Package 'proxistat'

December 27, 2019

Title Create a Proximity Score in Each Census Unit, Based on Distances to Specified Points

Version 1.0.0

Date 2020-01-01

Description This package has functions helping to calculate distances between geographic points, such as the distances between all points, distances to all points within some maximum distance, distance to nearest single point, etc. It also can create a proximity score for each spatial unit like a Census block group, to quantify the distance-weighted count of nearby points. This proximity score can be used in environmental justice (EJ) analysis, for example. This package uses the sp package for the basic distance calculation. Key functions include get.nearest() to find the one topoint nearest each frompoint, get.distances() to find distances quickly within an optional search radius, and get.distances.all() to find distances from all frompoints to alltopoints. The function proxistat() creates a proximity score that quantifies, for each spatial unit like a Census block group, how many topoints are nearby and how close they are.
For any imported/suggested packages not on CRAN, see http://ejanalysis.github.io

Imports sp,
analyze.stuff,
Hmisc,
data.table

Suggests matrixStats,
UScensus2010blocks

URL http://ejanalysis.github.io, https://github.com/ejanalysis/proxistat, http://www.ejanalysis.com/

BugReports https://github.com/ejanalysis/proxistat/issues

Depends R (>= 3.1.0)

RoxygenNote 7.0.2

License MIT + file LICENSE

Repository GitHub

Author info@ejanalysis.com

Maintainer info@ejanalysis.com <info@ejanalysis.com>

NeedsCompilation no

LazyData true

Encoding UTF-8

R topics documented:

`proxistat-package` *Find Distances between lat lon Points or Create Proximity Scores.*

Description

This package has functions helping to calculate distances between points, such as the distances between all points, distances to all points within some maximum distance, distance to nearest single point, etc. It also can create a proximity score for each spatial unit like a Census block group, to quantify the distance-weighted count of nearby points.

Details

This package has functions helping to calculate distances between geographic points, such as the distances between all points, distances to all points within some maximum distance, distance to nearest single point, etc. It also can create a proximity score for each spatial unit like a Census block group, to quantify the distance-weighted count of nearby points. This proximity score can be used in environmental justice (EJ) analysis, for example.

This package relies on the **sp** package for the actual calculation of distance.

A vector of points can be specified using a data.frame of two columns, "lat" and "lon" which specify latitude and longitude in decimal degrees. It returns the distances from one or more `frompoints` to one or more `topoints`.

Key functions include

- `get.nearest` to find the one among `topoints` nearest each `frompoints`
- `get.distances` to find distances quickly within an optional search radius
- `proxistat` to create a proximity score that quantifies, for each spatial unit like a Census block group, how many `topoints` are nearby and how close they are
- `convert` to convert units (miles, km)

Author(s)

info@ejanalysis.com<info@ejanalysis.com>

References

http://ejanalysis.github.io
http://www.ejanalysis.com/
sp package documentation for basic distance function.
Some discussion of this type of proximity indicator is available in the EJSCREEN mapping tool documentation:
U.S. Environmental Protection Agency (2015). EJSCREEN Technical Documentation. http://www.epa.gov/ejscreen
http://en.wikipedia.org/wiki/Longitude and http://en.wikipedia.org/wiki/Decimal_degrees

See Also

sp, US block points dataset: http://ejanalysis.github.io/UScensus2010blocks/, `deltalon.per.km`, `deltalon.per.km`, `meters.per.degree.lat`, `meters.per.degree.lon`

Examples

```
test.from <- structure(list(fromlat = c(38.9567309094, 38.9507043428),
  fromlon = c(-77.0896572305, -77.0896199948)),
  .Names = c("lat","lon"), row.names = c("6054762", "6054764"), class = "data.frame")
test.to <- structure(list(tolat = c(38.9575019287, 38.9507043428, 38.9514152435),
  tolon = c(-77.0892818598, -77.0896199948, -77.0972395245)),
  .Names = c("lat","lon"), class = "data.frame", row.names = c("6054762", "6054763", "6054764"))
  setseed(999)
t100=testpoints(100)
t10=testpoints(10)
t3=testpoints(3)

get.distances(
  test.from[1:2,], test.to[1:3, ], radius=0.7, units='km', return.rownums=TRUE, return.latlons=TRUE
)
get.nearest(test.from, test.to)

get.distances(  t3, t10, units='km', return.crosstab=TRUE)
get.distances(  t3, t10, units='km')
```

```
get.distances(  t3, t10, radius=300, units='km')
proxistat(t3, t10, radius = 300, units = "km", area = c(100001:100003))
proxistat(  t3, t10, radius=300, units='km')
1/get.nearest(   t3, t10, radius=300, units='km')
```

`bg.pts` *Block group internal points and areas (square meters) from Census Bureau for 2010 geographies*

Description

This data set provides a data.frame with 220740 block groups, with 5 variables.

Usage

```
data('bg.pts')
```

Format

A data.frame of 220740 block groups, with 5 variables.

- [1,] "FIPS (e.g., '010950302024')"
- [2,] "aland numeric, land area in square meters"
- [3,] "awater numeric, water area in square meters"
- [4,] "lat numeric, latitude in decimal degrees"
- [5,] "lon numeric, longitude in decimal degrees"

Source

Derived from Census Bureau, obtained early 2015.

See Also

UScensus2010blocks and **acs** and the Census 2010 packages http://lakshmi.calit2.uci.edu/census2000/ and http://www.jstatsoft.org/v37/i06

compile.dbfs *Read a series of dbf files and join compile them as a single data.frame*

Description

Can help with a series of downloaded Census data files, such as one file per State. Reads each using `read.dbf` and combines them all as a data.frame. They must all have the same columns, and each file just provides additional rows of data.

Usage

```
compile.dbfs(filepaths)
```

Arguments

`filepaths`	Required charater vector with full paths including file names of dbf files to be read.

Value

Returns a data.frame with as many columns as each dbf file, and as many rows as there are in all the dbf files read in.

See Also

`read.dbf`

convert *Convert units of distance or area*

Description

`convert` converts distance/length or area from specified units to other specified units.

Usage

```
convert(x, from = "km", towhat = "mi")
```

Arguments

`x`	A number or vector of numbers to be converted.
`from`	A string specifying original units of input parameter. Default is 'km' which is kilometers. Note all must be in the same units. Units can be specified as any of the following: c('millimeter', 'millimeters', 'centimeter', 'centimeters', 'meter', 'meters', 'kilometer', 'kilometers', "mm", "cm", "m", "km", "sqmm", "sqcm", "sqm", "sqkm", "mm2", "cm2", "m2", "km2", 'inch', 'inches', 'foot', 'feet', 'yard', 'yards', 'mile', 'miles', "in", "ft", "yd", "mi", "sqin", "sqft", "sqyd", "sqmi", "in2", 'ft2', 'yd2', 'mi2') Note that m2 is for square meters not square miles.
`towhat`	A strings specifying new units to convert to. Default is 'mi' which is miles.

Details

This function takes a number, or vector of numbers, representing distance/length or area in one type of specified units, such as miles, and returns the corresponding number(s) converted to some other units, such as kilometers. Units can be specified in various ways. All inputs must be in the same units. All outputs must be in a single set of units as well.

NOTE: For some purposes, Census Bureau does this: "The ANSI standard for converting square kilometers into square miles was used (1 square mile = 2.58998811 square kilometers)." (see https://www.census.gov/geo/reference/state-area.html) but the conversions in this function use 2.5899881034 not 2.58998811 sqkm/sqmi. The difference is only 6.6 per billion (roughly 1 in 152 million), which is less than one tenth of a square kilometer out the entire USA.

Value

Returns a number or vector of numbers then length of the input x, with each element corresponding to an input element converted to new units.

See Also

`get.distances` which allows you to specify a search radius and get distances only within that radius, and related functions.

Examples

```
convert(1, 'mi', 'km')
convert(c(1e6, 1), 'sqm', 'sqkm')
```

---

countiesall *Counties information from U.S. Census Bureau from 2015*

Description

This data set provides names and FIPS codes for U.S. Counties and County equivalents.

Format

A data.frame of 3234 U.S. Counties or County equivalents, with 5 variables.

> str(countiesall)
'data.frame': 3235 obs. of 5 variables:

- $ ST : chr "AL" "AL" "AL" "AL" ...
- $ countyname : chr "Autauga County" "Baldwin County" "Barbour County" "Bibb County" ...
- $ FIPS.COUNTY: chr "01001" "01003" "01005" "01007" ...
- $ statename : chr "Alabama" "Alabama" "Alabama" "Alabama" ...
- $ fullname : chr "Autauga County, Alabama" "Baldwin County, Alabama" "Barbour County, Alabama" "Bibb County, Alabama" ...

```
> head(countiesall)

ST countyname FIPS.COUNTY statename fullname
1 AL Autauga County 01001 Alabama Autauga County,Alabama
2 AL Baldwin County 01003 Alabama Baldwin County,Alabama
3 AL Barbour County 01005 Alabama Barbour County,Alabama
4 AL Bibb County 01007 Alabama Bibb County,Alabama
5 AL Blount County 01009 Alabama Blount County,Alabama
6 AL Bullock County 01011 Alabama Bullock County,Alabama
```

Source

Derived from Census Bureau, obtained early 2015.

See Also

get.county.info from the **ejanalysis** package (http://ejanalysis.github.io/ejanalysis/), and the **UScensus2010county** package and the **acs** package and the Census 2010 packages http://lakshmi.calit2.uci.edu/census2000/ and http://www.jstatsoft.org/v37/i06

---

countypointmap *Simple map of color-coded block group centroids in county*

Description

very simple mapping

Usage

```
countypointmap(
  query,
  vartext,
  varname,
  breaks = 5,
  coloring,
  asp = c(1, 1),
  pch = 1,
  ...
)
```

Arguments

query	See get.county.info
vartext	required, passed to pointmap.
varname	required Name of column in data.frame bg, which must be in memory already.
breaks	Default is 5, passed to cut
coloring	Default is colors[1:(length(breaks) + 1)] (unless breaks is 1, and then it is colors[1])
asp	Default is c(1,1), passed to pointmap.
pch	Default is 1, passed to pointmap.
...	Additional parameters to pass to pointmap.

Value

Just draws a map

See Also

pointmap

`deg2rad` *Convert degrees to radians*

Description

From http://www.r-bloggers.com/great-circle-distance-calculations-in-r/

Usage

```
deg2rad(deg)
```

Arguments

deg	Degrees as decimal degrees, required numeric vector

Value

Radians, as numeric vector same length as input

`deltalon.per.km` *Convert kilometers (East-West) to degrees longitude.*

Description

`deltalon.per.km` returns change in decimal degrees longitude per kilometer (East-West), given latitude (Northern Hemisphere).

Usage

```
deltalon.per.km(lat)
```

Arguments

lat	The decimal degrees of latitude of the Northern Hemisphere location(s) of interest, as number or vector of numbers.

Details

This function returns the change in decimal degrees longitude per kilometer moving East-West, at a given latitude (N. Hemisphere). This is an approximation and is less accurate further from the given latitude. The degrees of longitude per km traveled East-West is larger closer to the poles, and smallest at the equator. See http://en.wikipedia.org/wiki/Longitude and http://en.wikipedia.org/wiki/Decimal_degrees Not using meters.per.degree.lon the results would be roughly (1 / ((pi/180) * 6378.137 * cos(lat * 0.01745329))) where 0.01745329 is pi/180 to convert to radians. see http://en.wikipedia.org/wiki/Decimal_degrees The equator is divided into 360 degrees of longitude, so each degree at the equator represents 111,319.9 metres or approximately 111.32 km. As one gets further from the equator, one degree of longitude must be multiplied by the cosine of the latitude, decreasing distance per degree lon, approaching zero at the pole. For most northern point of USA http://en.wikipedia.org/wiki/Extreme_points_of_the_United_States, where one finds maximum degrees lon per mile, latitude is just under 72, and cos of 72 degrees= cos(72 * pi/180) = 0.309 , so miles/degrees is about 111.32 km * 0.309 = 34.4 km = 21.4 miles roughly.

Value

Returns decimal degrees longitude per km traveled E-W, as a number or vector of numbers the same length as the input lat.

See Also

meters.per.degree.lon for the inverse of this function (other than a factor of 1000), and get.distances which allows you to specify a search radius and get distances only within that radius, and related functions.

Examples

```
deltalon.per.km(0)
  # Roughly 111 km/degree moved East-West at the equator roughly, or around 21 miles.
deltalon.per.km(c(0,45,72))
```

---

gcd *Distance between two points by Haversine, Spherical Law of Cosines, or Vincenty inverse formula*

Description

Calculates the geodesic distance between two points (or multiple pairs of points) specified by degrees (DD) latitude/longitude using Haversine formula (hf), Spherical Law of Cosines (slc) and Vincenty inverse formula for ellipsoids (vif)
Taken from http://www.r-bloggers.com/great-circle-distance-calculations-in-r/
Note these are not the most accurate method for long distances (e.g., >1000 km) nor the fastest.

Usage

```
gcd(frompoints, topoints, dfunc = "hf", units = "km")
```

Arguments

frompoints	Required matrix or data.frame of 2 columns, named lat and lon, with latitude and longitude(s) in degrees, one row per point.
topoints	Required matrix or data.frame of 2 columns, named lat and lon, with latitude and longitude(s) in degrees, one row per point.
dfunc	Character string "hf" for Haversine by default, or "slc" for Spherical Law of Cosines (and may add "vif" for Vincenty inverse formula but that is not implemented here). For "sp" algorithm (**sp**), see get.distances
units	Optional character variable specifying 'km' or 'miles', 'km' by default.

Details

*** frompoints and topoints must have same number of rows, defining all pairs of points. Alternatively can define a single point while topoints defines a series of points, or vice versa. Taken from http://www.r-bloggers.com/great-circle-distance-calculations-in-r/ but use pmin instead of min to vectorize it to handle at least pairs, and parameters changed to keep lat/lon as a matrix or data.frame rather than 2 separate vectors.

Value

Distance in kilometers by default, or in miles if units='miles'

See Also

convert, gcd, get.distances, get.distances.all

---

gcd.hf *Distance based on Haversine formula*

Description

Calculates the geodesic distance between two points, or multiple pairs of points, specified by radian latitude/longitude.

Calculates the geodesic distance between two points (or multiple pairs of points) specified by radian latitude/longitude.

Usage

```
gcd.hf(long1, lat1, long2, lat2)

gcd.hf(long1, lat1, long2, lat2)
```

Arguments

long1	longitude(s) in radians, vector of one or more numbers
lat1	latitude(s) in radians, vector of one or more numbers
long2	longitude(s) in radians, vector of one or more numbers
lat2	latitude(s) in radians, vector of one or more numbers

Details

long1 and lat1 must be same length. long2 and lat1 must be same length. All four must be the same length, defining pairs of points. Alternatively long1 and lat1 can define a single point while long2 and lat2 define a series of points, or vice versa. Taken from http://www.r-bloggers.com/great-circle-distance-calculations-in-r/ but use pmin instead of min to vectorize it to handle at least pairs.

long1 and lat1 must be same length. long2 and lat1 must be same length. All four must be the same length, defining pairs of points. Alternatively long1 and lat1 can define a single point while long2 and lat2 define a series of points, or vice versa. Taken from http://www.r-bloggers.com/great-circle-distance-calculations-in-r/ but use pmin instead of min to vectorize it to handle at least pairs.

Value

Distance in kilometers

Distance in kilometers

See Also

convert, gcd, get.distances, get.distances.all

convert, gcd, get.distances, get.distances.all

---

gcd.slc *Distance based on spherical law of cosines*

Description

Calculates the geodesic distance between two points (or multiple pairs of points) specified by radian latitude/longitude.

Usage

```
gcd.slc(long1, lat1, long2, lat2)
```

Arguments

long1	longitude(s) in radians, vector of one or more numbers
lat1	latitude(s) in radians, vector of one or more numbers
long2	longitude(s) in radians, vector of one or more numbers
lat2	latitude(s) in radians, vector of one or more numbers

Details

long1 and lat1 must be same length. long2 and lat1 must be same length. All four must be the same length, defining pairs of points. Alternatively long1 and lat1 can define a single point while long2 and lat2 define a series of points, or vice versa. Taken from http://www.r-bloggers.com/great-circle-distance-calculations-in-r/ but use pmin instead of min to vectorize it to handle at least pairs.

Value

Distance in kilometers

See Also

convert, gcd, get.distances, get.distances.all

`get.bg.latlons` *Was used to get block group internal points directly from Census shapefiles*

Description

Download, unzip, read, and assemble lat lon of internal points for US Census block groups.

Usage

```
get.bg.latlons(
  myyear = 2014,
  mytempdir,
  mystatenums,
  overwrite = FALSE,
  silent = FALSE
)
```

Arguments

`myyear`	Year such as 2014 (default)
`mytempdir`	Optional. Default is TIGERTEMP created inside the current working directory. Character string of path to where temporary directory is or will be, for downloaded zip and dbf files.
`mystatenums`	Optional, default is all including PR/VI/DC, etc. Vector of strings of 2-character FIPS codes for states to get data for.
`overwrite`	Optional, FALSE by default (unlike in unzip), which means if zip file exists locally do not download and if contents exist do not unzip. BUT note this does not yet check to see if contents exist and zip does not, in which case could avoid downloading.
`silent`	Optional logical FALSE by default. If TRUE, print more results.

Details

Note this is obsolete if used to create data() containing the results: bg.pts = get.bg.latlons(mytempdir = getwd(), overwrite=FALSE, silent = TRUE) save(bg.pts, file='bg.pts') # and then this can be saved in the data folder of a package, then build pkg, then access via data('bg.pts')

Examples

```
#get.bg.latlons(mystatenums=c('09','10'))
```

`get.distances` *Find distances between nearby points, within specified radius.*

Description

WORK IN PROGRESS. Returns the distances from one set of points to nearby members of another set of points.

Usage

```
get.distances(
  frompoints,
  topoints,
  radius = 5200,
  units = "miles",
  ignore0 = FALSE,
  dfunc = "sp",
  as.df = FALSE,
  return.rownums = TRUE,
  return.latlons = FALSE,
  return.crosstab = FALSE,
  tailored.deltalon = FALSE
)
```

Arguments

`frompoints`	A matrix or data.frame with two cols, 'lat' and 'lon' with datum=WGS84 assumed.
`topoints`	A matrix or data.frame with two cols, 'lat' and 'lon' with datum=WGS84 assumed.
`radius`	A single number defining nearby, the maximum distance searched for or recorded. Default is max allowed... radius must be less than about 8,368 kilometers (5,200 miles, or the distance from Hawaii to Maine)
`units`	A string that is 'miles' by default, or 'km' for kilometers, specifying units for radius and distances returned.
`ignore0`	A logical, default is FALSE, specifying whether to ignore distances that are zero and report only nonzero distances. Useful if want distance to points other than self, where frompoints=topoints, for example. Ignored if return.crosstab = TRUE.
`dfunc`	Optional character element "hf" or "slc" to specify distance function Haversine or spherical law of cosines. If "sp" (default), it uses the **sp** package to find distances more accurately and more quickly.
`as.df`	Optional logical, default is TRUE
`return.rownums`	Logical value, TRUE by default. If TRUE, value returned also includes two extra columns: a col of index numbers starting at 1 specifying the frompoint and a similar col specifying the topoint.
`return.latlons`	Logical value, FALSE by default. If TRUE, value returned also includes four extra columns, showing fromlat, fromlon, tolat, tolon.

`return.crosstab`
: Logical value, FALSE by default. If TRUE, value returned is a matrix of the distances, with a row per frompoint and col per topoint. (Distances larger than max search radius are not provided, even in this format).

`tailored.deltalon`
: Logical value, FALSE by default, but ignored. Leftover from older get.distances function. Defined size of initially searched area as function of lat, for each frompoint, rather than initially searching a conservatively large box.

Details

This function returns a matrix or vector of distances, which are the distances from one set of points to the nearby members of another set of points. It searches within a circle (of radius = radius, defining what is considered nearby), to calculate distance (in miles or km) from each of frompoints to each of topoints that is within the specified radius. Points are specified using latitude and longitude in decimal degrees.

Uses `get.distances.all`. Relies on the **sp** package for the `spDists` and `SpatialPoints` functions.

Regarding distance calculation, also see `http://en.wikipedia.org/wiki/Vincenty%27s_formulae`, `http://williams.best.vwh.net/avform.htm#Dist`, `http://sourceforge.net/projects/geographiclib/`, and `http://www.r-bloggers.com/great-circle-distance-calculations-in-r/`.

Finding distance to all of the 11 million census blocks in usa within 5 km, for 100 points, can take a while. May want to look at js library like turf, or investigate using data.table to index and more quickly subset the (potentially 11 million Census blocks of) topoints (or pre-index that block point dataset and allow this function to accept a data.table as input).

Value

By default, returns a dataframe that has 3 columns: fromrow, torow, distance (where fromrow or torow is the row number of the corresponding input, starting at 1). Distance returned is in miles by default, but with option to set units='km' to get kilometers. See parameters for details on other formats that may be returned if specified.

See Also

`get.distances.all` which allows you to get distances between all points, `get.distances.prepaired` for finding distances when data are already formatted as pairs of points, `get.nearest` which finds the distance to the single nearest point within a specified search radius instead of all topoints, and `proxistat` which calculates a proximity score for each spatial unit based on distances to nearby points.

Examples

```
#
set.seed(999)
t1=testpoints(1)
t10=testpoints(10)
t100=testpoints(100,  minlat = 25, maxlat = 45, minlon = -100, maxlon = -60)
t1k=testpoints(1e3,   minlat = 25, maxlat = 45, minlon = -100, maxlon = -60)
t10k=testpoints(1e4)
t100k=testpoints(1e5)
```

```
t1m=testpoints(1e6)
#t10m=testpoints(1e7)
   test.from <- structure(list(fromlat = c(38.9567309094, 45),
     fromlon = c(-77.0896572305, -100)), .Names = c("lat", "lon"),
     row.names = c("1", "2"), class = "data.frame")

   test.to <- structure(list(tolat = c(38.9575019287, 38.9507043428, 45),
    tolon = c(-77.0892818598, -77.2, -90)),
    .Names = c("lat", "lon"), class = "data.frame",
    row.names = c("1", "2", "3"))

   #*** Can fail if radius=50 miles? ... Error in rbind() numbers of
   #  columns of arguments do not match !
  #big = get.distances(t100, t1k, radius=100, units='miles', return.latlons=TRUE, as.df=TRUE)
     head(big)
     #summary(big$d)
  big = get.distances(t100, t1k, radius=100, units='miles', return.latlons=TRUE, as.df=TRUE)
     head(big)
     summary(big$d)

   # see as map of many points
    plot(big$fromlon, big$fromlat,main='from black circles...
      closest is red, others nearby are green ')
    points(t1k$lon, t1k$lat, col='blue',pch='.')
    points(big$tolon, big$tolat, col='green')
   junk=as.data.frame( get.nearest(t100, t1k, return.latlons=TRUE) )
   points(junk$tolon, junk$tolat, col='red')
   # Draw lines from frompoint to nearest:
   with(junk,linesegments(fromlon, fromlat, tolon, tolat) )

    # more test cases
 length(get.distances(t10,t10,radius=4999,ignore0 = TRUE, units='km')$d)
 get.distances(t10,t10,radius=4999,ignore0 = TRUE, units='km')
get.distances(test.from[1,],test.to[1,],radius=3000,return.rownums=F,return.latlons=F)
get.distances(test.from[1,],test.to[1,],radius=3000,return.rownums=FALSE,return.latlons=TRUE)
get.distances(test.from[1,],test.to[1,],radius=3000,return.rownums=TRUE,return.latlons=FALSE)
get.distances(test.from[1,],test.to[1,],radius=3000,return.rownums=TRUE,return.latlons=TRUE)

get.distances(test.from[1,],test.to[1:3,],radius=3000,return.rownums=F,return.latlons=F)
get.distances(test.from[1,],test.to[1:3,],radius=3000,return.rownums=FALSE,return.latlons=TRUE)
get.distances(test.from[1,],test.to[1:3,],radius=3000,return.rownums=TRUE,return.latlons=FALSE)
get.distances(test.from[1,],test.to[1:3,],radius=3000,return.rownums=TRUE,return.latlons=TRUE)

get.distances(test.from[1:2,],test.to[1,],radius=3000,return.rownums=F,return.latlons=F)
get.distances(test.from[1:2,],test.to[1,],radius=3000,return.rownums=FALSE,return.latlons=TRUE)
get.distances(test.from[1:2,],test.to[1,],radius=3000,return.rownums=TRUE,return.latlons=FALSE)
get.distances(test.from[1:2,],test.to[1,],radius=3000,return.rownums=TRUE,return.latlons=TRUE)

get.distances(test.from[1:2,],test.to[1:3,],radius=3000,return.rownums=F,return.latlons=F)
get.distances(test.from[1:2,],test.to[1:3,],radius=3000,return.rownums=FALSE,return.latlons=T)
get.distances(test.from[1:2,],test.to[1:3,],radius=3000,return.rownums=TRUE,return.latlons=F)
get.distances(test.from[1:2,],test.to[1:3,],radius=3000,return.rownums=TRUE,return.latlons=TRUE)
get.distances(test.from[1:2,],test.to[1:3,], radius=0.7,return.rownums=TRUE,
  return.latlons=TRUE, units='km')
get.distances(test.from[1:2,],test.to[1:3,], radius=0.7,return.rownums=TRUE,
  return.latlons=TRUE, units='miles')
```

```
  # Warning messages:
  # Ignoring return.crosstab because radius was specified
get.distances(test.from[1,],test.to[1:3, ], return.crosstab=TRUE)
get.distances(test.from[1:2,],test.to[1, ], return.crosstab=TRUE)
get.distances(test.from[1:2,],test.to[1:3, ], return.crosstab=TRUE)
get.distances(test.from[1:2,],test.to[1:3, ], radius=0.7, return.crosstab=TRUE)
```

`get.distances.all` *Find all distances between two sets of points (based on lat/lon)*

Description

Returns all the distances from one set of geographic points to another set of points. Can return a matrix of distances (m x n points) or vector or data.frame with one row per pair. Lets you specify units and whether you need lat/lon etc, but essentially just a wrapper for the **sp** package for the `spDists` and `SpatialPoints` functions.

Usage

```
get.distances.all(
  frompoints,
  topoints,
  units = "miles",
  return.crosstab = FALSE,
  return.rownums = TRUE,
  return.latlons = TRUE,
  as.df = TRUE
)
```

Arguments

`frompoints`	A matrix or data.frame with two cols, 'lat' and 'lon' with datum=WGS84 assumed.
`topoints`	A matrix or data.frame with two cols, 'lat' and 'lon' with datum=WGS84 assumed.
`units`	A string that is 'miles' by default, or 'km' for kilometers, specifying units for distances returned.
`return.crosstab`	Logical value, FALSE by default. If TRUE, value returned is a matrix of the distances, with a row per frompoint and col per topoint.
`return.rownums`	Logical value, TRUE by default. If TRUE, value returned also includes two extra columns: a col of index numbers starting at 1 specifying the frompoint and a similar col specifying the topoint. If crosstab=TRUE, ignores return.rownums and return.latlons
`return.latlons`	Logical value, TRUE by default. If TRUE, value returned also includes four extra columns, showing fromlat, fromlon, tolat, tolon. If crosstab=TRUE, ignores return.rownums and return.latlons
`as.df`	Logical, default is TRUE, in which case returns a data.frame (unless vector), otherwise a matrix (unless vector).

Details

*** Probably slower than it needs to be partly by using data.frame instead of matrix class? Roughly 10-20 Just using get.distances.all is reasonably fast? (30-40 seconds for 100 million distances, but slow working with results so large), Sys.time(); x=get.distances.all(testpoints(1e5), testpoints(1000), return.crosstab=TRUE); Sys.time()
[1] "2015-03-10 18:59:08 EDT"
[1] "2015-03-10 18:59:31 EDT" 23 SECONDS for 100 million distances IF NO PROCESSING OTHER THAN CROSSTAB
Sys.time(); x=get.distances.all(testpoints(1e6), testpoints(100), return.crosstab=TRUE); Sys.time()
[1] "2015-03-10 21:54:11 EDT"
[1] "2015-03-10 21:54:34 EDT" 23 SECONDS for 100 million distances (1m x 100, or 100k x 1000)
Sys.time(); x=get.distances.all(testpoints(1e6), testpoints(300), return.crosstab=TRUE); Sys.time()
[1] "2015-03-10 21:56:11 EDT"
[1] "2015-03-10 21:57:18 EDT" 67 seconds for 300 million pairs.
plus 20 seconds or so for x[x>100] <- Inf
#' so 11m blocks to 1k points could take >40 minutes! (you would want to more quickly remove the ones outside some radius)
>3 minutes per 100 sites?
About 2.6 seconds per site for 11m blocks?

> Sys.time(); x=get.distances.all(testpoints(1e5), testpoints(1000), units='miles',return.rownums=TRUE); Sys.time()
[1] "2015-03-09 21:23:04 EDT"
[1] "2015-03-09 21:23:40 EDT" 36 SECONDS IF DATA.FRAME ETC. DONE TO FORMAT RESULTS AND GET ROWNUMS
> Sys.time(); x=get.distances.all(testpoints(1e5), testpoints(1000), units='miles',return.rownums=TRUE)$d; Sys.time()
[1] "2015-03-09 21:18:47 EDT"
[1] "2015-03-09 21:19:26 EDT" 49 SECONDS IF DATA.FRAME ETC. DONE TO FORMAT RESULTS AND GET ROWNUMS IN get.distances.all

Value

By default, returns a dataframe that has 3 columns: fromrow, torow, distance (where fromrow or torow is the row number of the corresponding input, starting at 1). If return.crosstab=FALSE, which is default, and return.rownums and/or return.latlons is TRUE, returns a row per from-to pair, and columns depending on parameters, sorted first cycling through all topoints for first frompoint, and so on. If return.crosstab=FALSE and return.rownums and return.latlons are FALSE, returns a vector of distances in same order as rows described above. If return.crosstab=TRUE, returns a matrix of distances, with one row per frompoint and one column per topoint.

See Also

`get.distances` which allows you to specify a search radius and get distances only within that radius which can be faster, `get.distances.prepaired` for finding distances when data are already formatted as pairs of points, `get.nearest` which finds the distance to the single nearest point within a specified search radius instead of all topoints, and `proxistat` which calculates a proximity score for each spatial unit based on distances to nearby points.

Examples

```
set.seed(999)
t1=testpoints(1)
t10=testpoints(10)
t100=testpoints(100, minlat=25,maxlat=48)
t1k=testpoints(1e3)
t10k=testpoints(1e4)
t100k=testpoints(1e5)
t1m=testpoints(1e6)
#t10m=testpoints(1e7)

get.distances.all(t1, t1)
get.distances.all(t1, t10[2, ,drop=FALSE])
x=get.distances.all(t10, t100[1:20 , ], units='km')
 plot(x$tolon, x$tolat,pch='.')
 points(x$fromlon, x$fromlat)
 with(x, linesegments(fromlon, fromlat, tolon, tolat ))
 with(x[x$d<500, ], linesegments(fromlon, fromlat, tolon, tolat ,col='red'))
x=get.distances.all(t10, t1k); head(x);summary(x$d)
x=get.distances.all(t10, t1k, units='km'); head(x);summary(x$d)
x=get.distances.all(t10, t1k, units='km'); head(x);summary(x$d)

## Not run:
require(UScensus2010blocks)
blocks <- get.blocks(fields=c('fips','lat','lon'),charfips = FALSE)


## End(Not run)

   test.from <- structure(list(fromlat = c(38.9567309094, 45),
     fromlon = c(-77.0896572305, -100)), .Names = c("lat", "lon"),
     row.names = c("1", "2"), class = "data.frame")

   test.to <- structure(list(tolat = c(38.9575019287, 38.9507043428, 45),
    tolon = c(-77.0892818598, -77.2, -90)),
    .Names = c("lat", "lon"), class = "data.frame",
    row.names = c("1", "2", "3"))

get.distances.all(test.from, test.to)
get.distances.all(test.from, test.to, return.crosstab=TRUE)
get.distances.all(test.from, test.to, return.rownums=FALSE)
get.distances.all(test.from, test.to, return.latlons=FALSE)
get.distances.all(test.from, test.to, return.latlons=FALSE, return.rownums=FALSE)

     # test cases

get.distances.all(test.from[1,],test.to[1,],return.rownums=F,return.latlons=F)
get.distances.all(test.from[1,],test.to[1,],return.rownums=FALSE,return.latlons=TRUE)
get.distances.all(test.from[1,],test.to[1,],return.rownums=TRUE,return.latlons=FALSE)
get.distances.all(test.from[1,],test.to[1,],return.rownums=TRUE,return.latlons=TRUE)

get.distances.all(test.from[1,],test.to[1:3,],return.rownums=F,return.latlons=F)
get.distances.all(test.from[1,],test.to[1:3,],return.rownums=FALSE,return.latlons=TRUE)
get.distances.all(test.from[1,],test.to[1:3,],return.rownums=TRUE,return.latlons=FALSE)
get.distances.all(test.from[1,],test.to[1:3,],return.rownums=TRUE,return.latlons=TRUE)
```

```
get.distances.all(test.from[1:2,],test.to[1,],return.rownums=F,return.latlons=F)
get.distances.all(test.from[1:2,],test.to[1,],return.rownums=FALSE,return.latlons=TRUE)
get.distances.all(test.from[1:2,],test.to[1,],return.rownums=TRUE,return.latlons=FALSE)
get.distances.all(test.from[1:2,],test.to[1,],return.rownums=TRUE,return.latlons=TRUE)

round(get.distances.all(test.from[1:2,],test.to[1:3,],return.rownums=F,return.latlons=F),1)
get.distances.all(test.from[1:2,],test.to[1:3,],return.rownums=FALSE,return.latlons=T)
get.distances.all(test.from[1:2,],test.to[1:3,],return.rownums=TRUE,return.latlons=F)
get.distances.all(test.from[1:2,],test.to[1:3,],return.rownums=TRUE,return.latlons=TRUE)
get.distances.all(test.from[1:2,],test.to[1:3,], return.rownums=TRUE,
  return.latlons=TRUE, units='km')
get.distances.all(test.from[1:2,],test.to[1:3,], return.rownums=TRUE,
  return.latlons=TRUE, units='miles')

get.distances.all(test.from[1,],test.to[1:3, ], return.crosstab=TRUE)
get.distances.all(test.from[1:2,],test.to[1, ], return.crosstab=TRUE)
round(get.distances.all(test.from[1:2,],test.to[1:3, ],return.crosstab=TRUE, units='miles'),2)
round(get.distances.all(test.from[1:2,],test.to[1:3, ],return.crosstab=TRUE, units='km'),2)
```

`get.distances.chunked` *Call a function once per chunk & save output as file (breaks large input data into chunks)*

Description

Call get.distances function in chunks, when list of frompoints is so long it taxes RAM (e.g. 11m blocks), saving each chunk as a separate .RData file in current working directory

Usage

```
get.distances.chunked(
  frompoints,
  topoints,
  fromchunksize,
  tochunksize,
  FUN = get.distances,
  folder = getwd(),
  ...
)
```

Arguments

`frompoints`	Require matrix or data.frame of lat/lon vauels that can be passed to get.distances function (colnames 'lat' and 'lon')
`topoints`	Require matrix or data.frame of lat/lon vauels that can be passed to get.distances function (colnames 'lat' and 'lon')
`fromchunksize`	Required, number specifying how many points to analyze at a time (per chunk).
`tochunksize`	(not yet implemented - current default is to use all topoints at once) number specifying how many points to analyze at a time (per chunk).
`FUN`	Optional function, `get.distances` by default, no other value allowed currently.
`folder`	Optional path specifying where to save .RData files, default is getwd()
`...`	Other parameters to pass to `get.distances`, such as `radius` or `units`

Details

filesizes if crosstab format (FASTEST & avoid needing rownums which take >twice as long & 1.25x sized file):
80MB file/chunk if 1k blocks x 11k topoints/chunk:
y=get.distances.chunked(testpoints(11e6), testpoints(11000), 1e3, units='km',return.crosstab=TRUE)
800MB file/chunk if 10k blocks x 11k topoints/chunk:
y=get.distances.chunked(testpoints(11e6), testpoints(11000), 1e4, units='km',return.crosstab=TRUE)

Value

Returns vector of character elements that are filenames for saved .RData output files in current working directory or specified folder.

See Also

ff and others related to parallelization, etc.

`get.distances.prepaired`

Find distances between points, for pairs of points already organized as pairs.

Description

`get.distances.prepaired` returns all the distances between each specified pair of points.

Usage

```
get.distances.prepaired(pts)
```

Arguments

`pts`	A matrix or data.frame that has columns names 'fromlon', 'fromlat', 'tolon', 'tolat' with datum=WGS84 assumed.

Details

May need to fix cases where only a single row is input. This function returns a matrix or vector of distances, between points specified as pairs of lat/lon values. Points are specified using latitude and longitude in decimal degrees. Relies on the **sp** package for the `spDists` and `SpatialPoints` functions.

Value

Returns a vector of distances as numbers, in kilometers. Each element corresponds to one row in pts.

See Also

get.distances.all for a useful general function, get.distances for get.distances() which allows you to specify a search radius and get distances only within that radius which can be faster, get.nearest which finds the distance to the single nearest point within a specified search radius instead of all topoints, and proxistat which calculates a proximity score for each spatial unit based on distances to nearby points.

Examples

```
test.from <- structure(list(fromlat = c(38.9567309094, 38.9507043428, 38.0),
 fromlon = c(-77.0896572305, -77.0896199948, -77.0)),
 .Names = c("lat", "lon"), row.names = c("one", "two", "three"), class = "data.frame")
test.to <- structure(list(tolat = c(38.9575019287, 38.9507043428, 38.9514152435),
 tolon = c(-77.0892818598, -77.0896199948, -77.0972395245)),
 .Names = c("lat", "lon"), class = "data.frame", row.names = c("a", "b", "c"))
get.distances.prepaired(data.frame(
  fromlat=test.from$lat, fromlon=test.from$lon, tolat=test.to$lat, tolon=test.to$lon)
)
```

---

get.nearest *Find distance from each point in a set to the nearest of a second set of points (by lat/lon).*

Description

get.nearest returns the distance from each point in a set to the nearest of a second set of points (by lat/lon).

Usage

```
get.nearest(
  frompoints,
  topoints,
  units = "miles",
  ignore0 = FALSE,
  return.rownums = FALSE,
  return.latlons = FALSE,
  radius = Inf
)
```

Arguments

frompoints	A matrix or data.frame with two cols, 'lat' and 'lon' (or only 2 cols that are lat and lon in that order) with datum=WGS84 assumed.
topoints	A matrix or data.frame with two cols, 'lat' and 'lon' (or only 2 cols that are lat and lon in that order) with datum=WGS84 assumed.
units	A string that is 'miles' by default, or 'km' for kilometers, specifying units for distances returned.
ignore0	A logical, default is FALSE, specifying whether to ignore distances that are zero and report only the minimum nonzero distance. Useful if nearest point other than self, where frompoints=topoints, for example.

`return.rownums` Logical value, TRUE by default. If TRUE, value returned also includes these 2 columns: a col named fromrow of index numbers starting at 1 specifying the frompoint and a similar col named n specifying the row of the nearest topoint.

`return.latlons` Logical value, FALSE by default. If TRUE, value returned also includes four extra columns, showing fromlat, fromlon, tolat, tolon.

`radius` Optional number, default is Inf. Distance within which search should be limited, or max distance that will be returned.

Details

This function returns a vector of distances, which are the distances from one set of points to the nearest single member (if any) of another set of points. Points are specified using latitude and longitude in decimal degrees. Relies on the **sp** package for the `spDists` and `SpatialPoints` functions.

A future version may use get.distances.all() but for performance only use it for distance pairs (pairs of points) that have been initially quickly filtered using lat/lon to be not too far, in an attempt to go much faster in an initial pass. *** old get.nearest with loops takes 42 seconds vs 3 seconds for this version, for 100k frompoints and 100 topoints: Sys.time(); x=get.nearest(t100k, t100); Sys.time() > Sys.time(); x=get.nearest(testpoints(1e6), testpoints(100)); Sys.time()
[1] 14:33:05 EDT
[1] 14:33:33 EDT <30 seconds from 1 mill to 100 points, as in finding nearest of 100 sites for 9 But R hung/crashed on 11mill frompoints – Probably out of memory. *** Need to break it up into batches of maybe 1 to 100 million distances at a time? There are 11,078,297 blocks according to http://www.census.gov/geo/maps-data/data/tallies/national_geo_tallies.html

Value

By default, returns a vector of distances, but can return a matrix of numbers, with columns that can include fromrow and torow indexing which is nearest (first if >1 match) of topoints, fromlat, fromlon, tolat, tolon, and d (distance). ** Returns Inf when no topoints are found within the radius, and also when a distance to nearest is zero but ignore0=TRUE. Distance returned is in miles by default, but with option to set units='km' to get kilometers. See parameters for details on other formats that may be returned if specified.

See Also

`get.distances` which gets distances between all points (within an optional search radius), `get.distances.all` which allows you to get distances between all points, `get.distances.prepaired` for finding distances when data are already formatted as pairs of points, and `proxistat` which calculates a proximity score for each spatial unit based on distances to nearby points.

Examples

```
set.seed(999)
t1=testpoints(1)
t10=testpoints(10)
t100=testpoints(100)
t1k=testpoints(1e3)
t10k=testpoints(1e4)
t100k=testpoints(1e5)
t1m=testpoints(1e6)
#t10m=testpoints(1e7)
```

```
get.nearest(t1, t1)
get.nearest(t1, t10[2, ,drop=FALSE])
get.nearest(t10, t1k)
get.nearest(t10, t1k, radius=500, units='km')
get.nearest(t10, t1k, radius=10, units='km')

test.from <- structure(list(fromlat = c(38.9567309094, 38.9507043428),
 fromlon = c(-77.0896572305, -77.0896199948)), .Names = c("lat", "lon"),
 row.names = c("6054762", "6054764"), class = "data.frame")
test.to <- structure(list(tolat = c(38.9575019287, 38.9507043428, 38.9514152435),
 tolon = c(-77.0892818598, -77.0896199948, -77.0972395245)), .Names = c("lat", "lon"),
 class = "data.frame", row.names = c("6054762", "6054763", "6054764"))
get.nearest(test.from, test.to)
get.nearest(testpoints(10), testpoints(30))
```

`latlon.colnames.check` *Utility function to check for valid lat/lon columns*

Description

Used by functions like `get.distances` to check input parameters frompoints and topoints

Usage

```
latlon.colnames.check(mypoints)
```

Arguments

`mypoints`	A matrix or data.frame

Value

Returns a vector of colnames such as c('lat', 'lon') or stops if problem found

See Also

`get.distances.all` which allows you to get distances between all points, `get.distances.prepaired` for finding distances when data are already formatted as pairs of points, `get.nearest` which finds the distance to the single nearest point within a specified search radius instead of all topoints, and `proxistat` which calculates a proximity score for each spatial unit based on distances to nearby points.

`get.distances` which allows you to specify a search radius and get distances only within that radius which can be faster, `get.distances.prepaired` for finding distances when data are already formatted as pairs of points, `get.nearest` which finds the distance to the single nearest point within a specified search radius instead of all topoints, and `proxistat` which calculates a proximity score for each spatial unit

Examples

```
#
```

`linesegments` *Add line segments connecting pairs of points, to a plot*

Description

Accepts vectors of x and y values of pairs of points (e.g., longitude and latitude), reformats them and adds line segments to an existing plot. Each line segment drawn connects one point its paired point. The two sets of points have to be equal in length, set up as pairs.

Usage

```
linesegments(xfrom, yfrom, xto, yto, ...)
```

Arguments

`xfrom`	required numeric vector of x values for starting points
`yfrom`	required numeric vector of y values for starting points
`xto`	required numeric vector of x values for ending points
`yto`	required numeric vector of y values for ending points
`...`	optional additional parameters to pass to `lines`

Details

This function also silently returns a matrix of two columns. Each column is a vector that has elements sequenced in groups of three – the from point, then the to point, then NA to signify a break in line drawing. See `lines`

Value

Draws lines(), one line segment from each starting point to its corresponding ending point.

See Also

`testpoints` and `get.nearest`

Examples

```
t10  <- testpoints(10,  minlat = 25, maxlat = 45, minlon = -100, maxlon = -60)
t100 <- testpoints(100, minlat = 25, maxlat = 45, minlon = -100, maxlon = -60)
nears=as.data.frame( get.nearest(t10, t100, return.latlons=TRUE) )
plot(t10)
plot(t100, pch='.')
linesegments(nears$fromlon, nears$fromlat, nears$tolon, nears$tolat)
```

`lookup.states` *States and related areas dataset*

Description

This data set provides a variety of fields for US States, DC, PR, and Island Areas, including FIPS codes and area in square miles or square kilometers, for example, from the Census Bureau.

Format

A data.frame with 58 rows (States etc.) and 26 columns (variables like statename). See `get.state.info` for more details.

- [1,] "statename"
- [2,] "FIPS.ST"
- [3,] "ST"
- [4,] "ftpname"
- [5,] "REGION"
- [6,] "is.usa.plus.pr"
- [7,] "is.usa"
- [8,] "is.state"
- [9,] "is.contiguous.us"
- [10,] "is.island.areas"
- [11,] "area.sqmi"
- [12,] "area.sqkm"
- [13,] "landarea.sqmi"
- [14,] "landarea.sqkm"
- [15,] "waterarea.sqmi"
- [16,] "waterarea.sqkm"
- [17,] "inland.sqmi"
- [18,] "inland.sqkm"
- [19,] "coastal.sqmi"
- [20,] "coastal.sqkm"
- [21,] "greatlakes.sqmi"
- [22,] "greatlakes.sqkm"
- [23,] "territorial.sqmi"
- [24,] "territorial.sqkm"
- [25,] "lat"
- [26,] "lon"

Details

For information on FIPS codes, see http://www.census.gov/geo/reference/ansi.html, and also see https://www.census.gov/geo/reference/geoidentifiers.html

Regarding Island Areas see http://www.census.gov/geo/reference/gtc/gtc_island.html which states the following: Separate from the Island Areas is the term "U.S. Minor Outlying Islands." The Island Areas of the United States are American Samoa, Guam, the Commonwealth of the Northern Mariana Islands (Northern Mariana Islands), and the United States Virgin Islands. The U.S. Minor Outlying Islands refers to certain small islands under U.S. jurisdiction in the Caribbean and Pacific: Baker Island, Howland Island, Jarvis Island, Johnston Atoll, Kingman Reef, Midway Islands, Navassa Island, Palmyra Atoll, and Wake Island. These areas usually are not part of standard data products.

Source

Derived from https://www.census.gov/geo/reference/state-area.html (for area data in square miles etc.) obtained 4/2015, and FIPS codes from Census Bureau.

See Also

get.state.info in **ejanalysis** package (http://ejanalysis.github.io/ejanalysis/), and state.abb via data(state) and the Census 2010 packages http://lakshmi.calit2.uci.edu/census2000/ and http://www.jstatsoft.org/v37/i06

---

meters.per.degree.lat *Convert degrees latitude to meters North-South.*

Description

meters.per.degree.lat returns meters traveled North-South per decimal degrees latitude, given latitude (Northern Hemisphere).

Usage

```
meters.per.degree.lat(theta)
```

Arguments

theta	The decimal degrees of latitude of the Northern Hemisphere location(s) of interest, as number or vector of numbers.

Details

This function returns the meters traveled North-South per decimal degree latitude, at a given latitude (Northern Hemisphere). This is an approximation and is less accurate further from the given latitude. Based on http://en.wikipedia.org/wiki/Latitude#Length_of_a_degree_of_latitude and
http://en.wikipedia.org/wiki/Longitude#Length_of_a_degree_of_longitude
Input theta is latitude on WGS84.
Also see http://en.wikipedia.org/wiki/Longitude and http://en.wikipedia.org/wiki/Decimal_degrees

Value

Returns meters traveled N-S, as a number or vector of numbers the same length as the input.

See Also

`meters.per.degree.lon` for a similar function but for travel East-West, with more detailed explanation/help, and `get.distances.all` and `get.distances` for distances between points, and related functions.

Examples

```
meters.per.degree.lat(32)
meters.per.degree.lat(c(0,45,72))
```

`meters.per.degree.lon` *Convert degrees longitude to meters East-West.*

Description

`meters.per.degree.lon` returns meters traveled East-West per decimal degree longitude, given latitude (Northern Hemisphere).

Usage

```
meters.per.degree.lon(theta)
```

Arguments

`theta`	The decimal degrees of latitude of the Northern Hemisphere location(s) of interest, as number or vector of numbers.

Details

This function returns the meters traveled East-West per decimal degree longitude, at a given latitude (Northern Hemisphere). This is an approximation and is less accurate further from the given latitude. Based on http://en.wikipedia.org/wiki/Latitude#Length_of_a_degree_of_latitude and
http://en.wikipedia.org/wiki/Longitude#Length_of_a_degree_of_longitude

Input theta is latitude on WGS84. Function is as follows:
theta.r <- 0.01745329 * theta
ecc2 <- 0.00669438
return(20037508 * cos(theta.r) / (180 * sqrt(1- ecc2 * (sin(theta.r))^2)))
Based on the following calculations:
The equatorial.radius used is 6378137.0 in meters
2 pi / 360 = 0.01745329
for the WGS84 ellipsoid with a = 6,378,137.0 m and b = 6,356,752.3142 m.
equatorial.radius <- 6378137.0 # a in meters
b=6356752.3142
ecc2 <- (equatorial.radius^2 - b^2)/equatorial.radius^2
ecc2 <- (6378137.0^2 - 6356752.3142^2) / 6378137.0^2

pi * equatorial.radius = 20037508

Also see http://en.wikipedia.org/wiki/Longitude and http://en.wikipedia.org/wiki/Decimal_degrees

Value

Returns meters traveled East-West per decimal degree longitude, as a number or vector of numbers the same length as the input.

See Also

meters.per.degree.lat for a similar function but for travel North-South, and deltalon.per.km for the inverse of this function (other than a factor of 1000), and get.distances.all and also get.distances to get distances between points, and related functions.

Examples

```
meters.per.degree.lon(32)
meters.per.degree.lon(c(0,45,72))
```

---

pointmap *Read a series of dbf files and join compile them as a single data.frame*

Description

Can help with a series of downloaded Census data files, such as one file per State. Reads each using read.dbf and combines them all as a data.frame. They must all have the same columns, and each file just provides additional rows of data.

Usage

```
pointmap(
  bin,
  lat,
  lon,
  vartext = "x",
  areatext = "area",
  coloring = rainbow(length(unique(bin))),
  asp = c(1, 1),
  pch = ".",
  ...
)
```

Arguments

bin	Indicates color of each point. A vector of integers used to index the coloring vector. That would ideally include just 1:n where n is the n number of unique values and the index to the coloring vector, so each unique value 1:5, for example, is assigned a map color that is coloring[bin]
lat	vector of latitudes just interpreted as y values to plot

lon	vector of longitudes just interpreted as x values to plot
vartext	default is 'x', text to use in describing the field mapped
areatext	default is 'area', text to use in describing the area mapped
coloring	default is some basic colors from rainbow(). A vector specifying color for each bin (should be same length as number of unique bin values)
asp	default is c(1,1), aspect ratio of graphic (not right for AK, e.g.)
pch	default is '.', see par – defines type of marker for each point
...	other params to send to plot

Value

Draws a map

See Also

countypointmap

Examples

```
myfips <- bg.pts$FIPS[substr(bg.pts$FIPS,1,2)=='06'] # CA
pointmap(bin = floor(runif(n = length(myfips),1,5.99)), lat = bg.pts$lat[match(myfips, bg.pts$FIPS)], lon =
```

---

proxistat *Proximity Statistic for Each Location and Nearby Points*

Description

Calculate proximity statistic for each location, quantifying number of and proximities of nearby points. proxistat returns a proximity statistic (score) for each location (e.g., Census block).

Usage

```
proxistat(
  frompoints,
  topoints,
  area = 0,
  radius = 5,
  units = "km",
  decay = "1/d",
  wts,
  return.count = FALSE,
  return.nearest = FALSE,
  FIPS,
  pop,
  testing = FALSE,
  dfunc = "sp"
)
```

Arguments

frompoints	Locations of internal points of Census subunits. A matrix or data.frame with two cols, 'lat' and 'lon' with datum=WGS84 assumed. Decimal degrees. Required.
topoints	Locations of nearby points of interest, proximity to which is the basis of each Census unit's score. A matrix or data.frame with two cols, 'lat' and 'lon' with datum=WGS84 assumed. Decimal degrees. Required.
area	A number or vector of numbers giving size of each spatial unit with FIPS.pop, in square miles or square kilometers depending on the units parameter. Optional. Default is 0, in which case no adjustment is made for small or even zero distance, which can cause unrealistically large or even infinite/undefined scores. For zero distance if area=0, Inf will be returned for the score.
radius	NOTE: This default is not the same as the default in get.distances! Optional, a number giving distance defining nearby, i.e. the search radius, in km or miles depending on the codeunits parameter. Default is 5 (km if units='km'). Max is 5200 miles (roughly the distance from Hawaii to Maine).
units	A string that is 'miles' or 'km' for kilometers (default is 'km'), specifying units for distances returned and for radius input.
decay	A string specifying type of function to use when weighting by distance. The Default is '1/d' For '1/d' decay weighting (default), score is count of points within radius, divided by harmonic mean of distances (when count>0). Decay weighting also can be '1/d^2' or '1/1' to represent decay by inverse of squared distance, or no decay (equal weighting for all points).
wts	Optional vector of numbers same length as number of topoints. If wts is specified, the score for each of the frompoints will be the weighted sum of influences of topoints. For example, if decay='1/d' (default), proximity score = sum(wts/d) for all the topoints nearby. If decay='1/1', proximity score = sum(wts) for all the topoints nearby.
return.count	Optional, logical, defaults to FALSE, specifies if results returned should include a column with the count of topoints that were within radius, for each of the frompoints
return.nearest	Optional, logical, defaults to FALSE, specifies if results returned should include a column with the distance to the nearest single of the topoints, for each of the frompoints
FIPS	NOT USED CURRENTLY - COULD BE USED LATER TO AGGREGATE (rollup) TO BLOCK GROUPS FROM BLOCKS, FOR EXAMPLE. A vector of strings designating places that will be assigned scores where each is the Census FIPS code or other ID. Optional. Might want to have this be a factor not string to be faster, or ensure it is indexed on fips, or have separate FIPS.BG passed to this function.
pop	NOT USED CURRENTLY - COULD BE USED LATER TO AGGREGATE (rollup) TO BLOCK GROUPS FROM BLOCKS, FOR EXAMPLE. A number or vector of numbers giving population count of each spatial unit. Default is 1, which would give the unweighted average.
testing	Logical during work in progress
dfunc	Optional character element hf or slc to specify distance function Haversine or spherical law of cosines. If sp (default, fastest), it uses the **sp** package to find distances more accurately and more quickly.

Details

This uses get.distances with return.crosstab=TRUE. This function returns a vector of proximity scores, one for each location such as a Census block. For example, the proximity score may be used to represent how many hazardous waste sites are near any given neighborhood and how close they are. A proximity score quantifies the proximity and count of nearby points using a specified formula.
Proximity Score = distance-weighted count of points nearby (within search radius) (or with another optional weight for each topoint)
(or weighted distance to nearest single point if there are none within the radius).
This is the sum of 1/d or 1/d^2 or 1/1, depending on the decay weighting, (or with another optional weight for each topoint instead of the number 1) where d is the distance from census unit's internal point to user-defined point. The default proximity score, using 1/d, is the count of nearby points divided by the harmonic mean of their distances (n/harmean), (but adjusted when distance is very small, and using the nearest single one if none are nearby). This is the same as the sum of inverse distances. The harmonic mean distance (see harmean) is the inverse of the arithmetic mean of the inverses, or n / (sum of inverses).

Nearby is defined as a user-specified parameter, so only points within the specified distance are counted, except if none are nearby, the single nearest point (at any distance) is used.

Default relies on the **sp** package for the spDists and SpatialPoints functions. Other values of dfunc parameter are slower.

IMPORTANT:

To create a proximity score for a block group, one can find the score for each block in the block group and then find the population-weighted average of those block scores, for a single block group. FIPS for blocks can be used to find FIPS for block groups. FIPS for block groups can be used to find FIPS for tracts.

ADJUSTMENT FOR SMALL DISTANCES:

The adjustment for small distances ensure that each distance represents roughly the distance to the average resident within a spatial unit like a block, rather than just the distance to the center or internal point. The adjustment uses the area of the spatial unit and assumes residents are evenly spread across the unit. Distance is adjusted in each place if area of each spatial unit is specified, to ensure it represents roughly distance to average resident in the unit: The distance is capped to be no less than 0.9 x radius of a circle of area equal to census unit's area. This approximation treats unit as if it were a circle and assumes pop is evenly distributed within that circle's area, since
0.9r = 0.9 x sqrt(area/pi) = approx solution to dist from avg point (resident) in circle to a random point in the circle (facility or point of interest). The use of a minimum distance per areal unit is intended to help approximate the distance from the average resident rather than from the internal point or center of the areal unit. The approximation assumes distance to the average resident can be estimated as if homes and facilities were on average uniformly distributed within blocks (or whatever units are used) that were roughly circular on average. It relies on the fact that the average distance between two random points in a circle of radius R is 90 percent of R (Weisstein, Eric W. Disk Line Picking. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/DiskLinePicking.html). This means that if a population is randomly spread over a roughly circular block, a facility inside the block (i.e., very close to the internal point) typically would be 0.9R from the average person. The same math shows that the average point in the circle is 0.67R from the center, and 1.13R from the edge of the circle. We can describe this relationship using an equation that is a portion of the formula for the distance between two

random points in a circle of radius = 1. The formula uses b = the distance of the facility from the center as a fraction of the radius, and the integral over a represents distances of residences from the center. We can solve the equation using http://WolframAlpha.com, for b = 0, 0.5, or 1, representing points at the center, halfway to the edge, and at the edge of the circle. For example, we can use this equation for b = 0.5 to find that the average person, if randomly located in a circle of radius R, is a distance of about 0.8 R from a facility that is halfway between the center and edge of the circle. Note this is not the same as the expected location of a randomly placed facility, which would use b = sqrt(0.5) instead and gives a distance of about 0.9R. The following would be used as the input to WolframAlpha to derive the 0.9 approximation: Integrate((1/Pi) Sqrt(a + (Sqrt(0.5))^2 - 2(Sqrt(0.5)) Sqrt(a) cos(t)), a, 0, 1, t, 0, pi) http://bit.ly/1GJ9UID

Value

By default, returns a vector of numbers, the proximity scores, one for each of the frompoints (or if testing, a matrix with 2 columns: fromrow and d for distance). Based on miles by default, or km depending on units. Returns +Inf for a unit if that area's area and distance are both zero.

See Also

get.distances and get.distances.all for distances between points, and get.nearest which finds the distance to the single nearest point within a specified search radius instead of all topoints.

Examples

```
test.from <- structure(list(fromlat = c(38.9567309094, 38.9507043428),
  fromlon = c(-77.0896572305, -77.0896199948)), .Names = c("lat", "lon"),
  row.names = c("6054762", "6054764"), class = "data.frame")
test.to <- structure(list(tolat = c(38.9575019287, 38.9507043428, 38.9514152435),
  tolon = c(-77.0892818598, -77.0896199948, -77.0972395245)), .Names = c("lat", "lon"),
  class = "data.frame", row.names = c("6054762", "6054763", "6054764"))

set.seed(999)
t1=testpoints(1)
t10=testpoints(10)
t100=testpoints(100)
t1k=testpoints(1e3)
t10k=testpoints(1e4)
t100k=testpoints(1e5)
t1m=testpoints(1e6)

proxistat(t1, t10k, radius=1, units='km')

proxistat(t10, t10k)

subunitscores = proxistat(frompoints=test.from, topoints=test.to,
  area=rep(0.2, length(test.from[,1])), radius=1, units='km')
print(subunitscores)
subunitpop = rep(1000, length(test.from$lat))
subunits = data.frame(FIPS=substr(rownames(test.from), 1, 5),
  pop=subunitpop, stringsAsFactors=FALSE )
unitscores = aggregate(subunits,
 by=list(subunits$FIPS), FUN=function(x) {Hmisc::wtd.mean(x$score, wts=x$pop, na.rm=TRUE)}
)
print(unitscores)
## Not run:
output = proxistat.chunked(blocks[ , c('lon','lat')], topoints=rmp, fromchunksize=10000, area=blocks$area /
```

```
   return.count=TRUE, return.nearest=TRUE )
output=as.data.frame(output)
if (class(blocks$fips)!='character') {blocks$fips <- lead.zeroes(blocks$fips, 15)}
blocks$FIPS.BG <- get.fips.bg(blocks$fips)
bg.proxi <- data.frame()
bg.proxi$scores  <-  aggregate( cbind(d=output$scores, pop=blocks$pop), by=list(blocks$FIPS.BG), function(x)
if ('nearestone.d' %in% colnames(output)) { bg.proxi$nearestone.d <- aggregate( output$d, by=list(blocks$FIP
if ('count.near' %in% colnames(output))  { bg.proxi$count.near  <- aggregate( cbind(d=output$count.near, po

## End(Not run)
```

`proxistat.assemble.chunks` *read files storing proxistat result chunks and assemble into one result*

Description

read files storing proxistat result chunks and assemble into one result

Usage

```
proxistat.assemble.chunks(files, folder = getwd())
```

Arguments

`files`	Required character vector of names of .RData files to read and combine
`folder`	Optional character element specifying directory where files are stored, defaults to getwd()

Value

Matrix that contains combined results found in all the files

Examples

```
## Not run:
 fnames=proxistat.chunked(testpoints(10), testpoints(5), fromchunksize = 4, assemble=FALSE,
   folder=file.path(getwd(), 'temp'))
 output=proxistat.assemble.chunks(files=fnames, folder=file.path(getwd(), 'temp'))

## End(Not run)
```

`proxistat.chunked` *Call proxistat once per chunk & save output as file (breaks large input data into chunks)*

Description

Call proxistat function in chunks, when list of frompoints is so long it taxes RAM (e.g. 11m blocks), saving each chunk as a separate .RData file in current working directory

Usage

```
proxistat.chunked(
  frompoints,
  topoints,
  fromchunksize,
  tochunksize,
  startchunk = 1,
  FUN = proxistat,
  folder = getwd(),
  savechunks = FALSE,
  assemble = TRUE,
  saveproxistats = FALSE,
  area,
  file = "proxistats.RData",
  ...
)
```

Arguments

`frompoints`	Require matrix or data.frame of lat/lon vauels that can be passed to get.distances function (colnames 'lat' and 'lon')
`topoints`	Require matrix or data.frame of lat/lon vauels that can be passed to get.distances function (colnames 'lat' and 'lon')
`fromchunksize`	Required, number specifying how many points to analyze at a time (per chunk).
`tochunksize`	(not currently required - current default is to use all topoints at once) number specifying how many points to analyze at a time (per chunk).
`startchunk`	Optional integer defaults to 1. Specifies which chunk to start with, in case some already have been done. Currently, still must pass entire dataset to this function even if some of the earlier chunks have already been analyzed.
`FUN`	Optional function, `proxistat` by default, and other values not implemented yet.
`folder`	Optional path specifying where to save .RData file(s) – chunk-specific files and/or assembled results file – default is getwd()
`savechunks`	Optional logical defaults to FALSE. Specifies whether to save .RData file of each chunk
`assemble`	Optional logical defaults to TRUE. Specifies whether to assemble all chunks into one variable called proxistats, which is saved as `file` in `folder` and returned by this function.
`saveproxistats`	Optional logical defaults to FALSE. Specifies whether to save .RData file of assembled results as proxistats matrix. Ignored if assemble=FALSE.

area	Optional number or vector of numbers giving size of each spatial unit with FIPS.pop, in square miles or square kilometers depending on the units parameter. Optional. Default is to pass nothing to proxistat, and default there is 0, in which case no adjustment is made for small or even zero distance, which can cause unrealistically large or even infinite/undefined scores. For zero distance if area=0, Inf will be returned for the score.
file	Optional name of file created if assemble=TRUE and saveproxistats=TRUE, defaults to proxistats.RData using save(proxistats, 'proxistats.RData')
...	Other parameters to pass to proxistat such as units or wts

Details

*** Still slow for all blocks in USA & 10k topoints (several hours) Filesizes:

80MB file/chunk if 1k blocks x 11k topoints/chunk: y=get.distances.chunked(testpoints(11e6), testpoints(11000), 1e3, units='km')

800MB file/chunk if 10k blocks x 11k topoints/chunk: y=get.distances.chunked(testpoints(11e6), testpoints(11000), 1e4, units='km')

Value

If assemble=TRUE, returns assembled set of all chunks as matrix of 1 or more columns. If assemble=FALSE but savechunks=TRUE, returns vector of character elements that are filenames for saved .RData output files in current working directory or specified folder. Each saved output is a vector of proximity scores if FUN=proxistat, or matrix with extra columns depending on return. parameters above. Otherwise, returns NULL.

---

proxistat.rollup *Convert Census Block Proximity Statistics to Block Group Statistics*

Description

Aggregate proximity statistics already calculated for each Census block, up to one summary for each Census block group. The resulting proximity score, distance to nearest single point, or count of nearby points is just the population-weighted mean of values in the blocks within a given block group.

Usage

```
proxistat.rollup(output, blocksfips, blocksfipsbg, blockspop)
```

Arguments

output	Required matrix of results from proxistat or proxistat.chunked. Must be same number of rows and order as blocksfips. Output parameter must be output of proxistat or proxistat.chunked and contain the colname scores, and can also have colnames nearestone.d and/or count.near.
blocksfips	Required character vector of 15-digit Census block FIPS codes (not numeric, must have leading zeroes as needed).

blocksfipsbg	Required character vector of 12-digit Census block group FIPS codes (not numeric, must have leading zeroes as needed). Same length and order as blocksfips.
blockspop	Required numeric vector of population counts in Census blocks. Same length and order as blocksfips.

Details

The population-weighted mean might not be the only statistic of interest.
To get the maximum count of sites near any single block in the block group, try aggregate(output[, 'count.near'], by=list(blocks$FIPS.BG), FUN=max).
To get the shortest distance from any block in the block group to the nearest site, try aggregate(output[, 'nearestone.d'], by=list(blocks$FIPS.BG), FUN=min).
To find out how many unique sites are within X km of the internal point of any block in the block group, for example, is harder, because it requires retaining details on which sites were near a given block, i.e., much more data would be the input to an aggregating function.

Value

Returns a data.frame with FIPS.BG and same fields proxistat can provide (depending on what is in the parameter called output): scores, nearestone.d, count.near, but with one row for each of the block groups defined by FIPS.BG. Units (miles or km) are unchanged from those used to create input parameters.

See Also

proxistat and proxistat.chunked to create proximity statistics, and see get.distances and get.distances.all for distances between points, and get.nearest which finds the distance to the single nearest point within a specified search radius instead of all topoints. See also rollup via http://ejanalysis.github.io/ejanalysis/

Examples

```
## Not run:
require(UScensus2010blocks); require(Hmisc); require(data.table); require(analyze.stuff); require(ejanalysis
blocks=get.blocks()
bgp <- proxistat.rollup(output=output, blocksfips=blocks$fips, blocksfipsbg=blocks$FIPS.BG, blockspop=blocks

## End(Not run)
```

---

testpoints *Generate a number of randomly placed points, as latitude/longitude values.*

Description

Generate a number of randomly placed points, as latitude/longitude values.

Usage

```
testpoints(
  n,
  minlat = 40,
  maxlat = 42,
  minlon = -125,
  maxlon = -70,
  as.df = TRUE
)
```

Arguments

n	Numeric value, required, TRUE by default. Specifies how many testpoints to return. Must be an integer between zero and 50 million, and not NA.
minlat	Default 40. A number that is the minimum latitude in decimal degrees to use for generating random points within some range.
maxlat	Default 42. A number that is the maximum latitude in decimal degrees to use for generating random points within some range.
minlon	Default -125. A number that is the minimum longitude in decimal degrees to use for generating random points within some range.
maxlon	Default -70. A number that is the maximum longitude in decimal degrees to use for generating random points within some range.
as.df	Logical, default is TRUE, in which case returns a data.frame, otherwise a matrix.

Details

This function returns n points at random locations using uniform distributions of latitude and longitude values, with specified ranges. Points are specified using latitude and longitude in decimal degrees.

Value

By default, returns a data.frame (or matrix if as.df=FALSE) that has 2 columns: lat and lon, in decimal degrees, with 1 row per point.

See Also

`get.distances.all` which allows you to get distances between all points, `get.distances.prepaired` for finding distances when data are already formatted as pairs of points, `get.nearest` which finds the distance to the single nearest point within a specified search radius instead of all topoints, and `proxistat` which calculates a proximity score for each spatial unit based on distances to nearby points.

Examples

```
testpoints(19,minlat=47,maxlat=48)
get.distances(testpoints(1000),testpoints[10],radius=999,return.rownums=TRUE,return.latlons=TRUE)
```

Index