, 10]);
Show[meridians, parallels, DisplayFunction → $DisplayFunction,
  PlotRange → All, AspectRatio → Automatic];
On[ParametricPlot::pptr];
```

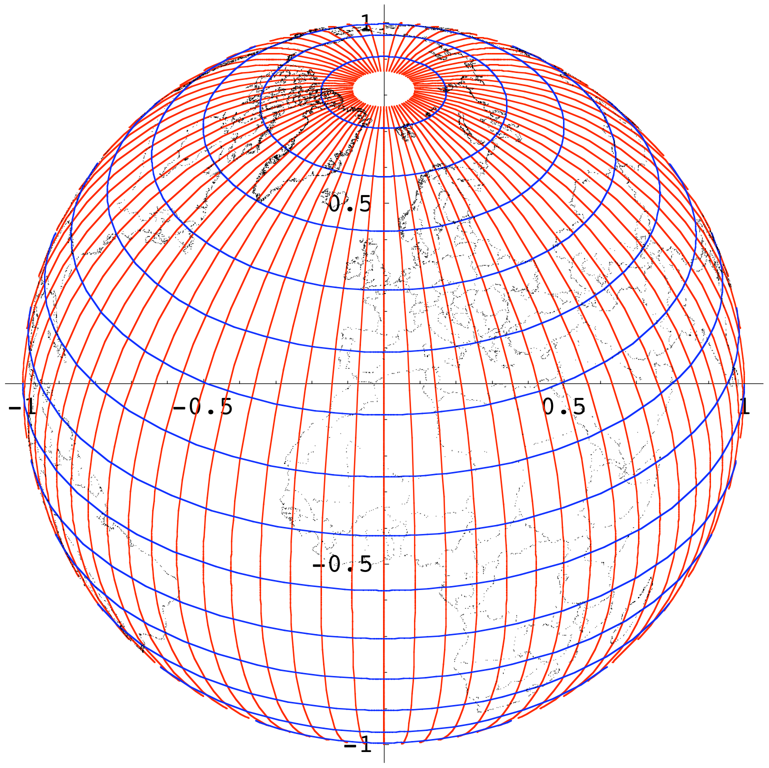


These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

Project them and display them. All the points on the far side return a value of `Null`

```
Show[
  meridians,
  parallels,
  ListPlot[
   Select[ProjectBLH[#⟦2⟧°, #⟦3⟧°] & /@ thinDataPoints, # =!= Null &],
   AspectRatio → Automatic, DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Remove[meridians, parallels, maxLat, maxLon];
```

Stereographic, Spherical

These examples construct a graticule for the Mercator projection and then project physical and political boundary points over the graticule to see how the Mercator projection depicts the Earth. This section uses a spherical Earth model of unit radius.

```
Needs["GeometricalGeodesy`"]
```

This is the example from Snyder, p. 312.

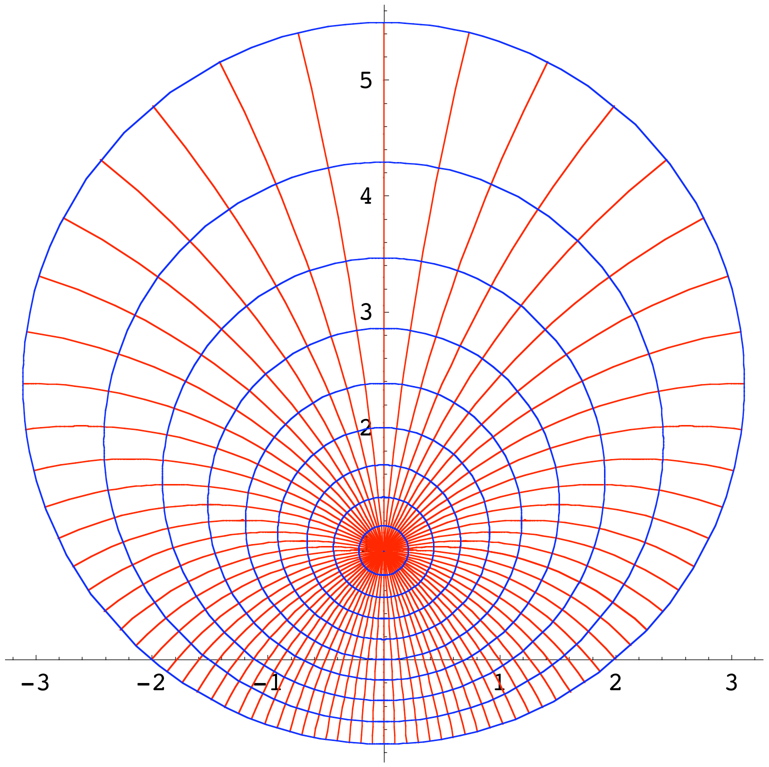
```
SetProjectionSystem[MakeProjectionSystem[Stereographic,
  EarthModel → Spherical, SphericalRadius → 1, CentralParallel → (40 °),
  CentralMeridian → (-100 °), CentralScaleFactor → 1.0, GrdForm → False]];
{{e, n}, k} = ProjectBLH[30 °, -75 °, IncludeScaleFactor → True];
{e, n, k} // P
InverseProjection[e, n] // PPDMS
Remove[e, n, k]
```

```
{0.3807223928, -0.1263801845, 1.040230373}
```

```
{30°0'0", -75°0'0"}
```

Here's the Mercator graticule. Build two lists of graphics. The first holds meridians (red); the second holds parallels (blue)

```
minLat = 0; maxLat = 90; meridians =
 (ParametricPlot[ProjectBLH[ϕ °, # °], {ϕ, minLat, maxLat}, PlotStyle → {RGBColor[1, 0, 0]},
     DisplayFunction → Identity] & /@ Range[0, 360, 5]);
parallels = (ParametricPlot[ProjectBLH[# °, λ °], {λ, 0, 360}, PlotStyle →
        {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[minLat, maxLat, 10]);
Show[meridians, parallels, DisplayFunction → $DisplayFunction,
  PlotRange → All, AspectRatio → Automatic];
```

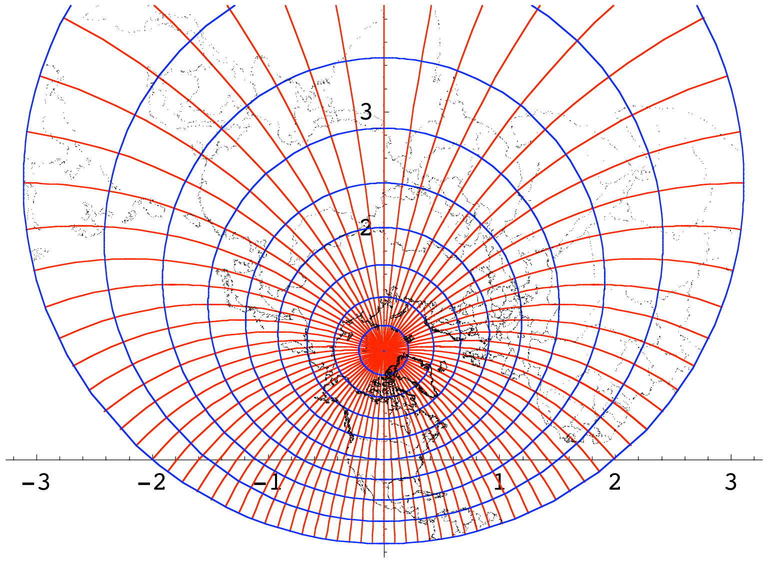


These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

Project them and display them.

```
Show[
  meridians,
  parallels,
  ListPlot[
   (ProjectBLH[#⟦2⟧ °, #⟦3⟧ °]) & /@ Select[thinDataPoints, #⟦2⟧ < maxLat && #⟦2⟧ > minLat &],
   AspectRatio → Automatic, DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Clear[minLat, maxLat, meridians, parallels]
```

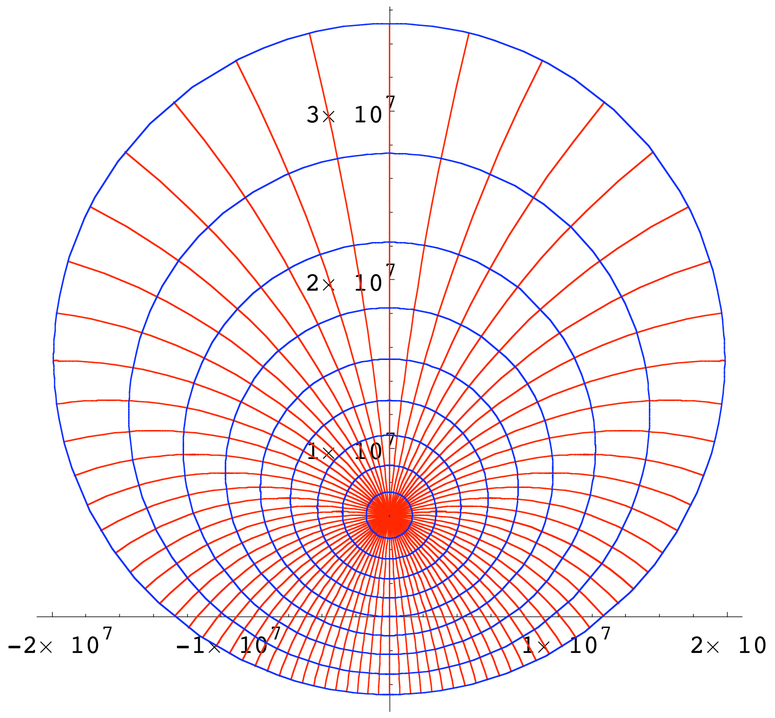
Stereographic, Ellipsoidal

This section repeats what was done for the spherical model but using an ellipsoidal Earth model, GRS 80.

```
SetProjectionSystem[MakeProjectionSystem[Stereographic,
  EarthModel → GRS80, SphericalRadius → 1, CentralParallel → (40 °),
  CentralMeridian → (-100 °), CentralScaleFactor → 1.0, GrdForm → False]];
{{e, n}, k} = ProjectBLH[30 °, -75 °, IncludeScaleFactor → True];
{e, n, k} // P
InverseProjection[e, n] // PPDMS
Remove[e, n, k]
```

```
{2429270.159, -803215.8508, 1.039774973}

{30°0'0", -75°0'0"}
```

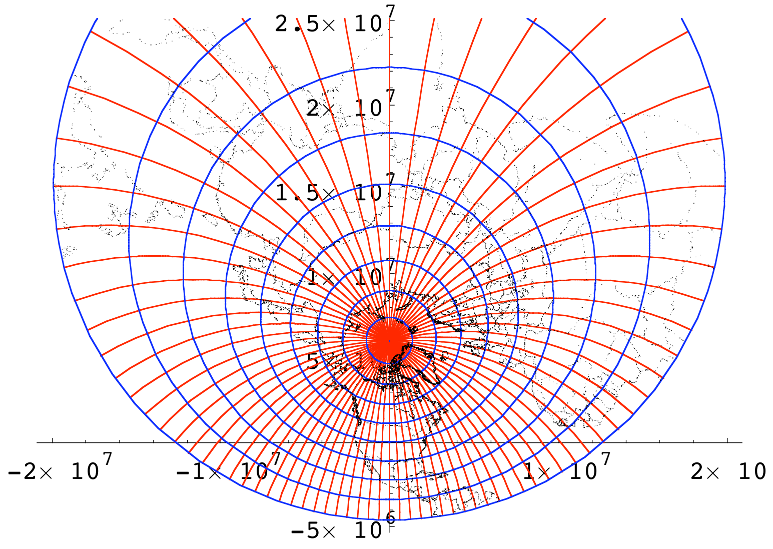
```
minLat = 0; maxLat = 90; meridians =
 (ParametricPlot[ProjectBLH[ϕ °, # °], {ϕ, minLat, maxLat}, PlotStyle → {RGBColor[1, 0, 0]},
     DisplayFunction → Identity] & /@ Range[0, 360, 5]);
parallels = (ParametricPlot[ProjectBLH[# °, λ °], {λ, 0, 360}, PlotStyle →
       {RGBColor[0, 0, 1]}, DisplayFunction → Identity] & /@ Range[minLat, maxLat, 10]);
Show[meridians, parallels, DisplayFunction → $DisplayFunction,
  PlotRange → All, AspectRatio → Automatic];
```



These projection examples display the world's countries to illustrate the image they create. The following code reads in the world points. If they've already been read in, you do not need to do it again.

```
If[Head[exDataPoints] === Symbol,
 exDataPoints =
  Import[ToFileName[$GeometricalGeodesyDataDirectory, "worldpoints.txt"], "CSV"];
 thinDataPoints = Select[exDataPoints, Random[Integer, {1, 5}] == 1 &];
]
```

```
Show[
  meridians,
  parallels, ListPlot[
   (ProjectBLH[#⟦2⟧ °, #⟦3⟧ °]) & /@ Select[thinDataPoints, #⟦2⟧ < maxLat && #⟦2⟧ > minLat &],
   AspectRatio → Automatic, DisplayFunction → Identity, PlotStyle → PointSize[0.001]],
  DisplayFunction → $DisplayFunction, AspectRatio → Automatic];
```



```
Clear[maxLat, minLat, meridians, parallels]
```