# Introduction to the "geosphere" package (Version 1.5-18)

Robert J. Hijmans

November 13, 2022

## 1 Introduction

This vignette describes the R package 'geosphere'. The package implements spherical trigonometry functions for geographic applications. Many of the functions have applications in navigation, but others are more general, or have no relation to navigation at all.

There is a number of functions to compute distance and direction (= bearing, azimuth, course) along great circles (= shortest distance on a sphere, or "as the crow flies") and along rhumb lines (lines of constant bearing).

There are also functions that compute distances on a spheroid.

Other functions include the computation of the location of an object at a given direction and distance; and the area, perimeter, and centroid of a spherical polygon.

Geographic locations must be specified in longitude and latitude (and in that order!) in degrees (i.e., NOT in radians). Degrees are (obviously) in decimal notation. Thus 12 degrees, 10 minutes, 30 seconds = $12 + 10/60 + 30/3600 = 12.175$ degrees. The southern and western hemispheres have a negative sign.

The default unit of distance is meter; but this can be adjusted by supplying a different radius 'r' to functions. Directions are expressed in degrees (N = 0 and 360, E = 90, S = 180, and W = 270 degrees). If arguments of functions that take several arguments (e.g. points, bearings, radius of the earth), do not have the same length (for vectors) or number of rows (for matrices) the shorter arguments are re-cycled.

Many functions in this package are based on formulae provided by Ed Williams (http://www.edwilliams.org/ftp/avsig/avform.txt, and partly on javascript implementations of these formulae by Chris Veness (http://www.movable-type.co.uk/scripts/latlong.html )

Most geodesic computations (for a spheroid rather than a sphere) use the GeographicLib by C.F.F. Karney (http://geographiclib.sourceforge.net/.
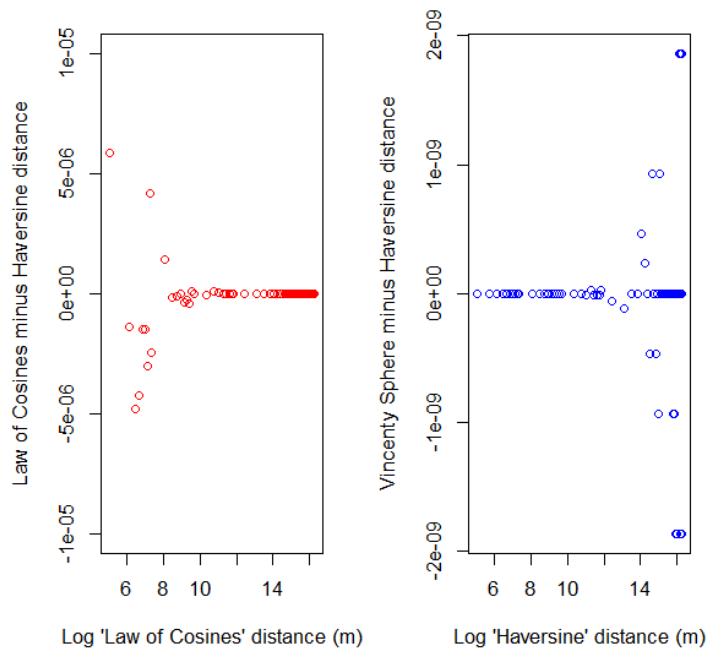
## 2 Great circle distance

There are four different functions to compute distance between two points. These are, in order of increasing complexity of the algorithm, the 'Spherical law of cosines', 'Haversine' (Sinnott, 1984), 'Vincenty Sphere' and 'Vincenty Ellipsoid' (Vincenty, 1975) methods. The first three assume the earth to be a sphere, while the 'Vincenty Ellipsoid' assumes it is an ellipsoid (which is closer to the truth).

The results from the first three functions are identical for practical purposes. The Haversine ('half-versed-sine') formula was published by R.W. Sinnott in 1984, although it has been known for much longer. At that time computational precision was lower than today (15 digits precision). With current precision, the spherical law of cosines formula appears to give equally good results down to very small distances. If you want greater accuracy, you could use the distVincentyEllipsoid method.
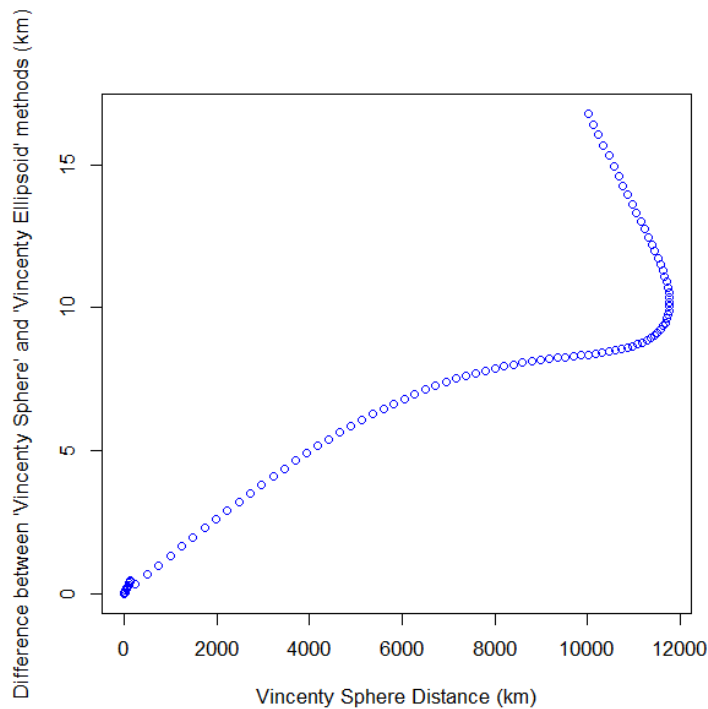
Below the differences between the three spherical methods are illustrated. At very short distances, there are small differences between the 'law of the Cosine' and the other two methods. There are even smaller differences between the 'Haversine' and 'Vincenty Sphere' methods at larger distances.

```
> library(geosphere)
> Lon <- c(1:9/1000, 1:9/100, 1:9/10, 1:90*2)
> Lat <- c(1:9/1000, 1:9/100, 1:9/10, 1:90)
> dcos <- distCosine(c(0,0), cbind(Lon, Lat))
> dhav <- distHaversine(c(0,0), cbind(Lon, Lat))
> dvsp <- distVincentySphere(c(0,0), cbind(Lon, Lat))
> par(mfrow=(c(1,2)))
> plot(log(dcos), dcos-dhav, col='red', ylim=c(-1e-05, 1e-05),
+            xlab="Log 'Law of Cosines' distance (m)",
+            ylab="Law of Cosines minus Haversine distance")
> plot(log(dhav), dhav-dvsp, col='blue',
+            xlab="Log 'Haversine' distance (m)",
+            ylab="Vincenty Sphere minus Haversine distance")
```

The difference with the 'Vincenty Ellipsoid' method is more pronounced. In the example below (using the default WGS83 ellipsoid), the difference is about 0.3

```
> dvse <- distVincentyEllipsoid(c(0,0), cbind(Lon, Lat))
> plot(dvsp/1000, (dvsp-dvse)/1000, col='blue', xlab='Vincenty Sphere Distance (km)',
+          ylab="Difference between 'Vincenty Sphere' and 'Vincenty Ellipsoid' methods (km)")
```

For the most precise distance computation use the 'distGeo' function.

# 3 Points on great circles

Points on a great circle are returned by the function 'greatCircle', using two points on the great circle to define it, and an additional argument to indicate how many points should be returned. You can also use greatCircleBearing, and provide starting points and bearing as arguments. gcIntermediate only returns points on the great circle that are on the track of shortest distance between the two points defining the great circle; and midPoint computes the point half-way between the two points. You can use onGreatCircle to test whether a point is on a great circle between two other points.

```
> LA <- c(-118.40, 33.95)
> NY <- c(-73.78,  40.63)
> data(wrld)
> plot(wrld, type='l')
> gc <- greatCircle(LA, NY)
> lines(gc, lwd=2, col='blue')
> gci <- gcIntermediate(LA, NY)
> lines(gci, lwd=4, col='green')
> points(rbind(LA, NY), col='red', pch=20, cex=2)
```
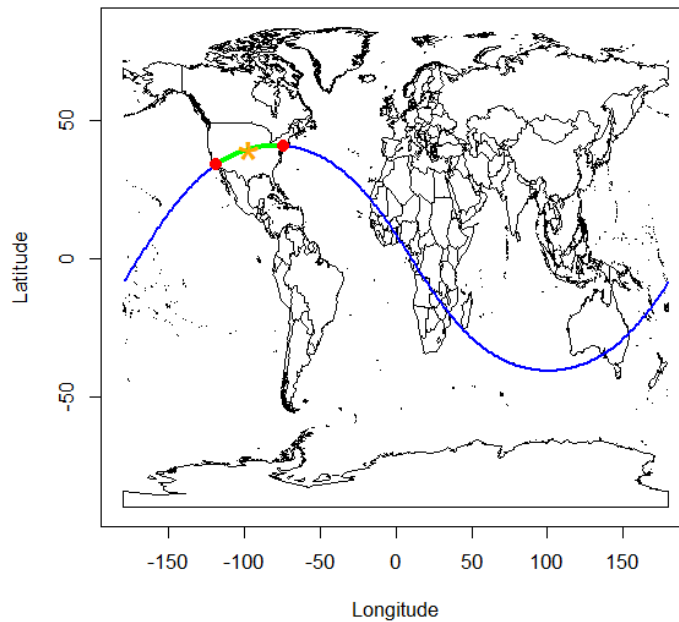
4

```
> mp <- midPoint(LA, NY)
> onGreatCircle(LA,NY, rbind(mp,c(0,0)))

[1] FALSE FALSE

> points(mp, pch='*', cex=3, col='orange')
> greatCircleBearing(LA, brng=270, n=10)

         lon        lat
 [1,] -144  31.210677
 [2,] -108  33.532879
 [3,]  -72  25.004950
 [4,]  -36   5.254184
 [5,]    0 -17.620746
 [6,]   36 -31.210677
 [7,]   72 -33.532879
 [8,]  108 -25.004950
 [9,]  144  -5.254184
[10,]  180  17.620746
```
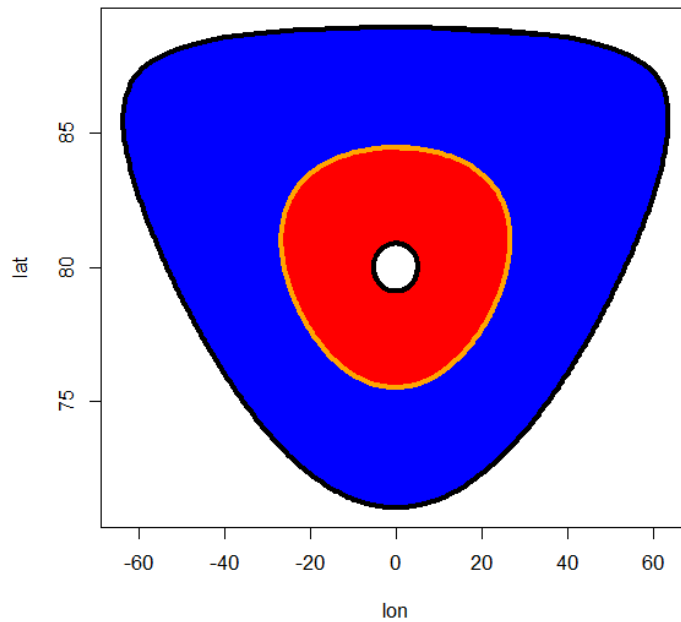
# 4 Point at distance and bearing

Function destPoint returns the location of point given a point of origin, and a distance and bearing. Its perhaps obvious use in georeferencing locations of distant sitings. It can also be used to make circular polygons (with a fixed radius, but in longitude/latitude coordinates)

```
> destPoint(LA, b=65, d=100000)

            lon       lat
[1,] -117.4152 34.32706

> circle=destPoint(c(0,80), b=1:365, d=1000000)
> circle2=destPoint(c(0,80), b=1:365, d=500000)
> circle3=destPoint(c(0,80), b=1:365, d=100000)
> plot(circle, type='l')
> polygon(circle, col='blue', border='black', lwd=4)
> polygon(circle2, col='red', lwd=4, border='orange')
> polygon(circle3, col='white', lwd=4, border='black')
```
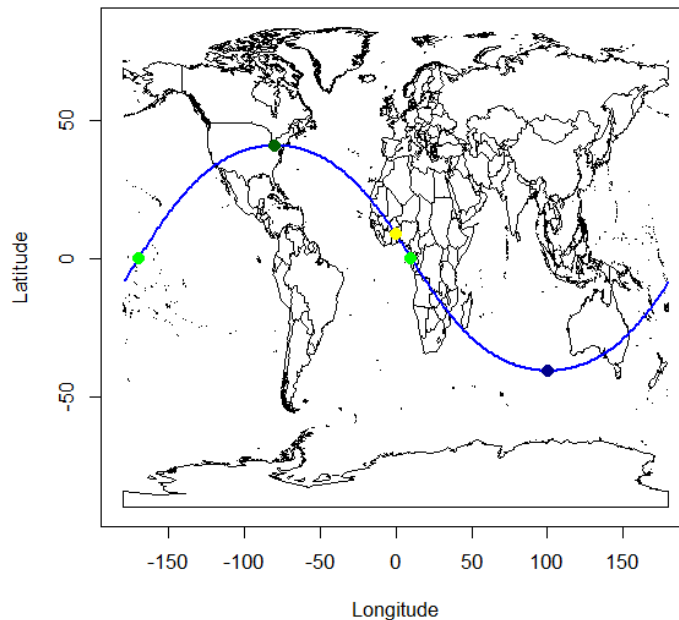
# 5 Maximum latitude on a great circle

You can use the functions illustrated below to find out what the maximum latitude is that a great circle will reach; at what latitude it crosses a specified longitude; or at what longitude it crosses a specified latitude. From the map below it appears that Clairaut's formula, used in gcMaxLat is not very accurate. Through optimization with function greatCircle, a more accurate value was found. The southern-most point is the antipode (a point at the opposite end of the world) of the northern-most point.

```
> ml <- gcMaxLat(LA, NY)
> lat0 <- gcLat(LA, NY, lon=0)
> lon0 <- gcLon(LA, NY, lat=0)
> plot(wrld, type='l')
> lines(gc, lwd=2, col='blue')
> points(ml, col='red', pch=20, cex=2)
> points(cbind(0, lat0), pch=20, cex=2, col='yellow')
> points(t(rbind(lon0, 0)), pch=20, cex=2, col='green' )
> f <- function(lon){gcLat(LA, NY, lon)}
> opt <- optimize(f, interval=c(-180, 180), maximum=TRUE)
> points(opt$maximum, opt$objective, pch=20, cex=2, col='dark green' )
> anti <- antipode(c(opt$maximum, opt$objective))
> points(anti, pch=20, cex=2, col='dark blue' )
```

# 6 Great circle intersections

Points of intersection of two great circles can be computed in two ways. We use a second great circle that connects San Francisco with Amsterdam. We first compute where they cross by defining the great circles using two points on it (gcIntersect). After that, we compute the same points using a start point and initial bearing (gcIntersectBearing). The two points where the great circles cross are antipodes. Antipodes are connected with an infinite number of great circles.

```
> SF <- c(-122.44, 37.74)
> AM <- c(4.75, 52.31)
> gc2 <- greatCircle(AM, SF)
> plot(wrld, type='l')
> lines(gc, lwd=2, col='blue')
> lines(gc2, lwd=2, col='green')
> int <- gcIntersect(LA, NY, SF, AM)
> int

          lon1      lat1      lon2      lat2
[1,] 52.62562 -30.15099 -127.3744 30.15099

> antipodal(int[,1:2], int[,3:4])

[1] TRUE

> points(rbind(int[,1:2], int[,3:4]), col='red', pch=20, cex=2)
> bearing1 <- bearing(LA, NY)
> bearing2 <- bearing(SF, AM)
> bearing1

[1] 65.93893

> bearing2

[1] 29.75753

> gcIntersectBearing(LA, bearing1, SF, bearing2)

          lon       lat       lon       lat
[1,] 52.63076 -30.16041 -127.3692 30.16041
```
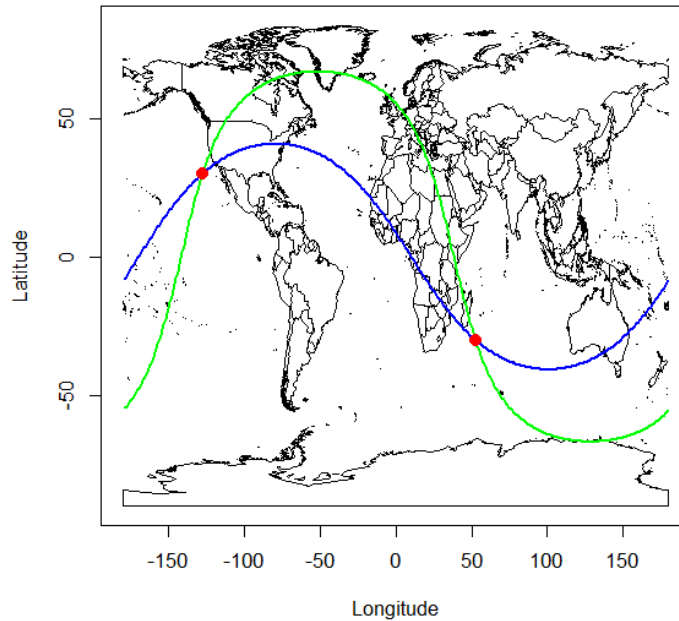
8

## 7 Triangulation

Below is triangulation example. We have three locations (NY, LA, MS) and three directions (281, 60, 195) towards a target. Because we are on a sphere, there are two (antipodal) results. We only show one here (by only using int[,1:2]). We compute the centroid from the polygon defined with the three points. To accurately draw a spherical polygon, we can use makePoly. This function inserts intermediate points along the paths between the vertices provided (default is one point every 10 km).

```
> MS <- c(-93.26, 44.98)
> gc1 <- greatCircleBearing(NY, 281)
> gc2 <- greatCircleBearing(MS, 195)
> gc3 <- greatCircleBearing(LA, 55)
> plot(wrld, type='l', xlim=c(-125, -70), ylim=c(20, 60))
> lines(gc1, col='green')
> lines(gc2, col='blue')
> lines(gc3, col='red')
> int <- gcIntersectBearing(rbind(NY, NY, MS),
+            c(281, 281, 195), rbind(MS, LA, LA), c(195, 55, 55))
> int
```

9

```
             lon      lat      lon      lat
[1,]  -94.40975 41.77229 85.59025 -41.77229
[2,] -102.91692 41.15816 77.08308 -41.15816
[3,]  -93.63298 43.97765 86.36702 -43.97765

> distm(rbind(int[,1:2], int[,3:4]))

             [,1]        [,2]        [,3]        [,4]        [,5]
[1,]          0.0    713665.0    253077.7 20003931.5 19308947.3
[2,]     713665.0         0.0    823532.8 19308947.3 20003931.5
[3,]     253077.7    823532.8         0.0 19751854.8 19195862.0
[4,] 20003931.5 19308947.3 19751854.8         0.0    713665.0
[5,] 19308947.3 20003931.5 19195862.0    713665.0         0.0
[6,] 19751854.8 19195862.0 20003931.5    253077.7    823532.8
             [,6]
[1,] 19751854.8
[2,] 19195862.0
[3,] 20003931.5
[4,]    253077.7
[5,]    823532.8
[6,]         0.0

> int <- int[,1:2]
> points(int)
> poly <- rbind(int, int[1,])
> centr <- centroid(poly)
> poly2 <- makePoly(int)
> polygon(poly2, col='yellow')
> points(centr, pch='*', col='dark red', cex=2)
```
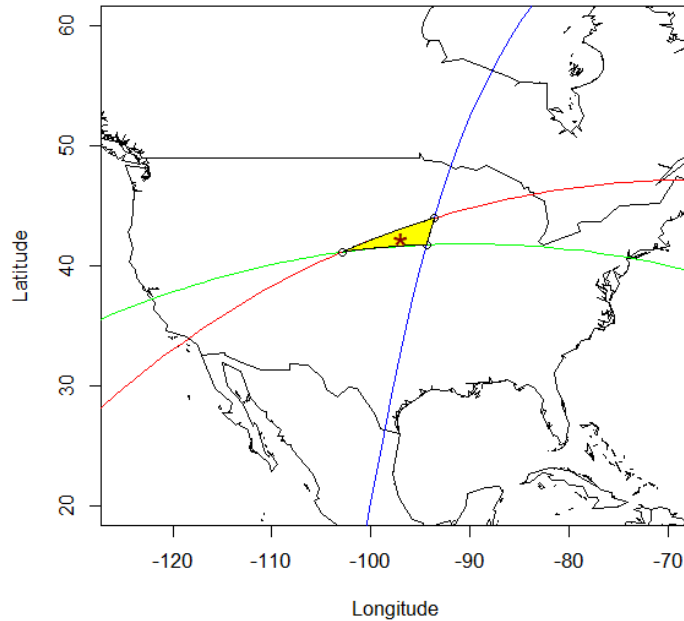
## 8 Bearing

Below we first compute the distance and bearing from Los Angeles (LA) to New York (NY). These are then used to compute the point from LA at that distance in that (initial) bearing (direction). Bearing changes continuously when traveling along a Great Circle. The final bearing, when approaching NY, is also given.

```
> d <- distCosine(LA, NY)
> d

[1] 3977614

> b <- bearing(LA, NY)
> b

[1] 65.93893

> destPoint(LA, b, d)

          lon      lat
[1,] -73.83063 40.63262
```

```
> NY

[1] -73.78  40.63

> finalBearing(LA, NY)

[1] 93.90995
```

# 9 Getting off-track

What if we went off-course and were flying over Minneapolis (MS)? The closest point on the planned route (p) can be computed with the alongTrackDistance and destPoint functions. The distance from 'p' to MS can be computed with the dist2gc (distance to great circle, or cross-track distance) function. The light green line represents the along-track distance, and the dark green line represents the cross-track distance.
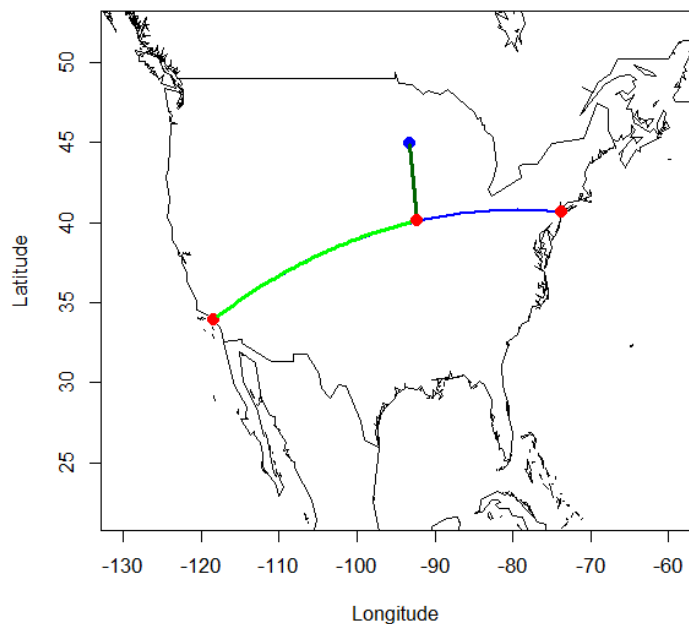
```
> atd <- alongTrackDistance(LA, NY, MS)
> p <- destPoint(LA, b, atd)
> plot(wrld, type='l', xlim=c(-130,-60), ylim=c(22,52))
> lines(gci, col='blue', lwd=2)
> points(rbind(LA, NY), col='red', pch=20, cex=2)
> points(MS[1], MS[2], pch=20, col='blue', cex=2)
> lines(gcIntermediate(LA, p), col='green', lwd=3)
> lines(gcIntermediate(MS, p), col='dark green', lwd=3)
> points(p, pch=20, col='red', cex=2)
> dist2gc(LA, NY, MS)

[1] 547448.8

> distCosine(p, MS)

[1] 548738.9
```
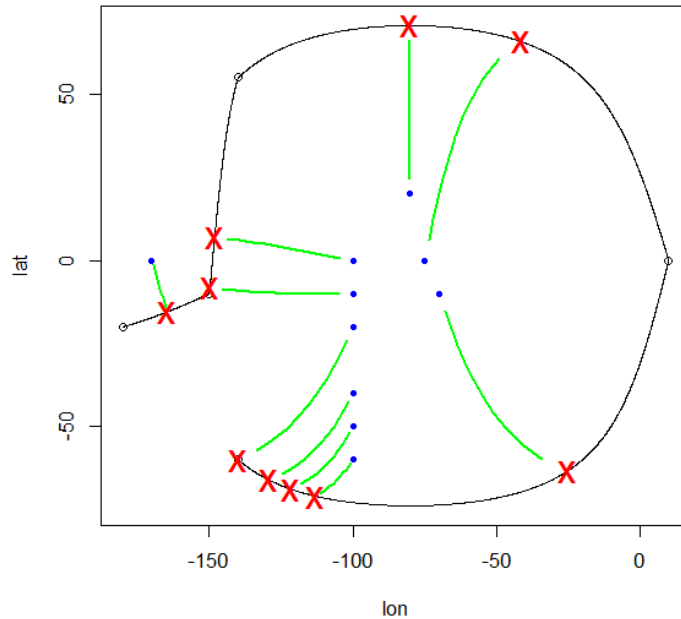
# 10 Distance to a polyline

The two function describe above are used in the dist2Line function that computes the shortest distance between a set of points and a set of spherical poly-lines (or polygons).

```
> line <- rbind(c(-180,-20), c(-150,-10), c(-140,55), c(10, 0), c(-140,-60))
> pnts <- rbind(c(-170,0), c(-75,0), c(-70,-10), c(-80,20), c(-100,-50),
+               c(-100,-60), c(-100,-40), c(-100,-20), c(-100,-10), c(-100,0))
> d = dist2Line(pnts, line)
> plot( makeLine(line), type='l')
> points(line)
> points(pnts, col='blue', pch=20)
> points(d[,2], d[,3], col='red', pch='x', cex=2)
> for (i in 1:nrow(d)) lines(gcIntermediate(pnts[i,], d[i,2:3], 10), lwd=2, col='green')
```

13

## 11 Rhumb lines

Rhumb (from the Spanish word for course, 'rumbo') lines are straight lines on a Mercator projection map (and at most latitudes pretty straight on an equirectangular projection (=unprojected lon/lat) map). They were used in navigation because it is easier to follow a constant compass bearing than to continually adjust direction as is needed to follow a great circle, even though rhumb lines are normally longer than great-circle (orthodrome) routes. Most rhumb lines will gradually spiral towards one of the poles.
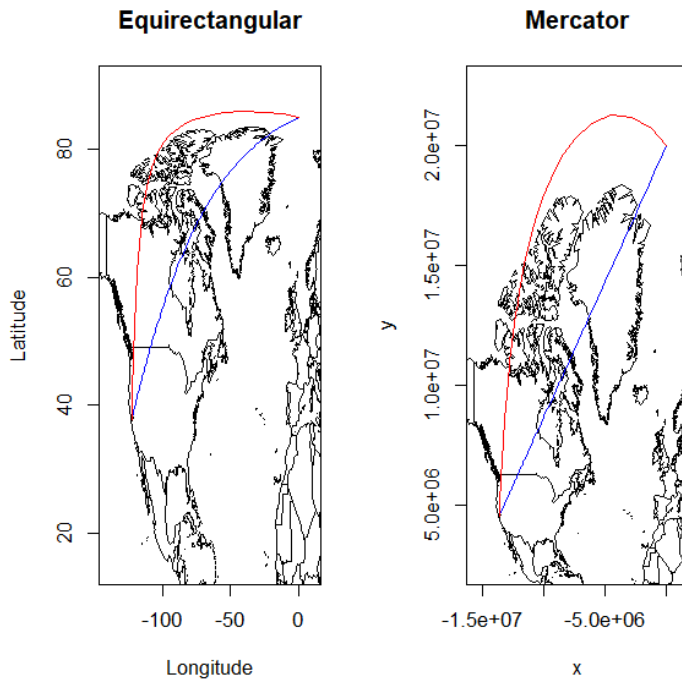
```
> NP <- c(0, 85)
> bearing(SF, NP)

[1] 5.15824

> b <- bearingRhumb(SF, NP)
> b

[1] 41.45714

> dc <- distCosine(SF, NP)
> dr <- distRhumb(SF, NP)
> dc / dr
```

```
[1] 0.8730767

> pr <- destPointRhumb(SF, b, d=round(dr/100) * 1:100)
> pc <- rbind(SF, gcIntermediate(SF, NP), NP)
> par(mfrow=c(1,2))
> data(wrld)
> plot(wrld, type='l', xlim=c(-140,10), ylim=c(15,90), main='Equirectangular')
> lines(pr, col='blue')
> lines(pc, col='red')
> data(merc)
> plot(merc, type='l', xlim=c(-15584729, 1113195),
+                      ylim=c(2500000, 22500000), main='Mercator')
> lines(mercator(pr), col='blue')
> lines(mercator(pc), col='red')
```



## 12 Characterizing polygons

The package has functions to compute the area, perimeter, centroid, and 'span' of a spherical polygon. One approach to compute these measures is to project the polygons first. Here we directly compute them based on spherical coordinates (longitude / latitude), except for centroid, which is computed by

projecting the data to the Mercator projection (and inversely projecting the result). The function makePoly inserts additional vertices into a spherical polygon such that it can be plotted (perhaps after first projecting it) more correctly in a plane. Vertices are inserted, where necessary, at a specified distance. The function is only beneficial for polygons with large inter-vertex distances (in terms of longitude), particularly at high latitudes.

```
> pol <- rbind(c(-120,-20), c(-80,5), c(0, -20), c(-40,-60), c(-120,-20))
> areaPolygon(pol)

[1] 4.903757e+13

> perimeter(pol)

[1] 485

> centroid(pol)

          lon      lat
[1,] -57.54653 -28.96994

> span(pol, fun=max)

         lon     lat
[1,] 9040759 7187331

> nicepoly = makePoly(pol)
> plot(pol, xlab='longitude', ylab='latitude', cex=2, lwd=3, xlim=c(-140, 0))
> lines(wrld, col='grey')
> lines(pol, col='red', lwd=2)
> lines(nicepoly, col='blue', lwd=2)
> points(centroid(pol), pch='*', cex=3, col='dark green')
> text(centroid(pol)-c(0,2.5), 'centroid')
> legend(-140, -48, c('planar','spherical'), lty=1, lwd=2,
+                   col=c('red', 'blue'), title='polygon type')
```
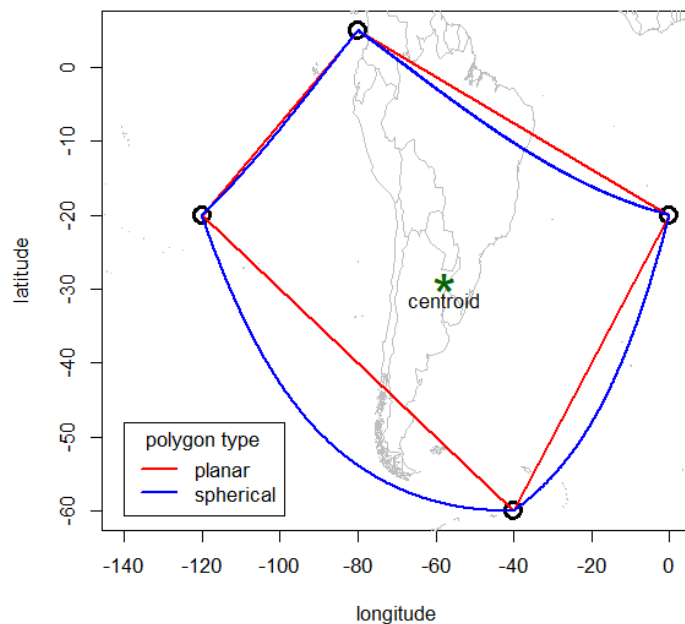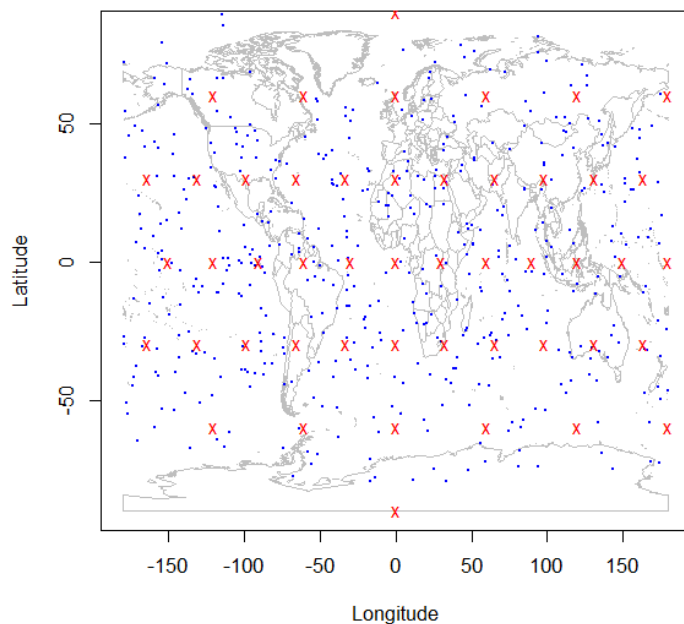
# 13  Sampling

Random or regular sampling of longitude/latitude values on the globe needs to consider that the globe is spherical. That is, if you would take random points for latitude between -90 and 90 and for longitude between -180 and 180, the density of points would be higher near the poles than near the equator. In contrast, functions 'randomCoordinates' and 'randomCoordinates' return samples that are spatially balanced.

```
> plot(wrld, type='l', col='grey')
> a = randomCoordinates(500)
> points(a, col='blue', pch=20, cex=0.5)
> b = regularCoordinates(3)
> points(b, col='red', pch='x')
```

# 14 Daylength

You can compute daylenght according to the formula by Forsythe et al. (1995). For any day of the year (an integer between 1 and 365; or a 'Date' object.
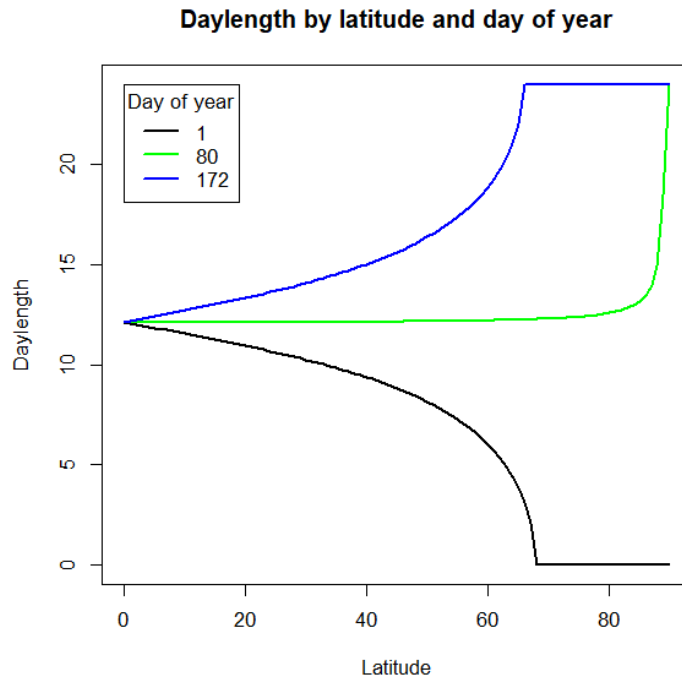
```
> as.Date(80, origin='2009-12-31')

[1] "2010-03-21"

> as.Date(172, origin='2009-12-31')

[1] "2010-06-21"

> plot(0:90, daylength(lat=0:90, doy=1), ylim=c(0,24), type='l', xlab='Latitude',
+         ylab='Daylength', main='Daylength by latitude and day of year', lwd=2)
> lines(0:90, daylength(lat=0:90, doy=80), col='green', lwd=2)
> lines(0:90, daylength(lat=0:90, doy=172), col='blue', lwd=2)
> legend(0,24, c('1','80','172'), lty=1, lwd=2, col=c('black', 'green', 'blue'),
+         title='Day of year')
```

**Daylength by latitude and day of year**



# 15 References

Forsythe, W.C., E.J. Rykiel Jr., R.S. Stahl, H. Wu and R.M. Schoolfield, 1995.
A model comparison for daylength as a function of latitude and day of
the year. Ecological Modeling 80:87-95.

Sinnott, R.W, 1984. Virtues of the Haversine. Sky and Telescope 68(2): 159

Vincenty, T. 1975. Direct and inverse solutions of geodesics on the ellipsoid
with application of nested equations. Survey Review 23(176): 88-93.
Available here: `http://www.ngs.noaa.gov/PUBS_LIB/inverse.pdf`