**Mapping Toolbox™**

Reference

MATLAB®

# How to Contact MathWorks

| | | |
|---|---|---|
| | Latest news: | www.mathworks.com |
| | Sales and services: | www.mathworks.com/sales_and_services |
| | User community: | www.mathworks.com/matlabcentral |
| | Technical support: | www.mathworks.com/support/contact_us |
| | Phone: | 508-647-7000 |

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

*Mapping Toolbox™ Reference*

**Revision History**

| | | |
|---|---|---|
| May 1997 | First printing | New for Version 1.0 |
| October 1998 | Second printing | Version 1.1 |
| November 2000 | Third printing | Version 1.2 (Release 12) |
| July 2002 | Online only | Revised for Version 1.3 (Release 13) |
| September 2003 | Online only | Revised for Version 1.3.1 (Release 13SP1) |
| January 2004 | Online only | Revised for Version 2.0 (Release 13SP1+) |
| April 2004 | Online only | Revised for Version 2.0.1 (Release 13SP1+) |
| June 2004 | Fourth printing | Revised for Version 2.0.2 (Release 14) |
| October 2004 | Online only | Revised for Version 2.0.3 (Release 14SP1) |
| March 2005 | Fifth printing | Revised for Version 2.1 (Release 14SP2) |
| August 2005 | Sixth printing | Minor revision for Version 2.1 |
| September 2005 | Online only | Revised for Version 2.2 (Release 14SP3) |
| March 2006 | Online only | Revised for Version 2.3 (Release 2006a) |
| September 2006 | Seventh printing | Revised for Version 2.4 (Release 2006b) |
| March 2007 | Online only | Revised for Version 2.5 (Release 2007a) |
| September 2007 | Eighth printing | Revised for Version 2.6 (Release 2007b) |
| March 2008 | Online only | Revised for Version 2.7 (Release 2008a) |
| October 2008 | Online only | Revised for Version 2.7.1 (Release 2008b) |
| March 2009 | Online only | Revised for Version 2.7.2 (Release 2009a) |
| September 2009 | Online only | Revised for Version 3.0 (Release 2009b) |
| March 2010 | Online only | Revised for Version 3.1 (Release 2010a) |
| September 2010 | Online only | Revised for Version 3.2 (Release 2010b) |
| April 2011 | Online only | Revised for Version 3.3 (Release 2011a) |
| September 2011 | Online only | Revised for Version 3.4 (Release 2011b) |
| March 2012 | Online only | Revised for Version 3.5 (Release 2012a) |
| September 2012 | Online only | Revised for Version 3.6 (Release 2012b) |
| March 2013 | Online only | Revised for Version 3.7 (Release 2013a) |
| September 2013 | Online only | Revised for Version 4.0 (Release 2013b) |
| March 2014 | Online only | Revised for Version 4.0.1 (Release 2014a) |
| October 2014 | Online only | Revised for Version 4.0.2 (Release 2014b) |
| March 2015 | Online only | Revised for Version 4.1 (Release 2015a) |
| September 2015 | Online only | Revised for Version 4.2 (Release 2015b) |

# Contents

**Functions — Alphabetical List**

**1**

# Functions — Alphabetical List

# aer2ecef

Local spherical AER to geocentric ECEF

## Syntax

```
[X,Y,Z] = aer2ecef(az,elev,slantRange,lat0,lon0,h0,spheroid)
[ ___ ] = aer2ecef( ___ ,angleUnit)
```

## Description

`[X,Y,Z] = aer2ecef(az,elev,slantRange,lat0,lon0,h0,spheroid)` returns Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian coordinates corresponding to coordinates `az`, `elev`, `slantRange` in a local spherical system having the same origin. Any of the first six numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = aer2ecef( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `az`, `elev`, `lat0`, and `lon0`.

## Input Arguments

### `az` — Azimuth angles
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `elev` — Elevation angles
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface

normal. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `slantRange` — Distances from local origin
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is used. Units are determined by the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

**spheroid — Reference spheroid**
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

**angleUnit — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

# Output Arguments

**X — ECEF x-coordinates**
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single | double`

**Y — ECEF y-coordinates**
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single | double`

**Z — ECEF z-coordinates**
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: single | double

## See Also
aer2geodetic | ecef2aer | enu2ecef | ned2ecef

# aer2enu

Local spherical AER to local Cartesian ENU

## Syntax

```
[xEast,yNorth,zUp] = aer2enu(az,elev,slantRange)
[ ___ ] = aer2enu( ___ ,angleUnit)
```

## Description

`[xEast,yNorth,zUp] = aer2enu(az,elev,slantRange)` returns coordinates in a local east-north-up (ENU) Cartesian system corresponding to coordinates `az`, `elev`, `slantRange` in a local spherical system having the same origin. Any of the three numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = aer2enu( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `az` and `elev`..

## Input Arguments

### `az` — Azimuth angles
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `elev` — Elevation angles
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface normal. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: single | double

### slantRange — Distances from local origin
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is used. Units are determined by the LengthUnit property of the spheroid input.

Data Types: single | double

### angleUnit — Units of angles
'degrees' (default) | 'radians'

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: char

# Output Arguments

### xEast — Local ENU x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array.

### yNorth — Local ENU y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array.

### zUp — Local ENU z-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array.

## See Also
aer2ned | enu2aer

# aer2geodetic

Local spherical AER to geodetic

## Syntax

```
[lat,lon,h] = aer2geodetic(az,elev,slantRange,lat0,lon0,h0,spheroid)
[ ___ ] = aer2geodetic( ___ ,angleUnit)
```

## Description

`[lat,lon,h] = aer2geodetic(az,elev,slantRange,lat0,lon0,h0,spheroid))` returns geodetic coordinates corresponding to coordinates `az`, `elev`, `slantRange` in a local spherical system. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = aer2geodetic( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `az`, `elev`, `lat0`, `lon0`, and outputs `lat`, `lon`.

## Examples

### Zermatt to the Matterhorn

Compute the latitude, longitude and orthometric height of the summit of the Matterhorn (Monte Cervino) from its azimuth, elevation and (slant) range relative to Zermatt, Switzerland. All distances and lengths are in meters.

Origin (reference point): Zermatt.

```
fmt = get(0,'Format');
format short g

lat0 = dm2degrees([46  1])  % convert degree-minutes to degrees
lon0 = dm2degrees([ 7 45])
hOrthometric0 = 1620;
hGeoid = 53;
h0 = hOrthometric0 + hGeoid
```

```
lat0 =

      46.017


lon0 =

       7.75


h0 =

       1673
```

Azimuth, elevation, and slant range to Matterhorn summit.

```
az = 237.8;
elev = 18.755;
slantRange = 8871.7;
```

Latitude, longitude, and ellipsoidal height of summit.

```
[lat, lon, hEllipsoidal] = aer2geodetic( ...
    az, elev, slantRange, lat0, lon0, h0, wgs84Ellipsoid)

lat =

      45.976


lon =

      7.6583


hEllipsoidal =

        4531
```

Orthometric height of summit.

```
hGeoid = 53;
hOrthometric = hEllipsoidal - hGeoid
format(fmt)

hOrthometric =
```

4478

# Input Arguments

### `az` — Azimuth angles
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `elev` — Elevation angles
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface normal. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `slantRange` — Distances from local origin
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is used. Units are determined by the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### `lat` — Geodetic latitudes
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the closed interval [-90 90].

### lon — Longitudes
scalar value | vector | matrix | N-D array

Longitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the interval [-180 180].

### h — Ellipsoidal heights
scalar value | vector | matrix | N-D array

Ellipsoidal heights of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object

## See Also
aer2ecef | enu2geodetic | geodetic2aer | ned2geodetic

# aer2ned

Local spherical AER to local Cartesian NED

## Syntax

```
[xNorth,yEast,zDown] = aer2ned(az,elev,slantRange)
[ ___ ] = aer2ned( ___ ,angleUnit)
```

## Description

`[xNorth,yEast,zDown] = aer2ned(az,elev,slantRange)` returns coordinates in a local north—east—down (NED) Cartesian system corresponding to coordinates `az`, `elev`, `slantRange` in a local spherical system having the same origin. Any of the three numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = aer2ned( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `az` and `elev`..

## Input Arguments

**az — Azimuth angles**
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

**elev — Elevation angles**
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, specified as a scalar, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface normal. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single | double`

### `slantRange` — Distances from local origin
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is used. Units are determined by the `LengthUnit` property of the `spheroid` input.

Data Types: `single | double`

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### `xNorth` — Local NED x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array.

### `yEast` — Local NED y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array.

### `zDown` — Local NED z-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array.

## See Also
`aer2enu | ned2aer`

# almanac

Parameters for Earth, planets, Sun, and Moon

## Syntax

```
almanac
almanac(body)
data = almanac(body,parameter)
data = almanac(body,parameter,units)
data = almanac(parameter,units,referencebody)
```

## Description

almanac is not recommended. Use `earthRadius`, referenceEllipsoid, referenceSphere, or `wgs84Ellipsoid` instead.

`almanac` displays the names of the celestial objects available in the almanac.

`almanac(body)` lists the options, or parameters, available for each celestial body. Valid *body* strings are

```
'earth'       'pluto'
'jupiter'     'saturn'
'mars'        'sun'
'mercury'     'uranus'
'moon'        'venus'
'neptune'
```

`data = almanac(body,parameter)` returns the value of the requested parameter for the celestial body specified by *body*.

Valid *parameter* strings are `'radius'` for the planetary radius, `'ellipsoid'` or `'geoid'` for the two-element ellipsoid vector, `'surfarea'` for the surface area, and `'volume'` for the planetary volume.

For the Earth, *parameter* can also be any valid predefined ellipsoid string. In this case, the two-element ellipsoid vector for that ellipsoid model is returned. Valid ellipsoid definition strings for the Earth are

| | |
|---|---|
| `'everest'` | 1830 Everest ellipsoid |
| `'bessel'` | 1841 Bessel ellipsoid |
| `'airy'` | 1830 Airy ellipsoid |
| `'clarke66'` | 1866 Clarke ellipsoid |
| `'clarke80'` | 1880 Clarke ellipsoid |
| `'international'` | 1924 International ellipsoid |
| `'krasovsky'` | 1940 Krasovsky ellipsoid |
| `'wgs60'` | 1960 World Geodetic System ellipsoid |
| `'iau65'` | 1965 International Astronomical Union ellipsoid |
| `'wgs66'` | 1966 World Geodetic System ellipsoid |
| `'iau68'` | 1968 International Astronomical Union ellipsoid |
| `'wgs72'` | 1972 World Geodetic System ellipsoid |
| `'grs80'` | 1980 Geodetic Reference System ellipsoid |
| `'wgs84'` | 1984 World Geodetic System ellipsoid |

For the Earth, the *parameter* strings `'ellipsoid'` and `'geoid'` are equivalent to`'grs80'`.

data = almanac(*body*,*parameter*,*units*) specifies the units to be used for the output measurement, where *units* is any valid distance units string. Note that these are linear units, but the result for surface area is in square units, and for volume is in cubic units. The default units are `'kilometers'`.

data = almanac(*parameter*,*units*,*referencebody*) specifies the source of the information. This sets the assumptions about the shape of the celestial body used in the calculation of volumes and surface areas. A *referencebody* string of `'actual'` returns a tabulated value rather than one dependent upon a ellipsoid model assumption. Other possible *referencebody* strings are `'sphere'` for a spherical assumption and `'ellipsoid'` for the default ellipsoid model. The default reference body is `'sphere'`.

For the Earth, any of the preceding predefined ellipsoid definition strings can also be entered as a reference body.

For Mercury, Pluto, Venus, the Sun, and the Moon, the eccentricity of the ellipsoid model is zero, that is, the `'ellipsoid'` reference body is actually a sphere.

## More About

### Tips

Take care when using angular arc length units for distance measurements. All planets have a radius of 1 radian, for example, and an area unit of *square degrees* indicates unit squares, 1 degree of arc length on a side, not 1-degree-by-1-degree quadrangles.

## See Also

distance | earthRadius | referenceEllipsoid | referenceSphere | wgs84Ellipsoid

# angl2str

Format angle strings

## Syntax

```
str = angl2str(angle)
str = angl2str(angle,signcode)
str = angl2str(angle,signcode,units)
str = angl2str(angle,signcode,units,n)
```

## Description

`str = angl2str(angle)` converts a numerical vector of angles in degrees to a string matrix.

`str = angl2str(angle,signcode)` uses the string *signcode* to specify the method for indicating that a given angle is positive or negative. *signcode* may be one of the following:

| | |
|---|---|
| `'ew'` | east/west notation; trailing 'e' (positive longitudes) or 'w' (negative longitudes) |
| `'ns'` | north/south notation; trailing 'n' (positive latitudes) or 's' (negative latitudes) |
| `'pm'` | plus/minus notation; leading '+' (positive angles) or '-' (negative angles) |
| `'none'` | blank/minus notation; leading '-' for negative angles or sign omitted for positive angles (the default value) |

`str = angl2str(angle,signcode,units)` uses the string *units* to indicate both the units in which `angle` is provided *and* to control the output format. *units* can be `'degrees'` (the default value), `'radians'`, `'degrees2dm'`, or `'degrees2dms'`. *units* may be abbreviated and is case-insensitive. The interpretations of *units* are as follows:

| Units | Units of Angle | Output Format |
|---|---|---|
| `'degrees'` | degrees | decimal degrees |

| Units | Units of Angle | Output Format |
|---|---|---|
| `'degrees2dm'` | degrees | degrees + decimal minutes |
| `'degrees2dms'` | degrees | degrees + minutes + decimal seconds |
| `'radians'` | radians | decimal radians |

`str = angl2str(angle,`*`signcode`*`,`*`units`*`,n)` uses the integer `n` to control the number of significant digits provided in the output. `n` is the power of 10 representing the last place of significance in the number of degrees, minutes, seconds, or radians -- for *units* of `'degrees'`, `'degrees2dm'`, `'degrees2dms'`, and `'radians'`, respectively. For example, if `n = -2` (the default), `angl2str` rounds to the nearest hundredth. If `n = -0`, `angl2str` rounds to the nearest integer. And if `n = 1`, `angl2str` rounds to the tens place, although positive values of `n` are of little practical use. In all cases, the interpretation of the parameter `n` is consistent between `angl2str` and `roundn`.

## Examples

Create a string matrix to represent a series of values in DMS units, using the north-south format:

```
a = -3:1.5:3;
str = angl2str(a,'ns','degrees2dms',-3)

str =
 3^{\circ} 00' 00.000" S
 1^{\circ} 30' 00.000" S
 0^{\circ} 00' 00.000"
 1^{\circ} 30' 00.000" N
 3^{\circ} 00' 00.000" N
```

These LaTeX strings are displayed (using either `text` or `textm`) as

```
3º 00' 00.000" S
1º 30' 00.000" S
0º 00' 00.000"
1º 30' 00.000" N
3º 00' 00.000" N
```

## More About

### Tips

The purpose of this function is to make angular-valued variables into strings suitable for map display. In general, the interpretation of the parameter n by `angl2str` is consistent with that of `roundn`.

### See Also
`str2angle` | `dist2str`

# angledim

Convert angles units

---

**Note:** The `angledim` function has been replaced by four, more specific, functions: `fromRadians`, `fromDegrees`, `toRadians`, and `toDegrees`. However, `angledim` will be maintained for backward compatibility. The functions `deg2rad`, `rad2deg`, and `unitsratio` provide additional alternatives.

---

## Syntax

```
angleOut = angledim(angleIn,from,to)
```

## Description

`angleOut = angledim(angleIn,from,to)` returns the value of the input angle `angleIn`, which is in units specified by the valid angle units string *from*, in the desired units given by the valid angle units string *to*. Angle units strings are `'degrees'` for "decimal" degrees or `'radians'` for radians

## Examples

Convert from degrees to radians:

```
angledim(23.45134,'degrees','radians')

ans =
    0.4093
```

## See Also

`degrees2dms` | `deg2rad` | `fromDegrees` | `fromRadians` | `toDegrees` | `toRadians` | `rad2deg` | `unitsratio`

# antipode

Point on opposite side of globe

## Syntax

```
[newlat,newlon] = antipode(lat,lon)
[newlat,newlon] = antipode(lat,lon,angleunits)
```

## Description

`[newlat,newlon] = antipode(lat,lon)` returns the geographic coordinates of the points exactly opposite on the globe from the input points given by `lat` and `lon`. All angles are in degrees.

`[newlat,newlon] = antipode(lat,lon,angleunits)` specifies the input and output units with the string *angleunits*. The string *angleunits* can be either `'degrees'` or `'radians'`. It can be abbreviated and is case-insensitive.

## Examples

### Find Antipode of Given Point

Given a point (43ºN, 15ºE), find its antipode:

```
[newlat,newlong] = antipode(43,15)
newlat =
        -43
newlong =
        -165
```

or (43ºS, 165ºW).

### Find Antipode of North and South Poles

Perhaps the most obvious antipodal points are the North and South Poles. The function `antipode` demonstrates this:

```
[newlat,newlong] = antipode(90,0,'degrees')
newlat =
         -90
newlong =
         180
```

Note that in this case longitudes are irrelevant because all meridians converge at the poles.

## Find Antipode of MathWorks Headquarters

This example shows how to find the antipode of the location of the MathWorks corporate headquarters in Natick, Massachusetts. The example maps the headquarters location and its antipode in an orthographic projection.

Specify latitude and longitude as degree-minutes-seconds and then convert to decimal degrees.

```
mwlat = dms2degrees([ 42 18 2.5])
mwlon = dms2degrees([-71 21 7.9])


mwlat =

    42.3007


mwlon =

   -71.3522
```

Find the antipode.

```
[amwlat amwlon] = antipode(mwlat,mwlon)


amwlat =

   -42.3007


amwlon =
```

```
108.6478
```

Prove that these points are antipodes. The `distance` function shows them to be 180 degrees apart.

```
dist = distance(mwlat,mwlon,amwlat,amwlon)
```

```
dist =

  180.0000
```

Generate a map centered on the original point and then another map centered on the antipode.

```
figure
subplot(1,2,1)
axesm ('MapProjection','ortho','origin',[mwlat mwlon],...
       'frame','on','grid','on')
load coast
geoshow(lat,long,'displaytype','polygon')
geoshow(mwlat,mwlon,'Marker','o','Color','red')
title(sprintf('Looking down at\n(%s,%s)', ...
    angl2str(mwlat,'ns'), angl2str(mwlon,'ew')))

subplot(1,2,2)
axesm ('MapProjection','ortho','origin',[amwlat amwlon],...
       'frame','on','grid','on')
geoshow(lat,long,'displaytype','polygon')
geoshow(amwlat,amwlon,'Marker','o','Color','red')
title(sprintf('Looking down at\n(%s,%s)', ...
    angl2str(amwlat,'ns'), angl2str(amwlon,'ew')))
```

Looking down at
( 42.30° N , 71.35° W )

Looking down at
( 42.30° S , 108.65° E )

# arcgridread

Read gridded data set in Arc ASCII Grid Format

## Syntax

```
[Z,R] = arcgridread(filename)
```

## Description

`[Z,R] = arcgridread(filename)` reads a grid from a file in Arc ASCII Grid format. `Z` is a 2-D array containing the data values. `R` is a referencing matrix (see `makrefmat`). `NaN` is assigned to elements of `V` corresponding to null data values in the grid file.

## Examples

```
[Z,R] = arcgridread('MtWashington-ft.grd');
mapshow(Z,R,'DisplayType','surface');
xlabel('x (easting in meters)'); ylabel('y (northing in meters)')
demcmap(Z)

% View the terrain in 3-D
axis normal; view(3); axis equal; grid on
zlabel('elevation in feet')
```
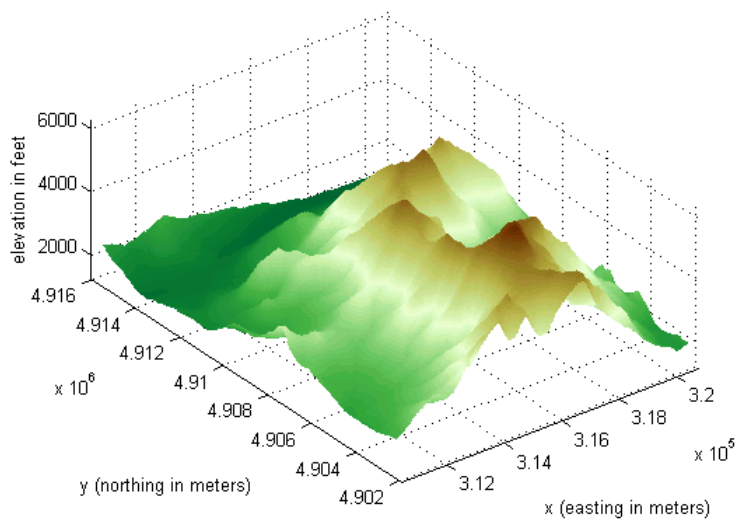
## See Also

```
makerefmat | mapshow | sdtsdemread
```

# areaint

Surface area of polygon on sphere or ellipsoid

## Syntax

```
area = areaint(lat,lon)
area = areaint(lat,lon,ellipsoid)
area = areaint(lat,lon,units)
area = areaint(lat,lon,ellipsoid,units)
```

## Description

`area = areaint(lat,lon)` calculates the spherical surface area of the polygon specified by the input vectors `lat` and `lon`. The calculation uses a line integral approach. The output, `area`, is the fraction of surface area covered by the polygon on a unit sphere. To supply multiple polygons, separate the polygons by NaNs in the input vectors. Accuracy of the integration method is inversely proportional to the distance between lat/lon points.

`area = areaint(lat,lon,ellipsoid)` calculates the surface area of the polygon on the ellipsoid or sphere defined by the input `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form [`semimajor_axis eccentricity`]. The output, `area`, is in squares units corresponding to the units of `ellipsoid`.

`area = areaint(lat,lon,units)` uses the units defined by the input string *units*. If omitted, default units of degrees are assumed.

`area = areaint(lat,lon,ellipsoid,units)` uses both the inputs `ellipsoid` and *units* in the calculation.

## Examples

Consider the area enclosed by a 30º lune from pole to pole and bounded by the prime meridian and 30ºE. You can use the function `areaquad` to get an exact solution:

```
area = areaquad(90,0,-90,30)
area =
    0.0833
```

This is 1/12 the spherical area. The more points used to define this polygon, the more integration steps `areaint` takes, improving the estimate. This first attempt takes a point every 30º of latitude:

```
lats = [-90:30:90,60:-30:-60]';
lons = [zeros(1,7), 30*ones(1,5)]';
area = areaint(lats,lons)
area =
    0.0792
```

Now, calculate a better estimate, with one point every 1º of latitude:

```
lats = [-90:1:90,89:-1:-89]';
lons = [zeros(1,181), 30*ones(1,179)]';
area = areaint(lats,lons)
area =
    0.0833
```

## More About

### Algorithms

This function enables the measurement of areas enclosed by arbitrary polygons. This is a numerical estimate, using a line integral based on Green's Theorem. As such, it is limited by the accuracy and resolution of the input data.

Areas are computed for arbitrary polygons on the ellipsoid or sphere
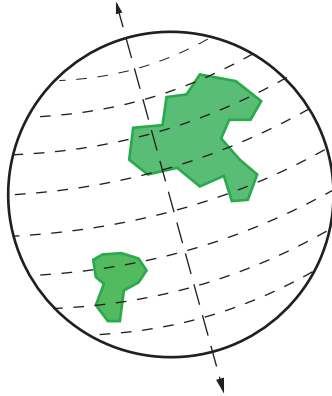


An area is returned for each NaN-separated polygon

Given sufficient data, the `areaint` function is the best method for determining the areas of complex polygons, such as continents, cloud cover, and other natural or derived features. The calculations in this function employ a spherical Earth assumption. For nonspherical ellipsoids, the latitude data is converted to the auxiliary authalic sphere.

## See Also
`almanac` | `areamat` | `areaquad`

# areamat

Surface area covered by nonzero values in binary data grid

## Syntax

```
A = areamat(BW,R)
A = areamat(BW,refvec,ellipsoid)
[A, cellarea] = areamat(...)
```

## Description

`A = areamat(BW,R)` returns the surface area covered by the elements of the binary regular data grid `BW`, which contain the value 1 (`true`). `BW` can be the result of a logical expression such as `BW = (topo > 0)`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(BW)` and its `RasterInterpretation` must be `'cells'`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

The output `A` expresses surface area as a fraction of the surface area of the unit sphere (4*pi), so the result ranges from 0 to 1.

`A = areamat(BW,refvec,ellipsoid)` calculates the surface area on the ellipsoid or sphere defined by the input `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form [`semimajor_axis eccentricity`]. The units of the output, `A`, are the square of the length units in which the semimajor axis is provided. For example, if `ellipsoid` is replaced with `wgs84Ellipsoid('kilometers')`, then `A` is in square kilometers.

`[A, cellarea] = areamat(...)` returns a vector, `cellarea`, describing the area covered by the data cells in `BW`. Because all the cells in a given row are exactly the same size, only one value is needed per row. Therefore `cellarea` has size `M-by-1`, where `M = size(BW,1)` is the number of rows in `BW`.

## Examples

```
load topo
area = areamat((topo>127),topolegend)

area =
    0.2411
```

Approximately 24% of the Earth has an altitude greater than 127 meters. The surface area of this portion of the Earth in square kilometers if a spherical ellipsoid is:

```
earth = referenceSphere('earth', 'km');
area = areamat((topo>127),topolegend,earth)

area =
   1.2299e+08
```

To illustrate the `cellarea` output, consider a smaller map:

```
BW = ones(9,18);
refvec = [.05 90 0] % each cell 20x20 degrees
[area,cellarea] = areamat(BW,refvec)

area =
    1.0000
cellarea =
    0.0017
    0.0048
    0.0074
    0.0091
```

```
0.0096
0.0091
0.0074
0.0048
0.0017
```

Each entry of `cellarea` represents the portion of the unit sphere's total area a cell in that row of `BW` would contribute. Since the column extends from pole to pole in this case, it is symmetric.

# More About

### Tips

Given a regular data grid that is a logical 0-1 matrix, the `areamat` function returns the area corresponding to the true, or 1, elements. The input data grid can be a logical statement, such as `(topo>0)`, which is 1 everywhere that `topo` is greater than 0 meters, and 0 everywhere else. This is an illustration of that matrix:



This calculation is based on the `areaquad` function and is therefore limited only by the granularity of the cellular data.

## See Also
`areaint | areaquad`

# areaquad

Surface area of latitude-longitude quadrangle

## Syntax

```
area = areaquad(lat1,lon1,lat2,lon2)
area = areaquad(lat1,lon1,lat2,lon2,ellipsoid)
area = areaquad(lat1,lon1,lat2,lon2,ellipsoid,units)
```

## Description

`area = areaquad(lat1,lon1,lat2,lon2)` returns the surface area bounded by the parallels `lat1` and `lat2` and the meridians `lon1` and `lon2`. The output `area` is a fraction of the unit sphere's area of 4π, so the result ranges from 0 to 1.

`area = areaquad(lat1,lon1,lat2,lon2,ellipsoid)` allows the specification of the ellipsoid model with `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. When `ellipsoid` is input, the resulting `area` is given in terms of the (squared) units of the ellipsoid. For example, if the ellipsoid `referenceEllipsoid('grs80','kilometers')` is used, the resulting area is in km$^2$.

`area = areaquad(lat1,lon1,lat2,lon2,ellipsoid,units)` specifies the units of the inputs. The default is `'degrees'`.

## Definitions

A latitude-longitude quadrangle is a region bounded by two meridians and two parallels. In spherical geometry, it is the intersection of a *lune* (a section bounded by two meridians) and a *zone* (a section bounded by two parallels).
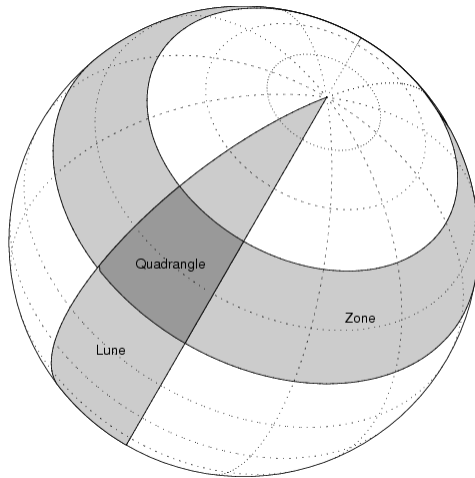
## Examples

Find the fraction of the Earth's surface that lies between 30ºN and 45ºN, and also between 25ºW and 60ºE:

```
area = areaquad(30,-25,45,60)

area =
    0.0245
```

Assuming a spherical ellipsoid, find the surface area of the Earth in square kilometers.

```
earthellipsoid = referenceSphere('earth','km');
area = areaquad(-90,-180,90,180,earthellipsoid)

area =
   5.1006e+08
```

For comparison,

```
earthellipsoid.SurfaceArea

ans =
   5.1006e+08
```

## More About

### Algorithms

The `areaquad` calculation is exact, being based on simple spherical geometry. For nonspherical ellipsoids, the data is converted to the auxiliary authalic sphere.

## See Also

`almanac` | `areaint` | `areamat`

# avhrrgoode

Read AVHRR data product stored in Goode Projection

## Syntax

```
[latgrat,longrat,z] = avhrrgoode(region,filename)
[...] = avhrrgoode(region,filename,scalefactor)
[...] = avhrrgoode(region,filename,scalefactor,latlim,lonlim)
[...] = avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize)
[...] =
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,...
nrows,ncols)
[...] =
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,...
nrows,ncols,resolution)
[...] =
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,...
nrows,ncols,resolution,precision)
```

## Description

`[latgrat,longrat,z] = avhrrgoode(region,filename)` reads data from an Advanced Very High Resolution Radiometer (AVHRR) data set with a nominal resolution of 1 km that is stored in the Goode projection. Data in this format includes a nondimensional vegetation index (NDVI) and Global Land Cover Characteristics (GLCC) data sets. `region` is a string that specifies the geographic coverage of the file. Valid region strings are:

- `'g'` or `'global'`
- `'af'` or `'africa'`
- `'ap'` or `'australia/pacific'`
- `'ea'` or `'eurasia'`
- `'na'` or `'north america'`
- `'sa'` or `'south america'`

`filename` is a string specifying the name of the data file. Output `Z` is a geolocated data grid with coordinates `latgrat` and `longrat` in units of degrees. `Z`, `latgrat`, and `longrat` are of class double. Projected coordinates that lie within the interrupted areas of the projection are set to NaN. A scale factor of 100 is applied to the original data set, so that `Z` contains every $100^{th}$ point in both X and Y directions.

`[...] = avhrrgoode(region,filename,scalefactor)` uses the integer `scalefactor` to downsample the data. A scale factor of 1 returns every point. A scale factor of 10 returns every $10^{th}$ point. The default value is 100.

`[...] = avhrrgoode(region,filename,scalefactor,latlim,lonlim)` returns data for the specified region. The returned data can extend somewhat beyond the requested area. Limits are two-element vectors in units of degrees, with `latlim` in the range `[-90 90]` and `lonlim` in the range `[-180 180]`. `latlim` and `lonlim` must be ascending. If `latlim` and `lonlim` are empty, the entire area covered by the data file is returned. If the quadrangle defined by `latlim` and `lonlim` (when projected to form a polygon in the appropriate Goode projection) fails to intersect the bounding box of the data in the projected coordinates, then `Z`, `latgrat`, and `longrat` are returned as empty.

`[...] = avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize)` controls the size of the graticule matrices. `gsize` is a two-element vector containing the number of rows and columns desired. By default, `latgrat`, and `longrat` have the same size as `Z`.

`[...] = avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,...` `nrows,ncols)` overrides the dimensions for the standard file format for the selected region. This syntax is useful for data stored on CD-ROM, which may have been truncated to fit. Some global data sets were distributed with 16347 rows and 40031 columns of data on CD-ROMs. The default size for global data sets is 17347 rows and 40031 columns of data.

`[...] = avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,...` `nrows,ncols,`*`resolution`*`)` reads a data set with the spatial resolution specified in meters. Specify `resolution` as either `1000` or `8000` (meters). If empty, the full resolution of 1000 meters is assumed. Data is also available at 8000-meter resolution. Nondimensional vegetation index data at 8-km spatial resolution has 2168 rows and 5004 columns.

```
[...] =
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,...
nrows,ncols,resolution,precision) reads a data set expecting the integer
```
`[...] = avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,...nrows,ncols,resolution,precision)` reads a data set expecting the integer `precision` specified. If empty, `'uint8'` is assumed. `'uint16'` is appropriate for some files. Check the metadata (`.txt` or README) file in the GLCC `ftp` folder for specification of the file format and contents. In either case, `Z` is converted to class double.

## Background

The United States maintains a family of satellite-based sensors to measure climate change under the Earth Observing System (EOS) program. The precursors to the EOS data are the data sets produced by NOAA and NASA under the Pathfinder program. These are data derived from the Advanced High Resolution Radiometer sensor flown on the NOAA Polar Orbiter satellites, NOAA-7, -9, and -11, and have spatial resolutions of about 1 km. The data from the AVHRR sensor is processed into separate land, sea, and atmospheric indices. Land area data is processed to a nondimensional vegetation index (NDVI) or land cover classification and stored in binary files in the Plate Carrée, Goode, and Lambert projections. Sea data is processed to surface temperatures and stored in HDF formats. `avhrrgoode` reads land data saved in the Goode projection with global and continental coverage at 1 km. It can also read 8 km data with global coverage.

## Limitations

Most files store the data in scaled integers. Though this function returns the data as double, the scaling from integer to float is not performed. Check the data's README file for the appropriate scaling parameters.

## Examples

### Example 1 — Downsampled Classified Global GLCC Coverage

Read and display every 50th point from the Global Land Cover Characteristics (GLCC) file covering the entire globe with the USGS classification scheme, named `gusgs2_Og.img`. (To run the example, you must first download the file.)

```
[latgrat, longrat, Z] = avhrrgoode('global', ...
```

```
       'gusgs2_0g.img',50);

% Convert the geolocated data grid to an geolocated image.
uniqueClasses = unique(Z);
RGB = ind2rgb8(uint8(Z), jet(numel(uniqueClasses)));

% Display the data as an image using the Goode projection.
origin = [0 0 0];
ellipsoid = [6370997 0];
figure
axesm('MapProjection', 'goode', 'Origin', origin, ...
   'Geoid', ellipsoid)
geoshow(latgrat, longrat, RGB, 'DisplayType', 'image');
axis image off

% Plot the coastlines.
hold on
load coast
plotm(lat,long)
```
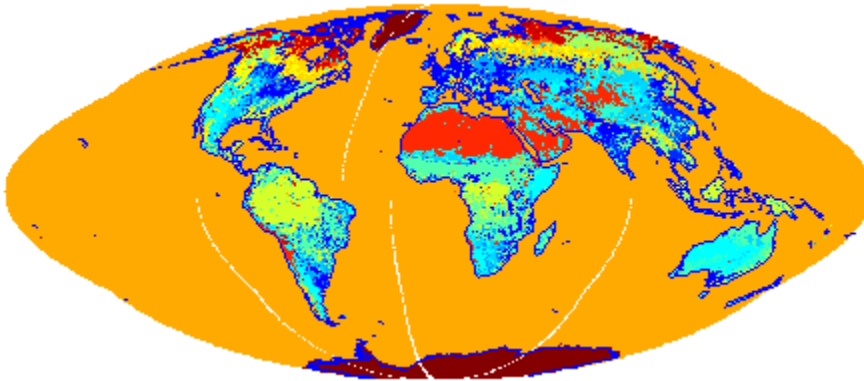


## Example 2 — Classified GLCC Data for California

Read and display every point from the Global Land Cover Characteristics (GLCC) file covering California with the USGS classification scheme, named nausgs1_2g.img. You must first download the file to run this example.

```
figure
```

```
usamap california
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
scalefactor = 1;
[latgrat, longrat, Z] = ...
   avhrrgoode('na', 'nausgs1_2g.img', scalefactor, latlim, lonlim);
geoshow(latgrat, longrat, Z, 'DisplayType', 'texturemap');

% Overlay vector data from usastatehi.shp.
california = shaperead('usastatehi', 'UseGeoCoords', true,...
   'BoundingBox', [lonlim;latlim]);
geoshow([california.Lat], [california.Lon], 'Color', 'black');
```
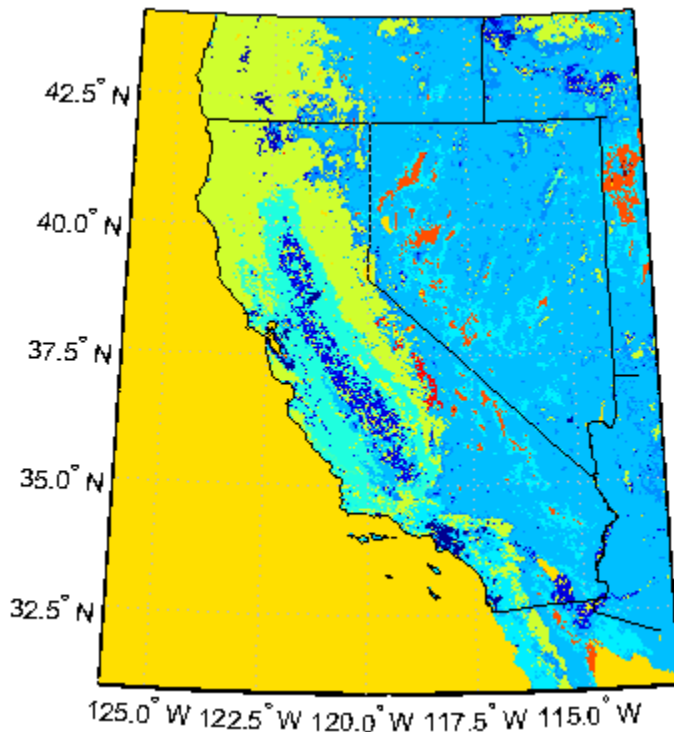
# More About

### Tips

This function reads the binary files as is. You should not use byte-swapping software on these files.

The AVHRR project and data sets are described in and provided by various U.S. Government Web sites. See the entry for Global Land Cover Characteristics (GLCC) in the tech note referred to below.

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

## See Also
avhrrlambert

# avhrrlambert

Read AVHRR data product stored in eqaazim projection

## Syntax

```
[latgrat,longrat,Z] = avhrrlambert(region,filename)
[...] = avhrrlambert(region,filename, scalefactor)
[...] = avhrrlambert(region,filename, scalefactor, latlim, lonlim)
[...] = avhrrlambert(region,filename, scalefactor, latlim, lonlim,
gsize)
[...] = avhrrlambert(region,filename, scalefactor, latlim, lonlim,
gsize,precision)
```

## Description

`[latgrat,longrat,Z] = avhrrlambert(region,filename)` reads data from an Advanced Very High Resolution Radiometer (AVHRR) data set with a nominal resolution of 1 km that is stored in the Lambert Equal Area Azimuthal projection. Data of this type includes the Global Land Cover Characteristics (GLCC). *region* specifies the coverage of the file. Valid regions are listed in the following table. *filename* is a string specifying the name of the data file. `Z` is a geolocated data grid with coordinates `latgrat` and `longrat` in units of degrees. A scale factor of 100 is applied to the original data set such that `Z` contains every 100th point in both X and Y.

| Region Specifiers |
| --- |
| `'a'` or `'asia'` |
| `'af'` or `'africa'` |
| `'ap'` or `'australia/pacific'` |
| `'e'` or `'europe'` |
| `'na'` or `'north america'` |
| `'sa'` or `'south america'` |

`[...] = avhrrlambert(region,filename, scalefactor)` uses the integer `scalefactor` to downsample the data. A scale factor of 1 returns every point. A scale factor of 10 returns every 10th point. The default value is 100.

[...] = avhrrlambert(*region*,*filename*, scalefactor, latlim, lonlim) returns data for the specified region. The result may extend somewhat beyond the requested area. The limits are two-element vectors in units of degrees, with `latlim` in the range [-90 90] and `lonlim` in the range [-180 180]. If `latlim` and `lonlim` are empty, the entire area covered by the data file is returned. If the quadrangle defined by `latlim` and `lonlim` (when projected to form a polygon in the appropriate Lambert Equal Area Azimuthal projection) fails to intersect the bounding box of the data in the projected coordinates, then `latgrat`, `longrat`, and `Z` are empty.

[...] = avhrrlambert(*region*,*filename*, scalefactor, latlim, lonlim, gsize) controls the size of the graticule matrices. `gsize` is a two-element vector containing the number of rows and columns desired. If omitted or empty, a graticule the size of the grid is returned.

[...] = avhrrlambert(*region*,*filename*, scalefactor, latlim, lonlim, gsize,*precision*) reads a data set with the integer *precision* specified. If omitted, 'uint8' is assumed. 'uint16' is appropriate for some files. Check the metadata (.txt or README) file in the ftp folder for specification of the file format and contents.

## Background

The United States plans to build a family of satellite-based sensors to measure climate change under the Earth Observing System (EOS) program. Early precursors to the EOS data are the data sets produced by NOAA and NASA under the Pathfinder program. These are data derived from the Advanced High Resolution Radiometer sensor flown on the NOAA Polar Orbiter satellites, NOAA-7, -9, and -11 with a spatial resolution of about 1 km. The data from the AVHRR sensor is processed into separate land, sea, and atmospheric indices. Land area data is processed to a nondimensional vegetation index or land cover classification and stored in binary files in the Plate Carrée, Goode, and Lambert Equal Area Azimuthal projections. Sea data is processed to surface temperatures and stored in HDF formats. This function reads land cover data for the continents saved in the Lambert Equal Area Azimuthal projection at 1 km.

# Examples

## Example 1

Read and display every 100th point from the Global Land Cover Characteristics (GLCC) file covering North America with the USGS classification scheme, named nausgs1_2l.img.

```
[latgrat, longrat, Z] = avhrrlambert('na','nausgs1_2l.img');
```

Display the data using the Lambert Equal Area Azimuthal projection.

```
origin = [50 -100 O];
ellipsoid = [6370997 O];
figure
axesm('MapProjection', 'eqaazim', 'Origin', ...
   origin, 'Geoid', ellipsoid)
geoshow(latgrat, longrat, Z, 'DisplayType', 'texturemap');
```



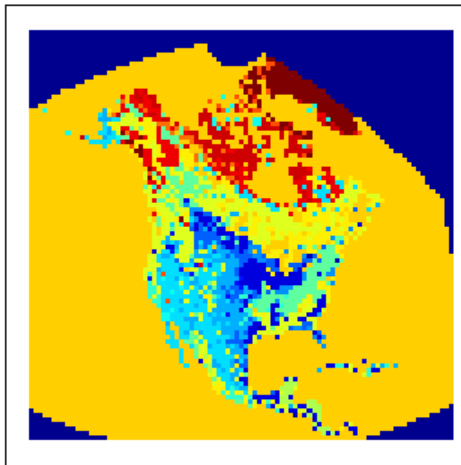## Example 2

Read and display every other point from the Global Land Cover Characteristics (GLCC) file covering Europe with the USGS classification scheme, named eausgs1_2le.img.

```
figure
```

```
worldmap france
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
scalefactor = 2;
[latgrat, longrat, Z] = avhrrlambert('e', 'eausgs1_2le.img', ...
    scalefactor, latlim, lonlim);
geoshow(latgrat, longrat, Z, 'DisplayType', 'texturemap');
geoshow('landareas.shp','FaceColor','none','EdgeColor','black')
```
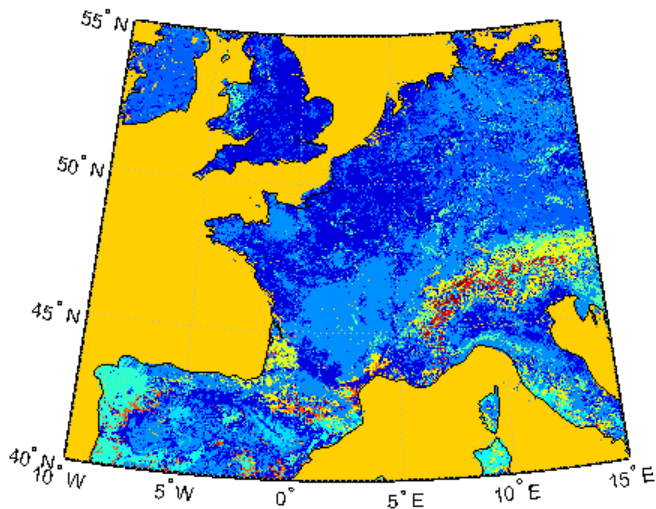


## More About

### Tips

This function reads the binary files as is. You should not use byte-swapping software on these files.

The AVHRR project and data sets are described in and provided by various U.S. Government Web sites.

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

## See Also
avhrrgoode

# axes2ecc

Eccentricity of ellipse from axes lengths

## Syntax

```
ecc = axes2ecc(semimajor,semiminor)
ecc = axes2ecc(vec)
```

## Description

`ecc = axes2ecc(semimajor,semiminor)` computes the eccentricity of an ellipse (or ellipsoid of revolution) given the semimajor and semiminor axes. The input data can be scalar or matrices of equal dimensions.

`ecc = axes2ecc(vec)` assumes a 2 element vector (vec) is supplied, where `vec` = [`semimajor semiminor`].

## See Also
`ecc2flat` | `ecc2n` | `majaxis` | `minaxis`

# axesm

Define map axes and set map properties

## Syntax

```
axesm
axesm(PropertyName,PropertyValue,...)
axesm(projid,PropertyName,PropertyValue,...)
```

## Description

`axesm` with no input arguments, initiates the `axesmui` map axes graphical user interface, which can be used to set map axes properties. This is detailed on the `axesmui` reference page.

`axesm(PropertyName,PropertyValue,...)` creates a map axes using the specified properties. Properties may be specified in any order, but the `MapProjection` property must be included.

`axesm(projid,PropertyName,PropertyValue,...)` uses the string `projid` to designate which map projection to use. `projid` should match one of the entries in the last column displayed by the `maps` function. You can also find a list of these strings in the User's Guide section "Summary and Guide to Projections".

The `axesm` function creates a map axes into which both vector and raster geographic data can be projected using functions such as `plotm` and `geoshow`. Properties specific to map axes can be assigned upon creation with `axesm`, and for an existing map axes they can be queried and changed using `getm` and `setm`. Use the standard `get` and `set` methods to query and control the standard MATLAB® axes properties of a map axes.

## Axes Definition

Map axes are standard MATLAB axes with different default settings for some properties and a MATLAB structure for storing projection parameters and other data. The main differences in default settings are

- Axes properties `XGrid`, `YGrid`, `XTick`, `YTick` are set to `'off'`.
- The `hold` mode is `'on'`.

The map projection structure stores the map axes properties, which, in addition to the special standard axes settings described here, allow Mapping Toolbox™ functions to recognize an axes or an opened FIG-file as a map axes. See "Map Axes Object Properties" on page 1-51, below, for descriptions of the map axes properties.

---

**Note:** In general, after re-opening a saved figure that contains a map axes, you should not attempt to modify the projection properties of that map axes.

---

**Note:** When you create a map axes with `axesm` and right click in the axes, a context menu appears. If you do not need the menu or it interferes with your application, you can disable it by resetting the `'ButtonDownFcn'` property of the axes:

```
ax = axesm('mercator');      % Right-clicking brings up context menu.
set(ax,'ButtonDownFcn',[])   % Context menu has been disabled.
```

## Examples

Create map axes for a Mercator projection, with selected latitude limits:

```
axesm('MapProjection','mercator','MapLatLimit',[-70 80])
```

In the preceding example, all properties not explicitly addressed in the call are set to either fixed or calculated defaults. The file `mercator.m` defines a projection function, so the same result could have been achieved with the function

```
axesm('mercator','MapLatLimit',[-70 80])
```

Each projection function includes default values for all properties. Any following property name/property value pairs are treated as overrides.

In either of the above examples, data displayed in the given map axes is in a Mercator projection. Any data falling outside the prescribed limits is not displayed.

---

**Note** The names of projection files are case sensitive. The projection files included in Mapping Toolbox software use only lowercase letters and Arabic numerals.

---

# Map Axes Object Properties

- "Properties That Control the Map Projection" on page 1-51
- "Properties That Control the Frame" on page 1-55
- "Properties That Control the Grid" on page 1-57
- "Properties That Control Grid Labeling" on page 1-60

## Properties That Control the Map Projection

AngleUnits

  {degrees} | radians

  *Angular unit of measure* — Controls the units of measure used for angles (including latitudes and longitudes) in the map axes. All input data are assumed to be in the given units; `'degrees'` is the default. For more information on angle units, see "Working with Angles: Units and Representations" in the *Mapping Toolbox User's Guide*.

Aspect

  {normal} | transverse

  *Display aspect* — Controls the orientation of the base projection of the map. When the aspect is `'normal'` (the default), *north* in the base projection is up. In a `transverse` aspect, north is to the right. A cylindrical projection of the whole world would look like a *landscape* display under a `'normal'` aspect, and like a *portrait* under a `'transverse'` aspect. Note that this property is not the same as projection aspect, which is controlled by the `Origin` property vector discussed later.

FalseEasting

  scalar {0}

  *Coordinate shift for projection calculations* — Modifies the position of the map within the axes. The projected coordinates are shifted in the *x*-direction by the amount of `FalseEasting`. The `FalseEasting` is in the same units as the projected coordinates, that is, the units of the first element of the `Geoid` map axes property. False eastings and northings are sometimes used to ensure nonnegative values of the projected coordinates. For example, the Universal Transverse Mercator uses a false easting of 500,000 meters.

FalseNorthing

    scalar {O}

*Coordinate shift for projection calculations* — Modifies the position of the map within the axes. The projected coordinates are shifted in the *y*-direction by the amount of `FalseNorthing`. The `FalseNorthing` is in the same units as the projected coordinates, that is, the units of the first element of the `Geoid` map axes property. False eastings and northings are sometimes used to ensure nonnegative values of the projected coordinates. For example, the Universal Transverse Mercator uses a false northing of 0 in the northern hemisphere and 10,000,000 meters in the southern.

FixedOrient

    scalar {[]} (read-only)

*Projection-based orientation* — This read-only property fixes the orientation of certain projections (such as the Cassini and Wetch). When empty, which is true for most projections, the user can alter the orientation of the projection using the third element of the `Origin` property. When fixed, the fixed orientation is always used.

Geoid

    [semimajor_axis eccentricity] or spheroid object

*Reference spheroid definition* — The spheroid (ellipsoid or sphere) for calculating the projections of any displayed map objects. It can be an referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form [semimajor_axis eccentricity]. The default value is an ellipsoid vector representing the unit sphere: [1 0].

MapLatLimit

    [southern_limit northern_limit]

*Geographic latitude limits of the display area* — Expressed as a two element vector of the form [southern_limit northern_limit]. This property can be set for many typical projections and geometries, but cannot be used with oblique projections or with `globe`, for example. When applicable, the `MapLatLimit` property may affect the origin latitude if the `Origin` property is not set explicitly when calling `axesm`. It may also determine the value used for `FLatLimit`. See "Accessing and Manipulating Map Axes Properties" for a more complete description of the applicability of `MapLatLimit` and its interaction with the origin, frame limits, and other properties.

MapLonLimit

[western_limit eastern_limit]

*Geographic longitude limits of the display area* — Expressed as a two element vector of the form [western_limit eastern_limit]. This property can be set for many typical projections and geometries, but cannot be used with oblique projections or with globe, for example. When applicable, the MapLonLimit property may affect the origin longitude if the Origin property is not set explicitly when calling axesm. It may also determine the value used for FLonLimit. See "Accessing and Manipulating Map Axes Properties" for a more complete description of the applicability of MapLonLimit and its interaction with the origin, frame limits, and other properties.

MapParallels

[lat] | [lat1 lat2]

*Projection standard parallels* — Sets the standard parallels of projection. It can be an empty, one-, or two-element vector, depending upon the projection. The elements are in the same units as the map axes AngleUnits. Many projections have specific, defining standard parallels. When a map axes object is based upon one of these projections, the parallels are set to the appropriate defaults. For conic projections, the default standard parallels are set to 15ºN and 75ºN, which biases the projection toward the northern hemisphere.

For projections with one defined standard parallel, setting the parallels to an empty vector forces recalculation of the parallel to the middle of the map latitude limits. For projections requiring two standard parallels, setting the parallels to an empty vector forces recalculation of the parallels to one-sixth the distance from the latitude limits (e.g., if the map latitude limits correspond to the northern hemisphere [0 90], the standard parallels for a conic projection are set to [15 75]). For azimuthal projections, the MapParallels property always contains an empty vector and cannot be altered.

See the *Mapping Toolbox User's Guide* for more information on standard parallels.

MapProjection

*projection_name* {no default}

*Map projection* — Sets the projection, and hence all transformation calculations, for the map axes object. It is required in the creation of map axes. It must be a member of the recognized projection set, which you can list by typing getm('MapProjection') or maps. For more information on projections, see the

*Mapping Toolbox User's Guide.* Some projections set their own defaults for other properties, such as parallels and trim limits.

`Origin`

`[latitude longitude orientation]`

*Origin and orientation for projection calculations* — Sets the map origin for all projection calculations. The latitude, longitude, and orientation should be in the map axes `AngleUnits`. Latitude and longitude refer to the coordinates of the map origin; orientation refers to an angle of skewness or rotation about the axis running through the origin point and the center of the earth. The default origin is $0^o$ latitude and a longitude centered between the map longitude limits. If a scalar is entered, it is assumed to refer to the longitude; if a two-element vector is entered, the default orientation is $0^o$, a normal projection. If an empty origin vector is entered, the origin is centered on the map longitude limits. For more information on the origin, see the *Mapping Toolbox User's Guide.*

`Parallels`

`0, 1, or 2 (read-only, projection-dependent)`

*Number of standard parallels* — This read-only property contains the number of standard parallels associated with the projection. See the *Mapping Toolbox User's Guide* for more information on standard parallels.

`ScaleFactor`

`scalar {1}`

*Scale factor for projection calculations* — Modifies the size of the map in projected coordinates. The geographic coordinates are transformed to Cartesian coordinates by the map projection equations and multiplied by the scale factor. Scale factors are sometimes used to minimize the scale distortion in a map projection. For example, the Universal Transverse Mercator uses a scale factor of 0.996 to shift the line of zero scale distortion to two lines on either side of the central meridian.

`Zone`

`ZoneSpec | {[] or 31N}`

*Zone for certain projections* — Specifies the zone for certain projections. A zone is a region on the globe that has a special set of projection parameters. In the Universal Transverse Mercator Projection, the world is divided into quadrangles that are generally 6 degrees wide and 8 degrees tall. The number in the zone designation refers to the longitude range, while the letter refers to the latitude range. Most projections use the same parameters for the entire globe, and do not require a zone.

## Properties That Control the Frame

Frame

    on | {off}

*Frame visibility* — Controls the visibility of the display frame box. When the frame is `'off'` (the default), the frame is not displayed. When the frame is `'on'`, an enclosing frame is visible. The frame is a patch that is plotted as the lowest layer of displayed map objects. Regardless of its display status, the frame always operates in terms of trimming map data.

FFill

    *scalar plotting point density* {100}

*Frame plotting precision* — Sets the number of points to be used in plotting the frame for display. The default value is 100, which for a rectangular frame results in a plot with 100 points for each side, or a total of 400 points. The number of points required for a reasonable display varies with the projection. Cylindrical projections such as the Miller require very few. Projections resulting in more complex frames, such as the Werner, look better with higher densities. The default value is generally sufficient.

FEdgeColor

    ColorSpec | {[O O O]}

*Color of the displayed frame edge* — Specifies the color used for the displayed frame. You can specify a color using a vector of RGB values or a MATLAB `colorspec` name. By default, the frame edge is displayed in black ([O O O]).

FFaceColor

    ColorSpec | {none}

*Color of the displayed frame face* — Specifies the color used for the displayed frame face. You can specify a color using a vector of RGB values or a MATLAB `colorspec` name. By default, the frame face is `'none'`, meaning no face color is filled in. Another useful color is `'cyan'` ([O 1 1]), which looks like water.

FLatLimit

    [southern_limit northern_limit]

*Latitude limits of map frame relative to projection origin* — The map frame encloses the area in which data and graticule lines are plotted and beyond which they are

trimmed. For non-oblique and non-azimuthal projections, which have quadrangular frames, this property controls the north-south extent of the frame. If a projection is made oblique by the inclusion of a non-zero rotation angle (the third element of the `Origin` vector), `FLatLimit` still applies, but in the rotated latitude-longitude system rather than in the geographic system. In the case of azimuthal projections, which have circular frames, `FLatLimit` takes the special form `[-Inf radius]` where radius is the spherical distance (in degrees or radians, depending on the `AngleUnits` property of the projection) from the projection origin to the edge of the frame.

---

**Note:** In most common situations, including non-oblique cylindrical and conic projections and polar azimuthal projections, there is no need to set `FLatLimit`; use `MapLatLimit` instead.

---

`FLineWidth`

*scalar* {2}

*Frame edge line width* — Sets the line width of the displayed frame edge. The value is a scalar representing points, which is 2 by default.

`FLonLimit`

`[western_limit eastern_limit]`

*Latitude limits of map frame relative to projection origin* — The map frame encloses the area in which data and graticule lines are plotted and beyond which they are trimmed. For non-oblique and non-azimuthal projections, which have quadrangular frames, this property controls the east-west extent of the frame. If a projection is made oblique by the inclusion of a non-zero rotation angle (the third element of the `Origin` vector), `FLonLimit` still applies, but in the rotated latitude-longitude system rather than in the geographic system. The `FLonLimit` property is ignored for azimuthal projections.

---

**Note:** In most common situations, including non-oblique cylindrical and conic projections, there is no need to set `FLonLimit`; use `MapLonLimit` instead.

---

`TrimLat`

`[southern_limit northern_limit]`
`(read-only, projection-dependent)`

*Bounds on* `FLatLimit` — This read-only property sets bounds on the values that `axesm` and `setm` will accept for the `MapLatLimit` and `FLatLimit` properties, which is necessary because some map projections cannot display the entire globe without extending to infinity. For example, `TrimLat` is `[-90 90]` degrees for most cylindrical projections and `[-86 86]` degrees for the Mercator projection because the north-south scale becomes infinite as one approaches either pole.

TrimLon

    [western_limit eastern_limit]
    (read-only, projection-dependent)

*Bounds on* `FLonLimit` — This read-only property sets bounds on the values that `axesm` and `setm` will accept for the `MapLonLimit` and `FLonLimit` properties, which is necessary because some map projections cannot display the entire globe without extending to infinity. For example, `TrimLon` is `[-135 135]` degrees for most conic projections.

## Properties That Control the Grid

Grid

    on | {off}

*Grid visibility* — Controls the visibility of the display grid. When the grid is `'off'` (the default), the grid is not displayed. When the grid is `'on'`, meridians and parallels are visible. The grid is plotted as a set of line objects.

GAltitude

    scalar z-axis value {Inf}

*Grid z-axis setting* — Sets the *z*-axis location for the grid when displayed. Its default value is infinity, which is displayed above all other map objects. However, you can set this to some other value for stacking objects above the grid, if desired.

GColor

    ColorSpec | {[O O O]}

*Color of the displayed grid* — Specifies the color used for the displayed grid. You can specify a color using a vector of RGB values or one of the MATLAB `colorspec` names. By default, the map grid is displayed in black (`[O O O]`).

GLineStyle

    LineStyle {:}

*Grid line style* — Determines the style of line used when the grid is displayed. You can specify any line style supported by the MATLAB `line` function. The default line style is a dotted line (that is, `':'`).

GLineWidth

    scalar {0.5}

*Grid line width* — Sets the line width of the displayed grid. The value is a scalar representing points, which is `0.5` by default.

MLineException

    vector of longitudes {[]}

*Exceptions to grid meridian limits* — Allows specific meridians of the displayed grid to extend beyond the grid meridian limits to the poles. The value must be a vector of longitudes in the appropriate angle units. For longitudes so specified, grid lines extend from pole to pole regardless of the existence of any grid meridian limits. This vector is empty by default.

MLineFill

    scalar plotting point density {100}

*Grid meridian plotting precision* — Sets the number of points to be used in plotting the grid meridians. The default value is 100 points. The number of points required for a reasonable display varies with the projection. Cylindrical projections such as the Miller require very few. Projections resulting in more complex shapes, such as the Werner, look better with higher densities. The default value is generally sufficient.

MLineLimit

    [north south] | [south north] {[]}

*Grid meridian limits* — Establishes latitudes beyond which displayed grid meridians do not extend. By default, this property is empty, so the meridians extend to the poles. There are two exceptions to the meridian limits. No meridian extends beyond the map latitude limits, and exceptions to the meridian limits for selected meridians are allowed (see above).

MLineLocation

    scalar interval or specific vector {30°}

*Grid meridian interval or specific locations* — Establishes the interval between displayed grid meridians. When a scalar interval is entered in the map axes

`MLineLocation`, meridians are displayed, starting at 0° longitude and repeating every interval in both directions, which by default is 30°. Alternatively, you can enter a vector of longitudes, in which case a meridian is displayed for each element of the vector.

`PLineException`

`vector of latitudes {[]}`

*Exceptions to grid parallel limits* — Allows specific parallels of the displayed grid to extend beyond the grid parallel limits to the International Date Line. The value must be a vector of latitudes in the appropriate angle units. For latitudes so specified, grid lines extend from the western to the eastern map limit, regardless of the existence of any grid parallel limits. This vector is empty by default.

`PLineFill`

`scalar plotting point density {100}`

*Grid parallel plotting precision* — Sets the number of points to be used in plotting the grid parallels. The default value is `100`. The number of points required for a reasonable display varies with the projection. Cylindrical projections such as the Miller require very few. Projections resulting in more complex shapes, such as the Bonne, look better with higher densities. The default value is generally sufficient.

`PLineLimit`

`[east west] | [west east] {[]}`

*Grid parallel limits* — Establishes longitudes beyond which displayed grid parallels do not extend. By default, this property is empty, so the parallels extend to the date line. There are two exceptions to the parallel limits. No parallel extends beyond the map longitude limits, and exceptions to the parallel limits for selected parallels are allowed (see above).

`PLineLocation`

`scalar interval or specific vector {15°}`

*Grid parallel interval or specific locations* — Establishes the interval between displayed grid parallels. When a scalar interval is entered in the map axes `PLineLocation`, parallels are displayed, starting at 0° latitude and repeating every interval in both directions, which by default is 15°. Alternatively, you can enter a vector of latitudes, in which case a parallel is displayed for each element of the vector.

## Properties That Control Grid Labeling

FontAngle

{normal} | italic | oblique

*Select italic or normal font for all grid labels* — Selects the character slant for all displayed grid labels. 'normal' specifies nonitalic font. 'italic' and 'oblique' specify italic font.

FontColor

ColorSpec | {black}

*Text color for all grid labels* — Sets the color of all displayed grid labels. ColorSpec is a three-element vector specifying an RGB triple or a predefined MATLAB color string (colorspec).

FontName

courier | {helvetica} | symbol | times

*Font family name for all grid labels* — Sets the font for all displayed grid labels. To display and print properly, FontName must be a font that your system supports.

FontSize

scalar in units specified in FontUnits {9}

*Font size* — An integer specifying the font size to use for all displayed grid labels, in units specified by the FontUnits property. The default point size is 9.

FontUnits

{points} | normalized | inches | centimeters | pixels

*Units used to interpret the FontSize property* — When set to normalized, the toolbox interprets the value of FontSize as a fraction of the height of the axes. For example, a normalized FontSize of 0.1 sets the text characters to a font whose height is one-tenth of the axes' height. The default units (points) are equal to 1/72 of an inch.

FontWeight

bold | {normal}

*Select bold or normal font* — The character weight for all displayed grid labels.

LabelFormat

{compass} | signed | none

*Labeling format for grid* — Specifies the format of the grid labels. If `'compass'` is employed (the default), meridian labels are suffixed with an "E" for east and a "W" for west, and parallel labels are suffixed with an "N" for north and an "S" for south. If `'signed'` is used, meridian labels are prefixed with a "+" for east and a "-" for west, and parallel labels are suffixed with a "+" for north and a "-" for south. If `'none'` is selected, straight latitude and longitude numerical values are employed, so western meridian labels and southern parallel labels will have a "-", but no symbol precedes eastern and northern (positive) labels.

LabelRotation

on | {off}

*Label Rotation* — Determines whether the meridian and parallel labels are displayed without rotation (the default) or rotated to align to the graticule. This option is not available for the Globe display.

LabelUnits

{degrees} | dm | dms | radians

*Specify units and formatting for grid labels* — The display of meridian and parallel labels is controlled by the map axes `LabelUnits` property, as described in the following table.

| LabelUnits value | Label format |
| --- | --- |
| `'degrees'` | decimal degrees |
| `'dm'` | degrees/decimal minutes |
| `'dms'` | degrees/minutes/decimal seconds |
| `'radians'` | decimal radians |

`LabelUnits` does not have a default of its own; instead it defaults to the value of `AngleUnits` at the time the map axes is constructed, which itself defaults to degrees. Although you can specify `'dm'` and `'dms'` for `LabelUnits`, these values are not accepted when setting `AngleUnits`.

MeridianLabel

on | {off}

*Toggle display of meridian labels* — Specifies whether the meridian labels are visible or not.

MLabelLocation

```
scalar interval or vector of longitudes
```

*Specify meridians for labeling* — Meridian labels need not coincide with the displayed meridian lines. Labels are displayed at intervals if a scalar in the map axes `MLabelLocation` is entered, starting at the prime meridian and repeating at every interval in both directions. If a vector of longitudes is entered, labels are displayed at those meridians. The default locations coincide with the displayed meridian lines, as specified in the `MLineLocation` property.

MLabelParallel

```
{north} | south | equator | scalar latitude
```

*Specify parallel for meridian label placement* — Specifies the latitude location of the displayed meridian labels. If a latitude is specified, all meridian labels are displayed at that latitude. If `'north'` is specified, the maximum of the `MapLatLimit` is used; if `'south'` is specified, the minimum of the `MapLatLimit` is used. If `'equator'` is specified, a latitude of 0º is used.

MLabelRound

```
integer scalar {O}
```

*Specify significant digits for meridian labels* — Specifies to which power of ten the displayed labels are rounded. For example, if `MLabelRound` is -1, labels are displayed down to the *tenths*. The default value of `MLabelRound` is 0; that is, displayed labels have no decimal places, being rounded to the *ones* column ($10^0$).

ParallelLabel

```
on | {off}
```

*Toggle display of parallel labels* — Specifies whether the parallel labels are visible or not.

PLabelLocation

```
scalar interval or vector of latitudes
```

*Specify parallels for labeling* — Parallel labels need not coincide with the displayed parallel lines. Labels are displayed at intervals if a scalar in the map axes `PLabelLocation` is entered, starting at the equator and repeating at every interval in both directions. If a vector of latitudes is entered, labels are displayed at those parallels. The default locations coincide with the displayed parallel lines, as specified in the `PLineLocation` property.

PLabelMeridian

```
east | {west} | prime | scalar longitude
```

*Specify meridian for parallel label placement* — Specifies the longitude location of the displayed parallel labels. If a longitude is specified, all parallel labels are displayed at that longitude. If `'east'` is specified, the maximum of the `MapLonLimit` is used; if `'west'` is specified, the minimum of the `MapLonLimit` is used. If `'prime'` is specified, a longitude of 0º is used.

PLabelRound

```
integer scalar {0}
```

*Specify significant digits for parallel labels* — Specifies to which power of ten the displayed labels are rounded. For example, if `PLabelRound` is -1, labels are displayed down to the tenths. The default value of `PLabelRound` is 0; that is, displayed labels have no decimal places, being rounded to the ones column ($10^0$).

## See Also

axes | gcm | getm | setm

# axesscale

Resize axes for equivalent scale

## Syntax

```
axesscale
axesscale(hbase)
axesscale(hbase,hother)
```

## Description

`axesscale` resizes all axes in the current figure to have the same scale as the current axes (`gca`). In this context, scale means the relationship between axes *x*- and *y*-coordinates and figure and paper coordinates. When `axesscale` is used, a unit of length in *x* and *y* is printed and displayed at the same size in all the affected axes. The `XLimMode` and `YLimMode` of the axes are set to `'manual'` to prevent autoscaling from changing the scale.

`axesscale(hbase)` uses the axes `hbase` as the reference axes, and rescales the other axes in the current figure.

`axesscale(hbase,hother)` uses the axes `hbase` as the base axes, and rescales only the axes in `hother`.

## Examples

Display the conterminous United States, Alaska, and Hawaii in separate axes in the same figure, with a common scale.

```
% Read state names and coordinates, extract Alaska and Hawaii
states = shaperead('usastatehi', 'UseGeoCoords', true);
statenames = {states.Name};
alaska = states(strcmp('Alaska', statenames));
hawaii = states(strcmp('Hawaii', statenames));

% Create a figure for the conterminous states
f1 = figure; hconus = usamap('conus'); tightmap
```

```
geoshow(states, 'FaceColor', [0.5 1 0.5]);
framem off; gridm off; mlabel off; plabel off
load conus gtlakelat gtlakelon
geoshow(gtlakelat, gtlakelon,...
  'DisplayType', 'polygon', 'FaceColor', 'cyan')
gridm off;

% Working figure for additional calls to usamap
f2 = figure('Visible','off');

halaska = axes; usamap('alaska'); tightmap;
geoshow(alaska, 'FaceColor', [0.5 1 0.5]);
gridm off;
framem off; mlabel off; plabel off; gridm off;
set(halaska,'Parent',f1)

hhawaii = axes; usamap('hawaii'); tightmap;
geoshow(hawaii, 'FaceColor', [0.5 1 0.5]);
gridm off;
framem off; mlabel off; plabel off; gridm off;
set(hhawaii,'Parent',f1)

close(f2)

% Arrange the axes as desired
set(hconus,'Position',[0.1 0.25 0.85 0.6])
set(halaska,'Position',[0.019531 -0.020833 0.2 0.2])
set(hhawaii,'Position',[0.5 0 .2 .2])

% Resize alaska and hawaii axes
axesscale(hconus)
hidem([halaska hhawaii])
```
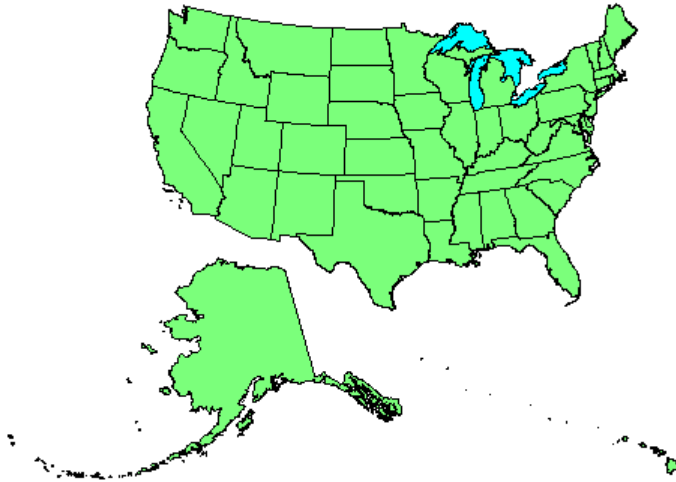
## Limitations

The equivalence of scales holds only as long as no commands are issued that can change the scale of one of the axes. For example, changing the units of the ellipsoid or the scale factor in one of the axes would change the scale.

## More About

### Tips

To ensure the same map scale between axes, use the same ellipsoid and scale factors.

## See Also
```
paperscale
```

# azimuth

Azimuth between points on sphere or ellipsoid

## Syntax

```
az = azimuth(lat1,lon1,lat2,lon2)
az = azimuth(lat1,lon1,lat2,lon2,ellipsoid)
az = azimuth(lat1,lon1,lat2,lon2,units)
az = azimuth(lat1,lon1,lat2,lon2,ellipsoid,units)
az = azimuth(track,...)
```

## Description

`az = azimuth(lat1,lon1,lat2,lon2)` calculates the great circle azimuth from point 1 to point 2, for pairs of points on the surface of a sphere. The input latitudes and longitudes can be scalars or arrays of matching size. If you use a combination of scalar and array inputs, the scalar inputs will be automatically expanded to match the size of the arrays. The function measures azimuths clockwise from north and expresses them in degrees or radians.

`az = azimuth(lat1,lon1,lat2,lon2,ellipsoid)` computes the azimuth assuming that the points lie on the ellipsoid defined by the input `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The default ellipsoid is a unit sphere.

`az = azimuth(lat1,lon1,lat2,lon2,units)` uses the input string *units* to define the angle units of `az` and the latitude-longitude coordinates. Use `'degrees'` (the default value), in the range from 0 to 360, or `'radians'`, in the range from 0 to 2*pi.

`az = azimuth(lat1,lon1,lat2,lon2,ellipsoid,units)` specifies both the `ellipsoid` vector and the units of `az`.

`az = azimuth(track,...)` uses the input string *track* to specify either a great circle or a rhumb line azimuth calculation. Enter `'gc'` for the *track* string (the default value), to obtain great circle azimuths for a sphere or geodesic azimuths for an ellipsoid. (Hint to remember string name: the letters "g" and "c" are in both great circle and geodesic.)

Enter `'rh'` for the *track* string to obtain rhumb line azimuths for either a sphere or an ellipsoid.

## Examples

Find the azimuth between two points on the same parallel, for example, (10ºN, 10ºE) and (10ºN, 40ºE). The azimuth between two points depends on the *track* string selected.

```
% Try the 'gc' track string.
az = azimuth('gc',10,10,10,40)

% Compare to the result obtained from the 'rh' track string.
az = azimuth('rh',10,10,10,40)
```

Find the azimuth between two points on the same meridian, say (10ºN, 10ºE) and (40ºN, 10ºE):

```
% Try the 'gc' track string.
az = azimuth(10,10,40,10)

% Compare to the 'rh' track string.
az = azimuth('rh',10,10,40,10)
```

Rhumb lines and great circles coincide along meridians and the Equator. The azimuths are the same because the paths coincide.

## Alternatives

If you are calculating both the distance and the azimuth, you can call just the `distance` function. The function returns the azimuth as the second output argument. It is unnecessary to call `azimuth` separately.

## More About

### Azimuth

An *azimuth* is the angle at which a smooth curve crosses a meridian, taken clockwise from north. The North Pole has an azimuth of 0º from every other point on the globe. You can calculate azimuths for great circles or rhumb lines.

### Geodesic

A *geodesic* is the shortest distance between two points on a curved surface, such as an ellipsoid.

### Great Circle

A *great circle* is a type of geodesic that lies on a sphere. It is the intersection of the surface of a sphere with a plane passing through the center of the sphere. For great circles, the azimuth is calculated at the starting point of the great circle path, where it crosses the meridian. In general, the azimuth along a great circle is not constant. For more information, see "Great Circles".

### Rhumb Line

A *rhumb line* is a curve that crosses each meridian at the same angle. For rhumb lines, the azimuth is the *constant* angle between true north and the entire rhumb line passing through the two points. For more information, see "Rhumb Lines".

### Algorithms

## Azimuths over Long Geodesics

Azimuth calculations for geodesics degrade slowly with increasing distance and can break down for points that are nearly antipodal or for points close to the Equator. In addition, for calculations on an ellipsoid, there is a small but finite input space. This space consists of pairs of locations in which both points are nearly antipodal *and* both points fall close to (but not precisely on) the Equator. In such cases, you will receive a warning and `az` will be set to NaN for the "problem pairs."

## Eccentricity

Geodesic azimuths on an ellipsoid are valid only for small eccentricities typical of the Earth (for example, 0.08 or less).

## See Also
`distance` | `elevation` | `reckon` | `track` | `track1` | `track2`

# bufferm

Buffer zones for latitude-longitude polygons

## Syntax

```
[latb,lonb] = bufferm(lat,lon,bufwidth)
[latb,lonb] = bufferm(lat,lon,bufwidth,direction)
[latb,lonb] = bufferm(lat,lon,dist,direction,npts)
```

## Description

`[latb,lonb] = bufferm(lat,lon,bufwidth)` computes the buffer zone around a line or polygon. If the vectors `lat` and `lon`, in units of degrees, define a line, then `latb` and `lonb` define a polygon that contains all the points that fall within a certain distance, `bufwidth`, of the line. `bufwidth` is a scalar specified in degrees of arc along the surface. If the vectors `lat` and `lon` define a polygon, then `latb` and `lonb` define a region that contains all the points exterior to the polygon that fall within `bufwidth` of the polygon.

`[latb,lonb] = bufferm(lat,lon,bufwidth,direction)` uses the optional string *direction* to specify whether the buffer zone is inside `'in'`or `'out'` of the polygon. A third option, `'outPlusInterior'`, returns the union of an exterior buffer (as would be computed using `'out'`) with the interior of the polygon. If you do not supply a *direction* string, `bufferm` uses `'out'` as the default and returns a buffer zone outside the polygon. If you supply `'in'` as the *direction* string, `bufferm` returns a buffer zone inside the polygon. If you are finding the buffer zone around a line, `'out'` is the only valid option.

`[latb,lonb] = bufferm(lat,lon,dist,direction,npts)` controls the number of points used to construct circles about the vertices of the polygon. A larger number of points produces smoother buffers, but requires more time. If `npts` is omitted, 13 points per circle are used.
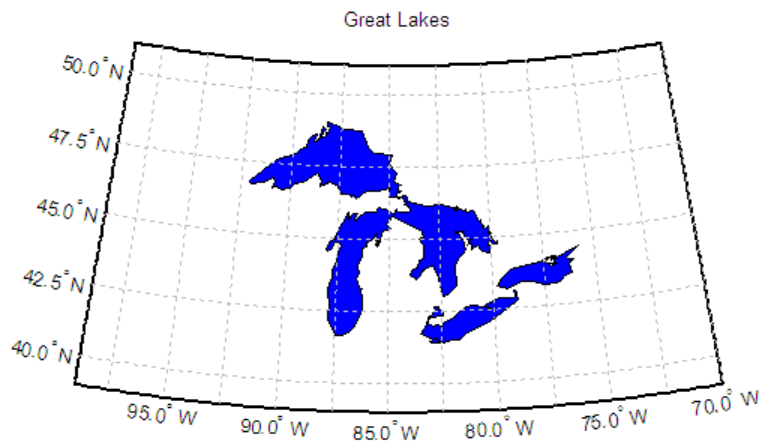
## Examples

Display buffer zones inside and outside the Great Lakes:

```
% Display a simplified version of the five polygons that
% represent the Great Lakes.
load conus
tol = 0.05;
[latr, lonr] = reducem(gtlakelat, gtlakelon, tol);
figure('Color','w')
ax = usamap({'MN','NY'});
setm(ax,'MLabelLocation',5)
geoshow(latr, lonr, 'DisplayType', 'polygon', ...
   'FaceColor', 'blue')
title('Great Lakes')
```
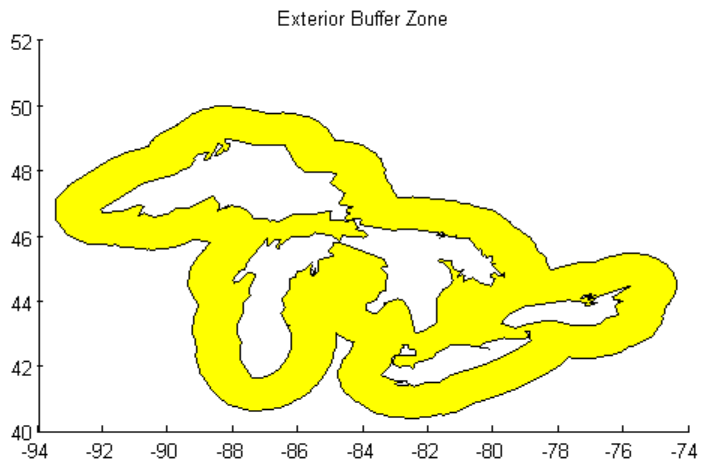


```
% Set the buffer width and display a buffer zone outside
% the lakes.
figure;
bufwidth = 1;
[latb, lonb] = bufferm(latr, lonr, bufwidth);
geoshow(latb, lonb, 'DisplayType', 'polygon', ...
   'FaceColor', 'yellow')
title('Exterior Buffer Zone')
```

Exterior Buffer Zone

```
% Display a buffer zone inside the polygon.
figure;
[lati, loni] = bufferm(latr, lonr, 0.3*bufwidth, 'in');
geoshow(lati, loni, 'DisplayType', 'polygon', ...
    'FaceColor', 'magenta')
title('Interior Buffer Zone')
```

Interior Buffer Zone

```
% Display the Great Lakes with interior and exterior
% buffer zones on a backdrop of neighboring states.
figure('Color','w')
ax = usamap({'MN','NY'});
setm(ax,'MLabelLocation',5)
geoshow(latb, lonb, 'DisplayType', 'polygon', 'FaceColor', 'yellow')
geoshow(latr, lonr, 'DisplayType', 'polygon', 'FaceColor', 'blue')
geoshow(lati, loni, 'DisplayType', 'polygon', 'FaceColor', 'magenta')
geoshow(uslat, uslon)
geoshow(statelat, statelon)
title('Great Lakes and Buffer Zones with Neighboring States')
```
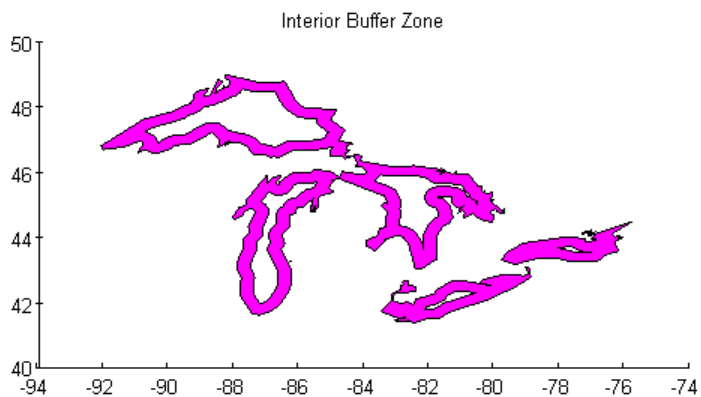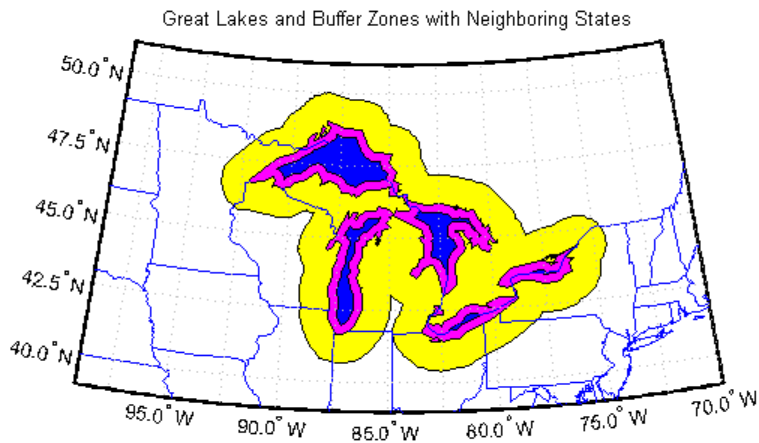
Great Lakes and Buffer Zones with Neighboring States

```
% Example using 'outPlusInterior' option:
bufWidth = 0.5;
[latz, lonz] = bufferm(latr, lonr, bufWidth,'outPlusInterior');
figure
geoshow(latz,lonz,'DisplayType','polygon','FaceColor','yellow')
title('Exterior Buffer Zone including Polygon Interior');
```

Exterior Buffer Zone including Polygon Interior
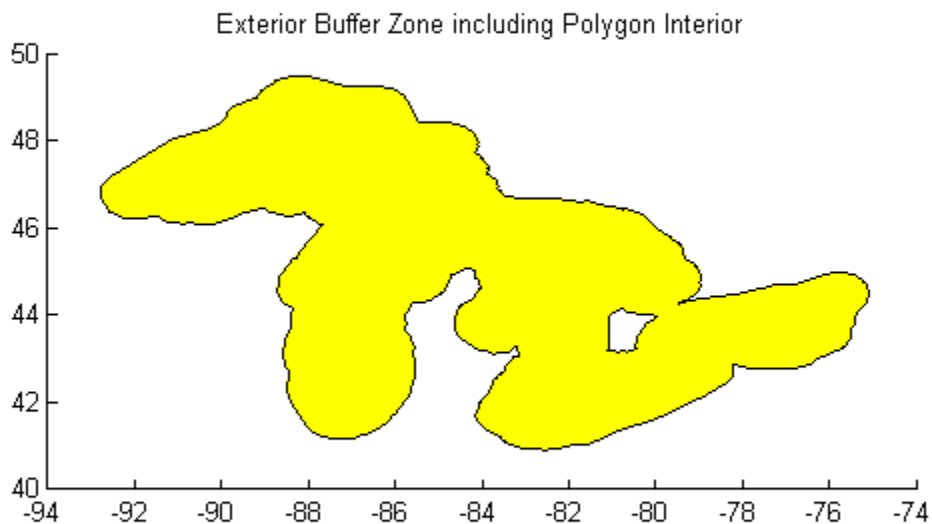


# More About

### Tips

Close all polygons before processing them with `bufferm`. If a polygon is not closed, `bufferm` assumes it is a line.

## See Also
`polybool`

# bufgeoquad

Expand limits of geographic quadrangle

## Syntax

```
[latlim,lonlim] = bufgeoquad(latlim,lonlim,buflat,buflon)
```

## Description

`[latlim,lonlim] = bufgeoquad(latlim,lonlim,buflat,buflon)` returns an expanded version of the geographic quadrangle defined by `latlim` and `lonlim`.

## Examples

### Bounding Quadrangle for U.S.

Bounding quadrangle for the Conterminous United States, buffered 2 degrees to the north and south and 3 degrees to the east and west.

Load data and expand the limits of the quadrangle.

```
conus = load('conus.mat');
[latlim, lonlim] = geoquadline(conus.uslat,conus.uslon);
[latlim,lonlim] = bufgeoquad(latlim,lonlim,2,3)

latlim =

   23.1200   51.3800


lonlim =
```

```
-127.7200   -63.9700
```

# Input Arguments

### `latlim` — Latitude limits
1-by-2 vector

Latitude limits of a geographic quadrangle, specified as a 1-by-2 vector of the form `[southern_limit northern_limit]`, with latitudes in degrees. The two elements must be in ascending order, and lie in the closed interval [-90 90].

Data Types: `single` | `double`

### `lonlim` — Longitude limits
1-by-2 vector

Longitude limits of a geographic quadrangle, specified as a 1-by-2 vector of the form `[western_limit eastern_limit]`, with longitudes in degrees. The two limits need not be in numerical ascending order.

Data Types: `single` | `double`

### `buflat` — Latitude buffer size
nonnegative scalar

Latitude buffer size, specified as a nonnegative scalar, in units of degrees.

Data Types: `double`

### `buflon` — Longitude buffer size
nonnegative scalar

Longitude buffer size, specified as a nonnegative scalar, in units of degrees.

Data Types: `double`

# Output Arguments

### `latlim` — Latitude limits
1–by-2 vector

Latitude limits of a geographic quadrangle, returned as a 1-by-2 vector of the form `[southern_limit northern_limit]`, in units of degrees. The elements are in ascending order, and both lie in the closed interval [-90 90].

### `lonlim` — Longitude limits
1–by-2 vector

Longitude limits of a geographic quadrangle, returned as a 1-by-2 vector of the form `[western_limit eastern_limit]`, in units of degrees. The limits are wrapped to the interval [-180 180]. They are not necessarily in numerical ascending order.

## See Also
geoquadline | geoquadpt | outlinegeoquad

# camposm

Set camera position using geographic coordinates

## Syntax

```
camposm(lat,long,alt)
[x,y,z] = camposm(lat,long,alt)
```

## Description

camposm(lat,long,alt) sets the axes CameraPosition property of the current map axes to the position specified in geographic coordinates. The inputs lat and long are assumed to be in the angle units of the current map axes.

[x,y,z] = camposm(lat,long,alt) returns the camera position in the projected Cartesian coordinate system.

## Examples

Look at northern Australia from a point south and one Earth radius above New Zealand:

```
figure
axesm('globe','galt',0)
gridm('glinestyle','-')
load topo
geoshow(topo,topolegend,'DisplayType','texturemap');
demcmap(topo)
camlight;
material(0.6*[ 1 1 1])
plat = -50; plon = 160;
tlat = -10; tlon = 130;
camtargm(tlat,tlon,0);
camposm(plat,plon,1);
camupm(tlat,tlon)
set(gca,'CameraViewAngle',75)
land = shaperead('landareas.shp','UseGeoCoords',true)
linem([land.Lat],[land.Lon])
```

```
axis off
```



## See Also
```
camtargm | camupm | campos | camva
```

# camtargm

Set camera target using geographic coordinates

## Syntax

```
camtargm(lat,long,alt)
[x,y,z] = camtargm(lat,long,alt)
```

## Description

camtargm(lat,long,alt) sets the axes `CameraTarget` property of the current map axes to the position specified in geographic coordinates. The inputs `lat` and `long` are assumed to be in the angle units of the current map axes.
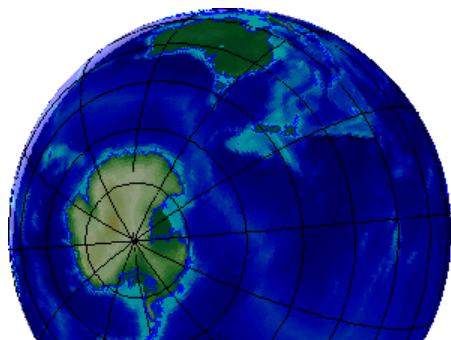
`[x,y,z] = camtargm(lat,long,alt)` returns the camera target in the projected Cartesian coordinate system.

## Examples

Look down the spine of the Andes from a location three Earth radii above the surface:

```
figure
axesm('globe','galt',0)
gridm('glinestyle','-')
load topo
geoshow(topo,topolegend,'DisplayType','texturemap');
demcmap(topo)
lightm(-80,-180);
material(0.6*[ 1 1 1])
plat = 10; plon = -65;
tlat = -30; tlon = -70;
camtargm(tlat,tlon,0);
camposm(plat,plon,3);
camupm(tlat,tlon);
camva(20)
set(gca,'CameraViewAngle',30)
land = shaperead('landareas.shp','UseGeoCoords',true)
```

```
linem([land.Lat],[land.Lon])
axis off
```



## See Also
camposm | camupm | camtarget | camva

# camupm

Set camera up vector using geographic coordinates

## Syntax

```
camupm(lat,long)
[x,y,z] = camupm(lat,long)
```

## Description

camupm(lat,long) sets the axes CameraUpVector property of the current map axes to the position specified in geographic coordinates. The inputs lat and long are assumed to be in the angle units of the current map axes.
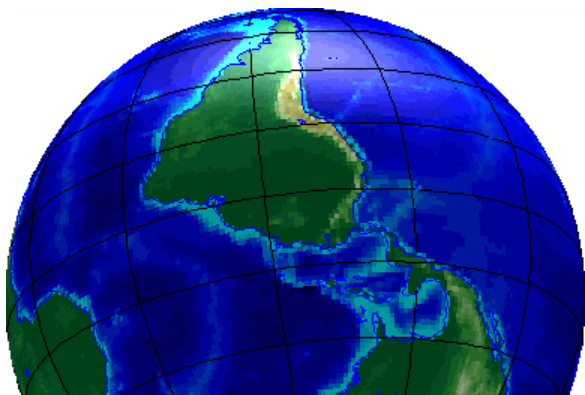
[x,y,z] = camupm(lat,long) returns the camera position in the projected Cartesian coordinate system.

## Examples

Look at northern Australia from a point south of and one Earth radius above New Zealand. Set CameraUpVector to the antipode of the camera target for that *down under* view:

```
figure
axesm('globe','galt',0)
gridm('glinestyle','-')
load topo
geoshow(topo,topolegend,'DisplayType','texturemap');
demcmap(topo)
camlight;
material(0.6*[ 1 1 1])
plat = -50; plon = 160;
tlat = -10; tlon = 130;
[alat,alon] = antipode(tlat,tlon);
camtargm(tlat,tlon,0);
camposm(plat,plon,1);
```

```
camupm(alat,alon)
set(gca,'CameraViewAngle',80)
land = shaperead('landareas.shp','UseGeoCoords',true)
linem([land.Lat],[land.Lon])
axis off
```
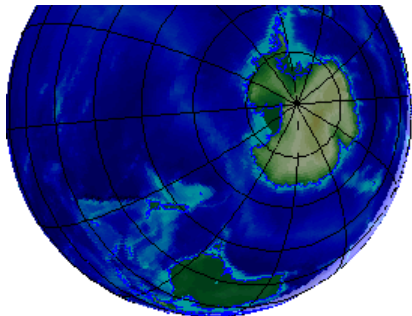


## See Also
```
camtargm | camposm | camup | camva
```

# cart2grn

Transform projected coordinates to Greenwich system

## Syntax

```
[lat,lon,alt] = cart2grn
[lat,lon,alt] = cart2grn(hndl)
[lat,lon,alt] = cart2grn(hndl,mstruct)
```

## Description

When objects are projected and displayed on map axes, they are plotted in Cartesian coordinates appropriate for the selected projection. This function transforms those coordinates back into the Greenwich frame, in which longitude is measured positively East from Greenwich (longitude 0), England and negatively West from Greenwich.

`[lat,lon,alt] = cart2grn` returns the latitude, longitude, and altitude data in geographic coordinates of the current map object, removing any clips or trims introduced during the display process from the output data.

`[lat,lon,alt] = cart2grn(hndl)` specifies the displayed map object desired with its handle `hndl`. The default handle is `gco`.

`[lat,lon,alt] = cart2grn(hndl,mstruct)` specifies the map structure associated with the object. The map structure of the current axes is the default.

## See Also

gcm | mfwdtran | minvtran | project

# changem

Substitute values in data array

## Syntax

```
mapout = changem(Z,newcode,oldcode)
```

## Description

`mapout = changem(Z,newcode,oldcode)` returns a data grid `mapout` identical to the input data grid, except that each element of `Z` with a value contained in the vector `oldcode` is replaced by the corresponding element of the vector `newcode`.

`oldcode` is 0 (scalar) by default, in which case `newcode` must be scalar. Otherwise, `newcode` and `oldcode` must be the same size.

## Examples

Invent a map:

```
A = magic(3)

A =
     8     1     6
     3     5     7
     4     9     2
```

Replace instances of 8 or 9 with 0s:

```
B = changem(A,[0 0],[9 8])
B =
     0     1     6
     3     5     7
     4     0     2
```

# circcirc

Intersections of circles in Cartesian plane

## Syntax

```
[xout,yout] = circcirc(x1,y1,r1,x2,y2,r2)
```

## Description

`[xout,yout] = circcirc(x1,y1,r1,x2,y2,r2)` finds the points of intersection (if any), given two circles, each defined by center and radius in $x$-$y$ coordinates. In general, two points are returned. When the circles do not intersect or are identical, `NaNs` are returned.

When the two circles are tangent, two identical points are returned. All inputs must be scalars.

### See Also

`linecirc`

# clabelm

Add contour labels to map contour display

## Syntax

```
clabelm(C)
clabelm(C,h)
clabelm(C,v)
clabelm(C,h,v)
clabelm( ___ ,Name,Value)

clabelm(C,'manual')
clabelm(C,h,'manual')

text_handles = clabel( ___ )
```

## Description

clabelm(C) labels all contours displayed in the current contour plot. Labels are upright and displayed with '+' symbols. clabelm randomly selects label positions.

clabelm(C,h) rotates the labels and inserts them in the contour lines. This syntax inserts only those labels that fit within the contour, depending on the size of the contour.

clabelm(C,v) labels only the contour levels specified by the vector, v.

clabelm(C,h,v) labels only the contour levels specified by vector v, rotates the labels, and inserts them in the contour lines.

clabelm( ___ ,Name,Value) specifies the text object properties and the 'LabelSpacing' contourgroup property, using one or more Name,Value pair arguments, in addition to any of the input arguments in previous syntaxes.

clabelm(C,'manual') places contour labels at locations you select with a mouse. Click the mouse or press the space bar to label the contour closest to the center of

the crosshair. Press the **Return** key while the cursor is within the figure window to terminate labeling.

`clabelm(C,h,'manual')` places contour labels at locations you select with a mouse. Press the **Return** key while the cursor is within the figure window to terminate labeling. The labels are rotated and inserted in the contour lines.

`text_handles = clabel( ___ )` additionally returns an array containing the handles of the text objects created, using any of the input arguments in the previous syntaxes. If you call `clabel` without the h argument, `text_handles` also contains the handles of line objects used to create the '+' symbols.

# Input Arguments

### C — Contour matrix
2-by-`n` matrix

Contour matrix containing the data that defines the contour lines. `C` is returned by the `contourm`, `contourfm`, or `contour3m` functions.

### h — Handle to the contourgroup object

Handle to the contourgroup object returned by the `contourm`, `contourfm`, or `contour3m` functions.

### v — Contour level values
vector

Contour level values, specified as a row or column vector of individual values.

Example: `[0,10,20]`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

### `'LabelSpacing'` — Spacing between labels
144 (default) | scalar

Spacing between labels on each contour line, specified as the comma-separated pair consisting of `'LabelSpacing'` and a scalar. Specify the label spacing in points, where 1 point = $^{1}/_{72}$ inch.

Example: `'LabelSpacing',72`

## Output Arguments

### `text_handles` — Handles of text objects

Handles of the text objects that `clabelm` creates. The `UserData` properties of the text objects contain the contour values displayed.

If you call `clabelm` without the `h` argument, `text_handles` also contains the handles of line objects used to create the `'+'` symbols.

## Examples

### Add Labels to a Contour Map

Generate a contour map and obtain the contour matrix, `C`, and the handle to the contourgroup object, `h`. Then display labels on the contour map.

```
load geoid
axesm miller
framem
tightmap
[C,h] = contourm(geoid,geoidlegend,-100:50:80);
clabelm(C,h)
```
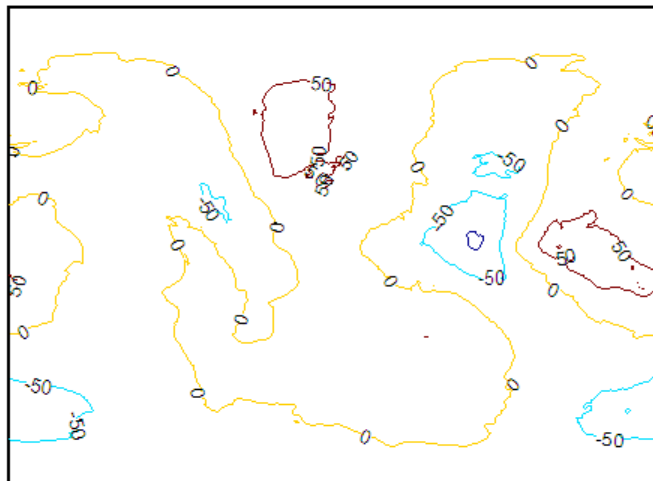
## See Also

clegendm | contourm | contour3m | contourfm | clabel

# clegendm

Add legend labels to map contour display

## Syntax

```
clegendm(cs,h)
clegendm(cs,h,loc)
clegendm(...,unitstr)
clegendm(...,strings)
h = clegendm(...)
```

## Description

clegendm(cs,h) adds a legend specifying the contour line heights to the current map contour plot.

clegendm(cs,h,loc) places the legend in the specified location.

clegendm(...,unitstr) appends unitstr to each entry in the legend.

clegendm(...,strings) uses the strings specified in strings to label the legend. strings must have the same number of entries as the line children of h.

h = clegendm(...) returns the handle to the legend object created.

## Input Arguments

**cs**

Contour matrix output from contourm, contour3m, or contourfm

**h**

Object handle output from contourm, contour3m, or contourfm

**loc**

Any of the following integers, with the indicated placement:

| Integer | Placement |
|---------|-----------|
| 0 | Automatic placement (default) |
| 1 | Upper right corner |
| 2 | Upper left corner |
| 3 | Lower left corner |
| 4 | Lower right corner |
| -1 | To the right of the plot |

**unitstr**

Character string appended to each entry in the legend

**strings**

Cell array of strings used to label the legend

## Output Arguments

**h**

Handle to the legend object

## Examples

Create a legend in the lower right-hand corner with a unit string indicating that the contour elevations are in meters:

```
load topo
R = georasterref('RasterSize', size(topo), ...
   'Latlim', [-90 90], 'Lonlim', [O 360]);
figure('Color','w'); worldmap('world')
[c,h] = contourm(topo, R, -6000:1500:6000);
clegendm(c,h,4,' m')
```
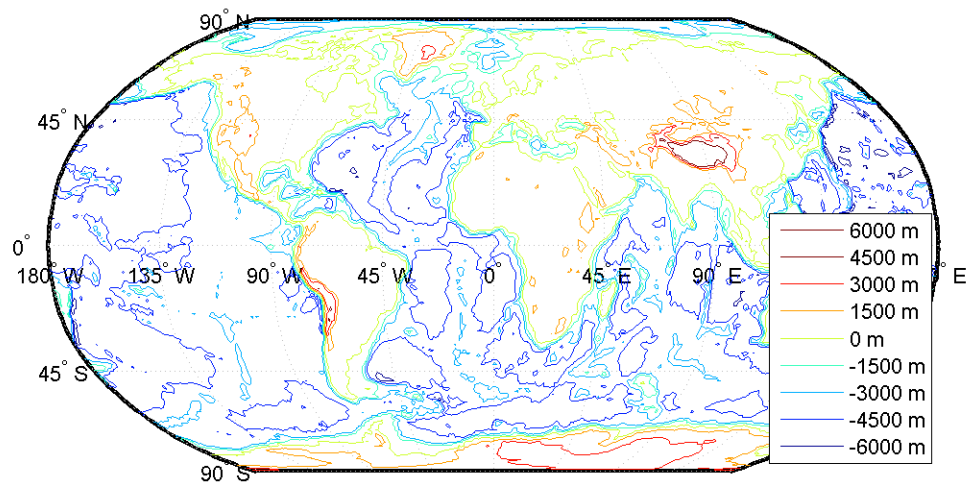
## See Also

`clabelm` | `contourm` | `contour3m` | `contourfm` | `contourc` | `contourcbar`

# clipdata

Clip data at `+/-pi` in longitude, `+/-pi` in latitude

## Syntax

```
[lat,long,splitpts] = clipdata(lat,long,'object')
```

## Description

`[lat,long,splitpts] = clipdata(lat,long,'object')` inserts NaNs at the appropriate locations in a map object so that a displayed map is clipped at the appropriate edges. It assumes that the clipping occurs at `+/- pi/2` radians in the latitude (*y*) direction and `+/- pi` radians in the longitude (*x*) direction.

The input data must be in radians and properly transformed for the particular aspect and origin so that it fits in the specified clipping range.

The output data is in radians, with clips placed at the proper locations. The output variable `splitpts` returns the row and column indices of the clipped elements (columns 1 and 2 respectively). These indices are necessary to restore the original data if the map parameters or projection are ever changed.

Allowable object strings are:

- `surface` for clipping graticules
- `light` for clipping lights
- `line` for clipping lines
- `patch` for clipping patches
- `text` for clipping text object location points
- `point` for clipping point data
- `none` to skip all clipping operations

## See Also
`trimdata` | `undoclip` | `undotrim`

# clma

Clear current map axes

## Syntax

```
clma
clma all
clma purge
```

## Description

`clma` deletes all displayed map objects from the current map axes but leaves the frame if it is displayed.

`clma all` deletes all displayed map objects, including the frame, but it leaves the map structure intact, thereby retaining the map axes.

`clma purge` removes the map definition from the current axes, but leaves all objects projected on the axes intact.

### See Also
`cla` | `clmo` | `handlem` | `hidem` | `namem` | `showm` | `tagm`

# clmo

Clear specified graphics objects from map axes

## Syntax

```
clmo
clmo(handle)
clmo(object)
```

## Description

`clmo` deletes all displayed graphics objects on the current axes.

`clmo(handle)` deletes those objects specified by their handles.

`clmo(object)` deletes those objects with names identical to the input string. This can be any string recognized by the `handlem` function, including entries in the `Tag` property of each object, or the object `Type` if the `Tag` property is empty.

### See Also
clma | handlem | hidem | namem | showm | tagm

# closePolygonParts

Close all rings in multipart polygon

## Syntax

```
[xdata, ydata] = closePolygonParts(xdata, ydata)
[lat, lon] = closePolygonParts(lat, lon, angleunits)
```

## Description

`[xdata, ydata] = closePolygonParts(xdata, ydata)` ensures that each ring in a multipart (`NaN`-separated) polygon is "closed" by repeating the start point at the end of each ring, unless the start and end points are already identical. Coordinate vectors `xdata` and `ydata` must match in size and have identical `NaN` locations.

`[lat, lon] = closePolygonParts(lat, lon, angleunits)` works with latitude-longitude data and accounts for longitude wrapping with a period of `360` if *angleunits* is `'degrees'` and `2*pi` if *angleunits* is `'radians'`. For a ring to be considered closed, the latitudes of its first and last vertices must match exactly, but their longitudes need only match modulo the appropriate period. Such rings are returned unaltered.

## Examples

### Close Polygon in Plane Coordinates

Create two vectors of planar coordinates.

```
xOpen = [1 0 2 NaN 0.5 0.5 1 1];
yOpen = [0 1 2 NaN 0.8 1 1 0.8];
```

Create a closed polygon from these coordinates.

```
[xClosed, yClosed] = closePolygonParts(xOpen,yOpen)
```

```
xClosed =
```

```
  Columns 1 through 7

    1.0000         0    2.0000    1.0000       NaN    0.5000    0.5000

  Columns 8 through 10

    1.0000    1.0000    0.5000


yClosed =

  Columns 1 through 7

         0    1.0000    2.0000         0       NaN    0.8000    1.0000

  Columns 8 through 10

    1.0000    0.8000    0.8000
```

Display all variables.

```
whos
```

```
  Name         Size              Bytes  Class     Attributes

  xClosed      1x10                 80  double
  xOpen        1x8                  64  double
  yClosed      1x10                 80  double
  yOpen        1x8                  64  double
```

### Close Polygon in Latitude-Longitude Coordinates

Load coastline data from MAT-file.

```
load coastlines
```

Construct a two-part polygon based on the coastlines data. The first ring is Antarctica. The longitude of its first vertex is -180 and the longitude of its last vertex is 180. The second ring is a small island from which the last vertex, a replica of the first vertex, is removed.

```
[latparts, lonparts] = polysplit(coastlat, coastlon);
```

```
latparts{2}(end) = [];
lonparts{2}(end) = [];
latparts(3:end) = [];
lonparts(3:end) = [];
[lat, lon] = polyjoin(latparts, lonparts);
```

Examine how `closePolygonParts` treats the two rings. In both cases, the first and last vertices differ. However, Antarctica remains unchanged while the small island is closed back up.

```
[latClosed, lonClosed] = closePolygonParts(lat, lon, 'degrees');
[latpartsClosed, lonpartsClosed] = polysplit(latClosed, lonClosed);
lonpartsClosed{1}(end) - lonpartsClosed{1}(1)   % Result is 360
lonpartsClosed{2}(end) - lonpartsClosed{2}(1)   % Result is 0


ans =

   360


ans =

     0
```

## See Also
`isshapemultipart` | `removeextranseparators`

# colorui

Interactively define RGB color

---

**Note:** colorui will be removed in a future release. Use uisetcolor instead.

---

## Syntax

```
c = colorui
c = colorui(InitClr)
c = colorui(InitClr,FigTitle)
```

## Description

c = colorui will create an interface for the definition of an RGB color triplet. On Windows® platforms, colorui will produce the same interface as uisetcolor. On other machines, colorui produces a platform-independent dialog for specifying the color values.

c = colorui(InitClr) will initialize the color value to the RGB triple given in initclr.

c = colorui(InitClr,FigTitle) will use the string in FigTitle as the window label.

The output value c is the selected RGB triple if the **Accept** or **OK** button is pushed. If the user presses **Cancel**, then the output value is set to 0.

### See Also
uisetcolor

# combntns

All possible combinations of set of values

## Compatibility

`combntns` will be removed in a future release. Use `nchoosek` instead.

## Syntax

```
combos = combntns(set,subset)
```

## Description

`combos = combntns(set,subset)` returns a matrix whose rows are the various combinations that can be taken of the elements of the vector `set` of length `subset`. Many combinatorial applications can make use of a vector `1:n` for the input set to return generalized, indexed combination subsets.

The `combntns` function provides the combinatorial subsets of a set of numbers. It is similar to the mathematical expression *a choose b*, except that instead of the number of such combinations, the actual combinations are returned. In combinatorial counting, the ordering of the values is not significant.

The numerical value of the mathematical statement *a choose b* is `size(combos,1)`.

## Examples

How can the numbers 1 to 5 be taken in sets of three (that is, what is *5 choose 3*)?

```
combos = combntns(1:5,3)

combos =
     1     2     3
     1     2     4
```

```
     1      2      5
     1      3      4
     1      3      5
     1      4      5
     2      3      4
     2      3      5
     2      4      5
     3      4      5
size(combos,1)  % "5 choose 3"

ans =
    10
```

Note that if a value is repeated in the input vector, each occurrence is treated as independent:

```
combos = combntns([2 2 5],2)

combos =
     2      2
     2      5
     2      5
```

# More About

## Tips

This is a recursive function.

# comet3m

Project 3-D comet plot on map axes

## Compatibility

`comet3m` will be removed in a future release. Use the following instead:

```
[x,y,z] = mfwdtran(lat,lon,z);
comet3(x,y,z,p)
```

## Syntax

```
comet3m(lat,lon,z)
comet3m(lat,lon,z,p)
```

## Description

`comet3m(lat,lon,z)` traces a comet plot through the points specified by the input latitude, longitude, and altitude vectors.

`comet3m(lat,lon,z,p)` specifies a comet body of length `p*length(lat)`. The input `p` is `0.1` by default.

A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

## Examples

Create a 3-D comet plot of the coastlines data:

```
load coast
z = (1:length(lat))'/3000;
axesm miller
framem; gridm;
```

```
setm(gca,'galtitude',max(z)+.5)
view(3)
comet3m(lat,long,z,0.01)
```

## See Also

comet3 | cometm

# cometm

Project 2-D comet plot on map axes

## Compatibility

`cometm` will be removed in a future release. Use the following instead:

```
[x,y] = mfwdtran(lat,lon);
comet(x,y,p)
```

## Syntax

```
cometm(lat,lon)
cometm(lat,lon,p)
```

## Description

`cometm(lat,lon)` traces a comet plot through the points specified by the input latitude and longitude vectors.

`cometm(lat,lon,p)` specifies a comet body of length `p*length(lat)`. The input `p` is `0.1` by default.

A comet plot is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

## Examples

Create a comet plot of the coastlines data:

```
load coast
axesm miller
framem
cometm(lat,long,0.01)
```

## See Also

comet | comet3m

# contour3m

Project 3-D contour plot of map data

## Description

The contour3m function is the same as the contourm function except that the lines for each contour level are drawn in their own horizontal plane, at the *z*-coordinate equal to the value of that level.

## Examples

### Contour EGM96 Geoid Heights as 3-D Surface

This example shows how to contour the EGM96 geoid heights in an ordinary axes as a 3-D surface with 50 levels and set the contour line color to black.

Load the geoid.

```
figure('Color','white')
load geoid
contour3m(geoid,geoidrefvec,50,'LineColor','black');
```
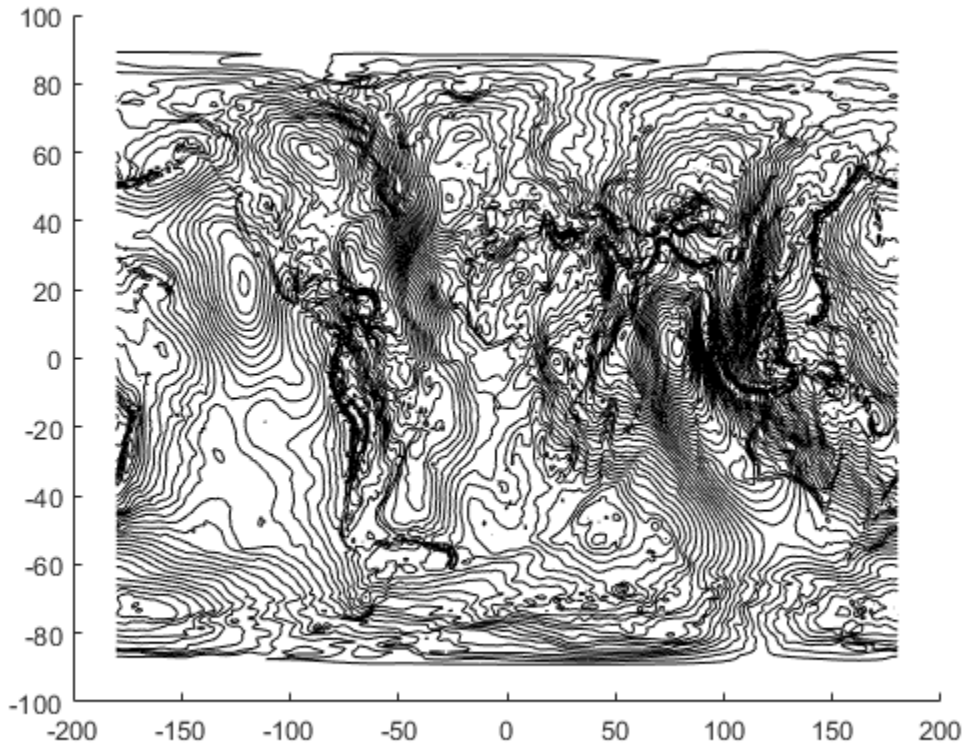
Add the geoid as a surface.

```
hold on
geoshow(geoid,geoidrefvec,'DisplayType','surface')
```

**1-109**
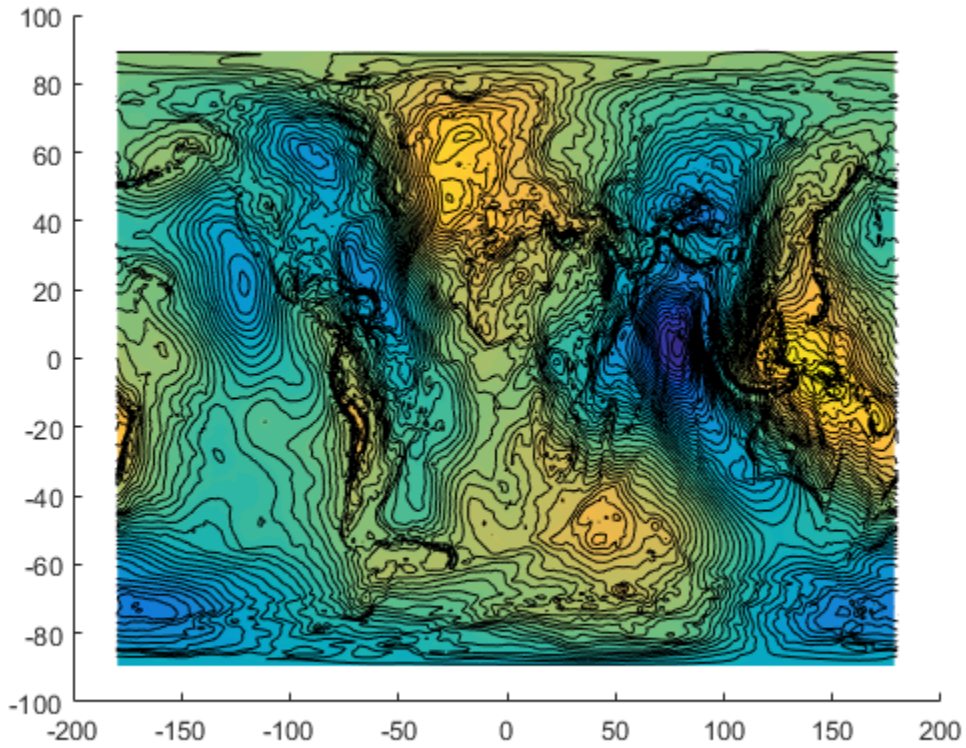
Add a title.

```
title('EGM96 Global Geoid Heights with 50 Contour Levels');
```

**EGM96 Global Geoid Heights with 50 Contour Levels**
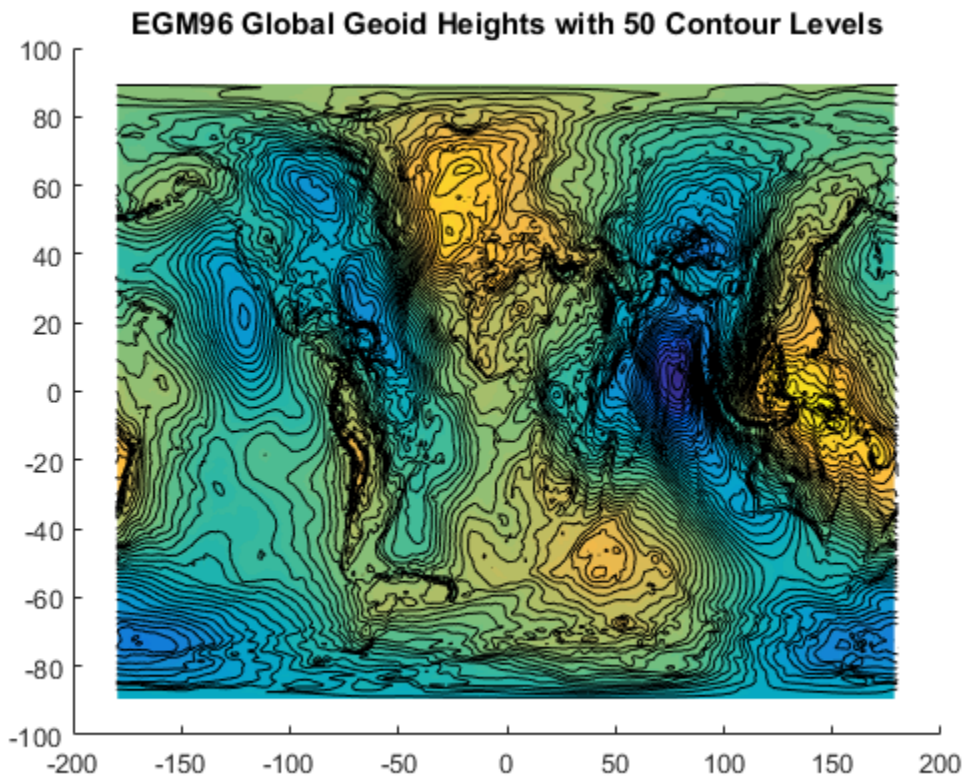
View in 3-D.

```
view(3)
```

**EGM96 Global Geoid Heights with 50 Contour Levels**
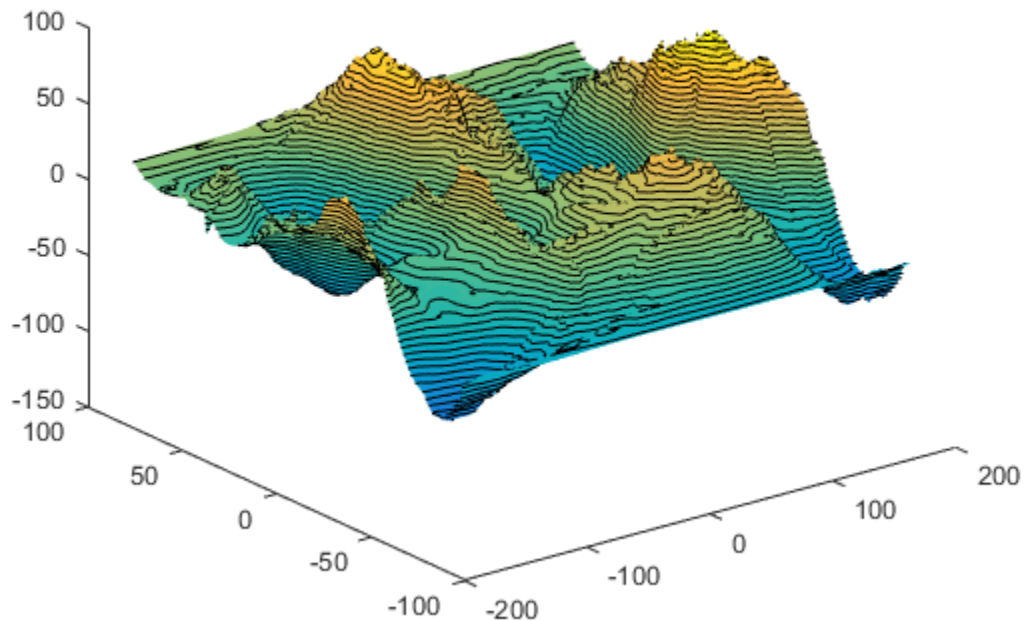


**Contour topography and bathymetry of South Asia**

This example shows how to contour in a map axes the topography and bathymetry of South Asia and the northern Indian Ocean with a contour interval of 500 meters.

Load topo.

```
load topo
latlim = [ 0  50];
lonlim = [35 115];
[Z, refvec] = maptrims(topo, topolegend, latlim, lonlim);
figure('Color','white')
axesm('lambertstd','MapLatLimit', latlim, 'MapLonLimit', lonlim)
tightmap; axis off
```

```
contour3m(Z,refvec,'black','LevelStep',500)
```
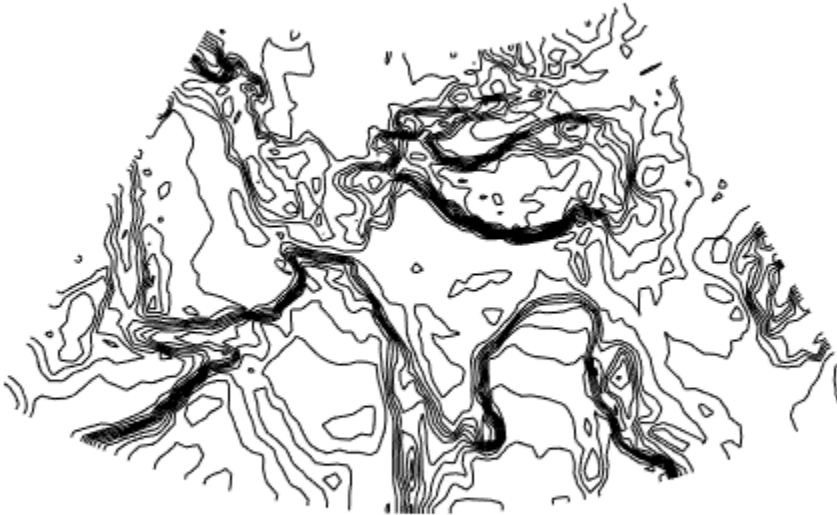


Add the geoid as a surface and set the colormap.

```
geoshow(Z,refvec,'DisplayType','surface')
demcmap(Z)
```
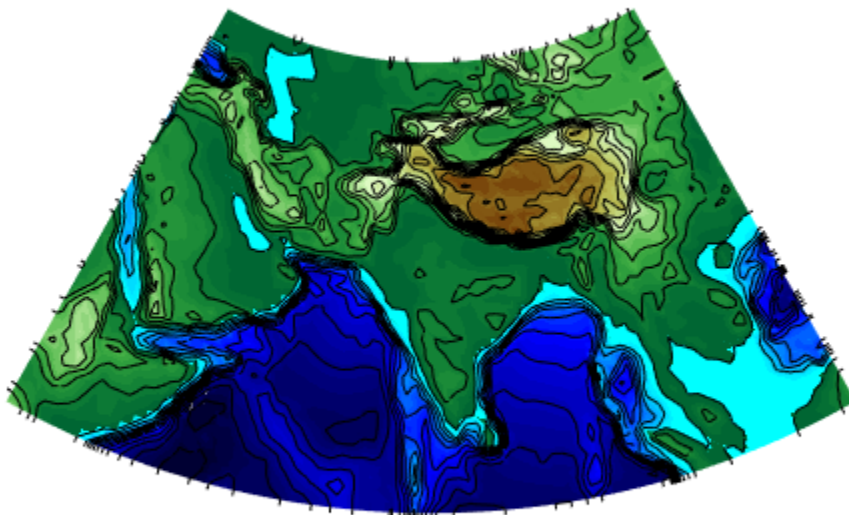
Add a title.

```
title({'South Asia Topography and Bathymetry', ...
    'with 500 m Contours'});
```

**South Asia Topography and Bathymetry with 500 m Contours**
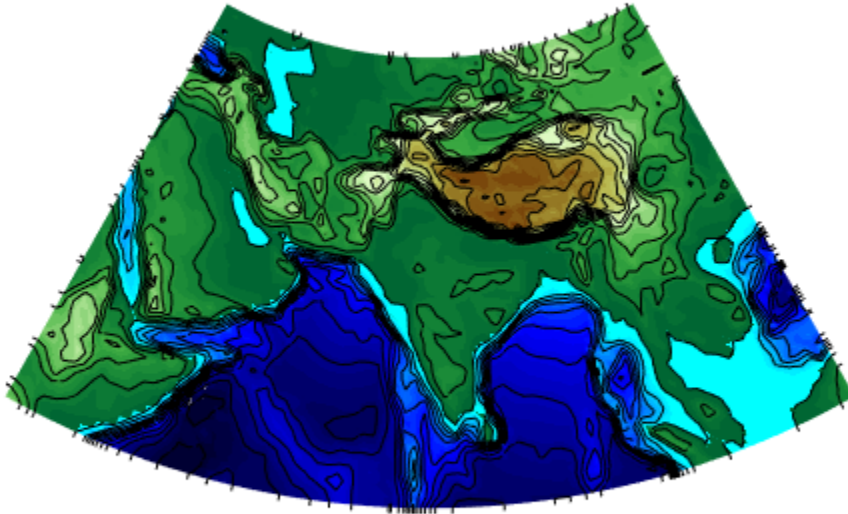
View in 3-D

```
set(gca,'DataAspectRatio',[1 1 40000])
view(3)
```

South Asia Topography and Bathymetry
with 500 m Contours
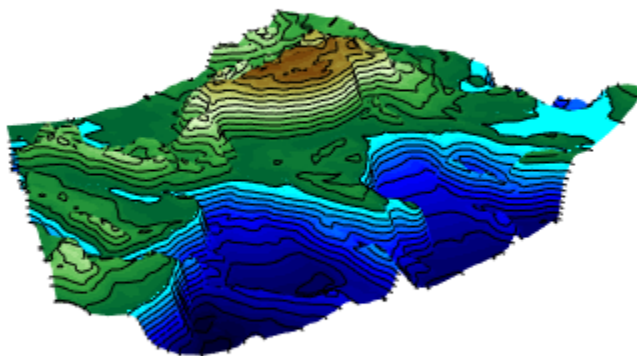


# More About

### Tips

- If you use `contour3m` with the `globe` map display, the `contour3m` function warns. Be careful to scale the input data relative to the radius of your reference sphere.

## See Also

`clabel` | `clabelm` | `clegendm` | `contour` | `contourm` | `contourfm` | `geoshow` | `plot` | `contour3`

# contourcbar

Color bar for filled contour map display

## Syntax

```
H = contourcbar(...)
```

## Description

`H = contourcbar(...)` creates a color bar associated with a filled contour display created with `contourfm`, `contourm`, `contour3m`, or `geoshow`. It supports the same syntax and usage options as the MATLAB function `colorbar`.

## Examples

**Add colorbar to map showing topography of North America**

Create world map.

```
figure('Color','white')
worldmap('north america')
```
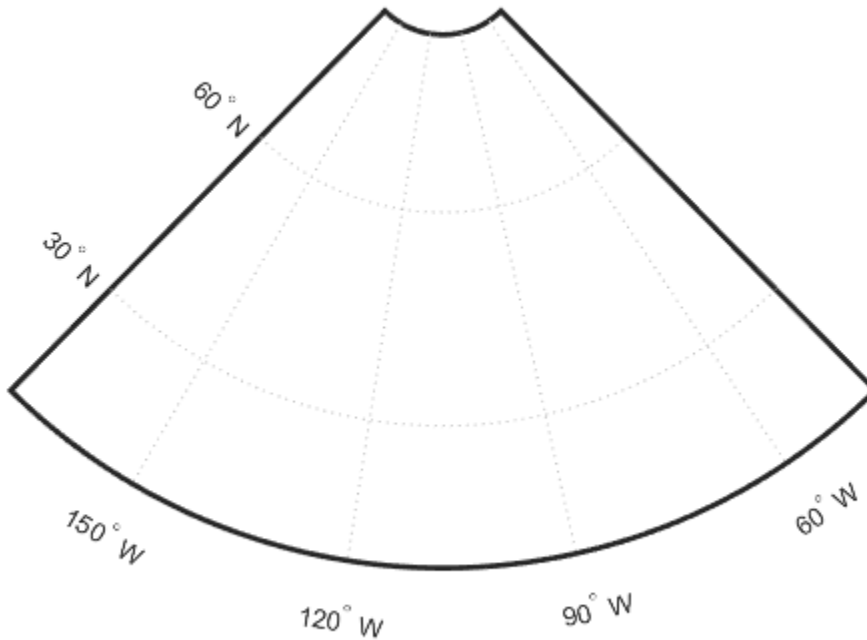
Load topographical data from MAT-file `topo.mat` .

```
load topo
```

Create geographic raster reference object.

```
R = georasterref('RasterSize',[180 360], ...
    'Latlim',topolatlim,'Lonlim',topolonlim);
```

Create a contour plot where every contour is filled with a particular color.

```
contourfm(topo, R, -7000:1000:3000)
caxis([-8000 4000])
```
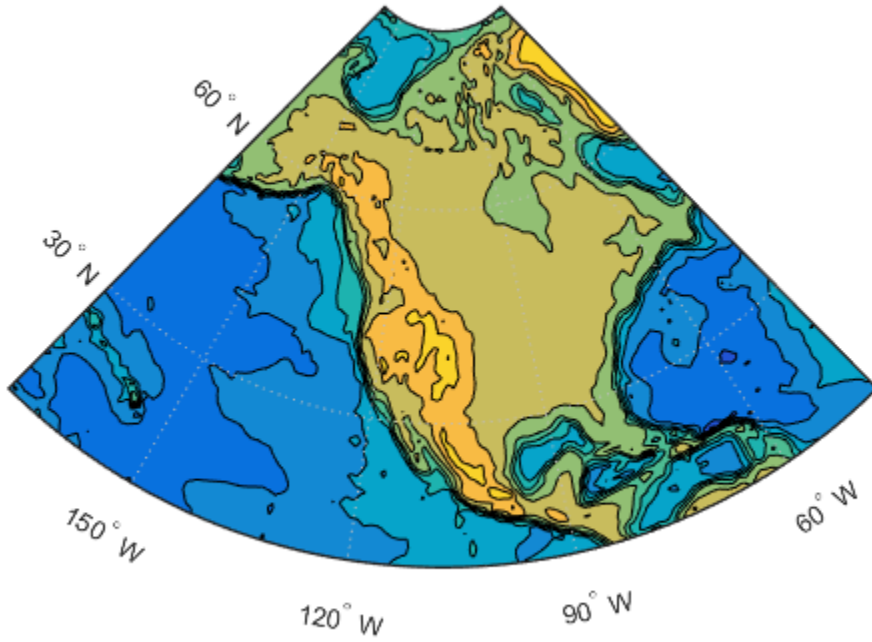
Add a color bar to the figure to explain the color used with the contours.

```
contourcbar
```
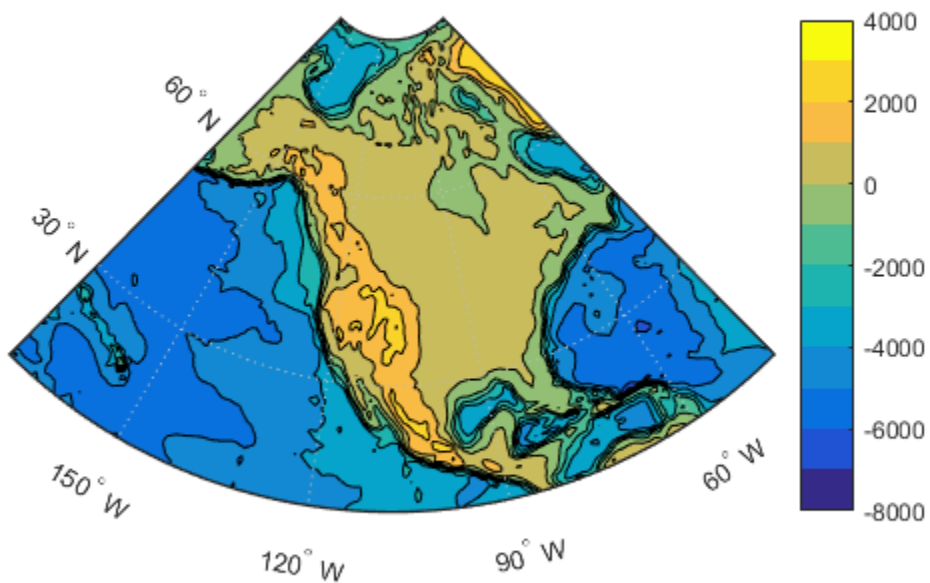
**Add colorbar to map showing geoid with non-uniform levels**

Create a world map.

```
figure('Color','white')
ax = worldmap('world');
```
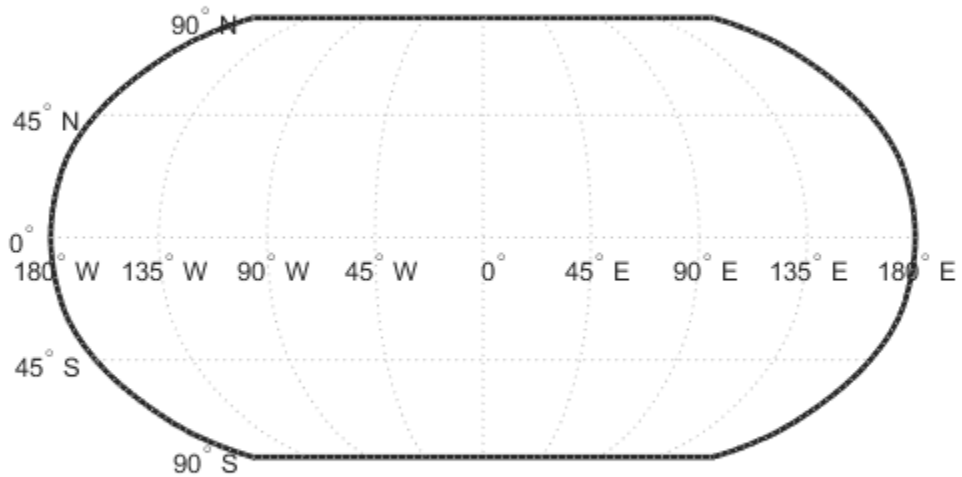
Move the meridian labels to the bottom and reduce the number of labels.

```
setm(gca,'MLabelParallel',-90)
setm(gca,'MLabelLocation',90)
```
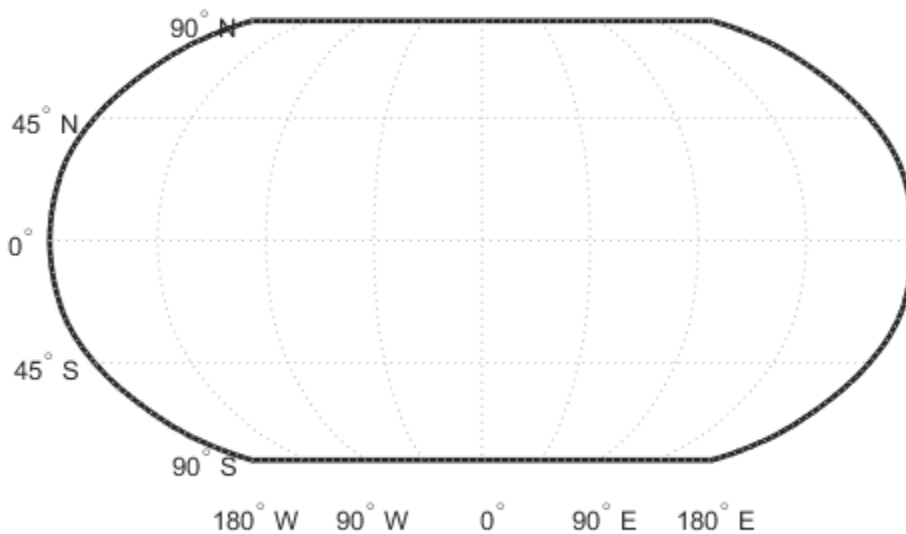
Load the geoid data set MAT-file.

```
load geoid60c
```

Specify levels (in meters) at which to contour the geoid.

```
levels = [-70 -40 -20 -10 0 10 20 40 70];
```

Display the geoid as a filled contour plot.

```
geoshow(geoid60c, geoid60cR, 'DisplayType', 'contour',...
    'LevelList',levels,'Fill','on','LineColor','black')
```
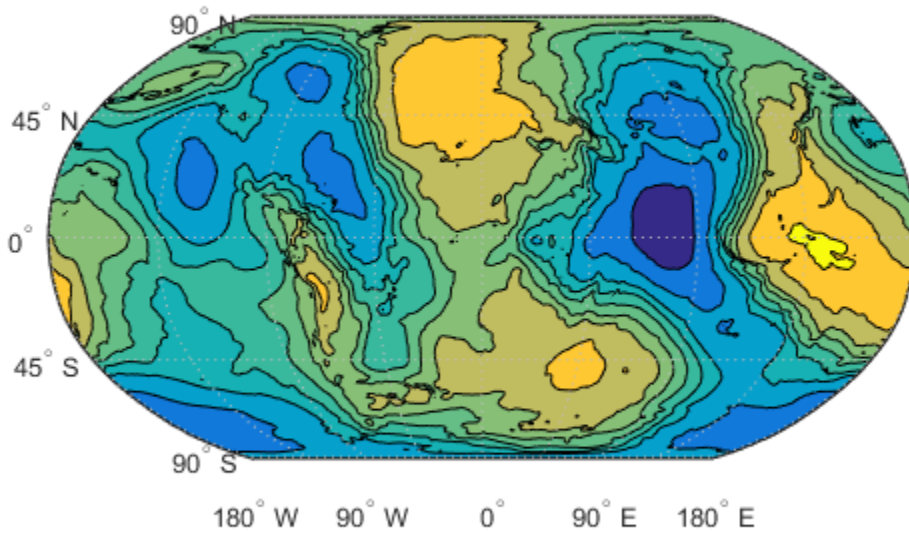
Load the coastline data MAT file, `coastlines.mat` .

```
load coastlines
```

Overlay the coastlines.

```
geoshow(coastlat, coastlon, 'Color', 'white', 'LineWidth', 1.5)
```
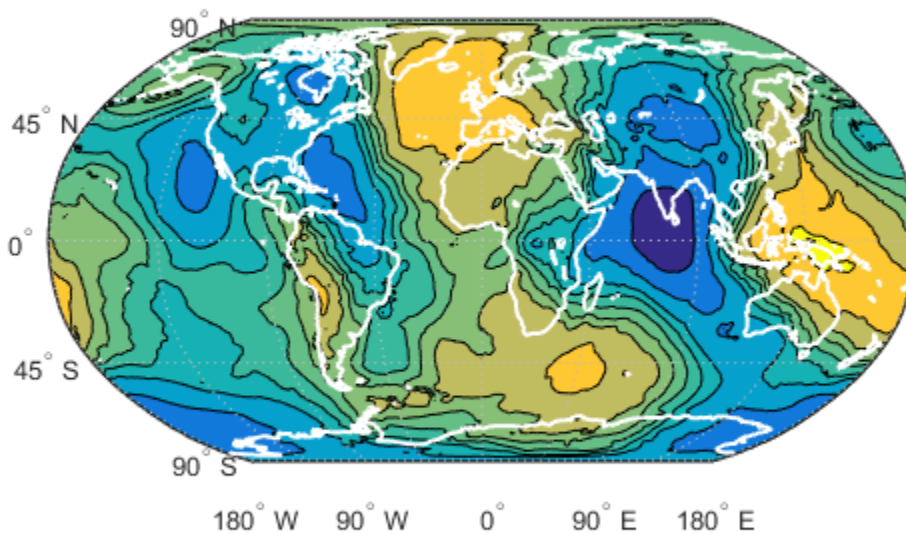
Add a color bar to indicate the range in geoid height spanned by each contour interval.

```
cb = contourcbar('peer',ax,'Location','southoutside');
```
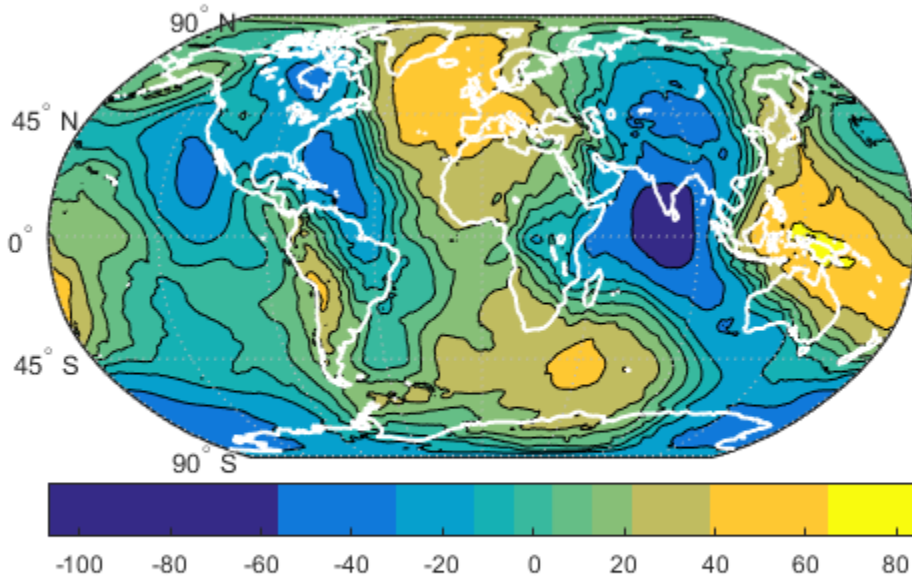
Reset the colormap and the axes color limits (CLim).

```
caxis([-110 90])
colormap(hsv)
cb.XLabel.String = 'Geoid Undulation in Meters';
```

Geoid Undulation in Meters
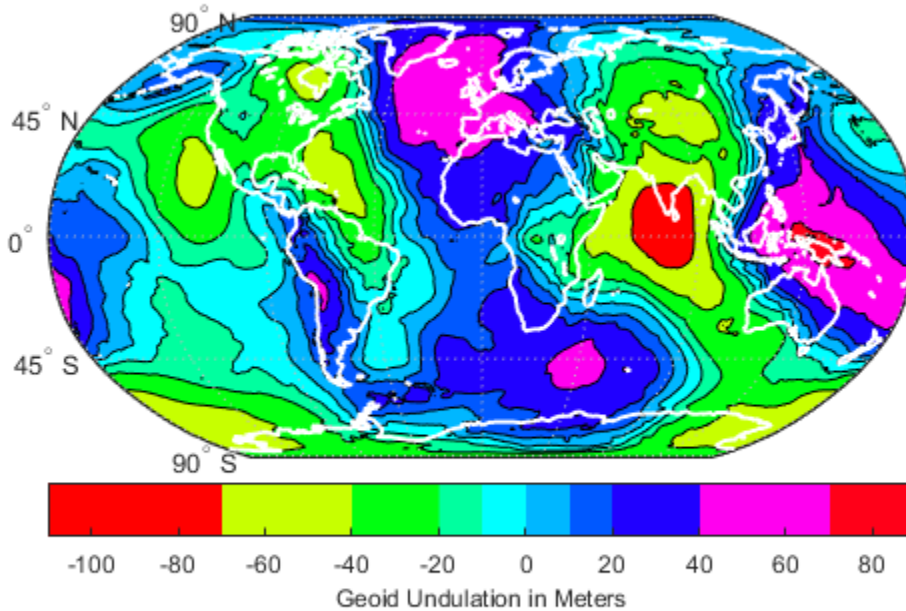
## More About

**Tips**

- If a *peer* axes is specified when calling `contourcbar`, it should be a map axes containing an object created using one of the Mapping Toolbox functions listed previously. Otherwise the current axes should contain such an object.

- If a Mapping Toolbox contour object is present, then the color bar is filled with solid blocks of color which bound each other at the contour levels used in the plot. Thus, the contour levels bounding a fill polygon of a given color can be inferred graphically by

inspecting the upper and lower limits of the corresponding block in the color bar. In the absence of a Mapping Toolbox contour object an ordinary color bar is created.

- If multiple Mapping Toolbox contour objects are present in the same axes, then the levels used to divide the color bar into blocks will correspond to the first contour object that is found. This situation could occur when a larger data set is broken up into multiple grid tiles, for example, but as long the tiles all use the same contour level list, the color bar will correctly represent them all.

## See Also

clegendm | colorbar | contourfm

# contourcmap

Contour colormap and colorbar for current axes

## Syntax

```
contourcmap(cmapstr)
contourcmap(cmapstr,cdelta)
contourcmap(...,Name,Value)
h = contourcmap(...)
```

## Description

contourcmap(cmapstr) updates the figure's colormap for the current axes with the colormap specified by cmapstr. If the axes contains Mapping Toolbox contour objects, the resultant colormap contains the same number of colors as the original colormap. Otherwise, the resultant colormap contains ten colors.

contourcmap(cmapstr,cdelta) updates the figure's colormap with colors varying according to cdelta. If the axes contains Mapping Toolbox contour objects, the value of cdelta is ignored.

contourcmap(...,Name,Value) allows you to add a colorbar and control the colorbar's properties. Parameter names can be abbreviated and are case-insensitive.

h = contourcmap(...) returns a handle to the colorbar axes.

## Input Arguments

**cmapstr**

A string that specifies a colormap. Valid entries for cmapstr include 'pink', 'hsv', 'jet', or the name of any similar MATLAB colormap function.

**cdelta**

A scalar or vector. If `cdelta` is a scalar, it represents a step size, and colors are generated at multiples of `cdelta`. If `cdelta` is a vector of evenly spaced values, colors are generated at those values; otherwise an error is issued.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'  '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**`'Colorbar'`**

String with values `'on'` or `'off'` specifying whether a colorbar is present, `'on'`, or absent from the axes, `'off'`.

**Default:** `'off'`

**`'Location'`**

String specifying the location of the colorbar. Permissible values are `'vertical'`, `'horizontal'`, or `'none'`.

**Default:** `'vertical'`

**`'ColorAlignment'`**

String specifying the alignment of the labels in the colorbar. Permissible values are `'center'`, where the labels are centered on the color bands, or `'ends'`, where the labels are centered on the color breaks. If the axes contains Mapping Toolbox contour objects, the `ColorAlignment` will be set automatically to `'center'` for contour lines and `'ends'` for filled contours, and cannot be modified.

**`'SourceObject'`**

Handle of the graphics object which is used to determine the color limits for the colormap. The `SourceObject` value is the handle of a currently displayed object.

**Default:** `gca`

**'TitleString'**

String specifying the title of the colorbar axes.

**'XLabelString'**

String specifying the X label of the colorbar axes.

**'YLabelString'**

String specifying the Y label of the colorbar axes.

**'ZLabelString'**

String specifying the Z label of the colorbar axes. In addition, properties and values that can be applied to the title and labels of the colorbar axes are valid.

# Output Arguments

**h**

A handle to the colorbar axes.

# Examples

### Display World Map with Colormap

This example shows how to display a world map with a colormap representing contour intervals in meters.

```
load topo
R = georasterref('RasterSize', size(topo), ...
   'Latlim', [-90 90], 'Lonlim', [0 360]);
figure('Color','white')
worldmap(topo, R)
contourfm(topo, R);
contourcmap('jet', 'Colorbar', 'on', ...
   'Location', 'horizontal', ...
```

```
'TitleString', 'Contour Intervals in Meters');
```
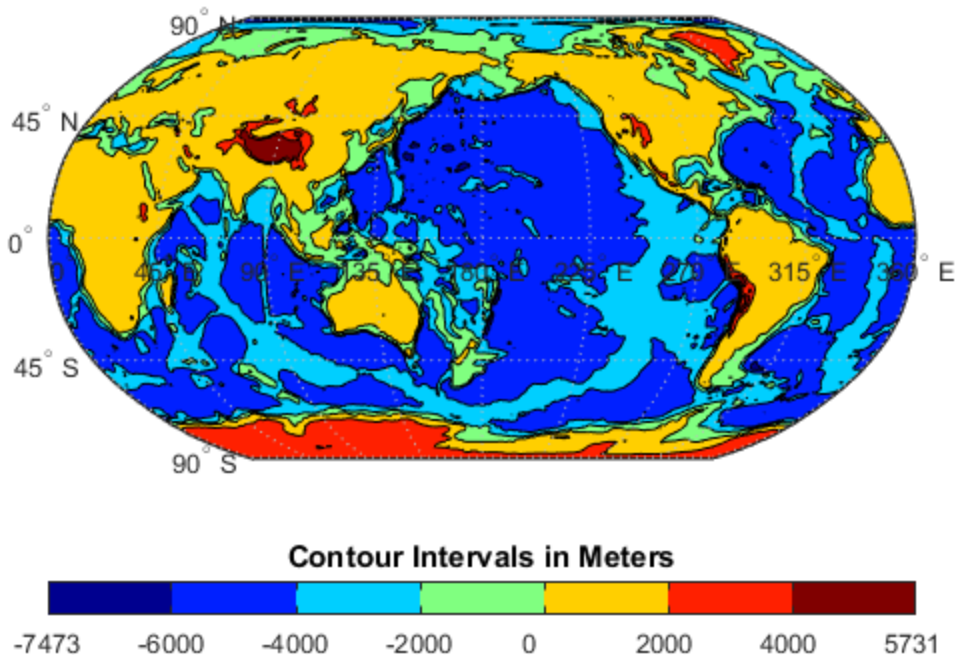


## Display World Map with Colormap with Custom Contour Intervals

This example shows how to display a world map with a colormap in which the colors vary at a step size of 2000.

Load data from MAT-files.

```
load topo
load coastlines
```

Create a raster reference object associated with the topo data set.

```
R = georefcells(topolatlim,topolonlim,size(topo));
```

Display the topo data set on a worldmap.

```
figure('Color','white')
worldmap(topo,R)
geoshow(topo,R,'DisplayType','texturemap');
```
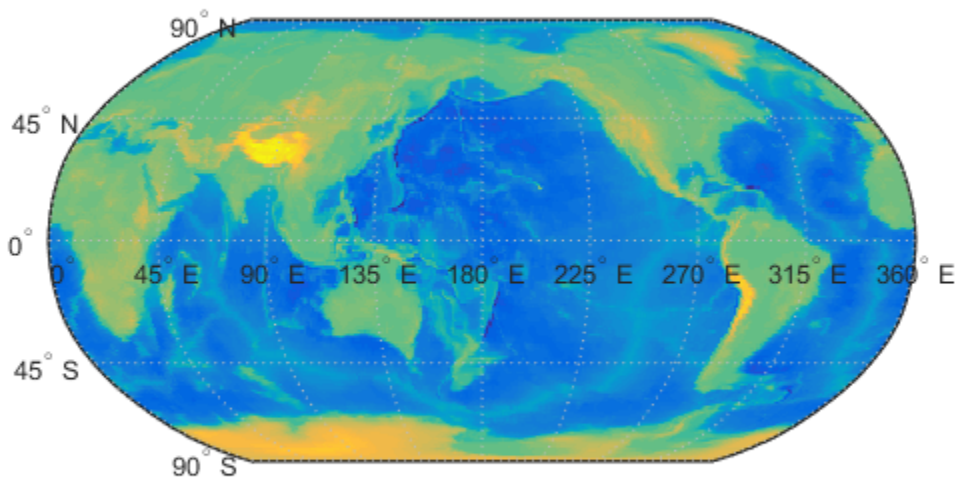


Change the color map of the contours.

```
contourcmap('summer',2000,'Colorbar','on', ...
    'Location','horizontal', ...
    'TitleString','Contour Intervals in Meters');
```

**Contour Intervals in Meters**

| -8000 | -6000 | -4000 | -2000 | 0 | 2000 | 4000 | 6000 |
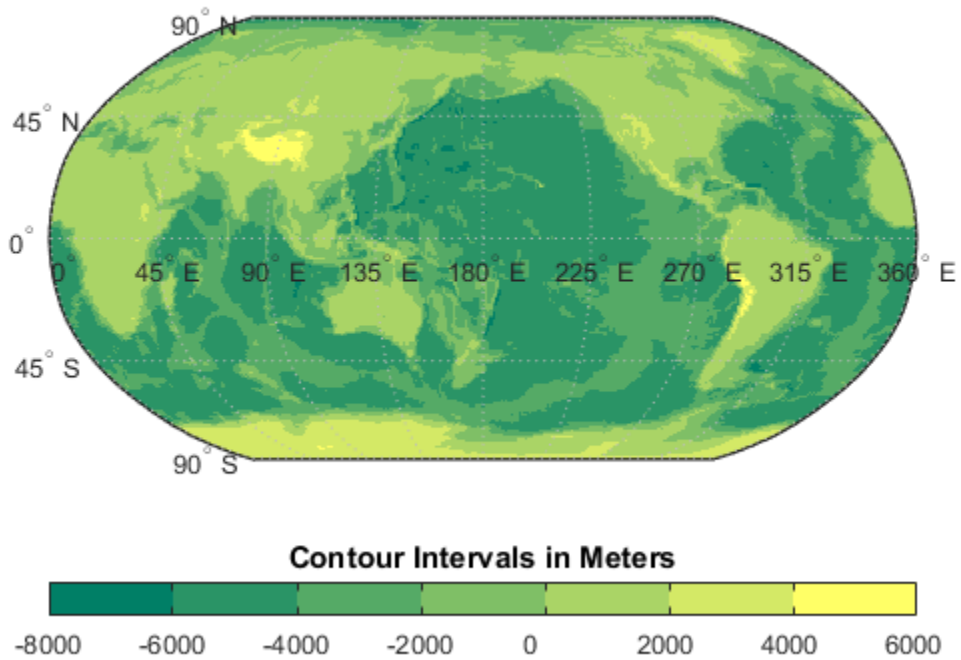
Display coastlines on the map.

```
geoshow(coastlat,coastlon,'Color', 'black')
```
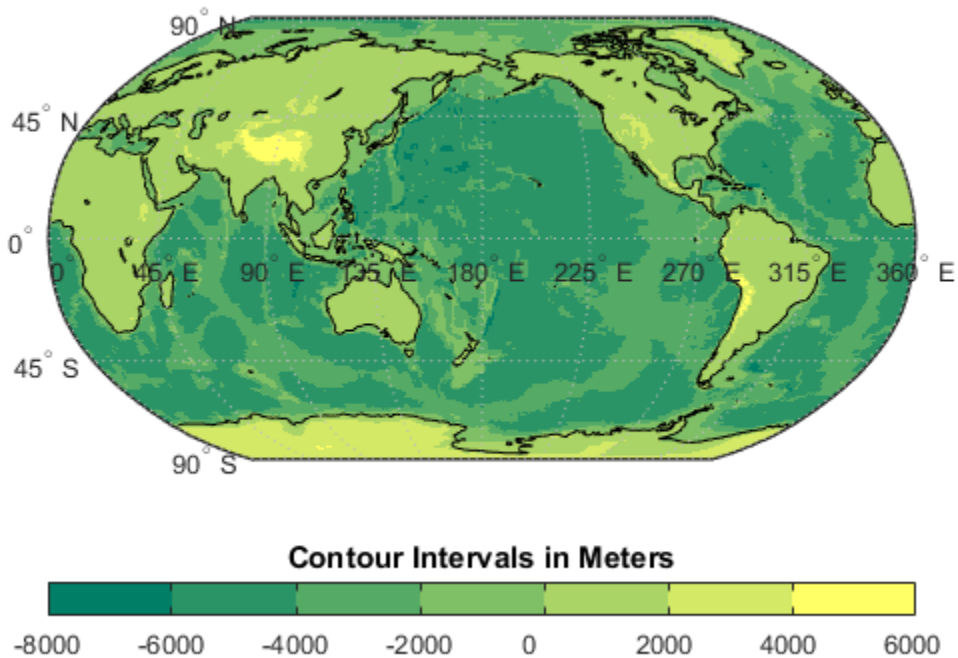
## See Also
clabelm | clegendm | colormap | contourfm | contourm | contour3m |
contourcbar

# contourfm

Project filled 2-D contour plot of map data

## Description

The `contourfm` function is the same as the `contourm` function except that the areas between contours are filled with colors. For each contour interval, `contourfm` selects a distinct color from the figure's colormap. You can obtain the same result by setting `'Fill','on'` and `'LineColor','black'` when calling `contourm`.

## Examples

### Plot Earth's Geoid with Filled Contours

This example shows how to plot the Earth's geoid with filled contours, measure in meters.

```
figure
axesm eckert4; framem; gridm; axis off; tightmap

load geoid
contourfm(geoid, geoidrefvec, -120:20:100, 'LineStyle', 'none');

coast = load('coast');
geoshow(coast.lat, coast.long, 'Color', 'black')

contourcbar
```
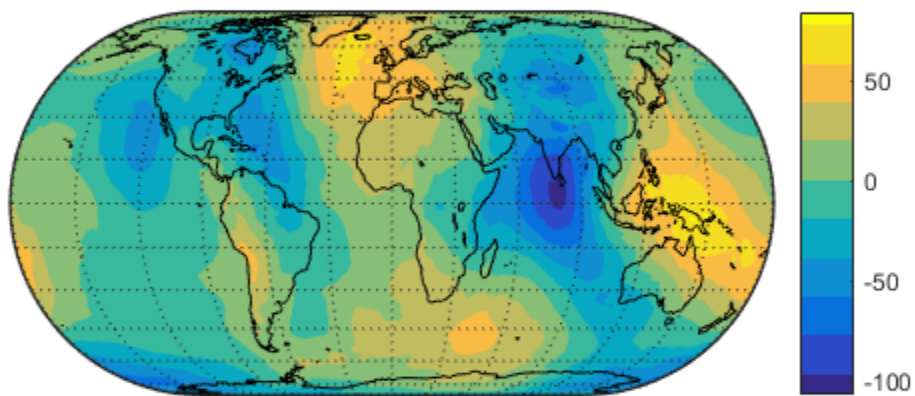
## See Also

contourcbar | contourm | contour3m | clabelm | meshm | surfm

# contourm

Project 2-D contour plot of map data

## Syntax

```
contourm(Z,R)
contourm(lat,lon,Z)
contourm( ___ ,n)
contourm( ___ ,V)
contourm( ___ ,LineSpec)
contourm( ___ ,Name,Value,...)
C = contourm( ___ )
[C,h] = contourm( ___ )
```

## Description

`contourm(Z,R)` creates a contour plot of the regular data grid `Z`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

`contourm(lat,lon,Z)` displays a contour plot of the geolocated, M-by-N data grid, `Z`. `lat,lon` can be the size of `Z` or can specify the corresponding row and column dimensions for `Z`.

`contourm( ___ ,n)` draws `n` contour levels.

`contourm( ___ ,V)` draws contours at the levels specified by the input vector `V`. Use `V = [v v]` to compute a single contour at level `v`.

`contourm( ___ ,LineSpec)` uses any valid `LineSpec` string to draw the contour lines.

`contourm( ___ ,Name,Value,...)` allows you to set optional parameters. Parameter names can be abbreviated, and case does not matter. In addition, any of the following `hggroup` properties may be specified: HandleVisibility, Parent, Tag, UserData, and Visible.

`C = contourm( ___ )` returns a standard contour matrix `C`, with the first row representing longitude data and the second row representing latitude data

**1-137**

`[C,h] = contourm( ___ )` returns the contour matrix and the handle to the contour patches drawn onto the current axes. The handle is of type `hggroup`.

## Input Arguments

**Z**

Regular or geolocated data grid.

If the grid contains regions with missing data, set the corresponding elements of `Z` to NaN. Contour lines terminate when entering such areas. Similarly, if you use `'Fill','on'` or call `contourfm`, such null-data areas will not be filled. If the syntax `contourm(lat,lon,Z, ...)` is used, however, `lat` and `lon` must have finite, non-NaN values everywhere. In this case, set `Z` to NaN in null data areas, but make sure the corresponding elements of `lat` and `lon` have finite values that specify actual locations on the Earth.

**R**

Geographic raster reference object, referencing vector, or referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

`[cells/degree northern_latitude_limit western_longitude_limit]`

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

`[lon lat] = [row col 1] * R`

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. For more information about referencing vectors and matrices, see "Georeferenced Raster Data". If the current axis is a map axis, the coordinates of `Z` will be projected using the projection structure from the axis. The contours are drawn at their corresponding `Z` level.

**lat**

Geolocation array having the same size as `Z`, or a vector with length matching the number of rows in `Z`.

**lon**

Geolocation array having the same size as `Z`, or a vector with length matching the number of columns in `Z`.

**n**

Scalar specifying the number of contour levels.

**V**

Vector specifying contour levels.

**LineSpec**

Line specification string. See the MATLAB function reference page for `LineSpec` for more information.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'Fill'**

off | on

*Color areas between contour lines.* By default `contourm` draws a line (which may have multiple parts) for each contour level. If you set `Fill` to `on`, `contourm` colors the polygonal regions between the lines, selecting a distinct color for each contour interval from the colormap of the figure in which the contours are drawn. Setting `Fill` to `on` is almost the same as calling `contourfm`; the only difference is that `contourfm` also sets `LineColor` to black by default.

**Default:** off

**`'LabelSpacing'`**

scalar

*Spacing between labels on each contour line.* When you display contour line labels either by calling clabelm or by specifying `'ShowText','on'`, the labels by default are spaced 144 points (2 inches) apart on each line. You can specify the spacing by setting LabelSpacing to a value in points. If the length of an individual contour line is less than the specified value, only one contour label is displayed on that line.

**`'LevelList'`**

vector

*Values at which contour lines are drawn.* This property uses a row vector of increasing values to specify the levels at which contour lines are drawn.

**`'LevelStep'`**

scalar

*Spacing of contour lines.* When LevelStep, which must be a positive real number, is specified, the contourm function draws contour lines at regular intervals determined by the value of LevelStep, unless the optional third argument, n (number of contour levels) or V (vector specifying contour levels) is provided. If n or V is used in combination with the LevelStep parameter, then the LevelStep parameter is ignored. If n, v, and the LevelStep parameter are all omitted, contourm selects a uniform step automatically.

**`'LineColor'`**

flat | ColorSpec | none

*Contour line colors.* To specify a single color to be used for all the contour lines, you can set ColorSpec to a three-element RGB vector or one of the MATLAB predefined names. See the MATLAB ColorSpec reference page for more information on specifying color. If you omit LineColor or set it to flat, contourm selects a distinct color for lines at each contour level from the colormap of the figure in which the contours are drawn. If you set LineColor to none, the contour lines will not be visible.

**Default:** flat

**`'LineStyle'`**

- | -- | : | -. | none

*Line style for contour lines.* Options for `LineStyle` include solid (specified by -), dashed (--), dotted (:), dash-dot (-.), and none. The specifier strings work the same as for line objects in MATLAB graphics.

**Default:** -

**'LineWidth'**

scalar

*Width of the contour lines in points (1 point = 1/72 inch).*

**Default:** 0.5

**'ShowText'**

on | off

*Display labels on contour lines.* If you set `ShowText` to `on`, `contourm` displays text labels on each contour line indicating the value of the corresponding contour level. Another way to add labels to your contour lines is to call `clabelm` after calling `contourm`.

**Default:** off

# Output Arguments

**C**

Standard contour matrix with the first row representing longitude data and the second row representing latitude data.

**h**

Handle to the `hggroup` containing the contour lines.

# Examples

Contour geoid heights for an area including Korea with a backdrop of terrain elevations and bathymetry:

```
% Load the data.
```

```
load korea
load geoid

% Create a map axes that includes Korea.
figure
worldmap(map, refvec)

% Display the digital elevation data and apply a colormap.
geoshow(map, refvec, 'DisplayType', 'texturemap');
demcmap(map)

% Contour the geoid values from -100 to 100 in increments of 5.
[c,h] = contourm(geoid, geoidlegend, -100:5:100, 'k');

% Add red labels with white backgrounds to the contours.
ht = clabelm(c,h);
set(ht,'Color','r','BackgroundColor','white','FontWeight','bold')
```
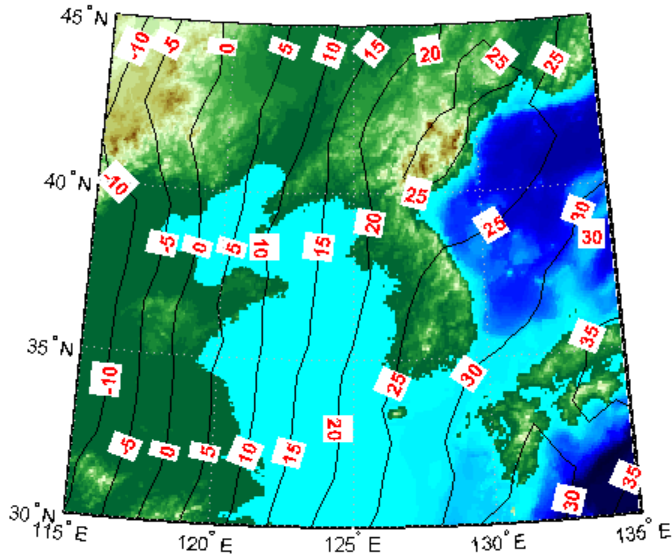


## Display Contours for EGM96 Geoid Heights

Load EGM96 geoid heights.

```
load geoid
```

Display contours.

```
figure
[c,h] = contourm(geoid,geoidrefvec,'LevelStep',20,'ShowText','on');
```
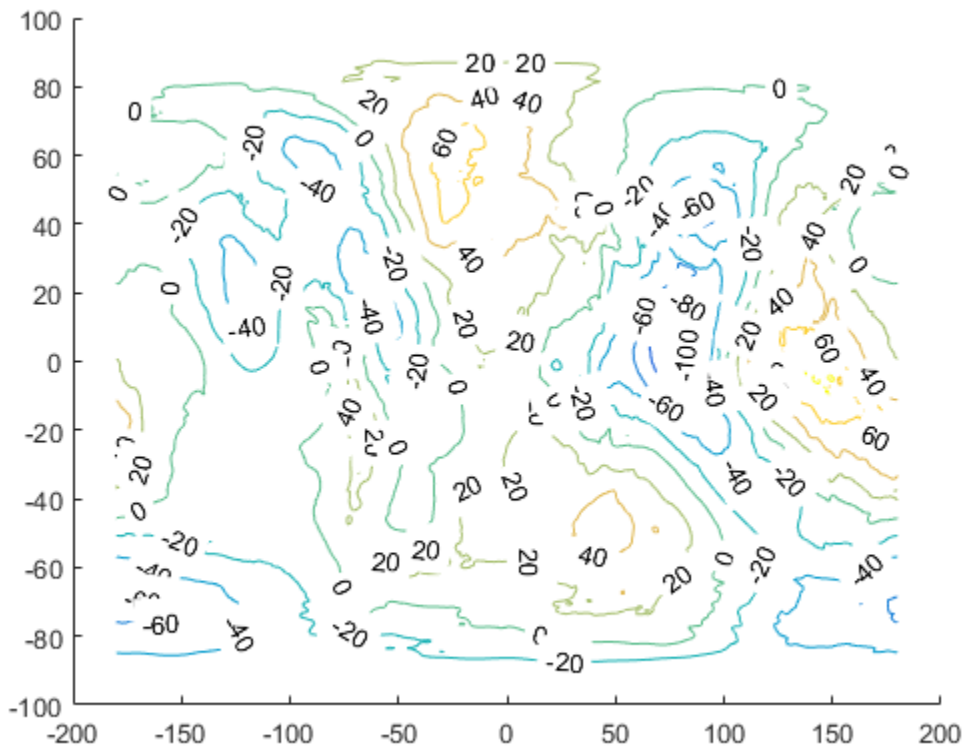


Add axes labels and a legend.

```
xlabel('Longitude')
ylabel('Latitude')
clegendm(c,h,-1)
```

**1-143**
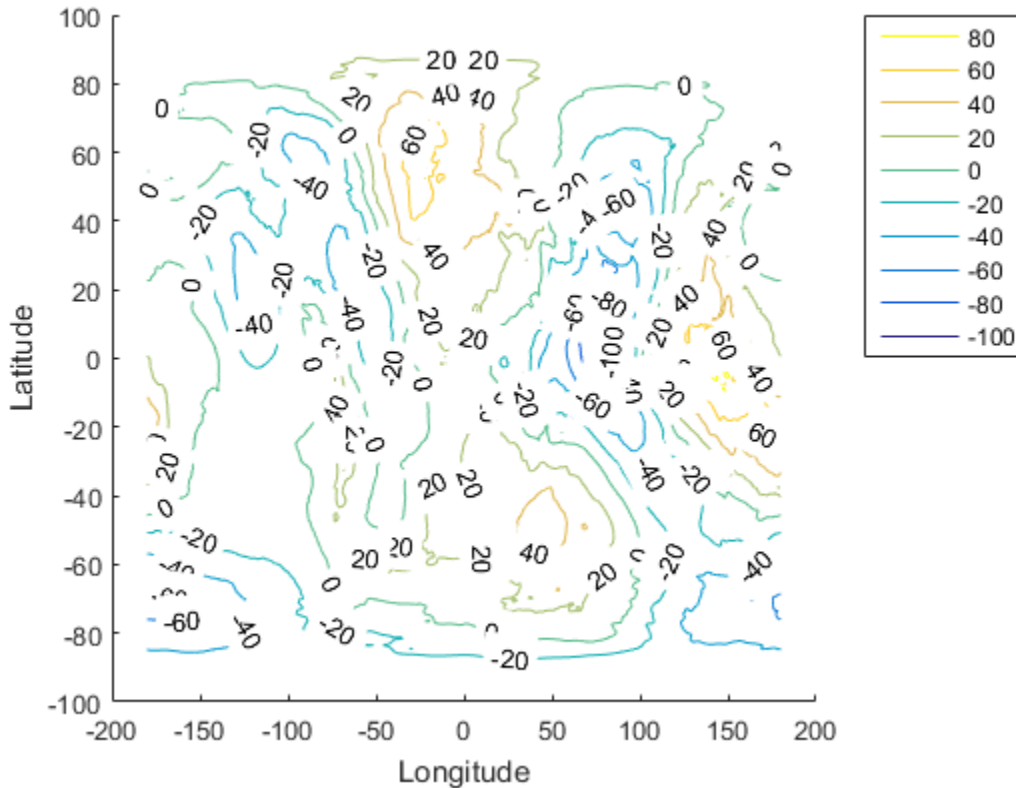
### Display Contour Geoid Heights for Area Including Korea

Contour geoid heights for an area including Korea with a backdrop of terrain elevations and bathymetry:

Load geoid heights and Korea data.

```
load korea
load geoid
```

Create a map axes that includes Korea.

```
figure
```

```
worldmap(map, refvec)
```



Display the digital elevation data and apply a colormap.

```
geoshow(map, refvec, 'DisplayType', 'texturemap');
demcmap(map)
```

**1-145**
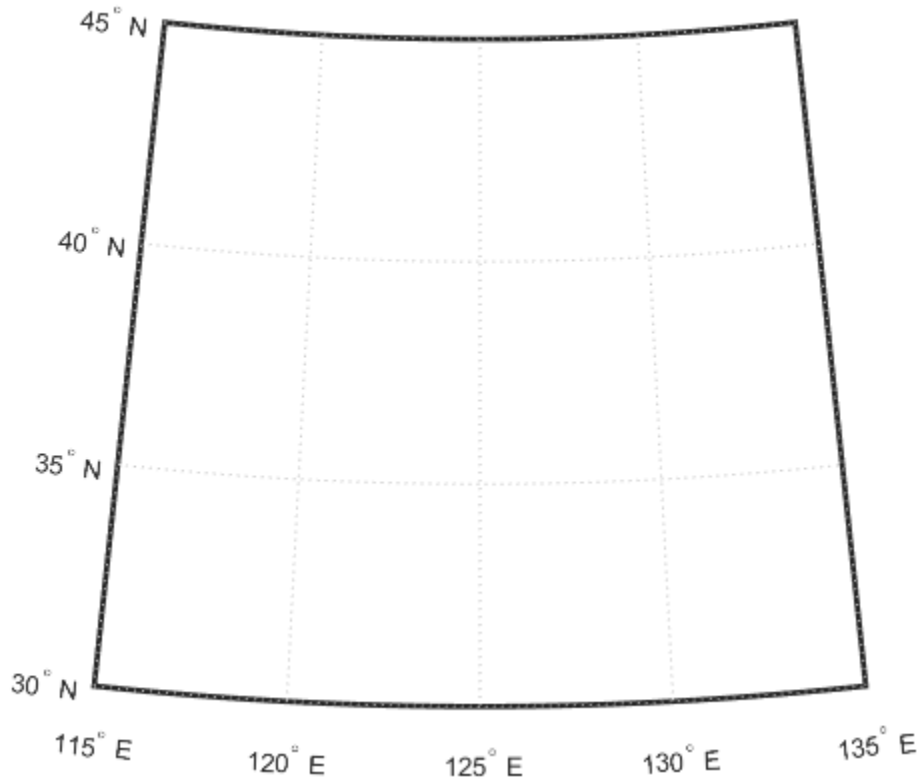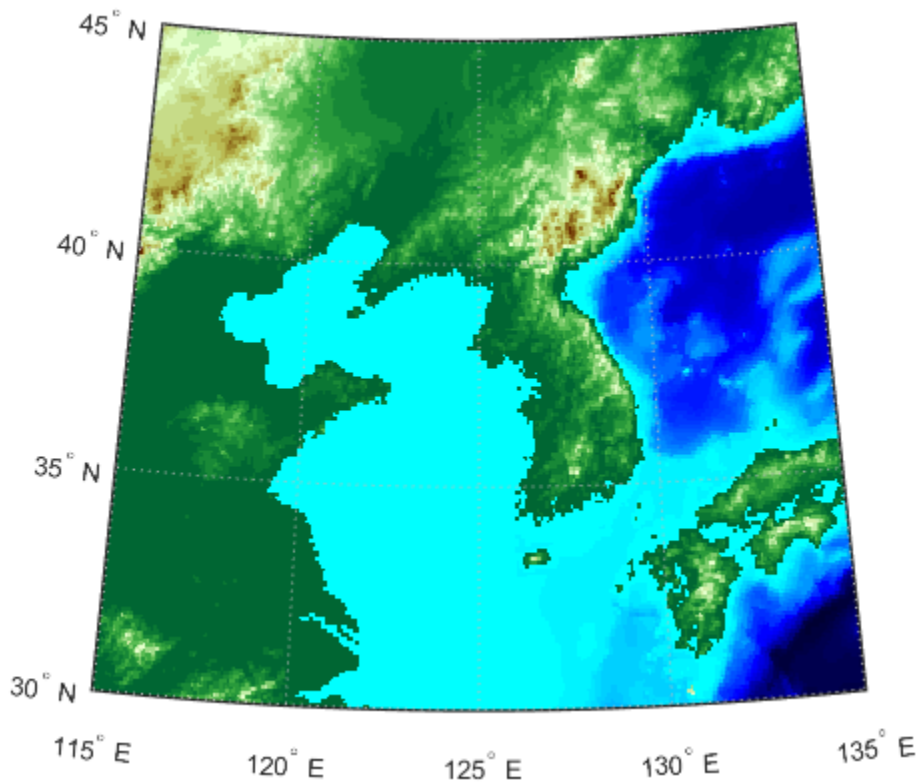
Display contours of the geoid values from -100 to 100 in increments of 5.

```
[c,h] = contourm(geoid, geoidlegend, -100:5:100, 'k');
```
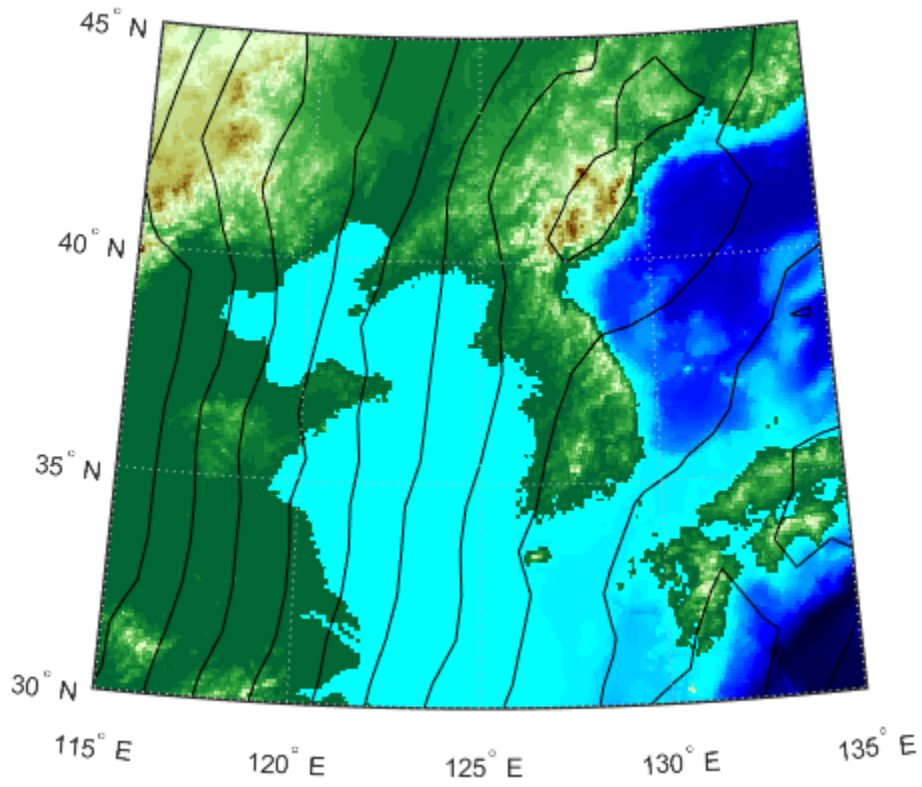
Add red labels with white backgrounds to the contours.

```
ht = clabelm(c,h);
[ht.Color] = deal('r');
[ht.BackgroundColor] = deal('white');
[ht.FontWeight] = deal('bold');
```
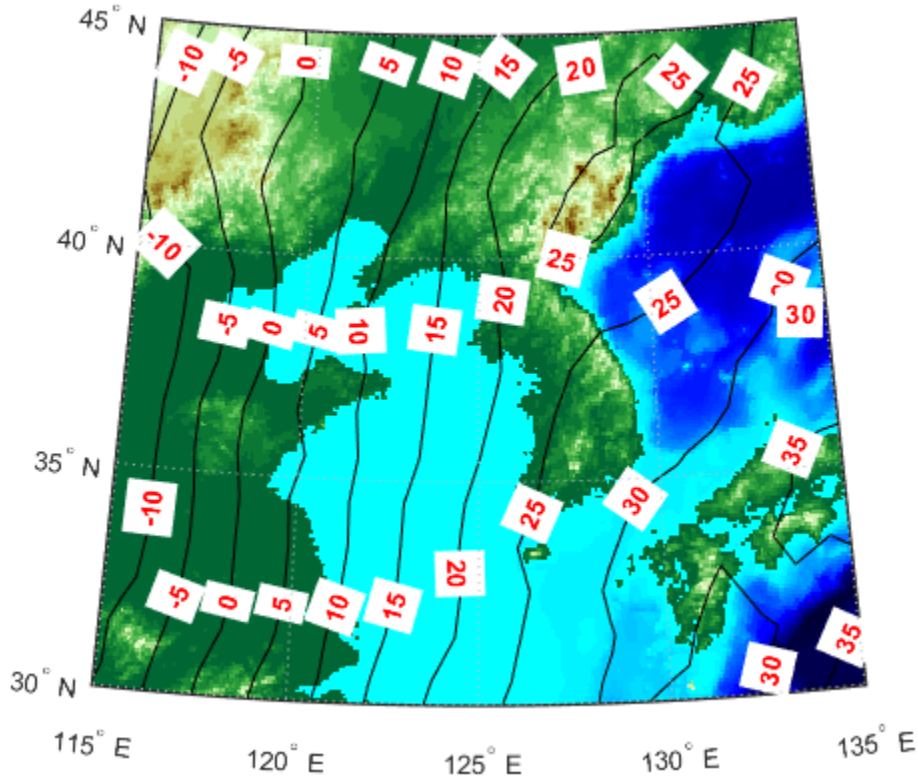
## More About

**Tips**

You have three ways to control the number of contour levels that display in your map:

1   Set the number of contour levels by specifying the scalar n in the syntax
    `contourm(Z,R,n)` or `contourm(lat,lon,Z,n)`.

2   Use the vector V to specify the levels at which contours are drawn with the syntax
    `contourm(Z,R,V)` or `contourm(lat,lon,Z,V)`.

**3** Choose regular intervals at which the contours are drawn by setting the `LevelStep` parameter.

If you do not use any of the above methods to set your contour levels, the `contourm` function will display around five contour levels.

## See Also

clabelm | clegendm | contour | contourc | contour3 | contour3m | contourfm | geoshow | plot

# convertlat

Convert between geodetic and auxiliary latitudes

## Syntax

```
latout = convertlat(ellipsoid,latin,from,to,units)
```

## Description

`latout = convertlat(ellipsoid,latin,from,to,units)` converts latitude values in `latin` from type FROM to type TO. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form [`semimajor_axis eccentricity`].

`latin` is an array of input latitude values. `from` and `to` are each one of the latitude type strings listed below (or unambiguous abbreviations). `latin` has the angle units specified by `units`: either `'degrees'`, `'radians'`, or unambiguous abbreviations. The output array, `latout`, has the same size and units as `latin`.

| Latitude Type | Description |
|---|---|
| geodetic | The geodetic latitude is the angle that a line perpendicular to the surface of the ellipsoid at the given point makes with the equatorial plane. |
| authalic | The authalic latitude maps an ellipsoid to a sphere while preserving surface area. Authalic latitudes are used in place of the geodetic latitudes when projecting the ellipsoid using an equal area projection. |
| conformal | The conformal latitude maps an ellipsoid conformally onto a sphere. Conformal latitudes are used in place of the geodetic latitudes when projecting the ellipsoid with a conformal projection. |
| geocentric | The geocentric latitude is the angle that a line connecting a point on the surface of the ellipsoid to its center makes with the equatorial plane. |

| Latitude Type | Description |
|---|---|
| `isometric` | The isometric latitude is a nonlinear function of the geodetic latitude. |
| `parametric` | The parametric latitude of a point on the ellipsoid is the latitude on a sphere of radius a, where a is the semimajor axis of the ellipsoid, for which the parallel has the same radius as the parallel of geodetic latitude. |
| `rectifying` | The rectifying latitude is used to map an ellipsoid to a sphere in such a way that distance is preserved along meridians. |

To properly project rectified latitudes, the radius must also be scaled to ensure the equal meridional distance property. This is accomplished by `rsphere`.

## Examples

```
% Plot the difference between the auxiliary latitudes
% and geocentric latitude, from equator to pole,
% using the GRS 80 ellipsoid. Avoid the polar region with
% the isometric latitude, and scale down the difference
% by a factor of 200.
grs80 = referenceEllipsoid('grs80');
geodetic = 0:2:90;
authalic = ...
convertlat(grs80,geodetic,'geodetic','authalic', 'deg');
conformal = ...
convertlat(grs80,geodetic,'geodetic','conformal', 'deg');
geocentric = ...
convertlat(grs80,geodetic,'geodetic','geocentric','deg');
parametric = ...
convertlat(grs80,geodetic,'geodetic','parametric','deg');
rectifying = ...
convertlat(grs80,geodetic,'geodetic','rectifying','deg');
isometric = ...
convertlat(grs80,geodetic(1:end-5), ...
'geodetic','isometric','deg');
plot(geodetic, (authalic - geodetic),...
geodetic, (conformal - geodetic),...
geodetic, (geocentric - geodetic),...
geodetic, (parametric - geodetic),...
geodetic, (rectifying - geodetic),...
```

```
geodetic(1:end-5), (isometric - geodetic(1:end-5))/200);
title('Auxiliary Latitudes vs. Geodetic')
xlabel('geodetic latitude, degrees')
ylabel('departure from geodetic, degrees');
legend('authalic','conformal','geocentric', ...
'parametric','rectifying', 'isometric/200',...
'Location','NorthWest');
```
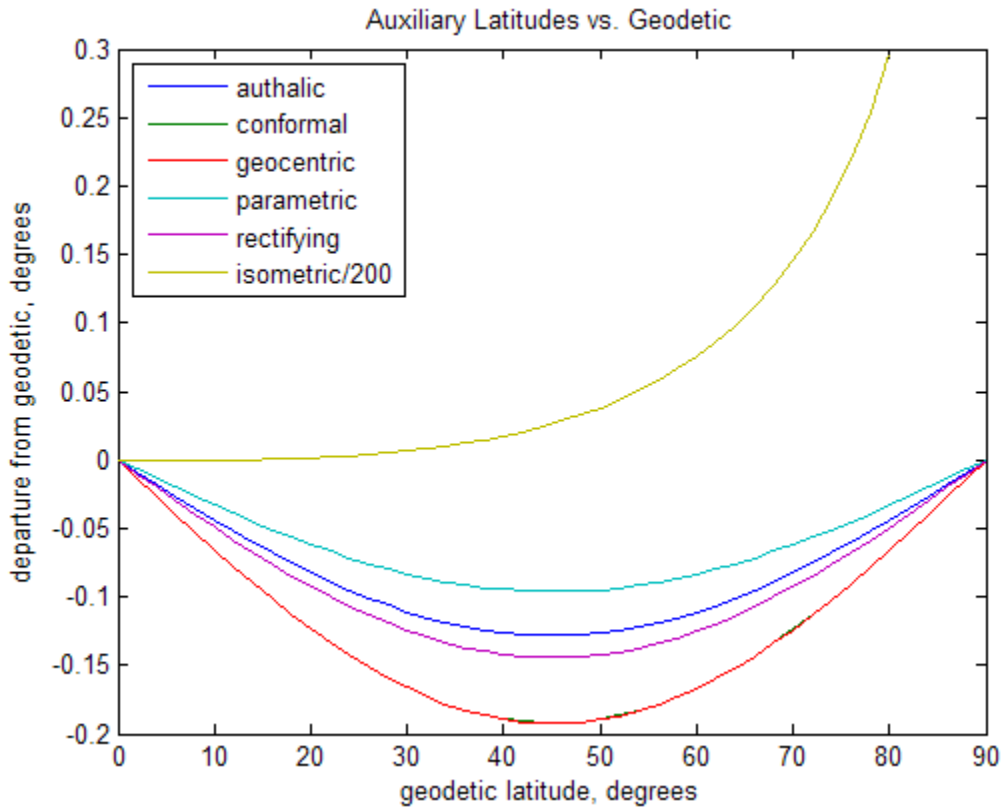


## See Also

referenceEllipsoid | referenceSphere | oblateSpheroid | `rsphere`

# crossfix

Cross-fix positions from bearings and ranges

## Syntax

```
[newlat,newlon] = crossfix(lat,lon,az)
[newlat,newlon] = crossfix(lat,lon,az_range,case)
[newlat,newlon] = crossfix(lat,lon,az_range,case,drlat,drlon)
[newlat,newlon] = crossfix(lat,lon,az,units)
[newlat,newlon] = crossfix(lat,lon,az_range,case,units)
[newlat,newlon] = crossfix(lat,lon,az_range,drlat,drlon,units)
[newlat,newlon] = crossfix(lat,lon,az_range,case,drlat,drlon,units)
mat = crossfix(...)
```

## Description

[newlat,newlon] = crossfix(lat,lon,az) returns the intersection points of all pairs of great circles passing through the points given by the column vectors lat and lon that have azimuths az at those points. The outputs are two-column matrices newlat and newlon in which each row represents the two intersections of a possible pairing of the input great circles. If there are *n* input objects, there will be *n choose 2* pairings.

[newlat,newlon] = crossfix(lat,lon,az_range,case) allows the input az_range to specify either azimuths or ranges. Where the vector case equals 1, the corresponding element of az_range is an azimuth; where case is 0, az_range is a range. The default value of case is a vector of ones (azimuths).

[newlat,newlon] = crossfix(lat,lon,az_range,case,drlat,drlon) resolves the ambiguities when there is more than one intersection between two objects. The scalar-valued drlat and drlon provide the location of an estimated (dead reckoned) position. The outputs newlat and newlon are column vectors in this case, returning only the intersection closest to the estimated point. When this option is employed, if any pair of objects fails to intersect, no output is returned and the warning No Fix is displayed.

[newlat,newlon] = crossfix(lat,lon,az,*units*), [newlat,newlon] = crossfix(lat,lon,az_range,case,*units*), [newlat,newlon] =

crossfix(lat,lon,az_range,drlat,drlon,*units*), and [newlat,newlon] = crossfix(lat,lon,az_range,case,drlat,drlon,*units*) allow the specification of the angle units to be used for all angles and ranges, where *units* is any valid angle units string. The default value of *units* is 'degrees'.

mat = crossfix(...) returns the output in a two- or four-column matrix mat.

This function calculates the points of intersection between a set of objects taken in pairs. Given great circle azimuths and/or ranges from input points, the locations of the possible intersections are returned. This is different from the navigational function navfix in that crossfix uses great circle measurement, while navfix uses rhumb line azimuths and nautical mile distances.

# Examples

### Find intersections of points on circles

This example shows how to find the intersection of points on circles.

Create map axes.

```
figure('color','w');
ha = axesm('mapproj','mercator', ...
    'maplatlim',[-10 15],'maplonlim',[-10 20],...
    'MLineLocation',5,'PLineLocation',5);
axis off
gridm on
framem on
mlabel on
plabel on
```
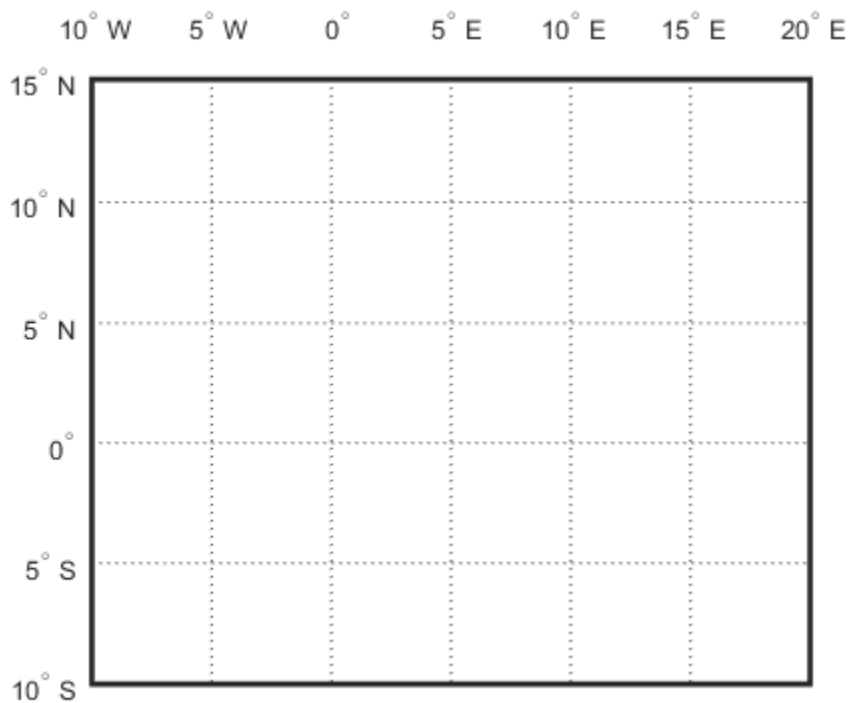
Define latitudes and longitudes of three arbitray points, and then define three radii, all 8 degrees.

```
latpts = [0;5;0];
lonpts = [0;5;10];
radii = [8;8;8];
```

Obtain the intersections of imagined small circles around these points.

```
[newlat,newlon] = crossfix(latpts,lonpts,radii,[0;0;0])
```

```
newlat =

   7.5594   -2.5744
```

```
    6.2529    -6.2529
    7.5594    -2.5744


newlon =

   -2.6260     7.5770
    5.0000     5.0000
   12.6260     2.4230
```

Draw red circle markers at the given points.

```
geoshow(latpts,lonpts,'DisplayType','point',...
    'markeredgecolor','r','markerfacecolor','r','marker','o')
```
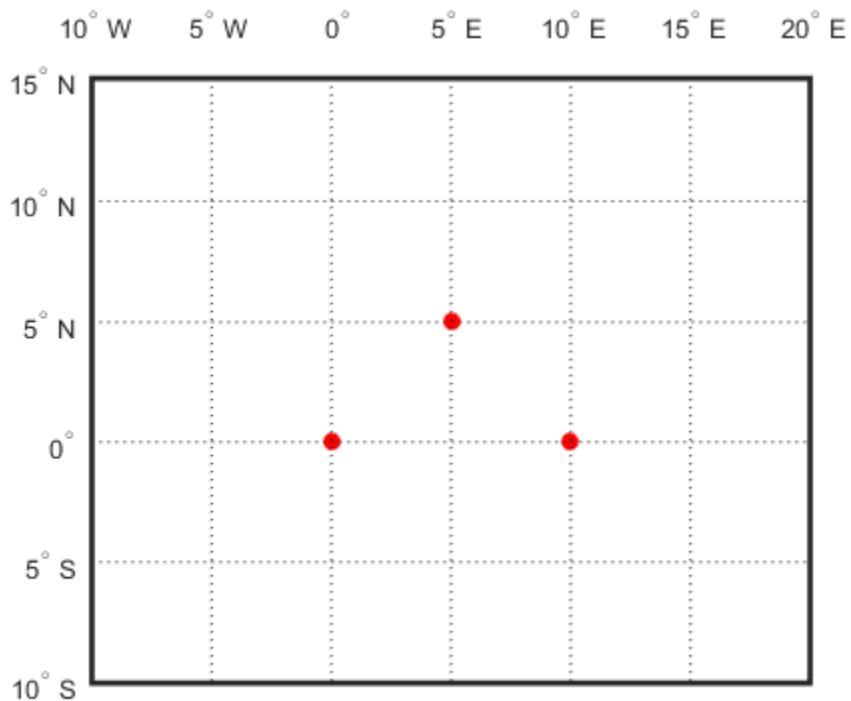
Draw magenta diamond markers at the points of intersection.

```
geoshow(reshape(newlat,6,1),reshape(newlon,6,1),'DisplayType','point',...
    'markeredgecolor','m','markerfacecolor','m','marker','d')
```
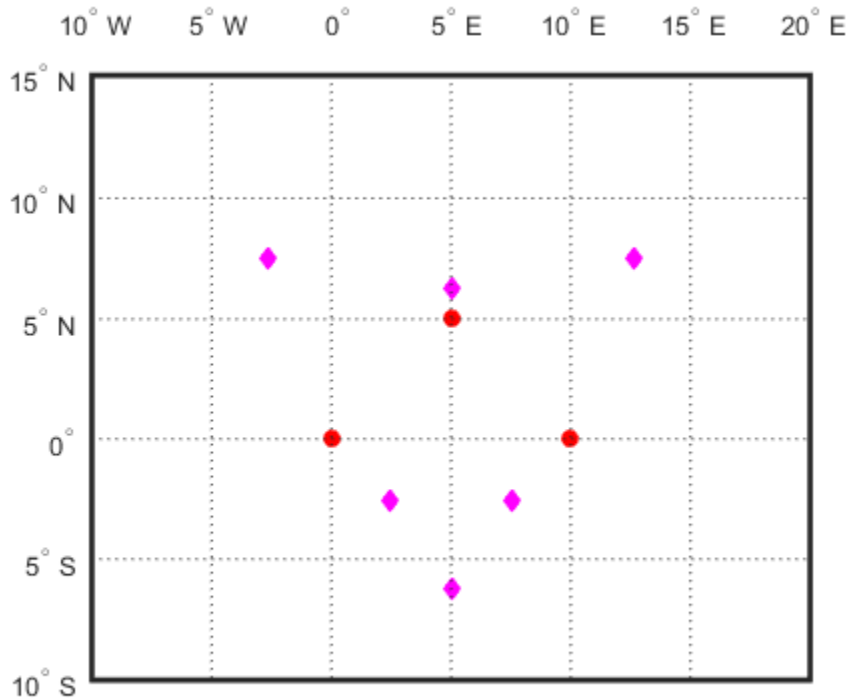


Generate a small circle 8 deg radius for each original point.

```
[latc1,lonc1] = scircle1(latpts(1),lonpts(1),radii(1));
[latc2,lonc2] = scircle1(latpts(2),lonpts(2),radii(2));
[latc3,lonc3] = scircle1(latpts(3),lonpts(3),radii(3));
```

Plot the small circles to show the intersections are as determined.

```
geoshow(latc1,lonc1,'DisplayType','line',...
    'color','b','linestyle','-')
```

```
geoshow(latc2,lonc2,'DisplayType','line',...
    'color','b','linestyle','-')
geoshow(latc3,lonc3,'DisplayType','line',...
    'color','b','linestyle','-')
```
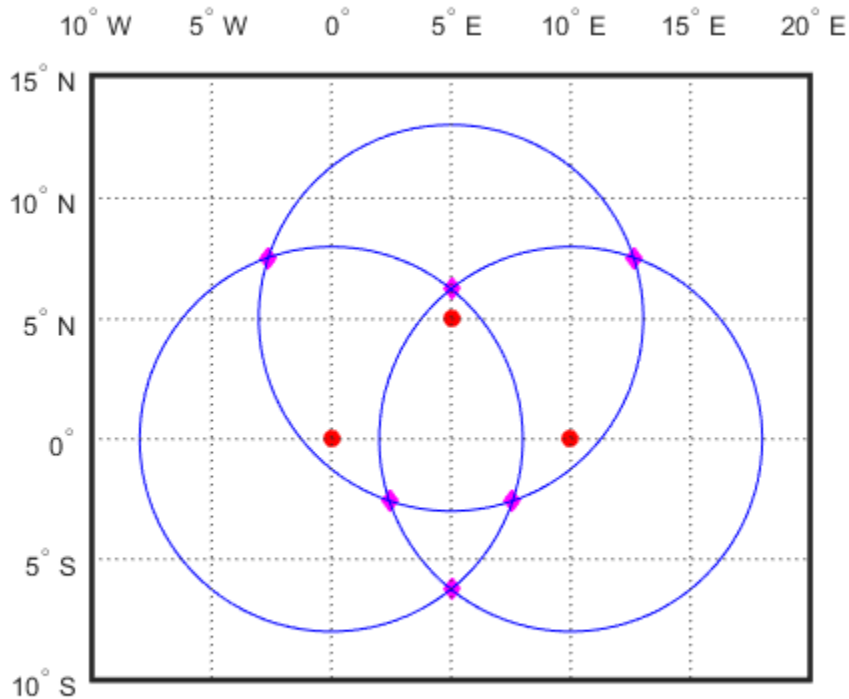


### Find Intersection when provided with dead reckoning position

Find intersection when a dead reckoning position is provided, (0º,5ºE). `crossfix` returns one from each pair (the closest one).

```
[newlat,newlon] = crossfix([0 5 0]',[0 5 10]',...
                           [8 8 8]',[0 0 0]',0,5)

newlat =
```

```
    -2.5744
     6.2529
    -2.5744

newlon =

     7.5770
     5.0000
     2.4230
```

## See Also

gcxgc | gcxsc | scxsc | rhxrh | polyxpoly | navfix

# daspectm

Control vertical exaggeration in map display

## Syntax

```
daspectm(zunits)
daspectm(zunits,vfac)
daspectm(zunits,vfac,lat,long)
daspectm(zunits,vfac,lat,long,az)
daspectm(zunits,vfac,lat,long,az,radius)
```

## Description

daspectm(*zunits*) sets the 'DataAspectRatio' property of the map axes so that the *z*-axis is in proportion to the *x*-and *y*-projected coordinates. This permits elevation data to be displayed without vertical distortion. The string *zunits* specifies the units of the elevation data, and can be any length units recognized by unitsratio.

daspectm(*zunits*,vfac) sets the 'DataAspectRatio' property so that the *z*-axis is vertically exaggerated by the factor vfac. If omitted, the default is no vertical exaggeration.

daspectm(*zunits*,vfac,lat,long) sets the aspect ratio based on the local map scale at the specified geographic location. If omitted, the default is the center of the map limits.

daspectm(*zunits*,vfac,lat,long,az) also specifies the direction along which the scale is computed. If omitted, 90 degrees (west) is assumed.

daspectm(*zunits*,vfac,lat,long,az,radius) uses the last input to determine the radius of the sphere. radius can be one of the strings supported by km2deg, or it can be the (numerical) radius of the desired sphere in zunits. If omitted, the default radius of the Earth is used.

# Examples

### Elevation Map with Vertical Exaggeration

Show the elevation map of the Korean peninsula with a vertical exaggeration factor of 30.

Read data from the MAT-file containing information about the Korean peninsula.

```
load korea
```

Display the data grid.

```
[latlim,lonlim] = limitm(map,refvec);
worldmap(latlim,lonlim)
meshm(map,refvec,size(map),map)
demcmap(map)
view(3)
```
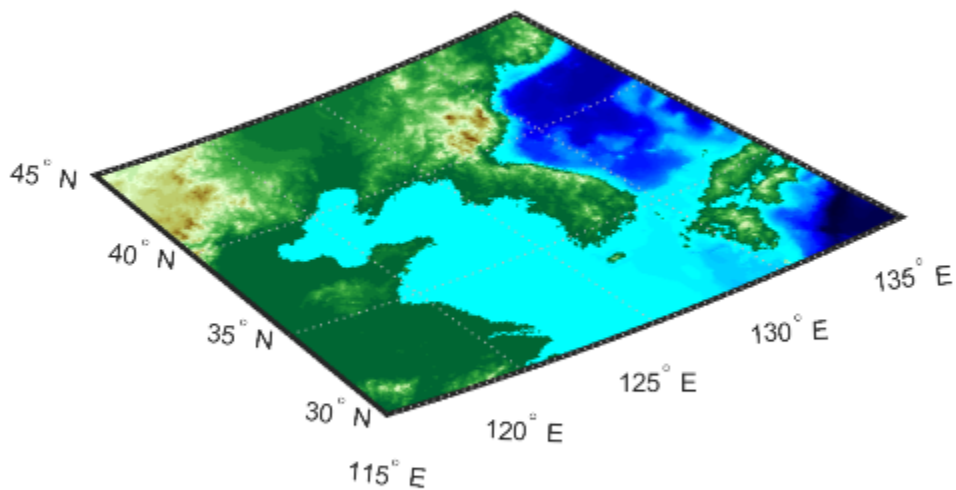
Set the vertical exaggeration factor.

```
daspectm('m',30)
tightmap
camlight
```
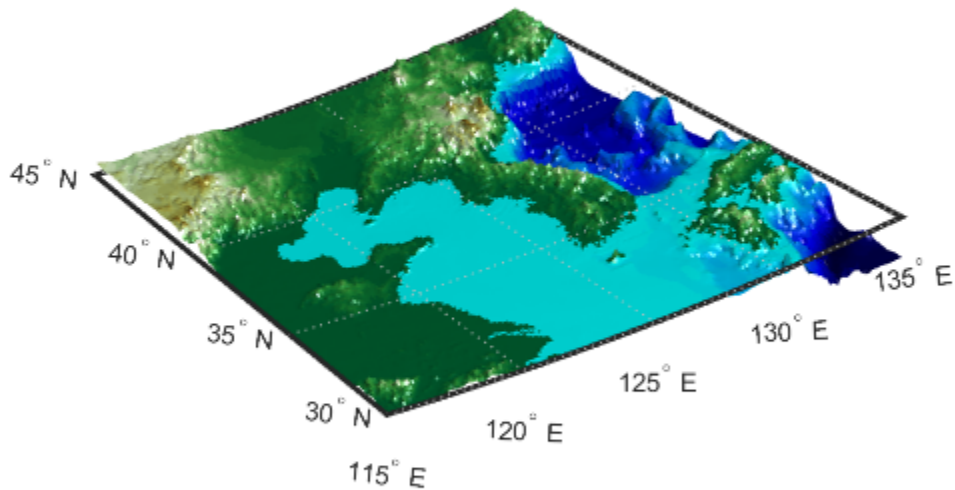
## Limitations

The relationship between the vertical and horizontal coordinates holds only as long as the geoid or scale factor properties of the map axes remain unchanged. If you change the scaling between geographic coordinates and projected axes coordinates, execute daspectm again.

## See Also

daspect | paperscale

# dcwdata

Read selected DCW worldwide basemap data

## Compatibility

dcwdata will be removed in a future release. The VMAP0 dataset has replaced DCW and can be accessed using vmap0data.

## Syntax

```
struct = dcwdata(library,latlim,lonlim,theme,topolevel)
struct = dcwdata(devicename,library,...)
[struct1, struct2,...] = dcwdata(...,{topolevel1,topolevel2,...})
```

## Description

struct = dcwdata(*library*,latlim,lonlim,theme,*topolevel*) reads data for the specified theme and topology level directly from the DCW CD-ROM. There are four CDs, one for each of the libraries: 'NOAMER' (North America), 'SASAUS' (Southern Asia and Australia), 'EURNASIA' (Europe and Northern Asia), and 'SOAMAFR' (South America and Africa). The desired theme is specified by a two-letter code string. A list of valid codes is displayed when an invalid code, such as '?', is entered. The region of interest can be given as a point latitude and longitude or as a region with two-element vectors of latitude and longitude limits. The units of latitude and longitude are degrees. The data covering the requested region is returned, but will include data extending to the edges of the 5-by-5 degree tiles. The result is returned as a Version 1 Mapping Toolbox display structure.

struct = dcwdata(*devicename*,*library*,...) specifies the logical device name of the CD-ROM for computers that do not automatically name the mounted disk.

[struct1, struct2,...] = dcwdata(...,{*topolevel1*,*topolevel2*,...}) reads several topology levels. The levels must be specified as a cell array with the entries 'patch', 'line', 'point', or 'text'. Entering {'all'} for the topology level argument is equivalent to {'patch', 'line', 'point', 'text'}. Upon output, the data

structures are returned in the output arguments by topology level in the same order as they were requested.

# Background

The Digital Chart of the World (DCW) is a detailed and comprehensive source of publicly available global vector data. It was digitized from the Operational Navigation Charts (scale 1:1,000,000) and Jet Navigation Charts (1:2,000,000), compiled by the U.S. Defense Mapping Agency (DMA) along with mapping agencies in Australia, Canada, and the United Kingdom. The digitized data was published on four CD-ROMS by the DMA and is distributed by the U.S. Geological Survey (USGS).

The DCW is out of print and has been succeeded by the Vector Map Level 0 (VMAP0).

The DCW organizes data into 17 different themes, such as political/oceans (PO), drainage (DN), roads (RD), or populated places (PP). The data is further tiled into 5-by-5 degree tiles and separated by topology level (patches, lines, points, and text).

# Examples

On a Macintosh computer,

```
s = dcwdata('NOAMER',41,-69,'?','patch');

??? Error using ==> dcwdata
Theme not present in library NOAMER
Valid two-letter theme identifiers are:
PO: Political/Oceans
PP: Populated Places
LC: Land Cover
VG: Vegetation
RD: Roads
RR: Railroads
UT: Utilities
AE: Aeronautical
DQ: Data Quality
DN: Drainage
DS: Supplemental Drainage
HY: Hypsography
HS: Supplemental Hypsography
```

```
CL: Cultural Landmarks
OF: Ocean Features
PH: Physiography
TS: Transportation Structure
POpatch = dcwdata('NOAMER',[41 44],[-72 -69],'PO','patch')
POpatch =
1x234 struct array with fields:
    type
    otherproperty
    tag
    altitude
    lat
    long
    tag2
    tag3
```

On an MS-DOS based operating system with the CD-ROM as the `'d:'` drive,

```
[RDtext,RDline] = dcwdata('d:','SASAUS',[-48 -34],[164 180],...
     'RD',{'text','line'});
```

On a UNIX® operating system with the CD-ROM mounted as `'\cdrom'`,

```
[POpatch,POline,POpoint,POtext] = dcwdata('\cdrom',...
     'EURNASIA',-48 ,164,'PO',{'all'});
```

## More About

### Tips

Latitudes and longitudes use WGS84 as a horizontal datum. Elevations are in feet above mean sea level. The data set does not contain bathymetric data.

Some DCW themes do not contain all topology levels. In those cases, empty matrices are returned.

The data is tagged with strings describing the objects. Some data is provided with alternate tags in `tag2` and `tag3` fields. These alternate tags contain information that supplements the standard tag, such as the names of political entities or values of elevation. The `tag2` field generally has the actual values or codes associated with the data. If the information in the `tag2` field expands to more verbose descriptions, these are provided in the `tag3` field.

Point data for which there are descriptions of both the type and the individual names of objects is returned twice within the structure. The first set is a collection of points of the same type with appropriate tag. The second is a set of individual points with the tag `'Individual Points'` and the name of the object in the `tag2` field.

Patches are broken at the tile boundaries. Setting the `EdgeColor` to `'none'` and plotting the lines gives the map a normal appearance.

The DCW was published in 1992 based on data compiled some years earlier. The political boundaries do not reflect recent changes such as the dissolution of the Soviet Union, Czechoslovakia, and Yugoslavia. In some cases, the boundaries of the successor nations are present as lower level political units. A new version, called VMAP0.

For information about the format of display structures, see "Version 1 Display Structures" on page 1-200 in the reference page for `displaym`.

## References

The format and the history of the DCW are described in reference [1] of the "Bibliography" at the end of this chapter.

## See Also

dcwgaz | dcwread | dcwrhead | displaym | extractm | mlayers | updategeostruct | vmap0data

# dcwgaz

Search DCW worldwide basemap gazette file

## Compatibility

`dcwgaz` will be removed in a future release. The VMAP0 dataset has replaced DCW and can be explored using `vmap0ui`.

## Syntax

```
dcwgaz(library,object)
dcwgaz(devicename,library,object)
mtextstruc = dcwgaz(...)
[mtextstruc,mpointstruc] = dcwgaz(...)
```

## Description

`dcwgaz(library,object)` searches the DCW library for items beginning with the *object* string. There are four CDs, one for each of the libraries: `'NOAMER'` (North America), `'SASAUS'` (Southern Asia and Australia), `'EURNASIA'` (Europe and Northern Asia), and `'SOAMAFR'` (South America and Africa). Items that exactly match or begin with the *object* string are displayed on screen.

`dcwgaz(devicename,library,object)` specifies the logical device name of the CD-ROM for computers that do not automatically name the mounted disk.

`mtextstruc = dcwgaz(...)` displays the matched items on screen and returns a Mapping Toolbox display structure with the matches as text entries.

`[mtextstruc,mpointstruc] = dcwgaz(...)` returns the matches in structures formatted both as text and as points.

## Background

In addition to the geographic data, the Digital Chart of the World (DCW) also includes an extensive gazette feature. The gazette is a collection of the names of geographic items

mentioned in the various themes of a DCW disk. One DCW disk can contain about 10,000 to 15,000 names. This function allows you to search the gazette for names beginning with a particular string.

## Examples

On a Macintosh computer,

```
s = dcwgaz('EURNASIA','apatin')

APATIN
s =
              type: 'text'
    otherproperty: {1x2 cell}
               tag: 'Built up area'
            string: 'APATIN'
          altitude: []
               lat: 45.6660
              long: 18.9830
```

On a UNIX operating system with the CD-ROM mounted as `'\cdrom'`,

```
[mtextstruc,mpointstruc] = ...
    dcwgaz('\cdrom','SOAMAFR', 'cape good')

Cape Goodenough
Cape Goodenough
Cape Goodenough
mtextstruc =
1x3 struct array with fields:
    type
    otherproperty
    tag
    string
    altitude
    lat
    long
mpointstruc =
1x3 struct array with fields:
    type
    otherproperty
    tag
    string
```

```
altitude
lat
long
```

# More About

### Tips

The search is not case sensitive. Items that match are those that begin with the *object* string. Spaces are significant.

## See Also
dcwdata | dcwread | dcwrhead | mlayers | updategeostruct

# dcwread

Read DCW worldwide basemap file

## Compatibility

dcwread will be removed in a future release. The VMAP0 dataset has replaced DCW and can be read using vmap0read.

## Syntax

```
dcwread(filepath,filename)
dcwread(filepath,filename,recordIDs)
dcwread(filepath,filename,recordIDs,field,varlen)
struc = dcwread(...)
[struc,field] = dcwread(...)
[struc,field,varlen] = dcwread(...)
[struc,field,varlen,description] = dcwread(...)
[struc,field,varlen,description,narrativefield] = dcwread(...)
```

## Description

dcwread reads a DCW file. The user selects the file interactively.

dcwread(*filepath*,*filename*) reads the specified file. The combination [*filepath filename*] must form a valid complete filename.

dcwread(*filepath*,*filename*,recordIDs) reads selected records or fields from the file. If recordIDs is a scalar or a vector of integers, the function returns the selected records. If recordIDs is a cell array of integers, all records of the associated fields are returned.

dcwread(*filepath*,*filename*,recordIDs,field,varlen) uses previously read field and variable-length record information to skip parsing the file header (see below).

struc = dcwread(...) returns the file contents in a structure.

[struc,field] = dcwread(...) returns the file contents and a structure describing the format of the file.

[struc,field,varlen] = dcwread(...) also returns a vector describing the fields that have variable-length records.

[struc,field,varlen,description] = dcwread(...) also returns a string describing the contents of the file.

[struc,field,varlen,description,narrativefield] = dcwread(...) also returns the name of the narrative file for the current file.

## Background

The Digital Chart of the World (DCW) uses binary files in a variety of formats. This function determines the format of the file and returns the contents in a structure. The field names of this structure are the same as the field names in the DCW file.

## Examples

The following examples use the Macintosh directory system and file separators for the pathname:

```
s = dcwread('NOAMER:DCW:NOAMER:','GRT')
s =
                   ID: 1
            DATA_TYPE: 'GEO'
                UNITS: 'O14'
            ELLIPSOID: 'WGS 84'
     ELLIPSOID_DETAIL: 'A=6378137,B=6356752 Meters'
       VERT_DATUM_REF: 'MEAN SEA LEVEL'
      VERT_DATUM_CODE: 'O15'
         SOUND_DATUM: 'MEAN SEA LEVEL'
     SOUND_DATUM_CODE: 'O15'
       GEO_DATUM_NAME: 'WGS 84'
       GEO_DATUM_CODE: 'WGE'
      PROJECTION_NAME: 'DECIMAL DEGREES'

s = dcwread('NOAMER:DCW:NOAMER:AE:','INT.VDT')
s =
5x1 struct array with fields:
```

```
          ID
       TABLE
       ATTRIBUTE
       VALUE
       DESCRIPTION
for i = 1:length(s); disp(s(i)); end
              ID: 1
           TABLE: 'AEPOINT.PFT'
       ATTRIBUTE: 'AEPTTYPE'
           VALUE: 1
     DESCRIPTION: 'Active civil'

              ID: 2
           TABLE: 'AEPOINT.PFT'
       ATTRIBUTE: 'AEPTTYPE'
           VALUE: 2
     DESCRIPTION: 'Active civil and military'
    ID: 3
           TABLE: 'AEPOINT.PFT'
       ATTRIBUTE: 'AEPTTYPE'
           VALUE: 3
     DESCRIPTION: 'Active military'

              ID: 4
           TABLE: 'AEPOINT.PFT'
       ATTRIBUTE: 'AEPTTYPE'
           VALUE: 4
     DESCRIPTION: 'Other'

              ID: 5
           TABLE: 'AEPOINT.PFT'
       ATTRIBUTE: 'AEPTTYPE'
           VALUE: 5
     DESCRIPTION: 'Added from ONC when not available from DAFIF'
s = dcwread('NOAMER:DCW:NOAMER:AE:','AEPOINT.PFT',1)
s =
              ID: 1
       AEPTTYPE: 4
       AEPTNAME: 'THULE AIR BASE'
        AEPTVAL: 251
       AEPTDATE: '19900502000000000000'
       AEPTICAO: '1261'
       AEPTDKEY: 'BR17652'
        TILE_ID: 94
```

```
      END_ID: 1

s = dcwread('NOAMER:DCW:NOAMER:AE:','AEPOINT.PFT',{1,2})
s =
4678x1 struct array with fields:
    ID
    AEPTTYPE
```

## More About

### Tips

This function reads all DCW files except index files (files with names ending in `'X'`), thematic index files (files with names ending in `'TI'`), and spatial index files (files with names ending in `'SI'`).

File separators are platform dependent. The *filepath* input must use appropriate file separators, which you can determine using the MATLAB `filesep` function.

### See Also
`dcwdata` | `dcwgaz` | `dcwrhead`

# dcwrhead

Read DCW worldwide basemap file headers

## Compatibility

dcwrhead will be removed in a future release. The VMAP0 dataset has replaced DCW and the header data can be read using vmap0rhead.

## Syntax

```
dcwrhead
dcwrhead(filepath,filename)
dcwrhead(filepath,filename,fid)
dcwrhead(...)
str = dcwrhead(...)
```

## Description

dcwrhead allows the user to select the header file interactively.

dcwrhead(*filepath*,*filename*) reads from the specified file. The combination [filepath filename] must form a valid complete filename.

dcwrhead(*filepath*,*filename*,fid) reads from the already open file associated with fid.

dcwrhead(...) with no output arguments displays the formatted header information on the screen.

str = dcwrhead(...) returns a string containing the DCW header.

## Background

The Digital Chart of the World (DCW) uses header strings in most files to document the contents and format of that file. This function reads the header string, displays a formatted version in the command window, or returns it as a string.

## Examples

The following example uses the Macintosh file separators and pathname:

```
dcwrhead('NOAMER:DCW:NOAMER:AE:','AEPOINT.PFT')
Aeronautical Points
AEPOINT.DOC
ID=I,   1,P,Row Identifier,-,-,
AEPTTYPE=I,  1,N,Airport Type,INT.VDT,-,
AEPTNAME=T, 50,N,Airport Name,-,-,
AEPTVAL=I,  1,N,Airport Elevation Value,-,-,
AEPTDATE=D,  1,N,Aeronautical Information Date,-,-,
AEPTICAO=T,  4,N,International Civil Organization Number,-,-,
AEPTDKEY=T,  7,N,DAFIF Reference Number,-,-,
TILE_ID=S,  1,F,Tile Reference Identifier,-,AEPOINT.PTI,
END_ID=I  1,F,Entity Node Primitive Foreign Key,-,-,


s = dcwrhead('NOAMER:DCW:NOAMER:AE:','AEPOINT.PFT')
s =
;Aeronautical Points;AEPOINT.DOC;ID=I,   1,P,Row
Identifier,-,-,:AEPTTYPE=I,  1,N,Airport
Type,INT.VDT,-,:AEPTNAME=T, 50,N,Airport Name,-,-,:AEPTVAL=I,
1,N,Airport Elevation Value,-,-,:AEPTDATE=D,  1,N,Aeronautical
Information Date,-,-,:AEPTICAO=T,  4,N,International Civil
Organization Number,-,-,:AEPTDKEY=T,  7,N,DAFIF Reference
Number,-,-,:TILE_ID=S,  1,F,Tile Reference
Identifier,-,AEPOINT.PTI,:END_ID=I  1,F,Entity Node Primitive
Foreign Key,-,-,:;
```

## More About

### Tips

This function reads all DCW files except index files (files with names ending in `'X'`), thematic index files (files with names ending in `'TI'`), and spatial index files (files with names ending in `'SI'`).

File separators are platform dependent. The *filepath* input must use appropriate file separators, which you can determine using the MATLAB `filesep` function.

### See Also
dcwdata | dcwgaz | dcwread

# defaultm

Initialize or reset map projection structure

## Syntax

```
mstruct = defaultm(projid)
mstruct = defaultm(mstruct)
```

## Description

mstruct = defaultm(*projid*) initializes a map projection structure. The string projid should match one of the entries in the last column of the table displayed by the maps function. The output mstruct is a map projection structure. It is a scalar structure whose fields correspond to "Map Axes Object Properties" on page 1-51.

mstruct = defaultm(mstruct) checks an existing map projection structure, sets empty properties, and adjusts dependent properties. The Origin, FLatLimit, FLonLimit, MapLatLimit, and MapLonLimit properties may be adjusted for compatibility with each other and with the MapProjection property and (in the case of UTM or UPS) the Zone property.

With defaultm, you can construct a map projection structure (mstruct) that contains all the information needed to project and unproject geographic coordinates using mfwdtran, minvtran, vfwdtran, or vintran without creating a map axes or making any use at all of MATLAB graphics. Relevant parameters in the mstruct include the projection name, angle units, zone (for UTM or UPS), origin, aspect, false easting, false northing, and (for conic projections) the standard parallel or parallels. In very rare cases you might also need to adjust the frame limit (FLatLimit and FLonLimit) or map limit (MapLatLimit and MapLonLimit) properties.

You should make exactly two calls to defaultm to set up your mstruct, using the following sequence:

1 Construct a provisional version containing default values for the projection you've selected: mstruct = defaultm(*projection*);

2 Assign appropriate values to mstruct.angleunits, mstruct.zone, mstruct.origin, etc.

**3** Set empty properties and adjust interdependent properties as needed to finalize your map projection structure: `mstruct = defaultm(mstruct);`

If you've set field *prop1* of mstruct to *value1*, field *prop2* to *value2*, and so forth, then the following sequence

```
mstruct = defaultm(projection);
mstruct.prop1 = value1;
mstruct.prop2 = value2;
...
mstruct = defaultm(mstruct);
```

produces exactly the same result as the following:

```
f = figure;
ax = axesm(projection, prop1, value1, prop2, value2, ...);
mstruct = getm(ax);
close(f)
```
but it avoids the use of graphics and is more efficient.

---

**Note:** Angle-valued properties are in degrees by default. If you want to work in radians instead, you can make the following assignment in between your two calls to `defaultm`:

```
mstruct.angleunits = 'radians';
```

You must also use values in radians when assigning any angle-valued properties (such as `mstruct.origin`, `mstruct.parallels`, `mstruct.maplatlimit`, `mstruct.maplonlimit`, etc.).

---

See the Mapping Toolbox User's Guide section on "Working in UTM Without a Map Axes" for information and an example showing the use of `defaultm` in combination with UTM.

## Examples

Create an empty map projection structure for a Mercator projection:

```
mstruct = defaultm('mercator')

mstruct =
    mapprojection: 'mercator'
```

```
           zone: []
     angleunits: 'degrees'
         aspect: 'normal'
   falseeasting: []
  falsenorthing: []
    fixedorient: []
          geoid: [1 0]
     maplatlimit: []
     maplonlimit: []
    mapparallels: 0
      nparallels: 1
          origin: []
      scalefactor: []
         trimlat: [-86 86]
         trimlon: [-180 180]
           frame: []
           ffill: 100
       fedgecolor: [0 0 0]
       ffacecolor: 'none'
        flatlimit: []
       flinewidth: 2
        flonlimit: []
            grid: []
        galtitude: Inf
          gcolor: [0 0 0]
       glinestyle: ':'
       glinewidth: 0.5000
  mlineexception: []
        mlinefill: 100
       mlinelimit: []
    mlinelocation: []
     mlinevisible: 'on'
   plineexception: []
        plinefill: 100
       plinelimit: []
    plinelocation: []
     plinevisible: 'on'
        fontangle: 'normal'
        fontcolor: [0 0 0]
         fontname: 'helvetica'
         fontsize: 9
        fontunits: 'points'
       fontweight: 'normal'
      labelformat: 'compass'
```

```
     labelrotation: 'off'
         labelunits: []
     meridianlabel: []
     mlabellocation: []
     mlabelparallel: []
        mlabelround: 0
      parallellabel: []
      plabellocation: []
      plabelmeridian: []
        plabelround: 0
```

Now change the map origin to [0 90 0], and fill in default projection parameters accordingly:

```
mstruct.origin = [0 90 0];
mstruct = defaultm(mstruct)

mstruct =
      mapprojection: 'mercator'
               zone: []
         angleunits: 'degrees'
             aspect: 'normal'
        falseeasting: 0
       falsenorthing: 0
         fixedorient: []
               geoid: [1 0]
         maplatlimit: [-86 86]
         maplonlimit: [-90 270]
         mapparallels: 0
          nparallels: 1
              origin: [0 90 0]
          scalefactor: 1
             trimlat: [-86 86]
             trimlon: [-180 180]
               frame: 'off'
               ffill: 100
           fedgecolor: [0 0 0]
           ffacecolor: 'none'
            flatlimit: [-86 86]
           flinewidth: 2
            flonlimit: [-180 180]
                grid: 'off'
            galtitude: Inf
               gcolor: [0 0 0]
            glinestyle: ':'
```

```
      glinewidth: 0.5
   mlineexception: []
         mlinefill: 100
        mlinelimit: []
     mlinelocation: 30
      mlinevisible: 'on'
   plineexception: []
         plinefill: 100
        plinelimit: []
     plinelocation: 15
      plinevisible: 'on'
         fontangle: 'normal'
         fontcolor: [0 0 0]
          fontname: 'Helvetica'
          fontsize: 10
         fontunits: 'points'
        fontweight: 'normal'
       labelformat: 'compass'
     labelrotation: 'off'
        labelunits: 'degrees'
     meridianlabel: 'off'
    mlabellocation: 30
    mlabelparallel: 86
       mlabelround: 0
     parallellabel: 'off'
    plabellocation: 15
    plabelmeridian: -90
       plabelround: 0
```

## See Also

axesm | gcm | mfwdtran | minvtran | setm

# deg2km

Convert distance from degrees to kilometers

## Syntax

```
km = deg2km(deg)
km = deg2km(deg,radius)
km = deg2km(deg,sphere)
```

## Description

`km = deg2km(deg)` converts distances from degrees to kilometers as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`km = deg2km(deg,radius)` converts distances from degrees to kilometers as measured along a great circle on a sphere having the specified radius. `radius` must be in units of kilometers.

`km = deg2km(deg,sphere)` converts distances from degrees to kilometers, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

## See Also

`deg2nm` | `degtorad` | `deg2sm` | `km2deg` | `sm2deg`

# deg2nm

Convert distance from degrees to nautical miles

## Syntax

```
nm = deg2nm(deg)
nm = deg2nm(deg,radius)
nm = deg2nm(deg,sphere)
```

## Description

nm = deg2nm(deg) converts distances from degrees to nautical miles, as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

nm = deg2nm(deg,radius) converts distances from degrees to nautical miles, as measured along a great circle on a sphere having the specified radius. radius must be in units of nautical miles.

nm = deg2nm(deg,sphere) converts distances from degrees to nautical miles, as measured along a great circle on a sphere approximating an object in the Solar System. sphere may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive.

## Examples

A degree of arc length is about 60 nautical miles:

```
deg2nm(1)

ans =
   60.0405
```

This is not true on Mercury, of course:

```
deg2nm(1,'mercury')
```

```
ans =
   22.9852
```

## See Also
degtorad | deg2sm | km2deg | sm2deg

# deg2sm

Convert distance from degrees to statute miles

## Syntax

```
sm = deg2sm(deg)
sm = deg2sm(deg,radius)
sm = deg2sm(deg,sphere)
```

## Description

`sm = deg2sm(deg)` converts distances from degrees to statute miles, as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`sm = deg2sm(deg,radius)` converts distances from degrees to statute miles, as measured along a great circle on a sphere having the specified radius. `radius` must be in units of statute miles.

`sm = deg2sm(deg,sphere)` converts distances from degrees to statute miles, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

## See Also
`deg2nm` | `degtorad` | `km2deg` | `sm2deg`

# degrees2dm

Convert degrees to degrees-minutes

## Syntax

```
DM = degrees2dm(angleInDegrees)
```

## Description

DM = `degrees2dm(angleInDegrees)` converts angles from values in degrees which may include a fractional part (sometimes called "decimal degrees") to degree-minutes representation. The input should be a real-valued column vector. Given N-by-1 input, DM will be N-by-2, with one row per input angle. The first column of DM contains the "degrees" element and is integer-valued. The second column contains the "minutes" element and may have a nonzero fractional part. In any given row of DM, the sign of the first nonzero element indicates the sign of the overall angle. A positive number indicates north latitude or east longitude; a negative number indicates south latitude or west longitude. Any remaining elements in that row will have nonnegative values.

## Examples

```
angleInDegrees = [ 30.8457722555556; ...
                  -82.0444189583333; ...
                   -0.504756513888889;...
                    0.004116666666667];
dm = degrees2dm(angleInDegrees)

dm =
   30.000000000000000   50.746335333336106
  -82.000000000000000    2.665137499997741
                    0  -30.285390833333338
                    0    0.247000000000020
```

## See Also

dm2degrees | deg2rad | degrees2dms | rad2deg

# degrees2dms

Convert degrees to degrees-minutes-seconds

## Syntax

```
DMS = degrees2dms(angleInDegrees)
```

## Description

`DMS = degrees2dms(angleInDegrees)` converts angles from values in degrees which may include a fractional part (sometimes called "decimal degrees") to degree-minutes-seconds representation. The input should be a real-valued column vector. Given N-by-1 input, `DMS` will be N-by-3, with one row per input angle. The first column of `DMS` contains the "degrees" element and is integer-valued. The second column contains the "minutes" element and is integer valued. The third column contains the "seconds" element, and can have a nonzero fractional part. In any given row of `DMS`, the sign of the first nonzero element indicates the sign of the overall angle. A positive number indicates north latitude or east longitude; a negative number indicates south latitude or west longitude. Any remaining elements in that row will have nonnegative values.

## Examples

Convert four angles from values in degrees to degree-minutes-seconds representation.

```
format long g
angleInDegrees = [ 30.8457722555556; ...
                   -82.0444189583333; ...
                    -0.504756513888889; ...
                     0.004116666666667];
dms = degrees2dms(angleInDegrees)
```

The output appears as follows:

```
dms =
                30                50       44.7801200001663
               -82                 2       39.9082499998644
                 0               -30       17.1234500000003
```

```
                    0                   0       14.8200000000012
```

Convert angles to a string, with each angle on its own line.

```
nonnegative = all((dms >= 0),2);
hemisphere = repmat('N', size(nonnegative));
hemisphere(~nonnegative) = 'S';
absvalues = num2cell(abs(dms'));
values = [absvalues; num2cell(hemisphere')];
str = sprintf('%2.0fd%2.0fm%7.5fs%s\n', values{:})
```

The output appears as follows:

```
str =
    30d50m44.78012sN
    82d 2m39.90825sS
     0d30m17.12345sS
     0d 0m14.82000sN
```

Split the string into cells as delimited by the newline character, then return to the original values using `str2angle`.

```
newline = sprintf('\n');
C = textscan(str,'%s',-1,'delimiter',newline);
a = deal(C{:});
for k = 1:numel(a)
    str2angle(a{k})
end
```

The output appears as follows:

```
ans =
        30.8457722555556

ans =
       -82.0444189583333

ans =
      -0.504756513888889

ans =
     0.00411666666666667
```

## See Also

dms2degrees | deg2rad | degrees2dm | rad2deg

# degtorad

Convert angles from degrees to radians

## Syntax

```
angleInRadians = degtorad(angleInDegrees)
```

## Description

`angleInRadians = degtorad(angleInDegrees)` converts angle units from degrees to radians. This is both an angle conversion function and a distance conversion function, since arc length can be a measure of distance in either radians or degrees, provided that the radius is known.

## Examples

Show that there are 2π radians in a full circle:

```
2*pi - degtorad(360)

ans =
     0
```

## See Also

fromDegrees | fromRadians | toDegrees | toRadians | radtodeg

# demcmap

Colormaps appropriate to terrain elevation data

## Syntax

```
demcmap(Z)
demcmap(Z,ncolors)
demcmap(Z,ncolors,cmapsea,cmapland)

demcmap('inc',Z,deltaz)
demcmap('inc',Z,deltaz,cmapsea,cmapland)

[cmap,climits] = demcmap( ___ )
```

## Description

`demcmap(Z)` sets the colormap and color axis limits based on the elevation data limits derived from input argument`Z`.

- The default colormap assigns shades of green and brown for positive elevations, and various shades of blue for negative elevation values below sea level.
- The number of colors assigned to land and to sea are in proportion to the ranges in terrain elevation and bathymetric depth and total 64 by default. The color axis limits are computed such that the interface between land and sea maps to the zero elevation contour.
- The colormap is applied to the current figure and the color axis limits are applied to the current axes.

`demcmap(Z,ncolors)` creates a colormap of length `ncolors`.

`demcmap(Z,ncolors,cmapsea,cmapland)` assigns `cmapsea` and `cmapland` to elevations below and above sea level respectively.

`demcmap('inc',Z,deltaz)` chooses number of colors and color axis limits such that each color approximately represents the increment of elevation `deltaz`.

- The literal string `'inc'` signals `demcmap` that the first argument after `Z` will be `deltaz`.

demcmap('inc',Z,deltaz,cmapsea,cmapland) assigns cmapsea and cmapland to elevations below and above sea level respectively.

[cmap,climits] = demcmap( ___ ) returns colormap cmap and color axis limit climits, using any of the above syntaxes, but does not apply them to figure or axes properties.

- Even if only one output argument is specified, no change occurs to figure or axes properties.

# Examples

### Displaying Elevation Data With Default Colormap

Explicitly determine maximum and minimum values of elevation data matrix

```
load topo
axesm hatano
meshm(topo,topolegend)
zlimits = [min(topo(:)) max(topo(:))];
demcmap(zlimits);
colorbar;
```
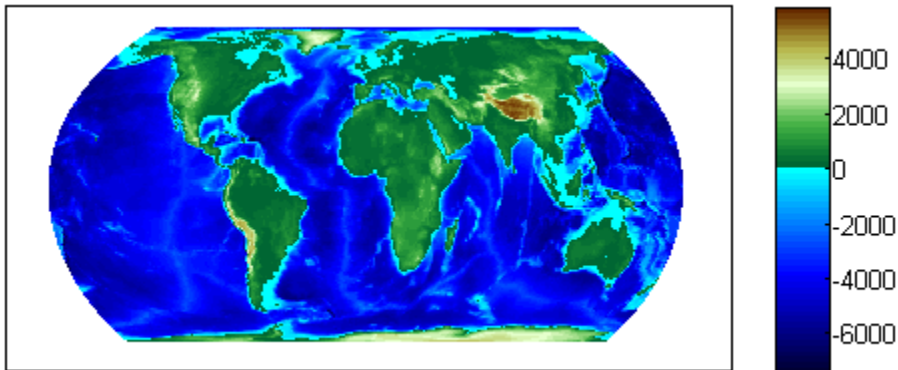


### Defining Custom Land And Sea Colormaps

Custom RGB colormaps, cmapssea; cmapland , of differing lengths are used to populate figure colormap by interpolation. The colors in each colormap map to the land and sea

regions of the map. Fewer colors have been specified in total than the default number of 64. `demcmap` determines maximum and minimum elevation data limits internally as shown in the below example when the first argument is the elevation data grid.

```
load topo   % grid of elevation data
axesm hatano
meshm(topo,topolegend)
cmapsea  = [.8  0 .8;  0 0 .8];
cmapland = [.7  0  0; .8 .8 0; 1  1 .8 ];
demcmap(topo,32,cmapsea,cmapland)
colorbar;
```
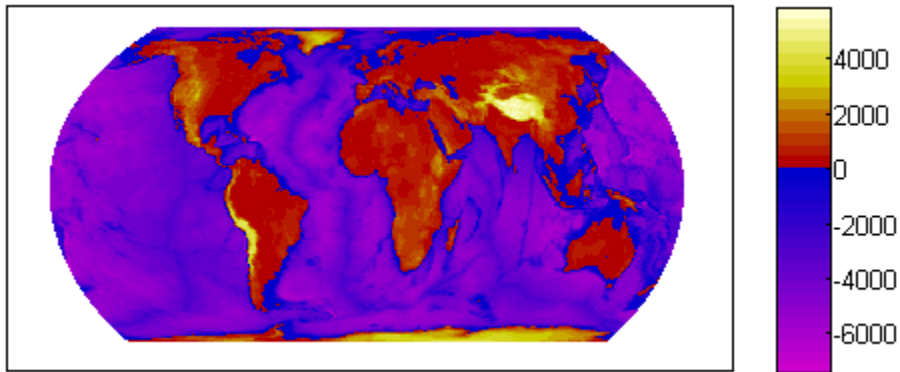


**Colormap in Which Each Color Approximates a User Defined Increment**

The following `demcmap` example controls the color quantization by choosing an optimal number of colors such that each color represents an elevation increment of approximately 2000 .

```
load topo
R = georasterref('RasterSize', size(topo),'Latlim', [-90 90], 'Lonlim', [0 360]);
figure('Color','white')
worldmap('world')
geoshow(topo, R, 'DisplayType', 'texturemap')
demcmap('inc',[max(topo(:)) min(topo(:))],2000);
colorbar
```
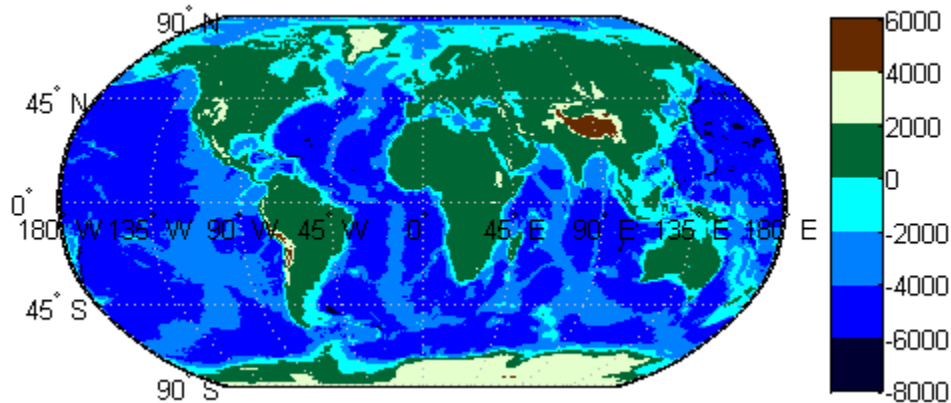
## Input Arguments

### `Z` — Terrain elevation limits
vector | matrix

Terrain elevation limits specified as a vector or matrix. If `Z` is a 2 element vector, then it specifies the minimum and maximum limits of terrain elevation data; ordering is not important. If `Z` is a matrix, then it specifies an elevation grid in which positive and negative values represent points above and below sea level respectively. The above two syntaxes for `demcmap` are identical in their effect on the figure colormap and axes properties.

Data Types: `single` | `double` | `int8` | `int32` | `uint8` | `uint16` | `uint32`

### `ncolors` — Number of colors in colormap
64 (default) | scalar

Number of colors in the colormap specified as a scalar. It defines the number of rows `m` in the mx3 RGB matrix of the figure colormap.

Data Types: `double`

### `cmapsea,cmapland` — RGB colormap matrices
matrix

- RGB colormaps specified as mx3 arrays containing any number of rows. The two colormaps need not be equal in length. They serve as the basis set for populating the figure colormap by interpolation.
- `cmapsea` and `cmapland` replace the default colormap. The default colormap for land or sea can be retained by providing an empty matrix in place of either colormap matrix.

That part of the figure colormap assigned to negative elevations is derived from `cmapssea`; `cmapland` plays a similar role for positive elevations.

Data Types: `double`

### `deltaz` — Increment of elevation
scalar

The increment of elevation specified as a scalar. The color quantization of the default or user supplied colormap is adjusted such that each discrete color approximately represents a `deltaz` increment in elevation.

Data Types: `double`

## Output Arguments

### `cmap` — RGB colormap
matrix

RGB colormap returned as a matrix constructed for the figure colormap. Supply output arguments when you want to obtain the colormap and color axis limits without applying them automatically to the figure or axes properties. These properties remain unchanged even if only one output (`cmap`) is specified.

Data Types: `double`

### `climits` — Color axis limits
vector

Color axis limits returned as a vector. `climits` may differ somewhat from those derived from input argument `Z` due to the quantization which results from fitting a limited number of colors over the range limit of the elevation data.

Supply output arguments when you want to obtain the colormap and color axis limits without applying them automatically to the figure or axes.

Data Types: `double`

# More About

### Algorithms

If the elevation grid data contains both positive and negative values, then the computed colormap, `cmap`, has a "sea" partition of length `nsea` and "land" partition of length `nland`. The sum of `nsea` and `nland` equals the total number of entries in the computed colormap. The actual values of `nsea` and `nland` depend upon the number of entries and the relative range of the negative and positive limits of the elevation data. The sea partition consists of rows 1 through `nsea`, and the land partition consists of rows `nsea` + 1 through `ncolors`. The sea and land partitions of the figure colormap are populated with colors interpolated from the basis RGB colormaps, `cmapsea` and `cmapland`. In the figure below, the sea and land 3x3 RGB colormaps shown are the default colors used by `demcmap` to populate the figure colormap when no user specified colormaps are provided.

demcmap([100 -100], 24)
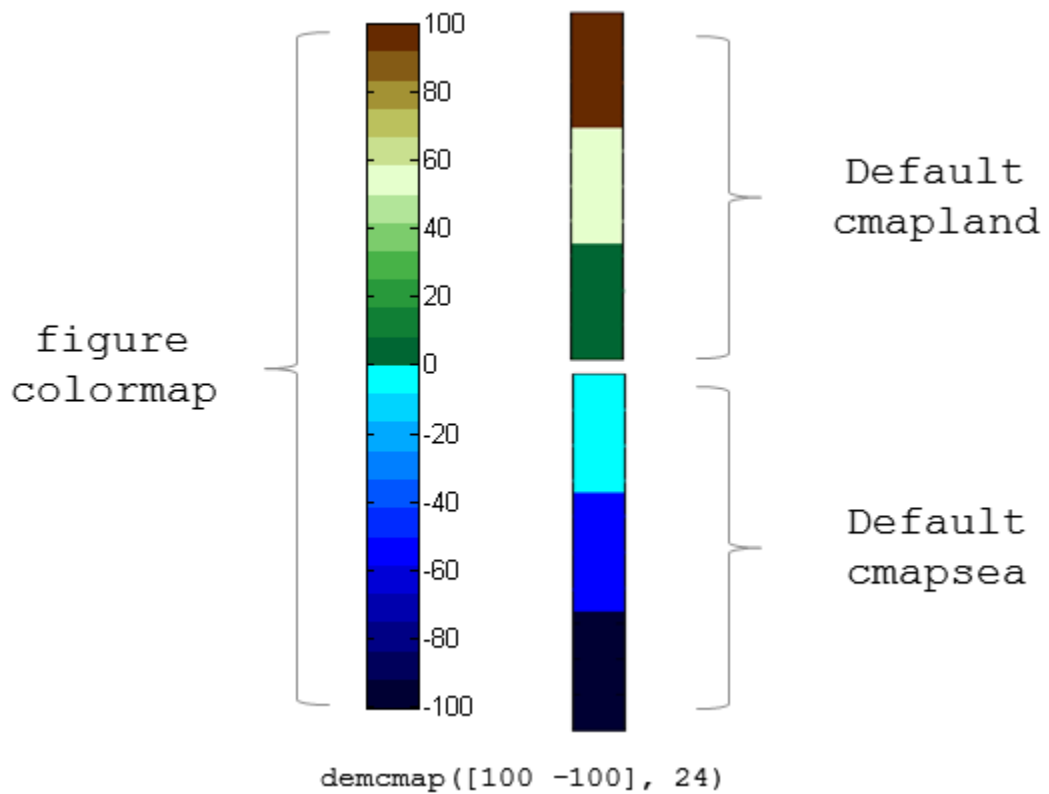
If the elevation grid data contains only positive or negative values, then the figure colormap is derived solely from the corresponding sea or land colormap.

## See Also

caxis | colormap | meshlsrm | meshm | surflsrm | surfm

# departure

Departure of longitudes at specified latitudes

## Syntax

```
dist = departure(long1,long2,lat)
dist = departure(long1,long2,lat,ellipsoid)
dist = departure(long1,long2,lat,units)
dist = departure(long1,long2,lat,geoid,units)
```

## Description

`dist = departure(long1,long2,lat)` computes the departure distance from `long1` to `long2` at the input latitude `lat`. Departure is the distance along a specific parallel between two meridians. The output `dist` is returned in degrees of arc length on a sphere.

`dist = departure(long1,long2,lat,ellipsoid)` computes the departure assuming that the input points lie on the ellipsoid defined by the input `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`.

`dist = departure(long1,long2,lat,units)` uses the input string *units* to define the angle units of the input and output data. In this form, the departure is returned as an arc length in the units specified by *units*. If *units* is omitted, `'degrees'` is assumed.

`dist = departure(long1,long2,lat,geoid,units)` is a valid calling form. In this case, the departure is computed in the same units as the semimajor axes of the ellipsoid.

## Definitions

*Departure* is the distance along a parallel between two points. Whereas a degree of latitude is always the same distance, a degree of longitude is different in length at different latitudes. In practice, this distance is usually given in nautical miles.

## Examples

On a spherical Earth, the departure is proportional to the cosine of the latitude:

```
distance = departure(0, 10, 0)

distance =
    10

distance = departure(0, 10, 60)

distance =
     5
```

When an ellipsoid is used, the result is more complicated. The distance at 60° is not exactly twice the 0° value:

```
distance = departure(0, 10, 0, referenceEllipsoid('earth', 'nm'))

distance =
  601.0772
distance = departure(0, 10, 60, referenceEllipsoid('earth', 'nm'))

distance =
  299.7819
```

## See Also
distance | stdm

# displaym

Display geographic data from display structure

## Syntax

```
displaym(displaystruct)
displaym(displaystruct,str)
displaym(displaystruct,strings)
displaym(displaystruct,strings,searchmethod)
h = displaym(displaystruct)
```

## Description

displaym(displaystruct) projects the data contained in the input displaystruct, a Version 1 Mapping Toolbox display structure, in the current axes. The current axes must be a map axes with a valid map definition. See the remarks about "Version 1 Display Structures" on page 1-200 below for details on the contents of display structures.

displaym(displaystruct,str) displays the vector data elements of displaystruct whose 'tag' fields contains strings beginning with the string str. Vector data elements are those whose 'type' field is either 'line' or 'patch'. The string match is case-insensitive.

displaym(displaystruct,strings) displays the vector data elements of displaystruct whose 'tag' field matches begins with one of the elements (or rows) of strings. strings is a cell array of strings (or a 2-D character array). In the case of character array, trailing blanks are stripped from each row before matching.

displaym(displaystruct,strings,searchmethod) controls the method used to match the values of the tag field in displaystruct, as follows:

- 'strmatch' — Search for matches at the beginning of the tag
- 'findstr' — Search within the tag
- 'exact' — Search for exact matches

Note that when `searchmethod` is specified the search is case-sensitive.

`h = displaym(displaystruct)` returns handles to the graphic objects created by `displaym`.

---

**Note:** The type of *display structure* accepted by `displaym` is not the same as a *geographic data structure* (geostructs and mapstructs). introduced in Mapping Toolbox Version 2. Use `geoshow` or `mapshow` instead of `displaym` to display geostructs or mapstructs—created using `shaperead` and `gshhs`, for example. For more information, see "Mapping Toolbox Geographic Data Structures".

---

# More About

### Tips

The following section documents the contents of display structures.

## Version 1 Display Structures

A display structure is a MATLAB structure array with a specific set of fields:

- A `tag` field names an individual feature or object
- A `type` field specifies a MATLAB graphics object type (`'line'`, `'patch'`, `'surface'`, `'text'`, or `'light'`) or has the value `'regular'`, specifying a regular data grid
- `lat` and `long` fields contain coordinate vectors of latitudes and longitudes, respectively
- An `altitude` field contains a vector of vertical coordinate values
- A `string` property contains text to be displayed if `type` is `'text'`
- MATLAB graphics properties are specified explicitly, on a per-feature basis, in an `otherproperty` field

The choice of options for the `type` field reveals that a display structure can contain

- Vector geodata (`type` is `'line'` or `'patch'`)
- Raster geodata (`type` is `'surface'` or `'regular'`)
- Graphic objects (`type` is `'text'` or `'light'`)

The following table indicates which fields are used in the six types of display structures:

| Field Name | Type 'light' | Type 'line' | Type 'patch' | Type 'regular' | Type 'surface' | Type 'text' |
|---|---|---|---|---|---|---|
| type | • | • | • | • | • | • |
| tag | • | • | • | • | • | • |
| lat | • | • | • | | • | • |
| long | • | • | • | | • | • |
| map | | | | • | • | |
| maplegend | | | | • | | |
| meshgrat | | | | • | | |
| string | | | | | | • |
| altitude | • | • | • | • | • | • |
| otherproperty | • | • | • | • | • | • |

Some fields can contain empty entries, but each indicated field must exist for the objects in the struct array to be displayed correctly. For instance, the `altitude` field can be an empty matrix and the `otherproperty` field can be an empty cell array.

The `type` field must be one of the specified map object types: `'line'`, `'patch'`, `'regular'`, `'surface'`, `'text'`, or `'light'`.

The `tag` field must be a string different from the `type` field usually containing the name or kind of map object. Its contents must not be equal to the name of the object type (i.e., line, surface, text, etc.).

The `lat`, `long`, and `altitude` fields can be scalar values, vectors, or matrices, as appropriate for the map object type.

The `map` field is a data grid. If `map` is a regular data grid, `maplegend` is its corresponding referencing vector, and `meshgrat` is a two-element vector specifying the graticule mesh size. If `map` is a geolocated data grid, `lat` and `long` are the matrices of latitude and longitude coordinates.

The `otherproperty` field is a cell array containing any additional display properties appropriate for the map object. Cell array entries can be a line specification string, such as `'r+'`, or property name/property value pairs, such as `'color'`,`'red'`. If the

`otherproperty` field is left as an empty cell array, default colors are used in the display of lines and patches based on the `tag` field.

---

**Note:** In some cases you can use the `geoshow` function as a direct alternative to `displaym`. It accepts display structures of type `line` and `patch`.

---

## See Also

`extractm` | `geoshow` | `mapshow` | `mlayers` | `updategeostruct`

# dist2str

Format distance strings

## Syntax

```
str = dist2str(distin)
str = dist2str(dist,format)
str = dist2str(dist,format,units)
str = dist2str(dist,format,digits)
str = dist2str(dist,format,units,digits)
```

## Description

`str = dist2str(distin)`converts a numerical vector of distances in kilometers, `distin`, to a string matrix. The output string matrix is useful for the display of distances.

`str = dist2str(dist,format)` uses the `format` string to specify the notation to be used for the string matrix. If blank or `'none'`, the result is a simple numerical representation (no indicator for positive distances, minus signs for negative distances). The only other format is `'pm'` (for *plus-minus*) prefixes a + for positive distances.

`str = dist2str(dist,format,units)` defines the units in which the input distances are supplied, and which are encoded in the string matrix. Units must be one of the following: `'feet'`, `'kilometers'`, `'meters'`, `'nauticalmiles'`, `'statutemiles'`, `'degrees'`, or `'radians'`. Note that statute miles are encoded as `'mi'` in the string matrix, whereas in most Mapping Toolbox functions, `'mi'` indicates international miles. If omitted or blank, `'kilometers'` is assumed.

`str = dist2str(dist,format,digits)` or `str = dist2str(dist,format,units,digits)` uses the input `digits` to determine the number of decimal digits in the output matrix. `digits = -2` uses accuracy in the hundredths position, `digits = 0` uses accuracy in the units position. Default is `digits = -2`. For further discussion of specifying `digits`, see `roundn`.

The purpose of this function is to make distance-valued variables into strings suitable for map display.

## Examples

Create a vector of values and convert to strings:

```
d = [-3.7 2.95 87];
str = dist2str(d,'none','km')

str =
-3.70 km
 2.95 km
87.00 km
```

Now change the units to nautical miles, add plus signs to positive values, and truncate to the tenths $(10^{-1})$ slot:

```
str = dist2str(d,'pm','nm',-1)

str =
 -3.7 nm
 +3.0 nm
+87.0 nm
```

## See Also

```
angl2str | roundn
```

# distance

Distance between points on sphere or ellipsoid

## Syntax

```
[arclen,az] = distance(lat1,lon1,lat2,lon2)
[arclen,az] = distance(lat1,lon1,lat2,lon2,ellipsoid)
[arclen,az] = distance(lat1,lon1,lat2,lon2,units)
[arclen,az] = distance(lat1,lon1,lat2,lon2,ellipsoid,units)
[arclen,az] = distance(track,...)
[arclen,az] = distance(pt1,pt2)
[arclen,az] = distance(pt1,pt2,ellipsoid)
[arclen,az] = distance(pt1,pt,units)
[arclen,az] = distance(pt1,pt2,ellipsoid,units)
[arclen,az] = distance(track,pt1,...)
```

## Description

`[arclen,az] = distance(lat1,lon1,lat2,lon2)` computes the lengths, `arclen`, of the great circle arcs connecting pairs of points on the surface of a sphere. In each case, the shorter (minor) arc is assumed. The function can also compute the azimuths, `az`, of the second point in each pair with respect to the first (that is, the angle at which the arc crosses the meridian containing the first point). The input latitudes and longitudes, `lat1`, `lon1`, `lat2`, `lon2`, can be scalars or arrays of equal size and must be expressed in degrees. `arclen` is expressed in degrees of arc and will have the same size as the input arrays. `az` is measured clockwise from north, in units of degrees. When given a combination of scalar and array inputs, the scalar inputs are automatically expanded to match the size of the arrays.

`[arclen,az] = distance(lat1,lon1,lat2,lon2,ellipsoid)` computes geodesic arc length and azimuth assuming that the points lie on the reference ellipsoid defined by the input `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The output, `arclen`, is expressed in the same length units as the semimajor axis of the ellipsoid.

1-205

`[arclen,az] = distance(lat1,lon1,lat2,lon2,`*`units`*`)` uses the input string `units` to define the angle unit of the outputs `arclen` and `az` and the input latitude-longitude coordinates. `units` may equal `'degrees'` (the default value) or `'radians'`.

`[arclen,az] = distance(lat1,lon1,lat2,lon2,ellipsoid,`*`units`*`)` uses the `units` string to specify the units of the latitude-longitude coordinates, but the output range has the same units as the semimajor axis of the ellipsoid.

`[arclen,az] = distance(track,...)` uses the input string `track` to specify either a great circle/geodesic or a rhumb line arc. If `track` equals `'gc'` (the default value), then great circle distances are computed on a sphere and geodesic distances are computed on an ellipsoid. If `track` equals `'rh'`, then rhumb line distances are computed on either a sphere or ellipsoid.

`[arclen,az] = distance(pt1,pt2)` accepts *N*-by-2 coordinate arrays `pt1` and `pt2` such that `pt1 = [lat1 lon1]` and `pt2 = [lat2 lon2]`, where `lat1`, `lon1`, `lat2`, and `lon2` are column vectors. It is equivalent to `arclen = distance(pt1(:,1),pt1(:,2),pt2(:,1),pt2(:,2))`.

`[arclen,az] = distance(pt1,pt2,ellipsoid)`,

`[arclen,az] = distance(pt1,pt,units)`,

`[arclen,az] = distance(pt1,pt2,ellipsoid,units)`, and

`[arclen,az] = distance(track,pt1,...)` are all valid calling forms.

## Examples

Using `pt1,pt2` notation, find the distance from Norfolk, Virginia (37ºN, 76ºW), to Cape St. Vincent, Portugal (37ºN, 9ºW), just outside the Straits of Gibraltar. The distance between these two points depends upon the *track* string selected.

```
arclen = distance('gc',[37,-76],[37,-9])

arclen =
   52.3094

arclen = distance('rh',[37,-76],[37,-9])

arclen =
```

```
      53.5086
```

The difference between these two tracks is 1.1992 degrees, or about 72 nautical miles. This represents about 2% of the total trip distance. The trade-off is that at the cost of those 72 miles, the entire trip can be made on a rhumb line with a fixed course of $90^{\circ}$, due east, while in order to follow the shorter great circle path, the course must be changed continuously.

On a meridian and on the Equator, great circles and rhumb lines coincide, so the distances are the same. For example,

```
% Great circle distance
arclen = distance(37,-76,67,-76)

arclen =
     30.0000

% Rhumb line distance
arclen = distance('rh',37,-76,67,-76)

arclen =
     30.0000
```

The distances are the same, $30^{\circ}$, or about 1800 nautical miles. (There are about 60 nautical miles in a degree of arc length.)

## Alternatives

Distance between two points can be calculated in two ways. For great circles (on the sphere) and geodesics (on the ellipsoid), the distance is the shortest surface distance between two points. For rhumb lines, the distance is measured along the rhumb line passing through the two points, which is not, in general, the shortest surface distance between them.

When you need to compute both distance and azimuth for the same point pair(s), it is more efficient to do so with a single call to distance. That is, use

```
[arclen az] = distance(...);
```
rather than the slower

```
arclen = distance(...)
az = azimuth(...)
```

**1-207**

To express the output `arclen` as an arc length in either degrees or radians, omit the `ellipsoid` argument. This is possible only on a sphere. If `ellipsoid` is supplied, `arclen` is a distance expressed in the same units as the semimajor axis of the ellipsoid. Specify `ellipsoid` as `[R 0]` to compute `arclen` as a distance on a sphere of radius `R`, with `arclen` having the same units as `R`.

# More About

### Algorithms

Distance calculations for geodesics degrade slowly with increasing distance and may break down for points that are nearly antipodal, as well as when both points are very close to the Equator. In addition, for calculations on an ellipsoid, there is a small but finite input space, consisting of pairs of locations in which both the points are nearly antipodal *and* both points fall close to (but not precisely on) the Equator. In this case, a warning is issued and both `arclen` and `az` are set to NaN for the "problem pairs."

- "Great Circles"
- "Rhumb Lines"
- "Small Circles"

# See Also

referenceEllipsoid | referenceSphere | oblateSpheroid | `azimuth` | `elevation` | `reckon` | `track` | `track1` | `track2` | `trackg`

# distortcalc

Distortion parameters for map projections

## Syntax

```
areascale = distortcalc(lat,long)
areascale = distortcalc(mstruct,lat,long)
[areascale,angdef,maxscale,minscale,merscale,parscale] =
distortcalc(...)
```

## Description

`areascale = distortcalc(lat,long)` computes the area distortion for the current map projection at the specified geographic location. An area scale of 1 indicates no scale distortion. Latitude and longitude can be scalars, vectors, or matrices in the angle units of the defined map projection.

`areascale = distortcalc(mstruct,lat,long)` uses the projection defined in the map structure `mstruct`.

`[areascale,angdef,maxscale,minscale,merscale,parscale] = distortcalc(...)` computes the area scale, maximum angular deformation of right angles (in the angle units of the defined projection), the particular maximum and minimum scale distortions in any direction, and the particular scale along the meridian and parallel. You can also call `distortcalc` with fewer output arguments, in the order shown.

## Background

Map projections inevitably introduce distortions in the shapes and sizes of objects as they are transformed from three-dimensional spherical coordinates to two-dimensional Cartesian coordinates. The amount and type of distortion vary between projections, over the projection, and with the selection of projection parameters such as standard parallels. This function allows a quantitative evaluation of distortion parameters.

# Examples

At the equator, the Mercator projection is free of both area and angular distortion:

```
axesm mercator
[areascale,angdef] = distortcalc(0,0)

areascale =
      1.0000
angdef =
      8.5377e-007
```

At 60 degrees north, objects are shown at 400% of their true area. The projection is conformal, so angular distortion is still zero.

```
[areascale,angdef] = distortcalc(60,0)

areascale =
      4.0000
angdef =
      4.9720e-004
```

# More About

### Tips

This function uses a finite difference technique. The geographic coordinates are perturbed slightly in different directions and projected. A small amount of error is introduced by numerical computation of derivatives and the variation of map distortion parameters.

## See Also

```
mdistort | tissot
```

# distdim

Convert length units

## Syntax

```
distOut = distdim(distIn,from,to)
distOut = distdim(distIn,from,to,radius)
distOut = distdim(distIn,from,to,sphere)
```

---

**Note:** `distdim` has been replaced by `unitsratio`, but will be maintained for backward compatibility. See "Replacing distdim" on page 1-214 for details.

---

## Description

`distOut = distdim(distIn,from,to)` converts `distIn` from the units specified by the string *from* to the units specified by the string *to*. *from* and *to* are case-insensitive, and may equal any of the following:

| | |
|---|---|
| `'meters'` or `'m'` | |
| `'feet'` or `'ft'` | U.S. survey feet |
| `'kilometers'` or `'km'` | |
| `'nauticalmiles'` or `'nm'` | |
| `'miles'`, `'statutemiles'`, `'mi'`, or `'sm'` | Statute miles |
| `'degrees'` or `'deg'` | |
| `'radians'` or `'rad'` | |

If either *from* or *to* indicates angular units (`'degrees'` or `'radians'`), the conversion to or from linear distance is made along a great circle arc on a sphere with a radius of 6371 km, the mean radius of the Earth.

distOut = distdim(distIn,*from*,*to*,radius), where one of the unit strings, either *from* or *to*, indicates angular units and the other unit string indicates length units, uses a great circle arc on a sphere of the given radius. The specified length units must apply to radius as well as to the input distance (when *from* indicates length) or output distance (when *to* indicates length). If neither *from* nor *to* indicates angular units, or if both do, then the value of radius is ignored.

distOut = distdim(distIn,*from*,*to*,*sphere*), where either *from* or *to* indicates angular units, uses a great circle arc on a sphere approximating a body in the Solar System. *sphere* may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive. If neither *to* nor *from* is angular, *sphere* is ignored.

## Examples

Convert 100 kilometers to nautical miles:

```
distkm = 100

distkm =
    100

distnm = distdim(distkm,'kilometers','nauticalmiles')

distnm =
    53.9957
```

A degree of arc length is about 60 nautical miles:

```
distnm = distdim(1,'deg','nm')

distnm =
    60.0405
```

This is not accidental. It is the original definition of the nautical mile. Naturally, this assumption does not hold on other planets:

```
distnm = distdim(1,'deg','nm','mars')

distnm =
    31.9474
```

# More About

## Tips

## Arc Lengths of Angles Not Constant

Distance is expressed in one of two general forms: as a linear measure in some unit (kilometers, miles, etc.) or as angular arc length (degrees or radians). While the use of linear units is generally understood, angular arc length is not always as clear. The conversion from angular units to linear units for the arc along any circle is the angle in radians multiplied by the radius of the circle. On the sphere, this means that radians of latitude are directly translatable to kilometers, say, by multiplying by the radius of the Earth in kilometers (about 6,371 km). However, the linear distance associated with radians of longitude changes with latitude; the radius in question is then not the radius of the Earth, but the (chord) radius of the small circle defining that parallel. The angle in radians or degrees associated with any distance is the arc length of a great circle passing through the points of interest. Therefore, the radius in question always refers to the radius of the relevant sphere, consistent with the `distance` function.

## Exercise Caution with 'feet' and 'miles'

*Exercise caution with 'feet' and 'miles'.* `distdim` interprets `'feet'` and `'ft'` as U.S. survey feet, and does not support international feet at all. In contrast, `unitsratio` follows the opposite, and more standard approach, interpreting both `'feet'` and `'ft'` as international feet. `unitsratio` provides separate options, including `'surveyfeet'` and `'sf'`, to indicate survey feet. By definition, one international foot is exactly 0.3048 meters and one U.S. survey foot is exactly 1200/3937 meters. For many applications, the difference is significant. Most projected coordinate systems use either the meter or the survey foot as a standard unit. International feet are less likely to be used, but do occur sometimes. Likewise, `distdim` interprets `'miles'` and `'mi'` as statute miles (also known as U.S. survey miles), and does not support international miles at all. By definition, one international mile is 5,280 international feet and one statute mile is 5,280 survey feet. You can evaluate:

```
unitsratio('millimeter','statute mile') - ...
    unitsratio('millimeter','mile')
```

to see that the difference between a statute mile and an international mile is just over three millimeters. This may seem like a very small amount over the length of a single mile, but mixing up these units could result in a significant error over a sufficiently

long baseline. Originally, the behavior of `distdim` with respect to `'miles'` and `'mi'` was documented only indirectly, via the now-obsolete `unitstr` function. As with feet, `unitsratio` takes a more standard approach. `unitsratio` interprets `'miles'` and `'mi'` as international miles, and `'statute miles'` and `'sm'` as statute miles. (`unitsratio` accepts several other strings for each of these units; see the `unitsratio` help for further information.)

## Replacing distdim

If both *from* and *to* are known at the time of coding, then you may be able to replace `distdim` with a direct conversion utility, as in the following examples:

| | |
|---|---|
| `distdim(dist,'nm',km')` | ⇒`nm2km(dist)` |
| `distdim(dist,'sm','deg')` | ⇒ `sm2deg(dist)` |
| `distdim(dist, 'rad', 'km', 'moon')` | ⇒`rad2km(dist,'moon')` |

If there is no appropriate direct conversion utility, or you won't know the values of *from* and/or *to* until run time, you can generally replace

```
distdim(dist, FROM, TO)
```
with

```
unitsratio(TO, FROM) * dist
```
If you are using units of feet or miles, see the cautionary note above about how they are interpreted. For example, with `distIn` in meters and `distOut` in survey feet, `distOut = distdim(distIn, 'meters', 'feet')` should be replaced with `distOut = unitsratio('survey feet','meters') * distIn`. Saving a multiplicative factor computed with `unitsratio` and using it to convert in a separate step can make code cleaner and more efficient than using `distdim`. For example, replace

```
dist1_meters = distdim(dist1_nm, 'nm', 'meters');
dist2_meters = distdim(dist2_nm, 'nm', 'meters');
```
with

```
metersPerNM = unitsratio('meters','nm');
dist1_meters = metersPerNM * dist1_nm;
dist2_meters = metersPerNM * dist2_nm;
```
`unitsratio` does not perform great-circle conversion between units of length and angle, but it can be easily combined with other functions to do so. For example, to convert

degrees to meters along a great-circle arc on a sphere approximating the planet Mars, you could replace

```
distdim(dist, 'degrees', 'meters', 'mars')
```
with

```
unitsratio('meters','km') * deg2km(dist, 'mars')
```

## See Also

deg2km | deg2nm | deg2sm | km2deg | km2nm | km2rad | km2sm | nm2deg | nm2km | nm2rad | nm2sm | rad2km | rad2nm | rad2sm | sm2deg | sm2km | sm2nm | sm2rad | unitsratio

# dm2degrees

Convert degrees-minutes to degrees

## Syntax

```
angleInDegrees = dm2degrees(DM)
```

## Description

`angleInDegrees = dm2degrees(DM)` converts angles from degree-minutes representation to values in degrees which may include a fractional part (sometimes called "decimal degrees"). DM should be N-by-2 and real-valued, with one row per angle. The output will be an N-by-1 column vector whose $k^{th}$ element corresponds to the $k^{th}$ row of DM. The first column of DM contains the "degrees" element and should be integer-valued. The second column contains the "minutes" element and may have a fractional part For an angle that is positive (north latitude or east longitude) or equal to zero, all elements in the row need to be nonnegative. For a negative angle (south latitude or west longitude), the first nonzero element in the row should be negative and the remaining value, if any, should be nonzero. Thus, for an input row with value `[D M]`, with integer-valued D and real M, the output value will be

```
SGN * (abs(D) + abs(M)/60)
```
where SGN is 1 if D and M are both nonnegative and -1 if the first nonzero element of `[D M]` is negative (an error results if a nonzero D is followed by a negative M). Any fractional parts in the first (degrees) columns of DM are ignored. An error results unless the absolute values of all elements in the second (minutes) column are less than 60.

## Examples

```
dm = [ ...
      30  44.78012; ...
     -82  39.90825; ...
       0 -17.12345; ...
       0  14.82000];
format long g
```

```
angleInDegrees = dm2degrees(dm)

angleInDegrees =
          30.7463353333333
              -82.6651375
        -0.285390833333333
                    0.247
```

## See Also
degrees2dm | deg2rad | dms2degrees | str2angle

# dms2degrees

Convert degrees-minutes-seconds to degrees

## Syntax

```
angleInDegrees = dms2degrees(DMS)
```

## Description

`angleInDegrees = dms2degrees(DMS)` converts angles from degree-minutes-seconds representation to values in degrees which may include a fractional part (sometimes called "decimal degrees"). `DMS` should be N-by-3 and real-valued, with one row per angle. The output will be an N-by-1 column vector whose $k^{th}$ element corresponds to the $k^{th}$ row of `DMS`. The first column of `DMS` contains the "degrees" element and should be integer-valued. The second column contains the "minutes" element and should be integer-valued. The third column contains the "seconds" element and may have a fractional part. For an angle that is positive (north latitude or east longitude) or equal to zero, all elements in the row need to be nonnegative. For a negative angle (south latitude or west longitude), the first nonzero element in the row should be negative and the remaining values should be positive. Thus, for an input row with value `[D M S]`, with integer-valued `D` and `M`, and real `D`, `M`, and `S`, the output value will be

```
SGN * (abs(D) + abs(M)/60 + abs(S)/3600)
```
where `SGN` is 1 if `D`, `M`, and `S` are all nonnegative and -1 if the first nonzero element of `[D M S]` is negative (an error results if a nonzero element is followed by a negative element). Any fractional parts in the first (degrees) and second (minutes) columns of `DMS` are ignored. An error results unless the absolute values of all elements in the second (minutes) and third (seconds) columns are less than 60.

## Examples

```
dms = [ ...
          30   50 44.78012; ...
         -82    2 39.90825; ...
           0  -30 17.12345; ...
```

```
              0    0 14.82000];
format long g
angleInDegrees = dms2degrees(dms)

angleInDegrees =
          30.8457722555556
         -82.0444189583333
         -0.504756513888889
        0.00411666666666667
```

## See Also

degrees2dm | deg2rad | dm2degrees | str2angle

# dreckon

Dead reckoning positions for track

## Syntax

```
[drlat,drlong,drtime] = dreckon(waypoints,time,speed)
[drlat,drlong,drtime] = dreckon (waypoints,time,speed,spdtimes)
```

## Description

`[drlat,drlong,drtime] = dreckon(waypoints,time,speed)` returns the positions and times of required dead reckoning (DR) points for the input track that starts at the input time. The track should be in navigational track format (two columns, latitude then longitude, in order of traversal). These waypoints are the starting and ending points of each leg of the track. There is one fewer track leg than waypoints, as the last point included is the end of the track. In navigation, the first waypoint would be a navigational fix, taken at `time`. The `speed` input can be a scalar, in which case a constant speed is used throughout, or it can be a vector in which one speed is given for each track leg (that is, speed changes coincide with course changes).

`[drlat,drlong,drtime] = dreckon (waypoints,time,speed,spdtimes)` allows speed changes to occur independent of course changes. The elements of the `speed` vector must have a one-to-one correspondence with the elements of the `spdtimes` vector. This latter variable consists of the time interval after `time` at which each speed order *ends*. For example, if `time` is 6.75, and the first element of `spdtimes` is 1.35, then the first `speed` element is in effect from 6.75 to 8.1 hours. When this syntax is used, the last output DR is the *earlier* of the final `spdtimes` time or the final `waypoints` point.

## Background

This is a navigational function. It assumes that all latitudes and longitudes are in degrees, all distances are in nautical miles, all times are in hours, and all speeds are in knots, that is, nautical miles per hour.

Dead reckoning is an estimation of position at various times based on courses, speeds, and times elapsed from the last certain position, or fix. In navigational practice, a dead

reckoning position, or DR, must be plotted at every course change, every speed change, and at every hour, on the hour. Navigators also DR at other times that are not relevant to this function.

Often in practice, when two events occur that require DRs within a very short time, only one DR is generated. This function mimics that practice by setting a tolerance of 3 minutes (0.05 hours). No two DRs will fall closer than that.

Refer to "Navigation" in the *Mapping Toolbox Guide* for further information.

## Examples

Assume that a navigator gets a fix at noon, 1200Z, which is (10.3ºN, 34.67ºW). He's in a hurry to make a 1330Z rendezvous with another ship at (9.9ºN, 34.5ºW), so he plans on a speed of 25 knots. After the rendezvous, both ships head for (0º, 37ºW). The engineer wants to take an engine off line for maintenance at 1430Z, so at that time, speed must be reduced to 15 knots. At 1530Z, the maintenance will be done. Determine the DR points up to the end of the maintenance.

```
waypoints = [10.1 -34.6; 9.9 -34.5; 0 -37]

waypoints =
   10.1000  -34.6000        % Fix at noon
    9.9000  -34.5000        % Rendezvous point
         0  -37.0000        % Ultimate destination

speed = [25; 15];
spdtimes = [2.5; 3.5];       % Elapsed times after fix
noon = 12;
[drlat,drlong,drtime] = dreckon(waypoints,noon,speed,spdtimes);
[drlat,drlong,drtime]

ans =
    9.8999  -34.4999   12.5354        % Course change at waypoint
    9.7121  -34.5478   13.0000        % On the hour
    9.3080  -34.6508   14.0000        % On the hour
    9.1060  -34.7022   14.5000        % Speed change to 15 kts
    8.9847  -34.7330   15.0000        % On the hour
    8.8635  -34.7639   15.5000        % Stop at final spdtime, last
            % waypoint has not been reached
```

## See Also

legs | navfix | track

# driftcorr

Heading to correct for wind or current drift

## Syntax

```
heading = driftcorr(course,airspeed,windfrom,windspeed)
[heading,groundspeed,windcorrangle] = driftcorr(...)
```

## Description

`heading = driftcorr(course,airspeed,windfrom,windspeed)` computes the heading that corrects for drift due to wind (for aircraft) or current (for watercraft). `course` is the desired direction of movement (in degrees), `airspeed` is the speed of the vehicle relative to the moving air or water mass, `windfrom` is the direction facing into the wind or current (in degrees), and `windspeed` is the speed of the wind or current (in the same units as `airspeed`).

`[heading,groundspeed,windcorrangle] = driftcorr(...)` also returns the ground speed and wind correction angle. The wind correction angle is positive to the right, and negative to the left.

## Examples

An aircraft cruising at a speed of 160 knots plans to fly to an airport due north of its current position. If the wind is blowing from 310 degrees at 45 knots, what heading should the aircraft fly to remain on course?

```
course=0; airspeed=160;windfrom=310; windspeed = 45;
[heading,groundspeed,windcorrangle] =
driftcorr(course,airspeed,windfrom,windspeed)

heading =

      347.56

groundspeed =
```

```
    127.32
```

```
windcorrangle =
```

```
    -12.442
```

The required heading is 348 degrees, which amounts to a wind correction angle of 12 degrees to the left of course. The headwind component reduces the aircraft's ground speed to 127 knots.

## See Also

```
driftvel
```

# driftvel

Wind or current from heading, course, and speeds

## Syntax

```
[windfrom,windspeed] = driftvel
(course,groundspeed,heading,airspeed)
```

## Description

`[windfrom,windspeed] = driftvel
(course,groundspeed,heading,airspeed)` computes the wind (for aircraft) or
current (for watercraft) from course, heading, and speeds. `course` and `groundspeed`
are the direction and speed of movement relative to the ground (in degrees), `heading` is
the direction in which the vehicle is steered, and `airspeed` is the speed of the vehicle
relative to the air mass or water. The output `windfrom` is the direction facing into the
wind or current (in degrees), and `windspeed` is the speed of the wind or current (in the
same units as `airspeed` and `groundspeed`).

## Examples

An aircraft is cruising at a true air speed of 160 knots and a heading of 10 degrees. From
the Global Positioning System (GPS) receiver, the pilot determines that the aircraft is
progressing over the ground at 155 knots in a northerly direction. What is the wind aloft?

```
course = 0; groundspeed = 155; heading = 10; airspeed = 160;
[windfrom,windspeed] =
driftvel(course,groundspeed,heading,airspeed)

windfrom =
      84.717

windspeed =
      27.902
```

The wind is blowing from the right, 085 degrees at 28 knots.

## See Also

driftcorr

# dted

Read U.S. Department of Defense Digital Terrain Elevation Data (DTED)

## Syntax

```
[Z, refvec] = dted
[Z, refvec] = dted(filename)
[Z, refvec] = dted(filename, samplefactor)
[Z, refvec] = dted(filename, samplefactor, latlim, lonlim)
[Z, refvec] = dted(foldername, samplefactor, latlim, lonlim)
[Z, refvec, UHL, DSI, ACC] = dted(...)
```

## Description

`[Z, refvec] = dted` returns all of the elevation data in a DTED file as a regular data grid, `Z`, with elevations in meters. The file is selected interactively. This function reads the DTED elevation files, which generally have filenames ending in `.dtN`, where `N` is 0,1,2,3,... `refvec` is the associated three-element referencing vector that geolocates `Z`.

`[Z, refvec] = dted(filename)` returns all of the elevation data in the specified DTED file. The file must be found on the MATLAB path. If not found, the file may be selected interactively.

`[Z, refvec] = dted(filename, samplefactor)` subsamples data from the specified DTED file. `samplefactor` is a scalar integer. When `samplefactor` is 1 (the default), DTED reads the data at its full resolution. When `samplefactor` is an integer `n` greater than one, every `n`th point is read.

`[Z, refvec] = dted(filename, samplefactor, latlim, lonlim)` reads the data for the part of the DTED file within the latitude and longitude limits. The limits must be two-element vectors in units of degrees.

`[Z, refvec] = dted(foldername, samplefactor, latlim, lonlim)` reads and concatenates data from multiple files within a DTED CD-ROM or folder structure. The `foldername` input is a string with the name of a folder containing the DTED folder. Within the DTED folder are subfolders for each degree of longitude, each of which

contain files for each degree of latitude. For DTED CD-ROMs, `foldername` is the device name of the CD-ROM drive.

`[Z, refvec, UHL, DSI, ACC] = dted(...)` returns structures containing the DTED User Header Label (UHL), Data Set Identification (DSI) and Accuracy metadata records.

# Background

The U. S. Department of Defense, through the National Geospatial Intelligence Agency, produces several kinds of digital cartographic data. One is digital elevation data, in a series called DTED, for Defense Digital Terrain Elevation Data. The data is available as 1-by-1 degree quadrangles at horizontal resolutions ranging from about 1 kilometer to 1 meter. The lowest resolution data is available to the public. Certain higher resolution data is restricted to the U.S. Department of Defense and its contractors.

DTED Level 0 files have 121-by-121 points. DTED Level 1 files have 1201-by-1201. The edges of adjacent tiles have redundant records. Maps extend a half a cell outside the requested map limits. The 1 kilometer data and some higher-resolution data is available online, as are product specifications and documentation. DTED files are binary. No line ending conversion or byte-swapping is required when downloading a DTED file.

# Examples

### Read Digital Terrain Elevation Data (DTED)

This example shows how to read Digital Terrain Elevation Data (DTED). To run this example, you must have a DTED file. For details on locating DTED for download over the Internet, see "Finding Geospatial Data".

Read the DTED file.

```
[Z,refvec] = dted('n38.dt0');
```

### Read Digital Terrain Elevation Data (DTED) Specifying Sample Factor and Latitude and Longitude

This example shows how to read Digital Terrain Elevation Data (DTED), specifying the sample factor and latitude and longitude. To run this example, you must have a DTED file. For details on locating DTED for download over the Internet, see "Finding Geospatial Data".

Read the DTED file, specifying sample factor along with latitude and longitude. The example returns structures containing the DTED User Header Label (UHL), Data Set Identification (DSI) and Accuracy metadata records.

```
[Z,refvec,UHL,DSI,ACC] = dted('n38.dt0',1,[38.5 38.8],...
  [-76.8 -76.6]);
```

**Read Digital Terrain Elevation Data (DTED) Specifying Folder Name**

This example shows how to read Digital Terrain Elevation Data (DTED) specifying a folder name. To run this example, you must have DTED files. For details on locating DTED for download over the Internet, see "Finding Geospatial Data".

Read the DTED files in the folder specified. The example returns structures containing the DTED User Header Label (UHL), Data Set Identification (DSI) and Accuracy metadata records.

```
[Z,refvec,UHL,DSI,ACC] = dted('f:',1,[38.5 38.8],...
  [-76.8 -76.6]);
```

# More About

**Tips**

## Latitude-Dependent Sampling

In DTED files north of 50° North and south of 50° South, where the meridians have converged significantly relative to the equator, the longitude sampling interval is reduced to half of the latitude sampling interval. In order to retain square output cells, this function reduces the latitude sampling to match the longitude sampling. For example, it will return a 121-by-121 elevation grid for a DTED file covering from 49 to 50 degrees north, but a 61-by-61 grid for a file covering from 50 to 51 degrees north. When you supply a folder name instead of a file name, and `latlim` spans either 50° North or 50° South, an error results.

## Snapping Latitude and Longitude Limits

If you call `dted` specifying arbitrary latitude-longitude limits for a region of interest, the grid and referencing vector returned will not exactly honor the limits you specified unless

they fall precisely on grid cell boundaries. Because grid cells are discrete and cannot be arbitrarily divided, the data grid returned will include all areas between your latitude-longitude limits and the next row or column of cells, potentially in all four directions.

## Data Sources and Information

DTED files contain digital elevation maps covering 1-by-1-degree quadrangles at horizontal resolutions ranging from about 1 kilometer to 1 meter. For details on locating DTED for download over the Internet, see "Finding Geospatial Data".

## Null Data Values

Some DTED Level 1 and higher data tiles contain null data cells, coded with value -32767. When encountered, these null data values are converted to NaN.

## Nonconforming Data Encoding

DTED files from some sources may depart from the specification by using two's complement encoding for binary elevation files instead of "sign-bit" encoding. This difference affects the decoding of negative values, and incorrect decoding usually leads to nonsensical elevations.

Thus, if the DTED function determines that all the (nonnull) negative values in a file would otherwise be less than -12,000 meters, it issues a warning and assumes two's complement encoding.

## See Also
usgsdem | gtopo30 | tbase | etopo

# dteds

DTED filenames for latitude-longitude quadrangle

## Syntax

```
fname = dteds(latlim,lonlim)
fname = dteds(latlim,lonlim,level)
```

## Description

`fname = dteds(latlim,lonlim)` returns Level 0 DTED file names (folder and name) required to cover the geographic region specified by `latlim` and `lonlim`.

`fname = dteds(latlim,lonlim,level)` controls the level for which the file names are generated. Valid inputs for the `level` of the DTED files include 0, 1, or 2.

## Background

The U. S. Department of Defense produces several kinds of digital cartographic data. One is digital elevation data, in a series called DTED, for Defense Digital Terrain Elevation Data. The data is available as 1-by-1 degree quadrangles at horizontal resolutions ranging from about 1 kilometer to 1 meter. The lowest resolution data is available to the public. Higher resolution data is restricted to the U.S. Department of Defense and its contractors.

Determining the files needed to cover a particular region requires knowledge of the DTED database naming conventions. This function constructs the file names for a given geographic region based on these conventions.

## Examples

Which files are needed for Cape Cod?

```
latlim = [ 41.15 42.22]; lonlim = [-70.94 -69.68];
```

```
dteds(latlim,lonlim,1)

ans =
    '\DTED\W071\N41.dt1'
    '\DTED\W070\N41.dt1'
    '\DTED\W071\N42.dt1'
    '\DTED\W070\N42.dt1'
```

## See Also
dted

# earthRadius

Mean radius of planet Earth

## Syntax

```
R = earthRadius
R = earthRadius(lengthUnit)
```

## Description

`R = earthRadius` returns the scalar value 6371000, the mean radius of the Earth in meters.

`R = earthRadius(lengthUnit)` returns the mean radius of the Earth using the specified unit of length. The `lengthUnit` input may be any string accepted by the `validateLengthUnit` function.

## Examples

```
earthRadius              % Returns 6371000

earthRadius('meters')    % Returns 6371000

earthRadius('km')        % Returns 6371
```

## See Also

`unitsratio` | `validateLengthUnit`

# eastof

Wrap longitudes to values east of specified meridian

---

**Note:** The `eastof` function is obsolete and will be removed in a future release of Mapping Toolbox software. Replace it with the following calls, which are also more efficient:

```
eastof(lon,meridian,'degrees') ==> meridian+mod(lon-meridian,360)
```

```
eastof(lon,meridian,'radians') ==> meridian+mod(lon-meridian,2*pi)
```

---

## Syntax

```
lonWrapped = eastof(lon,meridian)
lonWrapped = eastof(lon,meridian,angleunits)
```

## Description

`lonWrapped = eastof(lon,meridian)` wraps angles in `lon` to values in the interval `[meridian meridian+360)`. `lon` is a scalar longitude or vector of longitude values. All inputs and outputs are in degrees.

`lonWrapped = eastof(lon,meridian,angleunits)` specifies the input and output units with the string *angleunits*. *angleunits* can be either `'degrees'` or `'radians'`. It may be abbreviated and is case-insensitive. If *angleunits* is `'radians'`, the input is wrapped to the interval `[meridian meridian+2*pi)`.

# ecc2flat

Flattening of ellipse from eccentricity

## Syntax

```
f = ecc2flat(ecc)
f = ecc2flat(ellipsoid)
```

## Description

`f = ecc2flat(ecc)` computes the flattening of an ellipse (or ellipsoid of revolution) given its eccentricity `ecc`. Except when the input has 2 columns (or is a row vector), each element is assumed to be an eccentricity and the output `f` has the same size as `ecc`.

`f = ecc2flat(ellipsoid)`, where `ellipsoid` has two columns (or is a row vector), assumes that the eccentricity is in the second column, and a column vector is returned.

## See Also
`ecc2n` | `flat2ecc` | `majaxis` | `minaxis`

# ecc2n

Third flattening of ellipse from eccentricity

## Syntax

```
n = ecc2n(ecc)
n = ecc2n(ellipsoid)
```

## Description

n = ecc2n(ecc) computes the parameter *n* (the "third flattening") of an ellipse (or ellipsoid of revolution) given its eccentricity ecc. *n* is defined as $(a\text{-}b)/(a\text{+}b)$, where *a* is the semimajor axis and *b* is the semiminor axis. Except when the input has 2 columns (or is a row vector), each element is assumed to be an eccentricity and the output *n* has the same size as ecc.

n = ecc2n(ellipsoid), where ellipsoid has two columns (or is a row vector), assumes that the eccentricity is in the second column, and a column vector is returned.

## See Also
ecc2flat | majaxis | minaxis | n2ecc

# ecef2aer

Geocentric ECEF to local spherical AER

## Syntax

```
[az,elev,slantRange] = ecef2aer(X,Y,Z,lat0,lon0,h0,spheroid)
[ ___ ] = ecef2aer( ___ ,angleUnit)
```

## Description

`[az,elev,slantRange] = ecef2aer(X,Y,Z,lat0,lon0,h0,spheroid)` returns coordinates in a local spherical system corresponding to coordinates X, Y, Z in an Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian system. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ecef2aer( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0`, `lon0`, and outputs `az`, `elev`.

## Input Arguments

### X — ECEF x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### Y — ECEF y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### Z — ECEF y-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### h0 — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### `az` — Azimuth angles
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the half-open interval [0 360).

### `elev` — Elevation angles
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface normal. Units determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the closed interval [-90 90].

### `slantRange` — Distances from local origin
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is computed. Units are determined by the `LengthUnit` property of the `spheroid` input.

## See Also
`aer2ecef` | `ecef2enu` | `ecef2ned` | `geodetic2aer`

# ecef2enu

Geocentric ECEF to local Cartesian ENU

## Syntax

```
[xEast,yNorth,zUp] = ecef2enu(X,Y,Z,lat0,lon0,h0,spheroid)
[ ___ ] = ecef2enu( ___ ,angleUnit)
```

## Description

`[xEast,yNorth,zUp] = ecef2enu(X,Y,Z,lat0,lon0,h0,spheroid)` returns coordinates in a local east-north-up (ENU) Cartesian system corresponding to coordinates `X`, `Y`, `Z` in an Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian system. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ecef2enu( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs, `lat0`, and `lon0`.

## Input Arguments

### X — ECEF x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### Y — ECEF y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### `Z` — ECEF y-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

**`angleUnit` — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

# Output Arguments

**`xEast` — Local ENU x-coordinates**
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

**`yNorth` — Local ENU y-coordinates**
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

**`zUp` — Local ENU z-coordinates**
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

# See Also
`ecef2aer` | `ecef2ned` | `enu2ecef` | `geodetic2enu`

# ecef2enuv

Rotate vector from geocentric ECEF to local ENU

## Syntax

```
[uEast,vNorth,wUp] = ecef2enuv(U,V,W,lat0,lon0)
[ ___ ] = ecef2enuv( ___ ,angleUnit)
```

## Description

`[uEast,vNorth,wUp] = ecef2enuv(U,V,W,lat0,lon0)` returns Cartesian 3-vector components in a local east-north-up (ENU) system corresponding to the 3-vector with components `U`, `V`, `W` in an Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian system. Any of the five numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ecef2enuv( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0` and `lon0`.

## Input Arguments

### U — Vector x-components in ECEF system
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### V — Vector y-components in ECEF system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### `W` — Vector z-components in ECEF system
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### `uEast` — Vector x-components in ENU system
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the local ENU system, returned as a scalar value, vector, matrix, or N-D array.

### vNorth — Vector y-components in ENU system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the local ENU system, returned as a scalar value, vector, matrix, or N-D array.

### wUp — Vector z-components in ENU system
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the local ENU system, returned as a scalar value, vector, matrix, or N-D array.

## See Also
`ecef2enu` | `ecef2nedv` | `enu2ecefv`

# ecef2geodetic

Transform geocentric (ECEF) to geodetic coordinates

## Syntax

```
[lat,lon,h] = ecef2geodetic(spheroid,x,y,z)
[lat,lon,h] = ecef2geodetic( ___ ,angleUnit)

[lat,lon,h] = geodetic2ecef(x,y,z,spheroid)
```

## Description

`[lat,lon,h] = ecef2geodetic(spheroid,x,y,z)` transforms point locations in geocentric Earth-Centered Earth-Fixed (ECEF) Cartesian coordinates, stored in the coordinate arrays X, Y, and Z, to geodetic coordinates `lat` (geodetic latitude), `lon` (longitude), and `h` (ellipsoidal height). The geodetic coordinates refer to the reference body specified by the spheroid object, `spheroid`.

`[lat,lon,h] = ecef2geodetic( ___ ,angleUnit)` uses the string `angleUnit`, which matches either `'degrees'` or `'radians'`, to specify the units of the latitude and longitude coordinates.

`[lat,lon,h] = geodetic2ecef(x,y,z,spheroid)` This syntax is supported but not recommended. This syntax returns lat and lon in radians. spheroid can be either a spheroid or an ellipsoid vector of the form: `[semimajor_axis, eccentricity]`. X, Y, and `Z` must use the same units as the semimajor axis. In this case, `H` must use the same units as the semimajor axis.

## Examples

### Convert Geocentric Coordinates to Geodetic Coordinates

Convert the geocentric Earth-Centered Earth-Fixed (ECEF) Cartesian coordinates for Paris, France into geodetic coordinates (latitude,longitude).

```
x =   4.2010e+06;
```

```
y =   1.7246e+05;
z =   4.7801e+06;
wgs84 = wgs84Ellipsoid('meters');
[lat,lon,h] = ecef2geodetic(wgs84,x,y,z)


lat =

   48.8562


lon =

    2.3508


h =

   67.3700
```

## Input Arguments

### spheroid — Reference body
spheroid object

Reference body, specified as a spheroid object: oblateSpheroid,
referenceEllipsoid, or referenceSphere.

### x — Geocentric Earth-Centered Earth-Fixed Cartesian X coordinates
numeric array

Geocentric Earth-Centered Earth-Fixed Cartesian X coordinates, specified as a numeric
array.

Data Types: single | double

### y — Geocentric Earth-Centered Earth-Fixed Cartesian Y coordinates
numeric array

Geocentric Earth-Centered Earth-Fixed Cartesian Ycoordinates, returned as a numeric
array.

Data Types: `single` | `double`

### z — Geocentric Earth-Centered Earth-Fixed Cartesian Z coordinates
scalar

Geocentric Earth-Centered Earth-Fixed Cartesian Z coordinates, specified as a numeric array.

Example:

Data Types: `single` | `double`

### angleUnit — Units of the latitude and longitude coordinates
`'degrees'` | `'radians'`

Units of the latitude and longitude coordinates, specified as the character string `'degrees'` or `'radians'`.

Data Types: `char`

## Output Arguments

### lat — Geodetic latitude in degrees
geodetic coordinate array

Geodetic latitude in degrees, returned as geodetic coordinate array.

### lon — Geodetic longitude in degrees
geodetic coordinate array

Geodetic longitude in degrees, returned as a geodetic coordinate array.

### h — Ellipsoidal height
numeric array

Ellipsoidal height, returned as a numeric array. `h` is expressed in the same unit of length as the `spheroid`.

## More About

### Tips

- The geocentric Cartesian (ECEF) coordinate system is fixed with respect to the Earth, with its origin at the center of the spheroid and its positive X-, Y-, and Z axes intersecting the surface at the following points:

| | Latitude | Longitude | Notes |
|---|---|---|---|
| X-axis | 0 | 0 | Equator at the Prime Meridian |
| Y-axis | 0 | 90 | Equator at 90-degrees East |
| Z-zxis | 90 | 0 | North Pole |

### See Also

oblateSpheroid | referenceEllipsoid | referenceSphere | ecef2geodetic | ecefOffset | geodetic2ecef

**Introduced in R2012b**

# ecef2lv

Convert geocentric (ECEF) to local vertical coordinates

---

**Note:** `ecef2lv` will be removed in a future release. Use `ecef2enu` instead. In `ecef2enu`, the latitude and longitude of the local origin are in degrees by default, so the optional `angleUnit` input should be included, with the value `'radians'`.

---

## Syntax

`[xl,yl,zl] = ecef2lv(x,y,z,phi0,lambda0,h0,ellipsoid)`

## Description

`[xl,yl,zl] = ecef2lv(x,y,z,phi0,lambda0,h0,ellipsoid)` converts geocentric point locations specified by the coordinate arrays `x`, `y`, and `z` to the local vertical coordinate system, with its origin at geodetic latitude `phi0`, longitude `lambda0`, and ellipsoidal height `h0`. The arrays `x`, `y`, and `z` may be of any shape, as long as they all match in size. `phi0`, `lambda0`, and `h0` must be scalars. `ellipsoid` is a referenceEllipsoid (oblateSpheroid) object, a referenceSphere object, or a vector of the form [`semimajor axis, eccentricity`]). `x`, `y`, `z`, and `h0` must have the same length units as the semimajor axis. `phi0` and `lambda0` must be in radians. The output coordinate arrays, `xl`, `yl`, and `zl` are the local vertical coordinates of the input points. They have the same size as `x`, `y`, and `z` and have the same length units as the semimajor axis.

In the local vertical Cartesian system defined by `phi0`, `lambda0`, `h0`, and `ellipsoid`, the `xl` axis is parallel to the plane tangent to the ellipsoid at (`phi0`,`lambda0`) and points due east. The `yl` axis is parallel to the same plane and points due north. The `zl` axis is normal to the ellipsoid at (`phi0`,`lambda0`) and points outward into space. The local vertical system is sometimes referred to as east-north-up or ENU.

## Definitions

For a definition of the *geocentric system*, also known as Earth-Centered, Earth-Fixed (ECEF), see the help for `geodetic2ecef`.

## See Also

`ecef2enu`

# ecef2ned

Geocentric ECEF to local Cartesian NED

## Syntax

```
[xNorth,yEast,zDown] = ecef2ned(X,Y,Z,lat0,lon0,h0,spheroid)
[ ___ ] = ecef2ned( ___ ,angleUnit)
```

## Description

`[xNorth,yEast,zDown] = ecef2ned(X,Y,Z,lat0,lon0,h0,spheroid)` returns coordinates in a local north-east-down (NED) Cartesian system corresponding to coordinates `X`, `Y`, `Z` in an Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian system. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ecef2ned( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0`, and `lon0`.

## Input Arguments

### X — ECEF x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### Y — ECEF y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### `Z` — ECEF y-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### `xNorth` — Local NED x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

### `yEast` — Local NED y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

### `zDown` — Local NED z-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

## See Also
`ecef2aer` | `ecef2enu` | `geodetic2ned` | `ned2ecef`

# ecef2nedv

Rotate vector from geocentric ECEF to local NED

## Syntax

```
[uNorth,vEast,wDown] = ecef2nedv(U,V,W,lat0,lon0)
[ ___ ] = ecef2nedv( ___ ,angleUnit)
```

## Description

`[uNorth,vEast,wDown] = ecef2nedv(U,V,W,lat0,lon0)` returns Cartesian 3-vector components in a local north-east-down (NED) system corresponding to the 3-vector with components U, V, W in an Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian system. Any of the five numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ecef2nedv( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0` and `lon0`.

## Input Arguments

### U — Vector x-components in ECEF system
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### V — Vector y-components in ECEF system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**W — Vector z-components in ECEF system**
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the spheroid-centric ECEF system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**lat0 — Geodetic latitude of local origin**
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

**lon0 — Longitude of local origin**
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

**angleUnit — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

**uNorth — Vector x-components in NED system**
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the local NED system, returned as a scalar value, vector, matrix, or N-D array.

### `vEast` — Vector y-components in NED system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the local NED system, returned as a scalar value, vector, matrix, or N-D array.

### wDown — Vector z-components in NED system
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the local NED system, returned as a scalar value, vector, matrix, or N-D array.

## See Also
`ecef2enuv` | `ecef2ned` | `ned2ecefv`

# ecefOffset

Cartesian ECEF offset between geodetic positions

## Syntax

```
[deltaX,deltaY,deltaZ] = ecefOffset(spheroid,lat1,lon1,h1,lat2,lon2,
h2)
[] = ecefOffset( ___ ,angleUnit)
```

## Description

`[deltaX,deltaY,deltaZ] = ecefOffset(spheroid,lat1,lon1,h1,lat2,lon2,h2)` returns the offset from the geodetic position specified by `lat1`,`lon1`, and`h1` to the position specified by `lat2`,`lon2`, and `h2`, as a Cartesian 3-vector with components `deltaX`,`deltaY`,`deltaZ`. The geodetic latitudes, longitudes, and heights refer to the reference body specified by the spheroid object, `spheroid`. The components `deltaX`,`deltaY`, and `deltaZ` refer to a spheroid-centric Earth-Centered Earth-Fixed (ECEF) Cartesian system. `lat1`,`lon1`, `lat2`, and `lon2` are in degrees by default. `h1` and `h2` must be expressed in the same length unit as the spheroid. `deltaX`,`deltaY`, and `deltaZ` will be expressed in these units, also.

`[] = ecefOffset( ___ ,angleUnit)` uses the string `angleUnit`, which matches either `'degrees'` or `'radians'`, to specify the units of the latitude and longitude coordinates.

## Examples

### Calculate Cartesian ECEF Offset Between Geodetic Positions

Define two locations in geodetic coordinates. Note that heights are relative to the ellipsoid, not sea level. Thus, the height for Miami is a negative number.

```
latParis = dms2degrees([48 51 24]);
lonParis = dms2degrees([2 21 03]);
hParis = 80;
```

```
latMiami = dms2degrees([ 25 46 31]);
lonMiami = dms2degrees([-80 12 32]);
hMiami = -25;
wgs84 = wgs84Ellipsoid('meters');
```

Calculate the offset vector between the two geodetic positions.

```
[dx,dy,dz] = ecefOffset(wgs84, ...
    latParis,lonParis,hParis,latMiami,lonMiami,hMiami)


dx =

  -3.2236e+06


dy =

  -5.8359e+06


dz =

  -2.0235e+06
```

Calculate the straight-line (through the Earth) distance from Paris to Miami.

```
d = hypot(hypot(dx,dy),dz)


d =

   6.9674e+06
```

## Input Arguments

### spheroid — Reference body
spheroid object

Reference body, specified as a spheroid object: oblateSpheroid,
referenceEllipsoid, or referenceSphere.

### `lat1` — Geodetic latitude in degrees
geodetic coordinate array

Geodetic latitude in degrees, specified as geodetic coordinate array.

Data Types: `single | double`

### `lon1` — Geodetic longitude in degrees
geodetic coordinate array

Geodetic longitude in degrees, specified as a geodetic coordinate array.

Data Types: `single | double`

### `h1` — Ellipsoidal height
numeric array

Ellipsoidal height, specified as numeric array

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

### `lat2` — Geodetic latitude in degrees
geodetic coordinate array

Geodetic latitude in degrees, specified as geodetic coordinate array.

Data Types: `single | double`

### `lon2` — Geodetic longitude in degrees
geodetic coordinate array

Geodetic longitude in degrees, specified as a geodetic coordinate array.

Data Types: `single | double`

### `h2` — Ellipsoidal height
numeric array

Ellipsoidal height, specified as a numeric array.

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

### `angleUnit` — Units of the latitude and longitude coordinates
`'degrees' | 'radians'`

Units of the latitude and longitude coordinates, specified as the character string
`'degrees'` or `'radians'`.

Data Types: `char`

## Output Arguments

### `deltaX` — Offset between geodetic positions `lat1` and `lat2`
geodetic coordinate array

Offset between geodetic positions `lat1` and `lat2`, returned as a Cartesian 3-vector.

### `deltaY` — Offset between geodetic positions `lon1` and `lon2`
geodetic coordinate array

Offset between geodetic positions `lon1` and `lon2`, returned as a Cartesian 3-vector.

### `deltaZ` — Offset between geodetic positions `h1` and `h2`
scalar

Offset between geodetic positions `h1` and `h2`, returned as a Cartesian 3-vector.

## See Also
oblateSpheroid | referenceEllipsoid | referenceSphere | `ecef2geodetic` |
`geodetic2ecef`

**Introduced in R2012b**

# egm96geoid

Read 15-minute gridded geoid heights from EGM96

## Syntax

```
[Z, refvec] = egm96geoid(samplefactor)
[Z, refvec] = egm96geoid(samplefactor,latlim,lonlim)
```

## Description

[Z, refvec] = egm96geoid(samplefactor) imports global geoid height in meters from the EGM96 geoid model. The data set is gridded at 15-minute intervals, but may be down-sampled as specified by the positive integer samplefactor. The result is returned in the regular data grid Z along with referencing vector refvec. At full resolution (a samplefactor of 1), Z will be 721-by-1441.

The gridded EGM96 data set must be on your path in a file named 'WW15MGH.GRD'.

[Z, refvec] = egm96geoid(samplefactor,latlim,lonlim) imports data for the part of the world within the specified latitude and longitude limits. The limits must be two-element vectors in units of degrees. Longitude limits can be defined in the range [-180 180] or [0 360]. For example, lonlim = [170 190] returns data centered on the dateline, while lonlim = [-10 10] returns data centered on the prime meridian.

## Background

Although the Earth is round, it is not exactly a sphere. The shape of the Earth is usually defined by the geoid, which is defined as a gravitational equipotential surface, but can be conceptualized as the shape the ocean surface would take in the absence of waves, weather, and land. For cartographic purposes it is generally sufficient to treat the Earth as a sphere or ellipsoid of revolution. For other applications, a more detailed model of the geoid such as EGM 96 may be required. EGM 96 is a spherical harmonic model of the geoid complete to degree and order 360. This function reads from a file of gridded geoid heights derived from the EGM 96 harmonic coefficients.

## Examples

Read the EGM 96 geoid grid for the world, taking every 10th point.

```
[N,refvec] = egm96geoid(10);
```

Read a subset of the geoid grid at full resolution and interpolate to find the geoid height at a point between grid points.

```
[N,refvec] = egm96geoid(1,[-10 -12],[129 132]);
n = ltln2val(N,refvec,-11.1,130.22,'bicubic')

n =
 52.7151
```

## More About

### Tips

This function reads the 15-minute EGM96 grid file `WW15MGH.GRD`. The grid is available as either a DOS self-extracting compressed file or a UNIX compressed file. Do not modify the file once it has been extracted.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

---

Maps will extend a half a cell outside the requested map limits.

There are 721 rows and 1441 columns of values in the grid at full resolution. The low resolution data in `GEOID.MAT` is derived from the EGM 96 grid.

### See Also
`ltln2val`

# elevation

Local vertical elevation angle, range, and azimuth

---

**Note:** `elevation` will be removed in a future release. Use `geodetic2aer` instead.

The reference point comes second in the `geodetic2aer` argument list, and the outputs are ordered differently. The replacement pattern is:

```
[azimuthangle, elevationangle, slantrange] = geodetic2aer(lat2,
lon2, alt2, lat1, lon1, alt1, spheroid, ...)
```

Unlike `elevation`, `geodetic2aer` requires a spheroid input, and it must be must be an `oblateSpheroid`, `referenceEllipsoid`, or `referenceSphere` object, not a 2-by-1 ellipsoid vector.

You can use the following steps to convert an ellipsoid vector, `ellipsoid`, to an `oblateSpheroid` object, `spheroid`:

- `spheroid = oblateSpheroid;`
- `spheroid.SemimajorAxis = ellipsoid(1);`
- `spheroid.Eccentricity = ellipsoid(2);`

When elevation is called with only 6 inputs, the GRS 80 reference ellipsoid, in meters, is used by default. To replace this usage, use `referenceEllipsoid('GRS80','meters')` as the spheroid input for `geodetic2aer`.

If an `angleunits` input is included, it must follow the `spheroid` input in the call to `geodetic2aer`, rather than preceding it.

`elevation` can be called with a `lengthunits` string, but `geodetic2aer` has no such input. Set the `LengthUnit` property of the input spheroid to the desired value instead. In this case a `referenceEllipsoid` or `referenceSphere` object must be used (not an `oblateSpheroid` object).

---

## Syntax

```
[elevationangle,slantrange,azimuthangle] = ...
```

```
elevation(lat1,lon1,alt1,lat2,lon2,alt2)
[...] = elevation(lat1,lon1,alt1,lat2,lon2,alt2,...
  angleunits)
[...] = elevation(lat1,lon1,alt1,lat2,lon2,alt2,...
  angleunits,distanceunits)
[...] = elevation(lat1,lon1,alt1,lat2,lon2,alt2,...
  angleunits,ellipsoid)
```

## Description

`[elevationangle,slantrange,azimuthangle] = ...`
  `elevation(lat1,lon1,alt1,lat2,lon2,alt2)` computes the elevation angle,
slant range, and azimuth angle of point 2 (with geodetic coordinates `lat2`, `lon2`, and
`alt2`) as viewed from point 1 (with geodetic coordinates `lat1`, `lon1`, and `alt1`). The
coordinates `alt1` and `alt2` are ellipsoidal heights. The elevation angle is the angle
of the line of sight above the local horizontal at point 1. The slant range is the three-
dimensional Cartesian distance between point 1 and point 2. The azimuth is the angle
from north to the projection of the line of sight on the local horizontal. Angles are in units
of degrees; altitudes and distances are in meters. The figure of the earth is the default
ellipsoid (GRS 80).

Inputs can be vectors of points, or arrays of any shape, but must match in size, with the
following exception: Elevation, range, and azimuth from a single point to a set of points
can be computed very efficiently by providing scalar coordinate inputs for point 1 and
vectors or arrays for point 2.

`[...] = elevation(lat1,lon1,alt1,lat2,lon2,alt2,...`
  `angleunits)` uses the string *angleunits* to specify the units of the input and output
angles. If the string *angleunits* is omitted, `'degrees'` is assumed.

`[...] = elevation(lat1,lon1,alt1,lat2,lon2,alt2,...`
  `angleunits,distanceunits)` uses the string *distanceunits* to specify the
altitude and slant-range units. If the string *distanceunits* is omitted, `'meters'` is
assumed. Any units string recognized by `unitsratio` may be used.

`[...] = elevation(lat1,lon1,alt1,lat2,lon2,alt2,...`
  `angleunits,ellipsoid)` uses `ellipsoid` to specify the ellipsoid. `ellipsoid` is
a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form
`[semimajor_axis eccentricity]`. If `ellipsoid` is supplied, the altitudes must be in

the same units as the semimajor axis, and the slant range will be returned in these units. If `ellipsoid` is omitted, the default is a unit sphere. Distances are in meters unless otherwise specified.

---

**Note** The line-of-sight azimuth angles returned by `elevation` will generally differ slightly from the corresponding outputs of `azimuth` and `distance`, except for great circle azimuths on a spherical earth.

---

## Examples

Find the elevation angle of a point 90 degrees from an observer assuming that the observer and the target are both 1000 km above the Earth.

```
lat1 = 0; lon1 = 0; alt1 = 1000*1000;
lat2 = 0; lon2 = 90; alt2 = 1000*1000;
elevang = elevation(lat1,lon1,alt1,lat2,lon2,alt2)

elevang =
    -45
```

Visually check the result using the `los2` line of sight function. Construct a data grid of zeros to represent the Earth's surface. The `los2` function with no output arguments creates a figure displaying the geometry.

```
Z = zeros(180,360);
refvec = [1 90 -180];
los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt1);
```
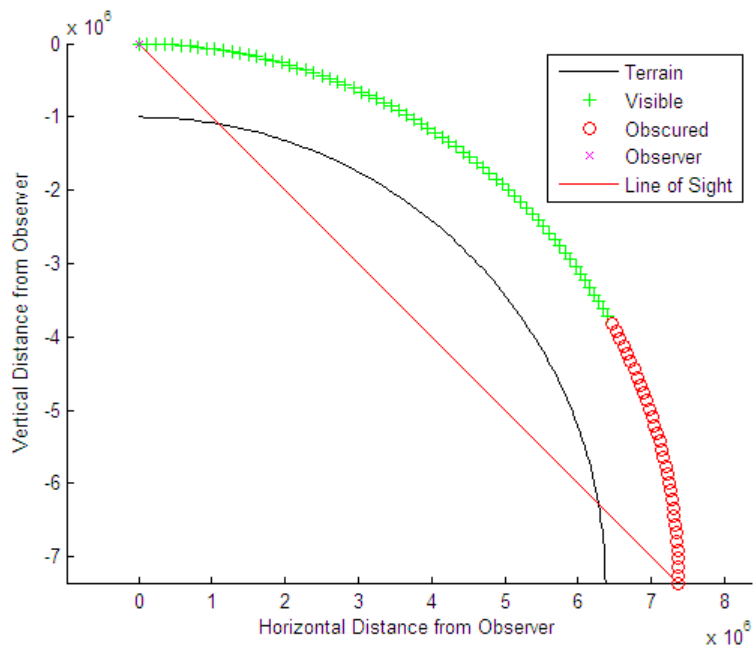
## See Also
oblateSpheroid

# ellipse1

Geographic ellipse from center, semimajor axes, eccentricity, and azimuth

## Syntax

```
[lat,lon] = ellipse1(lat0,lon0,ellipse)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid)
[lat,lon] = ellipse1( ___ ,angleUnit)
[lat,lon] =
ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,angleUnit,npts)
[lat,lon] = ellipse1(trackStr,...)
mat = ellipse1(...)
```

## Description

`[lat,lon] = ellipse1(lat0,lon0,ellipse)` computes ellipse(s) with center(s) at `lat0,lon0`. The ellipse is defined by the third input, which is of the form `[semimajor axis,eccentricity]`, where the eccentricity input can be a two-element row vector or a two-column matrix. The ellipse input must have the same number of rows as the input scalar or column vectors `lat0` and `lon0`. The input semimajor axis is in degrees of arc length on a sphere. All ellipses are oriented so that their major axes run north-south.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset)` computes the ellipse(s) where the major axis is rotated from due north by an azimuth offset. The `offset` angle is measured clockwise from due north. If `offset = []`, then no offset is assumed.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az)` uses the input `az` to define the ellipse arcs computed. The arc azimuths are measured clockwise from due north. If `az` is a column vector, then the arc length is computed from due north. If `az` is a two-column matrix, then the ellipse arcs are computed starting at the azimuth in the first column and ending at the azimuth in the second column. If `az = []`, then a complete ellipse is computed.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid)` computes the ellipse on the ellipsoid defined by the input `ellipsoid`. `ellipsoid` is

a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. If omitted, the unit sphere, is assumed. When an ellipsoid is supplied, the input semimajor axis must be in the same units as the ellipsoid semimajor axes. In this calling form, the units of the ellipse semimajor axis are not assumed to be in degrees.

`[lat,lon] = ellipse1( ___ ,angleUnit)` where `angleUnit` defines the units of the inputs and outputs. `angleUnit` can be `'degrees'` or `'radians'`. If the `angleUnit` string is omitted, `'degrees'` is assumed.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,angleUnit,npts)` uses the scalar `npts` to determine the number of points per ellipse computed. If `npts` is omitted, 100 points are used.

`[lat,lon] = ellipse1(trackStr,...)` uses the `trackStr` string to define either great circle or rhumb line distances from the ellipse center. If `trackStr = 'gc'`, then great circle distances are computed (the default). If `trackStr = 'rh'`, then rhumb line distances are computed.

`mat = ellipse1(...)` returns a single output argument where `mat = [lat lon]`. This is useful if only one ellipse is computed.

You can define multiple ellipses with a common center by providing scalar `lat0` and `lon0` inputs and a two-column `ellipse` matrix.

## Examples

Create and plot the small ellipse centered at (0º,0º), with a semimajor axis of 10º and a semiminor axis of 5º.

```
axesm mercator
ecc = axes2ecc(10,5);
plotm(0,0,'r+')
[elat,elon] = ellipse1(0,0,[10 ecc],45);
plotm(elat,elon)
```

If the desired radius is known in some nonangular distance unit, use the radius returned by the `earthRadius` function as the ellipsoid input to set the range units. (Use an empty azimuth entry to specify a full ellipse.)

```
[elat,elon] = ellipse1(0,0,[550 ecc],45,[],earthRadius('nm'));
plotm(elat,elon,'m--')
```

For just an arc of the ellipse, enter an azimuth range:

```
[elat,elon] = ellipse1(0,0,[5 ecc],45,[-30 70]);
plotm(elat,elon,'c-')
```
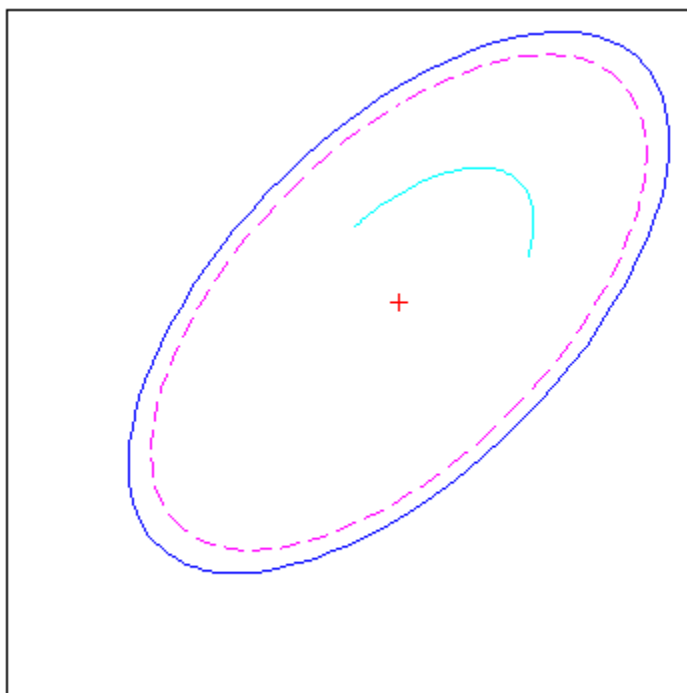


## See Also
axes2ecc | scircle1 | track1

# encodem

Fill in regular data grid from seed values and locations

## Syntax

```
newgrid = encodem(Z,seedmat)
newgrid = encodem(Z,seedmat,stopvals)
```

## Description

`newgrid = encodem(Z,seedmat)` fills in regions of the input data grid, `Z`, with desired new values. The boundary consists of the edges of the matrix and any entries with the value 1. The *seeds*, or starting points, and the values associated with them, are specified by the three-column matrix `seedmat`, the rows of which have the form `[row column value]`.

`newgrid = encodem(Z,seedmat,stopvals)` allows you to specify a vector, `stopvals`, of stopping values. Any value that is an element of `stopvals` will act as a boundary.

This function *fills in* regions of data grids with desired values. If a *boundary* exists, the new value replaces all entries in all four directions until the boundary is reached. The boundary is made up of selected stopping values and the edges of the matrix. The new value tries to flood the region exhaustively, stopping only when no new spaces can be reached by moving up, down, left, or right without hitting a stopping value.

## Examples

For this imaginary map, fill in the upper right region with 7s and the lower left region with 3s:

```
Z = eye(4)

Z =
     1     0     0     0
     0     1     0     0
```

```
      0     0     1     0
      0     0     0     1

newgrid = encodem(Z,[4,1,3; 1,4,7])

newgrid =
      1     7     7     7
      3     1     7     7
      3     3     1     7
      3     3     3     1
```

## See Also

getseeds | imbedm

# enu2aer

Local Cartesian ENU to local spherical AER

## Syntax

```
[az,elev,slantRange] = enu2aer(xEast,yNorth,zUp)
[ ___ ] = enu2aer( ___ ,angleUnit)
```

## Description

`[az,elev,slantRange] = enu2aer(xEast,yNorth,zUp)` returns coordinates in a local spherical system corresponding to coordinates `xEast`, `yNorth`, `zUp` in a local east-north-up (ENU) Cartesian system having the same local origin. Any of the three numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = enu2aer( ___ ,angleUnit)` adds `angleUnit` which specifies the units outputs `az`, `elev`.

## Input Arguments

### `xEast` — Local ENU x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local ENU system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### `yNorth` — Local ENU y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local ENU system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**zUp — Local ENU z-coordinates**
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local ENU system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**angleUnit — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

# Output Arguments

**az — Azimuth angles**
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the half-open interval [0 360).

**elev — Elevation angles**
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface normal. Units determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the closed interval [-90 90].

**slantRange — Distances from local origin**
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is computed.

# See Also
aer2enu | ned2aer

# enu2ecef

Local Cartesian ENU to geocentric ECEF

## Syntax

```
[X,Y,Z] = enu2ecef(xEast,yNorth,zUp,lat0,lon0,h0,spheroid)
[ ___ ] = enu2ecef( ___ ,angleUnit)
```

## Description

`[X,Y,Z] = enu2ecef(xEast,yNorth,zUp,lat0,lon0,h0,spheroid)` returns Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian coordinates corresponding to coordinates `xEast`, `yNorth`, `zUp` in a local east-north-up (ENU) Cartesian system. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = enu2ecef( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0` and `lon0`.

## Input Arguments

### `xEast` — Local ENU x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local ENU system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `yNorth` — Local ENU y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local ENU system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `zUp` — Local ENU z-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local ENU system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### X — ECEF x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

### Y — ECEF y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

### Z — ECEF y-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

## See Also
`aer2ecef` | `ecef2enu` | `enu2geodetic` | `ned2ecef`

# enu2ecefv

Rotate vector from local ENU to geocentric ECEF

## Syntax

```
[U,V,W] = enu2ecefv(uEast,vNorth,wUp,lat0,lon0)
[ ___ ] = enu2ecefv( ___ ,angleUnit)
```

## Description

`[U,V,W] = enu2ecefv(uEast,vNorth,wUp,lat0,lon0)` returns coordinates in a local spherical system corresponding to coordinates `uEast`, `vNorth`, and `wUp`, in a local north-east-up (ENU) Cartesian system having the same local origin. Any of the three numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = enu2ecefv( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0` and `lon0`.

## Input Arguments

### `uEast` — Vector x-components in ENU system
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the local ENU system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### `vNorth` — Vector y-components in ENU system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the local ENU system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**wUp — Vector z-components in ENU system**
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the local ENU system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**lat0 — Geodetic latitude of local origin**
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

**lon0 — Longitude of local origin**
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

**angleUnit — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

**U — Vector x-components in ECEF system**
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array.

### V — Vector y-components in ECEF system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array.

### W — Vector z-components in ECEF system
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array.

## See Also
`ecef2enu` | `enu2ecef` | `ned2ecefv`

# enu2geodetic

Local Cartesian ENU to geodetic

## Syntax

```
[lat,lon,h] = enu2geodetic(xEast,yNorth,zUp,lat0,lon0,h0,spheroid)
[ ___ ] = enu2geodetic( ___ ,angleUnit)
```

## Description

`[lat,lon,h] = enu2geodetic(xEast,yNorth,zUp,lat0,lon0,h0,spheroid))`
returns geodetic coordinates corresponding to coordinates `xEast`, `yNorth`, `zUp` in a local
east-north-up (ENU) Cartesian system. Any of the first six numeric input arguments
can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments
must match in size.

`[ ___ ] = enu2geodetic( ___ ,angleUnit)` adds `angleUnit` which specifies the
units of inputs `lat0`, `lon0`, and outputs `lat`, `lon`.

## Input Arguments

### `xEast` — Local ENU x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local ENU system, specified as a scalar value,
vector, matrix, or N-D array. Values must be in units that match the `LengthUnit`
property of the `spheroid` input.

Data Types: `single` | `double`

### `yNorth` — Local ENU y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local ENU system, specified as a scalar value,
vector, matrix, or N-D array. Values must be in units that match the `LengthUnit`
property of the `spheroid` input.

Data Types: `single` | `double`

**`zUp`** — **Local ENU z-coordinates**
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local ENU system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

**`lat0`** — **Geodetic latitude of local origin**
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

**`lon0`** — **Longitude of local origin**
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

**`h0`** — **Ellipsoidal height of local origin**
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

**`spheroid`** — **Reference spheroid**
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### `lat` — Geodetic latitudes
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the closed interval [-90 90].

### `lon` — Longitudes
scalar value | vector | matrix | N-D array

Longitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the interval [-180 180].

### `h` — Ellipsoidal heights
scalar value | vector | matrix | N-D array

Ellipsoidal heights of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object

## See Also
`aer2geodetic` | `enu2ecef` | `geodetic2enu` | `ned2geodetic`

# epsm

Accuracy in angle units for certain map computations

## Compatibility

epsm will be removed in a future release. If necessary, you can replace the expressions listed in the following table with the constants shown:

| epsm() | 1.0E-6 |
|---|---|
| epsm('deg') | 1.0E-6 |
| epsm('rad') | deg2rad(1.0E-6) |

## Syntax

```
epsm
epsm(units)
```

## Description

epsm is the limit of map angular precision. It is useful in avoiding trigonometric singularities, among other things.

epsm(*units*) returns the same angle in units corresponding to any valid angle units string. The default is 'degrees'.

## Examples

The value of epsm is $10^{-6}$ degrees. To put this in perspective, in terms of an angular arc length, the distance is

```
epsmkm = deg2km(epsm)

epsmkm =
```

```
1.1119e-04     % kilometers
```

This is about 11 centimeters, a very small distance on a global scale.

## See Also
```
roundn
```

# eqa2grn

Convert from equal area to Greenwich coordinates

## Syntax

```
[lat,lon] = eqa2grn(x,y)
[lat,lon] = eqa2grn(x,y,origin)
[lat,lon] = eqa2grn(x,y,origin,ellipsoid)
[lat,lon] = eqa2grn(x,y,origin,units)
mat = eqa2grn(x,y,origin...)
```

## Description

`[lat,lon] = eqa2grn(x,y)` converts the equal-area coordinate points `x` and `y` to the Greenwich (standard geographic) coordinates `lat` and `lon`.

`[lat,lon] = eqa2grn(x,y,origin)` specifies the location in the Greenwich system of the *x-y* origin (0,0). The two-element vector `origin` must be of the form `[latitude longitude]`. The default places the origin at the Greenwich coordinates (0º,0º).

`[lat,lon] = eqa2grn(x,y,origin,ellipsoid)` specifies the ellipsoidal model of the figure of the Earth using `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The `ellipsoid` is a unit sphere by default.

`[lat,lon] = eqa2grn(x,y,origin,units)` specifies the units for the outputs, where *units* is any valid angle units string. The default value is `'degrees'`.

`mat = eqa2grn(x,y,origin...)` packs the outputs into a single variable.

This function converts data from equal-area *x-y* coordinates to geographic (latitude-longitude) coordinates. The opposite conversion can be performed with `grn2eqa`.

## Examples

```
[lat,lon] = eqa2grn(.5,.5)
```

```
lat =
    30.0000
lon =
    28.6479
```

## See Also

`grn2eqa` | `hista`

# etopo

Read gridded global relief data (ETOPO products)

## Syntax

```
[Z, refvec] = etopo
[Z, refvec] = etopo(samplefactor)
[Z, refvec] = etopo(samplefactor, latlim, lonlim)
[Z, refvec] = etopo(folder, ...)
[Z, refvec] = etopo(filename, ...)
[Z, refvec] = etopo({'etopo5.northern.bat',
'etopo5.southern.bat'}, ...)
```

## Description

[Z, refvec] = etopo reads the ETOPO data for the entire world from the ETOPO data in the current folder. The etopo function searches the current folder first for ETOPO1c binary data, then ETOPO2V2c binary data, then ETOPO2 (2001) binary data, then ETOPO5 binary data, and finally ETOPO5 ASCII data. Once the function finds a case-insensitive file name match, it reads the data. See the table Supported ETOPO Data File Names for a list of possible file names. The etopo function returns the data grid, Z, as an array of elevations. Data values, in whole meters, represent the elevation of the center of each cell. refvec, the associated three-element referencing vector, geolocates Z.

[Z, refvec] = etopo(samplefactor) reads the data for the entire world, downsampling the data by samplefactor. The scalar integer samplefactor when equal to 1 gives the data at its full resolution (10800 by 21600 values for ETOPO1 data, 5400 by 10800 values for ETOPO2 data, and 2160 by 4320 values for ETOPO5 data). When samplefactor is an integer $n$ greater than one, the etopo function returns every $n^{\text{th}}$ point. If you omit samplefactor or leave it empty, it defaults to 1. (If the etopo function reads an older, ASCII ETOPO5 data set, then samplefactor must divide evenly into the number of rows and columns of the data file.)

[Z, refvec] = etopo(samplefactor, latlim, lonlim) reads the data for the part of the world within the specified latitude and longitude limits. Specify the limits of the desired data as two-element vectors of latitude, latlim, and longitude, lonlim, in degrees. Specify the elements of latlim and lonlim in ascending order. Specify lonlim

in the range `[0 360]` for ETOPO5 data and `[-180 180]` for ETOPO2 and ETOPO1 data. If `latlim` is empty, the latitude limits are [-90 90]. If `lonlim` is empty, the file type determines the longitude limits.

`[Z, refvec] = etopo(folder, ...)` allows you to use the variable `folder` to specify the path for the ETOPO data file. Otherwise, the `etopo` function searches the current folder for the data.

`[Z, refvec] = etopo(filename, ...)` reads the ETOPO data from `filename`. The variable `filename`, a case-insensitive string, specifies the name of the ETOPO file, as referenced in the ETOPO data file names table. Include the folder name in `filename` or place the file in the current folder or in a folder on the MATLAB path.

`[Z, refvec] = etopo({'etopo5.northern.bat', 'etopo5.southern.bat'}, ...)` reads the ETOPO data from the specified case-insensitive ETOPO5 ASCII data files. Place the files in the current folder or in a folder on the MATLAB path.

## Definitions

According to the National Geophysical Data Center (NGDC) Web site, ETOPO models combine regional and global land topography and ocean bathymetry data from many data sources. ETOPO1, the most recent model, has an Ice Surface version showing the top of the Antarctic and Greenland ice sheets and a Bedrock version showing the bedrock below the ice sheets. For detailed information about the data sources used to create the ETOPO1 model, see the NGDC Web site. NGDC lists the ETOPO2 and ETOPO5 models as deprecated but still available.

| Model | Cell Size | Date Available |
|---|---|---|
| ETOPO1 | 1-arc-minute | March 2009 |
| ETOPO2v2 | 2-minute | 2006 |
| ETOPO2 | 2-minute | 2001 |
| ETOPO5 | 5-minute | 1988 |

## Examples

Read and display ETOPO2V2c data from the file `'ETOPO2V2c_i2_LSB.bin'` downsampled to half-degree cell size and display the boundary of the land areas.

```
samplefactor = 15;
[Z, refvec] = etopo('ETOPO2V2c_i2_LSB.bin', samplefactor);
figure
worldmap world
geoshow(Z, refvec, 'DisplayType', 'texturemap');
demcmap(Z, 256);
geoshow('landareas.shp', 'FaceColor', 'none', ...
    'EdgeColor', 'black');
```
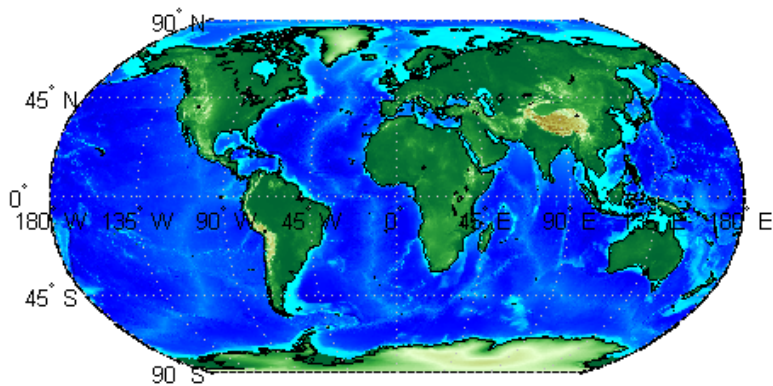


Read and display ETOPO1 data for a region around Australia.

```
figure
worldmap australia
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
[Z, refvec] = etopo('etopo1_ice_c.flt', 1, latlim, lonlim);
geoshow(Z, refvec, 'DisplayType', 'surface');
demcmap(Z, 256);
```
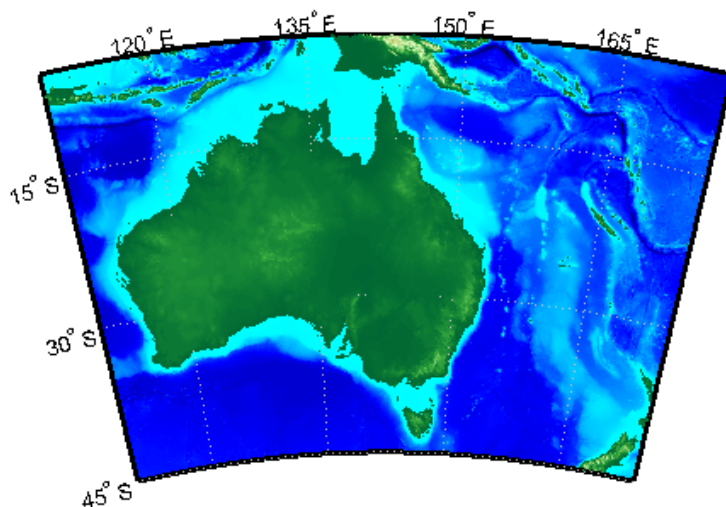
## More About

**Tips**

**Supported ETOPO Data File Names**

| Format | Filenames |
|---|---|
| ETOPO1c (cell) | • `etopo1_ice_c.flt` |
| | • `etopo1_bed_c.flt` |
| | • `etopo1_ice_c_f4.flt` |
| | • `etopo1_bed_c_f4.flt` |
| | • `etopo1_ice_c_i2.bin` |
| | • `etopo1_bed_c_i2.bin` |
| ETOPO2V2c (cell) | • `ETOPO2V2c_i2_MSB.bin` |
| | • `ETOPO2V2c_i2_LSB.bin` |
| | • `ETOPO2V2c_f4_MSB.flt` |
| | • `ETOPO2V2c_f4_LSB.flt` |

| Format | Filenames |
|---|---|
| | • `ETOPO2V2c.hdf` |
| ETOPO2 (2001) | • `ETOPO2.dos.bin` |
| | • `ETOPO2.raw.bin` |
| ETOPO5 (binary) | • `ETOPO5.DOS` |
| | • `ETOPO5.DAT` |
| ETOPO5 (ASCII) | • `etopo5.northern.bat` |
| | • `etopo5.southern.bat` |

• For details on locating ETOPO data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

## References

[1] "2-minute Gridded Global Relief Data (ETOPO2v2)," U.S. Department of Commerce, National Oceanic and Atmospheric Administration, National Geophysical Data Center, 2006.

[2] Amante, C. and B. W. Eakins, "ETOPO1 1 Arc-Minute Global Relief Model: Procedures, Data Sources and Analysis," *NOAA Technical Memorandum NESDIS NGDC-24*, March 2009.

[3] "Digital Relief of the Surface of the Earth," *Data Announcement 88-MGG-02*, NOAA, National Geophysical Data Center, Boulder, Colorado, 1988.

[4] "ETOPO2v2 Global Gridded 2-minute Database," National Geophysical Data Center, National Oceanic and Atmospheric Administration, U.S. Dept. of Commerce.

## See Also
`gtopo30 | tbase | usgsdem`

# etopo5

Read global 5-min digital terrain data

## Syntax

---

**Note** `etopo5` will be removed in a future version; use `etopo` instead.

---

```
[Z, refvec] = etopo5
[Z, refvec] = etopo5(samplefactor)
[[Z, refvec] = etopo5(samplefactor, latlim, lonlim)
[Z, refvec] = etopo5(folder, ...)
[Z, refvec] = etopo5(file, ...)
```

## Description

`[Z, refvec] = etopo5` reads the topography data for the entire world for the data in the current folder. The current folder is searched first for `ETOPO2` binary data, followed by ETOPO5 binary data, followed by ETOPO5 ASCII data from the file names `etopo5.northern.bat` and `etopo5.southern.bat`. Once a match is found the data is read. The data grid, `Z`, is returned as an array of elevations. Data values are in whole meters, representing the elevation of the center of each cell. `refvec` is the associated three-element referencing vector that geolocates `Z`.

`[Z, refvec] = etopo5(samplefactor)` reads the data for the entire world, downsampling the data by `samplefactor`. `samplefactor` is a scalar integer, which when equal to 1 gives the data at its full resolution (1080 by 4320 values). When `samplefactor` is an integer n greater than one, every n$^{\text{th}}$ point is returned. `samplefactor` must divide evenly into the number of rows and columns of the data file. If `samplefactor` is omitted or empty, it defaults to 1.

`[[Z, refvec] = etopo5(samplefactor, latlim, lonlim)` reads the data for the part of the world within the specified latitude and longitude limits. The limits of the desired data are specified as two-element vectors of latitude, `latlim`, and longitude, `lonlim`, in degrees. The elements of `latlim` and `lonlim` must be in ascending order.

If latlim is empty the latitude limits are [-90 90]. lonlim must be specified in the range [0 360]. If lonlim is empty, the longitude limits are [0 360].

[Z, refvec] = etopo5(folder, ...) allows the path for the data file to be specified by folder rather than the current folder.

[Z, refvec] = etopo5(file, ...) reads the data from file, where file is a string or a cell array of strings containing the name or names of the data files.

ETOPO5 is being superseded by ETOPO2 and the TerrainBase digital terrain model. See the tbase external interface function for more information.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web Site: "Finding Geospatial Data".

---

# Examples

## Example 1

Read every tenth point in the data set:

```
% Read and display the ETOPO5 data from the folder 'etopo5'
% downsampled by a factor of 10.
[Z, refvec] = etopo5('etopo5',10);
axesm merc
geoshow(Z, refvec, 'DisplayType', 'surface');
demcmap(Z);
```

## Example 2

Read in data for Korea and Japan at the full resolution:

```
samplefactor = 1; latlim = [30 45]; lonlim = [115 145];
[Z,refvec] = etopo5(samplefactor,latlim,lonlim);
whos Z

  Name      Size           Bytes  Class
  Z        180x360        518400  double array
```

## See Also

etopo | gtopo30 | tbase | usgsdem

# extractfield

Field values from structure array

## Syntax

```
a = extractfield(s, name)
```

## Description

`a = extractfield(s, name)` returns the field values specified by the field named `name` into the 1-by-n output array `a`. n is the total number of elements in the field `name` of structure s, that is, `n = numel([s(:).(name)])`. `name` is a case-sensitive string defining the field name of the structure `s`. `a` is a cell array if any field values in the field name contain a string or if the field values are not uniform in type; otherwise `a` is the same type as the field values. The shape of the input field is not preserved in `a`.

## Examples

```
% Plot the X, Y coordinates of the road's shape
roads = shaperead('concord_roads.shp');
plot(extractfield(roads,'X'),extractfield(roads,'Y'));

% Extract the names of the roads
roads = shaperead('concord_roads.shp');
names = extractfield(roads,'STREETNAME');

% Extract a mix-type field into a cell array
S(1).Type = 0;
S(2).Type = logical(0);
mixedType = extractfield(S,'Type');
```
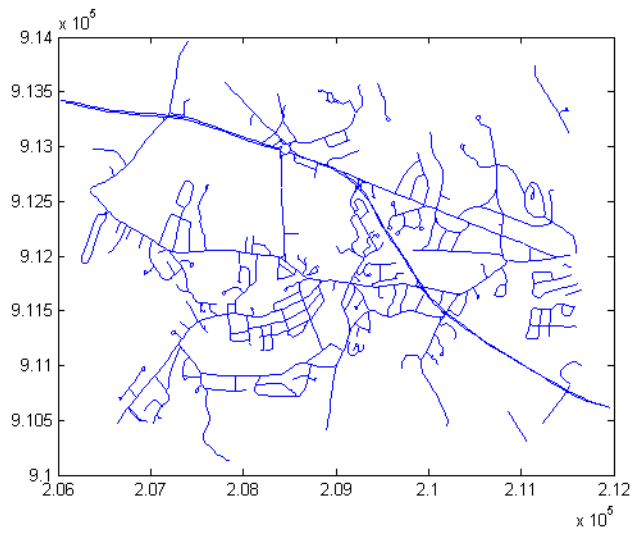
## See Also
```
struct | shaperead
```

# extractm

Coordinate data from line or patch display structure

## Compatibility

`extractm` will be removed in a future release. The use of display structures is not recommended. Use geoshape vectors instead.

## Syntax

```
[lat,lon] = extractm(display_struct,object_str)
[lat,lon] = extractm(display_struct,object_strings)
[lat,lon] = extractm(display_struct,object_strings,searchmethod)
[lat,lon] = extractm(display_struct)
[lat,lon,indx] = extractm(...)
mat = extractm(...)
```

## Description

`[lat,lon] = extractm(display_struct,object_str)` extracts latitude and longitude coordinates from those elements of `display_struct` having `'tag'` fields that begin with the string specified by *object_str*. `display_struct` is a Mapping Toolbox display structure in which the `'type'` field has a value of either 'line' or `'patch'`. The output `lat` and `lon` vectors include NaNs to separate the individual map features. The comparison of `'tag'` values is not case-sensitive.

`[lat,lon] = extractm(display_struct,object_strings)`, where *object_strings* is a character array or a cell array of strings, selects features with `'tag'` fields matching any of several different strings. Character array objects will have trailing spaces stripped before matching.

`[lat,lon] = extractm(display_struct,object_strings,searchmethod)` controls the method used to match the values of the `'tag'` field in `display_struct`. *searchmethod* can be one of three strings:

| | |
|---|---|
| `'strmatch'` | Search for matches at the beginning of the tag |
| `'findstr'` | Search within the tag |
| `'exact'` | Search for exact matches. Note that when *searchmethod* is specified the search is case-sensitive. |

`[lat,lon] = extractm(display_struct)` extracts all vector data from the input map structure.

`[lat,lon,indx] = extractm(...)` also returns the vector `indx` identifying which elements of `display_struct` met the selection criteria.

`mat = extractm(...)` returns the vector data in a single matrix, where `mat = [lat lon]`.

## Examples

Extract the District of Columbia from the low-resolution U.S. vector data:

```
load greatlakes
[lat, lon] = extractm(greatlakes, 'Erie');
axesm mercator
geoshow(lat,lon, 'DisplayType','polygon', 'FaceColor','blue')
```
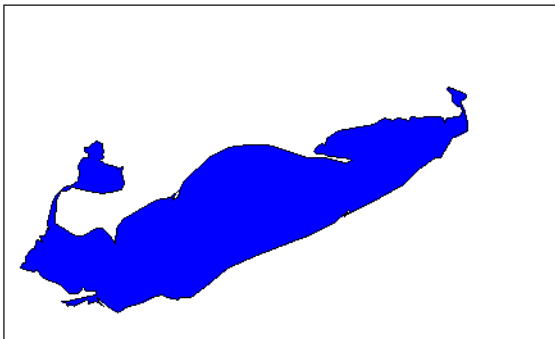
# More About

### Tips

A Version 1 display structure is a MATLAB structure that can contain line, patch, text, regular data grid, geolocated data grid, and certain other objects and fixed attributes. In Mapping Toolbox Version 2, a new data structure for vector geodata was introduced (called a *mapstruct* or a *geostruct*, depending on whether coordinates it contains are projected or unprojected). Geostructs and mapstructs have few required fields and can include any number of user-defined fields, giving them much greater flexibility to represent vector geodata. For information about the contents and format of display structures, see "Version 1 Display Structures" on page 1-200 in the reference page for `displaym`. For information about converting display structures to geographic data structures, see the reference page for `updategeostruct`, which performs such conversions.

## See Also

`displaym` | `extractfield` | `mlayers` | `geoshow` | `mapshow` | `updategeostruct` | geoshape

# fill3m

Project filled 3-D patch objects on map axes

## Syntax

```
h = fill3m(lat,lon,z,cdata)
h = fill3m(lat,lon,z,PropertyName,PropertyValue,...)
```

## Description

`h = fill3m(lat,lon,z,cdata)` projects and displays any patch object with vertices defined by vectors `lat` and `lon` to the current map axes. The scalar `z` indicates the altitude plane at which the patch is displayed. The input `cdata` defines the patch face color. The patch handle or handles, `h`, can be returned.

`h = fill3m(lat,lon,z,PropertyName,PropertyValue,...)` allows any property name/property value pair supported by `patch` to be assigned to the `fill3m` object.

## Examples

```
lat = [30 15 0 0 0 15 30 30]';
lon = [-60 -60 -60 0 60 60 60 0]';
axesm bonne; framem
view(3)
fill3m(lat,lon,2,'b')
```
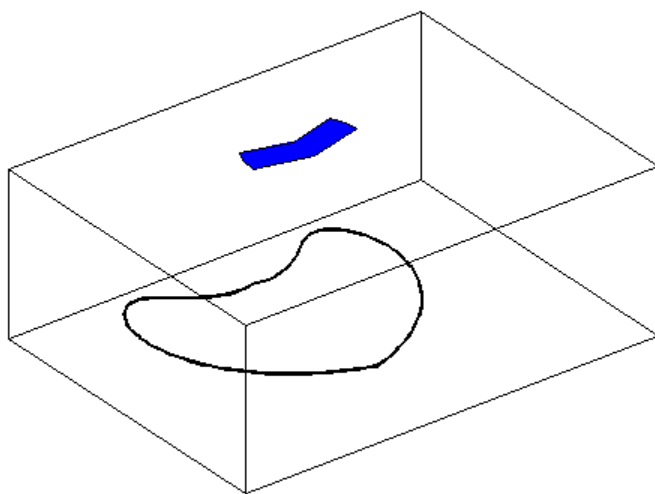
## See Also
`fillm` | `patchesm` | `patchm`

# fillm

Project filled 2-D patch objects on map axes

## Syntax

```
h = fillm(lat,lon,cdata)
h = fillm(lat,lon,'PropertyName',PropertyValue,...)
```

## Description

h = fillm(lat,lon,cdata) projects and displays any patch object with vertices defined by the vectors lat and lon to the current map axes. The input cdata defines the patch face color. The patch handle or handles, h, can be returned.

h = fillm(lat,lon,'PropertyName',PropertyValue,...) allows any property name/property value pair supported by patch to be assigned to the fillm object.

## Examples

```
lat = [30 15 0 0 0 15 30 30]';
lon = [-60 -60 -60 0 60 60 60 0]';
axesm bonne; framem
fillm(lat,lon,'b')
```
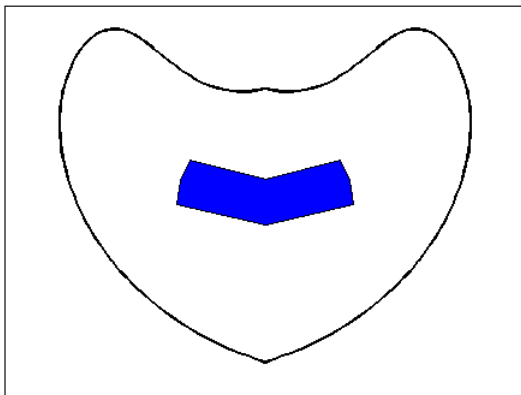
## See Also

fill3m | patchesm | patchm

# filterm

Filter latitudes and longitudes based on underlying data grid

## Syntax

```
[latout,lonout] = filterm(lat,lon,Z,R,allowed)
[latout,lonout,indx] = filterm(lat,lon,Z,R,allowed)
```

## Description

`[latout,lonout] = filterm(lat,lon,Z,R,allowed)` filters a set of latitudes and longitudes to include only those data points which have a corresponding value in `Z` equal to `allowed`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

`[latout,lonout,indx] = filterm(lat,lon,Z,R,allowed)` also returns the indices of the included points.

## Examples

Filter a random set of 100 geographic points. Use the topo map for starters:

```
load topo
```

Then generate 100 random points:

```
lat = -90+180*rand(100,1);
long = -180+360*rand(100,1);
```

Make a land map, which is 1 where topo>0 elevation:

```
land = topo>0;
[newlat,newlong] = filterm(lat,long,land,topolegend,1);
size(newlat)

ans =
    15     1
```

15 of the 100 random points fall on *land*.

## See Also
imbedm | hista | histr

# findm

Latitudes and longitudes of nonzero data grid elements

## Syntax

```
[lat,lon] = findm(Z,R)
[lat,lon] = findm(latz,lonz,Z)
[lat,lon,val] = findm(...)
mat = findm(...)
```

## Description

`[lat,lon] = findm(Z,R)` computes the latitudes and longitudes of the nonzero elements of a regular data grid, `Z`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

`[cells/degree northern_latitude_limit western_longitude_limit]`

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

`[lon lat] = [row col 1] * R`

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

`[lat,lon] = findm(latz,lonz,Z)` returns the latitudes and longitudes of the nonzero elements of a geolocated data grid `Z`, which is an M-by-N logical or numeric array. Typically `latz` and `lonz` are M-by-N latitude-longitude arrays, but `latz` may be a latitude vector of length M and `lonz` may be a longitude vector of length N.

[lat,lon,val] = findm(...) returns the values of the nonzero elements of Z, in addition to their locations.

mat = findm(...) returns a single output, where mat = [lat lon].

This function works in two modes: with a regular data grid and with a geolocated data grid.

## Examples

The data grid can be the result of a logical operation. For instance, you can find all locations with elevations greater than 5500 meters.

```
load topo
[lat, lon] = findm((topo>5500),topolegend);
[lat lon]

ans =
   34.5000    79.5000
   34.5000    80.5000
   30.5000    84.5000
   28.5000    86.5000
```

These points are in the Himalayas. Find the grid values at these locations with setpostn:

```
heights = topo(setpostn(topo,topolegend,lat,lon))

heights =
        5559
        5515
        5523
        5731
```
Use a regular data grid to retrieve the elevations from setpostn.

## See Also
find

# fipsname

Read Federal Information Processing Standard (FIPS) name file used with TIGER thinned boundary files

## Compatibility

fipsname will be removed in a future release. More recent TIGER/Line data sets are available in shapefile format and can be imported using shaperead.

## Syntax

```
struc = fipsname
struc = fipsname(filename)
```

## Description

struc = fipsname opens a file selection window to pick the file, reads the FIPS codes, and returns them in a structure.

struc = fipsname(filename) reads the specified file.

## Background

The TIGER thinned boundary files provided by the U.S. Census use FIPS codes to identify geographic entities. This function reads the FIPS files as provided with the TIGER files. These files generally have names of the format _name.dat_.

## Examples

```
struc = fipsname('st_name.dat')

struc =
1x57 struct array with fields:
```

```
    name
    id

s(1)

ans =
    name: 'Alabama'
      id: 1
```

## More About

### Tips

The FIPS name files, along with TIGER thinned boundary files, are available over the Internet.

# flat2ecc

Eccentricity of ellipse from flattening

## Syntax

```
ecc = flat2ecc(f)
ecc = flat2ecc(f)
```

## Description

`ecc = flat2ecc(f)` computes the eccentricity of an ellipse (or ellipsoid of revolution) given its flattening `f`. Except when the input has 2 columns (or is a row vector), each element is assumed to be a flattening and the output `ecc` has the same size as `f`.

`ecc = flat2ecc(f)`, where `f` has two columns (or is a row vector), assumes that the second column is a flattening, and a column vector is returned.

### See Also
axes2ecc | ecc2flat | n2ecc

# flatearthpoly

Insert points along date line to pole

## Syntax

```
[latf,lonf] = flatearthpoly(lat,lon)
[latf,lonf] = flatearthpoly(lat,lon,longitudeOrigin)
```

## Description

`[latf,lonf] = flatearthpoly(lat,lon)` trims NaN-separated polygons specified by the latitude and longitude vectors `lat` and `lon` to the limits [-180 180] in longitude and [-90 90] in latitude, inserting straight segments along the +/- 180-degree meridians and at the poles. Inputs and outputs are in degrees.

`[latf,lonf] = flatearthpoly(lat,lon,longitudeOrigin)` centers the longitude limits on the longitude specified by the scalar longitude `longitudeOrigin`.

## Examples

Vector data for geographic objects that encompass a pole will inevitably encounter or cross the date line. While the toolbox properly displays such polygons, they can cause problems for functions like the polygon intersection and Boolean operations that work with Cartesian coordinates. When these polygons are treated as Cartesian coordinates, the date line crossing results in a spurious line segment, and the polygon displayed as a patch does not have the interior filled correctly.

```
antarctica = shaperead('landareas', 'UseGeoCoords', true,...
    'Selector', {@(name) strcmp(name,'Antarctica'), 'Name'});
figure; plot(antarctica.Lon, antarctica.Lat); ylim([-100 -60])
```
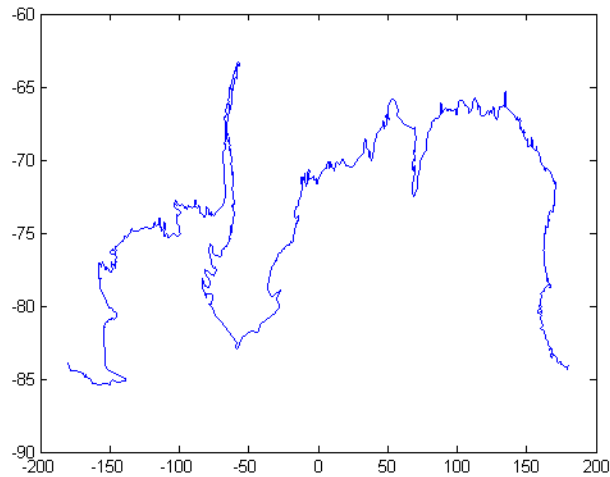
The polygons can be reformatted more appropriately for Cartesian coordinates using the `flatearthpoly` function. The result resembles a map display on a cylindrical projection. The polygon meets the date line, drops down to the pole, sweeps across the longitudes at the pole, and follows the date line up to the other side of the date line crossing.

```
[latf, lonf] = flatearthpoly(antarctica.Lat', antarctica.Lon');
figure; mapshow(lonf, latf, 'DisplayType', 'polygon')
ylim([-100 -60])
xlim([-200 200])
axis square
```
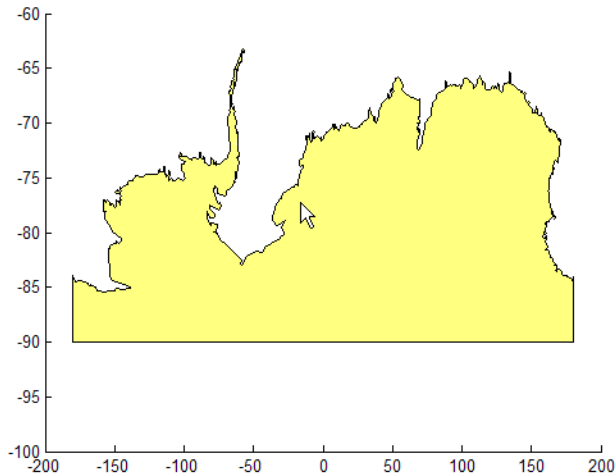
## More About

**Tips**

The polygon topology for the input vectors must be valid. This means that vertices for outer rings (main polygon or "island" polygons) must be in clockwise order, and any inner rings ("lakes") must run in counterclockwise order for the function to work properly. You can use the `ispolycw` function to check whether or not your `lat`, `lon` vectors meet this criterion, and the `poly2cw` and `poly2ccw` functions to correct any that run in the wrong direction.

## See Also
ispolycw | maptrimp | poly2cw | poly2ccw

# framem

Toggle and control display of map frame

## Syntax

```
framem
framem('on')
framem('off')
framem('reset')
framem(linespec)
framem(PropertyName,PropertyValue,...)
```

## Description

`framem` toggles the visibility of the map frame by setting the map axes property `Frame` to `'on'` or `'off'`. The default setting for map axes is `'off'`.

`framem('on')` sets the map axes property `Frame` to `'on'`.

`framem('off')` sets the map axes property `Frame` to `'off'`.

When called with the string argument `'off'`, the map axes property `Frame` is set to `'off'`.

`framem('reset')` resets the entire frame using the current properties. This is essentially a *refresh* option.

`framem(linespec)` sets the map axes `FEdgeColor` property to the color component of any *linespec* string recognized by the MATLAB `line` function.

`framem(PropertyName,PropertyValue,...)` sets the appropriate map axes properties to the desired values. These property names and values are described on the `axesm` reference page.

## More About

### Tips

- You can also create or alter map frame properties using the `axesm` or `setm` functions.
- By default the Clipping property is set to `'off'`. Override this setting with the following code:

```
hgrat = gridm('on');
set(hgrat,'Clipping','on')
```

## See Also

axesm | setm

# fromDegrees

Convert angles from degrees

## Syntax

```
[angle1, angle2, ...] = fromDegrees(toUnits, angle1InDegrees,
angle2InDegrees, ...)
```

## Description

[angle1, angle2, ...] = fromDegrees(*toUnits*, angle1InDegrees, angle2InDegrees, ...) converts angle1InDegrees, angle2InDegrees, ... from degrees to the specified output ("to") angle units. *toUnits* can be either 'degrees' or 'radians' and may be abbreviated. The inputs angle1InDegrees, angle2InDegrees, ... and their corresponding outputs are numeric arrays of various sizes, with size(angleN) matching size(angleNInDegrees).

### See Also
deg2rad | fromRadians | toDegrees | toRadians

# fromRadians

Convert angles from radians

## Syntax

```
[angle1, angle2, ...] = fromRadians(toUnits, angle1InRadians,
angle2InRadians, ...)
```

## Description

`[angle1, angle2, ...] = fromRadians(toUnits, angle1InRadians,
angle2InRadians, ...)` converts `angle1InRadians, angle2InRadians, ...`
from radians to the specified output ("to") angle units. *toUnits* can be either
`'degrees'` or `'radians'` and may be abbreviated. The inputs `angle1InRadians,
angle2InRadians, ...` and their corresponding outputs are numeric arrays of various
sizes, with `size(angleN)` matching `size(angleNInRadians)`.

## See Also
fromDegrees | rad2deg | toDegrees | toRadians

# gc2sc

Center and radius of great circle

## Syntax

```
[lat,lon,radius] = gc2sc(lat0,lon0,az)
[lat,lon,radius] = gc2sc(lat0,lon0,az,angleunits)
mat = gc2sc(...)
```

## Description

`[lat,lon,radius] = gc2sc(lat0,lon0,az)` converts a great circle from great circle notation (i.e., lat, lon, azimuth, where (lat, lon) is on the circle) to small circle notation (i.e., lat, lon, radius, where (lat, lon) is the center of the circle and the radius is 90 degrees, which is a definition of a great circle). A great circle has two centers and one is chosen arbitrarily. The other is its antipode. All inputs and outputs are in units of degrees.

`[lat,lon,radius] = gc2sc(lat0,lon0,az,angleunits)` uses the string `angleunits` to specify the angle units of the inputs and outputs. `angleunits` can equal either `'degrees'` or `'radians'`.

`mat = gc2sc(...)` returns a single output, where `mat = [lat lon radius]`.

## Definitions

A *small circle* is the intersection of a plane with the surface of a sphere. A *great circle* is a small circle with a radius of 90º.

## Examples

Represent a great circle passing through (25ºS,70ºW) on an azimuth of 45º as a small circle:

```
[lat,lon,radius] = gc2sc(-25,-70,45)

lat =
  -39.8557

lon =
   42.9098

radius =
    90
```

A great circle always bisects the sphere. As a demonstration of this statement, consider the Equator, which passes through any point with a latitude of 0º and proceeds on an azimuth of 90º or 270º. Represent the Equator as a small circle:

```
[lat,lon,radius] = gc2sc(0,-70,270)

lat =

    90

lon =

     0

radius =

    90
```

Not surprisingly, the small circle is centered on the North Pole. As always at the poles, the longitude is arbitrary because of the convergence of the meridians.

Note that the center coordinates returned by this function always lead to one of two possibilities. Since the great circle bisects the sphere, the antipode of the returned point is also a center with a radius of 90º. In the above example, the South Pole would also be a suitable center for the Equator in a small circle.

## See Also
antipode | crossfix | gcxgc | gcxsc | rhxrh

## gcm

Current map projection structure

## Syntax

```
mstruct = gcm
mstruct = gcm(hndl)
```

## Description

`mstruct = gcm` returns the map axes *map structure*, which contains the settings for all the current map axes properties.

`mstruct = gcm(hndl)` specifies the map axes by axes handle.

## Examples

Establish a map axes with default values, then look at the structure:

```
axesm mercator
mstruct = gcm

mstruct =
     mapprojection: 'mercator'
              zone: []
        angleunits: 'degrees'
            aspect: 'normal'
      falsenorthing: 0
       falseeasting: 0
        fixedorient: []
              geoid: [1 0]
        maplatlimit: [-86 86]
        maplonlimit: [-180 180]
       mapparallels: 0
         nparallels: 1
             origin: [0 0 0]
        scalefactor: 1
            trimlat: [-86 86]
```

```
       trimlon: [-180 180]
         frame: 'off'
         ffill: 100
     fedgecolor: [0 0 0]
     ffacecolor: 'none'
      flatlimit: [-86 86]
     flinewidth: 2
      flonlimit: [-180 180]
          grid: 'off'
      galtitude: Inf
         gcolor: [0 0 0]
     glinestyle: ':'
     glinewidth: 0.5000
 mlineexception: []
      mlinefill: 100
     mlinelimit: []
  mlinelocation: 30
   mlinevisible: 'on'
 plineexception: []
      plinefill: 100
     plinelimit: []
  plinelocation: 15
   plinevisible: 'on'
      fontangle: 'normal'
      fontcolor: [0 0 0]
       fontname: 'Helvetica'
       fontsize: 10
      fontunits: 'points'
     fontweight: 'normal'
    labelformat: 'compass'
  labelrotation: 'off'
     labelunits: 'degrees'
  meridianlabel: 'off'
  mlabellocation: 30
  mlabelparallel: 86
     mlabelround: 0
  parallellabel: 'off'
  plabellocation: 15
  plabelmeridian: -180
     plabelround: 0
```

# More About

### Tips

You create map structure properties with the `axesm` function. You can query them with the `getm` function and modify them with the `setm` function.

## See Also

`axesm | getm | setm`

# gcpmap

Current mouse point from map axes

## Syntax

```
pt = gcpmap
pt = gcpmap(hndl)
```

## Description

`pt = gcpmap` returns the current point (the location of last button click) of the current map axes in the form [`latitude longitude z-altitude`].

`pt = gcpmap(hndl)` specifies the map axes in question by its handle.

## Examples

Set up a map axes with a graticule and display a world map:

```
axesm robinson
gridm on
geoshow('landareas.shp')
```

Click somewhere near Boston, Massachusetts to obtain a current point:

```
pt = gcpmap

pt =
      44.171      -69.967           2
      44.171      -69.967           0
whos

  Name      Size                   Bytes  Class         Attributes
  pt        2x3                       48  double array
```
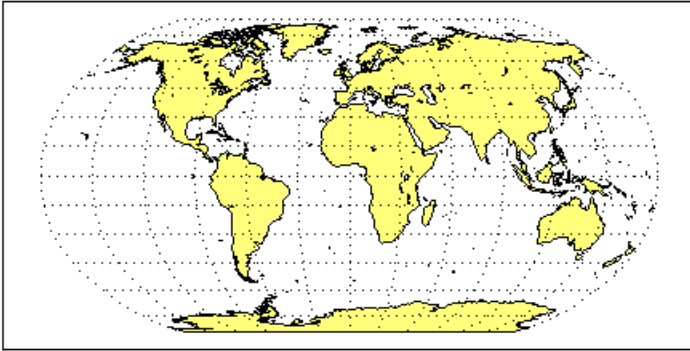
## More About

**Tips**

gcpmap works much like the standard MATLAB function get(gca,'CurrentPoint'), except that the returned matrix is in [lat lon z], not [x y z].

You must use view(2) and an ordinary projection (not the Globe projection) when working with the gcpmap function.

The CurrentPoint property is updated whenever a button-click event occurs in a MATLAB figure window. The pointer does not have to be within the axes, or even the figure window. Coordinates with respect to the requested axes are returned regardless of the pointer location. Likewise, gcpmap will return values that may look reasonable whether the current point is within the graticule bounds or not, and thus must be used with care.

## See Also
inputm

# gcwaypts

Equally spaced waypoints along great circle

## Syntax

```
[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2)
[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2,nlegs)
pts = gcwaypts(lat1,lon1,lat2,lon2...)
```

## Description

`[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2)` returns the coordinates of equally spaced points along a great circle path connecting two endpoints, (`lat1`,`lon1`) and (`lat2`,`lon2`).

`[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2,nlegs)` specifies the number of equal-length track legs to calculate. `nlegs`+1 output points are returned, since a final endpoint is required. The default number of legs is 10.

`pts = gcwaypts(lat1,lon1,lat2,lon2...)` packs the outputs, which are otherwise two-column vectors, into a two-column matrix of the form [`latitude longitude`]. This format for successive waypoints along a navigational track is called *navigational track format* in this guide. See the `navigational track format` reference page in this section for more information.

## Background

This is a navigational function. It assumes that all latitudes and longitudes are in degrees.

In navigational practice, great circle paths are often approximated by rhumb line segments. This is done to come reasonably close to the shortest distance between points without requiring course changes too frequently. The `gcwaypts` function provides an easy means of finding waypoints along a great circle path that can serve as endpoints for rhumb line segments (track legs).

# Examples

Imagine you own a sailing yacht and are planning a voyage from North Point, Barbados (13.33° N,59.62°W), to Brest, France (48.36°N,4.49°W). To divide the track into three equal-length segments,

```
figure('color','w');
ha = axesm('mapproj','mercator',...
    'maplatlim',[10 55],'maplonlim',[-80 10],...
    'MLineLocation',15,'PLineLocation',15);
axis off, gridm on, framem on;
load coast;
hg = geoshow(lat,long,'displaytype','line','color','b');
% Define point locations for Barbados and Brest
barbados = [13.33 -59.62];
brest = [48.36 -4.49];
[l,g] = gcwaypts(barbados(1),barbados(2),brest(1),brest(2),3);
geoshow(l,g,'displaytype','line','color','r',...
    'markeredgecolor','r','markerfacecolor','r','marker','o');
geoshow(barbados(1),barbados(2),'DisplayType','point',...
    'markeredgecolor','k','markerfacecolor','k','marker','o')
geoshow(brest(1),brest(2),'DisplayType','point',...
    'markeredgecolor','k','markerfacecolor','k','marker','o')
```
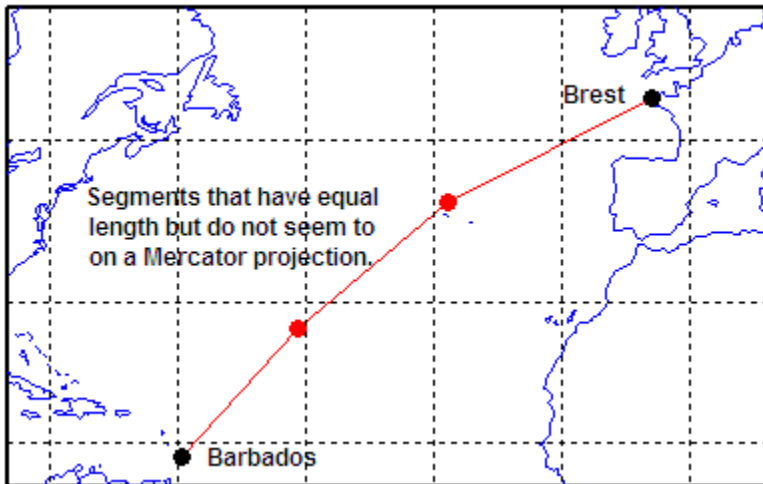
## See Also
dreckon | legs | navfix | track

# gcxgc

Intersection points for pairs of great circles

## Syntax

```
[newlat,newlon] = gcxgc(lat1,lon1,az1,lat2,lon2,az2)
[newlat,newlon] = gcxgc(lat1,lon1,az1,lat2,lon2,az2,units)
```

## Description

`[newlat,newlon] = gcxgc(lat1,lon1,az1,lat2,lon2,az2)` returns the two intersection points of pairs of great circles input in *great circle notation*. When the two great circles are identical (which is not, in general, apparent by inspection), two `NaN`s are returned instead and a warning is displayed. For multiple pairings, the inputs must be column vectors.

`[newlat,newlon] = gcxgc(lat1,lon1,az1,lat2,lon2,az2,units)` specifies the standard angle unit string. The default value is `'degrees'`.

For any pair of great circles, there are two possible intersection conditions: the circles are identical or they intersect exactly twice on the sphere.

*Great circle notation* consists of a point on the great circle and the azimuth at that point along which the great circle proceeds.

## Examples

Given a great circle passing through (10ºN,13ºE) and proceeding on an azimuth of 10º, where does it intersect with a great circle passing through (0º, 20ºE), on an azimuth of -23º (that is, 337º)?

```
[newlat,newlon] = gcxgc(10,13,10,0,20,-23)

newlat =
   14.3105  -14.3105
```

```
newlon =
   13.7838   -166.2162
```

Note that the two intersection points are always antipodes of each other. As a simple example, consider the intersection points of two meridians, which are just great circles with azimuths of 0º or 180º:

```
[newlat,newlon] = gcxgc(10,13,0,0,20,180)

newlat =

   -90     90

newlon =

     0    180
```

The two meridians intersect at the North and South Poles, which is exactly correct.

## See Also
antipode | gc2sc | scxsc | gcxsc | rhxrh | crossfix | polyxpoly

# gcxsc

Intersection points for great and small circle pairs

## Syntax

```
[newlat,newlon] = gcxsc(gclat,gclon,gcaz,sclat,sclon,scrange)
[newlat,newlon] = gcxsc(...,units)
```

## Description

`[newlat,newlon] = gcxsc(gclat,gclon,gcaz,sclat,sclon,scrange)` returns the points of intersection of a great circle in *great circle notation* followed by a small circle in *small circle notation*. For multiple pairings, the inputs must be column vectors. The results are two-column matrices with the coordinates of the intersection points. If the circles do not intersect, or are identical, two `NaN`s are returned and a warning is displayed. If the two circles are tangent, the single intersection point is repeated twice.

`[newlat,newlon] = gcxsc(...,units)` specifies the standard angle unit string. The default value is `'degrees'`.

For a pairing of a great circle with a small circle, there are four possible intersection conditions: the circles are identical (possible because great circles are a subset of small circles), they do not intersect, they are tangent to each other (the small circle interior to the great circle) and hence they intersect once, or they intersect twice.

*Great circle notation* consists of a point on the great circle and the azimuth at that point along which the great circle proceeds.

*Small circle notation* consists of a center point and a radius in units of angular arc length.

## Examples

Given a great circle passing through (43ºN,0º) and proceeding on an azimuth of 10º, where does it intersect with a small circle centered at (47ºN,3ºE) with an arc length radius of 12º?

```
[newlat,newlon] = gcxsc(43,0,10,47,3,12)

newlat =
   35.5068    58.9143

newlon =
   -1.6159     5.4039
```

## See Also

gc2sc | gcxgc | scxsc | rhxrh | crossfix | polyxpoly

# geocentric2geodeticLat

Convert geocentric to geodetic latitude

## Compatibility

`geocentric2geodeticLat` will be removed in a future release. Use `geodeticLatitudeFromGeocentric` instead.

## Syntax

`phiI = geocentric2geodeticlat(ecc, phi_g)`

## Description

`phiI = geocentric2geodeticlat(ecc, phi_g)` converts an array of geocentric latitude in radians, `phi_g`, to geodetic latitude in radians, `phiI`, on a reference ellipsoid with first eccentricity `ecc`.

## See Also

`geodeticLatitudeFromGeocentric`

# geocentricLatitude

Convert geodetic to geocentric latitude

## Syntax

```
psi = geocentricLatitude(phi,F)
psi = geocentricLatitude(phi,F,angleUnit)
```

## Description

`psi = geocentricLatitude(phi,F)` returns the geocentric latitude corresponding to geodetic latitude `phi` on an ellipsoid with flattening `F`.

`psi = geocentricLatitude(phi,F,angleUnit)` specifies the units of input `phi` and output `psi`.

## Examples

### Convert Geodetic Latitude to Geocentric Latitude

Create a reference ellipsoid and then convert the geodetic latitude to geocentric latitude. The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;

geocentricLatitude(45, s.Flattening)

ans =

   44.8076
```

### Convert Geodetic Latitude Expressed in Radians to Geocentric Latitude

Create a reference ellipsoid and then convert a geodetic latitude expressed in radians to geocentric latitude. The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;
```

```
geocentricLatitude(pi/3, s.Flattening, 'radians')

ans =

    1.0443
```

## Input Arguments

### `phi` — Geodetic latitude of one or more points
scalar value, vector, matrix, or N-D array

Geodetic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `F` — Flattening of reference spheroid
scalar

Flattening of reference spheroid, specified as a scalar value.

Data Types: `double`

### `angleUnit` — Unit of measurement for angle
'degrees' (default) | 'radians'

Unit of measurement for angle, specified as the text string 'degrees' or 'radians'.

Data Types: `char`

## Output Arguments

### `psi` — Geocentric latitudes of one or more points
scalar value, vector, matrix, or N-D array

Geocentric latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Values are in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

## See Also

map.geodesy.AuthalicLatitudeConverter | map.geodesy.ConformalLatitudeConverter | map.geodesy.IsometricLatitudeConverter | map.geodesy.RectifyingLatitudeConverter | geodeticLatitudeFromGeocentric | parametricLatitude

# oblateSpheroid class

Oblate ellipsoid of revolution

## Description

An oblate spheroid object encapsulates the interrelated intrinsic properties of an oblate ellipsoid of revolution. An oblate spheroid is symmetric about its polar axis and flattened at the poles, and includes the perfect sphere as a special case.

## Construction

`S = oblateSpheroid` returns an oblate spheroid object.

## Properties

**SemimajorAxis**

Equatorial radius of spheroid, `a`

When set to a new value, the `SeminorAxis` property scales as needed to preserve the shape of the spheroid and the values of shape-related properties including `InverseFlattening` and `Eccentricity`.

The only way to change the `SemimajorAxis` property is to set it directly.

**DataType:** Positive, finite scalar.

**Default:** 1

**SemiminorAxis**

Distance from center of spheroid to pole, `b`

The value is always less than or equal to `SemimajorAxis` property. When set to a new value, the `SemimajorAxis` property remains unchanged, but the shape of the spheroid changes, which is reflected in changes in the values of `InverseFlattening`, `Eccentricity`, and other shape-related properties.

**DataType:** Nonnegative, finite scalar.

**Default:** 0

### InverseFlattening

Reciprocal of flattening.

`1/f = a/(a-b)`, where `a` and `b` are the semimajor and semiminor axes. A value of `1/f = Inf` designates a perfect sphere. As `1/f` value approaches 1, the spheroid approaches a flattened disk. When set to a new value, other shape-related properties are updated, including `Eccentricity`. The `SemimajorAxis` value is unaffected by changes to `1/f`, but the value of the `SemiminorAxis` property adjusts to reflect the new shape.

**DataType:** Positive scalar in the interval `[1 Inf]`.

**Default:** 1

### Eccentricity

First eccentricity of spheroid, `ecc = sqrt(a^2-b^2)/a`

It is the normalized distance from the center to foci in the meridional plane. A value of 0 designates a perfect sphere. When set to a new value, other shape-related properties update, including `InverseFlattening`. The `SemimajorAxis` value is unaffected by changes to `ecc`, but the value of the `SemiminorAxis` property adjusts to reflect the new shape.

**DataType:** Nonnegative scalar less than or equal to 1.

**Default:** 1

### Flattening

Flattening of spheroid

`f = (a-b)/a`, where `a` and `b` are semimajor and semiminor axes of spheroid.

**Access:** Read only

### ThirdFlattening

Third flattening of spheroid

`n = (a-b)/(a+b)`, where `a` and `b` are the semimajor and semiminor axes of spheroid.

**Access:** Read only

### `MeanRadius`

Mean radius of spheroid, `(2*a+b)/3`

The `MeanRadius` property uses the same unit of length as the `SemimajorAxis` and `SemiminorAxis` properties.

**Access:** Read only

### `SurfaceArea`

Surface area of spheroid

The `SurfaceArea` is expressed in units of area consistent with the unit of length used for the `SemimajorAxis` and `SemiminorAxis` properties.

**Access:** Read only

### `Volume`

Volume of spheroid

The `Volume` is expressed in units of volume consistent with the unit of length used for the `SemimajorAxis` and `SemiminorAxis` properties.

**Access:** Read only

---

**Note:** When you define a spheroid in terms of semimajor and semiminor axes (rather than semimajor axis and inverse flattening or semimajor axis and eccentricity), a small loss of precision in the last few digits of `f`, `ecc`, and `n` is possible. This is unavoidable, but does not affect the results of practical computation.

---

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

# Examples

### Create GRS 80 ellipsoid

Create a GRS 80 ellipsoid using the `oblateSpheroid` class.

Start with a unit sphere by default.

```
s = oblateSpheroid

s =

oblateSpheroid with defining properties:

        SemimajorAxis: 1
        SemiminorAxis: 1
     InverseFlattening: Inf
         Eccentricity: O

   and additional properties:

     Flattening
     ThirdFlattening
     MeanRadius
     SurfaceArea
     Volume
```

Reset the semimajor axis to match the equatorial radius of the GRS 80 reference ellipsoid.

```
s.SemimajorAxis = 6378137

s =

oblateSpheroid with defining properties:

        SemimajorAxis: 6378137
        SemiminorAxis: 6378137
     InverseFlattening: Inf
         Eccentricity: O

   and additional properties:

     Flattening
```

```
        ThirdFlattening
        MeanRadius
        SurfaceArea
        Volume
```

The result is a sphere with radius 6,378,137 meters.

Reset the inverse flattening to the standard value for GRS 80, resulting in an oblate spheroid with a semiminor axis consistent with the value, 6,356,752.3141, tabulated in DMA Technical Memorandum 8358.1, "Datums, Ellipsoids, Grids, and Grid Reference Systems."

```
s.InverseFlattening = 298.257222101

s =

oblateSpheroid with defining properties:

        SemimajorAxis: 6378137
        SemiminorAxis: 6356752.31414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

## See Also
referenceEllipsoid | referenceSphere

# geodetic2aer

Geodetic to local spherical AER

## Syntax

```
[az,elev,slantRange] = geodetic2aer(lat,lon,h,lat0,lon0,h0,spheroid)
[ ___ ] = geodetic2aer( ___ ,angleUnit)
```

## Description

`[az,elev,slantRange] = geodetic2aer(lat,lon,h,lat0,lon0,h0,spheroid)` returns coordinates in a local spherical system corresponding to geodetic coordinates `lat`, `lon`, `h`. `spheroid` is an instance of a reference spheroid object. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = geodetic2aer( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat`, `lon`, `lat0`, `lon0`, and outputs `az`, `elev`.

## Examples

### Zermatt to the Matterhorn

Compute the azimuth (in degrees), elevation angle (in degrees), and line of sight distance from Zermatt, Switzerland to the summit of the Matterhorn (Monte Cervino). All distances and lengths are in meters.

Origin (reference point): Zermatt.

```
fmt = get(0,'Format');
format short g

lat0 = dm2degrees([46  1])  % convert degree-minutes to degrees
lon0 = dm2degrees([ 7 45])
hOrthometric0 = 1620;
hGeoid = 53;
h0 = hOrthometric0 + hGeoid
```

```
lat0 =

      46.017


lon0 =

       7.75


h0 =

       1673
```

Point of Interest: Summit of Matterhorn.

```
lat = dms2degrees([45 58 35])
lon = dms2degrees([ 7 39 30])
hOrthometric = 4478;
hGeoid = 53;
h = hOrthometric + hGeoid

lat =

      45.976


lon =

      7.6583


h =

       4531
```

Azimuth, elevation angle, and slant range (line of sight distance) from Zermatt to Matterhorn summit.

```
spheroid = referenceEllipsoid('WGS 84')

[az,elev,slantRange] = geodetic2aer( ...
    lat,lon,h,lat0,lon0,h0,spheroid)

format(fmt)
```

```
spheroid =

referenceEllipsoid with defining properties:

                Code: 7030
                Name: 'World Geodetic System 1984'
          LengthUnit: 'meter'
       SemimajorAxis: 6378137
       SemiminorAxis: 6356752.31424518
    InverseFlattening: 298.257223563
        Eccentricity: 0.0818191908426215

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume

az =

      237.8


elev =

      18.755


slantRange =

      8871.7
```

## Input Arguments

### `lat` — Geodetic latitudes
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon` — Longitudes
scalar value | vector | matrix | N-D array

Longitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h` — Ellipsoidal heights
scalar value | vector | matrix | N-D array

Ellipsoidal heights of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### `az` — Azimuth angles
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the half-open interval [0 360).

### `elev` — Elevation angles
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface normal. Units determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the closed interval [-90 90].

**`slantRange` — Distances from local origin**
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is computed. Units are determined by the `LengthUnit` property of the `spheroid` input.

## See Also
aer2geodetic | ecef2aer | geodetic2enu | geodetic2ned

# geodetic2ecef

Transform geodetic to geocentric (ECEF) coordinates

## Syntax

```
[x,y,z] = geodetic2ecef(spheroid,lat,lon,h)
[x,y,z] = geodetic2ecef( ___ ,angleUnit)

[x,y,z] = geodetic2ecef(lat,lon,h,spheroid)
```

## Description

`[x,y,z]` = `geodetic2ecef(spheroid,lat,lon,h)` transforms geodetic point locations specified by the geodetic coordinate arrays `lat` (geodetic latitude), `lon` (longitude), and `h` (ellipsoidal height) to geocentric Earth-Centered Earth-Fixed (ECEF) Cartesian coordinates `x`, `y`, and `z`. The geodetic coordinates refer to the reference body specified by the spheroid object, `spheroid`.

`[x,y,z]` = `geodetic2ecef( ___ ,angleUnit)` uses the string `angleUnit`, which matches either `'degrees'` or `'radians'`, to specify the units of the latitude and longitude coordinates.

`[x,y,z]` = `geodetic2ecef(lat,lon,h,spheroid)` This syntax is supported but not recommended. `lat` and `lon` must be in radians. `spheroid` can be either a spheroid or an ellipsoid vector of the form: `[semimajor_axis, eccentricity]`. In this case, `H` must use the same units as the semimajor axis. `x`, `y`, and `z` will be expressed in these units also.

## Examples

### Convert Geodetic Coordinates to Geocentric Coordinates

Convert the geodetic coordinates for Paris, France (48 degrees 51 minutes 24 seconds North, 2 degrees 21 minutes 03 seconds East) to geocentric Earth-Centered Earth-Fixed (ECEF) Cartesian coordinates.

```
lat = dms2degrees([48 51 24])
```

```
lon = dms2degrees([2 21 03])
h = 80;
wgs84 = wgs84Ellipsoid('meters');
[x,y,z] = geodetic2ecef(wgs84,lat,lon,h)


lat =

   48.8567


lon =

    2.3508


x =

   4.2010e+06


y =

   1.7246e+05


z =

   4.7801e+06
```

## Input Arguments

### `spheroid` — Reference body
spheroid object

Reference body, specified as a spheroid object: `oblateSpheroid`,
`referenceEllipsoid`, or `referenceSphere`.

### `lat` — Geodetic latitude in degrees
geodetic coordinate array

Geodetic latitude in degrees, specified as geodetic coordinate array.

Data Types: `single` | `double`

### `lon` — Geodetic longitude in degrees
geodetic coordinate array

Geodetic longitude in degrees, specified as a geodetic coordinate array.

Data Types: `single` | `double`

### `h` — Ellipsoidal height
numeric array

Ellipsoidal height, specified as a scalar. `h` must be expressed in the same length unit as the `spheroid`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### `angleUnit` — Units of the latitude and longitude coordinates
`'degrees'` | `'radians'`

Units of the latitude and longitude coordinates, specified as the character string `'degrees'` or `'radians'`.

Data Types: `char`

## Output Arguments

### `x` — Geocentric Earth-Centered Earth-Fixed Cartesian *x* coordinates
numeric array

Geocentric Earth-Centered Earth-Fixed Cartesian *x* coordinates, returned as a numeric array.

### `y` — Geocentric Earth-Centered Earth-Fixed Cartesian *y* coordinates
numeric array

Geocentric Earth-Centered Earth-Fixed Cartesian *y* coordinates, returned as a numeric array.

### `z` — Geocentric Earth-Centered Earth-Fixed Cartesian *z* coordinates
numeric array

Geocentric Earth-Centered Earth-Fixed Cartesian *z* coordinates, returned as a numeric array.

## More About

### Tips

- The geocentric Cartesian (ECEF) coordinate system is fixed with respect to the Earth, with its origin at the center of the spheroid and its positive X-, Y-, and Z axes intersecting the surface at the following points:

|        | Latitude | Longitude | Notes |
|--------|----------|-----------|-------|
| X-axis | 0        | 0         | Equator at the Prime Meridian |
| Y-axis | 0        | 90        | Equator at 90-degrees East |
| Z-zxis | 90       | 0         | North Pole |

## See Also

oblateSpheroid | referenceEllipsoid | referenceSphere | `ecef2geodetic` | `ecef2geodetic` | `ecefOffset`

**Introduced in R2012b**

# geodetic2enu

Geodetic to local Cartesian ENU

## Syntax

```
[xEast,yNorth,zUp] = geodetic2enu(lat,lon,h,lat0,lon0,h0,spheroid)
[xEast,yNorth,zUp] = geodetic2enu( ___ ,angleUnit)
```

## Description

`[xEast,yNorth,zUp] = geodetic2enu(lat,lon,h,lat0,lon0,h0,spheroid)` returns coordinates in a local east-north-up (ENU) Cartesian system corresponding to geodetic coordinates `lat`, `lon`, `h`. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[xEast,yNorth,zUp] = geodetic2enu( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat`, `lon`, `lat0`, and `lon0`.

## Input Arguments

### `lat` — Geodetic latitudes
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon` — Longitudes
scalar value | vector | matrix | N-D array

Longitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h` — Ellipsoidal heights
scalar value | vector | matrix | N-D array

Ellipsoidal heights of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

**`angleUnit` — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

# Output Arguments

**`xEast` — Local ENU x-coordinates**
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

**`yNorth` — Local ENU y-coordinates**
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

**`zUp` — Local ENU z-coordinates**
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local ENU system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

# See Also
`ecef2enu` | `enu2geodetic` | `geodetic2aer` | `geodetic2ned`

# geodetic2ned

Geodetic to local Cartesian NED

## Syntax

```
[xNorth,yEast,zDown] = geodetic2ned(lat,lon,h,lat0,lon0,h0,spheroid)
[ ___ ] = geodetic2ned( ___ ,angleUnit)
```

## Description

`[xNorth,yEast,zDown] = geodetic2ned(lat,lon,h,lat0,lon0,h0,spheroid)` transforms point locations from geodetic coordinates (`lat`, `lon`, `h`) to local, north-east-down (NED), Cartesian coordinates (`xNorth`, `yEast`, `zDown`), given a local coordinate system defined by the geodetic coordinates of its origin (`lat0`, `lon0`, `h0`). The geodetic coordinates refer to the reference body specified by the spheroid object, `spheroid`. Each coordinate input argument must match the others in size or be scalar.

`[ ___ ] = geodetic2ned( ___ ,angleUnit)` specifies the unit of the latitude and longitude input arguments, `lat`, `lon`, `lat0`, and `lon0`.

## Input Arguments

### `lat` — Geodetic latitudes
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon` — Longitudes
scalar value | vector | matrix | N-D array

Longitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### h — Ellipsoidal heights
scalar value | vector | matrix | N-D array

Ellipsoidal heights of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` object.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### h0 — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

**`angleUnit` — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

**`xNorth` — Local NED x-coordinates**
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

**`yEast` — Local NED y-coordinates**
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

**`zDown` — Local NED z-coordinates**
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local NED system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` input.

## See Also
`ecef2ned` | `geodetic2aer` | `geodetic2enu` | `ned2geodetic`

# geodetic2geocentricLat

Convert geodetic to geocentric latitude

## Compatibility

geodetic2geocentricLat will be removed in a future release. Use geocentricLatitude instead.

## Syntax

phi_g = geodetic2geocentriclat(ecc, phi)

## Description

phi_g = geodetic2geocentriclat(ecc, phi) converts an array of geodetic latitude in radians, phi, to geocentric latitude in radians, phi_g, on a reference ellipsoid with first eccentricity ecc.

## See Also
geocentricLatitude

# geodeticLatitudeFromGeocentric

Convert geocentric to geodetic latitude

## Syntax

```
phi = geodeticLatitudeFromGeocentric(psi,F)
phi = geodeticLatitudeFromGeocentric(psi,F,angleUnit)
```

## Description

`phi = geodeticLatitudeFromGeocentric(psi,F)` returns the geodetic latitude
corresponding to geocentric latitude `psi` on an ellipsoid with flattening `F`.

`phi = geodeticLatitudeFromGeocentric(psi,F,angleUnit)` specifies the units
of input `psi` and output `phi`.

## Examples

### Convert Geocentric Latitude to Geodetic Latitude

Create a reference ellipsoid and then convert the geocentric latitude to geodetic latitude.
The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;

geodeticLatitudeFromGeocentric(45, s.Flattening)

ans =

   45.1924
```

### Convert Geocentric Latitude Expressed in Radians to Geodetic Latitude

Create a reference ellipsoid and then convert a geocentric latitude expressed in radians
to geodetic latitude. The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;
```

```
geodeticLatitudeFromGeocentric(pi/3, s.Flattening, 'radians')

ans =

    1.0501
```

# Input Arguments

### `psi` — Geocentric latitude of one or more points
scalar value, vector, matrix, or N-D array

Geocentric latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Values are in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `F` — Flattening of reference spheroid
scalar

Flattening of reference spheroid, specified as a scalar value.

Data Types: `double`

### `angleUnit` — Unit of measurement for angle
'degrees' (default) | 'radians'

Unit of measurement for angle, specified as the text string `'degrees'` or `'radians'`.

Data Types: `char`

# Output Arguments

### `phi` — Geodetic latitude of one or more points
scalar value, vector, matrix, or N-D array

Geodetic latitude of one or more points, returned as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

## See Also

map.geodesy.AuthalicLatitudeConverter | map.geodesy.ConformalLatitudeConverter | map.geodesy.IsometricLatitudeConverter | map.geodesy.RectifyingLatitudeConverter | `geocentricLatitude` | `geodeticLatitudeFromParametric`

# geodeticLatitudeFromParametric

Convert parametric to geodetic latitude

## Syntax

```
phi = geodeticLatitudeFromParametric(beta,F)
phi = geodeticLatitudeFromParametric(beta,F,angleUnit)
```

## Description

`phi = geodeticLatitudeFromParametric(beta,F)` returns the geodetic latitude corresponding to parametric latitude `beta` on an ellipsoid with flattening `F`.

`phi = geodeticLatitudeFromParametric(beta,F,angleUnit)` specifies the units of input `beta` and output `phi`.

## Examples

### Convert Parametric Latitude to Geodetic Latitude

Create a reference ellipsoid and then convert the parametric latitude to geodetic latitude. The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;

geodeticLatitudeFromParametric(45, s.Flattening)

ans =

   45.0962
```

### Convert Parametric Latitude Expressed in Radians to Geodetic Latitude

Create a reference ellipsoid and then convert a parametric latitude expressed in radians to geodetic latitude. The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;
```

```
geodeticLatitudeFromParametric(pi/3, s.Flattening, 'radians')

ans =

    1.0487
```

## Input Arguments

### `beta` — Parametric latitude of one or more points
scalar value, vector, matrix, or N-D array

Parametric latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `F` — Flattening of reference spheroid
scalar

Flattening of reference spheroid, specified as a scalar value.

Data Types: `double`

### `angleUnit` — Unit of measurement for angle
'degrees' (default) | 'radians'

Unit of measurement for angle, specified as the text string 'degrees' or 'radians'.

Data Types: `char`

## Output Arguments

### `phi` — Geodetic latitudes of one or more points
scalar value, vector, matrix, or N-D array

Geodetic latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Values are in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

## See Also

map.geodesy.AuthalicLatitudeConverter | map.geodesy.ConformalLatitudeConverter |
map.geodesy.IsometricLatitudeConverter | map.geodesy.RectifyingLatitudeConverter |
geodeticLatitudeFromGeocentric | parametricLatitude

# geoloc2grid

Convert geolocated data array to regular data grid

## Syntax

```
[Z, refvec] = geoloc2grid(lat, lon, A, cellsize)
```

## Description

`[Z, refvec] = geoloc2grid(lat, lon, A, cellsize)` converts the geolocated data array `A`, given geolocation points in `lat` and `lon`, to produce a regular data grid, `Z`, and the corresponding three-element referencing vector `refvec`. `cellsize` is a scalar that specifies the width and height of data cells in the regular data grid, using the same angular units as `lat` and `lon`. Data cells in `Z` falling outside the area covered by `A` are set to `NaN`.

## Examples

```
% Load the geolocated data array 'map1'
% and grid it to 1/2-degree cells.
load mapmtx
cellsize = 0.5;
[Z, refvec] = geoloc2grid(lt1, lg1, map1, cellsize);

% Create a figure
f = figure;
[cmap,clim] = demcmap(map1);
set(f,'Colormap',cmap,'Color','w')

% Define map limits
latlim = [-35 70];
lonlim = [0 100];

% Display 'map1' as a geolocated data array in subplot 1
subplot(1,2,1)
ax = axesm('mercator','MapLatLimit',latlim,...
    'MapLonLimit',lonlim,'Grid','on',...
```

```
    'MeridianLabel','on','ParallelLabel','on');
set(ax,'Visible','off')
geoshow(lt1, lg1, map1, 'DisplayType', 'texturemap');

% Display 'Z' as a regular data grid in subplot 2
subplot(1,2,2)
ax = axesm('mercator','MapLatLimit',latlim,...
    'MapLonLimit',lonlim,'Grid','on',...
    'MeridianLabel','on','ParallelLabel','on');
set(ax,'Visible','off')
geoshow(Z, refvec, 'DisplayType', 'texturemap');
```
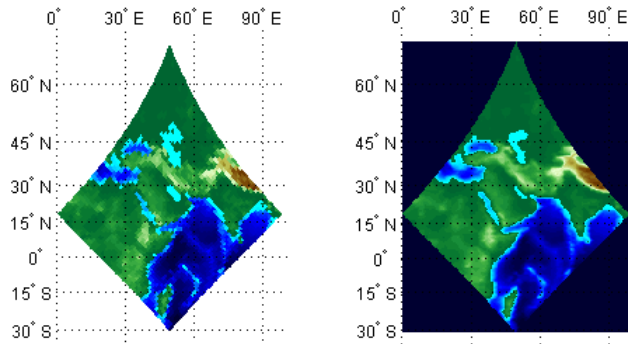


## More About

### Tips

geoloc2grid provides an easy-to-use alternative to gridding geolocated data arrays with imbedm. There is no need to preallocate the output map; there are no data gaps in the output (even if cellsize is chosen to be very small), and the output map is smoother.

# geopeaks

Generate synthetic data set on sphere

## Syntax

```
Z = geopeaks(lat,lon)
Z = geopeaks(R)
Z = geopeaks( ___ ,spheroid)
```

## Description

`Z = geopeaks(lat,lon)` evaluates a "peaks-like" function at specific latitudes and longitudes on the surface of a sphere, returning the synthetic data set `Z`. The function is continuous and smooth at all points, including the poles. Reminiscent of the MATLAB `peaks` function, `geopeaks` undulates gently between values of `-10` and `8`, with about a half dozen local extrema.

`Z = geopeaks(R)` evaluates the `geopeaks` function at cell centers or sample posting points defined by a geographic raster reference object, `R`.

`Z = geopeaks( ___ ,spheroid)` evaluates the function on a specific spheroid. The choice of spheroid makes very little difference; this option exists mainly to support formal testing.

## Examples

### Generate Profile Along Meridian

Define latitude and longitude values along meridian that includes Paris, France.

```
lon = dms2degrees([2 21 3]);
lat = -90:0.5:90;
```

Generate a data set, specifying a spheroid.

```
z = geopeaks(lat, lon, wgs84Ellipsoid);
```

Display the profile.

```
figure
plot(lat,z)
xlabel('Latitude')
```
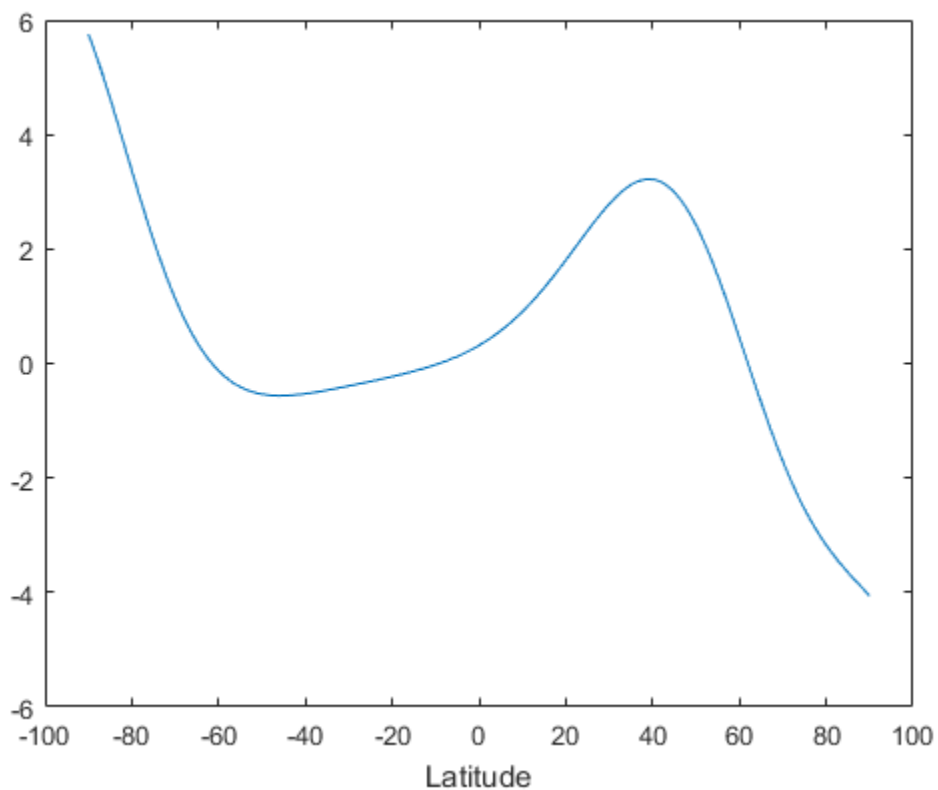


### Generate Global Raster and Display Results on World Map

Create a raster reference object for a 181-by-361 grid of postings.

```
latlim = [-90 90];
```

```
lonlim = [-180 180];
sampleSpacing = 1;
R = georefpostings(latlim,lonlim,sampleSpacing,sampleSpacing)


R =

  GeographicPostingsReference with properties:

                LatitudeLimits: [-90 90]
               LongitudeLimits: [-180 180]
                    RasterSize: [181 361]
         RasterInterpretation: 'postings'
             ColumnsStartFrom: 'south'
                 RowsStartFrom: 'west'
       SampleSpacingInLatitude: 1
      SampleSpacingInLongitude: 1
        RasterExtentInLatitude: 180
       RasterExtentInLongitude: 360
              XIntrinsicLimits: [1 361]
              YIntrinsicLimits: [1 181]
         CoordinateSystemType: 'geographic'
                     AngleUnit: 'degree'
```

Generate a data set.

```
Z = geopeaks(R);
```

Display the resultant data set on a map.

```
figure
worldmap world
geoshow(Z,R,'DisplayType','surface','CData',Z,'ZData',zeros(size(Z)))
load coastlines
geoshow(coastlat,coastlon,'Color','k')
colorbar
```
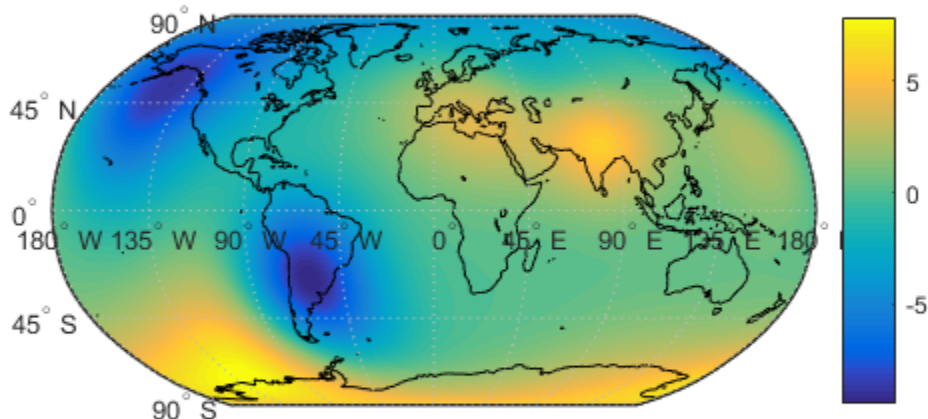
## Input Arguments

**`lat` — Geodetic latitude of one or more points**
scalar, vector, or matrix

Geodetic latitude of one or more points, specified as a scalar value, vector, or matrix. Values must be in degrees.

The `lat` input argument must match the `lon` input argument in size unless either value is scalar (in which case it will expand in size to match the other), or `lat` is a column vector and `lon` is a row vector (they will expand to form a plaid latitude-longitude mesh).

Example: `lat = -90:0.5:90`

Data Types: `single` | `double`

### `lon` — Geodetic longitude of one or more points
scalar, vector, or matrix

Geodetic longitude of one or more points, specified as a scalar value, vector, or matrix. Values must be in degrees.

The `lon` input argument must match the `lat` input argument in size unless either value is scalar (in which case it will expand in size to match the other), or `lon` is a column vector and `lat` is a row vector (they will expand to form a plaid latitude-longitude mesh).

Example: `lon = -180:0.5:180`

Data Types: `single` | `double`

### `R` — Latitude and longitude of points
Geographic raster reference object

Latitude and longitude of points, specified as a geographic raster reference object.

Example: `R = georefcells([0 80], [-140 60], 0.25, 0.25)`

### `spheroid` — Reference spheroid
`referenceEllipsoid` | `oblateSpheroid` | `referenceSphere`

Reference spheroid, specified as a `referenceEllipsoid`, `oblateSpheroid`, or `referenceSphere` object.

Example: `spheroid = referenceEllipsoid('GRS80')`

## Output Arguments

### `Z` — Synthetic data set
scalar value, vector, or matrix

Synthetic data set, returned as a scalar value, vector, or matrix of class `single` or `double`, depending on the class of the input. The function is evaluated at each element of `lat` and `lon` (following expansion as noted above), or at each cell center or posting point defined by `R`.

## See Also
peaks

**Introduced in R2015b**

# geopoint class

Geographic point vector

## Syntax

```
p = geopoint()
p = geopoint(lat,lon)
p = geopoint(lat,lon,Name,Value)
p = geopoint(structArray)
p = geopoint(lat,lon,structArray)
```

## Description

A geopoint vector is a container object that holds geographic point coordinates and attributes. The points are coupled, such that the size of the latitude and longitude coordinate arrays are always equal and match the size of any dynamically added attribute arrays. Each entry of a coordinate pair and associated attributes, if any, represent a discrete element in the geopoint vector.

## Construction

`p = geopoint()` constructs an empty geopoint vector, p, with these default property settings:

```
p =

 0x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: []
    Longitude: []
```

`p = geopoint(lat,lon)` constructs a new geopoint vector and assigns the `Latitude` and `Longitude` properties to the numeric array inputs, `lat` and `lon`. For examples, see "Construct geopoint Vector Using Latitude and Longitude Coordinates" on page 1-377 .

`p = geopoint(lat,lon,Name,Value)` constructs a geopoint vector from input arrays `lat` and `lon`, and then adds dynamic properties to the geopoint vector using the `Name`, `Value` argument pairs.

- If a specified name is `Metadata` and the corresponding value is a scalar structure, then the value is copied to the `Metadata` property. Otherwise, an error is issued.

See "Construct geopoint Vector Using Name-Value Pairs" on page 1-379 for examples.

`p = geopoint(structArray)` constructs a new `geopoint` vector from the fields of the structure, `structArray`.

- If `structArray` contains the field `Lat`, and does not contain a field `Latitude`, then the `Lat` values are assigned to the `Latitude` property.
- If `structArray` contains both `Lat` and `Latitude` fields, then both field values are assigned to `p`.
- If `structArray` contains the field, `Lon`, and does not contain a field, `Longitude`, then the `Lon` values are assigned to the `Longitude` property.
- If `structArray` contains both `Lon` and `Longitude` fields, then both field values are assigned to `p`.
- If `structArray` is a scalar structure containing the field `Metadata` and the field value is a scalar structure, then the `Metadata` field is copied to the `Metadata` property. Otherwise, an error is issued if the `Metadata` field is not a structure, or ignored if `structArray` is not scalar.
- Other fields of `structArray` are assigned to `p` and become dynamic properties. Field values in `structArray` that are not numeric or strings or cell arrays of numeric or string values are ignored.

See "Construct geopoint Vector From a Structure Array" on page 1-380 for an example.

`p = geopoint(lat,lon,structArray)` constructs a new `geopoint` vector and assigns the `Latitude` and `Longitude` properties to the numeric arrays, `lat` and `lon`, and sets dynamic properties from the field values of the structure, `structArray`.

- If `structArray` contains the fields `Lat`, `Latitude`, `Lon` or `Longitude`, then those field values are ignored.

**1-373**

- If `structArray` is a scalar structure containing the field `Metadata`, and the field value is a scalar structure, then it is copied to the `Metadata` property. Otherwise, an error is issued if the `Metadata` field is not a structure, or ignored if `structArray` is not scalar.

See "Construct a geopoint Vector Using Numeric Arrays and a Structure Array" on page 1-381.

## Input Arguments

### `lat` — Vector of latitude coordinates
numeric array

Vector of latitude coordinates, specified as a numeric array.

Data Types: `double` | `single`

### `lon` — Vector of longitude coordinates
numeric array

Vector of longitude coordinates, specified as a numeric array.

Data Types: `double` | `single`

### `structArray` — Dynamic properties
structure

Dynamic properties, specified as a structure.

### `Name` — Name of dynamic property
string

Name of dynamic property, specified as a string.

Data Types: `char`

### `Value` — Value associated with a dynamic property specified in Name
numeric, logical, char, or a cell array of strings

Value associated with a dynamic property, specified as a numeric, logical, char, or a cell array of strings.

## Output Arguments

**p — New object**
geopoint vector

New object, returned as a geopoint vector

# Properties

Each element in a geopoint vector is considered a feature. Feature properties contain one value (a scalar number or a string) for each element in the geopoint vector. The `Latitude` and `Longitude` coordinate properties are feature properties as there is one value for each feature.

`Geometry` and `Metadata` are collection properties. These properties contain only one value per class instance. The term *collection* is used to distinguish these two properties from other feature properties which have values associated with each feature (element in a geopoint vector). See "Metadata and Array Assignment" on page 1-392 for usage examples.

You can attach new dynamic feature properties to the object by using dot '.' notation. This is similar to adding dynamic fields to a structure. Dynamic feature properties apply to each individual feature in the geopoint vector.

**`Geometry` — Type of geometry**
`'point'` (default) | string

Type of geometry, specified as a string. For `geopoint`, `Geometry` is always `'point'`.

**Attributes:**

| Geometry | string |
|---|---|

**`Metadata` — Information for the entire set of geopoint vector elements**
scalar structure

Information for the entire set of geopoint vector elements, specified as a scalar structure. You can add any data type to the structure.

**Attributes:**

```
Metadata                          Scalar struct
```

**Latitude — Latitude coordinates**
numeric row or column vector

Latitude coordinates, specified as a numeric row or column vector.

**Attributes:**

```
Latitude                          single | double vector
```

**Longitude — Longitude coordinates**
numeric row or column vector

Longitude coordinates, specified as a numeric row or column vector.

**Attributes:**

```
Longitude                         single | double vector
```

## Methods

| | |
|---|---|
| append | Append features to geopoint vector |
| cat | Concatenate geopoint vectors |
| disp | Display geopoint vector |
| fieldnames | Dynamic properties of geopoint vector |
| isempty | True if `geopoint` vector is empty |
| isfield | Returns true if dynamic property exists |
| isprop | Returns true if property exists |
| length | Number of elements in geopoint vector |
| properties | Properties of a geopoint vector |
| rmfield | Remove dynamic property from geopoint vector |
| rmprop | Remove properties from geopoint vector |
| size | Size of `geopoint` vector |

| | |
|---|---|
| struct | Convert geopoint vector to scalar structure |
| vertcat | Vertical concatenation for geopoint vectors |

# Copy Semantics

To learn how Value classes affect copy operations, see Copying Objects.

# Class Behaviors

- If the `Latitude`, `Longitude`, or a dynamic property is set with more values than features in the geopoint vector, then all other properties expand in size using 0 for numeric values and empty string for cell values.
- If the `Latitude` or `Longitude` property of the geopoint vector is set with fewer values than contained in the object, then all other properties shrink in size.
- If a dynamic property is set with fewer values than the number of features contained in the object, then this dynamic property expands to match the size of the other properties by inserting a 0 if the value is numeric or an empty string if the value is a cell array.
- If either `Latitude` or `Longitude` is set to [ ], then both coordinate properties are set to [ ] and all dynamic properties are removed.

See "Manipulate a geopoint vector" on page 1-384 for examples of these behaviors.

# Examples

### Construct geopoint Vector Using Latitude and Longitude Coordinates

This example shows how to create a geopoint vector specifying latitude and longitude coordinates. The example then adds a field to the object dynamically.

Construct a geopoint vector from latitude and longitude values.

```
lat = 51.519;
lon = -.13;
```

```
p = geopoint(lat,lon)


p =

 1x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: 51.5190
    Longitude: -0.1300
```

Add a property to the object dynamically.

```
p.Name = 'London'


p =

 1x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: 51.5190
    Longitude: -0.1300
         Name: 'London'
```

### Construct geopoint Vector from File

Load coast line data from a MAT-file.

```
load coastlines
```

Construct a geopoint vector from the coastline latitude and longitude file.

```
p = geopoint(coastlat, coastlon);
```

Display the points on a world map.

```
figure
```

```
worldmap world
geoshow(p.Latitude, p.Longitude)
```
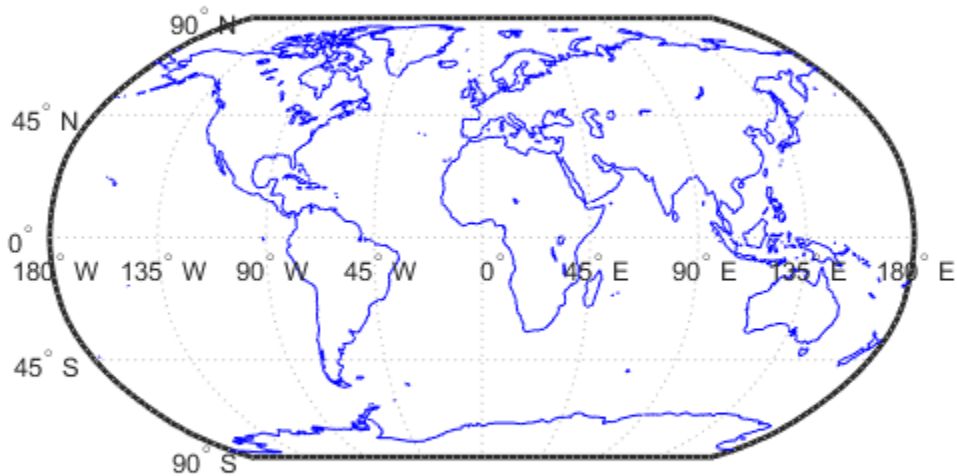


### Construct geopoint Vector Using Name-Value Pairs

Create a geopoint vector, specifying `Latitude`, `Longitude`, and `Temperature`, where `Temperature` is part of a Name-Value pair.

```
 point = geopoint(42, -72, 'Temperature', 89)
```

point =

 1x1 geopoint vector with properties:

```
Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
Feature properties:
      Latitude: 42
     Longitude: -72
   Temperature: 89
```

Construct a geopoint object specifying names.

```
p = geopoint([51.519 48.871], [-.13 2.4131],...
    'Name', {'London', 'Paris'})

p =

 2x1 geopoint vector with properties:

Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
Feature properties:
    Latitude: [51.5190 48.8710]
   Longitude: [-0.1300 2.4131]
        Name: {'London' 'Paris'}
```

### Construct geopoint Vector From a Structure Array

Read shape data into a geostruct (a structure array containing Lat and Lon fields).

```
S = shaperead('worldcities', 'UseGeoCoords', true)

S =

318x1 struct array with fields:

    Geometry
    Lon
    Lat
    Name
```

Create a geopoint vector specifying the geostruct.

```
p = geopoint(S)

p =
```

```
318x1 geopoint vector with properties:

Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
Feature properties:
    Latitude: [1x318 double]
   Longitude: [1x318 double]
        Name: {1x318 cell}
```

Add a `Filename` field to the `Metadata` structure. The `Metadata` property pertains to all elements of a geopoint vector.

```
p.Metadata.Filename = 'worldcities.shp';
m = p.Metadata

ans =

    Filename: 'worldcities.shp'
```

### Construct a geopoint Vector Using Numeric Arrays and a Structure Array

Create a structure array.

```
[structArray, A] = shaperead('worldcities', 'UseGeoCoords', true)

structArray =

318x1 struct array with fields:
    Geometry
    Lon
    Lat

A =

318x1 struct array with fields:
    Name
```

Use the numeric arrays and the structure containing the list of names to construct a geopoint vector.

```
p = geopoint([structArray.Lat], [structArray.Lon], A)

p =
```

```
318x1 geopoint vector with properties:

Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
Feature properties:
    Latitude: [1x318 double]
   Longitude: [1x318 double]
        Name: {1x318 cell}
```

### Add Coordinate and Dynamic properties

Generate an empty geopoint vector using the default constructor and then populate the geopoint vector by adding properties from data fields in structure `structArray` via assignment statements.

```
structArray = shaperead('worldcities', 'UseGeoCoords', true);
p = geopoint();
p.Latitude  = [structArray.Lat];
p.Longitude = [structArray.Lon];
p.Name = structArray.Name;
p

p =

 318x1 geopoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
    Latitude: [1x318 double]
   Longitude: [1x318 double]
        Name: {1x318 cell}
```

### Add New Values to Existing geopoint Vector

This example shows how to add new values to an existing geopoint vector. The example appends data about Paderborn Germany to the geopoint vector of data about world cities.

Read world cities data using the `shaperead` command. `shaperead` returns a structure array.

```
structArray = shaperead('worldcities.shp', 'UseGeoCoords', true);
```

Create a geopoint vector from the structure array. Display the last of the 318 elements in the vector.

```
p = geopoint(structArray);
p(end)


ans =

 1x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: 34.8519
    Longitude: 113.8061
         Name: 'Zhengzhou'
```

Add the Paderborn data to the end of the geopoint vector. Display the last of the existing elements and the new element.

```
lat = 51.715254;
lon = 8.75213;
p = append(p, lat, lon, 'Name', 'Paderborn');
p(end-1:end)


ans =

 2x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [34.8519 51.7153]
    Longitude: [113.8061 8.7521]
         Name: {'Zhengzhou'  'Paderborn'}
```

Another way to add a point at the end of a vector is to use linear indexing. For example, add data about Arlington, Massachusetts to the end of the world cities vector. Notice how, after the initial assignment statement appends a value to the Latitude property

vector, using `end+1` , all other property vectors automatically expand by one element. Display the last of the existing elements and the new element.

```
p(end+1).Latitude = 42.417060
p(end).Longitude = -71.170200;
p(end).Name = 'Arlington';
p(end-1:end)
```

```
p =

 320x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [1x320 double]
    Longitude: [1x320 double]
         Name: {1x320 cell}


ans =

 2x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [51.7153 42.4171]
    Longitude: [8.7521 -71.1702]
         Name: {'Paderborn'  'Arlington'}
```

### Manipulate a geopoint vector

Construct a geopoint vector containing two features and then add two dynamic properties.

```
lat = [51.519 48.871];
lon = [-.13 2.4131];
p = geopoint(lat, lon);

p.Name = {'London', 'Paris'};   % Add character feature dynamic property
```

```
p.ID = [1 2]      % Add numeric feature dynamic property

p =

 2x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [51.5190 48.8710]
    Longitude: [-0.1300 2.4131]
         Name: {'London' 'Paris'}
           ID: [1 2]
```

Add the coordinates for a third feature.

```
p(3).Latitude = 45.472;
p(3).Longitude = 9.184

p =

 3x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [51.5190 48.8710 45.4720]
    Longitude: [-0.1300 2.4131 9.1840]
         Name: {'London' 'Paris' ''}
           ID: [1 2 0]
```

Note that lengths of all feature properties are synchronized with default values.

Set the values for the ID feature dynamic property with more values than contained in Latitude or Longitude.

```
p.ID = 1:4

p =

 4x1 geopoint vector with properties:

 Collection properties:
```

```
     Geometry: 'point'
     Metadata: [1x1 struct]
Feature properties:
     Latitude: [51.5190 48.8710 45.4720 0]
    Longitude: [-0.1300 2.4131 9.1840 0]
         Name: {'London' 'Paris' '' ''}
           ID: [1 2 3 4]
```

Note that all feature properties are expanded to match in size.

Set the values for the ID feature dynamic property with fewer values than contained in the `Latitude` or `Longitude` properties.

```
p.ID = 1:2

p =

 4x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [51.5190 48.8710 45.4720 0]
    Longitude: [-0.1300 2.4131 9.1840 0]
         Name: {'London' 'Paris' '' ''}
           ID: [1 2 0 0]
```

The `ID` property values expand to match the length of the `Latitude` and `Longitude` property values.

Set the value of either coordinate property (`Latitude` or `Longitude`) with fewer values.

```
 p.Latitude = [51.519 48.871]

p =

 2x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [51.5190 48.8710]
    Longitude: [-0.1300 2.4131]
```

```
          Name: {'London' 'Paris'}
            ID: [1 2]
```

All properties shrink to match in size.

Remove the ID property by setting its value to [ ].

```
p.ID = []

p =

 2x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [51.5190 48.8710]
    Longitude: [-0.1300 2.4131]
         Name: {'London' 'Paris'}
```

Remove all dynamic properties and set the object to empty by setting a coordinate property value to [ ].

```
p.Latitude = []

p =

 0x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: []
    Longitude: []
```

### Sort dynamic properties and Extract subsets

Read data from shape file. Initially the field names of the class are in random order.

```
structArray = shaperead('tsunamis', 'UseGeoCoords', true); % Field names in random orde
p = geopoint(structArray)

p =
```

```
162x1 geopoint vector with properties:

Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
Feature properties:
      Latitude: [1x162 double]
     Longitude: [1x162 double]
          Year: [1x162 double]
         Month: [1x162 double]
           Day: [1x162 double]
          Hour: [1x162 double]
        Minute: [1x162 double]
        Second: [1x162 double]
      Val_Code: [1x162 double]
      Validity: {1x162 cell}
     Cause_Code: [1x162 double]
         Cause: {1x162 cell}
        Eq_Mag: [1x162 double]
       Country: {1x162 cell}
      Location: {1x162 cell}
    Max_Height: [1x162 double]
      Iida_Mag: [1x162 double]
     Intensity: [1x162 double]
    Num_Deaths: [1x162 double]
   Desc_Deaths: [1x162 double
```

Using the method `fieldnames` and typical MATLAB vector notation, the field names in the geopoint vector are alphabetically sorted.

```
p = p(:, sort(fieldnames(p)))

p =

 162x1 geopoint vector with properties:

 Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
 Feature properties:
      Latitude: [1x162 double]
     Longitude: [1x162 double]
         Cause: {1x162 cell}
     Cause_Code: [1x162 double]
       Country: {1x162 cell}
```

```
           Day: [1x162 double]
    Desc_Deaths: [1x162 double]
         Eq_Mag: [1x162 double]
           Hour: [1x162 double]
       Iida_Mag: [1x162 double]
      Intensity: [1x162 double]
       Location: {1x162 cell}
     Max_Height: [1x162 double]
         Minute: [1x162 double]
          Month: [1x162 double]
     Num_Deaths: [1x162 double]
         Second: [1x162 double]
       Val_Code: [1x162 double]
       Validity: {1x162 cell}
           Year: [1x162 double]
```

Using typical MATLAB vector notation, extract a subset of data from the base geopoint vector into a geopoint vector albeit smaller in size.

```
subp = p(20:40,{'Location','Country','Year'}) % get subset of data

subp =

 21x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [1x21 double]
    Longitude: [1x21 double]
     Location: {1x21 cell}
      Country: {1x21 cell}
         Year: [1x21 double]
```

Note that the coordinate properties, `Latitude` and `Longitude`, as well as the collection properties are retained in this subset of geopoint vectors.

### Set, Get, and Remove Dynamic Property Values

To set property values, use the `()` operator, or assign array values to corresponding fields, or use dot `'.'` notation (object.property) to assign new property values.

```
pts = geopoint();
pts.Latitude = [42 44 45];
```

**1-389**

```
pts.Longitude = [-72 -72.1 -71];
pts

pts =

 3x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [42 44 45]
    Longitude: [-72 -72.1000 -71]
```

Use ( ) to assign values to fields.

```
pts(3).Latitude = 45.5;
pts

pts =

 3x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [42 44 45.5000]
    Longitude: [-72 -72.1000 -71]
```

Use dot notation to create new dynamic properties

```
pts.Name = {'point1', 'point2', 'point3'}

pts =

 3x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [42 44 45.5000]
    Longitude: [-72 -72.1000 -71]
         Name: {'point1' 'point2' 'point3'}
```

Get property values

```
pts.Name
```

```
ans =

    'point1'    'point2'    'point3'
```

Remove dynamic properties. To delete or remove dynamic properties, set them to `[ ]` or set the `Latitude` or `Longitude` property to `[ ]`.

```
pts.Temperature = 1:3
```

```
pts =

 3x1 geopoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
       Latitude: [42 44 45.5000]
      Longitude: [-72 -72.1000 -71]
           Name: {'point1' 'point2' 'point3'}
    Temperature: [1 2 3]
```

By setting the `Temperature` property to `[ ]`, this dynamic property is deleted.

```
pts.Temperature = []
```

```
pts =

 3x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [42 44 45.5000]
    Longitude: [-72 -72.1000 -71]
         Name: {'point1' 'point2' 'point3'}
```

To clear all fields in the `geopoint` vector, set the `Latitude` or `Longitude` property to `[ ]`

```
pts.Latitude = []

pts =

 0x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: []
    Longitude: []
```

**Metadata and Array Assignment**

Modify a `geopoint` object and its metadata.

If you typically store latitude and longitude coordinates in an `N-by-2` or `2—by-M` array, you can assign these numeric values to a `geopoint` vector. If the coordinates are `N-by-2`, then the first column is assigned to the `Latitude` property and the second column to the `Longitude` property. If the coordinates are `2-by-M`, then the first row is assigned to the `Latitude` property and the second row to the `Longitude` property.

```
coast = load('coast');
ltln = [coast.lat coast.long];  % 9865x2 array
pts = geopoint;   % null constructor
pts(1:numel(coast.lat)) = ltln;  % assign array
pts.Metadata.Name = 'coastline';
pts
pts.Metadata

pts =

 9865x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [1x9865 double]
    Longitude: [1x9865 double]


ans =
```

```
    Name: 'coastline'
```

## See Also
mappoint | geoshape | mapshape | gpxread | shaperead

# append

**Class:** geopoint

Append features to geopoint vector

## Syntax

```
outPoints = append(inPoints,lat,lon)
outPoints = append( ___ ,field,value,...)
```

## Description

`outPoints = append(inPoints,lat,lon)` appends the latitude values in the numeric array, `lat` to the `Latitude` property of the geopoint vector, `inPoints`, and the longitude values in the numeric array, `lon`, to the `Longitude` property of `inPoints`.

`outPoints = append( ___ ,field,value,...)` appends the values specified in `value` to the corresponding dynamic property, `field`. If the property does not exist, `append` adds the dynamic property to the object using the value of `field` for the name and assigning the field the value specified in `value`.

## Input Arguments

**`inPoints`**

geopoint vector.

**`lat`**

Numeric vector of `Latitude` values.

**`lon`**

Numeric vector of `Longitude` values.

**field**

Field name, specified as a string. Field can specify the name of an existing property in the geopoint vector or the name you want assigned to a new property that you want to add to the geopoint vector.

**value**

Value you want to assign to the property specified by `field`, specified as a cell array, scalar, or vector of any numeric class or logical.

- When `value` is a cell array, `append` adds the property as a Vertex property.
- When `value` is a numeric array, `append` adds the property as a Feature property.
- When `value` is empty, `append` removes the property.

## Output Arguments

**outPoints**

Modified geopoint vector with additional entries in `Latitude` and `Longitude` fields along with any new fields for dynamic properties that you added.

## Examples

### Append Values to Fields

Append values to existing fields of a geopoint vector.

```
P = geopoint(42,-110, 'Temperature', 65);
P = append(P, 42.1, -110.4, 'Temperature', 65.5)

P =

 2x1 geopoint vector with properties:

 Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
 Feature properties:
      Latitude: [42 42.1000]
```

```
        Longitude: [-110 -110.4000]
      Temperature: [65 65.5000]
```

### Append Dynamic Property

Append dynamic property, `'Pressure'`, to a geopoint vector.

```
P = geopoint(42,-110, 'Temperature', 65);
P = append(P, 42.2, -110.5, 'Temperature', 65.6, 'Pressure', 100.0)

P =

 2x1 geopoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
       Latitude: [42 42.2000]
      Longitude: [-110 -110.5000]
    Temperature: [65 65.6000]
       Pressure: [0 100]
```

## See Also

geopoint | geopoint.vertcat

# cat

**Class:** geopoint

Concatenate geopoint vectors

# Syntax

```
P= cat(dim,P1, P2, ...)
```

# Description

`P= cat(dim,P1, P2, ...)` concatenates the geopoint vectors `P1`,`P2` and so on along dimensions `dim`. `dim` must be `1`.

# Input Arguments

**P1, P2, ...**

geopoint vectors to be concatenated.

# Output Arguments

**P**

Concatenated geopoint vector.

# Examples

### Concatenate two geopoint vectors

Create two geopoint vectors and concatenate them to a single vector.

```
pt1 = geopoint(42,-110, 'Temperature', 65);
```

```
pt2 = geopoint(42.2, -110.5, 'Temperature', 65.6);
p = cat(1,pt1,pt2)

p =

 2x1 geopoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
       Latitude: [42 42.2000]
      Longitude: [-110 -110.5000]
    Temperature: [65 65.6000]
```

## See Also

geopoint.vertcat

# disp

**Class:** geopoint

Display geopoint vector

## Syntax

```
disp(P)
```

## Description

`disp(P)` prints the size of the geopoint vector, `P`, and its properties and dynamic properties, if they exist. If the command window is large enough, the values of the properties are also shown, otherwise only their size is shown. You can control the display of the numerical values by using the format command.

## Input Arguments

**P**

geopoint vector

## Examples

### Display a geopoint vector

Create a geopoint vector and display it.

```
p = geopoint(shaperead('worldcities', 'UseGeo', true));
disp(p)
disp(p(1:2))

 318x1 geopoint vector with properties:

 Collection properties:
```

```
      Geometry: 'point'
      Metadata: [1x1 struct]
Feature properties:
      Latitude: [1x318 double]
     Longitude: [1x318 double]
          Name: {1x318 cell}

2x1 geopoint vector with properties:

Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
Feature properties:
      Latitude: [5.2985 24.6525]
     Longitude: [-3.9509 54.7589]
          Name: {'Abidjan' 'Abu Dhabi'}
```

## See Also

geopoint | `format`

# fieldnames

**Class:** geopoint

Dynamic properties of geopoint vector

## Syntax

```
names = fieldnames(P)
```

## Description

`names = fieldnames(P)` returns the names of the dynamic properties of the geopoint vector, `P`.

## Input Arguments

**P**

geopoint vector for which the properties are to be queried.

## Output Arguments

**names**

Names of the dynamic properties in the geopoint vector `P`

## Examples

### Find dynamic properties

Create a geopoint vector and view a dynamic property.

```
P = geopoint(shaperead('worldcities', 'UseGeo', true));
```

```
fieldnames(P)

ans =

    'Name'
```

## See Also

geopoint.properties

# isempty

**Class:** geopoint

True if `geopoint` vector is empty

## Syntax

```
TF = isempty(P)
```

## Description

`TF = isempty(P)` returns true if the geopoint vector, `P`, is empty and false otherwise.

## Input Arguments

**P**

geopoint vector.

## Examples

### Check if a geopoint vector is empty

Create a geopoint vector and check if it's empty.

```
P = geopoint();
isempty(P)

ans =

     1
```

# isfield

**Class:** geopoint

Returns true if dynamic property exists

## Syntax

```
TF = isfield(P,name)
TF = isfield(P,names)
```

## Description

`TF = isfield(P,name)` returns true if the value specified by the string `name` is a dynamic property of the geopoint vector, `P`.

`TF = isfield(P,names)` return true for each element of the cell array, `names`, that is a dynamic property of `P`. `TF` is a logical array of the same size as `names`.

## Input Arguments

**P**

geopoint vector

**name**

Name of the dynamic property

**names**

Cell array of names of dynamic properties

## Output Arguments

**TF**

`Boolean`. 1 if `P` contains the specified fields or 0 otherwise.

# Examples

### Check for Field Name

Create a geopoint vector and check field names.

```
p = geopoint(-33.961, 18.484, 'Name', 'Cape Town');
isfield(p, 'Latitude')
isfield(p, 'Name')

ans =

     0


ans =

     1
```

## See Also

geopoint.isprop | geopoint.fieldnames

# isprop

**Class:** geopoint

Returns true if property exists

## Syntax

```
TF = isprop(P,name)
TF = isprop(P,names)
```

## Description

`TF = isprop(P,name)` returns true if the value specified by the string, `name` is a property of the geopoint vector, `P`.

`TF = isprop(P,names)` returns true for each element of the cell array of strings, `names`, that is a property of `P`. `TF` is a logical array the same size as `names`.

## Input Arguments

**P**

geopoint vector.

**name**

String specifying the property of the geopoint vector, `P`.

**names**

Cell array of strings specifying the property of the geopoint vector, `P`.

## Output Arguments

**TF**

Boolean. 1 if the property exists with `P` ,0 otherwise.

# Examples

### Check if property exists

Create a geopoint vector and check if a property exists.

```
p = geopoint(-33.961, 18.484, 'Name', 'Cape Town');
isprop(p, 'Latitude')
isprop(p, 'Name')

ans =

     1


ans =

     1
```

## See Also

geopoint.isfield | geopoint.properties

# length

**Class:** geopoint

Number of elements in geopoint vector

## Syntax

```
N = length(P)
```

## Description

`N = length(P)` returns the number of elements contained in the geopoint vector, `P`. The result is equivalent to `size(P,1)`.

## Input Arguments

**P**

geopoint vector.

## Output Arguments

**N**

Length of the geopoint vector, P.

## Examples

### Find Length of geopoint Vector

Create a geopoint vector and find its length.

```
coast = load('coast');
```

```
p = geopoint(coast.lat, coast.long);
length(p)
length(coast.lat)

ans =

        9865


ans =

        9865
```

## See Also

geopoint.size

# properties

**Class:** geopoint

Properties of a geopoint vector

## Syntax

```
prop = properties(P)
properties(P)
```

## Description

`prop = properties(P)` returns a cell of the property names of the geopoint vector, `P`.

`properties(P)` displays the names of the properties of `P`.

## Input Arguments

**P**

geopoint vector

## Output Arguments

**prop**

Cell variable consisting of property names of the geopoint vector, `P`.

## Examples

### Properties of a geopoint vector

Create a geopoint vector and display a list of properties.

```
p = geopoint(shaperead('tsunamis', 'UseGeo', true));
properties(p)

Properties for class geopoint:

    Geometry
    Metadata
    Latitude
    Longitude
    Year
    Month
    Day
    Hour
    Minute
    Second
    Val_Code
    Validity
    Cause_Code
    Cause
    Eq_Mag
    Country
    Location
    Max_Height
    Iida_Mag
    Intensity
    Num_Deaths
    Desc_Deaths
```

## See Also
geopoint.fieldnames

# rmfield

**Class:** geopoint

Remove dynamic property from geopoint vector

## Syntax

```
P = rmfield(P, fieldname)
P = rmfield(P, fields)
```

## Description

`P = rmfield(P, fieldname)` removes the field specified by the string, `fieldname`, from the geopoint vector, `P`.

`P = rmfield(P, fields)` removes all the fields specified by the cell array, `fields`.

---

**Note:** `rmfield` cannot remove `Latitude`, `Longitude` and `Metadata` fields and the string specified is case sensitive.

---

## Input Arguments

**P**

geopoint vector.

**fieldname**

Exact string representing the name of the property.

**fields**

Cell array of strings specifying the names of the properties.

## Output Arguments

**P**

Updated geopoint vector with the field(s) removed.

## Examples

### Remove fields from a geopoint vector

Create a geopoint vector and remove a field.

```
p = geopoint(shaperead('tsunamis', 'UseGeo', true));
p2 = rmfield(p, 'Geometry')

p2 =

 162x1 geopoint vector with properties:

 Collection properties:
        Geometry: 'point'
        Metadata: [1x1 struct]
 Feature properties:
        Latitude: [1x162 double]
       Longitude: [1x162 double]
            Year: [1x162 double]
           Month: [1x162 double]
             Day: [1x162 double]
            Hour: [1x162 double]
          Minute: [1x162 double]
          Second: [1x162 double]
        Val_Code: [1x162 double]
        Validity: {1x162 cell}
      Cause_Code: [1x162 double]
           Cause: {1x162 cell}
          Eq_Mag: [1x162 double]
         Country: {1x162 cell}
        Location: {1x162 cell}
      Max_Height: [1x162 double]
        Iida_Mag: [1x162 double]
       Intensity: [1x162 double]
       Num_Deaths: [1x162 double]
```

```
Desc_Deaths: [1x162 double]
```

## See Also

geopoint.fieldnames | geopoint.rmprop

# rmprop

**Class:** geopoint

Remove properties from geopoint vector

## Syntax

```
PF = rmprop(P,propname)
PF = rmprop(P,propnames)
```

## Description

`PF = rmprop(P,propname)` removes the property specified by the string, `propname` from the geopoint vector, `P`.

`PF = rmprop(P,propnames)` removes all the properties specified in the cell array, `propnames`, from the geopoint vector, `P`. If `propnames` contains a coordinate property an error is issued.

**Note:** `rmprop` cannot remove `Latitude`, `Longitude` and `Metadata` fields and the string specified is case sensitive.

## Input Arguments

**P**

geopoint vector.

## Output Arguments

**PF**

Modified geopoint vector with the specified property(s) removed.

## Examples

### Remove a property of a geopoint vector

Create a geopoint vector and remove a property.

```
p = geopoint(shaperead('tsunamis', 'UseGeo', true));
p2 = rmprop(p, 'Validity')

p2 =

 162x1 geopoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
       Latitude: [1x162 double]
      Longitude: [1x162 double]
           Year: [1x162 double]
          Month: [1x162 double]
            Day: [1x162 double]
           Hour: [1x162 double]
         Minute: [1x162 double]
         Second: [1x162 double]
       Val_Code: [1x162 double]
     Cause_Code: [1x162 double]
          Cause: {1x162 cell}
         Eq_Mag: [1x162 double]
        Country: {1x162 cell}
       Location: {1x162 cell}
     Max_Height: [1x162 double]
       Iida_Mag: [1x162 double]
      Intensity: [1x162 double]
     Num_Deaths: [1x162 double]
    Desc_Deaths: [1x162 double]
```

## See Also
geopoint.fieldnames

# size

**Class:** geopoint

Size of `geopoint` vector

## Syntax

```
SZ = size(P)
SZ = size(P,1)
SZ = size(P, n)
[m,k] = size(P)
```

## Description

`SZ = size(P)` returns the vector `[length(P), 1]`.

`SZ = size(P,1)` returns the length of `P`.

`SZ = size(P, n)` returns 1 for `n >= 2`.

`[m,k] = size(P)` returns `length(P)` for `m` and 1 for `k`.

## Input Arguments

**P**

geopoint vector.

**n**

Number of the dimension at which size of `P` is required.

## Output Arguments

**SZ**

Vector of the form `[length(P), 1]`.

**m**

Length of P.

**k**

Length of second dimension of P. k is always 1.

## Examples

### Size of a geopoint vector

Create a geopoint vector and find its size.

```
coast = load('coast');
p = geopoint(coast.lat, coast.long);
size(p)

ans =

      9865            1
```

The second dimension is always 1.

## See Also

geopoint.length | size

# struct

**Class:** geopoint

Convert geopoint vector to scalar structure

# Syntax

```
S = struct(P)
```

# Description

`S = struct(P)` converts the geopoint vector, `P`, to a scalar structure, `S`.

# Input Arguments

**P**

geopoint vector

# Output Arguments

**S**

Scalar structure of the geopoint vector `P`.

# Examples

### Converting a geopoint vector into struct

Create a geopoint vector and convert it into a structure.

```
S = shaperead('worldcities', 'UseGeo', true);
p = geopoint(S)
```

```
S2 = struct(p)
class(S2)

p =

 318x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
     Latitude: [1x318 double]
    Longitude: [1x318 double]
         Name: {1x318 cell}


S2 =

     Geometry: 'point'
     Metadata: [1x1 struct]
     Latitude: [1x318 double]
    Longitude: [1x318 double]
         Name: {1x318 cell}


ans =

struct
```

## See Also
geopoint.properties

# vertcat

**Class:** geopoint

Vertical concatenation for geopoint vectors

## Syntax

```
P = vertcat(P1,P2, ...)
```

## Description

`P = vertcat(P1,P2, ...)` vertically concatenates the geopoint vector, `P1`, `P2`, and so on. If the class type of any property is a cell array, then the resultant field in the output `P` will also be a cell array.

## Input Arguments

**P1, P2, ...**

geopoint vectors that need to be concatenated.

## Output Arguments

**P**

Concatenated geopoint vector.

## Examples

### Concatenate geopoint vectors

Create several geopoint vectors and concatenate them.

```
pt1 = geopoint(42, -110, 'Temperature', 65, 'Name', 'point1');
pt2 = geopoint( 42.1, -110.4, 'Temperature', 65.5, 'Name', 'point2');
pts = vertcat(pt1, pt2)

pts =

 2x1 geopoint vector with properties:

 Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
 Feature properties:
      Latitude: [42 42.1000]
     Longitude: [-110 -110.4000]
   Temperature: [65 65.5000]
          Name: {'point1' 'point2'}
```

## See Also

geopoint.cat

# geoquadline

Geographic quadrangle bounding multi-part line

## Syntax

```
[latlim,lonlim] = geoquadline(lat,lon)
```

## Description

`[latlim,lonlim] = geoquadline(lat,lon)` returns the limits of the tightest possible geographic quadrangle that bounds a line connecting vertices with geographic coordinates specified by `lat` and `lon`.

## Examples

### Bounding Quadrangle for the Brahmaputra River.

Read shape data and then create a bounding box around the line.

```
brahmaputra = shaperead('worldrivers.shp','Selector',{@(name) strcmp(name,'Brahmaputra
[latlim, lonlim] = geoquadline(brahmaputra.Lat, brahmaputra.Lon)

latlim =

   23.8285   30.3831


lonlim =

   81.8971   95.4970
```

## Input Arguments

**`lat`** — Latitudes along a line
vector

Latitudes along a line, specified as a vector representing an ordered sequences of vertices, in units of degrees. The line may be broken into multiple parts, delimited by values of **NaN**.

Data Types: `single` | `double`

### `lon` — Longitudes along a line
vector

Longitudes along a line, specified as a vector representing an ordered sequences of vertices, in units of degrees. The line may be broken into multiple parts, delimited by values of **NaN**.

Data Types: `single` | `double`

## Output Arguments

### `latlim` — Latitude limits
1–by-2 vector

Latitude limits of a geographic quadrangle, returned as a 1-by-2 vector of the form `[southern_limit northern_limit]`, in units of degrees. The elements are in ascending order, and both lie in the closed interval [-90 90].

### `lonlim` — Longitude limits
1–by-2 vector

Longitude limits of a geographic quadrangle, returned as a 1-by-2 vector of the form `[western_limit eastern_limit]`, in units of degrees. The limits are wrapped to the interval [-180 180]. They are not necessarily in numerical ascending order.

## See Also
bufgeoquad | geoquadpt | ingeoquad | outlinegeoquad

# geoquadpt

Geographic quadrangle bounding scattered points

## Syntax

```
[latlim,lonlim] = geoquadpt(lat,lon)
```

## Description

`[latlim,lonlim] = geoquadpt(lat,lon)` returns the limits of the tightest possible geographic quadrangle that bounds a set of points with geographic coordinates `lat` and `lon`.

In most cases, `tf = ingeoquad(lat,lon,latlim,lonlim)` will return true, but `tf` may be false for points on the edges of the quadrangle, due to round off. `tf` will also be false for elements of `lat` that fall outside the interval [-90 90] and elements of `lon` that are not finite.

## Examples

### Bounding Quadrangle Including Tokyo and Honolulu.

In this case the output quadrangle straddles the 180-degree meridian, hence the elements of `lonlim` are in descending numerical order, although they are ordered from west to east.

Read a set of points and then create a bounding box around the points.

```
cities = shaperead('worldcities.shp','Selector',{@(name) any(strcmp(name,{'Tokyo','Hond
[latlim,lonlim] = geoquadpt([cities.Lat],[cities.Lon])

latlim =

  21.3178   35.7082
```

```
lonlim =

  139.6401 -157.8291
```

## Input Arguments

### `lat` — Point latitudes
vector | matrix | N-D array

Point latitudes, specified as a vector, matrix, or N-D array, in units of degrees.

Data Types: `single` | `double`

### `lon` — Point longitudes
vector | matrix | N-D array

Point longitudes, specified as a vector, matrix, or N-D array, in units of degrees.

Data Types: `single` | `double`

## Output Arguments

### `latlim` — Latitude limits
1–by-2 vector

Latitude limits of a geographic quadrangle, returned as a 1-by-2 vector of the form `[southern_limit northern_limit]`, in units of degrees. The elements are in ascending order, and both lie in the closed interval [-90 90].

### `lonlim` — Longitude limits
1–by-2 vector

Longitude limits of a geographic quadrangle, returned as a 1-by-2 vector of the form `[western_limit eastern_limit]`, in units of degrees. The limits are wrapped to the interval [-180 180]. They are not necessarily in numerical ascending order.

## See Also
`bufgeoquad` | `geoquadline` | `ingeoquad` | `outlinegeoquad`

# georasterref

Construct geographic raster reference object

## Compatibility

Use the `georefcells` function or the `georefpostings` function instead, except when constructing a raster reference object from world file input.

## Syntax

```
R = georasterref()
R = georasterref(Name,Value)
R = georasterref(W,rasterSize,rasterInterpretation)
```

## Description

`R = georasterref()` constructs a geographic raster reference object with default property values.

`R = georasterref(Name,Value)` accepts a list of name-value pairs that are used to assign selected properties when initializing a geographic raster reference object.

`R = georasterref(W,rasterSize,rasterInterpretation)` constructs a geographic raster reference object with the specified raster size and interpretation properties, and with remaining properties defined by the world file matrix W. The `rasterInterpretation` input is optional, can equal either `'cells'` or `'postings'`, and has a default value of `'cells'`.

## Input Arguments

**W**

2-by-3 world file matrix

**rasterSize**

Two-element vector [M N] specifying the number of rows (*M*) and columns (*N*) of the raster or image associated with the referencing object. For convenience, you may assign a size vector having more than two elements to `RasterSize`. This flexibility enables assignments like `R.RasterSize = size(RGB)`, for example, where `RGB` is *M*-by-*N*-by-3. However, in such cases, only the first two elements of the size vector are actually stored. The higher (non-spatial) dimensions are ignored.

**rasterInterpretation**

Controls handling of raster edges. The `rasterInterpretation` input is optional, and can equal either `'cells'` or `'postings'`.

**Default:** `'cells'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

You can include any of the following properties, overriding their default values as needed. Alternatively, you may omit any or all properties when constructing your geographic raster reference object. Then, you can customize the result by resetting properties from this list one at a time. The exception is the RasterInterpretation property. To have a raster interpretation of `'postings'` (rather than the default, `'cells'`), the name-value pair `'RasterInterpretation','postings'` must be specified in your call to `georasterref`.

**'LatitudeLimits'**

Limits in latitude of the geographic quadrangle bounding the georeferenced raster. A two-element vector of the form:

[southern_limit northern_limit]

**Default:** [0.5 2.5]

**'LongitudeLimits'**

Limits in longitude of the geographic quadrangle bounding the georeferenced raster. A two-element vector of the form:

[western_limit eastern_limit]

**Default:** [0.5 2.5]

**'RasterSize'**

Two-element vector [M N] specifying the number of rows (*M*) and columns (*N*) of the raster or image associated with the referencing object. For convenience, you may assign a size vector having more than two elements to RasterSize. This flexibility enables assignments like R.RasterSize = size(RGB), for example, where RGB is *M*-by-*N*-by-3. However, in such cases, only the first two elements of the size vector are actually stored. The higher (non-spatial) dimensions are ignored.

**Default:** [2 2]

**'RasterInterpretation'**

Controls handling of raster edges. A string that equals either 'cells' or 'postings'. If you want this property to have other than the default value, you must set it when you create the object. Once created, you cannot change the value of this property in a geographic raster reference object.

**Default:** 'cells'

**'ColumnsStartFrom'**

Edge from which column indexing starts. A string that equals either 'south' or 'north'.

**Default:** 'south'

**'RowsStartFrom'**

Edge from which row indexing starts. A string that equals either 'west' or 'east'.

**Default:** 'west'

## Output Arguments

**R**

Geographic raster reference object

## Examples

Construct a referencing object for a global raster comprising a grid of 180-by-360 one-degree cells, with rows that start at longitude -180, and with the first cell located in the northwest corner.

```
R = georasterref('RasterSize', [180 360], ...
      'RasterInterpretation', 'cells', 'ColumnsStartFrom', 'north', ...
      'LatitudeLimits', [-90 90], 'LongitudeLimits', [-180 180])
```

Construct a referencing object for the DTED Level 0 file that includes Sagarmatha (Mount Everest). The DTED columns run from south to north and the first column runs along the western edge of the (one-degree-by-one-degree) quadrangle, consistent with the default values for 'ColumnsStartFrom' and 'RowsStartFrom'.

```
R = georasterref('LatitudeLimits', [27 28], 'LongitudeLimits', [86 87], ...
        'RasterSize', [121 121], 'RasterInterpretation', 'postings')
```

Repeat the second example with a different strategy: Create an object by specifying only the RasterInterpretation value, then modify the object by resetting additional properties. (As noted above, the RasterInterpretation of an existing raster reference object cannot be changed.)

```
R = georasterref('RasterInterpretation','postings');
R.RasterSize = [121 121];
R.LatitudeLimits  = [27 28];
R.LongitudeLimits = [86 87];
```

Repeat the first example using a world file matrix as input.

```
W = [1    0   -179.5; ...
     0   -1      89.5];
rasterSize = [180 360];
rasterInterpretation = 'cells';
R = georasterref(W, rasterSize, rasterInterpretation);
```

## See Also

map.rasterref.GeographicCellsReference | map.rasterref.GeographicPostingsReference | maprasterref

# georefpostings

Reference raster postings to geographic coordinates

## Syntax

```
R = georefpostings()
R = georefpostings(latlim,lonlim,rasterSize)
R = georefpostings(latlim,lonlim,sampleSpacingInLatitude,
sampleSpacingInLongitude)
R = georefpostings(latlim,lonlim,___,Name,Value)
```

## Description

`R = georefpostings()` returns a default referencing object for a raster of regularly posted samples in geographic coordinates.

`R = georefpostings(latlim,lonlim,rasterSize)` constructs a referencing object for a raster spanning the specified limits in latitude and longitude, with the numbers of rows and columns specified by `rasterSize`.

`R = georefpostings(latlim,lonlim,sampleSpacingInLatitude, sampleSpacingInLongitude)` allows the geographic sample spacings to be set precisely. The geographic limits will be adjusted slightly, if necessary, to ensure an integer number of samples in each dimension.

`R = georefpostings(latlim,lonlim,___,Name,Value)` allows the directions of the columns and rows to be specified via name-value pairs.

## Examples

### Construct Geographic Referencing Object for Global Raster

Define latitude and longitude limits and the dimensions of the raster.

```
latlim = [-90 90];
lonlim = [-180 180];
```

```
rasterSize = [181 361];
```

Create the referencing object specifying the raster size.

```
R = georefpostings(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')
```

```
R =

  GeographicPostingsReference with properties:

              LatitudeLimits: [-90 90]
             LongitudeLimits: [-180 180]
                  RasterSize: [181 361]
       RasterInterpretation: 'postings'
            ColumnsStartFrom: 'north'
               RowsStartFrom: 'west'
     SampleSpacingInLatitude: 1
    SampleSpacingInLongitude: 1
      RasterExtentInLatitude: 180
     RasterExtentInLongitude: 360
             XIntrinsicLimits: [1 361]
             YIntrinsicLimits: [1 181]
         CoordinateSystemType: 'geographic'
                   AngleUnit: 'degree'
```

Obtain the same result by specifying the sample spacing.

```
spacing = 1;
```

```
R = georefpostings(latlim,lonlim,spacing,spacing,'ColumnsStartFrom','north')
```

```
R =

  GeographicPostingsReference with properties:

              LatitudeLimits: [-90 90]
             LongitudeLimits: [-180 180]
                  RasterSize: [181 361]
       RasterInterpretation: 'postings'
            ColumnsStartFrom: 'north'
               RowsStartFrom: 'west'
```

```
     SampleSpacingInLatitude: 1
    SampleSpacingInLongitude: 1
      RasterExtentInLatitude: 180
     RasterExtentInLongitude: 360
             XIntrinsicLimits: [1 361]
             YIntrinsicLimits: [1 181]
         CoordinateSystemType: 'geographic'
                    AngleUnit: 'degree'
```

## Input Arguments

### `latlim` — Latitude limits in degrees
[0.5 2.5] (default) | 1-by-2 numeric vector

Latitude limits in degrees, specified as a 1-by-2 numeric vector. The number of rows in the resulting raster is specified by `rasterSize`.

Example: `latlim = [-90 90];`

Data Types: `double`

### `lonlim` — Longitude limits in degrees
[0.5 2.5] (default) | 1-by-2 numeric vector

Longitude limits in degrees, specified as a 1-by-2 numeric vector. The number of columns in the resulting raster is specified by `rasterSize`.

Example: `lonlim = [-180 180];`

Data Types: `double`

### `rasterSize` — Size of the raster
[2 2] (default) | 1-by-2 numeric vector

Size of the raster, specified as a 1-by-2 numeric vector.

Example: `rasterSize = [180 360];`

Data Types: `double`

### `sampleSpacingInLatitude` — Vertical spacing of posting
1 (default) | numeric scalar

Vertical spacing of posting, specified as a numeric scalar.

Example: spacing = 1.5

Data Types: double

**sampleSpacingInLongitude** — **Horizontal spacing of postings**
1 (default) | numeric scalar

Horizontal spacing of postings, specified as a numeric scalar.

Example: spacing = 1.5

Data Types: double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: R = georefpostings(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')

**'ColumnsStartFrom'** — **Edge from which column indexing starts**
'south' (default) | 'north'

Edge from which column indexing starts, specified as the string 'north' or 'south'.

Example:  R = georefpostings(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')

Data Types: char

**'RowsStartFrom'** — **Edge from which row indexing starts**
'west' (default) | 'east'

Edge from which column indexing starts, specified as the string 'east' or 'west'.

Example:  R = georefpostings(latlim,lonlim,rasterSize,'RowsStartFrom','east')

Data Types: char

## Output Arguments

**R — Object that references raster postings to geographic coordinates**
Raster reference object

Object that references raster postings to geographic coordinates, returned as a raster reference object (map.rasterref.GeographicPostingsReference).

## More About

**Tips**

- To construct a geographic raster reference object from a world file matrix, use the `georasterref` function.

## See Also

map.rasterref.GeographicPostingsReference | `georefcells` | `maprefpostings`

**Introduced in R2015b**

# georefcells

Reference raster cells to geographic coordinates

## Syntax

```
R = georefcells()
R = georefcells(latlim,lonlim,rasterSize)
R = georefcells(latlim,lonlim,cellExtentInLatitude,
cellExtentInLongitude)
R = georefcells(latlim,lonlim,___,Name,Value)
```

## Description

`R = georefcells()` returns a default referencing object for a regular raster of cells in geographic coordinates.

`R = georefcells(latlim,lonlim,rasterSize)` constructs a referencing object for a raster of cells spanning the specified limits in latitude and longitude, with the numbers of rows and columns specified by rasterSize.

`R = georefcells(latlim,lonlim,cellExtentInLatitude, cellExtentInLongitude)` allows the geographic cell extents to be set precisely. If necessary, `georefcells` adjusts the geographic limits slightly to ensure an integer number of cells in each dimension.

`R = georefcells(latlim,lonlim,___,Name,Value)` allows the directions of the columns and rows to be specified via name-value pairs.

## Examples

### Construct Referencing Object for Global Raster

Construct a referencing object for a global raster comprising a grid of 180-by-360 one-degree cells, with rows that start at longitude -180, and with the first cell located in the northwest corner.

```
latlim = [-90 90];
lonlim = [-180 180];
rasterSize = [180 360];
```

Create a raster referencing object by specifying the raster size.

```
R = georefcells(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')
```

```
R =

  GeographicCellsReference with properties:

             LatitudeLimits: [-90 90]
            LongitudeLimits: [-180 180]
                 RasterSize: [180 360]
       RasterInterpretation: 'cells'
           ColumnsStartFrom: 'north'
              RowsStartFrom: 'west'
        CellExtentInLatitude: 1
       CellExtentInLongitude: 1
      RasterExtentInLatitude: 180
     RasterExtentInLongitude: 360
            XIntrinsicLimits: [0.5 360.5]
            YIntrinsicLimits: [0.5 180.5]
        CoordinateSystemType: 'geographic'
                   AngleUnit: 'degree'
```

Obtain the same result by specifying cell extents.

```
extent = 1;
```

```
R = georefcells(latlim,lonlim,extent,extent,'ColumnsStartFrom','north')
```

```
R =

  GeographicCellsReference with properties:

             LatitudeLimits: [-90 90]
            LongitudeLimits: [-180 180]
                 RasterSize: [180 360]
       RasterInterpretation: 'cells'
```

```
      ColumnsStartFrom: 'north'
         RowsStartFrom: 'west'
     CellExtentInLatitude: 1
    CellExtentInLongitude: 1
   RasterExtentInLatitude: 180
  RasterExtentInLongitude: 360
         XIntrinsicLimits: [0.5 360.5]
         YIntrinsicLimits: [0.5 180.5]
     CoordinateSystemType: 'geographic'
                AngleUnit: 'degree'
```

# Input Arguments

### `latlim` — Latitude limits in degrees
[0.5 2.5] (default) | 1-by-2 numeric vector

Latitude limits in degrees, specified as a 1-by-2 numeric vector. The number of rows in the resulting raster is specified by `rasterSize`.

Example: `latlim = [-90 90];`

Data Types: `double`

### `lonlim` — Longitude limits in degrees
[0.5 2.5] (default) | 1-by-2 numeric vector

Longitude limits in degrees, specified as a 1-by-2 numeric vector. The number of columns in the resulting raster is specified by `rasterSize`.

Example: `lonlim = [-180 180];`

Data Types: `double`

### `rasterSize` — Size of the raster
[2 2] (default) | 1-by-2 numeric vector

Size of the raster, specified as a 1-by-2 numeric vector.

Example: `rasterSize = [180 360];`

Data Types: `double`

### `cellExtentInLatitude` — Height of cells
1 (default) | numeric scalar

Height of cells, specified as a numeric scalar.

Example: `extent = 1.5`

Data Types: `double`

### `cellExtentInLongitude` — Width of cells
1 (default) | numeric scalar

Width of cells, specified as a numeric scalar.

Example: `extent = 1.5`

Data Types: `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `R = georefcells(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')`

### `'ColumnsStartFrom'` — Edge from which column indexing starts
`'south'` (default) | `'north'`

Edge from which column indexing starts, specified as the string `'north'` or `'south'`.

Example: `R = georefcells(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')`

Data Types: `char`

### `'RowsStartFrom'` — Edge from which row indexing starts
`'west'` (default) | `'east'`

Edge from which column indexing starts, specified as the string `'east'` or `'west'`.

Example: `R = georefcells(latlim,lonlim,rasterSize,'RowsStartFrom','east')`

Data Types: `char`

## Output Arguments

### R — Object that references raster cells to geographic coordinates
Raster reference object

Object that references raster cells to geographic coordinates, returned as a raster reference object (map.rasterref.GeographicCellsReference).

## More About

### Tips

- To construct a geographic raster reference object from a world file matrix, use the `georasterref` function.

## See Also

map.rasterref.GeographicCellsReference | `georefpostings` | `maprefcells`

**Introduced in R2015b**

# map.rasterref.GeographicCellsReference class

**Package:** map.rasterref

Reference raster cells to geographic coordinates

## Description

A geographic cells raster reference object encapsulates the relationship between a geographic coordinate system and a system of intrinsic coordinates anchored to the columns and rows of a 2-D spatially referenced raster grid or image. The raster must be sampled regularly in latitude and longitude, and its columns and rows must be aligned with meridians and parallels, respectively. For more information about coordinate systems, see "Intrinsic Coordinate System" on page 1-447.

## Construction

You can use any of the following functions to construct a `map.rasterref.CellsReference` object to reference a regular raster of cells to geographic coordinates.

- `georefcells`—Create a geographic raster reference object.
- `georasterref`—Convert a world file to a geographic raster reference object.
- `refmatToGeoRasterReference`—Convert a referencing matrix to a geographic raster reference object.
- `refvecTogeoRasterReference`—Convert a referencing vector to a geographic raster reference object.

For example, to construct a geographic raster reference object with default property settings, use this command:

```
R = georefcells()

R =

  GeographicCellsReference with properties:

          LatitudeLimits: [0.5 2.5]
         LongitudeLimits: [0.5 2.5]
```

```
                  RasterSize: [2 2]
       RasterInterpretation: 'cells'
           ColumnsStartFrom: 'south'
              RowsStartFrom: 'west'
       CellExtentInLatitude: 1
      CellExtentInLongitude: 1
     RasterExtentInLatitude: 2
    RasterExtentInLongitude: 2
            XIntrinsicLimits: [0.5 2.5]
            YIntrinsicLimits: [0.5 2.5]
        CoordinateSystemType: 'geographic'
                   AngleUnit: 'degree'
```

# Properties

### `LatitudeLimits` — Latitude limits of the geographic quadrangle bounding the georeferenced raster
`[0.5 2.5]` (default) | two-element vector

Latitude limits of the geographic quadrangle bounding the georeferenced raster, specified as a two-element vector of the form `[southern_limit northern_limit]`.

Example: `[-90 90]`

Data Types: `double`

### `LongitudeLimits` — Longitude limits of the geographic quadrangle bounding the georeferenced raster
`[0.5 2.5]` (default) | two-element vector

Longitude limits of the geographic quadrangle bounding the georeferenced raster, specified as a two-element vector of the form `[western_limit eastern_limit]`.

Example: `[-100 180]`

Data Types: `double`

### `RasterSize` — Number of rows and columns of the raster or image associated with the referencing object
`[2 2]` (default) | two-element vector of positive integers

Number of rows and columns of the raster or image associated with the referencing object, specified as a two-element vector, `[M N]`, where *M* represents the number of rows

and *N* the number of columns. For convenience, you can assign a size vector having more than two elements. This enables assignments like `R.RasterSize = size(RGB)`, where `RGB` is *M*-by-*N*-by-3. In cases like this, the object stores only the first two elements of the size vector and ignores the higher (nonspatial) dimensions.

Example: `[200 300]`

Data Types: `double`

### `RasterInterpretation` — Geometric nature of the raster
`'cells'` (default)

Geometric nature of the raster, specified as the string `'cells'`. The value `'cells'` indicates that the raster comprises a grid of quadrangular cells, and is bounded on all sides by cell edges. For an *M*-by-*N* raster, points with an intrinsic *X*-coordinate of 1 or *N* or an intrinsic *Y*-coordinate of 1 or *M* fall within the raster, not on its edges.

Cannot be set.

Data Types: `char`

### `AngleUnit` — Unit of measurement used for angle-valued properties
`'degree'` (default)

Unit of measurement used for angle-valued properties, specified as the string `'degree'`.

Cannot be set.

Data Types: `char`

### `ColumnsStartFrom` — Edge from which column indexing starts
`'south'` (default) | `'north'`

Edge from which column indexing starts, specified as the string `'south'` or `'north'`.

Example: `'south'`

Data Types: `char`

### `RowsStartFrom` — Edge from which row indexing starts
`'west'` (default) | `'east'`

Edge from which row indexing starts, specified as the string `'west'` or `'east'`.

Example: `'east'`

Data Types: char

### `CellExtentInLatitude` — Extent in latitude of individual cells
1 (default) | positive numeric scalar

Extent in latitude of individual cells, specified as a positive numeric scalar. Distance, in units of latitude, between the northern and southern limits of a single raster cell. The value is the same for all cells in the raster.

Example: `2.5`

Data Types: double

### `CellExtentInLongitude` — Extent in longitude of individual cells
1 (default) | positive numeric scalar

Extent in longitude of individual cells, specified as a positive numeric scalar. Distance, in units of longitude, between the western and eastern limits of a single raster cell. The value is always positive, and is the same for all cells in the raster.

Example: `2.5`

Data Types: double

### `RasterExtentInLatitude` — Latitude extent ("height") of the quadrangle covered by the raster
2 (default) | positive numeric scalar

Latitude extent ("height") of the quadrangle covered by the raster, specified as a positive numeric scalar.

Cannot be set.

Example: `2`

Data Types: double

### `RasterExtentInLongitude` — Longitude extent ("width") of the quadrangle covered by the raster
2 (default) | positive numeric scalar

Longitude extent ("width") of the quadrangle covered by the raster, specified as a positive numeric scalar.

Cannot be set.

Data Types: `double`

### `XIntrinsicLimits` — Raster limits in intrinsic *x* coordinates
`[0.5 2.5]` (default) | two-element row vector of positive integers

Raster limits in intrinsic *x* coordinates, specified as a two-element row vector of positive integers, `[xMin xMax]`. For an M-by-N raster, `XIntrinsicLimits` equals `[0.5, M +0.5]`, because the `RasterInterpretation` is `'cells'`.

Cannot be set.

Example: `[0.5 2.5]`

Data Types: `double`

### `YIntrinsicLimits` — Raster limits in intrinsic *y* coordinates
`[0.5 2.5]` (default) | two-element row vector of positive integers

Raster limits in intrinsic *y* coordinates, specified as a two-element row vector of positive integers, `[yMin yMax]`. For an M-by-N raster, `YIntrinsicLimits` equals `[0.5, M +0.5]`, because the `RasterInterpretation` is `'cells'`.

Cannot be set.

Data Types: `double`

### `CoordinateSystemType` — Type of coordinate system to which the image or raster is referenced
`'geographic'` (default)

Type of coordinate system to which the image or raster is referenced, specified as the string `'geographic'`.

Cannot be set.

Data Types: `char`

# Methods

| | |
|---|---|
| contains | True if raster contains latitude-longitude points |
| geographicToDiscrete | Transform geographic to discrete coordinates |

| | |
|---|---|
| geographicToIntrinsic | Convert from geographic to intrinsic coordinates |
| intrinsicToGeographic | Convert from intrinsic to geographic coordinates |
| intrinsicXToLongitude | Convert from intrinsic $x$ to longitude |
| intrinsicYToLatitude | Convert from intrinsic $y$ to latitude |
| latitudeToIntrinsicY | Convert from latitude to intrinsic $y$ |
| longitudeToIntrinsicX | Convert from longitude to intrinsic $x$ |
| sizesMatch | True if object and raster or image are size-compatible |
| worldFileMatrix | World file parameters for transformation |

# Definitions

### Intrinsic Coordinate System

A 2-D Cartesian system with its $x$-axis running parallel to the rows of a raster or image and its $y$-axis running parallel to the columns. $x$ increases by 1 from column to column, and $y$ increases by 1 from row to row.

Mapping Toolbox and Image Processing Toolbox™ use the convention for the location of the origin relative to the raster cells or sampling points such that, at a sample location or at the center of a cell, $x$ has an integer value equal to the column index. Likewise, at a sample location or at the center of a cell, $y$ has an integer value equal to the row index. For details, see Image Coordinate Systems in the Image Processing Toolbox documentation.

## See Also

map.rasterref.GeographicPostingsReference | map.rasterref.MapCellsReference | `georasterref` | `georefcells`

## More About

- Class Attributes
- Property Attributes

**Introduced in R2013b**

# contains

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

True if raster contains latitude-longitude points

## Syntax

```
TF = contains(R,lat,lon)
```

## Description

`TF = contains(R,lat,lon)` returns a logical array `TF` having the same size as `lat` and `lon` such that `TF(k)` is `true` if and only if the point `(lat(k),lon(k))` falls within the bounds of the raster associated with referencing object `R`. Elements of `lon` can be wrapped arbitrarily without affecting the result.

# geographicToDiscrete

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

Transform geographic to discrete coordinates

## Syntax

```
[I,J] = geographicToDiscrete(R,lat,lon)
```

## Description

`[I,J] = geographicToDiscrete(R,lat,lon)` returns subscript arrays `I` and `J`. When referencing object `R` has `RasterInterpretation 'cells'`, `I` and `J` are the row and column subscripts of the raster cells (or image pixels) containing each element of a set of points given their geographic coordinates (`lat, lon`).

`lat` and `lon` must have the same size. `I` and `J` will have the same size as `lat` and `lon`. For an *M*-by-*N* raster, `1 <= I <= M` and `1 <= J <= N`, except when a point `lat(k),lon(k)` falls outside the image, as defined by `contains(R,lat,lon)`. Then both `I(k)` and `J(k)` are `NaN`.

# geographicToIntrinsic

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

Convert from geographic to intrinsic coordinates

## Syntax

```
[xIntrinsic, yIntrinsic] = geographicToIntrinsic(R,lat, lon)
```

## Description

[xIntrinsic, yIntrinsic] = geographicToIntrinsic(R,lat, lon) returns the intrinsic coordinates (xIntrinsic, yIntrinsic) of a set of points given their geographic coordinates (lat, lon), based on the relationship defined by referencing object R. lat and lon must have the same size, and all (non-NaN) elements of lat must fall within the interval [-90 90] degrees. xIntrinsic and yIntrinsic have the same size as lat and lon. The input can include points outside the geographic quadrangle bounding the raster. As long as their latitudes are valid, the locations of such points are extrapolated outside the bounds of the raster in the intrinsic coordinate system.

# intrinsicToGeographic

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

Convert from intrinsic to geographic coordinates

## Syntax

```
[lat, lon] = intrinsicToGeographic(R,xIntrinsic, yIntrinsic)
```

## Description

`[lat, lon] = intrinsicToGeographic(R,xIntrinsic, yIntrinsic)` returns the geographic coordinates (`lat, lon`) of a set of points, given their intrinsic coordinates (`xIntrinsic, yIntrinsic`) and based on the relationship defined by referencing object `R`. `xIntrinsic` and `yIntrinsic` have the same size. `lat` and `lon` have the same size as `xIntrinsic` and `yIntrinsic`.

The input can include points that fall outside the limits of the raster (or image). Latitudes and longitudes for such points are linearly extrapolated outside the geographic quadrangle bounding the raster. However, for any point that extrapolates to a latitude beyond the poles (latitude < −90 degrees or latitude > 90 degrees), the values of `lat` and `lon` are set to NaN.

# intrinsicXToLongitude

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

Convert from intrinsic *x* to longitude

## Syntax

```
lon = intrinsicXToLongitude(R,xIntrinsic)
```

## Description

`lon = intrinsicXToLongitude(R,xIntrinsic)` returns the longitude of the meridian corresponding to the line `x = xIntrinsic`, based on the relationship defined by referencing object `R`. The input can include values that fall completely outside the intrinsic *x*-limits of the raster (or image). In this case, longitude is extrapolated outside the longitude limits. NaN-valued elements of `xIntrinsic` map to NaNs in `lon`.

# intrinsicYToLatitude

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

Convert from intrinsic *y* to latitude

## Syntax

```
lat = intrinsicYToLatitude(R,yIntrinsic)
```

## Description

`lat = intrinsicYToLatitude(R,yIntrinsic)` returns the latitude of the small circle corresponding to the line `y = yIntrinsic`, based on the relationship defined by the referencing object `R`. The input can include values that fall completely outside the intrinsic *y*-limits of the raster (or image). In this case latitude is extrapolated outside the latitude limits, but for input values that extrapolate to latitudes beyond the poles (latitude < –90 degrees or latitude > 90 degrees), the value of `lat` is set to NaN. NaN-valued elements of `yIntrinsic` map to NaNs in `lat`.

# latitudeToIntrinsicY

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

Convert from latitude to intrinsic *y*

## Syntax

```
yIntrinsic = latitudeToIntrinsicY(R,lat)
```

## Description

`yIntrinsic = latitudeToIntrinsicY(R,lat)` returns the intrinsic Y value of the line corresponding to the small circle at latitude `lat`, based on the relationship defined by referencing object `R`. The input can include values that fall completely outside the latitude limits of the raster (or image). In this case, `yIntrinsic` is either extrapolated outside the intrinsic Y limits, for elements of `lat` that fall within the interval `[-90 90]` degrees, or set to NaN, for elements of `lat` that do not correspond to valid latitudes. NaN-valued elements of `lat` map to NaNs in `yIntrinsic`.

# longitudeToIntrinsicX

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

Convert from longitude to intrinsic *x*

## Syntax

```
xIntrinsic = longitudeToIntrinsicX(R,lon)
```

## Description

`xIntrinsic = longitudeToIntrinsicX(R,lon)` returns the intrinsic *x* value
of the line corresponding to the meridian at longitude `lon`, based on the relationship
defined by referencing object `R`. The input can include values that fall completely outside
the longitude limits of the raster (or image). In this case, `xIntrinsic` is extrapolated
outside the intrinsic *x* limits. NaN-valued elements of `lon` map to NaNs in `xIntrinsic`.

# sizesMatch

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

True if object and raster or image are size-compatible

## Syntax

```
TF = sizesMatch(R,A)
```

## Description

`TF = sizesMatch(R,A)` returns `true` if the size of the raster (or image) `A` is consistent with the `RasterSize` property of the referencing object `R`. That is,

```
R.RasterSize == [size(A,1) size(A,2)]
```

# worldFileMatrix

**Class:** map.rasterref.GeographicCellsReference
**Package:** map.rasterref

World file parameters for transformation

## Syntax

```
W = worldFileMatrix(R)
```

## Description

`W = worldFileMatrix(R)` returns a 2-by-3 world file matrix. Each of the six elements in `W` matches one of the lines in a world file corresponding to the transformation defined by referencing object `R`.

Given `W` with the form:

```
W = [A B C;
     D E F]
```

a point `(xi, yi)` in intrinsic coordinates maps to a point `(lat, lon)` in geographic coordinates like this:

```
lon = A * (xi - 1) + B * (yi - 1) + C
lat = D * (xi - 1) + E * (yi - 1) + F
```

More compactly:

```
[lon lat]' = W * [xi - 1  yi − 1  1]'
```

The −1s allow the world file matrix to work with the Mapping Toolbox convention for intrinsic coordinates, which is consistent with the 1-based indexing used throughout MATLAB.

`W` is stored in a world file with one term per line in column-major order: A, D, B, E, C, F. That is, a world file contains the elements of `W` in this order:

```
W(1,1)
```

```
W(2,1)
W(1,2)
W(2,2)
W(1,3)
W(2,3)
```

More compactly:

```
[lon lat]' = W * [(xi-1)(yi-1) 1]'
```

The previous expressions hold for a general affine transformation. But in the matrix returned by this method B, D, W(2,1) and W(1,2) are identically 0 because longitude depends only on intrinsic $x$ and latitude depends only on intrinsic $y$.

## See Also

worldfileread | worldfilewrite

# map.rasterref.GeographicPostingsReference class

**Package:** map.rasterref

Reference raster postings to geographic coordinates

## Description

A geographic postings raster reference object encapsulates the relationship between a geographic coordinate system and a system of intrinsic coordinates anchored to the columns and rows of a 2-D spatially referenced grid of point samples (or "postings"). The raster must be sampled regularly in latitude and longitude, and its columns and rows must be aligned with meridians and parallels, respectively. For more information about coordinate systems, see "Intrinsic Coordinate System" on page 1-465.

## Construction

You can use any of the following functions to construct a `map.rasterref.PostingsReference` object to reference a regular raster of posted samples to geographic coordinates.

- `georefpostings`—Create a geographic raster reference object.
- `georasterref`—Convert a world file to a geographic raster reference object.
- `refmatToGeoRasterReference`—Convert a referencing matrix to a geographic raster reference object.

For example, to construct a geographic raster reference object with default property settings, use this command:

```
R = georefpostings()

R =

  GeographicPostingsReference with properties:

           LatitudeLimits: [0.5 1.5]
          LongitudeLimits: [0.5 1.5]
               RasterSize: [2 2]
```

```
       RasterInterpretation: 'postings'
             ColumnsStartFrom: 'south'
               RowsStartFrom: 'west'
     SampleSpacingInLatitude: 1
    SampleSpacingInLongitude: 1
      RasterExtentInLatitude: 1
     RasterExtentInLongitude: 1
             XIntrinsicLimits: [1 2]
             YIntrinsicLimits: [1 2]
         CoordinateSystemType: 'geographic'
                    AngleUnit: 'degree'
```

# Properties

### `LatitudeLimits` — Latitude limits of the geographic quadrangle bounding the georeferenced raster
`[0.5 1.5]` (default) | two-element vector

Latitude limits of the geographic quadrangle bounding the georeferenced raster, specified as a two-element vector of the form `[southern_limit northern_limit]`.

Example: `[-20 70]`

Data Types: `double`

### `LongitudeLimits` — Longitude limits of the geographic quadrangle bounding the georeferenced raster
`[0.5 1.5]` (default) | two-element vector

Longitude limits of the geographic quadrangle bounding the georeferenced raster, specified as a two-element vector of the form `[western_limit eastern_limit]`.

Example: `[-100 180]`

Data Types: `double`

### `RasterSize` — Number of rows and columns of the raster or image associated with the referencing object
`[2 2]` (default) | two-element vector of positive integers

Number of rows and columns of the raster or image associated with the referencing object, specified as a two-element vector, `[M N]`, where *M* represents the number of rows and *N* the number of columns. For convenience, you can assign a size vector having more

than two elements. This enables assignments like `R.RasterSize = size(RGB)`, where `RGB` is *M*-by-*N*-by-3. In cases like this, the object stores only the first two elements of the size vector and ignores the higher (nonspatial) dimensions.

Example: `[200 300]`

Data Types: `double`

### RasterInterpretation — Geometric nature of the raster
`'postings'` (default)

Geometric nature of the raster, specified as the string `'postings'`.

The value `'postings'` indicates that the raster comprises a grid of sample points, where rows or columns of samples run along the edge of the grid. For an *M*-by-*N* raster, points with an intrinsic *X*-coordinate of 1 or *N* and/or an intrinsic *Y*-coordinate of 1 or *M* fall right on an edge (or corner) of the raster.

Cannot be set.

Data Types: `char`

### AngleUnit — Unit of measurement used for angle-valued properties
`'degree'` (default)

Unit of measurement used for angle-valued properties, specified as the string `'degree'`.

Cannot be set.

Data Types: `char`

### ColumnsStartFrom — Edge from which column indexing starts
`'south'` (default) | `'north'`

Edge from which column indexing starts, specified as the string `'south'` or `'north'`.

Example: `'south'`

Data Types: `char`

### RowsStartFrom — Edge from which row indexing starts
`'west'` (default) | `'east'`

Edge from which row indexing starts, specified as the string `'west'` or `'east'`.

Example: `'east'`

Data Types: char

### `SampleSpacingInLatitude` — North-south distance in latitude between adjacent samples in the raster

1 (default) | positive numeric scalar

North-south distance in latitude between adjacent samples (postings) in the raster, specified as a positive numeric scalar. The value is always positive, and is the constant throughout the raster.

Example: 2.5

Data Types: double

### `SampleSpacingInLongitude` — East-west distance in longitude between adjacent samples in the raster

1 (default) | positive numeric scalar

East-west distance in longitude between adjacent samples (postings) in the raster, specified as a postive numeric scalar. The value is always positive, and is the constant throughout the raster.

Example: 2.5

Data Types: double

### `RasterExtentInLatitude` — Latitude extent ("height") of the quadrangle covered by the raster

1 (default) | positive numeric scalar

Latitude extent ("height") of the quadrangle covered by the raster, specified as a positive numeric scalar.

Cannot be set.

Data Types: double

### `RasterExtentInLongitude` — Longitude extent ("width") of the quadrangle covered by the raster

1 (default) | positive numeric scalar

Longitude extent ("width") of the quadrangle covered by the raster, specified as a positive numeric scalar.

Cannot be set.

Data Types: `double`

### `XIntrinsicLimits` — Raster limits in intrinsic *x* coordinates
`[1 2]` (default) | two-element row vector of positive integers

Raster limits in intrinsic *x* coordinates, specified as a two-element row vector of positive integers, `[xMin xMax]`. For an M-by-N raster, `XntrinsicLimits` equals `[1 M]`, because the `RasterInterpretation` is `'postings'`.

Cannot be set.

Data Types: `double`

### `YIntrinsicLimits` — Raster limits in intrinsic *y* coordinates
`[1 2]` (default) | two-element row vector of positive integers

Raster limits in intrinsic *y* coordinates, specified as a two-element row vector of positive integers, `[yMin yMax]`. For an M-by-N raster, `YIntrinsicLimits` equals `[1 M]`, because the `RasterInterpretation` is `'postings'`.

Cannot be set.

Data Types: `double`

### `CoordinateSystemType` — Type of coordinate system to which the image or raster is referenced
`'geographic'` (default) | two-element row vector of positive integers

Type of coordinate system to which the image or raster is referenced, specified as the string `'geographic'`.

Cannot be set.

Data Types: `char`

## Methods

| | |
|---|---|
| contains | True if raster contains latitude-longitude points |
| geographicToDiscrete | Transform geographic to discrete coordinates |

| geographicToIntrinsic | Convert from geographic to intrinsic coordinates |
| intrinsicToGeographic | Convert from intrinsic to geographic coordinates |
| intrinsicXToLongitude | Convert from intrinsic $x$ to longitude |
| intrinsicYToLatitude | Convert from intrinsic $y$ to latitude |
| latitudeToIntrinsicY | Convert from latitude to intrinsic $y$ |
| longitudeToIntrinsicX | Convert from longitude to intrinsic $x$ |
| sizesMatch | True if object and raster or image are size-compatible |
| worldFileMatrix | World file parameters for transformation |

# Definitions

## Intrinsic Coordinate System

A 2-D Cartesian system with its $x$-axis running parallel to the rows of a raster or image and its $y$-axis running parallel to the columns. $x$ increases by 1 from column to column, and $y$ increases by 1 from row to row.

Mapping Toolbox and Image Processing Toolbox use the convention for the location of the origin relative to the raster cells or sampling points such that, at a sample location or at the center of a cell, $x$ has an integer value equal to the column index. Likewise, at a sample location or at the center of a cell, $y$ has an integer value equal to the row index. For details, see Image Coordinate Systems in the Image Processing Toolbox documentation.

## See Also

map.rasterref.MapPostingsReference | map.rasterref.MapCellsReference | georasterref | georefpostings

## More About

- Class Attributes
- Property Attributes

**Introduced in R2013b**

# contains

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

True if raster contains latitude-longitude points

## Syntax

```
TF = contains(R,lat,lon)
```

## Description

TF = contains(R,lat,lon) returns a logical array TF having the same size as lat and lon such that TF(k) is true if and only if the point (lat(k),lon(k)) falls within the bounds of the raster associated with referencing object R. Elements of lon can be wrapped arbitrarily without affecting the result.

# geographicToDiscrete

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

Transform geographic to discrete coordinates

## Syntax

```
[I,J] = geographicToDiscrete(R,lat,lon)
```

## Description

`[I,J] = geographicToDiscrete(R,lat,lon)` returns subscript arrays `I` and `J`. When `R.RasterInterpretation` is `'postings'`, `I` and `J` refer to the nearest sample point (posting).

`lat` and `lon` must have the same size. `I` and `J` will have the same size as `lat` and `lon`. For an *M*-by-*N* raster, `1 <= I <= M` and `1 <= J <= N`, except when a point `lat(k),lon(k)` falls outside the image, as defined by `contains(R,lat,lon)`. Then both `I(k)` and `J(k)` are `NaN`.

# geographicToIntrinsic

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

Convert from geographic to intrinsic coordinates

## Syntax

```
[xIntrinsic, yIntrinsic] = geographicToIntrinsic(R,lat, lon)
```

## Description

`[xIntrinsic, yIntrinsic] = geographicToIntrinsic(R,lat, lon)` returns the intrinsic coordinates (`xIntrinsic, yIntrinsic`) of a set of points given their geographic coordinates (`lat, lon`), based on the relationship defined by referencing object `R`. `lat` and `lon` must have the same size, and all (non-NaN) elements of `lat` must fall within the interval `[-90 90]` degrees. `xIntrinsic` and `yIntrinsic` have the same size as `lat` and `lon`. The input can include points outside the geographic quadrangle bounding the raster. As long as their latitudes are valid, the locations of such points are extrapolated outside the bounds of the raster in the intrinsic coordinate system.

# intrinsicToGeographic

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

Convert from intrinsic to geographic coordinates

## Syntax

```
[lat, lon] = intrinsicToGeographic(R,xIntrinsic, yIntrinsic)
```

## Description

`[lat, lon] = intrinsicToGeographic(R,xIntrinsic, yIntrinsic)` returns the geographic coordinates (`lat, lon`) of a set of points, given their intrinsic coordinates (`xIntrinsic, yIntrinsic`) and based on the relationship defined by referencing object `R`. `xIntrinsic` and `yIntrinsic` have the same size. `lat` and `lon` have the same size as `xIntrinsic` and `yIntrinsic`.

The input can include points that fall outside the limits of the raster (or image). Latitudes and longitudes for such points are linearly extrapolated outside the geographic quadrangle bounding the raster. However, for any point that extrapolates to a latitude beyond the poles (latitude < −90 degrees or latitude > 90 degrees), the values of `lat` and `lon` are set to NaN.

# intrinsicXToLongitude

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

Convert from intrinsic *x* to longitude

## Syntax

```
lon = intrinsicXToLongitude(R,xIntrinsic)
```

## Description

`lon = intrinsicXToLongitude(R,xIntrinsic)` returns the longitude of the meridian corresponding to the line `x = xIntrinsic`, based on the relationship defined by referencing object `R`. The input can include values that fall completely outside the intrinsic *x*-limits of the raster (or image). In this case, longitude is extrapolated outside the longitude limits. NaN-valued elements of `xIntrinsic` map to NaNs in `lon`.

# intrinsicYToLatitude

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

Convert from intrinsic *y* to latitude

## Syntax

```
lat = intrinsicYToLatitude(R,yIntrinsic)
```

## Description

`lat = intrinsicYToLatitude(R,yIntrinsic)` returns the latitude of the small circle corresponding to the line `y = yIntrinsic`, based on the relationship defined by the referencing object `R`. The input can include values that fall completely outside the intrinsic *y*-limits of the raster (or image). In this case latitude is extrapolated outside the latitude limits, but for input values that extrapolate to latitudes beyond the poles (latitude < –90 degrees or latitude > 90 degrees), the value of `lat` is set to NaN. NaN-valued elements of `yIntrinsic` map to NaNs in `lat`.

# latitudeToIntrinsicY

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

Convert from latitude to intrinsic *y*

## Syntax

```
yIntrinsic = latitudeToIntrinsicY(R,lat)
```

## Description

`yIntrinsic = latitudeToIntrinsicY(R,lat)` returns the intrinsic Y value of the line corresponding to the small circle at latitude `lat`, based on the relationship defined by referencing object `R`. The input can include values that fall completely outside the latitude limits of the raster (or image). In this case, `yIntrinsic` is either extrapolated outside the intrinsic Y limits, for elements of `lat` that fall within the interval `[-90 90]` degrees, or set to NaN, for elements of `lat` that do not correspond to valid latitudes. NaN-valued elements of `lat` map to NaNs in `yIntrinsic`.

# longitudeToIntrinsicX

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

Convert from longitude to intrinsic *x*

## Syntax

```
xIntrinsic = longitudeToIntrinsicX(R,lon)
```

## Description

xIntrinsic = longitudeToIntrinsicX(R,lon) returns the intrinsic *x* value of the line corresponding to the meridian at longitude lon, based on the relationship defined by referencing object R. The input can include values that fall completely outside the longitude limits of the raster (or image). In this case, xIntrinsic is extrapolated outside the intrinsic *x* limits. NaN-valued elements of lon map to NaNs in xIntrinsic.

# sizesMatch

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

True if object and raster or image are size-compatible

## Syntax

```
TF = sizesMatch(R,A)
```

## Description

`TF = sizesMatch(R,A)` returns `true` if the size of the raster (or image) `A` is consistent with the `RasterSize` property of the referencing object `R`. That is,

```
R.RasterSize == [size(A,1) size(A,2)]
```

# worldFileMatrix

**Class:** map.rasterref.GeographicPostingsReference
**Package:** map.rasterref

World file parameters for transformation

## Syntax

```
W = worldFileMatrix(R)
```

## Description

`W = worldFileMatrix(R)` returns a 2-by-3 world file matrix. Each of the six elements in `W` matches one of the lines in a world file corresponding to the transformation defined by referencing object `R`.

Given `W` with the form:

```
W = [A B C;
     D E F]
```

a point (`xi, yi`) in intrinsic coordinates maps to a point (`lat, lon`) in geographic coordinates like this:

```
lon = A * (xi - 1) + B * (yi - 1) + C
lat = D * (xi - 1) + E * (yi - 1) + F
```

More compactly:

```
[lon lat]' = W * [xi - 1  yi − 1  1]'
```

The −1s allow the world file matrix to work with the Mapping Toolbox convention for intrinsic coordinates, which is consistent with the 1-based indexing used throughout MATLAB.

`W` is stored in a world file with one term per line in column-major order: A, D, B, E, C, F. That is, a world file contains the elements of `W` in this order:

```
W(1,1)
```

```
W(2,1)
W(1,2)
W(2,2)
W(1,3)
W(2,3)
```

More compactly:

```
[lon lat]' = W * [(xi-1)(yi-1) 1]'
```

The previous expressions hold for a general affine transformation. But in the matrix returned by this method B, D, W(2,1) and W(1,2) are identically 0 because longitude depends only on intrinsic *x* and latitude depends only on intrinsic *y*.

## See Also
worldfileread | worldfilewrite

# geoshape class

Geographic shape vector

## Syntax

```
s = geoshape()
s = geoshape(lat,lon)
s = geoshape(lat,lon,Name,Value)
s = geoshape(structArray)
s = geoshape(lat,lon,structArray)
```

## Description

A geoshape vector is an object that represents geographic vector features with either point, line, or polygon topology. The features consist of latitude and longitude coordinates and associated attributes. If these attributes vary spatially they are termed Vertex properties. These elements of the geoshape vector are coupled such that the length of the latitude and longitude coordinate property values are always equal in length to any additional dynamic Vertex properties. Attributes which only pertain to the overall feature (point, line, polygon) are termed Feature properties. Feature properties are not linked to the auto-sizing mechanism of the Vertex properties. Both of the property types can be dynamically added to a geoshape vector using the standard dot notation.

A geoshape vector is always a column vector.

## Construction

`s = geoshape()` constructs an empty geoshape vector, `s`, with the following default property settings.

```
s =

 0x1 geoshape vector with properties:
```

```
Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
Vertex properties:
    Latitude: []
   Longitude: []
```

For an additional example see: "Construct a Default geoshape Vector" on page 1-485

`s = geoshape(lat,lon)` constructs a geoshape vector and sets the `Latitude` and `Longitude` property values equal to vectors `lat` and `lon`. `lat` and `lon` may be either numeric vectors of class single or double, or cell arrays containing numeric vectors of class single or double. For an example, see "Construct a geoshape vector Specifying Latitude and Longitude Values" on page 1-485.

`s = geoshape(lat,lon,Name,Value)` constructs a geoshape vector from the input `lat` and `lon` vectors, and then adds dynamic properties to the geoshape vector using the `Name`, `Value` argument pairs.

- If `Value` is in a cell array containing numeric, logical or cell array of strings, then this property is designated as a Vertex property. Otherwise, this property is designated as a Feature property.

- If the specified `Name` is `Metadata` and the corresponding `Value` is a scalar structure, then `Value` is copied to the `Metadata` property. Otherwise, an error is issued.

For an example, see: "Construct a geoshape Vector Specifying a Name-Value Pair" on page 1-486.

`s = geoshape(structArray)` constructs a `geoshape` vector from the fields of the structure array, `structArray`.

- If `structArray` contains the field `Lat`, and does not contain a field `Latitude`, then the `Latitude` property values are set equal to the `Lat` field values. If `structArray` contains the field, `Lon`, and does not contain a field `Longitude`, then the `Longitude` property values are set equal to the `Lon` field values.

- If `structArray` contains both `Lat` and `Latitude` fields, then the `Latitude` property values are set equal to the `Latitude` field values and a `Lat` dynamic property is created whose values are set equal to the `Lat` field values. Similar behavior occurs for `Longitude` and `Lon` field combinations if present in `structArray`.

**1-479**

- If `structArray` is a scalar structure which contains the field `Metadata` and the field value is a scalar structure, then the `Metadata` field is copied to the `Metadata` property. If `structArray` is a scalar structure and the `Metadata` field is present and is not a scalar structure, then an error is issued. If `structArray` is not scalar then the `Metadata` field is ignored.

- Other fields of `structArray` are assigned to `s` and become dynamic properties. Field values in `structArray` that are not numeric, strings, logical, or cell arrays of numeric, logical, or string values are ignored.

For an example, see "Construct a geoshape vector from a Structure Array" on page 1-486.

`s = geoshape(lat,lon,structArray)` constructs a new `geoshape` vector and sets the `Latitude` and `Longitude` properties equal to the numeric vectors, `lat` and `lon`, and sets the field values of struct `structArray` as dynamic properties.

- If `structArray` contains the fields `Lat`, `Latitude`, `Lon` or `Longitude`, then those field values are ignored since the `Latitude` and `Longitude` property values are set by the `lat` and `lon` input vectors.

- If `structArray` is a scalar structure and contains the field `Metadata`, and the field value is a scalar structure, then it is copied to the `Metadata` property value. Otherwise, an error is issued if the `Metadata` field is not a structure, or ignored if `structArray` is not scalar.

For an example, see "Construct a geoshape vector Using Cell Arrays" on page 1-487.

## Input Arguments

### `lat` — Vector of latitude coordinates
numeric array

Vector of latitude coordinates, specified as a numeric array.

Data Types: `double` | `single` | `cell`

### `lon` — Vector of longitude coordinates
numeric array

Vector of longitude coordinates, specified as a numeric array.

Data Types: `double` | `single` | `cell`

### `structArray` — Dynamic properties
structure

Dynamic properties, specified as an array of structures containing fields to be assigned.

### `Name` — Name of dynamic property
string

Name of dynamic property, specified as a string.

Data Types: `char`

### `Value` — Value associated with a dynamic property specified in `Name`
numeric, logical, char, or a cell array of strings

Value associated with a dynamic property specified in `Name`, specified as numeric, logical, cell array of strings, or a cell array of numeric, logical, or cell array of strings, for Vertex dynamic properties. The class type of the values for the Feature dynamic properties may be either numeric, logical, char, or a cell array of strings.

## Output Arguments

**s**

geoshape vector.

## Properties

`geoshape` class is a general class that represents a variety of geographic features. The class permits features to have more than one vertex and can thus represent lines and polygons in addition to multipoints. The class has the following property types.

| Types of Properties | Description |
|---|---|
| Collection Properties | Collection properties contain only one value per class instance. This is in contrast to the other two property types which can have attribute values associated with each feature or with each vertex in a set that defines a feature. |

| Types of Properties | Description |
| --- | --- |
| | Geometry and Metadata are the only two Collection properties. |
| Vertex Properties | Vertex properties provide a scalar number or a string for each vertex in a geoshape object. Vertex properties are suitable for attributes that vary spatially from point to point (vertex to vertex) along a line. Examples of such spatially varying attributes could be elevation, speed, temperature, or time. Latitude and Longitude are vertex properties since they contain a scalar number for each vertex in a geoshape vector. Attribute values can be dynamically associated with each vertex by using dot notation. This is similar to adding dynamic fields to a structure. The dynamically added vertex property values of an individual feature match its Latitude and Longitude values in length. |
| Feature Properties | Feature properties provide one value (a scalar number or a string) for each feature in a geoshape vector. They are suitable for properties, such as name, owner, serial number, age, etc., that describe a given feature (an element of a geoshape vector) as a whole. Like Vertex properties, Feature properties can be added dynamically. |

**Geometry**

The `Geometry` property is a string that denotes the shape type for all the features in the geoshape vector. As a Collection Property there is only one value per object instance. Its purpose is purely informational; the three allowable string values for `Geometry` do not change class behavior. Additionally, the class does not provide validation for line or polygon topologies.

Default value for `Geometry` is `'line'`.

| Geometry | `'point'`, `'line'`, `'polygon'` |

**Metadata**

Metadata is a scalar structure containing information for all the features. You can add any data type to the structure. As a Collection Property type, only one instance per object is allowed.

```
Metadata                              Scalar struct
```

**Latitude**

Vector of latitude coordinates. The values can be either a row or column vector, but are stored as a row vector.

**Attributes:**

```
Latitude                              single | double vector
```

**Longitude**

Vector of longitude coordinates. The values can be either a row or column vector, but are stored as a row vector.

**Attributes:**

```
Longitude                             single | double vector
```

# Methods

| append | Append features to geoshape vector |
|---|---|
| cat | Concatenate geoshape vectors |
| disp | Display geoshape vector |
| fieldnames | Dynamic properties of geoshape vector |
| isempty | True if geoshape vector is empty |
| isfield | True if dynamic property exists |
| isprop | True if property exists |
| length | Number of elements in geoshape vector |
| properties | Properties of a geoshape vector |
| rmfield | Remove dynamic property from geoshape vector |

| | |
|---|---|
| rmprop | Remove properties from geoshape vector |
| size | Size of `geoshape` vector |
| struct | Convert geoshape vector to scalar structure |
| vertcat | Vertical concatenation for geoshape vectors |

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Class Behaviors

- The following examples show how to build geoshape vectors with single or multiple features.

  "Construct geoshape Vector and Dynamically Add Feature Property" on page 1-487

  "Construct a geoshape vector and define multiple point features" on page 1-489

- You can attach dynamic feature and vertex properties to a geoshape vector by either adding them at construction or by using the dot (.) notation after construction. This is similar to adding new fields to a structure. The following examples highlight the two ways by which a geoshape vector with the same features can be created. In each example the data is read from the same file into a structure array. The first example dynamically adds the features to a geoshape object created with the default constructor. The second example uses the structure array as an argument to constructor which builds the geoshape vector.

  "Construct a geoshape vector and Dynamically Add Multiple Line Features" on page 1-490

  "Construct a geoshape vector from a Structure Array " on page 1-491

- The geoshape vector can be indexed like any MATLAB vector. You can access any element of the vector to obtain a specific feature. The following examples demonstrate this behavior.

  "Use Indexing to Append a Single Point and a Shape to a geoshape Vector" on page 1-493

- The following example demonstrates the behavior of differing vector representations of the `Latitude` and `Longitude` properties.

- If either `Latitude` or `Longitude` is set to [ ], then both coordinate properties are set to [ ] and all dynamic Vertex or Feature properties are removed.
- If a Vertex or Feature property is set to [ ], then it is removed from the object.

## Examples

### Construct a Default geoshape Vector

Create a geoshape vector and dynamically set the `Latitude` and `Longitude` property values, and dynamically add Vertex property `Z`

```
s = geoshape();
s(1).Latitude = 0:45:90;
s(1).Longitude = [10 10 10];
s(1).Z = [10 20 30]

s =

 1x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
     Latitude: [0 45 90]
    Longitude: [10 10 10]
            Z: [10 20 30]
```

### Construct a geoshape vector Specifying Latitude and Longitude Values

Create a geoshape vector specifying latitude and longitude values.

```
s = geoshape([42 43 45], [10 11 15])

s =

 1x1 geoshape vector with properties:

 Collection properties:
      Geometry: 'line'
      Metadata: [1x1 struct]
 Vertex properties:
      Latitude: [42 43 45]
     Longitude: [10 11 15]
```

### Construct a geoshape Vector Specifying a Name-Value Pair

Create a geoshape vector using a Name-Value pair to define a new Feature property.

```
point = geoshape(42, -72, 'Temperature', 89);
point.Geometry = 'point'

point =

 1x1 geoshape vector with properties:

 Collection properties:
        Geometry: 'point'
        Metadata: [1x1 struct]
 Vertex properties:
        Latitude: 42
       Longitude: -72
 Feature properties:
     Temperature: 89
```

When Value is a cell array containing numeric, logical, or cell array of strings, it is construed as a Vertex property. Otherwise the Name-Value pair is designated as being a Feature property.

### Construct a geoshape vector from a Structure Array

Create a structure array and then create a geoshape vector.

```
structArray = shaperead('worldrivers', 'UseGeoCoords', true);
shape = geoshape(structArray)

s =
```

```
 128x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
  (128 features concatenated with 127 delimiters)
     Latitude: [1x5542 double]
    Longitude: [1x5542 double]
 Feature properties:
         Name: {1x128 cell}
```

### Construct a geoshape vector Using Cell Arrays

Use cell arrays to define multiple features and a structure array to define a set of
Feature properties.

```
[structArray2, structNames] = shaperead('worldrivers', 'UseGeoCoords', true);
s = geoshape({structArray2.Lat}, {structArray2.Lon}, structNames)

s =

 128x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
  (128 features concatenated with 127 delimiters)
     Latitude: [1x5542 double]
    Longitude: [1x5542 double]
 Feature properties:
         Name: {1x128 cell}
```

### Construct geoshape Vector and Dynamically Add Feature Property

Load coastline data from a MAT-file.

```
load coastlines
```

Create geoshape vector with latitude and longitude data from coastlines.

```
s = geoshape(coastlat,coastlon)
```

```
s =

 1x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
     Latitude: [1x9865 double]
    Longitude: [1x9865 double]
```

Add a Feature property to the geoshape dynamically and display it. Note that the new Feature property `Name` applies to the entire set of vertices defined by the `Geometry` property as representing a line.

```
s.Name = 'coastline'
```

```
s =

 1x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
     Latitude: [1x9865 double]
    Longitude: [1x9865 double]
 Feature properties:
         Name: 'coastline'
```

Display the map with coastline data.

```
figure
worldmap world
geoshow(s.Latitude,s.Longitude)
```
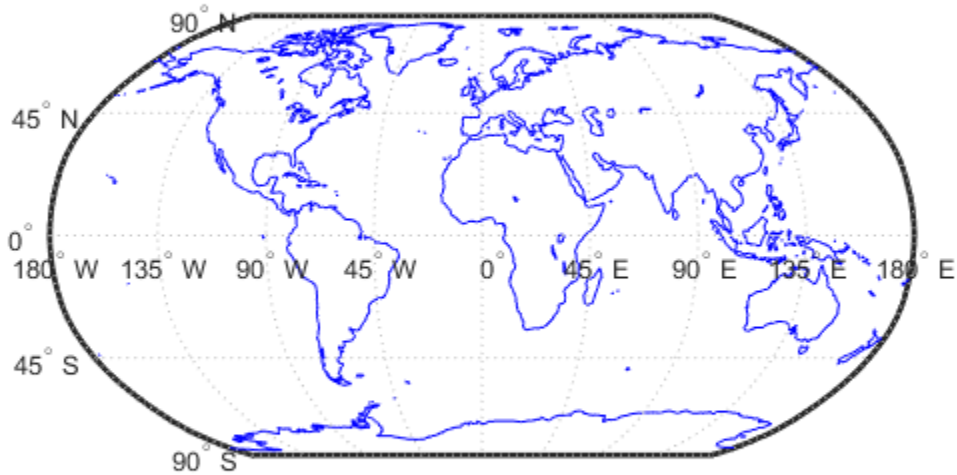
### Construct a geoshape vector and define multiple point features

Define the latitude and longitude data and create a geoshape vector.

```
lat = {42, 42.3};
lon = {-72, -72.85};
temperature = {89, 87.5};
s = geoshape(lat, lon, 'Temperature', temperature);
s.Geometry = 'point'

s =

 2x1 geoshape vector with properties:
```

```
 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
       Latitude: [42 NaN 42.3000]
      Longitude: [-72 NaN -72.8500]
    Temperature: [89 NaN 87.5000]
```

Note that the number of `temperature` entries match the number of `lat` and `lon` entries which causes this attribute to be classified as a `Vertex property`.

### Construct a geoshape vector and Dynamically Add Multiple Line Features

Create a structure array, `structArray`, contains 128 elements. Each element defines a river as a line using multiple location vertices.

```
structArray = shaperead('worldrivers', 'UseGeoCoords', true)

structArray =

128x1 struct array with fields:
    Geometry
    BoundingBox
    Lon
    Lat
    Name
```

Display one element in the 128 element structure array.

```
structArray(1)

ans =

      Geometry: 'Line'
    BoundingBox: [2x2 double]
           Lon: [126.7796 126.5321 126.3121 126.2383 126.0362 NaN]
           Lat: [73.4571 73.0669 72.8343 72.6010 72.2894 NaN]
          Name: 'Lena
```

Note that the `Lat` and `Lon` vectors are terminated with a `NaN` delimiter.

Dynamically add Vertex properties from each entry in the structure array. Then add a Feature property: the name of each of the 128 rivers in `shape`.

```
shape = geoshape();
```

```
for k=1:length(shape)
    shape(k).Latitude  = structArray(k).Lat;
    shape(k).Longitude = structArray(k).Lon;
end

shape.Name = {structArray.Name}

shape =

 128x1 geoshape vector with properties:

 Collection properties:
      Geometry: 'line'
      Metadata: [1x1 struct]
 Vertex properties:
  (128 features concatenated with 127 delimiters)
      Latitude: [1x5542 double]
     Longitude: [1x5542 double]
 Feature properties:
          Name: {1x128 cell}
```

Note that the features are concatenated with a NaN delimiter.

Since the default value of the Geometry Collection property is 'line' there is no need to set it explicitly in this example.

### Construct a geoshape vector from a Structure Array

Read in structure array containing 128 elements which define world rivers. Display one element of the array.

```
structArray = shaperead('worldrivers', 'UseGeoCoords', true)
structArray(1)

ans =

       Geometry: 'Line'
    BoundingBox: [2x2 double]
            Lon: [126.7796 126.5321 126.3121 126.2383 126.0362 NaN]
            Lat: [73.4571 73.0669 72.8343 72.6010 72.2894 NaN]
           Name: 'Lena'
```

Note that the Lat and Lon vectors are terminated with NaN delimiters which used to separate the Vertex feature data in the geoshape class.

**1-491**

The structure array contains all the data needed to construct the 128 element geoshape vector.

```
shape = geoshape(structArray)

shape =

 128x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
  (128 features concatenated with 127 delimiters)
     Latitude: [1x5542 double]
    Longitude: [1x5542 double]
 Feature properties:
         Name: {1x128 cell}
```

Display only the first 5 feature elements of the geoshape vector.

```
shape(1:5)

ans =

 5x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
  (5 features concatenated with 4 delimiters)
     Latitude: [1x52 double]
    Longitude: [1x52 double]
 Feature properties:
         Name: {'Lena' 'Lena' 'Mackenzie' 'Mackenzie' 'Kolyma'}
```

Note that indexing of the first five elements of the geoshape vector displays the corresponding number of Feature properties in `Name`.

Add a filename field to the `Metadata` structure which is a Collection property and thus applies to all features in the object.

```
shape.Metadata.Filename = 'worldcities.shp';
shape.Metadata
```

```
ans =

    Filename: 'worldcities.shp'
```

### Use Indexing to Append a Single Point and a Shape to a geoshape Vector

Create a geoshape vector containing a single feature of the locations of world cities.

```
S = shaperead('worldcities.shp', 'UseGeoCoords', true);
cities = geoshape([S.Lat], [S.Lon], 'Name', {{S.Name}});
cities.Geometry = 'point';
```

Append Paderborn Germany to the geoshape vector.

```
lat = 51.715254;
lon = 8.75213;
cities(1).Latitude(end+1) = lat;
cities(1).Longitude(end) = lon;
cities(1).Name{end} = 'Paderborn'

cities =

 1x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Vertex properties:
     Latitude: [1x319 double]
    Longitude: [1x319 double]
         Name: {1x319 cell}
```

The length of each vertex property grows by 1 when `Latitude(end+1)` is set. The remaining properties are indexed with `end`.

You can display the last point by constructing a geopoint vector.

```
paderborn = geopoint(cities.Latitude(end), cities.Longitude(end), ...
   'Name', cities.Name{end})

paderborn =

 1x1 geopoint vector with properties:

 Collection properties:
```

```
      Geometry: 'point'
      Metadata: [1x1 struct]
 Feature properties:
      Latitude: 51.7153
     Longitude: 8.7521
          Name: 'Paderborn'
```

Create a new geoshape vector with two new features containing the cities in the northern and southern hemispheres. Add a Location dynamic feature property to distinguish the different classifications.

```
northern = cities(1).Latitude >= 0;
southern = cities(1).Latitude < 0;
index = {northern; southern};
location = {'Northern Hemisphere', 'Southern Hemisphere'};
hemispheres = geoshape();
for k = 1:length(index)
   hemispheres = append(hemispheres, ...
      cities.Latitude(index{k}), cities.Longitude(index{k}), ...
      'Name', {cities.Name(index{k})}, 'Location', location{k});
end
hemispheres.Geometry = 'point'

hemispheres =

 2x1 geoshape vector with properties:

 Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
      Latitude: [1x320 double]
     Longitude: [1x320 double]
          Name: {1x320 cell}
 Feature properties:
      Location: {'Northern Hemisphere' 'Southern Hemisphere'}
```

Plot the northern cities in red and the southern cities in blue.
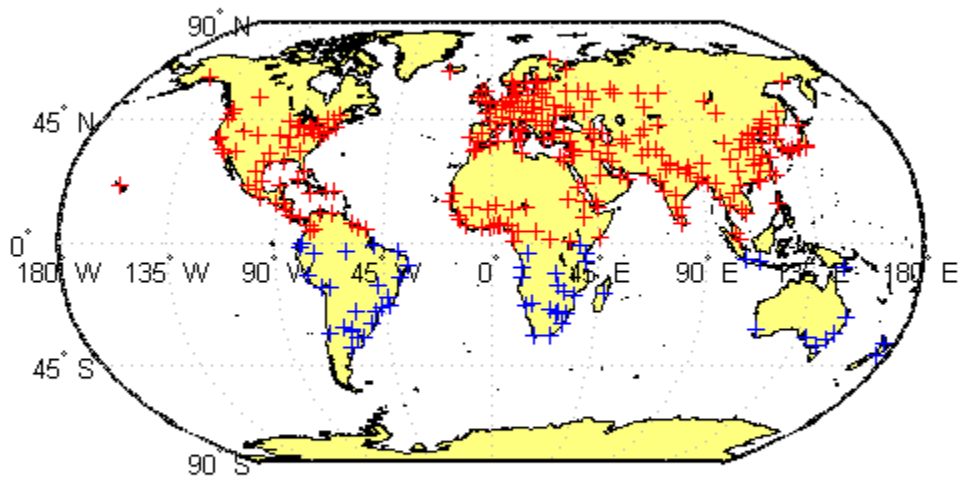
```
hemispheres.Color = {'red', 'blue'};
figure;worldmap('world')
geoshow('landareas.shp')
for k=1:2
   geoshow(hemispheres(k).Latitude, hemispheres(k).Longitude, ...
```

```
        'DisplayType', hemispheres.Geometry, ...
        'MarkerEdgeColor', hemispheres(k).Color)
end
```



### Using Indexing To Sort and Modify Dynamic Features

Construct a geoshape vector and sort its dynamic properties.

```
shape = geoshape(shaperead('tsunamis', 'UseGeoCoords', true));
shape.Geometry = 'point';
shape = shape(:, sort(fieldnames(shape)))
```

```
shape =
```

```
162x1 geoshape vector with properties:

Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
Vertex properties:
 (162 features concatenated with 161 delimiters)
       Latitude: [1x323 double]
      Longitude: [1x323 double]
Feature properties:
          Cause: {1x162 cell}
     Cause_Code: [1x162 double]
        Country: {1x162 cell}
            Day: [1x162 double]
    Desc_Deaths: [1x162 double]
         Eq_Mag: [1x162 double]
           Hour: [1x162 double]
       Iida_Mag: [1x162 double]
      Intensity: [1x162 double]
       Location: {1x162 cell}
     Max_Height: [1x162 double]
         Minute: [1x162 double]
          Month: [1x162 double]
     Num_Deaths: [1x162 double]
         Second: [1x162 double]
       Val_Code: [1x162 double]
       Validity: {1x162 cell}
           Year: [1x162 double
```

Modify the geoshape vector to contain only the dynamic properties, `Year`, `Month`, `Day`, `Hour`, `Minute`.

```
shape = shape(:, {'Year', 'Month', 'Day', 'Hour', 'Minute'})

shape =

 162x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Vertex properties:
  (162 features concatenated with 161 delimiters)
     Latitude: [1x323 double]
```

```
     Longitude: [1x323 double]
 Feature properties:
         Year: [1x162 double]
        Month: [1x162 double]
          Day: [1x162 double]
         Hour: [1x162 double]
       Minute: [1x162 double]
```

Display the first 5 elements.

```
shape(1:5)

ans =

 5x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Vertex properties:
  (5 features concatenated with 4 delimiters)
     Latitude: [-3.8000 NaN 19.5000 NaN -9.0200 NaN 42.1500 NaN 19.1000]
    Longitude: [128.3000 NaN -156 NaN 157.9500 NaN 143.8500 NaN -155]
 Feature properties:
         Year: [1950 1951 1951 1952 1952]
        Month: [10 8 12 3 3]
          Day: [8 21 22 4 17]
         Hour: [3 10 NaN 1 3]
       Minute: [23 57 NaN 22 58]
```

### Construct a Two-element geoshape vector Containing GPS Track Log Data

Read multiple GPS track log data from a file. `trk1` and `trk2` are geopoint objects.

```
trk1 = gpxread('sample_tracks')
trk2 = gpxread('sample_tracks', 'Index', 2);

trk1 =

 1851x1 geopoint vector with properties:

 Collection properties:
     Geometry: 'point'
     Metadata: [1x1 struct]
 Feature properties:
```

```
        Latitude: [1x1851 double]
       Longitude: [1x1851 double]
       Elevation: [1x1851 double]
            Time: {1x1851 cell}
```

To construct a geoshape vector with multiple features, place the coordinates into cell arrays.

```
lat = {trk1.Latitude, trk2.Latitude};
lon = {trk1.Longitude, trk2.Longitude};
```

To create dynamic vertex properties (a property with a value at each coordinate), place the elevation and time values into cell arrays.

```
elevation = {trk1.Elevation, trk2.Elevation};
time = {trk1.Time, trk2.Time};
```

Construct a geoshape vector containing two track log features that include `Elevation` and `Time` as dynamic Vertex properties.

```
tracks = geoshape(lat, lon, 'Elevation', elevation, 'Time', time)

tracks =

 2x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
     Latitude: [1x2592 double]
    Longitude: [1x2592 double]
    Elevation: [1x2592 double]
         Time: {1x2592 cell}
```

Each `Latitude` and `Longitude` coordinate pair has associated `Elevation` and `Time` values which are Vertex properties.

To construct a geoshape vector containing a dynamic Feature property, use an array that is the same length as the coordinate cell array. For example, construct a geoshape vector as before and add a `MaximumElevation` dynamic Feature property.

```
tracks_max = geoshape(lat, lon, 'Elevation', elevation, 'Time', time, ...
    'MaximumElevation', [max(trk1.Elevation) max(trk2.Elevation)])
```

```
tracks_max =

 2x1 geoshape vector with properties:

 Collection properties:
             Geometry: 'line'
             Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
             Latitude: [1x2592 double]
            Longitude: [1x2592 double]
            Elevation: [1x2592 double]
                 Time: {1x2592 cell}
 Feature properties:
    MaximumElevation: [92.4240 76.1000]
```

The Feature property value has only two numeric values, one for each feature.

**Use geoshape Vectors to Store Latitude and Longitude Values**

Load coastline data from a MAT-file.

```
load coastlines
```

Create an N-by-2 array of coastline latitude and longitude values.

```
pts = [coastlat coastlon];
```

Create a geoshape object and store the latitude and longitude data. If you store latitude and longitude coordinate values in an N-by-2 array, geoshape assigns the Latitude property values to the first **column** and the Longitude property values to the second **column**.

```
shape = geoshape();
shape(1) =  pts


shape =

 1x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
```

```
       Latitude: [1x9865 double]
      Longitude: [1x9865 double]
```

This time create a 2-by-M array of coastline latitude and longitude values. Note the semicolon inside the brackets.

```
pts2 = [coastlat; coastlon];
```

Create a geoshape object and store the latitude and longitude data. If you store latitude and longitude coordinate values in a 2-by-M array, `geoshape` assigns the `Latitude` property values to the first **row** and the `Longitude` property values to the second **row**.

```
shape2 = geoshape();
shape2(1) = pts2


shape2 =

 1x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
     Latitude: [1x19730 double]
    Longitude: [1x19730 double]
```

## See Also

mappoint | mapshape | geopoint | gpxread | shaperead

# append

**Class:** geoshape

Append features to geoshape vector

## Syntax

```
outShape = append(inShape,lat,lon)
outShape = append( ___ ,field,value,...)
```

## Description

`outShape = append(inShape,lat,lon)` appends the vector, `lat`, to the `Latitude` property of the geoshape vector, `inShape`, and the vector, `lon`, to the `Longitude` property of `inShape`. `lat` and `lon` are either vectors of class single or double or cell arrays containing numeric arrays of class single or double.

`outShape = append( ___ ,field,value,...)` appends the values specified in `value` to the corresponding dynamic property, `field`. If the property does not exist, `append` adds the dynamic property to the object using the value of `field` for the name and assigning the field the value specified in `value`.

## Input Arguments

**inShape**

geoshape vector

**lat**

Numeric vector of `Latitude` values

**lon**

Numeric vector of `Longitude` values

**field**

Field name, specified as a string. Field can specify the name of an existing property in the geoshape vector or the name you want assigned to a new property that you want to add to the geoshape vector.

**value**

Value you want to assign to the property specified by `field`, specified as a cell array, scalar, or vector of any numeric class or logical.

- When `value` is a cell array, `append` adds the property as a Vertex property.
- When `value` is a numeric array, `append` adds the property as a Feature property.
- When `value` is empty, `append` removes the property.

## Output Arguments

**outShape**

Modified geoshape vector with additional entries in `Latitude` and `Longitude` fields along with any new fields for dynamic properties that you added.

## Examples

### Append Values to Fields

Create a geoshape vector.

```
lat1 = [42, 42.2, 43];
lon1 = [-110, -110.3, -110.5];
temp1 = [65, 65.1, 68];
s = geoshape(lat1,lon1,'Temperature',{temp1})

s =

 1x1 geoshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
```

```
Vertex properties:
        Latitude: [42 42.2000 43]
       Longitude: [-110 -110.3000 -110.5000]
     Temperature: [65 65.1000 68]
```

Append data representing another feature.

```
lat2 = [43, 43.1, 44, 44.1];
lon2 = [-110.1, -111, -111.12, -110.8];
temp2 = [66, 66.1, 68.3, 69];
s = append(s,lat2,lon2,'Temperature', {temp2})

s =

 2x1 geoshape vector with properties:

 Collection properties:
        Geometry: 'line'
        Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
        Latitude: [42 42.2000 43 NaN 43 43.1000 44 44.1000]
       Longitude: [-110 -110.3000 -110.5000 NaN -110.1000 -111 -111.1200 -110.8000]
     Temperature: [65 65.1000 68 NaN 66 66.1000 68.3000 69]
```

Note that the geoshape vector grew from 1-by-1 to 2-by-1 in length.

## See Also

geoshape | geoshape.vertcat

# cat

**Class:** geoshape

Concatenate geoshape vectors

## Syntax

```
s= cat(dim,s1, s2, ...)
```

## Description

`s= cat(dim,s1, s2, ...)` concatenates the geoshape vectors `s1`,`s2` and so on along dimensions `dim`. `dim` must be `1`.

## Input Arguments

**s1, s2, ...**

geoshape vectors to be concatenated.

## Output Arguments

**s**

Concatenated geoshape vector.

## Examples

### Concatenate two geoshape vectors

Create two geoshape vectors and concatenate them into a single vector.

```
s1 = geoshape(42,-110, 'Temperature', 65);
```

```
s2 = geoshape(42.2, -110.5, 'Temperature', 65.6);
s1s2 = cat(1,s1,s2)

s1s2 =

 2x1 geoshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
       Latitude: [42 NaN 42.2000]
      Longitude: [-110 NaN -110.5000]
 Feature properties:
    Temperature: [65 65.6000]
```

## See Also

geoshape.vertcat

# disp

**Class:** geoshape

Display geoshape vector

# Syntax

```
disp(s)
```

# Description

disp(s) prints the size of the geoshape vector, s, and its properties and dynamic properties, if they exist. If the command window is large enough, the values of the properties are also shown, otherwise only their size is shown. You can control the display of the numerical values by using the format command.

# Input Arguments

**s**

geoshape vector.

# Examples

### Display a geoshape vector

Create a geoshape vector and display it.

```
s = geoshape(shaperead('worldcities', 'UseGeo', true));
disp(s)
disp(s(1:2))

 318x1 geoshape vector with properties:

 Collection properties:
```

```
     Geometry: 'point'
     Metadata: [1x1 struct]
Vertex properties:
 (318 features concatenated with 317 delimiters)
     Latitude: [1x635 double]
    Longitude: [1x635 double]
Feature properties:
        Name: {1x318 cell}

2x1 geoshape vector with properties:

Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
Vertex properties:
 (2 features concatenated with 1 delimiter)
    Latitude: [5.2985 NaN 24.6525]
   Longitude: [-3.9509 NaN 54.7589]
Feature properties:
        Name: {'Abidjan' 'Abu Dhabi'}
```

## See Also

geoshape | format

# fieldnames

**Class:** geoshape

Dynamic properties of geoshape vector

## Syntax

```
names = fieldnames(s)
```

## Description

`names = fieldnames(s)` returns the names of the dynamic properties of the geoshape vector, `s`.

## Input Arguments

**s**

geoshape vector for which the properties are to be queried.

## Output Arguments

**names**

Names of the dynamic properties in the geoshape vector `s`

## Examples

### Find dynamic properties

Create a geoshape vector and view the field names.

```
s = geoshape(shaperead('worldcities', 'UseGeo', true));
```

```
fieldnames(s)

ans =

    'Name'
```

## See Also
geoshape.properties

# isempty

**Class:** geoshape

True if geoshape vector is empty

## Syntax

```
tf = isempty(s)
```

## Description

`tf = isempty(s)` returns true if the geoshape vector, `s`, is empty and false otherwise.

## Input Arguments

**s**

geoshape vector.

## Output Arguments

**tf**

`Boolean`. 1 if `s` is empty or 0 if not.

## Examples

### Check if a geoshape vector Is Empty

Create a geoshape vector and check if its empty.

```
s = geoshape();
isempty(s)
```

```
ans =

     1
```

# isfield

**Class:** geoshape

True if dynamic property exists

## Syntax

```
tf = isfield(s,name)
tf = isfield(s,names)
```

## Description

`tf = isfield(s,name)` returns true if the value specified by the string `name` is a dynamic property of the geoshape vector, `s`.

`tf = isfield(s,names)` return true for each element of the cell array, `names`, that is a dynamic property of `s`. `tf` is a logical array of the same size as `names`.

## Input Arguments

**s**

geoshape vector.

**name**

Name of the dynamic property.

**names**

Cell array of names of dynamic properties.

## Output Arguments

**tf**

Boolean. 1 if `s` contains the specified fields or 0 otherwise.

## Examples

### Check if a Field is Present in a geoshape vector

Create a geoshape vector and check is a field is present.

```
s = geoshape(-33.961, 18.484, 'Name', 'Cape Town')
isfield(s, 'Latitude')
isfield(s, 'Name')

s =

 1x1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1x1 struct]
 Vertex properties:
     Latitude: -33.9610
    Longitude: 18.4840
 Feature properties:
         Name: 'Cape Town'


ans =

     0


ans =

     1
```

Note that `Latitude` returns 0 because it is not a dynamically added property.

## See Also

geoshape.isprop | geoshape.fieldnames

# isprop

**Class:** geoshape

True if property exists

## Syntax

```
tf = isprop(s,name)
tf = isprop(s,names)
```

## Description

`tf = isprop(s,name)` returns true if the value specified by the string, `name` is a property of the geoshape vector, `s`.

`tf = isprop(s,names)` returns true for each element of the cell array of strings, `names`, that is a property of `s`. `tf` is a logical array the same size as `names`.

## Input Arguments

**s**

geoshape vector.

**name**

String specifying the property of the geoshape vector, `s`.

**names**

Cell array of strings specifying the property of the geoshape vector, `s`.

## Output Arguments

**tf**

Boolean. 1 if the property exists with `s` ,0 otherwise.

# Examples

### Check if property exists

Create a geoshape vector and check if properties exist.

```
s = geoshape(-33.961, 18.484, 'Name', 'Cape Town');
isprop(s, 'Latitude')
isprop(s, 'Name')

ans =

     1


ans =

     1
```

## See Also

geoshape.isfield | geoshape.properties

# length

**Class:** geoshape

Number of elements in geoshape vector

## Syntax

```
N = length(s)
```

## Description

`N = length(s)` returns the number of elements contained in the geoshape vector, `s`. The result is equivalent to `size(s,1)`.

## Input Arguments

**s**

geoshape vector.

## Output Arguments

**N**

Length of the geoshape vector, `s`.

## Examples

### Find the Length of the geoshape vector

Create a geoshape vector and check if properties exist.

```
coast = load('coast');
```

```
s = geoshape(coast.lat, coast.long);
length(s)
length(coast.lat)


ans =

     1


ans =

       9865
```

## See Also

geoshape.size

# properties

**Class:** geoshape

Properties of a geoshape vector

## Syntax

```
prop = properties(s)
properties(s)
```

## Description

`prop = properties(s)` returns a cell of the property names of the geoshape vector, `s`.

`properties(s)` displays the names of the properties of `s`.

## Input Arguments

**s**

geoshape vector.

## Output Arguments

**prop**

Cell variable consisting of property names of the geoshape vector, `s`.

## Examples

### View Properties of a geoshape vector

Create a geoshape vector and view its properties.

```
s = geoshape(shaperead('tsunamis', 'UseGeo', true));
properties(s)

Properties for class geoshape:

    Geometry
    Metadata
    Latitude
    Longitude
    Year
    Month
    Day
    Hour
    Minute
    Second
    Val_Code
    Validity
    Cause_Code
    Cause
    Eq_Mag
    Country
    Location
    Max_Height
    Iida_Mag
    Intensity
    Num_Deaths
    Desc_Deaths
```

## See Also

geoshape.fieldnames

# rmfield

**Class:** geoshape

Remove dynamic property from geoshape vector

## Syntax

```
s = rmfield(s, fieldname)
s = rmfield(s, fields)
```

## Description

`s = rmfield(s, fieldname)` removes the field specified by the string, `fieldname`, from the geoshape vector, `s`.

`s = rmfield(s, fields)` removes all the fields specified by the cell array, `fields`.

---

**Note:** `rmfield` cannot remove `Latitude`, `Longitude`, `Metadata` and `Geometry` fields. The specified string, '`fieldname`, is case sensitive.

---

## Input Arguments

**s**

geoshape vector.

**fieldname**

Exact string representing the name of the property.

**fields**

Cell array of strings specifying the names of the properties.

## Output Arguments

**s**

Updated geoshape vector with the field(s) removed.

## Examples

### Remove fields from a geoshape vector

Create a geoshape vector and remove a field.

```
s = geoshape(shaperead('tsunamis', 'UseGeo', true));
tf = isfield(s,'Intensity')
s2 = rmfield(s,'Intensity');
tf = isfield(s2,'Intensity')

tf =

     1

tf =

     0
```

## See Also

geoshape.fieldnames | geoshape.rmprop

# rmprop

**Class:** geoshape

Remove properties from geoshape vector

## Syntax

```
sf = rmprop(s,propname)
sf = rmprop(s,propnames)
```

## Description

`sf = rmprop(s,propname)` removes the property specified by the string, `propname` from the geoshape vector, `s`.

`sf = rmprop(s,propnames)` removes all the properties specified in the cell array, `propnames`, from the geoshape vector, `s`. If `propnames` contains a coordinate property an error is issued.

**Note:** `rmprop` cannot remove `Latitude`, `Longitude`, `Metadata` and `Geometry` fields. The specified string, `propname`, is case sensitive.

## Input Arguments

**s**

geoshape vector.

## Output Arguments

**sf**

Modified geoshape vector with the specified property(s) removed.

# Examples

### Remove a property from a geoshape vector

Create a geoshape vector and remove a property.

```
s = geoshape(shaperead('tsunamis', 'UseGeo', true));
tf = isfield(s,'Validity')
s2 = rmprop(s, 'Validity');
tf = isfield(s2,'Validity')

tf =

     1

tf =

     0
```

## See Also

geoshape.fieldnames

# size

**Class:** geoshape

Size of `geoshape` vector

## Syntax

```
val = size(s)
val = size(s,1)
val = size(s, n)
[m,k] = size(s)
```

## Description

`val = size(s)` returns the vector `[length(val), 1]`.

`val = size(s,1)` returns the length of `s`.

`val = size(s, n)` returns 1 for `n >= 2`.

`[m,k] = size(s)` returns `length(s)` for `m` and 1 for `k`.

## Input Arguments

**s**

geoshape vector.

**n**

Number of the dimension at which size of `s` is required.

## Output Arguments

**val**

Vector of the form `[length(s), 1]`.

**m**

Length of `s`.

**k**

Length of second dimension of `s`. `k` is always `1`.

## Examples

### Find the Size of a geoshape vector

Create a structure, construct a geoshape vector, and get its size.

```
structArray = shaperead('worldrivers', 'UseGeoCoords', true);
s = geoshape(structArray);
structSize = size(structArray)
sSize = size(s)

structSize =

   128     1


sSize =

   128     1
```

The second dimension is always 1.

## See Also

geoshape.length | `size`

# struct

**Class:** geoshape

Convert geoshape vector to scalar structure

## Syntax

```
structArray = struct(s)
```

## Description

`structArray = struct(s)` converts the geoshape vector, `s`, to a scalar structure array, `structArray`.

## Input Arguments

**s**

geoshape vector.

## Output Arguments

**structArray**

Scalar structure of the geoshape vector `s`.

## Examples

### Convert a geoshape vector into Struct

Create a geoshape vector and convert it into a struct.

```
structArray = shaperead('worldcities', 'UseGeo', true)
```

```
s= geoshape(structArray);
structArray2 = struct(s)


structArray =

318x1 struct array with fields:
    Geometry
    Lon
    Lat
    Name


structArray2 =

     Geometry: 'point'
     Metadata: [1x1 struct]
     Latitude: [1x635 double]
    Longitude: [1x635 double]
         Name: {1x318 cell}
```

## See Also

geoshape.properties

# vertcat

**Class:** geoshape

Vertical concatenation for geoshape vectors

## Syntax

```
s = vertcat(s1,s2, ...)
```

## Description

`s = vertcat(s1,s2, ...)` vertically concatenates the geoshape vector, `s1`, `s2`, and so on. If the class type of any property is a cell array, then the resultant field in the output `s` will also be a cell array.

## Input Arguments

**s1, s2, ...**

geoshape vectors to be concatenated.

## Output Arguments

**s**

Concatenated geoshape vector.

## Examples

### Concatenate geoshape vectors

Create several geoshape vectors and concatenate them.

```
s1 = geoshape(42, -110, 'Temperature', 65, 'Name', 'point1');
s2 = geoshape( 42.1, -110.4, 'Temperature', 65.5, 'Name', 'point2');
s = vertcat(s1, s2)

s =

 2x1 geoshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
       Latitude: [42 NaN 42.1000]
      Longitude: [-110 NaN -110.4000]
 Feature properties:
    Temperature: [65 65.5000]
           Name: {'point1' 'point2'}
```

## See Also

geopoint.cat

# geoshow

Display map latitude and longitude data

## Syntax

```
geoshow(lat,lon)
geoshow(S)
geoshow(lat,lon,Z)
geoshow(Z,R)
geoshow(lat,lon,image)
geoshow(lat,lon,A,cmap)
geoshow(image,R)
geoshow(A,cmap,R)
geoshow(ax,...)
geoshow('Parent',ax,...)
geoshow(filename)
geoshow( ___ ,Name,Value,...)
h = geoshow( ___ )
```

## Description

geoshow(lat,lon) projects and displays the latitude and longitude vectors lat and lon using the projection stored in the axes. If there is no projection, lat and lon are projected using a default Plate Carrée projection. The default behavior for geoshow is to display lat and lon as lines.

geoshow(S) displays the vector geographic features stored in S as points, multipoints, lines, or polygons according to the 'Geometry' field of S. If S is either a geopoint vector, a geoshape vector, or a geostruct (with 'Lat' and 'Lon' coordinate fields), geoshow projects vertices to map coordinates. If S is either a mappoint vector, mapshape vector, or a mapstruct (with 'X' and 'Y' fields), geoshow plots vertices as (pre-projected) map coordinates and issues a warning.

geoshow(lat,lon,Z) projects and displays a geolocated data grid.

geoshow(Z,R) projects and displays a regular data grid, Z.

geoshow(lat,lon,image) or geoshow(lat,lon,A,cmap) projects and displays a geolocated image as a texturemap on a zero-elevation surface. lat, lon, and the image array must match in size. Examples of geolocated images include a color composite from a satellite swath or an image originally referenced to a different coordinate system.

geoshow(image,R) or geoshow(A,cmap,R) projects and displays an image georeferenced to latitude-longitude through the referencing matrix R. The image is shown as a texturemap on a zero-elevation surface.

geoshow(ax,...) and geoshow('Parent',ax,...) set the parent axes to ax.

geoshow(filename) projects and displays data from the file specified according to the type of file format.

geoshow( ___ ,Name,Value,...) specifies parameters and corresponding values that modify the type of display or set MATLAB graphics properties. Parameter names can be abbreviated, and case does not matter.

h = geoshow( ___ ) returns a handle to a MATLAB graphics object.

## Input Arguments

**lat**

Latitude vector. The vectors may contain embedded NaNs to delimit individual lines or polygon parts. lat must be of equal length to lon.

**lon**

Longitude vector. The vectors may contain embedded NaNs to delimit individual lines or polygon parts. lon must be of equal length to lat.

**Z**

M-by-N array. May contain NaN values.

**S**

Geographic data structure or dynamic vector

**image**

Grayscale, logical, or truecolor image.

**A**

Indexed image.

**cmap**

Colormap.

**R**

Geographic raster reference object, referencing vector, or referencing matrix. If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

`[cells/degree northern_latitude_limit western_longitude_limit]`

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

`[lon lat] = [row col 1] * R`

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

If `R` is a geographic raster reference object with raster interpretation `'postings'`, then the `'image'` and `'texturemap'` display types are not accepted.

**ax**

Axes object.

**filename**

Name of file.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**`'SymbolSpec'`**

A structure returned by `makesymbolspec` that specifies the symbolization rules to be used for displaying vector data. It is used only for vector data stored in geographic data structures. In cases where both `SymbolSpec` and one or more graphics properties are specified, the graphics properties override any settings in the symbolspec structure.

To change the default symbolization rule for a Name,Value pair in the symbolspec, prefix the word `'Default'` to the graphics property name.

**`'DisplayType'`**

Type of graphic display for the data. You can set any MATLAB Graphics line, patch, and surface properties. You can also set any Mapping Toolbox contour properties. See the table for links to the documentation on these properties.

| Data Type | DisplayType | Type of Property |
|---|---|---|
| Vector | `'point'` | *line marker* |
| | `'multipoint'` | *line marker* |
| | `'line'` (default) | *line* |
| | `'polygon'` | *patch* |
| Image | `'image'` | *surface* |
| Grid | `'surface'` | *surface* |
| | `'mesh'` | *surface* |
| | `'texturemap'` | *surface* |
| | `'contour'` | *contour* |

If `DisplayType` is `'texturemap'`, `geoshow` constructs a surface with `ZData` values set to 0.

When using the `filename` argument, the `DisplayType` parameter is automatically set, according to the following table:

| Format | DisplayType |
|---|---|
| Shape file | `'point'`, `'line'`, or `'polygon'` |

| Format | DisplayType |
|---|---|
| GeoTIFF | `'image'` |
| TIFF/JPEG/PNG with a world file | `'image'` |
| ARC ASCII GRID | `'surface'` (can be overridden) |
| SDTS raster | `'surface'` (can be overridden) |

## Output Arguments

**h**

Handle to a MATLAB graphics object or, in the case of polygons, a modified patch object. If a geostruct or shapefile name is input, geoshow returns the handle to an hggroup object with one child per feature in the geostruct or shapefile, excluding any features that are completely trimmed away. In the case of a polygon geostruct or shapefile, each child is a modified patch object; otherwise it is a line object.

## Class Support

| Display Type | Supported Class Types |
|---|---|
| Image | `logical`, `uint8`, `uint16`, and `double` |
| Surface | `single` and `double` |
| Texture map | All numeric types and `logical` |

## Examples

### Display World Land Areas Using Plate Carree Projection

Display map of the world using default Plate Carree projection.

```
figure
geoshow('landareas.shp', 'FaceColor', [0.5 1.0 0.5]);
```
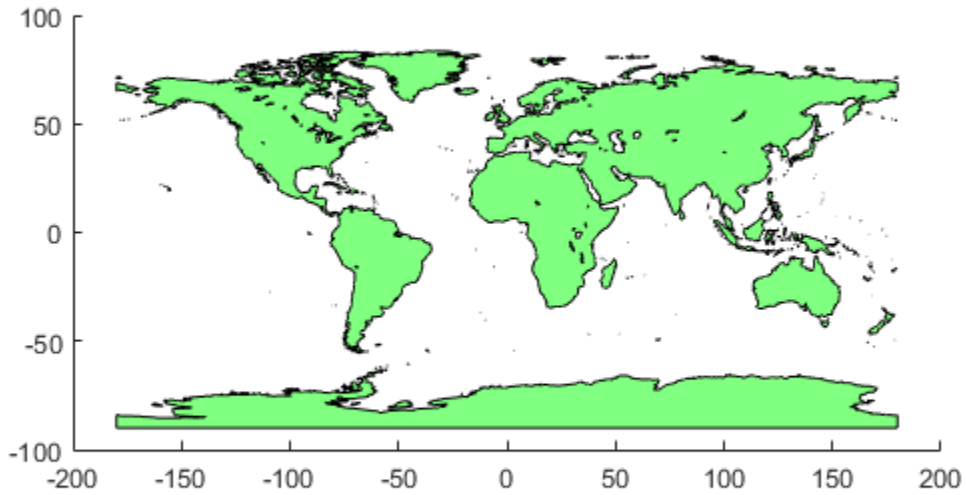
**Override Symbolspec Default Rule**

Create a map of North America.

```
figure
worldmap('na');
```
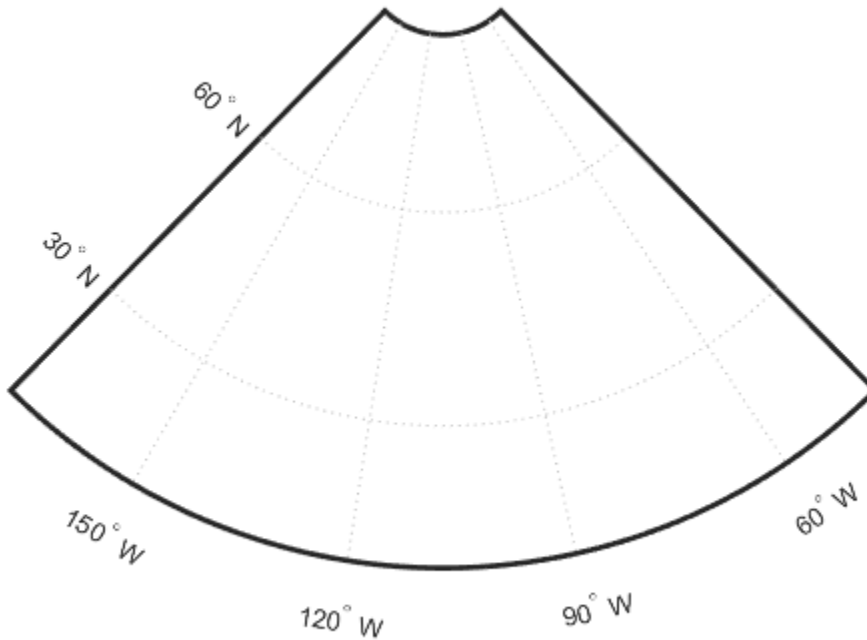
Read the USA high resolution data.

```
states = shaperead('usastatehi', 'UseGeoCoords', true);
```

Create a symbolspec that sets the color of Alaska and Hawaii polygons red.

```
symspec = makesymbolspec('Polygon', ...
   {'Name', 'Alaska', 'FaceColor', 'red'}, ...
   {'Name', 'Hawaii', 'FaceColor', 'red'});
```

Display all the other states in blue.

```
geoshow(states, 'SymbolSpec', symspec, ...
   'DefaultFaceColor', 'blue', ...
   'DefaultEdgeColor', 'black');
```
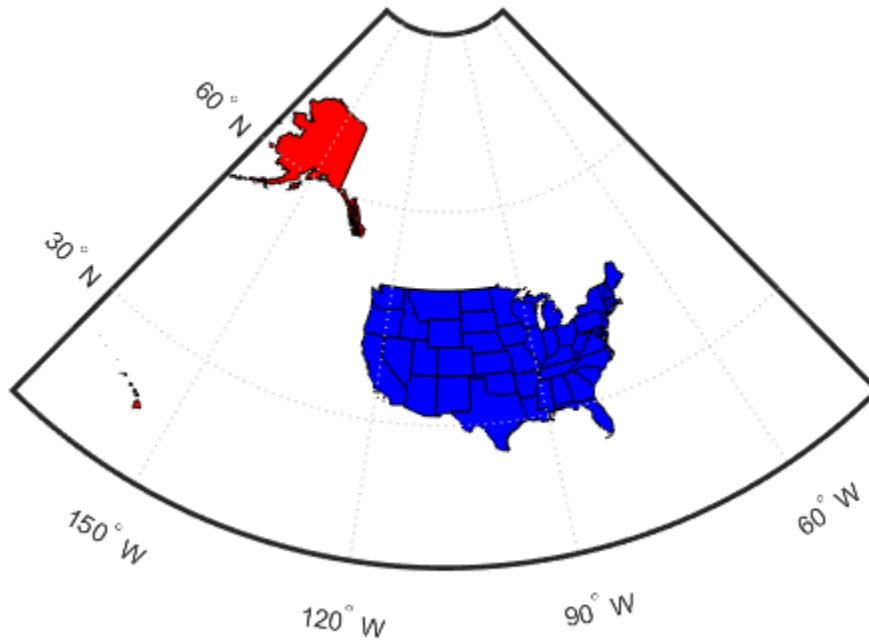
### Create Map of Korea and Display Data Grid as Texture Map

Display a map, using the `geoshow` function with the `'DisplayType'` parameter set to `'TextureMap'`.

Load MAT file containing map information about Korean peninsula.

```
load korea
```

Set up an empty map axes with projection and limits for the Korean peninsula.

```
figure;
worldmap(map, refvec)
```
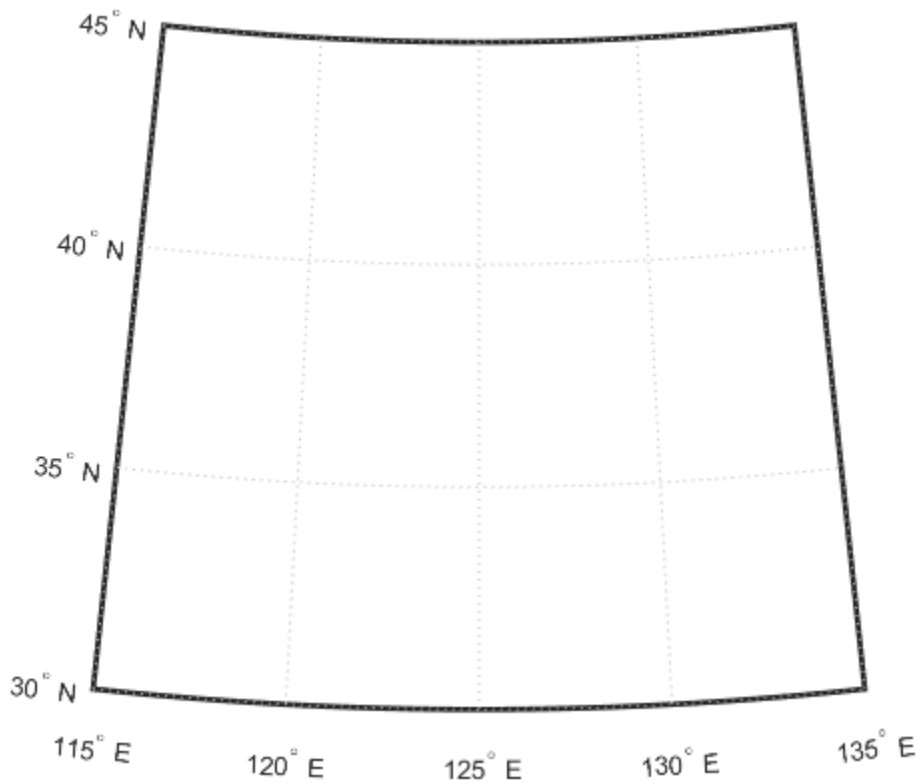
Display the Korean data grid as a texture map.

```
geoshow(gca,map,refvec,'DisplayType','texturemap');
demcmap(map)
```
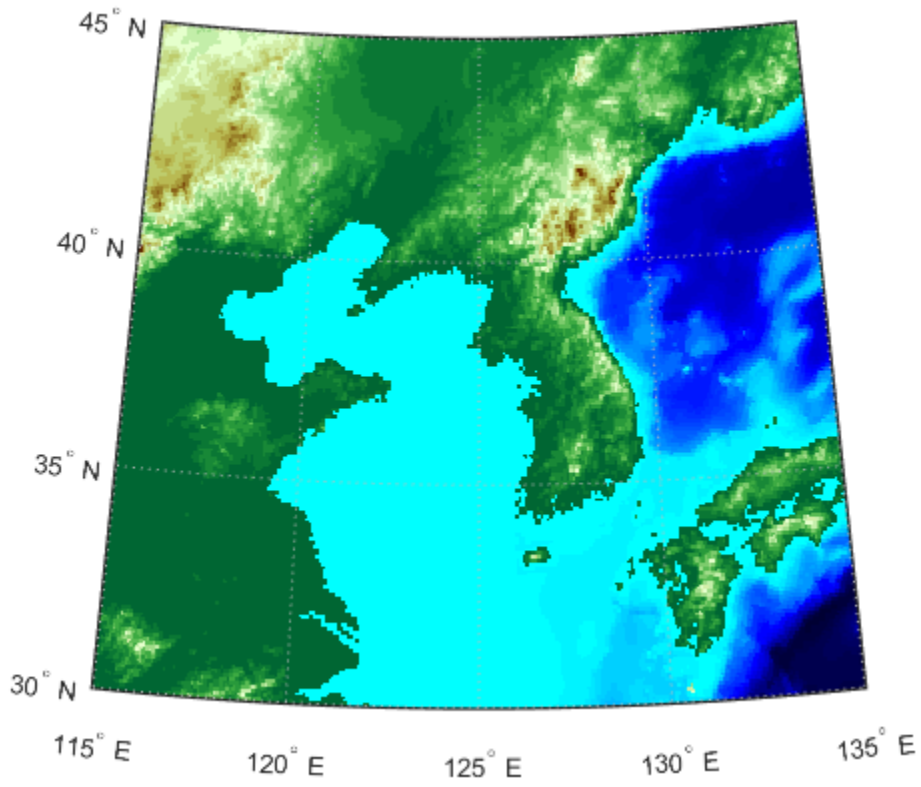
Display the land area boundary as a black line.

```
S = shaperead('landareas','UseGeoCoords',true);
geoshow([S.Lat], [S.Lon],'Color','black');
```
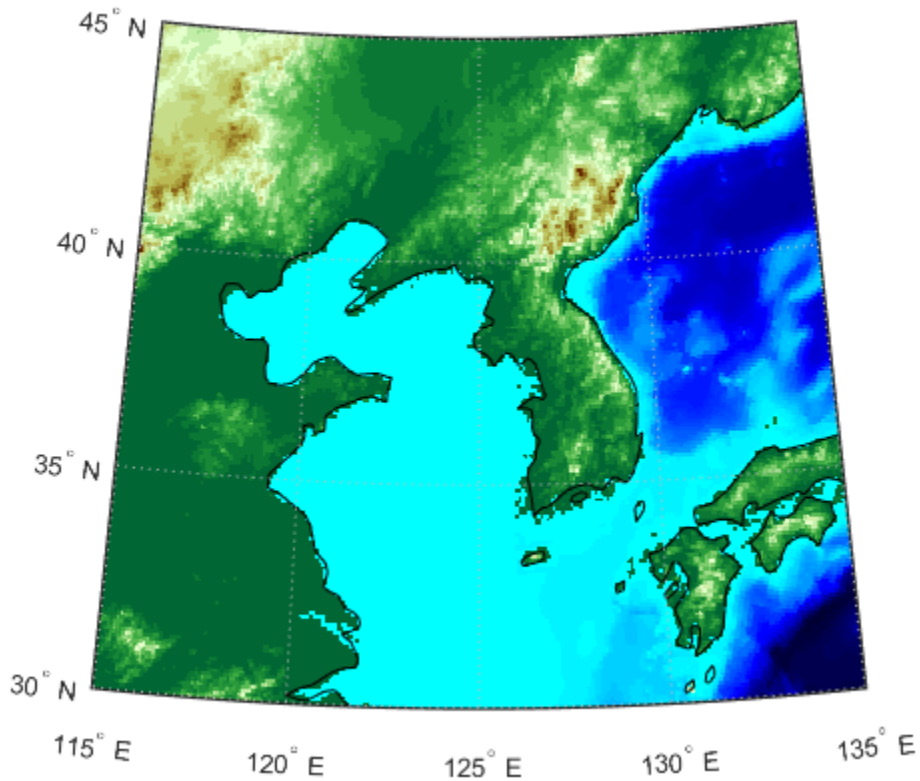
**Create Map of Korea and Display NaNs as Transparent**

Display a map, interpreting NaNs as transparent. This example uses the `geoshow` function with the `'DisplayType'` parameter set to `'surface'`.

Load map of Korean peninsula.

```
korea = load('korea');
Z = korea.map;
```

Insert an band of null values using NaNs.

```
Z(80:100,:) = NaN;
```

Create a raster reference object.

```
R = georasterref('RasterSize',size(Z),...
    'Latlim',[30 45], 'Lonlim', [115 135]);
```

Display world map with NaN values displayed as transparent. Note that the settings for ZData and CData ensure that the resulting surface falls in the z == 0 plane, instead of being 3-dimensional.

```
figure;
worldmap(Z,R)
geoshow(Z,R,'DisplayType','surface','ZData',zeros(size(Z)),'CData',Z)
demcmap(Z)
```
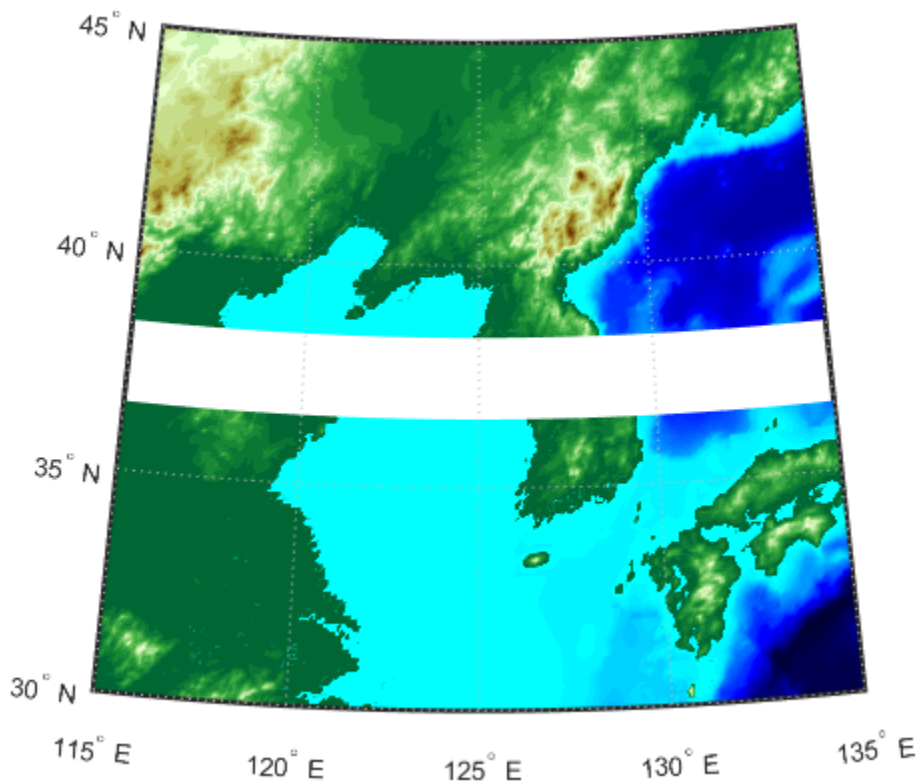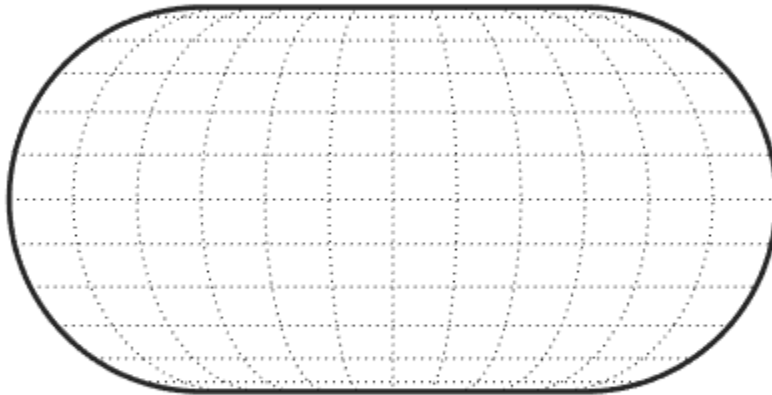


### Display EGM96 Geoid Heights Masking Out Land Areas

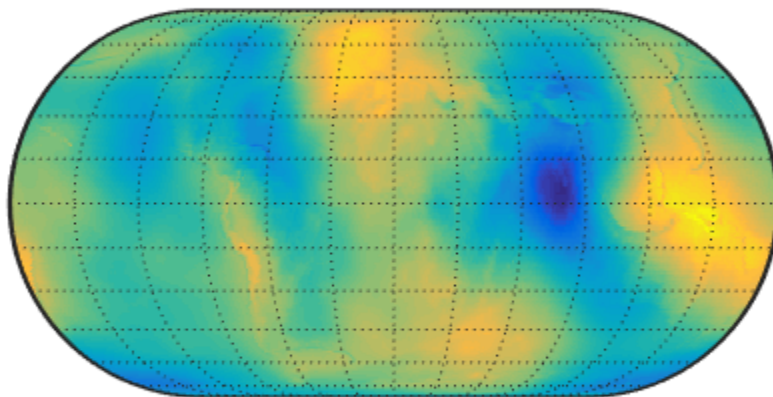Load geoid and display.

```
load geoid
```

Create a figure with an Eckert projection.

```
figure
axesm eckert4;
framem; gridm;
axis off
```
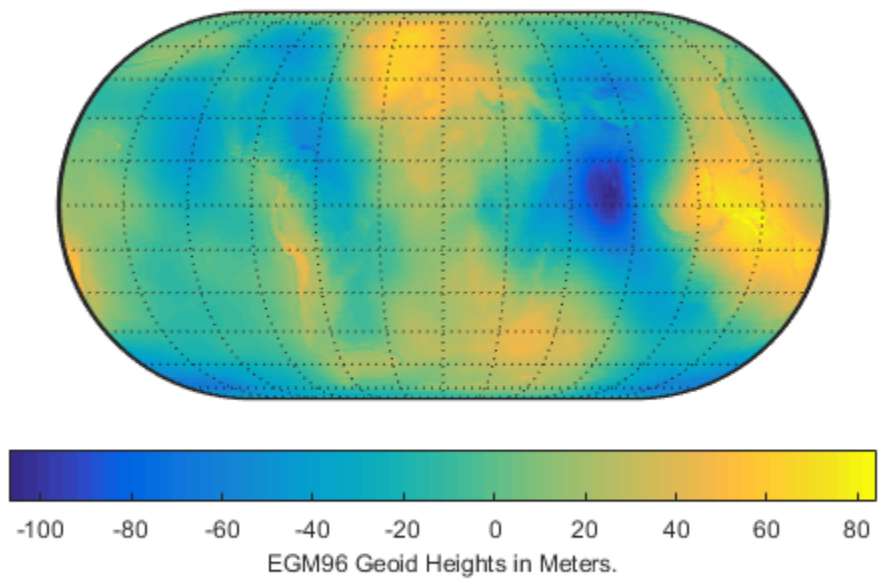


Display the geoid as a texture map.

```
geoshow(geoid, geoidrefvec, 'DisplayType', 'texturemap');
```
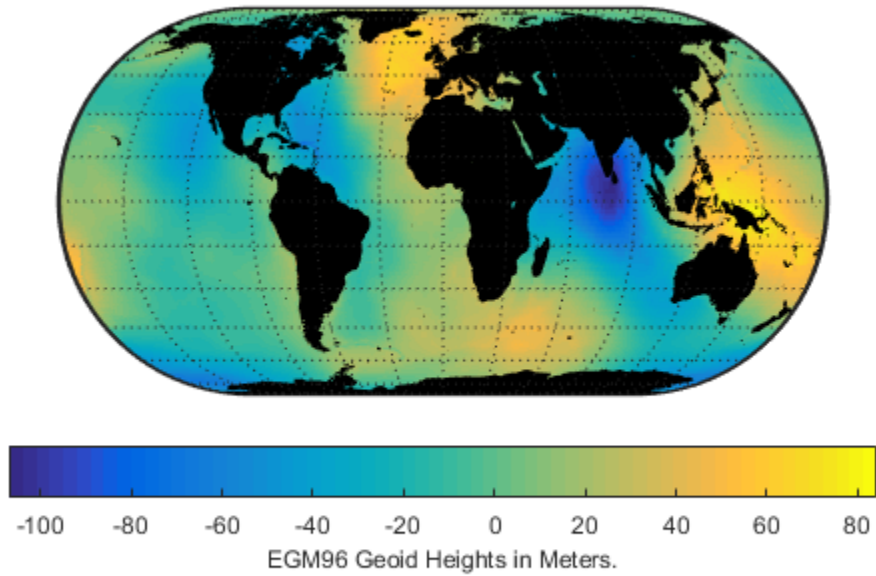
Create a colorbar and title.

```
hcb = colorbar('southoutside');
set(get(hcb,'Xlabel'),'String','EGM96 Geoid Heights in Meters.')
```

EGM96 Geoid Heights in Meters.

Mask out all the land.

```
geoshow('landareas.shp', 'FaceColor', 'black');
```

EGM96 Geoid Heights in Meters.

### Display the EGM96 Geoid Heights as a 3-D Surface Using the Eckert IV Projection

Load geoid.

```
load geoid

% Create the figure with an Eckert projection.
figure
axesm eckert4;
axis off

% Display the geoid as a surface.
h = geoshow(geoid, geoidrefvec, 'DisplayType','surface');
```
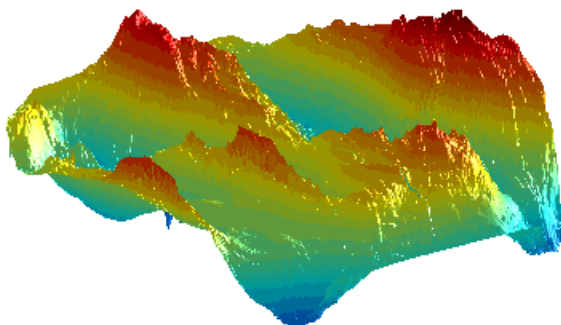
```
% Add light and material.
light; material(0.6*[ 1 1 1]);

% View as a 3-D surface.
view(3)
axis normal
tightmap
```
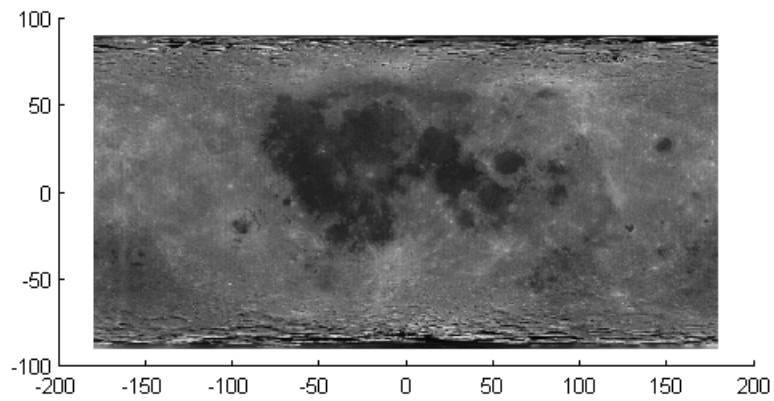


### Display the Moon Albedo Image Projected Using Plate Carree and in an Orthographic Projection

Display using Plate Carree.

```
load moonalb
```

```
% Projection not specified. Uses Plate Carree.
figure
geoshow(moonalb,moonalbrefvec)
```

View orthographic projection.

```
figure
axesm ortho
geoshow(moonalb, moonalbrefvec, 'DisplayType', 'texturemap')
colormap(gray(256))
axis off
```

### Read and Display the San Francisco South 24K DEM Data

Read every point of the 1:24,000 DEM file.

```
filenames = gunzip('sanfranciscos.dem.gz', tempdir);
demFilename = filenames{1};

[lat, lon,Z] = usgs24kdem(demFilename,2);

% Delete the temporary gunzipped file.
delete(demFilename);

% Move all points at sea level to -1 to color them blue.
Z(Z==0) = -1;

% Compute the latitude and longitude limits for the DEM.
latlim = [min(lat(:)) max(lat(:))];
lonlim = [min(lon(:)) max(lon(:))];

% Display the DEM values as a texture map.
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, 'DisplayType','texturemap')
```
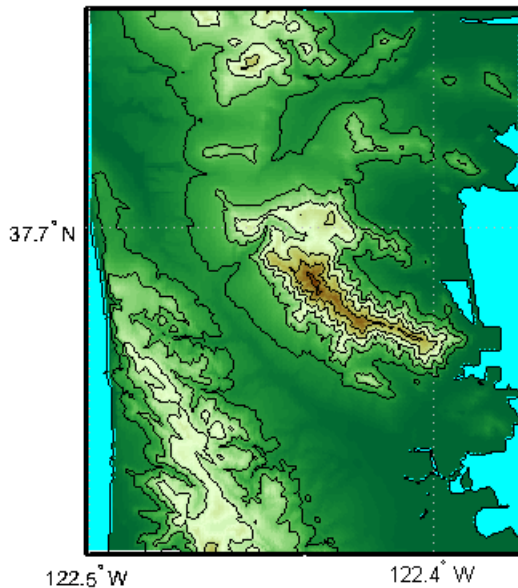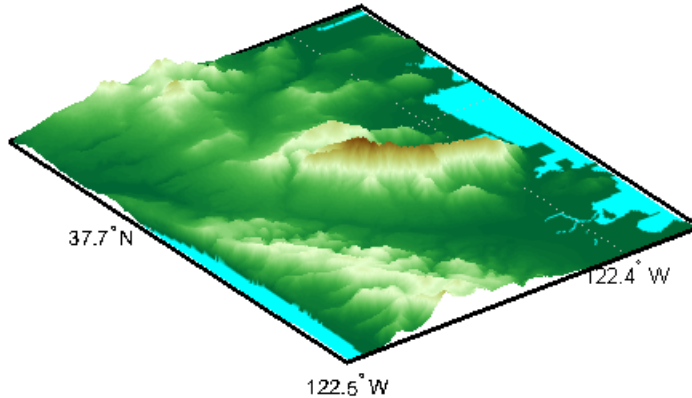
```
demcmap(Z)
daspectm('m',1)

% Overlay black contour lines onto the texturemap.
geoshow(lat, lon, Z, 'DisplayType', 'contour', ...
  'LineColor', 'black');
```



View the DEM values in 3-D.

```
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, 'DisplayType', 'surface')
demcmap(Z)
daspectm('m',1)
view(3)
```

## More About

**Tips**

- When calling `shaperead` to read files that contain coordinates in latitude and longitude, be sure to specify the `shaperead` argument pair `'UseGeoCoords'`,`true`. If you do not include this argument, `shaperead` will create a mapstruct, with coordinate fields labelled X and Y instead of `Lon` and `Lat`. In such cases, `geoshow` assumes that the geostruct is in fact a mapstruct containing projected coordinates, warns, and calls `mapshow` to display the geostruct data without projecting it.

- If you do not want `geoshow` to draw on top of an existing map, create a new figure or subplot before calling it.

- When you display vector data in a map axes using `geoshow`, you should not subsequently change the map projection using `setm`. You can, however, change the projection with `setm` for raster data. For more information, see "Changing Map Projections when Using geoshow".

- If you display a polygon, do not set `'EdgeColor'` to either `'flat'` or `'interp'`. This combination may result in a warning.

- When projecting data onto a map axes, `geoshow` uses the projection stored with the map axes. When displaying on a regular axes, it constructs a default Plate Carrée

projection with a scale factor of `180/pi`, enabling direct readout of coordinates in degrees.

- `geoshow` can generally be substituted for `displaym`. However, there are limitations where display of specific objects is concerned. See the remarks under `updategeostruct` for further information.

When you display raster data in a map using `geoshow`, columns near the eastern or western edge may fail to display. This is seldom noticeable, except when the raster is very coarse relative to the displayed area. To include additional columns in the display, it might help to:

- Resize the grid to a finer mesh
- Make sure the cell boundaries and map limits align
- Expand the map limits

## See Also

map.rasterref.GeographicCellsReference | map.rasterref.GeographicPostingsReference | `axesm` | `makesymbolspec` | `mapshow` | `mapview` | `updategeostruct`

# map.geotiff.RPCCoefficientTag class

**Package:** map.geotiff

Rational Polynomial Coefficients Tag

## Description

RPCCoefficientTag is a class used by the functions `geotiffinfo` and `geotiffwrite` to contain the Rational Polynomial Coefficients (RPC) TIFF tag. This tag describes the relationship of latitude, longitude, and height locations with row and column locations in an image. The class contains properties with names and permissible values corresponding to the tag elements listed in the technical note *RPCs in GeoTIFF* technical note, viewable at http://geotiff.maptools.org/rpc_prop.html.

## Construction

`rpctag = map.geotiff.RPCCoefficientTag` constructs a default `map.geotiff.RPCCoefficientTag` object. For an example, see "Create RPCCoefficientTag Object with Default Properties" on page 1-556.

`rpctag = map.geotiff.RPCCoefficientTag(tiffTagValue)` constructs a `map.geotiff.RPCCoefficientTag` object and sets the properties values to the corresponding values in the 92-element vector specified in `tiffTagValue`. For an example, see "Write Raw RPC Coefficient Metadata to GeoTIFF File" on page 1-556.

### Input Arguments

**`tiffTagValue` — Values of RPCCoefficientTag properties**
92-element vector

Value of RPCCoefficientTag properties, specified as a 92-element vector of class `double`. For an example, see "Write Raw RPC Coefficient Metadata to GeoTIFF File" on page 1-556.

Example: `newRpctag = map.geotiff.RPCCoefficientTag(tiffTagValue);`

Data Types: double

# Properties

### `BiasErrorInMeters` — Root mean square bias error in meters per horizontal axis
-1 (default) | scalar

Root mean square bias error in meters per horizontal axis, specified as a scalar of class `double`. The value is a positive number, 0, or -1 if unknown.

Data Types: double

### `RandomErrorInMeters` — Root mean square random error in meters per horizontal axis
-1 (default) | scalar

Root mean square random error in meters, specified as a scalar of class `double`. The value is a positive number, 0, or -1 if unknown.

Data Types: double

### `LineOffset` — Line offset in pixels
0 (default) | scalar

Line offset in pixels, specified as a scalar of class `double`. The value is a positive number or 0.

Data Types: double

### `SampleOffset` — Sample offset in pixels
0 (default) | scalar

Sample offset in pixels, specified as a scalar of class `double`. The value is a positive number or 0.

Data Types: double

### `GeodeticLatitudeOffset` — Geodetic latitude offset in degrees
0 (default) | scalar

Geodetic latitude offset in degrees, specified as a scalar of class `double`. The value can range from `-90 <= value <= 90`.

Data Types: double

### GeodeticLongitudeOffset — Geodetic longitude offset in degrees

0 (default) | scalar

Geodetic longitude offset in degrees, specified as a scalar of class `double`. The value can range from `-180 <= value <= 180`.

Data Types: `double`

### GeodeticHeightOffset — Geodetic height offset in meters

0 (default) | scalar

Geodetic height offset in meters, specified as a scalar of class `double`.

Data Types: `double`

### LineScale — Line scale factor in pixels

1 (default) | scalar

Line scale factor in pixels, specified as a scalar of class `double`. The value is a positive number.

Data Types: `double`

### SampleScale — Sample scale factor in pixels

1 (default) | scalar

Sample scale factor in pixels, specified as a scalar of class `double`. The value is a positive number.

Data Types: `double`

### GeodeticLatitudeScale — Geodetic latitude scale in degrees

1 (default) | scalar

Geodetic latitude scale in degrees, specified as a scalar of class `double`. The value is a positive number `<= 90`.

Data Types: `double`

### GeodeticLongitudeScale — Geodetic longitude scale in degrees

1 (default) | scalar

Geodetic longitude scale in degrees, specified as a scalar of class `double`. The value is a positive number `<= 180`.

Data Types: `double`

### `GeodeticHeightScale` — Geodetic height scale factor in meters
1 (default) | scalar

Geodetic height scale factor in meters, specified as a scalar of class `double`. The value is a positive number.

Data Types: `double`

### `LineNumeratorCoefficients` — Coefficients for the polynomial in the numerator of the $r(n)$ equation
20-element row vector of zeros (default) | 20-element row vector

Coefficients for the polynomial in the numerator of the $r(n)$ equation, specified as a 20-element row vector of class `double`.

Data Types: `double`

### `LineDenominatorCoefficients` — Coefficients for the polynomial in the denominator of the $r(n)$ equation
20-element row vector of zeros (default) | 20-element row vector

Coefficients for the polynomial in the denominator of the $r(n)$ equation, specified as a 20-element row vector of class `double`.

Data Types: `double`

### `SampleNumeratorCoefficents` — Coefficients for the polynomial in the numerator of the $c(n)$ equation
20-element row vector of zeros (default) | 20-element row vector

Coefficients for the polynomial in the numerator of the $c(n)$ equation, specified as a 20-element row vector of class `double`.

Data Types: `double`

### `SampleDenominatorCoefficients` — Coefficients for the polynomial in the denominator of the $c(n)$ equation
20-element row vector of zeros (default) | 20-element row vector

Coefficients for the polynomial in the denominator of the $c(n)$ equation, specified as a 20-element row vector of class `double`.

Data Types: `double`

## Methods

| | |
|---|---|
| double | Convert property values to row vector of doubles |

## Examples

### Create RPCCoefficientTag Object with Default Properties

Call the RPCCoefficientTag class constructor with no arguments.

```
rpctag = map.geotiff.RPCCoefficientTag


rpctag =

  RPCCoefficientTag with properties:

                 BiasErrorInMeters: -1
               RandomErrorInMeters: -1
                        LineOffset: 0
                      SampleOffset: 0
             GeodeticLatitudeOffset: 0
            GeodeticLongitudeOffset: 0
               GeodeticHeightOffset: 0
                         LineScale: 1
                       SampleScale: 1
             GeodeticLatitudeScale: 1
            GeodeticLongitudeScale: 1
               GeodeticHeightScale: 1
         LineNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       LineDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       SampleNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     SampleDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

### Write Raw RPC Coefficient Metadata to GeoTIFF File

This example shows how to write RPC coefficient metadata to a TIFF file. In a real workflow, you would create the RPC coefficient metadata according to the TIFF extension specification. This example does not show the specifics of how to create valid

RPC metadata. To simulate raw RPC metadata, the example creates a sample TIFF file with RPC metadata and then uses `imfinfo` to read this RPC metadata in raw, unprocessed form from the file. The example then writes this raw RPC metadata to a file using the `geotiffwrite` function.

### Create Raw RPC Coeffient Metadata

To simulate raw RPC metadata, create a simple test file and write some RPC metadata to the file. For this test file, create a toy image and a referencing object associated with the image.

```
myimage = zeros(180,360);
latlim = [-90 90];
lonlim = [-180 180];
R = georefcells(latlim,lonlim,size(myimage));
```

Create an RPCCoefficientTag metadata object and set some of the fields. The toolbox uses the RPCCoefficientTag object to represent RPC metadata in human readable form.

```
rpctag = map.geotiff.RPCCoefficientTag;
rpctag.LineOffset = 1;
rpctag.SampleOffset = 1;
rpctag.LineScale = 2;
rpctag.SampleScale = 2;
rpctag.GeodeticHeightScale = 500;
```

Write the image, the associated referencing object, and the RPCCoeffcentTag object to a file.

```
geotiffwrite('myfile',myimage,R,'RPCCoefficientTag',rpctag)
```

### Read Raw RPC Coefficient Metadata

Read the RPC coefficient metadata from the test file using the `imfinfo` function. When it encounters unfamiliar metadata, `imfinfo` returns the data, unprocessed, in the `UnknownTags` field. Note how the UnknownTags field contains an array of 92 doubles. This is the raw RPC coefficient metadata, read from the file in unprocessed form.

```
info = imfinfo('myfile.tif');
info.UnknownTags
```

ans =

```
        ID: 50844
    Offset: 10672
     Value: [1x92 double]
```

### Write Raw RPC Metadata to a File

Write the raw RPC metadata to a file. First, extract the RPC coefficient metadata from the info structure.

```
value = info.UnknownTags.Value;
```

Then, construct an RPCCoefficientTag object, passing the raw RPC metadata (array of 92 doubles) as an argument.

```
rpcdata = map.geotiff.RPCCoefficientTag(value)
```

```
rpcdata =

  RPCCoefficientTag with properties:

                BiasErrorInMeters: -1
              RandomErrorInMeters: -1
                       LineOffset: 1
                     SampleOffset: 1
            GeodeticLatitudeOffset: 0
           GeodeticLongitudeOffset: 0
              GeodeticHeightOffset: 0
                        LineScale: 2
                      SampleScale: 2
             GeodeticLatitudeScale: 1
            GeodeticLongitudeScale: 1
               GeodeticHeightScale: 500
         LineNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       LineDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       SampleNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     SampleDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Pass the RPCCoefficientTag object to the `geotiffwrite` function and write the RPC metadata to a file.

```
geotiffwrite('myfile2',myimage,R,'RPCCoefficientTag',rpcdata)
```

To verify that the data was written to the file, read the RPC metadata from the TIFF file using `geotiffinfo`. Compare the returned RPC metadata with the metadata written to the test file.

```
ginfo = geotiffinfo('myfile2');
ginfo.GeoTIFFTags.RPCCoefficientTag
```

```
ans =

  RPCCoefficientTag with properties:

                 BiasErrorInMeters: -1
               RandomErrorInMeters: -1
                        LineOffset: 1
                      SampleOffset: 1
            GeodeticLatitudeOffset: 0
           GeodeticLongitudeOffset: 0
              GeodeticHeightOffset: 0
                         LineScale: 2
                       SampleScale: 2
             GeodeticLatitudeScale: 1
            GeodeticLongitudeScale: 1
               GeodeticHeightScale: 500
         LineNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       LineDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       SampleNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     SampleDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

# References

## See Also
geotiffinfo | geotiffwrite

## More About
- Class Attributes
- Property Attributes

**Introduced in R2015b**

# double

**Class:** map.geotiff.RPCCoefficientTag
**Package:** map.geotiff

Convert property values to row vector of doubles

## Syntax

```
tiffTagValue = double(rpctag)
```

## Description

`tiffTagValue = double(rpctag)` returns a 92-element row vector of class `double`, representing the values of the TIFF tag. This is the format required to write the property values to a TIFF file.

## Input Arguments

**`rpctag` — RPCCoefficientTag properties**
RPCCoefficientTag object

`RPCCoefficientTag` properties, specified as an `RPCCoefficientTag` object.

Example: `tifftagvalues = double(rpctag);`

## Output Arguments

**`tiffTagValue` — RPCCoefficientTag property values**
92-element vector

RPCCoefficientTag property values, returned as 92-element vector of class `double`.

## Examples

### Convert RPCCoefficientTag Properties to 92-Element Vector

Create an RPCCoefficientTag object and view the object summary display.

```
rpctag = map.geotiff.RPCCoefficientTag


rpctag =

  RPCCoefficientTag with properties:

                 BiasErrorInMeters: -1
               RandomErrorInMeters: -1
                        LineOffset: 0
                      SampleOffset: 0
             GeodeticLatitudeOffset: 0
            GeodeticLongitudeOffset: 0
               GeodeticHeightOffset: 0
                         LineScale: 1
                       SampleScale: 1
              GeodeticLatitudeScale: 1
             GeodeticLongitudeScale: 1
                GeodeticHeightScale: 1
          LineNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
        LineDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
        SampleNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
      SampleDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Call the RPCCoefficientTag class method `double` to convert the object into a vector of `doubles`. View the result.

```
tifftagvalues = double(rpctag)


tifftagvalues =

  Columns 1 through 13

    -1    -1     0     0     0     0     0     1     1     1     1     1     0

  Columns 14 through 26
```

```
    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 27 through 39

    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 40 through 52

    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 53 through 65

    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 66 through 78

    0     0     0     0     0     0     0     0     0     0     0     0     0

Columns 79 through 91

    0     0     0     0     0     0     0     0     0     0     0     0     0

Column 92

    0
```

**Introduced in R2015b**

# geotiff2mstruct

Convert GeoTIFF information to map projection structure

## Syntax

```
mstruct = geotiff2mstruct(proj)
```

## Description

`mstruct = geotiff2mstruct(proj)` converts the GeoTIFF projection structure, `proj`, to the map projection structure, `mstruct`. The unit of length of the `mstruct` projection is meter.

The GeoTIFF projection structure, `proj`, must reference a projected coordinate system, as indicated by a value of `'ModelTypeProjected'` in the `ModelType` field. If `ModelType` has the value `'ModelTypeGeographic'` then it doesn't make sense to convert to a map projection structure and an error is issued.

## Examples

```
% Compare inverse transform of points using projinv and minvtran.
% Obtain the projection structure of 'boston.tif'.
proj = geotiffinfo('boston.tif');

% Convert the corner map coordinates to latitude and longitude.
x = proj.CornerCoords.X;
y = proj.CornerCoords.Y;
[latProj, lonProj] = projinv(proj, x, y);

% Obtain the mstruct from the GeoTIFF projection.
mstruct = geotiff2mstruct(proj);

% Convert the units of x and y to meter to match projection units.
x = unitsratio('meter','sf') * x;
y = unitsratio('meter','sf') * y;
```

```
% Convert the corner map coordinates to latitude and longitude.
[latMstruct, lonMstruct] = minvtran(mstruct, x, y);

% Verify the values are within a tolerance of each other.
abs(latProj - latMstruct) <= 1e-7
abs(lonProj - lonMstruct) <= 1e-7

ans =
     1     1     1     1

ans =
     1     1     1     1
```

## See Also

axesm | defaultm | geotiffinfo | projfwd | projinv | projlist

# geotiffinfo

Information about GeoTIFF file

## Syntax

```
info = geotiffinfo(filename)
info = geotiffinfo(url)
```

## Description

`info = geotiffinfo(filename)` returns a structure whose fields contain image properties and cartographic information about a GeoTIFF file.

`info = geotiffinfo(url)` reads the GeoTIFF image from a URL.

## Input Arguments

**filename**

String that specifies the name of the GeoTIFF file. Include the folder name in `filename` or place the file in the current folder or in a folder on the MATLAB path. If the named file includes the extension `.TIF` or `.TIFF` (either upper- or lowercase), you can omit the extension from `filename`.

If the named file contains multiple GeoTIFF images, `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file. If multiple images exist in the file, it is assumed that each image has the same cartographic information and image width and height.

**url**

URL that includes the protocol type (e.g., "`http://`").

# Output Arguments

**info**

Structure whose fields contain image properties and cartographic information about a GeoTIFF file. The info structure contains these fields:

| | |
|---|---|
| Filename | String containing the name of the file or URL. |
| FileModDate | String containing the modification date of the file. |
| FileSize | Integer indicating the size of the file in bytes. |
| Format | String containing the file format, which should always be 'tiff'. |
| FormatVersion | String or number specifying the file format version. |
| Height | Integer indicating the height of the image in pixels. |
| Width | Integer indicating the width of the image in pixels. |
| BitDepth | Integer indicating the number of bits per pixel. |
| ColorType | String indicating the type of image: 'truecolor' for a true-color (RGB) image, 'grayscale' for a grayscale image, or 'indexed' for an indexed image. |
| ModelType | String indicating the type of coordinate system used to georeference the image: 'ModelTypeProjected', 'ModelTypeGeographic', 'ModelTypeGeocentric', or ''. |
| PCS | String indicating the projected coordinate system. |
| Projection | String describing the EPSG identifier for the underlying projection method. |
| MapSys | String indicating the map system, if applicable: 'STATE_PLANE_27', 'STATE_PLANE_83', 'UTM_NORTH', 'UTM_SOUTH', or ''. |
| Zone | Double indicating the UTM or State Plane Zone number, empty ([ ]) if not applicable or unknown. |
| CTProjection | String containing the GeoTIFF identifier for the underlying projection method. |
| ProjParm | N-by-1 double containing projection parameter values. The identity of each element is specified by the corresponding |

| | |
|---|---|
| | element of `ProjParmId`. Lengths are in meters, angles in decimal degrees. |
| ProjParmId | N-by-1 cell array listing the projection parameter identifier for each corresponding numerical element of `ProjParm`: |

**Projection Parameter Identifiers**

- `'ProjNatOriginLatGeoKey'`
- `'ProjNatOriginLongGeoKey'`
- `'ProjFalseEastingGeoKey'`
- `'ProjFalseNorthingGeoKey'`
- `'ProjFalseOriginLatGeoKey'`
- `'ProjFalseOriginLongGeoKey'`
- `'ProjCenterLatGeoKey'`
- `'ProjCenterLongGeoKey'`
- `'ProjAzimuthAngleGeoKey'`
- `'ProjRectifiedGridAngleGeoKey'`
- `'ProjScaleAtNatOriginGeoKey'`
- `'ProjStdParallel1GeoKey'`
- `'ProjStdParallel2GeoKey'`

| | |
|---|---|
| GCS | String indicating the geographic coordinate system. |
| Datum | String indicating the projection datum type, such as `'North American Datum 1927'` or `'North American Datum 1983'`. |
| Ellipsoid | String indicating the ellipsoid name as defined by the `ellipsoid.csv` EPSG file. |
| SemiMajor | Double indicating the length of the semimajor axis of the ellipsoid, in meters. |
| SemiMinor | Double indicating the length of the semiminor axis of the ellipsoid, in meters. |
| PM | String indicating the prime meridian location, for example, `'Greenwich'` or `'Paris'`. |

| | |
|---|---|
| PmLongToGreenwich | Double indicating the decimal degrees of longitude between this prime meridian and Greenwich. Prime meridians to the west of Greenwich are negative. |
| UOMLength | String indicating the units of length used in the projected coordinate system. |
| UOMLengthInMeters | Double defining the UOMLength unit in meters. |
| UOMAngle | String indicating the angular units used for geographic coordinates. |
| UOMAngleInDegrees | Double defining the UOMAngle unit in degrees. |
| TiePoints | Structure containing the image tiepoints. The structure contains these fields: |

- ImagePoints — Structure containing row and column coordinates of each image tiepoint. The ImagePoints structure contains these fields:

  **Fields in ImagePoints Structure**

  Row — Double array of size 1-by-N.

  Col — Double array of size 1-by-N.
- WorldPoints — Structure containing the *x* and *y* world coordinates of the tiepoints. The WorldPoints structure contains these fields:

  **Fields in WorldPoints Structure**

  X — Double array of size 1-by-N.

  Y — Double array of size 1-by-N.

| | |
|---|---|
| PixelScale | 3-by-1 double array that specifies the X, Y, Z pixel scale values. |

| | |
|---|---|
| SpatialRef | Value depends on the value of the ModelType field.<br>If ModelType is 'ModelTypeProjected', SpatialRef is a map raster reference object.<br>If ModelType is 'ModelTypeGeographic', SpatialRef is a geographic raster reference object, unless the geometric transformation is affine, in which case SpatialRef is empty.<br>If ModelType is empty, geotiffinfo issues a warning and SpatialRef is a map raster reference object.<br>If ModelType is 'ModelTypeGeocentric', SpatialRef is empty.<br>If the spatial referencing is ambiguously defined by the GeoTIFF file, SpatialRef is empty. |
| RefMatrix | 3-by-2 double referencing matrix that must be unambiguously defined by the GeoTIFF file. Otherwise it is empty ([ ]). |
| BoundingBox | 2-by-2 double array that specifies the minimum (row 1) and maximum (row 2) values for each dimension of the image data in the GeoTIFF file. |
| CornerCoords | Structure with six fields that contains coordinates of the outer corners of the GeoTIFF image. Each field is a 1-by-4 double array, or empty ([ ]) if unknown. The arrays contain the coordinates of the outer corners of the corner pixels, starting from the (1,1) corner and proceeding clockwise: |

**Coordinates of the Outer Corners**

- X — Easting coordinates in the projected coordinate system. X equals Lon (below) if *ModelType* is 'ModelTypeGeographic'
- Y — Northing coordinates in the projected coordinate system. Y equals Lat (below) if *ModelType* is 'ModelTypeGeographic'
- Row — Row coordinates of the corner.
- Col — Column coordinates of the corner.
- Lat — Latitudes of the corner.
- Lon — Longitudes of the corner.

| | |
|---|---|
| GeoTIFFCodes | Structure containing raw numeric values for those GeoTIFF fields that are encoded numerically in the file. These raw values, converted to a string elsewhere in the `info` structure, are provided here for reference. |

**GeoTIFFCodes Fields**

- `Model`
- `PCS`
- `GCS`
- `UOMLength`
- `UOMAngle`
- `Datum`
- `PM`
- `Ellipsoid`
- `ProjCode`
- `Projection`
- `CTProjection`
- `ProjParmId`
- `MapSys`

Each is scalar, except for `ProjParmId`, which is a column vector.

| GeoTIFFTags | Structure containing field names that match the GeoTIFF tags in the file. At least one GeoTIFF tag must be present in the file or an error is issued. The following fields may be included: |
|---|---|

**GeoTIFF Tag Fields**

- `ModelPixelScaleTag`: 1-by-3 `double`
- `ModelTiepointTag`: 1-by-6 `double`
- `ModelTransformationTag`: 1-by-16 `double`
- `GeoKeyDirectoryTag`: scalar structure
- `GeoAsciiParamsTag`: string
- `GeoDoubleParamsTag`: 1-by-N `double`
- `RPCCoefficientTag`: scalar `map.geotiff.RPCCoefficientTag`

The `GeoKeyDirectoryTag` contains field names that match the names of the "GeoKeys". For more information about the "GeoKeys" refer to the GeoTIFF specification.

The `RPCCoefficientTag` contains properties with names corresponding to the tag elements listed in the *RPCs in GeoTIFF* technical note at: http://geotiff.maptools.org/rpc_prop.html

| ImageDescription | String describing the image. If no description is included in the file, the field is omitted. |
|---|---|

## Examples

### Return Information about Geotiff File

```
info = geotiffinfo('boston.tif')

info =

          Filename: 'B:\matlab\toolbox\map\mapdata\boston.tif'
       FileModDate: '13-May-2011 22:28:44'
```

```
            FileSize: 38729900
              Format: 'tif'
       FormatVersion: []
              Height: 2881
               Width: 4481
            BitDepth: 8
           ColorType: 'truecolor'
           ModelType: 'ModelTypeProjected'
                 PCS: 'NAD83 / Massachusetts Mainland'
          Projection: 'SPCS83 Massachusetts Mainland zone (meters)'
              MapSys: 'STATE_PLANE_83'
                Zone: 2001
        CTProjection: 'CT_LambertConfConic_2SP'
            ProjParm: [7x1 double]
          ProjParmId: {7x1 cell}
                 GCS: 'NAD83'
               Datum: 'North American Datum 1983'
           Ellipsoid: 'GRS 1980'
           SemiMajor: 6378137
           SemiMinor: 6.3568e+06
                  PM: 'Greenwich'
   PMLongToGreenwich: 0
           UOMLength: 'US survey foot'
    UOMLengthInMeters: 0.3048
            UOMAngle: 'degree'
    UOMAngleInDegrees: 1
           TiePoints: [1x1 struct]
          PixelScale: [3x1 double]
          SpatialRef: [1x1 map.rasterref.MapCellsReference]
           RefMatrix: [3x2 double]
         BoundingBox: [2x2 double]
        CornerCoords: [1x1 struct]
        GeoTIFFCodes: [1x1 struct]
         GeoTIFFTags: [1x1 struct]
    ImageDescription: '"GeoEye"'
```

## See Also

map.geotiff.RPCCoefficientTag | geotiffread | geotiffwrite | projfwd | projinv

# geotiffread

Read GeoTIFF file

## Syntax

```
[A, R] = geotiffread(filename)
[X, cmap, R] = geotiffread(filename)
[A, refmat, bbox] = geotiffread(filename)
[X, cmap, refmat, bbox] = geotiffread(filename)
[...] = geotiffread(filename, idx)
[...] = geotiffread(url, ...)
```

## Description

[A, R] = geotiffread(filename) reads a georeferenced grayscale, RGB, or multispectral image or data grid from the GeoTIFF file specified by filename into A and constructs a spatial referencing object, R.

[X, cmap, R] = geotiffread(filename) reads an indexed image into X and the associated colormap into cmap, and constructs a spatial referencing object, R. Colormap values in the image file are rescaled into the range [0,1].

[A, refmat, bbox] = geotiffread(filename) reads a georeferenced grayscale, RGB, or multispectral image or data grid into A; the corresponding referencing matrix into refmat; and the bounding box into bbox.

[X, cmap, refmat, bbox] = geotiffread(filename) reads an indexed image into X, the associated colormap into cmap, the referencing matrix into refmat, and the bounding box into bbox. The referencing matrix must be unambiguously defined by the GeoTIFF file, otherwise it and the bounding box are returned empty.

[...] = geotiffread(filename, idx) reads one image from a multi-image GeoTIFF file.

[...] = geotiffread(url, ...) reads the GeoTIFF image from a URL.

## Input Arguments

**filename**

String that specifies the name of the GeoTIFF file. `filename` can include the folder name. Otherwise, the file must be in the current folder or in a folder on the MATLAB path. If the named file includes the extension `'.TIF'` or `'.TIFF'` (either upper or lowercase), you can omit the extension from `filename`.

**idx**

Integer value that specifies the order that the image appears in the file. For example, if `idx` is 3, `geotiffread` reads the third image in the file.

**Default:** First image in the file

**url**

Internet URL. The URL must include the protocol type (e.g., "http://").

## Output Arguments

**A**

Two-dimensional array, if the file contains a grayscale image or data grid. An M-by-N-by-P array, if the file contains a color image, multispectral image, hyperspectral image, or data grid. The class of `A` depends on the storage class of the pixel data in the file, which is related to the `BitsPerSample` property as returned by the `imfinfo` function.

**R**

Geographic raster reference object if the image or data grid is referenced to a geographic coordinate system, or a map raster reference object if it is referenced to a projected coordinate system.

**X**

Indexed image

**cmap**

Colormap

**refmat**

Referencing matrix

**bbox**

Bounding box

## Examples

Read and display the Boston GeoTIFF image:

```
[boston, R] = geotiffread('boston.tif');
figure
mapshow(boston, R);
axis image off
```

## More About

**Tips**

- `geotiffread` imports pixel data using the TIFF-reading capabilities of the MATLAB function `imread` and likewise shares any limitations of `imread`. Consult the `imread` documentation for information on TIFF image support.

## See Also
`geoshow` | `geotiffinfo` | `imread` | `mapshow` | `geotiffwrite`

# geotiffwrite

Write GeoTIFF file

## Syntax

```
geotiffwrite(filename,A,R)
geotiffwrite(filename,X,cmap,R)
geotiffwrite(...,Name,Value)
```

## Description

geotiffwrite(filename,A,R) writes a georeferenced image or data grid, A, spatially referenced by R, into an output file, filename.

geotiffwrite(filename,X,cmap,R) writes the indexed image in X and its associated colormap, cmap, to filename. X is spatially referenced by R.

geotiffwrite(...,Name,Value) writes a georeferenced image or data grid with additional options that control various characteristics of the output file specified by one or more Name,Value pair arguments.

## Input Arguments

### filename — Name and location of output file
character string

Name and location of output file, specified as a character string. If your filename includes an extension, it must be '.tif' or '.TIF'. The output file is a tiled GeoTIFF file if the input, A, is at least 160-by-160 in size. Otherwise, the output file is organized as rows-per-strip.

Data Types: char

### A — Image data
M-by-N numeric array | M-by-N-by-P numeric array

Image data, specified as an M-by-N array (grayscale image or data grid) or M-by-N-by-P array (color or hyperspectral image, or data grid). The coordinates of A are geographic

and in the `'WGS 84'` coordinate system, unless you specify `'GeoKeyDirectoryTag'` or `'CoordRefSysCode'` and indicate a different coordinate system.

### R — Spatial referencing information
raster reference object | referencing matrix | referencing vector

Spatial referencing information, specified as a raster reference object, referencing matrix, or referencing vector. If you are working with image coordinates in a projected coordinate system and R is a map raster reference object or a referencing matrix, specify `'GeoKeyDirectoryTag'` or `'CoordRefSysCode'` accordingly.

### X — Indexed image
numeric array

Indexed image data, specified as a numeric array.

### cmap — Color map
M-by-3 array

Color map, specified as an M-by-3 array.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

The argument names are case insensitive.

### `'CoordRefSysCode'`

Scalar, positive, integer-valued number that specifies the coordinate reference system code for the coordinates of the data. You can specify coordinates in either a geographic or a projected coordinate system, and you can use a string, such as `'EPSG:4326'`. If you specify the coordinate system with a string, include the `'EPSG:'` prefix. See "References" on page 1-594 for a link to the GeoTiff Specification or the EPSG data files (`pcs.csv` and `gcs.csv`) for the code numbers.

If you specify both the `'GeoKeyDirectoryTag'` and the `'CoordRefSysCode'`, the coordinate system code in the `'CoordRefSysCode'` takes precedence over the coordinate system key found in the `'GeoKeyDirectoryTag'`. If one value specifies a

geographic coordinate system and the other value specifies a projected coordinate system, you receive an error.

If you do not specify a value for this argument, the default value is 4326, indicating that the coordinates are geographic and in the `'WGS 84'` geographic coordinate system.

**Default:** 4326

**`'GeoKeyDirectoryTag'`**

Structure that specifies the GeoTIFF coordinate reference system and meta-information. The structure contains field names that match the GeoKey names in the GeoTIFF specification. The field names are case insensitive. The structure can be obtained from the GeoTIFF information structure, returned by `geotiffinfo`, in the field, `GeoTIFFTags.GeoKeyDirectoryTag`. If you set certain fields of the `'GeoKeyDirectoryTag'` to inconsistent settings, you receive an error message.

The `GTRasterTypeGeoKey` field is ignored, if specified. The value for this GeoKey is derived from R. If you set certain fields of the `'GeoKeyDirectoryTag'` to inconsistent settings, you receive an error message. For instance, if R is either a `spatialref.GeoRasterReference` object or a refvec and you specify a `ProjectedCSTypeGeoKey` field or you set the `GTModelTypeGeoKey` field to 1 (projected coordinate system), you receive an error. Likewise, if R is a `spatialref.MapRasterReference` object and you do not specify a `ProjectedCSTypeGeoKey` field or a `'CoordRefSysCode'`, or the `GTModelTypeGeoKey` field is set to 2 (geographic coordinate system), you receive an error message.

**`'RPCCoefficientTag'`**

`map.geotiff.RPCCoefficientTag` object that specifies values for the optional Rational Polynomial Coefficients (RPC) TIFF tag. For more information about these values, see map.geotiff.RPCCoefficientTag.

**`'TiffTags'`**

Structure that specifies values for the TIFF tags in the output file. The field names of the structure match the TIFF tag names supported by the Tiff class. The field names are case insensitive.

You cannot set most TIFF tags using the structure input.

**TiffTags Exceptions**

- `BitsPerSample`

- SampleFormat
- SamplesPerPixel
- StripByteCounts
- StripOffsets
- SubFileType
- SubIFD
- TileByteCounts
- TileOffsets
- ColorMap
- ImageLength
- ImageWidth
- GeoAsciiParamsTag
- GeoDoubleParamsTag
- GeoKeyDirectoryTag
- ModelPixelScaleTag
- ModelTiepointTag
- ModelTransformationTag

The function sets several TIFF tags. The field names corresponding to the TIFF tag, their corresponding field values set by the function, their permissible values (if different from the Tiff class), and their data type are noted in the following table.

**Automatic TIFF Tags**

| Field Name | Description |
|---|---|
| Compression | String indicating the type of image compression. The default is `'PackBits'`. Other permissible values are `'LZW'`, `'Deflate'`, and `'none'`. <br><br> Numeric values, `Tiff.Compression.LZW`, `Tiff.Compression.PackBits`, `Tiff.Compression.Deflate`, or `Tiff.Compression.None` can also be used. |
| PhotometricInterpretation | String indicating the type of photometric interpretation. The field name can be shortened to `Photometric`. The |

| Field Name | Description |
|---|---|
| | value is set based on the input image characteristic, using the following algorithm: If A is [M-by-N-by-3] and is class type `uint8` or `uint16`, then the value is `'RGB'`. For all other sizes and data types, the value is `'MinIsBlack'`. If the X, CMAP syntax is supplied, the value is `'Palette'`. If the value is set to `'RGB'` and A is not [M-by-N-by-3], an error is issued. Permissible values are `'MinIsBlack'`, `'RGB'`, `'Palette'`, `'Separated'`. The numeric values, `Tiff.Photometric.MinIsBlack`, `Tiff.Photometric.RGB`, `Tiff.Photometric.Palette`, `Tiff.Photometric.Separated` can also be used. |
| `Software` | A string indicating the software maker of the file. The value is set to the string value `'MATLAB, Mapping Toolbox, The MathWorks, Inc.'`. To remove the value, set the tag to the empty string (`''`). |
| `RowsPerStrip` | A scalar positive integer-valued number specifying the desired rows per strip in the output file. `RowsPerStrip` is set to the value of 1 if the size of A is less than [160-by-160]. An error is issued if both `RowsPerStrip` and `TileWidth` and/or `TileLength` are specified. |
| `TileWidth` | A scalar positive integer-valued number and a multiple of 16 specifying the width of the tiles. `TileWidth` is set if the size of A is greater than [160-by-160]. If so, the value is such that a maximum of [10-by-10] tiles are created. An error is issued if both `RowsPerStrip` and `TileWidth` are specified. |
| `TileLength` | A scalar positive integer-valued number and a multiple of 16 specifying the length of the tiles. `TileLength` is set if the size of A is greater than [160-by-160]. If so, the value is such that a maximum of [10-by-10] tiles are created. An error is issued if both `RowsPerStrip` and `TileLength` are specified. |

## Class Support

The input array, A, can be any numeric class or logical. The indexed image, X, must be class uint8 or uint16. The colormap array, cmap, must be class double.

## Examples

### Write Image from JPEG File to GeoTIFF File

Read JPEG image from file.

```
basename = 'boston_ovr';
imagefile = [basename '.jpg'];
RGB = imread(imagefile);
```

Derive world file name from image file name, read the world file, and construct a spatial referencing object.

```
worldfile = getworldfilename(imagefile);
R = worldfileread(worldfile, 'geographic', size(RGB));
```

Write image data and referencing data to GeoTIFF file.

```
filename = [basename '.tif'];
geotiffwrite(filename, RGB, R)
```

Construct an empty map axes and display the map.

```
figure
usamap(RGB, R)
geoshow(filename)
```
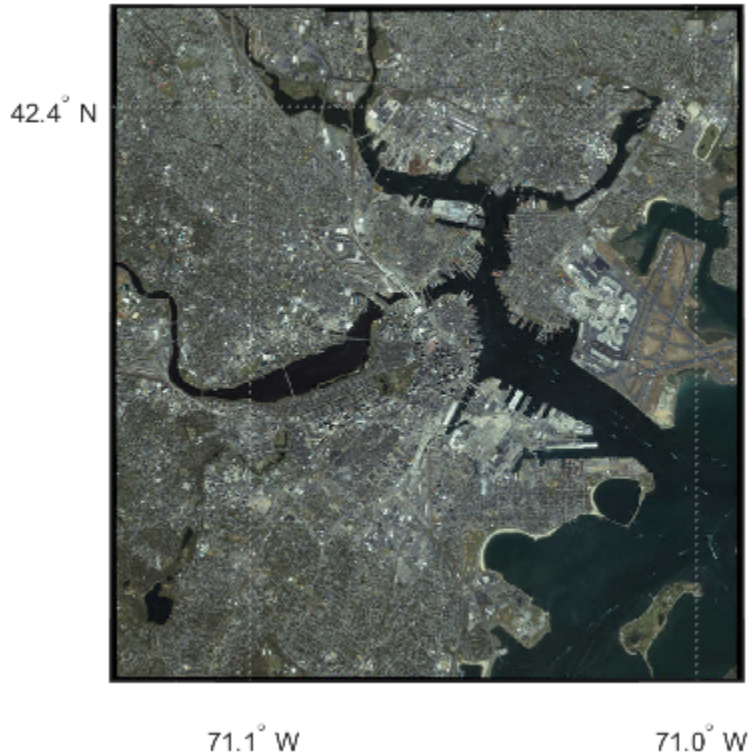
### Write WMS Image to GeoTIFF File
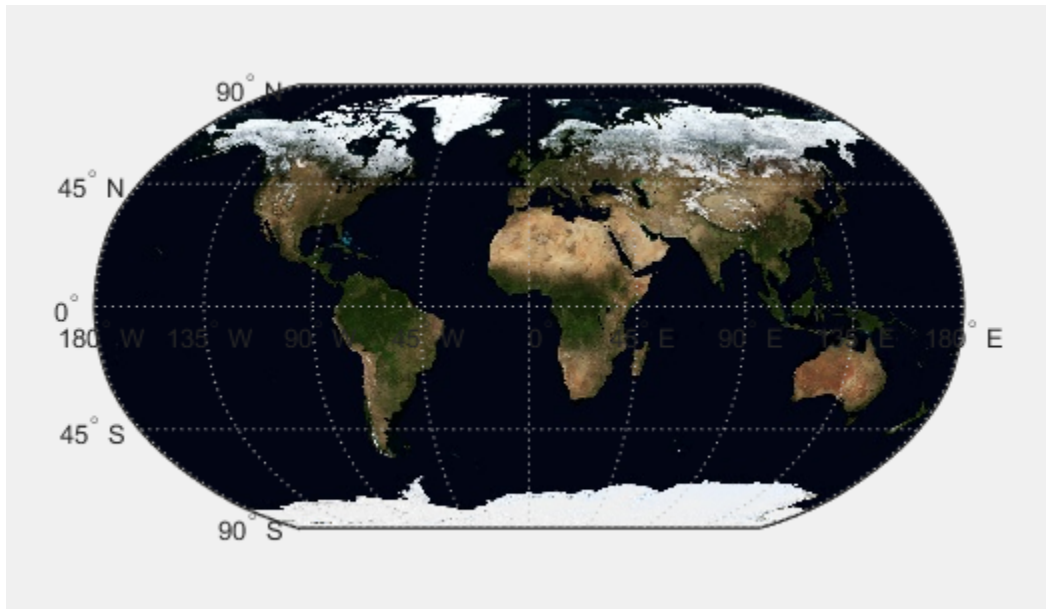
Read data from WMS server.

```
nasaLayers = wmsfind('nasa', 'SearchField', 'serverurl');
layerName = 'bluemarbleng';
layer = refine(nasaLayers, layerName,  'SearchField', 'layername', ...
  'MatchType', 'exact');
[A, R] = wmsread(layer(1));
```

Write data to GeoTIFF file.

```
filename = [layerName '.tif'];
geotiffwrite(filename, A, R)
```

View data in file.

```
figure
worldmap world
geoshow(filename)
```



### Write Concord Orthophotos to Single GeoTIFF File

Read the two adjacent orthophotos and combine them.

```
X_west = imread('concord_ortho_w.tif');
X_east = imread('concord_ortho_e.tif');
X = [X_west X_east];
```

Construct referencing objects for the orthophotos and for their combination.

```
R_west = worldfileread('concord_ortho_w.tfw', 'planar', size(X_west));
R_east = worldfileread('concord_ortho_e.tfw', 'planar', size(X_east));
R = R_west;
R.XLimWorld = [R_west.XLimWorld(1) R_east.XLimWorld(2)];
R.RasterSize = size(X);
```

Write the combined image to a GeoTIFF file. Use the code number, 26986, indicating the PCS_NAD83_Massachusetts Projected Coordinate System.

```
coordRefSysCode = 26986;
filename = 'concord_ortho.tif';
geotiffwrite(filename, X, R, 'CoordRefSysCode', coordRefSysCode);
```

Display the map.

```
figure
mapshow(filename)
```
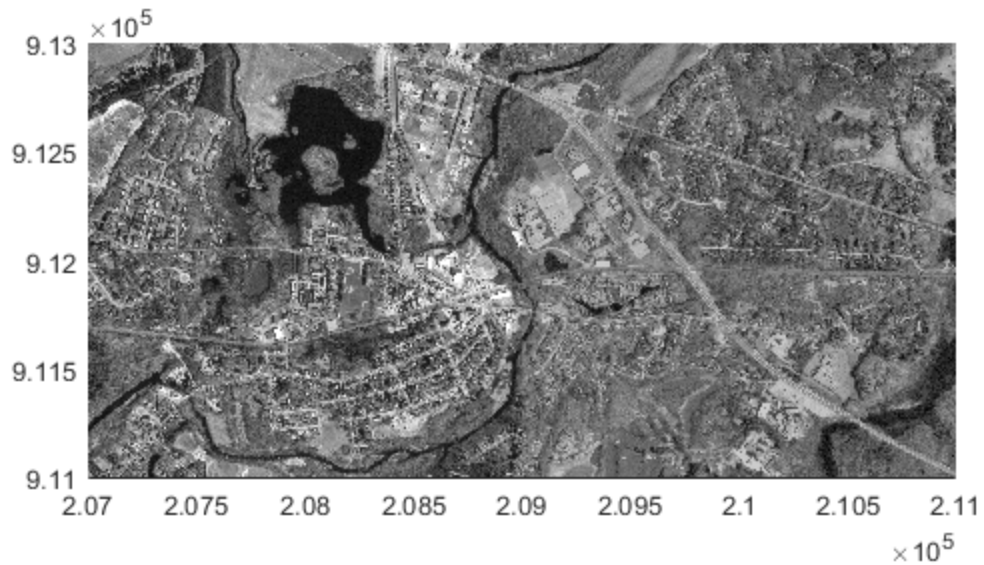


### Write Subset of GeoTIFF File to New GeoTIFF File
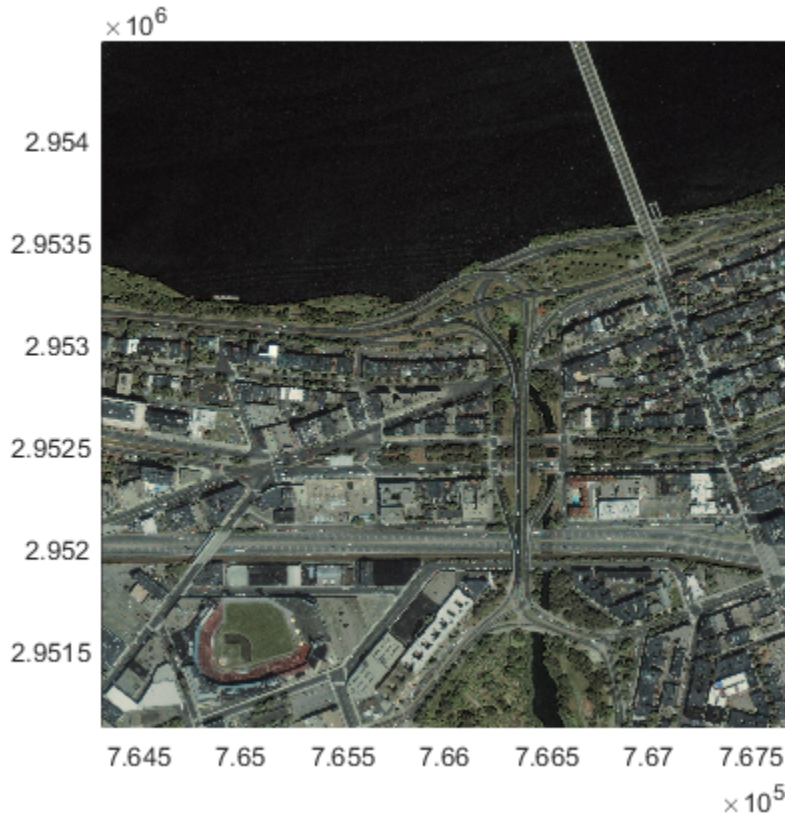
Read GeoTIFF file.

```
[A, R] = geotiffread('boston.tif');
```

Specify subset of data that you want to write.

```
m = size(A,1);
n = 1024;
firstrow = m-n+1;
lastrow = m;
firstcol = 1;
lastcol = n;
subImage = A(firstrow:lastrow, firstcol:lastcol, :);
xi = [firstcol - .5, lastcol + .5];
yi = [firstrow - .5, lastrow + .5];
[xlimits, ylimits] = intrinsicToWorld(R, xi, yi);
subR = R;
subR.RasterSize = size(subImage);
subR.XLimWorld = sort(xlimits);
subR.YLimWorld = sort(ylimits);
info = geotiffinfo('boston.tif');
```

Write subset to file.

```
filename = 'boston_subimage.tif';
geotiffwrite(filename, subImage, subR,  ...
       'GeoKeyDirectoryTag', info.GeoTIFFTags.GeoKeyDirectoryTag);
figure
mapshow(filename);
```

## Write SDTS DEM terrain data to GeoTIFF File

This example shows how to write the Mount Washington SDTS DEM terrain data to GeoTIFF. The data are referenced to Universal Transverse Mercator (UTM), Zone 19, in the North American Datum of 1927. This corresponds to the GeoTIFF PCS_NAD27_UTM_zone_19N code number 26719. Set the raster interpretation to 'postings' because the data is USGS DEM. This corresponds to the GeoTIFF raster type PixelIsPoint.

Read SDTS DEM data.

```
[Z, refmat] = sdtsdemread('9129CATD.ddf');
R = refmatToMapRasterReference(refmat, size(Z), 'postings');
key.GTModelTypeGeoKey  = 1;  % Projected Coordinate System (PCS)
```

```
key.GTRasterTypeGeoKey = 2;  % PixelIsPoint
key.ProjectedCSTypeGeoKey = 26719;
filename = '9129.tif';
geotiffwrite(filename, Z, R, 'GeoKeyDirectoryTag', key);
```
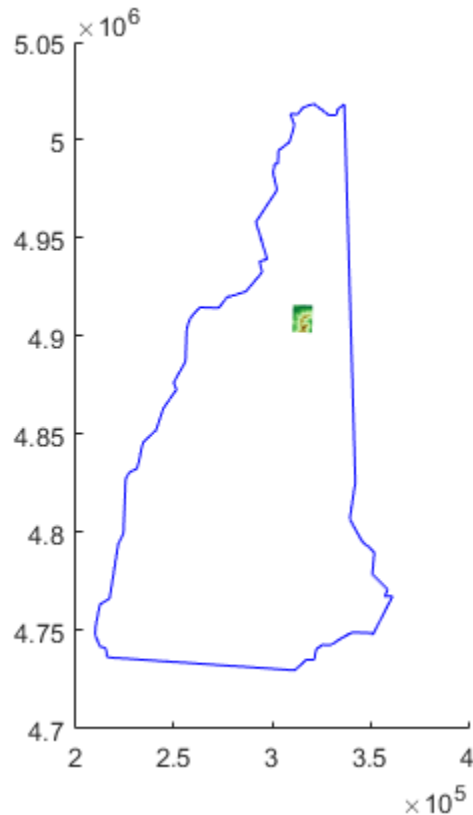
Plot the outline of the state of New Hampshire in UTM.

```
S = shaperead('usastatelo', 'UseGeoCoords', true, 'Selector',...
    {@(name) any(strcmp(name,{'New Hampshire'})), 'Name'});
proj = geotiffinfo(filename);
[x, y] = projfwd(proj, [S.Lat], [S.Lon]);
figure
mapshow(x,y)
```
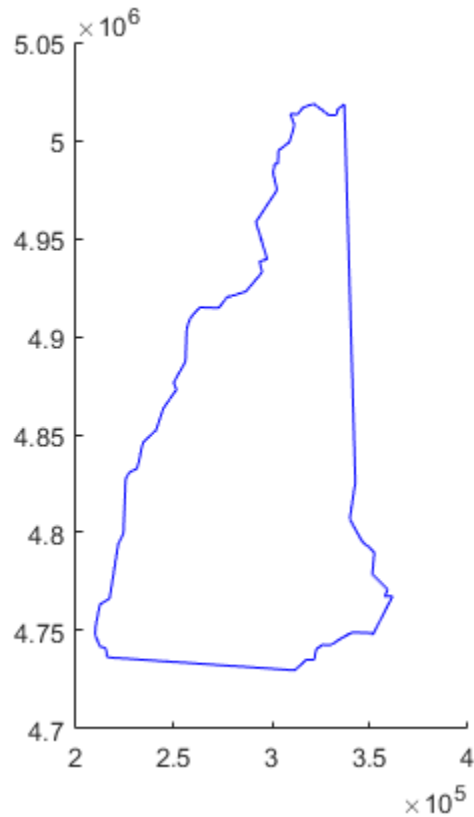


Display the GeoTIFF DEM file.

```
hold on
h = mapshow(filename, 'DisplayType', 'surface');
demcmap(get(h,'ZData'))
```



### Read and Write TIFF File Containing RPC Metadata

This example shows how to read a file containing RPC metadata, modify the file, and then write the file, including the RPC metadata in the new file. To illustrate this process, the example first creates a sample TIFF file containing RPC Metadata.

### Create a Sample TIFF File with RPC Metadata

Create a sample test TIFF file containing RPC metadata. For this test file, create a toy image and a referencing object associated with the image.

```
myimage = zeros(180,360);
latlim = [-90 90];
lonlim = [-180 180];
R = georefcells(latlim,lonlim,size(myimage));
```

Create an `RPCCoefficientTag` metadata object and set some fields with typical values. The toolbox uses the `RPCCoefficientTag` object to represent RPC metadata in human readable form.

```
rpctag = map.geotiff.RPCCoefficientTag;
rpctag.LineOffset = 1;
rpctag.SampleOffset = 1;
rpctag.LineScale = 2;
rpctag.SampleScale = 2;
rpctag.GeodeticHeightScale = 500;
```

Write the image, the associated referencing object, and the `RPCCoefficentTag` object to a file.

```
geotiffwrite('myfile',myimage,R,'RPCCoefficientTag',rpctag)
```

### Read the Metadata and Image Data from the Sample TIFF File

Read the metadata and the image data from the test file. `geotiffinfo` returns the RPC metadata in the GeoTIFFTags field of the structure returned.

```
info = geotiffinfo('myfile');
[imagedata, R] = geotiffread('myfile');
```

### Write Modified Image Data and RPC Metadata to File

First modify the image data read from the file. For example, set all the zero-valued pixels to 1 in the image.

```
imagedata(imagedata==0)= 1;
```

Write the modified image to a file, along with the RPC metadata from the original file.

```
geotiffwrite('myfile2',imagedata,R,...
    'RPCCoefficientTag',info.GeoTIFFTags.RPCCoefficientTag)
```

### Write Raw RPC Coefficient Metadata to GeoTIFF File

This example shows how to write RPC coefficient metadata to a TIFF file. In a real workflow, you would create the RPC coefficient metadata according to the TIFF extension specification. This example does not show the specifics of how to create valid

RPC metadata. To simulate raw RPC metadata, the example creates a sample TIFF file with RPC metadata and then uses `imfinfo` to read this RPC metadata in raw, unprocessed form from the file. The example then writes this raw RPC metadata to a file using the `geotiffwrite` function.

### Create Raw RPC Coeffient Metadata

To simulate raw RPC metadata, create a simple test file and write some RPC metadata to the file. For this test file, create a toy image and a referencing object associated with the image.

```
myimage = zeros(180,360);
latlim = [-90 90];
lonlim = [-180 180];
R = georefcells(latlim,lonlim,size(myimage));
```

Create an RPCCoefficientTag metadata object and set some of the fields. The toolbox uses the RPCCoefficientTag object to represent RPC metadata in human readable form.

```
rpctag = map.geotiff.RPCCoefficientTag;
rpctag.LineOffset = 1;
rpctag.SampleOffset = 1;
rpctag.LineScale = 2;
rpctag.SampleScale = 2;
rpctag.GeodeticHeightScale = 500;
```

Write the image, the associated referencing object, and the RPCCoeffcentTag object to a file.

```
geotiffwrite('myfile',myimage,R,'RPCCoefficientTag',rpctag)
```

### Read Raw RPC Coefficient Metadata

Read the RPC coefficient metadata from the test file using the `imfinfo` function. When it encounters unfamiliar metadata, `imfinfo` returns the data, unprocessed, in the `UnknownTags` field. Note how the UnknownTags field contains an array of 92 doubles. This is the raw RPC coefficient metadata, read from the file in unprocessed form.

```
info = imfinfo('myfile.tif');
info.UnknownTags


ans =

      ID: 50844
```

```
      Offset: 10672
       Value: [1x92 double]
```

### Write Raw RPC Metadata to a File

Write the raw RPC metadata to a file. First, extract the RPC coefficient metadata from the info structure.

```
value = info.UnknownTags.Value;
```

Then, construct an RPCCoefficientTag object, passing the raw RPC metadata (array of 92 doubles) as an argument.

```
rpcdata = map.geotiff.RPCCoefficientTag(value)
```

```
rpcdata =

  RPCCoefficientTag with properties:

               BiasErrorInMeters: -1
             RandomErrorInMeters: -1
                      LineOffset: 1
                    SampleOffset: 1
          GeodeticLatitudeOffset: 0
         GeodeticLongitudeOffset: 0
            GeodeticHeightOffset: 0
                       LineScale: 2
                     SampleScale: 2
           GeodeticLatitudeScale: 1
          GeodeticLongitudeScale: 1
             GeodeticHeightScale: 500
        LineNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
      LineDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
      SampleNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
    SampleDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Pass the RPCCoefficientTag object to the `geotiffwrite` function and write the RPC metadata to a file.

```
geotiffwrite('myfile2',myimage,R,'RPCCoefficientTag',rpcdata)
```

To verify that the data was written to the file, read the RPC metadata from the TIFF file using `geotiffinfo`. Compare the returned RPC metadata with the metadata written to the test file.

```
ginfo = geotiffinfo('myfile2');
ginfo.GeoTIFFTags.RPCCoefficientTag
```

```
ans =

  RPCCoefficientTag with properties:

                BiasErrorInMeters: -1
              RandomErrorInMeters: -1
                       LineOffset: 1
                     SampleOffset: 1
           GeodeticLatitudeOffset: 0
          GeodeticLongitudeOffset: 0
            GeodeticHeightOffset: 0
                        LineScale: 2
                      SampleScale: 2
            GeodeticLatitudeScale: 1
           GeodeticLongitudeScale: 1
             GeodeticHeightScale: 500
         LineNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       LineDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
       SampleNumeratorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     SampleDenominatorCoefficients: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

## More About

### Tips

- If you are working with image coordinates in a projected coordinate system and `R` is a map raster reference object or a referencing matrix, set the `'GeoKeyDirectoryTag'` or `'CoordRefSysCode'` argument, accordingly. See "Name-Value Pair Arguments" on page 1-578 for more information.

# References

Check the GeoTIFF specification for values of the following parameters:

- 'CoordRefSysCode' value for geographic coordinate systems
- 'CoordRefSysCode' value for projected coordinate systems
- GeoKey field names for the 'GeoKeyDirectoryTag'

The 'CoordRefSysCode' values may also be obtained from the EPSG data files (pcs.csv and gcs.csv) in the folder:

*matlabroot*/toolbox/map/mapproj/projdata/epsg_csv

## See Also

map.geotiff.RPCCoefficientTag | Tiff | geotiffinfo | geotiffread | imread | imwrite

**Introduced before R2006a**

# getm

Map object properties

## Syntax

```
mat = getm(h)
mat = getm(h,MapPropertyName)
getm('MapProjection')
getm('axes')
getm('units')
```

## Description

`mat = getm(h)` returns the map structure of the map axes specified by its handle. If the handle of a child of the map axes is specified, only its properties are returned.

`mat = getm(h,MapPropertyName)` returns the specified property value.

`getm('MapProjection')` lists all available projections.

`getm('axes')` lists the map axes properties by property name.

`getm('units')` lists the available units.

## Examples

Create a default map axes and query a property value:

```
axesm('mercator','AngleUnits','degrees')
getm(gca,'MapParallels')

ans =
     0
```

## See Also

axesm | setm

# getseeds

Interactively assign seeds for data grid encoding

## Compatibility

`getseeds` will be removed in a future release.

## Syntax

```
[row,col,val] = getseeds(map,R,nseeds)
[row,col,val] = getseeds(map,R,nseeds,seedval)
mat = getseeds(...)
```

## Description

`[row,col,val] = getseeds(map,R,nseeds)` allows user to identify geographical objects while customizing a raster map. It prompts the user for mouse click positions of objects and assigns them a code value. The user is prompted for the value to seed at each location. The outputs are the row and column of the seed location and the value assigned at that location. `R` is either a 1-by-3 vector containing elements:

`[cells/degree northern_latitude_limit western_longitude_limit]`
or a 3-by-2 referencing matrix that transforms raster row and column indices to/from geographic coordinates according to:

`[lon lat] = [row col 1] * R`
If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel.

`[row,col,val] = getseeds(map,R,nseeds,seedval)` assigns the value `seedval` to each location supplied. If `seedval` is a scalar then the same value is assigned at each location. Otherwise, if `seedval` is a vector it must be `length(nseeds)` and each entry is assigned to the corresponding location. `getseeds` operates on the current axes (`gca`).

`mat = getseeds(...)` returns a single output matrix where `mat = [row col val]`.

## Examples

Demonstrate this for yourself by typing the following and interactively selecting points:

```
load topo
axesm('gortho','grid','on')
seedmat = getseeds(topo,topolegend,3)
```

When you have selected three points, you are prompted for their values. The regular data grid need not be displayed to execute `getseeds` on it.

## See Also

`encodem`

# getworldfilename

Derive worldfile name from image filename

## Syntax

```
worldfilename = getworldfilename(imagefilename)
```

## Description

`worldfilename = getworldfilename(imagefilename)` returns the name of the corresponding worldfile derived from the name of an image file.

The worldfile and the image file have the same base name. If `imagefilename` follows the ".3" convention, then you create the worldfile extension by removing the middle letter and appending the letter `'w'`.

If `imagefilename` has an extension that does not follow the ".3" convention, then a `'w'` is appended to the full image name to construct the worldfile name.

If `imagefilename` has no extension, then `'.wld'` is appended to construct a worldfile name.

## Examples

Given the following image filenames, `worldfilename` would return these worldfile names:

| Image File Name | Worldfile Name |
|---|---|
| myimage.tif | myimage.tfw |
| myimage.jpeg | myimage.jpegw |
| myimage | myimage.wld |

## See Also
mapshow | mapview | worldfileread | worldfilewrite

# globedem

Read Global Land One-km Base Elevation (GLOBE) data

## Syntax

```
[Z,refvec] = globedem(filename,scalefactor)
[Z,refvec] = globedem(filename,scalefactor,latlim,lonlim)
[Z,refvec] = globedem(foldername,scalefactor,latlim,lonlim)
```

## Description

`[Z,refvec] = globedem(filename,scalefactor)` reads the GLOBE DEM files and returns the result as a regular data grid. The filename is given as a string that does not include an extension. GLOBEDEM first reads the ESRI header file found in the subfolder `'/esri/hdr/'` and then the binary data file filename. If the files are not found on the MATLAB path, they can be selected interactively. `scalefactor` is an integer that when equal to 1 gives the data at its full resolution. When `scalefactor` is an integer `n` larger than 1, every `n`th point is returned. The map data is returned as an array of elevations and associated three-element referencing vector. Elevations are given in meters above mean sea level, using WGS 84 as a horizontal datum.

`[Z,refvec] = globedem(filename,scalefactor,latlim,lonlim)` allows a subset of the map data to be read. The limits of the desired data are specified as vectors of latitude and longitude in degrees. The elements of `latlim` and `lonlim` must be in ascending order.

`[Z,refvec] = globedem(foldername,scalefactor,latlim,lonlim)` reads and concatenates data from multiple files within a GLOBE folder tree. The `foldername` input is a string with the name of the folder that contains both the uncompressed data files and the ESRI header files.

## Background

GLOBE, the Global Land One-km Base Elevation data, was compiled by the National Geophysical Data Center from more than 10 different sources of gridded elevation data.

GLOBE can be considered a higher resolution successor to TerrainBase. The data set consists of 16 tiles, each covering 50 by 90 degrees. Tiles require as much as 60 MB of storage. Uncompressed tiles take between 100 and 130 MB.

## Examples

Determine the file that contains the area around Cape Cod. (This example assumes you have already downloaded some GLOBE data tiles.)

```
latlim = [41 42.5]; lonlim = [-73 -69.9];
globedems(latlim,lonlim)

ans =
    'f10g'
```

Extract every 20th point from the tile covering the northeastern United States and eastern Canada. If you specify an empty file name (' '), globedem presents a file browser that you use to first select the header file and then select the data file interactively.

```
[Z,refvec] = globedem('',20);
size(Z)

ans =
   300    540
```

Extract a subset of the data for Massachusetts at the full resolution.

```
latlim = [41 42.5]; lonlim = [-73 -69.9];
[Z,refvec] = globedem('f10g',1,latlim,lonlim);
size(Z)

ans =
   181 373
```

Replace the NaNs in the ocean with -1 to color them blue.

```
Z(isnan(Z)) = -1;
```

Extract some data for southern Louisiana in an area that straddles two tiles. Provide the name of the folder containing the data files, and let globedem determine which files are required, read from the files, and concatenate the data into a single regular data grid.

```
latlim =[28.61 31.31]; lonlim = [-91.24 -88.62];
globedems(latlim,lonlim)

ans =
    'e10g'
    'f10g'

[Z,refvec] =
globedem('d:\externalData\globe\elev',1,latlim,lonlim);
size(Z)

ans =
      325.00        315.00
```

## More About

**Tips**

The `globedem` function reads data from GLOBE Version 1.0. The data is for elevations only. Elevations are given in meters above mean sea level using WGS 84 as a horizontal datum. Areas with no data, such as the oceans, are coded with `NaN`s.

The data set and documentation are available over the Internet.

---

**Note**  For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

---

## References

See Web site for the National Oceanic and Atmospheric Administration, National Geophysical Data Center

## See Also
demdataui | dted | gtopo30 | satbath | tbase | usgsdem

# globedems

GLOBE data filenames for latitude-longitude quadrangle

## Syntax

```
tileNames = globedems(latlim,lonlim)
tileNames = globedems(lat,lon)
```

## Description

`tileNames = globedems(latlim,lonlim)` returns a cell array of the tile names covering the geographic region for GLOBEDEM digital elevation maps. The region is specified by two-element vectors of latitude and longitude limits in units of degrees.

`tileNames = globedems(lat,lon)` returns a cell array of the tile names covering the geographic region for GLOBEDEM digital elevation maps. The region is specified by scalar latitude and longitude points, in units of degrees.

## Background

GLOBE, the Global Land One-km Base Elevation data, was compiled by the National Geophysical Data Center from more than 10 different sources of gridded elevation data. The data set consists of 16 tiles, each covering 50 by 90 degrees. Determining which tiles are needed to cover a particular region generally requires consulting an index map. This function takes the place of such a reference by returning the file names for a given geographic region.

## Examples

Which tiles are needed for southern Louisiana?

```
latlim =[28.61 31.31];
lonlim = [-91.24 -88.62];
globedems(latlim,lonlim)
```

```
ans =
    'e10g'
    'f10g'
```

# More About

### Tips

The `globedems` function reads data from the format GLOBE Version 1.0. `globedem` first reads the corresponding ESRI header file found in the subdirectory `'/esri/hdr/'` and then the binary data file (with no extension).

## See Also
globedem

# gpxread

Read GPX file

## Syntax

```
P = gpxread(filename)
P = gpxread(URL)

S = gpxread( ___ ,'Index',V)

___ = gpxread( ___ ,Name,Value)
```

## Description

`P = gpxread(filename)` reads point data from the GPS Exchange Format (GPX) file, `filename`, and returns an *n*-by-1 geopoint vector, `P`, where *n* is the number of waypoints, or points that define a route or track.

`gpxread` searches the file first for waypoints, then routes, and then tracks, and it returns the first type of data it finds. The Metadata field of `P` contains a string that identifies the feature type (`'waypoint'`, `'track'`, or `'route'`) and any additional metadata associated with waypoint, route, or track. If the file contains multiple tracks or routes, `P` contains the points that define the first track or route in the file. If `gpxread` cannot find any features in the file, it returns an empty geopoint vector.

`P = gpxread(URL)` reads the GPX data from a `URL`. The `URL` must include the protocol type (for example, `http://`).

`S = gpxread( ___ ,'Index',V)` returns data from the GPX file in a geoshape vector, rather than a geopoint vector, only if the file contains track or route data and you specify the value of `'Index'` as a vector, `V`. Use this syntax when you want to work with the data as a line, rather than as a collection of points.

`___ = gpxread( ___ ,Name,Value)` reads data from a GPX file with additional options, specified by one or more `Name,Value` pair arguments, that control various

characteristics of the import. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`) and is case insensitive. You can specify several name-value pair arguments in any order.

## Examples

### Read Waypoints and Display Them Over Image

Read and display waypoints from the `boston_placenames.gpx` file and overlay the points onto the `boston.tif` image.

Read waypoints from a GPX file.

```
p = gpxread('boston_placenames');
```
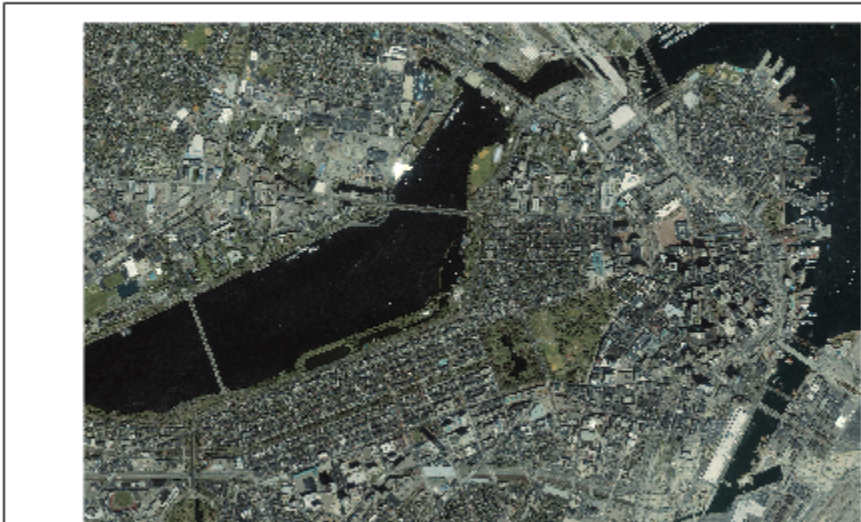
Read the image file.

```
[A,R] = geotiffread('boston');
```

Convert the length unit of the *X* and *Y* limits to meters for use with the projection structure.

```
proj = geotiffinfo('boston');
mstruct = geotiff2mstruct(proj);
R.XWorldLimits = R.XWorldLimits * proj.UOMLengthInMeters;
R.YWorldLimits = R.YWorldLimits * proj.UOMLengthInMeters;
```
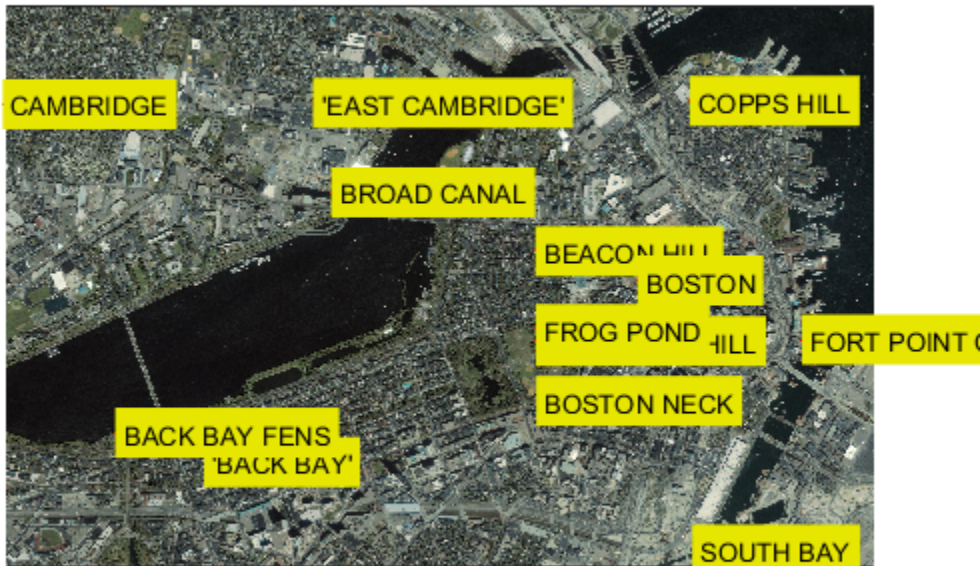
Display the map.

```
figure;
axesm(mstruct)
mapshow(A,R);
```

Display the names and positions of each point, overlaid over the image.

```
for k=1:length(p)
   textm(p(k).Latitude, p(k).Longitude, p(k).Name, ...
      'Color',[0 0 0],'BackgroundColor',[0.9 0.9 0],...
      'Interpreter','none');
end
geoshow(p);
xlim(R.XWorldLimits)
ylim(R.YWorldLimits)
```

### Read and Display a Route

Read and display a route from Boston Logan International Airport to Mathworks in Natick, MA.

Read the route information from the GPX file.

```
route = gpxread('sample_route');
```

Compute `latlim` and `lonlim` with a 0.05 buffer.

```
[latlim, lonlim] = geoquadline(route.Latitude, route.Longitude);
[latlim, lonlim] = bufgeoquad(latlim, lonlim, .05, .05);
```

Display the route.

```
fig = figure;
pos = fig.Position;
fig.Position = [300 300 1.25*pos(3) 1.25*pos(4)];
ax = usamap(latlim, lonlim);
setm(ax, 'MLabelParallel', 43.5)
geoshow(route.Latitude, route.Longitude)
```



Extract the elements of route that include descriptions of turns, mark and color code each turn on the map, and construct a legend that displays the descriptions. Reverse the order, so that the legend displays the first turn at the top and the last at the bottom.

```matlab
turns = route(~cellfun(@isempty, route.Description));
turns = turns(end:-1:1);
n = length(turns);
colors = cool(n);
for k=1:n
    geoshow(turns(k).Latitude, turns(k).Longitude, ...
        'DisplayType','point','MarkerEdgeColor',colors(k,:),...
        'Tag','turn','DisplayName',turns(k).Description)
end
legend(findobj(ax,'Tag','turn'),'Location','SouthOutside')
```
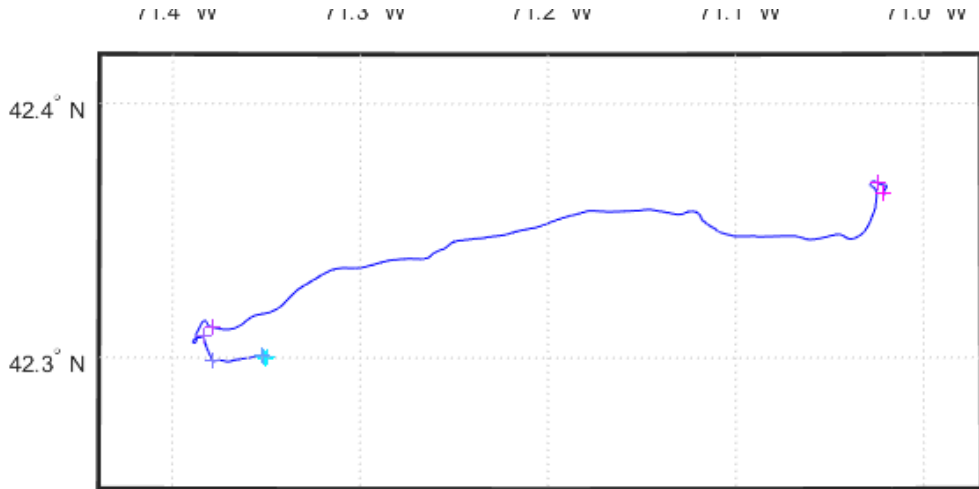
| + | Head southeast |
|---|---|
| + | Keep left at the fork, follow signs for I-90 W/I-93 S/Williams Tunnel/Mass Pike and merge onto I-90 W |
|   | Partial toll road |
| + | Take exit 13 to merge onto MA-30 E/Cochituate Rd toward Natick |
|   | Partial toll road |
| + | Turn right onto Speen St |
| + | Merge onto MA-9 E/Worcester St via the ramp on the left to Boston |
| + | Slight right onto Apple Hill Dr |
| + | Turn left toward Apple Hill Dr |
| + | Take the 1st right onto Apple Hill Dr |
|   | Destination will be on the right |
| + | The MathWorks, Inc., Natick, MA. |

### Read and Display Multiple Track Logs on a Webmap

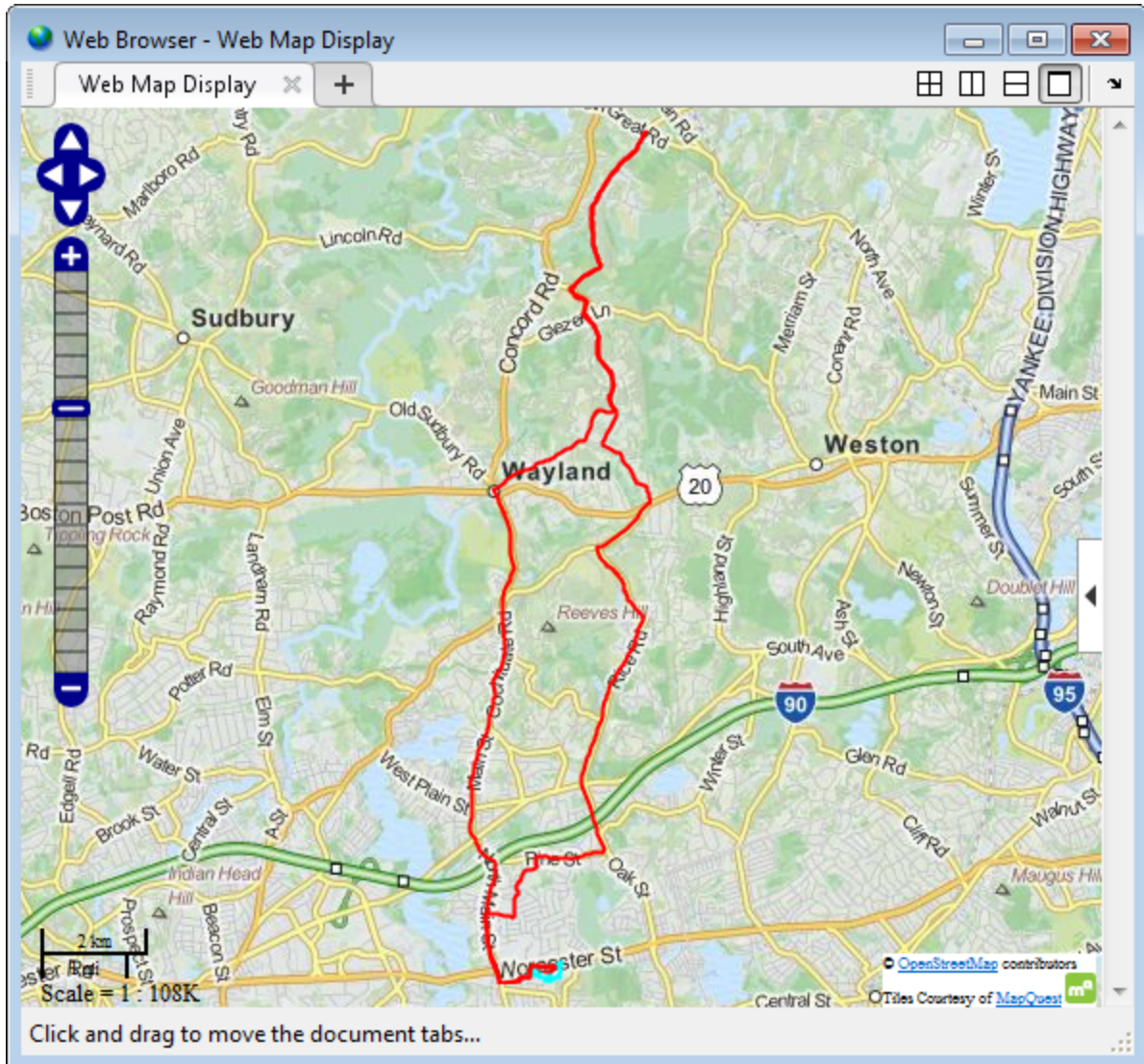Read track log from a GPX file and display overlaid on a webmap.

Read the track logs from a GPX file.

```
tracks = gpxread('sample_tracks', 'Index', 1:2);
```

Display the track logs on a webmap with a different color for each track log.

```
webmap('mapquestopenstreetmap')
colors = {'cyan', 'red'};
```
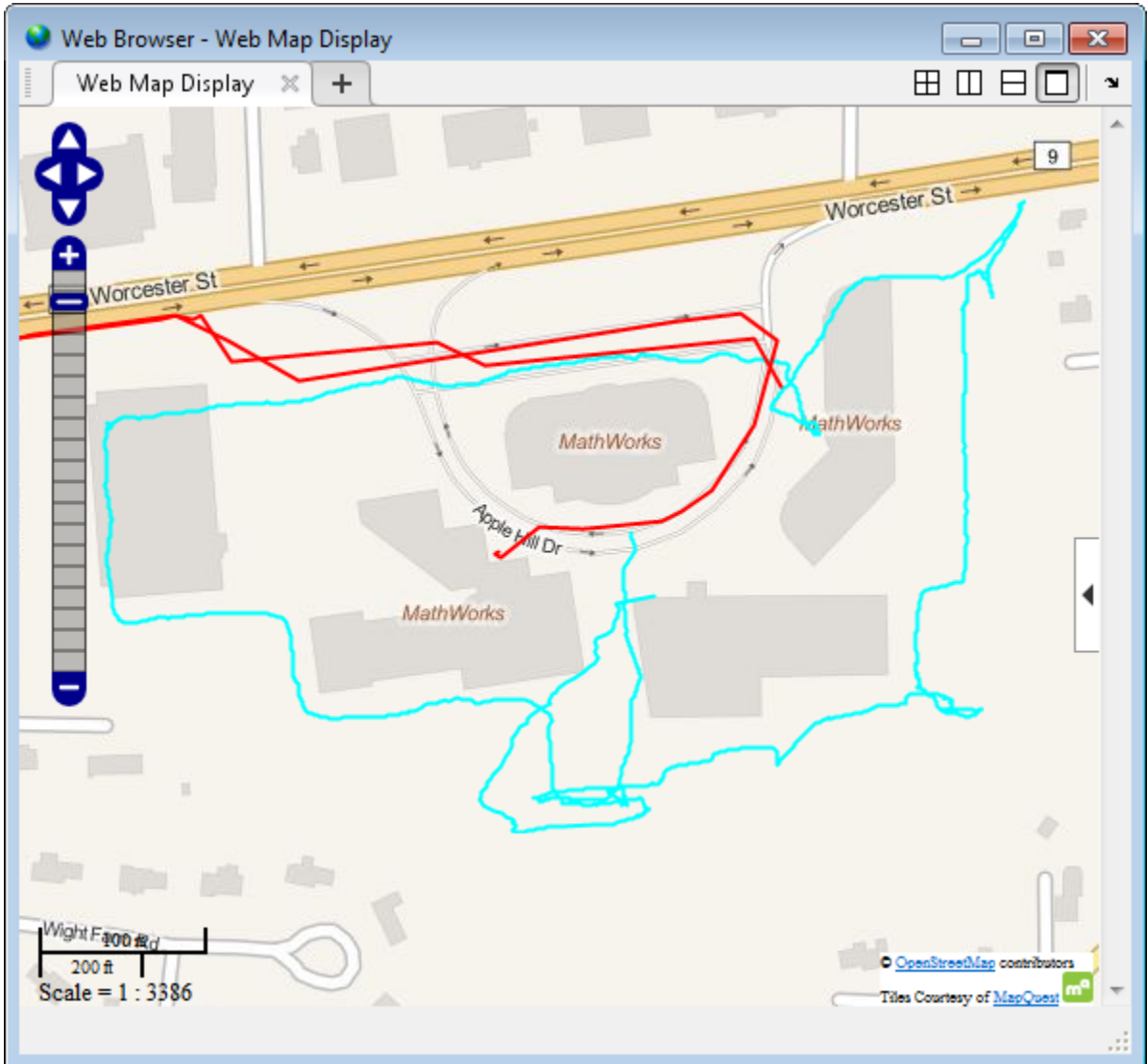
```
wmline(tracks, 'Color', colors)
```



Zoom the webmap to view the first track near the MathWorks campus in Natick.

```
[latlim, lonlim] = geoquadline(tracks(1).Latitude, tracks(1).Longitude);
```
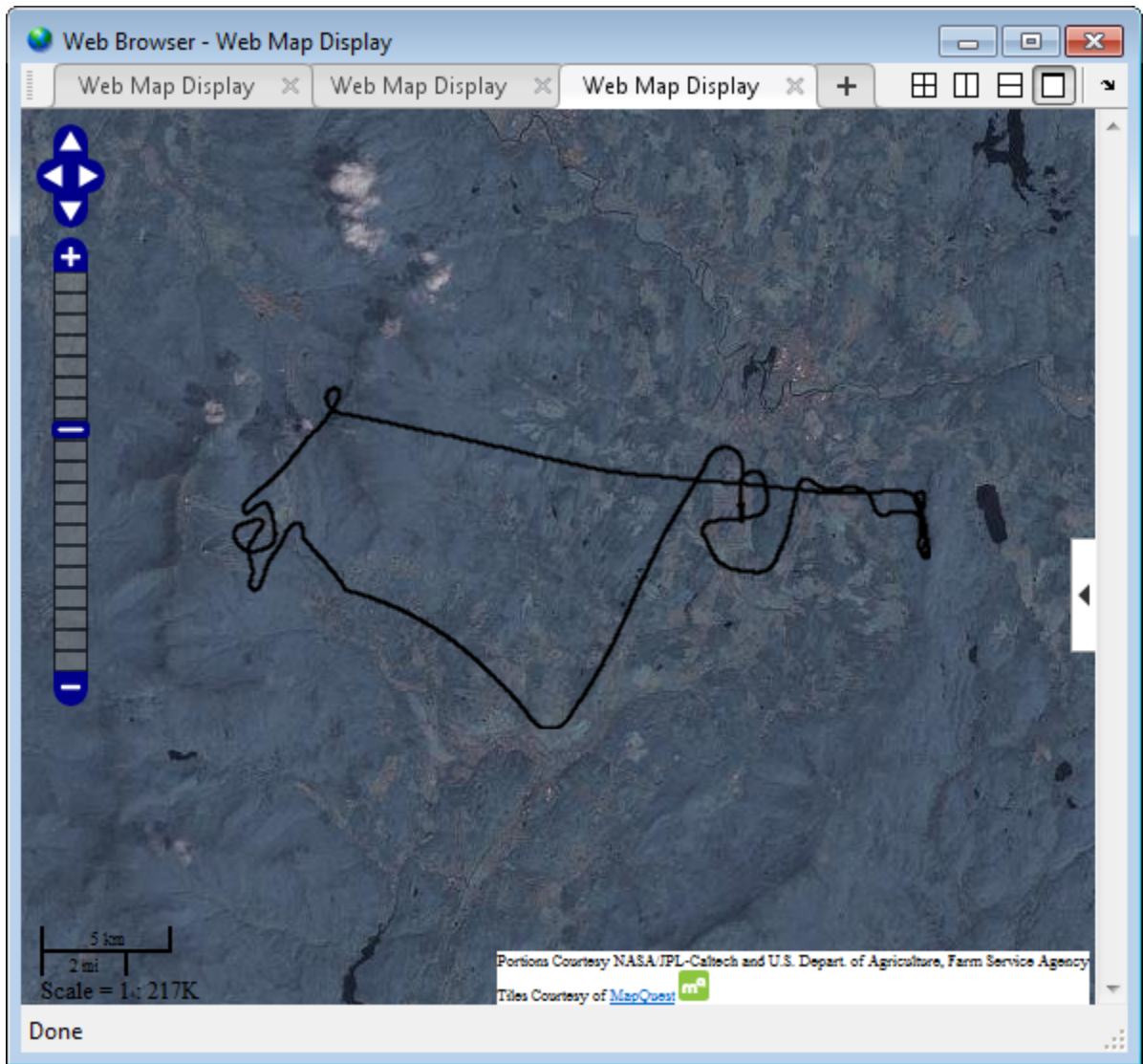
```
wmlimits(latlim, lonlim)
```



**Read and Display Waypoints and Track Log on a Webmap**

Read waypoints and track log from the `sample_mixed.gpx` file.

```
wpt = gpxread('sample_mixed');
trk = gpxread('sample_mixed', 'FeatureType', 'track');
```
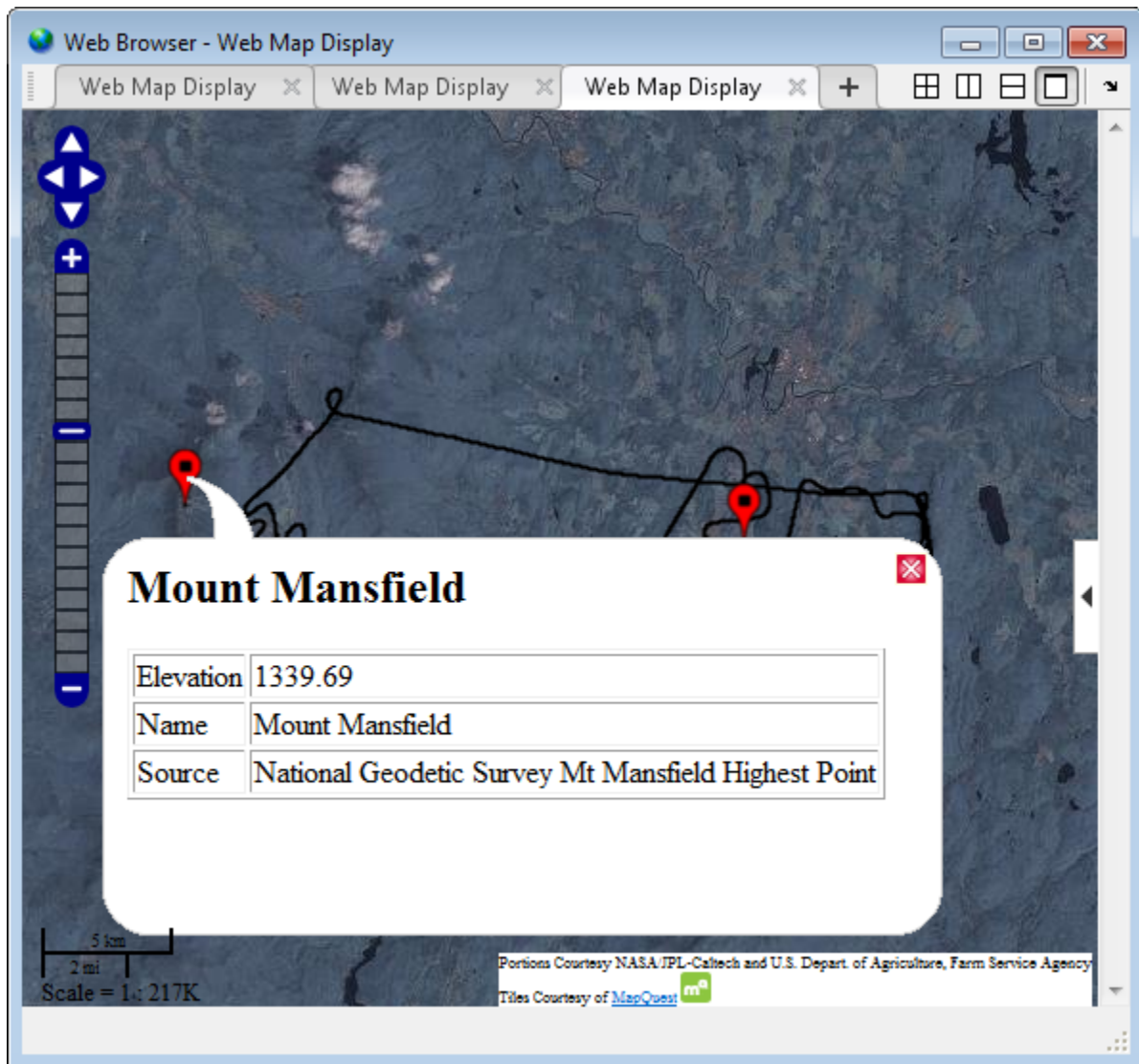
Display the waypoints and the track log on a webmap.

```
webmap('mapquestopenaerialmap')
wmline(trk, 'OverlayName', 'Track Logs');
```

Add web markers to mark the positions of each way point.

```
wmmarker(wpt, 'FeatureName', wpt.Name, 'OverlayName', 'Waypoints')
```

### Display Elevation and Time-area Maps

Display elevation and time area maps and calculate distance using track logs.

Read track log from the sample_mixed.gpx file.

```
trk = gpxread('sample_mixed', 'FeatureType', 'track');
```

Convert the time value strings to serial date numbers using `datenum`, and then compute the time-of-day in hours-minutes-seconds.

```
timeStr = strrep(trk.Time, 'T', ' ');
timeStr = strrep(timeStr, '.OOOZ', '');
trk.DateNumber = datenum(timeStr, 31);
day = fix(trk.DateNumber(1));
trk.TimeOfDay = trk.DateNumber - day;
```

Display an area plot of the elevation and time values.

```
figure
area(trk.TimeOfDay, trk.Elevation)
datetick('x', 13, 'keepticks', 'keeplimits')
ylabel('elevation (meters)')
xlabel('time(Z) hours:minutes:seconds')
title({'Elevation Area Plot', datestr(day)});
```

**Elevation Area Plot**
**28-Aug-2010**
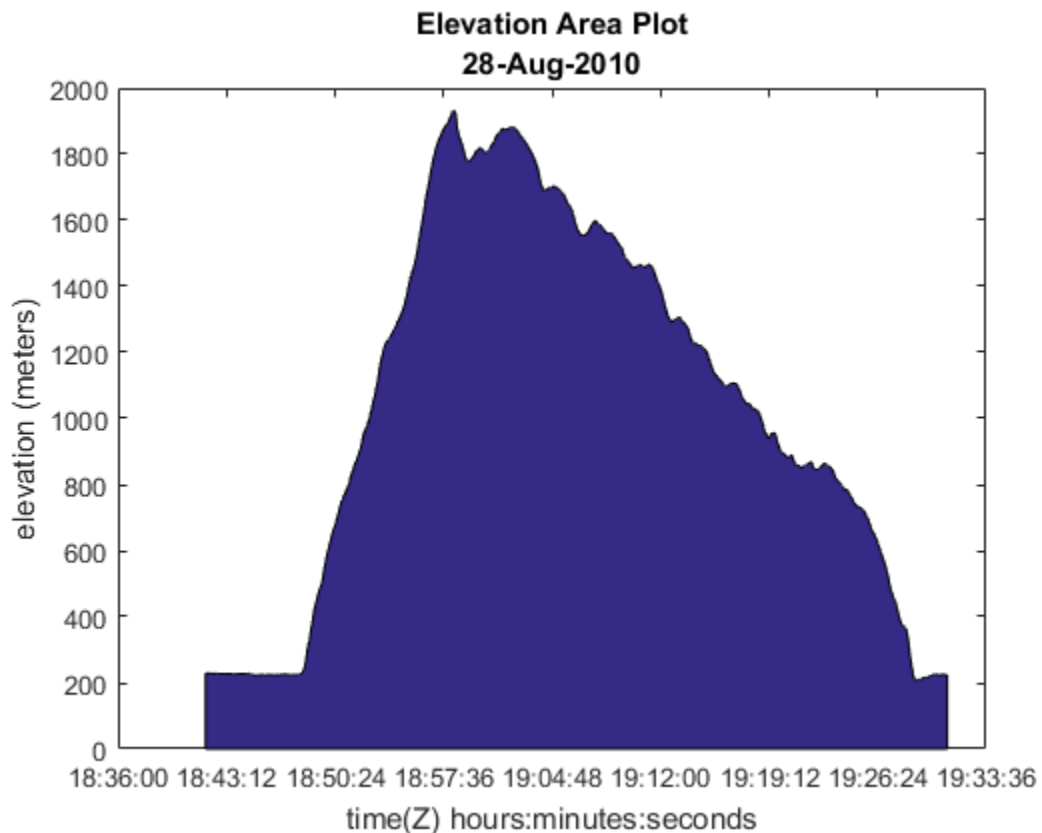


Calculate and display ground track distance. Convert distance in meter to distance in U.S. survey mile.

```
e = wgs84Ellipsoid;
lat = trk.Latitude;
lon = trk.Longitude;
d = distance(lat(1:end-1), lon(1:end-1), lat(2:end), lon(2:end), e);
d = d * unitsratio('sm', 'meter');
```

Display the cumulative ground track distance and elapsed time.

```
trk.ElapsedTime = trk.TimeOfDay - trk.TimeOfDay(1);
figure
line(trk.ElapsedTime(2:end), cumsum(d))
```

**1-617**

```
datetick('x', 13)
ylabel('cumulative ground track distance (statute mile)')
xlabel('elapsed time  (hours:minutes:seconds)')
title({'Cumulative Ground Track Distance in Miles', datestr(day),  ...
    ['Total Distance in Miles: ' num2str(sum(d))]});
```

**Cumulative Ground Track Distance in Miles**
**28-Aug-2010**
**Total Distance in Miles: 46.0637**
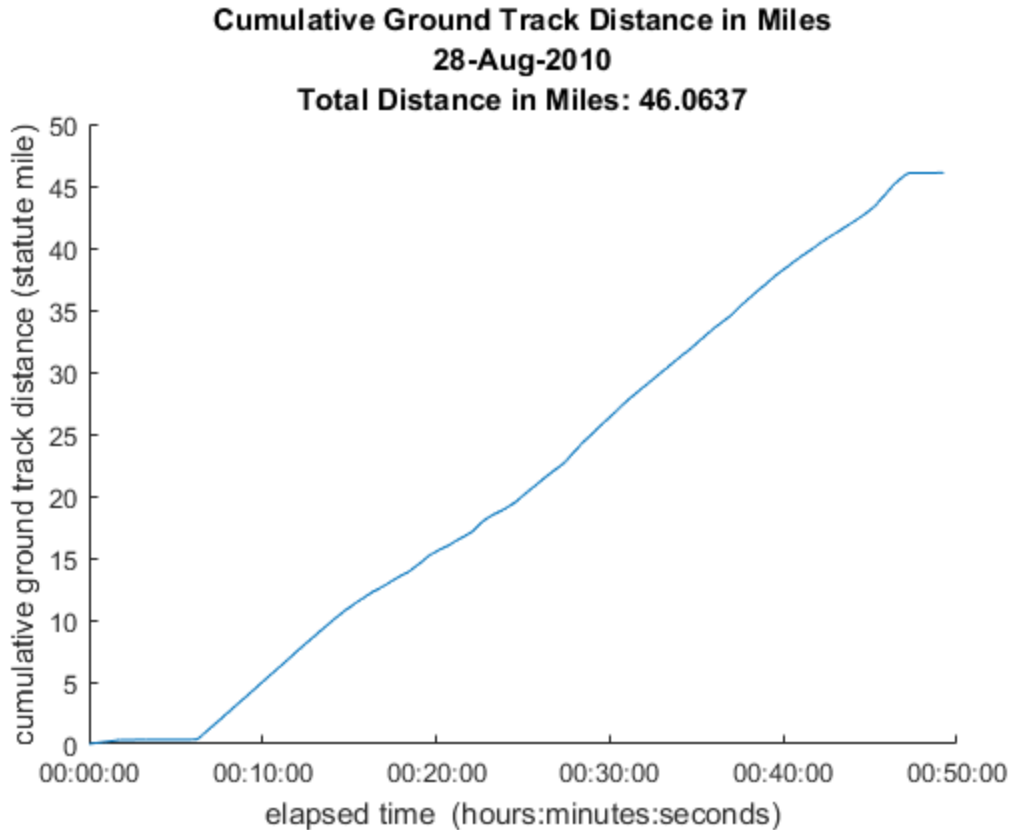


## Input Arguments

**`filename` — Name of GPX file to open**
character string

Name of GPX file to open, specified as a character string. If the file is not in the current folder or in a folder on the MATLAB path, you must specify the folder path. If the file

name includes the extension `'.gpx'` (either uppercase or lowercase), you can omit the extension from `filename`.

Example: `'boston_placenames'`

Data Types: `char`

### URL — Internet location containing GPX data
Uniform Resource Locator (URL)

Internet location containing GPX data, specified as a Uniform Resource Locator (URL). The URL must include protocol type (for example, `http://`).

Data Types: `char`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FeatureType','track'`

### `'FeatureType'` — Type of feature to read from file
`'auto'` (default) | `'track'` | `'route'` | `'waypoint'`

Type of feature to read from file, specified as one of the following character strings: `'track'`, `'route'`, `'waypoint'`, or `'auto'`. If `gpxread` cannot find the specified feature in the file, it returns an empty geopoint vector.

Example: `'FeatureType','waypoint'`

Data Types: `char`

### `'Index'` — Index of waypoint, track, or route data in file
scalar or vector of positive integers

Index of waypoint, track, or route data in file, specified as a scalar or vector of positive integers.

- If the value is a scalar, `gpxread` returns the specified waypoint, route, or track as a geopoint vector. If the scalar value is greater than the total number of elements found in the file, `gpxread` returns an empty geopoint vector.

- If the value is a vector, and the file contains waypoints, gpxread returns those waypoints specified by the vector. If the file contains routes or tracks (and does not contain waypoints), gpxread returns the specified routes or track logs in a geoshape vector. gpxread sets the Geometry field of the geoshape vector to 'line'.

Example: 'Index', [1:2] would read up to two routes or tracks, if the file contained routes or tracks, in a geoshape vector.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

## Output Arguments

### P — Waypoint, track, or route data
*n*-by-1 geopoint vector

Waypoint, track, or route data, returned as an *n*-by-1 geopoint vector, where *n* is the number of points.

For a track log or route with multiple segments, gpxread concatenates the coordinates of the segments with NaN separators. NaN denotes numeric elements not found in the file. The empty string (' ') is used to denote string elements not found in the file.

### S — Track or route data
*n*-by-1 geoshape vector

Track or route data, returned as an *n*-by-1 geopoint vector

## More About

### waypoint

A point of interest, or named feature on a map.

### track

An ordered list of waypoints that describe a path.

### route

An ordered list of waypoints representing a series of turn points leading to a destination.

**Tips**

- Excluding extensions, GPX version 1.1 is fully supported. If any other version is detected, a warning is issued. However, in most cases, version 1.0 GPX files can be read successfully unless they contain certain metadata tags. For more information, see the GPX 1.1 Schema Documentation.

## See Also

geopoint | geoshape | shaperead

# gradientm

Calculate gradient, slope and aspect of data grid

## Syntax

```
[ASPECT, SLOPE, gradN, gradE] = gradientm(Z, R)
[...] = gradientm(lat, lon, Z)
[...] = gradientm(..., ellipsoid)
[...] = gradientm(lat, lon, Z, ellipsoid, units)
```

## Description

`[ASPECT, SLOPE, gradN, gradE] = gradientm(Z, R)` computes the slope, aspect and north and east components of the gradient for a regular data grid `Z` with three-element referencing vector `refvec`. If the grid contains elevations in meters, the resulting aspect and slope are in units of degrees clockwise from north and up from the horizontal. The north and east gradient components are the change in the map variable per meter of distance in the north and east directions. The computation uses finite differences for the map variable on the default earth ellipsoid.

`R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a

parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

`[...] = gradientm(lat, lon, Z)` does the computation for a geolocated data grid. `lat` and `lon`, the latitudes and longitudes of the geolocation points, are in degrees.

`[...] = gradientm(..., ellipsoid)` uses the reference ellipsoid specified by the input `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. If the map contains elevations in the same units of length as the semimajor axis of the ellipsoid, the slope and aspect are in units of degrees. This calling form is most useful for computations on bodies other than the earth.

`[...] = gradientm(lat, lon, Z, ellipsoid, `*`units`*`)` specifies the angle units of the latitude and longitude inputs. If omitted, `'degrees'` are assumed. For elevation maps in the same units as `ellipsoid(1)`, the resulting slope and aspect are in the specified units. The components of the gradient are the change in the map variable per unit of length, using the same length unit as the semimajor axis of the ellipsoid.

## Examples

Compute and display the slope for the 30 arc-second (10 km) Korea elevation data. Slopes in the Sea of Japan are up to 8 degrees at this grid resolution.

```
load korea
[aspect, slope, gradN, gradE] = gradientm(map, refvec);
worldmap(slope, refvec)
geoshow(slope, refvec, 'DisplayType', 'texturemap')
cmap = cool(10);
demcmap('inc', slope, 1, [], cmap)
colorbar
latlim = getm(gca,'maplatlimit');
lonlim = getm(gca,'maplonlimit');
land = shaperead('landareas',...
  'UseGeoCoords', true, 'BoundingBox', [lonlim' latlim']);
geoshow(land, 'FaceColor', 'none')
set(gca, 'Visible', 'off')
```
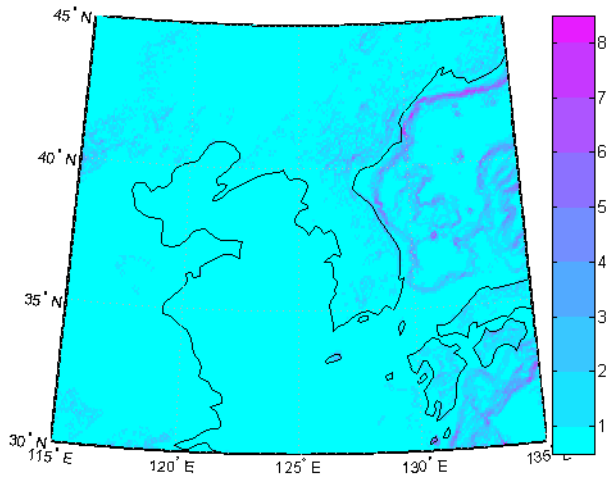
## More About

### Tips

Coarse digital elevation models can considerably underestimate the local slope. For the preceding map, the elevation points are separated by about 10 kilometers. The terrain between two adjacent points is modeled as a linear variation, while actual terrain can vary much more abruptly over such a distance.

· "Geolocated Data Grids"

## See Also
viewshed

# grepfields

Identify matching fields in fixed record length files

## Compatibility

grepfields will be removed in a future release. Use textscan instead.

## Syntax

```
grepfields(filename,searchstring)
grepfields(filename,searchstring,casesens)
grepfields(filename,searchstring,casesens,startcol)
grepfields(filename,searchstring,casesens,startfield,fields)
grepfields(filename,searchstring,casesens,startfield,fields,
machineformat)
indx = grepfields(...)
```

## Description

grepfields(*filename*,*searchstring*) displays lines in the file that begin with the search string. The file must have fixed-length records with line endings.

grepfields(*filename*,*searchstring*,casesens), with casesens 'matchcase', specifies a case-sensitive search. If omitted or 'none', the search string matches regardless of the case.

grepfields(*filename*,*searchstring*,casesens,startcol) searches starting with the specified column. startcol is an integer between 1 and the bytes per record in the file. In this calling form, the file is regarded as a text file with line endings.

grepfields(*filename*,*searchstring*,casesens,startfield,fields) searches within the specified field. startfield is an integer between 1 and the number of fields per record. The format of the file is described by the fields structure. See readfields for recognized fields structure entries. In this calling form, the file can be binary and lack line endings. The search is within startfield, which must be a character field.

grepfields(*filename*,*searchstring*,casesens,startfield,fields,
machineformat) opens the file with the specified machine format. machineformat
must be recognized by fopen.

indx = grepfields(...) returns the record numbers of matched records instead of
displaying them on screen.

## Examples

Write a binary file and read it:

```
fid = fopen('testbin','wb');
for i = 1:3
 fwrite(fid,['character' num2str(i) ],'char');
 fwrite(fid,i,'int8');
 fwrite(fid,[i i],'int16');
 fwrite(fid,i,'integer*4');
 fwrite(fid,i,'real*8');
end
fclose(fid);

fs(1).length = 10;fs(1).type = 'char';fs(1).name = 'field 1';
fs(2).length = 1;fs(2).type = 'int8';fs(2).name = 'field 2';
fs(3).length = 2;fs(3).type = 'int16';fs(3).name = 'field 3';
fs(4).length = 1;fs(4).type = 'integer*4';fs(4).name = 'field 4';
fs(5).length = 1;fs(5).type = 'float64';fs(5).name = 'field 5';
```

Find the record matching the string 'character2'. The record contains binary data,
which cannot be properly displayed.

```
grepfields('testbin','character2','none',1,fs)
character2? ? ?   ?@

indx = grepfields('testbin','character2','none',1,fs)
indx =
     2
```

Read the formatted file containing the following:

```
-------------------------------------------------------
character data 1  1  2  3 1e6 10D6
```

```
character data 2 11 22 33 2e6 20D6

character data 3111222333 3e6 30D6
--------------------------------------------------------

fs(1).length = 16;fs(1).type = 'char';fs(1).name = 'field 1';
fs(2).length = 3;fs(2).type = '%3d';fs(2).name = 'field 2';
fs(3).length = 1;fs(3).type = '%4g';fs(3).name = 'field 3';
fs(4).length = 1;fs(4).type = '%5D'; fs(4).name = 'field 4';
fs(5).length = 1;fs(5).type = 'char';fs(5).name = '';
```

Find the records that match at the beginning of the line.

```
grepfields('testfile1','character')
character data 1  1  2  3 1e6 10D6
character data 2 11 22 33 2e6 20D6
character data 3111222333 3e6 30D6

grepfields('testfile1','character data 2')
character data 2 11 22 33 2e6 20D6
```

Find the records that match, starting the search in column 11.

```
grepfields('testfile1','data 2','none',11)
character data 2 11 22 33 2e6 20D6
```

Search record number 1.

```
grepfields('testfile1','character data 2','none',1,fs)
character data 2 11 22 33 2e6 20D6
```

## Limitations

Searches are limited to fields containing character data.

## More About

### Tips

See `readfields` for a complete discussion of the format and contents of the `fields` argument.

## See Also

`readfields` | `fopen`

# gridm

Toggle and control display of graticule lines

## Syntax

```
gridm
gridm('on')
gridm('off')
gridm('reset')
gridm(linespec)
gridm(MapAxesPropertyName, PropertyValue,...)
h = gridm(...)
```

## Description

gridm toggles the display of a latitude-longitude graticule. The choice of meridians and parallels, as well as their graphics properties, depends on the property settings of the map axes.

gridm('on') creates the graticule, if it does not yet exist, and makes it visible.

gridm('off') makes the graticule invisible.

gridm('reset') redraws the graticule using the current map axes properties.

gridm(linespec) uses any valid linespec string to control the graphics properties of the lines in the graticule.

gridm(MapAxesPropertyName, PropertyValue,...) sets the appropriate graticule properties to the desired values. For a description of these property names and values, see the "Properties That Control the Grid" on page 1-57 section of the axesm reference page.

h = gridm(...) returns the handles of the graticule lines. If both parallels and meridians exist, then h is a two-element vector: h(1) is the handle to the line comprising the parallels, and h(2) is the handle to the line comprising the meridians.

## More About

### Tips

- You can also create or alter map grid properties using the `axesm` or `setm` functions.
- By default the Clipping property is set to `'off'`. Override this setting with the following code:

```
hgrat = gridm('on');
set(hgrat,'Clipping','on')
```

### See Also

axesm | setm

# grid2image

Display regular data grid as image

## Syntax

```
grid2image(Z,R)
grid2image(Z,R,'PropertyName',PropertyValue,...)
h = grid2image(...)
```

## Description

`grid2image(Z,R)` displays a regular data grid `Z` as an image. The image is displayed in unprojected form, with longitude as *x* and latitude as *y*, producing considerable distortion away from the Equator. `Z` can be *M*-by-*N* or *M*-by-*N*-by-3, and can contain `double`, `uint8`, or `uint16` data. The grid is georeferenced to latitude-longitude by `R`, which can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)` and its `RasterInterpretation` must be `'cells'`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel.

`grid2image(Z,R,'PropertyName',PropertyValue,...)` uses the specified image properties to display the map. See the `image` function reference page for a list of properties that can be changed.

`h = grid2image(...)` returns the handle of the image object displayed.

# Examples

Display a regular data grid as an image.

```
load topo
R = georasterref('RasterSize', size(topo), ...
   'Latlim', [-90 90], 'Lonlim', [O 360]);
figure; grid2image(topo, R)
```
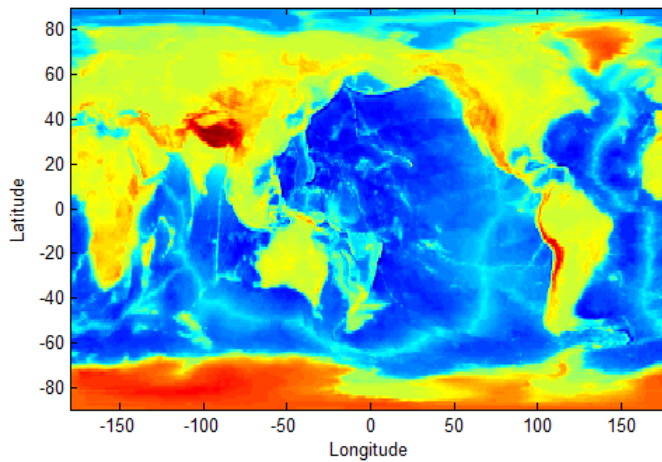


## See Also

`image` | `mapshow` | `mapview` | `meshm` | `surfacem` | `surfm`

# grn2eqa

Convert from Greenwich to equal area coordinates

## Syntax

```
[x,y] = grn2eqa(lat,lon)
[x,y] = grn2eqa(lat,lon,origin)
[x,y] = grn2eqa(lat,lon,origin,ellipsoid)
[x,y] = grn2eqa(lat,lon,origin,units)
mat = grn2eqa(lat,lon,origin...)
```

## Description

`[x,y] = grn2eqa(lat,lon)` converts the Greenwich coordinates `lat` and `lon` to the equal-area coordinate points `x` and `y`.

`[x,y] = grn2eqa(lat,lon,origin)` specifies the location in the Greenwich system of the *x-y* origin (0,0). The two-element vector `origin` must be of the form `[latitude, longitude]`. The default places the origin at the Greenwich coordinates (0º,0º).

`[x,y] = grn2eqa(lat,lon,origin,ellipsoid)` specifies the ellipsoidal model of the figure of the Earth using `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The `ellipsoid` is spherical by default.

`[x,y] = grn2eqa(lat,lon,origin,units)` specifies the units for the inputs, where *units* is any valid angle units string. The default value is `'degrees'`.

`mat = grn2eqa(lat,lon,origin...)` packs the outputs into a single variable.

The `grn2eqa` function converts data from Greenwich-based latitude-longitude coordinates to equal-area *x-y* coordinates. The opposite conversion can be performed with `eqa2grn`.

## Examples

```
lats = [56 34]; longs = [-140 23];
```

```
[x,y] = grn2eqa(lats,longs)

x =
   -2.4435    0.4014
y =
    0.8290    0.5592
```

## See Also

eqa2grn | hista

# gshhs

Read Global Self-Consistent Hierarchical High-Resolution Shoreline

## Syntax

```
S = gshhs(filename)
S = gshhs(filename, latlim, lonlim)
indexfilename = gshhs(filename, 'createindex')
```

## Description

`S = gshhs(filename)` reads GSHHS vector data for the entire world from `filename`. GSHHS files must have names of the form `gshhs_x.b`, `wdb_borders_x.b`, or `wdb_rivers_x.b`, where *x* is one of the letters `c`, `l`, `i`, `h` or `f`, corresponding to increasing resolution (and file size). The result returned in `S` is a polygon or line geographic data structure array (a *geostruct*, with `'Lat'` and `'Lon'` coordinate fields).

`S = gshhs(filename, latlim, lonlim)` reads a subset of the vector data from `filename`. The limits of the desired data are specified as two-element vectors of latitude, `latlim`, and longitude, `lonlim`, in degrees. The elements of `latlim` and `lonlim` must be in ascending order. Longitude limits range from `[-180 195]`. If `latlim` is empty the latitude limits are `[-90 90]`. If `lonlim` is empty, the longitude limits are `[-180 195]`.

`indexfilename = gshhs(filename, 'createindex')` creates an index file for faster data access when requesting a subset of a larger dataset. The index file has the same name as the GSHHS data file, but with the extension `'i'`, instead of `'b'` and is written in the same folder as `filename`. The name of the index file is returned, but no coastline data are read. A call using this option should be followed by an additional call to `gshhs` to import actual data. On that and subsequent calls, `gshhs` detects the presence of the index file and uses it to access records by location much faster than it would without an index.

## Output Structure

The output structure `S` contains the following fields. All latitude and longitude values are in degrees.

| Field Name | Field Contents |
|---|---|
| `'Geometry'` | `'Line'` or `'Polygon'` |
| `'BoundingBox'` | `[minLon minLat; maxLon maxLat]` |
| `'Lon'` | Coordinate vector |
| `'Lat'` | Coordinate vector |
| `'South'` | Southern latitude boundary |
| `'North'` | Northern latitude boundary |
| `'West'` | Western longitude boundary |
| `'East'` | Eastern longitude boundary |
| `'Area'` | Area of polygon in square kilometers |
| `'Level'` | Scalar value ranging from 1 to 4, indicates level in topological hierarchy |
| `'LevelString'` | `'land'`, `'lake'`, `'island_in_lake'`, `'pond_in_island_in_lake'`, or `''` |
| `'NumPoints'` | Number of points in the polygon |
| `'FormatVersion'` | Format version of data file. Positive integer for versions 3 and later; empty for versions 1 and 2. |
| `'Source'` | Source of data: `'WDBII'` or `'WVS'` |
| `'CrossesGreenwich'` | Scalar flag: `true` if the polygon crosses the prime meridian; `false` otherwise |
| `'GSHHS_ID'` | Unique polygon scalar id number, starting at 0 |

For releases 2.0 and higher (`FormatVersion` 7 and higher), the following additional fields are included in the output structure:

| Field Name | Field Contents |
|---|---|
| `'RiverLake'` | Scalar flag: `true` if the polygon is the fat part of a major river and the `Level` value is set to 2; `false` otherwise. |
| `'AreaFull'` | Area of original full-resolution polygon in units $\frac{1}{10}km^2$. |

| Field Name | Field Contents |
|---|---|
| `'Container'` | ID of container polygon that encloses this polygon. Set to -1 to indicate none. |
| `'Ancestor'` | ID of ancestor polygon in the full resolution set that was the source of this polygon. Set to -1 to indicate none. |

For Release 2.2 and higher (`FormatVersion` 9 and higher) the following additional field is included in the output structure:

| Field Name | Field Contents |
|---|---|
| `'CrossesDateline'` | Scalar flag: `true` if the polygon crosses the dateline; `false` otherwise. |

## Background

The Global Self-Consistent Hierarchical High-Resolution Shoreline was created by Paul Wessel of the University of Hawaii and Walter H.F. Smith of the NOAA Geosciences Lab. At the full resolution the data requires 85 MB uncompressed, but lower resolution versions are also provided. This database includes coastlines, major rivers, and lakes. The GSHHS data in various resolutions is available over the Internet from the National Oceanic and Atmospheric Administration, National Geophysical Data Center Web site.

Version 3 (Release 1.3) of the `gshhs_c.b` (coarse) data set ships with the toolbox in the `toolbox/map/mapdata` folder. For details, type

```
type gshhs_c.txt
```
at the MATLAB command prompt. The `gshhs` function has been qualified on GSHHS releases 1.1 through 2.1 (version 8). It should also be able to read newer versions, if they adhere to the same header format as releases 2.0 and 2.1.
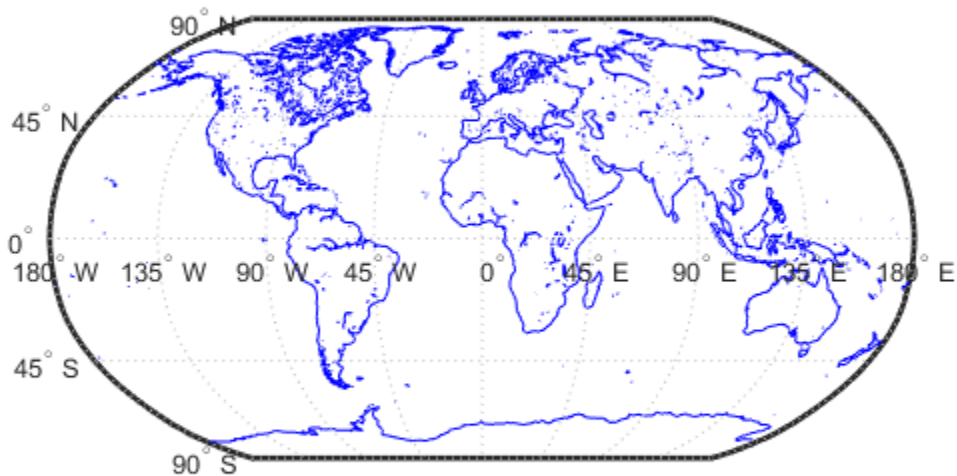
## Examples

### Read Entire Coarse Data Set and Display Levels

Read the entire coarse data set. The example uses a coarse data set because the toolbox includes a sample coarse data set in the `toolbox/map/mapdata` folder. You can use similar code to read intermediate, full-resolution, or high resolution GSHHS data sets.

```
filename = gunzip('gshhs_c.b.gz', tempdir);
shorelines = gshhs(filename{1});
delete(filename{1})
```

Display data as a coastline.

```
figure
worldmap world
geoshow([shorelines.Lat], [shorelines.Lon])
```
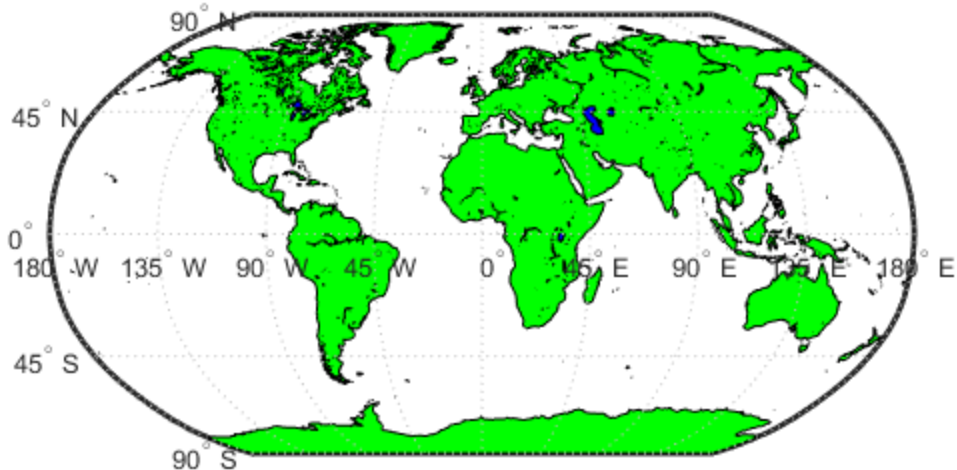


Display each level in a different color.

```
levels = [shorelines.Level];
land = (levels == 1);
```

```
lake = (levels == 2);
island = (levels == 3);
figure
worldmap world
geoshow(shorelines(land),   'FaceColor', [0 1 0])
geoshow(shorelines(lake),   'FaceColor', [0 0 1])
geoshow(shorelines(island),'FaceColor', [1 1 0])
```



### Read GSHHS Dataset Using an Index

Read the entire coarse data set, creating an index. The example uses a coarse data set
because the toolbox includes a sample coarse data set in the toolbox/map/mapdata
folder. You can use similar code to read intermediate, full-resolution, or high resolution
GSHHS data sets.

```
filename = gunzip('gshhs_c.b.gz', tempdir);
indexname = gshhs(filename{1}, 'createindex');
```

Display Africa as a green polygon. Note that gshhs detects and uses the index file automatically.

```
figure
worldmap Africa
projection = gcm;
latlim = projection.maplatlimit;
lonlim = projection.maplonlimit;
africa = gshhs(filename{1}, latlim, lonlim);
delete(filename{1})
delete(indexname)
```
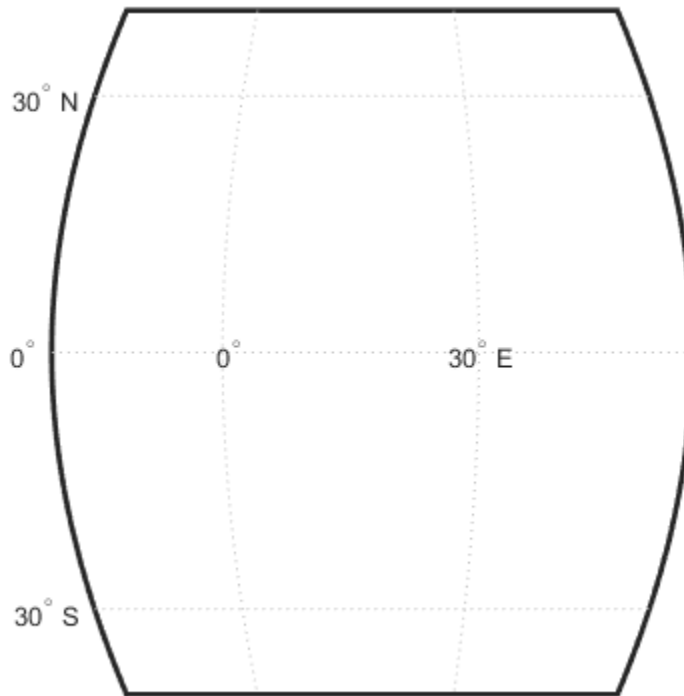
Sort by descending level to keep smaller level 2 and level 3 features on top.

```
[~,ix] = sort([africa.Level],'descend');
africa = africa(ix);
geoshow(africa, 'FaceColor', 'green')
setm(gca, 'FFaceColor', 'cyan')
```
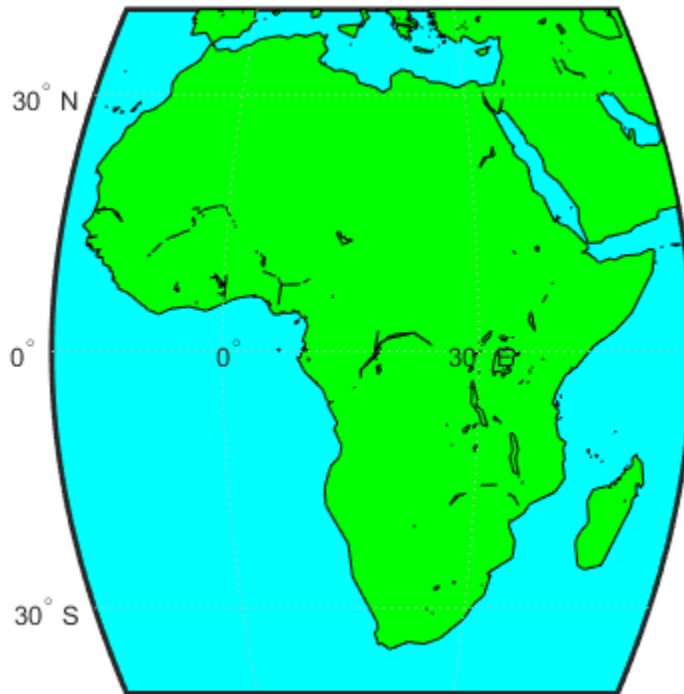


## More About

### Tips

- If you are extracting data within specified geographic limits and using data other than coarse resolution, consider creating an index file first. Also, to speed rendering

**1-641**

when mapping very large amounts of data, you might want to plot the data as `NaN`-clipped lines rather than as patches.

- When you specify latitude-longitude limits, polygons that completely fall outside those limits are excluded, but no trimming of features that partially traverse the region is performed. If you want to eliminate data outside of a rectangular region of interest, you can use `maptrimp` with the `Lat` and `Lon` fields of the geostruct returned by `gshhs` to clip the data to your region and still maintain polygon topology.

- You can read the WDB rivers and borders datasets but the `LevelString` field will be empty. The `Level` values vary from feature to feature but the interpretations of these values are not documented as part of the GSHHS distribution and are therefore not converted to strings.

- The following examples use publicly available GSHHS data files that do not ship with the Mapping Toolbox software. For details on locating GSHHS data for download over the Internet, see the following documentation at the MathWorks® Web site: "Finding Geospatial Data".

## See Also
dcwdata | geoshow | maptrimp | shaperead | vmap0data | worldmap

# gtextm

Place text on map using mouse

## Syntax

```
h = gtextm(string)
h = gtextm(string,PropertyName,PropertyValue,...)
```

## Description

`h = gtextm(`*string*`)` places the text object string at the position selected by mouse input. When this function is called, the current map axes are brought up and the cursor is activated for mouse-click position entry. The text object's handle is returned.

`h = gtextm(string,`*PropertyName*`,`*PropertyValue*`,...)` allows the specification of any properties supported by the MATLAB `text` function.

## Examples

Create map axes:

```
axesm('sinusoid','FEdgeColor','red')
gtextm('hello world','FontWeight','bold')
```

Click inside the frame and the text appears.

## See Also

```
axesm | textm
```

# gtopo30

Read 30-arc-second global digital elevation data (GTOPO30)

## Syntax

```
[Z,refvec] = gtopo30(tilename)
[Z,refvec] = gtopo30(tilename,samplefactor)
[Z,refvec] = gtopo30(tilename,samplefactor,latlim,lonlim)
[Z,refvec] = gtopo30(foldername, ...)
```

## Description

`[Z,refvec] = gtopo30(tilename)` reads the GTOPO30 tile specified by `tilename` and returns the result as a regular data grid. `tilename` is a string which does not include an extension and indicates a GTOPO30 tile in the current folder or on the MATLAB path. If `tilename` is empty or omitted, a file browser will open for interactive selection of the GTOPO30 header file. The data is returned at full resolution with the latitude and longitude limits determined from the GTOPO30 tile. The data grid, `Z`, is returned as an array of elevations. Elevations are given in meters above mean sea level using WGS84 as a horizontal datum. `refvec` is the associated referencing vector.

`[Z,refvec] = gtopo30(tilename,samplefactor)` reads a subset of the elevation data from `tilename`. `samplefactor` is a scalar integer, which when equal to 1 reads the data at its full resolution. When `samplefactor` is an integer n greater than one, every nth point is read. If `samplefactor` is omitted or empty, it defaults to 1.

`[Z,refvec] = gtopo30(tilename,samplefactor,latlim,lonlim)` reads a subset of the elevation data from `tilename` using the latitude and longitude limits `latlim` and `lonlim` specified in degrees. `latlim` is a two-element vector of the form:

`[southern_limit northern_limit]`

Likewise, `lonlim` has the form:

`[western_limit eastern_limit]`

If `latlim` and `lonlim` are omitted, the coordinate limits are determined from the file. The latitude and longitude limits are snapped outward to define the smallest possible

rectangular grid of GTOPO30 cells that fully encloses the area defined by the input limits. Any cells in this grid that fall outside the extent of the tile are filled with NaN.

`[Z,refvec] = gtopo30(foldername, ...)` is similar to the syntaxes above except that GTOPO30 data are read and concatenated from multiple tiles within a GTOPO30 CD-ROM or folder structure. The `foldername` input is a string with the name of the folder which contains the GTOPO30 tile folders or GTOPO30 tiles. Within the tile folders are the uncompressed data files. The `foldername` for CD-ROMs distributed by the USGS is the device name of the CD-ROM drive. As with the case with a single tile, any cells in the grid specified by `latlim` and `lonlim` are NaN filled if they are not covered by a tile within `foldername`. `samplefactor` if omitted or empty defaults to 1. `latlim` if omitted or empty defaults to `[-90 90]`. `lonlim` if omitted or empty defaults to `[-180 180]`.

For details on locating GTOPO30 data for download over the Internet, see "Finding Geospatial Data".
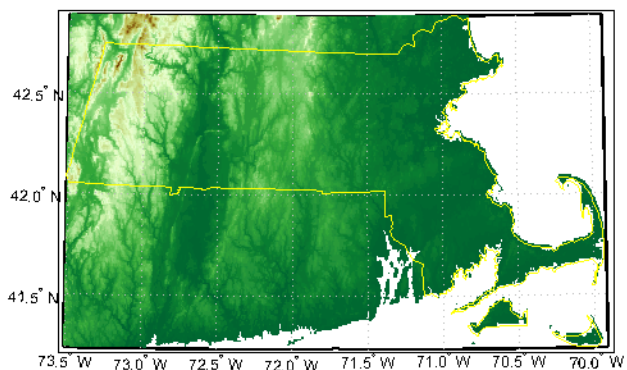
# Examples

## Example 1

Extract and display full resolution data for the state of Massachusetts:

```
% Read the stateline polygon boundary and calculate boundary limits.
Massachusetts = shaperead('usastatehi','UseGeoCoords',true, ...
   'Selector',{@(name) strcmpi(name,'Massachusetts'),'Name'});
latlim = [min(Massachusetts.Lat(:)) max(Massachusetts.Lat(:))];
lonlim = [min(Massachusetts.Lon(:)) max(Massachusetts.Lon(:))];

% Read the GTOPO30 data at full resolution.
[Z,refvec] = gtopo30('W100N90',1,latlim,lonlim);

% Display the data grid and overlay the stateline boundary.
figure
usamap(Z,refvec);
geoshow(Z,refvec,'DisplayType','surface')
demcmap(Z)
geoshow(Massachusetts,'DisplayType','polygon',...
   'facecolor','none','edgecolor','y')
```

## Example 2

```
% Extract every 20th point from a tile.
% Provide an empty filename and select the file interactively.
[Z,refvec] = gtopo30([],20);
```

## Example 3

```
% Extract data for Thailand, an area which straddles two tiles.
% The data is on CD number 3 distributed by the USGS.
% The CD-device is 'F:\'
latlim = [5.22 20.90];
lonlim = [96.72 106.38];
gtopo30s(latlim,lonlim)
% Extract every fifth data point for Thailand.
% Specify actual folder or mapped drive if not "F:\'
[Z,refvec] = gtopo30('F:\',5,latlim,lonlim);
worldmap(Z,refvec);
geoshow(Z,refvec,'DisplayType','surface')
demcmap(Z)
```
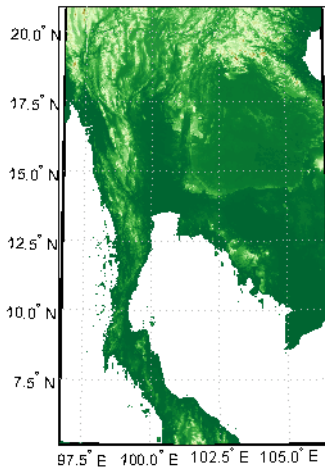
## Example 4

```
% Extract every 10th point from a column of data 5 degrees around
% the prime meridian. The current folder contains GTOPO30 data.
[Z,refvec] = gtopo30(pwd,10,[],[-5 5]);
```

## See Also

gtopo30s | globedem | dted | satbath | tbase | usgsdem

# gtopo30s

GTOPO30 data filenames for latitude-longitude quadrangle

## Syntax

```
tileNames = gtopo30s(latlim,lonlim)
tileNames = gtopo30s(lat,lon)
```

## Description

`tileNames = gtopo30s(latlim,lonlim)` returns a cell array of the tile names covering the geographic region for GTOPO30 digital elevation maps (also referred to as "30-arc second" DEMs). `latlim` and `lonlim` specify the region as two-element vectors of latitude and longitude limits in units of degrees.

`tileNames = gtopo30s(lat,lon)` returns a cell array of the tile names covering the geographic region for GTOPO30 digital elevation maps. `lat` and `lon` specify the region as scalar latitude and longitude points.

### See Also
gtopo30

# handlem

Handles of displayed map objects

## Syntax

```
handlem
handlem('taglist')
handlem('prompt')
h = handlem(object)
h = handlem(tagstr)
h = handlem( ___ ,axesh)
h = handlem( ___ ,axesh,'searchmethod')
h = handlem(handles)
```

## Description

handlem displays a dialog box for selecting objects that have their Tag property set.

handlem('taglist') displays a dialog box for selecting objects that have their Tag property set.

handlem('prompt') displays a dialog box for selecting objects based on the object strings listed below.

h = handlem(object) returns the handles of those objects in the current axes specified by the input string, object. The options for the object string are defined by the following list:

| | |
|---|---|
| 'all' | All children |
| 'clabel' | Contour labels |
| 'contour' | hggroups containing contours |
| 'fillcontour' | hggroups containing filled contours |
| 'frame' | Map frame |
| 'grid' | Map grid lines |

| | |
|---|---|
| `'hggroup'` | All hggroup objects |
| `'hidden'` | Hidden objects |
| `'image'` | Untagged image objects |
| `'light'` | Untagged light objects |
| `'line'` | Untagged line objects |
| `'map'` | All objects on the map, excluding the frame and grid |
| `'meridian'` | Longitude grid lines |
| `'mlabel'` | Longitude labels |
| `'parallel'` | Latitude grid lines |
| `'plabel'` | Latitude labels |
| `'patch'` | Untagged patch objects |
| `'scaleruler'` | Scaleruler objects |
| `'surface'` | Untagged surface objects |
| `''text''` | Untagged text objects |
| `'tissot'` | Tissot indicatrices |
| `'visible'` | Visible objects |

`h = handlem(`*tagstr*`)` returns the handles for any objects whose tags match the string `tagstr`.

`h = handlem( ___ ,axesh)` searches within the axes specified by the input handle `axesh`.

`h = handlem( ___ ,axesh,'`*searchmethod*`')` controls the method used to match the `object` input. If omitted, `'exact'` is assumed. Search method `'strmatch'` searches for matches that start at the beginning of the tag. Search method `'findstr'` searches anywhere within the tag for the object string.

`h = handlem(`handles`)` returns those elements of an input vector of handles that are still valid.

A prefix of `'all'` may be applied to strings defining a Handle Graphics® object type (text, line, patch, light, surface, or image) to find all object handles that meet the type criteria (for example, `'allimage'`). Without the `'all'` prefix, only handles with an empty tag are returned.

## See Also
`findobj`

# hidem

Hide specified graphic objects on map axes

## Syntax

```
hidem
hidem(handle)
hidem(object)
```

## Description

hidem brings up a dialog box for selecting the objects to hide (set their Visible property to 'off').

hidem(handle) hides the objects specified by a vector of handles.

hidem(*object*) hides those objects specified by the *object* string, which can be any string recognized by the handlem function.

## See Also

clma | clmo | handlem | namem | showm | tagm

# hista

Bin counts for geographic points using equal-area bins

## Syntax

```
[latbin,lonbin,count] = hista(lat,lon)
[latbin,lonbin,count] = hista(lat,lon,binarea)
[latbin,lonbin,count] = hista(lat,lon,binarea,spheroid)
[latbin,lonbin,count]] = hista( ___ ,angleunits)
```

## Description

`[latbin,lonbin,count] = hista(lat,lon)` bins the geographic locations indicated by vectors `lat` and `lon`, using equal area binning on a sphere. The default bin area is 100 square kilometers. The `latbin` and `lonbin` outputs are column vectors indicating the centers of non-empty bins. `count` matches `latbin` and `lonbin` in size, with each element containing a positive integer equal to the number of occurrences in the corresponding bin.

Binning is performed on a mesh within a quadrangle whose latitude and longitude limits match the extrema of the input locations. The input and output latitudes and longitudes are in units of degrees.

`[latbin,lonbin,count] = hista(lat,lon,binarea)` uses the bin size specified by the input `binarea`, which must be in square kilometers

`[latbin,lonbin,count] = hista(lat,lon,binarea,spheroid)` bins the data on the reference spheroid defined by `spheroid`. `spheroid` is a referenceEllipsoid (oblateSpheroid) object, a referenceSphere object, or a vector of the form `[semimajor_axis eccentricity]`. The eccentricity/flattening of the spheroid is used in determining the latitude extent of the bins. The semimajor axis of the spheroid is used to determine the longitude extent of the bins, but if the length unit of the spheroid is unspecified, the mean radius of the earth in kilometers is used as the equatorial radius.

`[latbin,lonbin,count]] = hista( ___ ,angleunits)` uses the string `angleunits` to define the angle units of the inputs and outputs. `angleunits` can be `'degrees'` or `'radians'`.

## Examples

Create random data:

```
lats = rand(4)

lats =
    0.4451    0.8462    0.8381    0.8318
    0.9318    0.5252    0.0196    0.5028
    0.4660    0.2026    0.6813    0.7095
    0.4186    0.6721    0.3795    0.4289

lons = rand(4)

lons =
    0.3046    0.3028    0.3784    0.4966
    0.1897    0.5417    0.8600    0.8998
    0.1934    0.1509    0.8537    0.8216
    0.6822    0.6979    0.5936    0.6449
```

Bin the data in 50-by-50 km cells (2500 sq km):

```
[lat,lon,num] = hista(lats,lons,2500);

[lat lon num]

ans =
    0.2574    0.3757    4.0000
    0.7070    0.3757    5.0000
   -0.1923    0.8253    1.0000
    0.2573    0.8253    2.0000
    0.7070    0.8254    4.0000
```

## See Also

```
eqa2grn | grn2eqa | histr
```

# histr

Histogram for geographic points with equirectangular bins

## Syntax

```
[lat,lon,num,wnum] = histr(lats,lons)
[lat,lon,num,wnum] = histr(lats,lons,units)
[lat,lon,num,wnum] = histr(lats,lons,bindensty)
```

## Description

`[lat,lon,num,wnum] = histr(lats,lons)` returns the center coordinates of equal-rectangular bins and the number of observations, `num`, falling in each based on the geographically distributed input data. Additionally, an area-weighted observation value, `wnum`, is returned. `wnum` is the bin's `num` divided by its normalized area. The largest bin has the same `num` and `wnum`; a smaller bin has a larger `wnum` than `num`.

`[lat,lon,num,wnum] = histr(lats,lons,units)` specifies the standard angle unit string. The default value is `'degrees'`.

`[lat,lon,num,wnum] = histr(lats,lons,bindensty)` sets the number of bins per angular unit. For example, if *units* is `'degrees'`, a *bindensty* of 10 would be 10 bins per degree of latitude or longitude, resulting in 100 bins per *square* degree. The default is one cell per angular unit.

The `histr` function sorts geographic data into equirectangular bins for histogram purposes. Equirectangular in this context means that each bin has the same angular measurement on each side (e.g., 1º-by-1º). Consequently, the result is not an equal-area histogram. The `hista` function provides that capability. However, the results of `histr` can be weighted by their area bias to correct for this, in some sense.

## Examples

Create random data:

```
lats = rand(4)
```

```
lats =
    0.4451    0.8462    0.8381    0.8318
    0.9318    0.5252    0.0196    0.5028
    0.4660    0.2026    0.6813    0.7095
    0.4186    0.6721    0.3795    0.4289

longs = rand(4)

longs =
    0.3046    0.3028    0.3784    0.4966
    0.1897    0.5417    0.8600    0.8998
    0.1934    0.1509    0.8537    0.8216
    0.6822    0.6979    0.5936    0.6449
```

Bin the data in 0.5-by-0.5 degree cells (two bins per degree):

```
[lat,lon,num,wnum] = histr(lats,longs,2);
[lat,lon,num,wnum]

ans =
    0.2500    0.2500    3.0000    3.0000
    0.7500    0.2500    4.0000    4.0003
    0.2500    0.7500    4.0000    4.0000
    0.7500    0.7500    5.0000    5.0004
```

The bins centered at 0.75ºN are slightly smaller in area than the others. `wnum` reflects the relative count per normalized unit area.

## See Also

`filterm` | `hista`

# imbedm

Encode data points into regular data grid

## Syntax

```
Z = imbedm(lat, lon, value, Z, R)
Z = imbedm(lat, lon, value, Z, R, units)
[Z, indxPointOutsideGrid] = imbedm(...)
```

## Description

`Z = imbedm(lat, lon, value, Z, R)` resets certain entries of a regular data grid, `Z`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

`Z = imbedm(lat, lon, value, Z, R, units)` specifies the units of the vectors `lat` and `lon`, where *units* is any valid angle units string (`'degrees'` by default).

`[Z, indxPointOutsideGrid] = imbedm(...)` returns the indices of `lat` and `lon` corresponding to points outside the grid in the variable `indxPointOutsideGrid`.

## Examples

Create a simple grid map and embed new values in it:

```
Z = ones(3,6)

Z =
    1    1    1    1    1    1
    1    1    1    1    1    1
    1    1    1    1    1    1
refvec = [1/60 90 -180]

refvec =
    0.0167   90.0000 -180.0000

newgrid = imbedm([23 -23], [45 -45],[5 5],Z,refvec)

newgrid =
    1    1    1    1    1    1
    1    1    5    5    1    1
    1    1    1    1    1    1
```

## See Also

```
ltln2val | setpostn
```

# ind2rgb8

Convert indexed image to uint8 RGB image

## Syntax

```
RGB = ind2rgb8(X,cmap)
```

## Description

`RGB = ind2rgb8(X,cmap)` creates an RGB image of class `uint8`. X must be `uint8`, `uint16`, or `double`, and `cmap` must be a valid MATLAB colormap.

## Examples

```
% Convert the 'concord_ortho_e.tif' image to RGB.
[X, cmap] = imread('concord_ortho_e.tif');
RGB = ind2rgb8(X, cmap);
R = worldfileread('concord_ortho_e.tfw');
mapshow(RGB, R);
```

## See Also
ind2rgb

# ingeoquad

True for points inside or on lat-lon quadrangle

## Syntax

```
tf = ingeoquad(lat, lon, latlim, lonlim)
```

## Description

`tf = ingeoquad(lat, lon, latlim, lonlim)` returns an array `tf` that has the same size as `lat` and `lon`. `tf(k)` is true if and only if the point `lat(k)`, `lon(k)` falls within or on the edge of the geographic quadrangle defined by `latlim` and `lonlim`. `latlim` is a vector of the form `[southern-limit northern-limit]`, and `lonlim` is a vector of the form `[western-limit eastern-limit]`. All angles are in units of degrees.

## Examples

**1**   Load and display a digital elevation model (DEM) including the Korean Peninsula:

```
korea = load('korea');
R = refvecToGeoRasterReference(korea.refvec, size(korea.map));
figure('Color','white')
worldmap([20 50],[90 150])
geoshow(korea.map, R, 'DisplayType', 'texturemap');
demcmap(korea.map)
```

**2**   Outline the quadrangle containing the DEM:

```
[outlineLat, outlineLon] = outlinegeoquad(R.Latlim, ...
   R.Lonlim, 90, 5);
geoshow(outlineLat,outlineLon,'DisplayType','line', ...
   'Color','black')
```

**3**   Generate a track that crosses the DEM:

```
[lat, lon] = track2(23, 110, 48, 149, [1 0], 'degrees', 20);
geoshow(lat, lon, 'DisplayType', 'line')
```

**4** Identify and mark points on the track that fall within the quadrangle outlining the DEM:

```
tf = ingeoquad(lat, lon, R.Latlim, R.Lonlim);
geoshow(lat(tf), lon(tf), 'DisplayType', 'point')
```
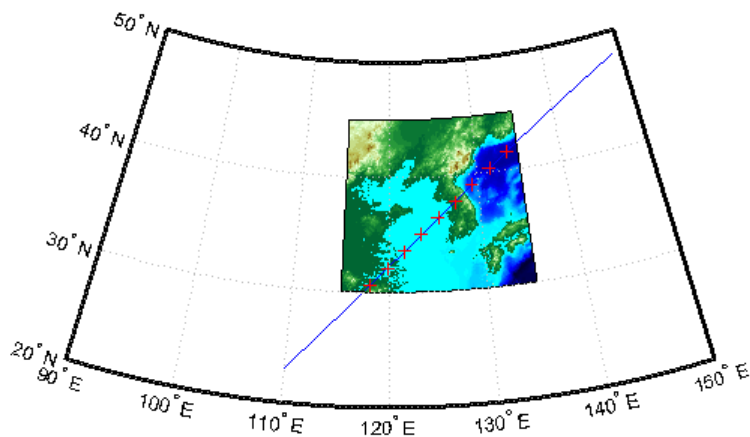


## See Also
inpolygon | intersectgeoquad

# intersectgeoquad

Intersection of two latitude-longitude quadrangles

## Syntax

```
[latlim, lonlim] = intersectgeoquad(latlim1, lonlim1, latlim2,
lonlim2)
```

## Description

`[latlim, lonlim] = intersectgeoquad(latlim1, lonlim1, latlim2, lonlim2)` computes the intersection of the quadrangle defined by the latitude and longitude limits `latlim1` and `lonlim1`, with the quadrangle defined by the latitude and longitude limits `latlim2` and `lonlim2`. `latlim1` and `latlim2` are two-element vectors of the form `[southern-limit northern-limit]`. Likewise, `lonlim1` and `lonlim2` are two-element vectors of the form `[western-limit eastern-limit]`. All input and output angles are in units of degrees. The intersection results are given in the output arrays `latlim` and `lonlim`. Given an arbitrary pair of input quadrangles, there are three possible results:

1   *The quadrangles fail to intersect.* In this case, both `latlim` and `lonlim` are empty arrays.

2   *The intersection consists of a single quadrangle.* In this case, `latlim` (like `latlim1` and `latlim2`) is a two-element vector that has the form `[southern-limit northern-limit]`, where `southern-limit` and `northern-limit` represent scalar values. `lonlim` (like `lonlim1` and `lonlim2`), is a two-element vector that has the form `[western-limit eastern-limit]`, with a pair of scalar limits.

3   *The intersection consists of a pair of quadrangles.* This can happen when longitudes wrap around such that the eastern end of one quadrangle overlaps the western end of the other and vice versa. For example, if `lonlim1 = [-90 90]` and `lonlim2 = [45 -45]`, there are two intervals of overlap: `[-90 -45]` and `[45 90]`. These limits are returned in `lonlim` in separate rows, forming a 2-by-2 array. In our example (assuming that the latitude limits overlap), `lonlim` would equal `[-90 -45; 45 90]`. It still has the form `[western-limit eastern-limit]`, but `western-limit` and `eastern-limit` are 2-by-1 rather than scalar. The two

output quadrangles have the same latitude limits, but these are replicated so that `latlim` is also 2-by-2.

To continue the example, if `latlim1 = [0 30]` and `latlim2 = [20 50]`, `latlim` equals `[20 30; 20 30]`. The form is still `[southern-limit northern-limit]`, but in this case `southern-limit` and `northern-limit` are 2-by-1.

# Examples

## Example 1

Nonintersecting quadrangles:

```
[latlim, lonlim] = intersectgeoquad( ...
                   [-40 -60], [-180 180], [40 60], [-180 180])
latlim =
     []

lonlim =
     []
```

## Example 2

Intersection is a single quadrangle:

```
[latlim, lonlim] = intersectgeoquad( ...
                   [-40 60], [-120 45], [-60 40], [160 -75])

latlim =
   -40     40

lonlim =
  -120    -75
```

## Example 3

Intersection is a pair of quadrangles:

```
[latlim, lonlim] = intersectgeoquad( ...
                   [-30 90],[-10 -170],[-90 30],[170 10])

latlim =
```

```
     -30     30
     -30     30

lonlim =
     -10     10
     170   -170
```

## Example 4

Inputs and output fully encircle the planet:

```
[latlim, lonlim] = intersectgeoquad( ...
                [-30 90],[-180 180],[-90 30],[0 360])

latlim =
     -30     30

lonlim =
    -180    180
```

## Example 5

Find and map the intersection of the bounding boxes of adjoining U.S. states:

```
usamap({'Minnesota','Wisconsin'})
S = shaperead('usastatehi','UseGeoCoords',true,'Selector',...
    {@(name) any(strcmp(name,{'Minnesota','Wisconsin'})), 'Name'});
geoshow(S, 'FaceColor', 'y')
textm([S.LabelLat], [S.LabelLon], {S.Name},...
    'HorizontalAlignment', 'center')
latlimMN = S(1).BoundingBox(:,2)'

latlimMN =
    43.4995    49.3844

lonlimMN = S(1).BoundingBox(:,1)'

lonlimMN =
   -97.2385   -89.5612

latlimWI = S(2).BoundingBox(:,2)'

latlimWI =
    42.4918    47.0773
```

```
lonlimWI = S(2).BoundingBox(:,1)'

lonlimWI =
  -92.8892  -86.8059

[latlim lonlim] = ...
    intersectgeoquad(latlimMN, lonlimMN, latlimWI, lonlimWI)

latlim =
   43.4995   47.0773

lonlim =
  -92.8892  -89.5612

geoshow(latlim([1 2 2 1 1]), lonlim([1 1 2 2 1]), ...
    'DisplayType','polygon','FaceColor','m')
```
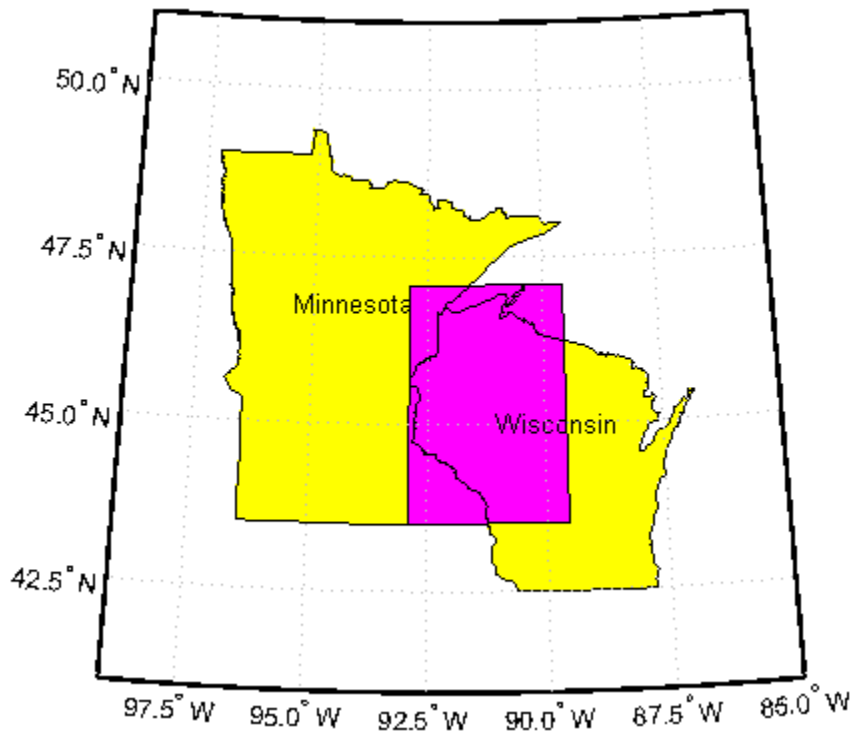
## More About

### Tips

`latlim1` and `latlim2` should normally be given in order of increasing numerical value. No error will result if, for example, `latlim1(2) < latlim1(1)`, but the outputs will both be empty arrays.

No such restriction applies to `lonlim1` and `lonlim2`. The first element is always interpreted as the western limit, even if it exceeds the second element (the eastern limit). Furthermore, `intersectgeoquad` correctly handles whatever longitude-wrapping convention may have been applied to `lonlim1` and `lonlim2`.

In terms of output, `intersectgeoquad` wraps `lonlim` such that all elements fall in the closed interval `[-180 180]`. This means that if (one of) the output quadrangle(s) crosses the 180° meridian, its western limit exceeds its eastern limit. The result would be such that

`lonlim(2) < lonlim(1)`
if the intersection comprises a single quadrangle or

`lonlim(k,2) < lonlim(k,1)`
where `k` equals 1 or 2 if the intersection comprises a pair of quadrangles.

If `abs(diff(lonlim1))` or `abs(diff(lonlim2))` equals `360`, its quadrangle is interpreted as a latitudinal zone that fully encircles the planet, bounded only by one parallel on the south and another parallel on the north. If two such quadrangles intersect, `lonlim` is set to `[-180 180]`.

If you want to display geographic quadrangles generated by this function or any other which are more than one or two degrees in extent, they may not follow curved meridians and parallels very well. The degree of departure depends on the extent of the quadrangle, the map projection, and the map scale. In such cases, you can interpolate intermediate vertices along quadrangle edges with the `outlinegeoquad` function.

### See Also
`ingeoquad` | `outlinegeoquad`

# inputm

Latitudes and longitudes of mouse-click locations

## Syntax

```
[lat, lon] = inputm
[lat, lon] = inputm(n)
[lat, lon] = inputm(n,h)
[lat, lon, button] = inputm(n)
MAT = imputm(...)
```

## Description

`[lat, lon] = inputm` returns the latitudes and longitudes in geographic coordinates of points selected by mouse clicks on a displayed grid. The point selection continues until the return key is pressed.

`[lat, lon] = inputm(n)` returns `n` points specified by mouse clicks.

`[lat, lon] = inputm(n,h)` prompts for points from the map axes specified by the handle `h`. If omitted, the current axes (gca) is assumed.

`[lat, lon, button] = inputm(n)` returns a third result, `button`, that contains a vector of integers specifying which mouse button was used (1,2,3 from left) or ASCII numbers if a key on the keyboard was used.

`MAT = imputm(...)` returns a single matrix, where `MAT = [lat lon]`.

## More About

### Tips

`inputm` works much like the standard MATLAB `ginput`, except that the returned values are latitudes and longitudes extracted from the projection, rather than axes *x-y* coordinates. If you click outside of the projection bounds (beyond the map frame in

the corners of a Robinson projection, for example), no coordinates are returned for that location.

`inputm` cannot be used with a 3-D display, including those created using `globe`.

## See Also
`gcpmap` | `ginput`

# interpm

Densify latitude-longitude sampling in lines or polygons

## Syntax

```
[latout,lonout] = interpm(lat,lon,maxdiff)
[latout,lonout] = interpm(lat,lon,maxdiff,method)
[latout,lonout] = interpm(lat,lon,maxdiff,method,units)
```

## Description

`[latout,lonout] = interpm(lat,lon,maxdiff)` fills in any gaps in latitude (`lat`) or longitude (`lon`) data vectors that are greater than a defined tolerance `maxdiff` apart in either dimension. The angle units of the three inputs need not be specified, but they must be identical. `latout` and `lonout` are the new latitude and longitude data vectors, in which any gaps larger than `maxdiff` in the original vectors have been filled with additional points. The default method of interpolation used by `interpm` is linear.

`[latout,lonout] = interpm(lat,lon,maxdiff,method)` interpolates between vector data coordinate points using a specified interpolation method. Valid interpolation method strings are `'gc'` for great circle, `'rh'` for rhumb line, and `'lin'` for linear interpolation.

`[latout,lonout] = interpm(lat,lon,maxdiff,method,units)` specifies the units used, where *units* is any valid angle units string. The default is `'degrees'`.

## Examples

```
lat = [1 2 4 5]; lon = [7 8 9 11];
[latout,lonout] = interpm(lat,lon,1);
[latout lonout]

ans =
    1.0000    7.0000
    2.0000    8.0000
    3.0000    8.5000
```

```
4.0000     9.0000
4.5000    10.0000
5.0000    11.0000
```

## See Also
`intrplat | intrplon`

# intrplat

Interpolate latitude at given longitude

## Syntax

```
newlat = intrplat(long,lat,newlong)
newlat = intrplat(long,lat,newlong,method)
newlat = intrplat(long,lat,newlong,method,units)
```

## Description

`newlat = intrplat(long,lat,newlong)` returns an interpolated latitude, `newlat`, corresponding to a longitude `newlong`. `long` must be a monotonic vector of longitude values. The actual entries must be monotonic; that is, the longitude vector `[350 357 3 10]` is not allowed even though the geographic *direction* is unchanged (use `[350 357 363 370]` instead). `lat` is a vector of the latitude values paired with each entry in `long`.

`newlat = intrplat(long,lat,newlong,method)` specifies the method of interpolation employed, listed in the table below.

| Method | Description |
|---|---|
| `'linear'` | Linear, or Cartesian, interpolation (default) |
| `'pchip'` | Piecewise cubic Hermite interpolation |
| `'rh'` | Returns interpolated points that lie on rhumb lines between input data |
| `'gc'` | Returns interpolated points that lie on great circles between input data |

`newlat = intrplat(long,lat,newlong,method,units)` specifies the units used, where *units* is any valid angle units string. The default is `'degrees'`.

The function `intrplat` is a geographic data analogy of the standard MATLAB function `interp1`.

## Examples

Compare the results of the various methods:

```
lats = [25 45]; longs = [30 60];
newlat = intrplat(longs,lats,45,'linear')

newlat =
    35

newlat = intrplat(longs,lats,45,'rh')

newlat =
    35.6213

newlat = intrplat(longs,lats,45,'gc')

newlat =
    37.1991
```

## More About

### Tips

There are separate functions for interpolating latitudes and longitudes, for although the cases are identical when using those methods supported by `interp1`, when latitudes and longitudes are treated like the spherical angles they are (using `'rh'` or `'gc'`), the results are different. Compare the example above to the example under `intrplon`, which reverses the values of latitude and longitude.

## See Also

`interpm` | `intrplon`

# intrplon

Interpolate longitude at given latitude

## Syntax

```
newlon = intrplon(lat,lon,newlat)
newlon = intrplon(lat,lon,newlat,method)
newlon = intrplon(lat,lon,newlat,method,units)
```

## Description

`newlon = intrplon(lat,lon,newlat)` returns an interpolated longitude, `newlon`, corresponding to a latitude `newlat`. `lat` must be a monotonic vector of longitude values. `lon` is a vector of the longitude values paired with each entry in `lat`.

`newlon = intrplon(lat,lon,newlat,method)` specifies the method of interpolation employed, listed in the table below.

| Method | Description |
|---|---|
| `'linear'` | Linear, or Cartesian, interpolation (default) |
| `'pchip'` | Piecewise cubic Hermite interpolation |
| `'rh'` | Returns interpolated points that lie on rhumb lines between input data |
| `'gc'` | Returns interpolated points that lie on great circles between input data |

`newlon = intrplon(lat,lon,newlat,method,units)` specifies the units used, where `units` is any valid angle units string. The default is `'degrees'`.

The function `intrplon` is a geographic data analogy of the MATLAB function `interp1`.

## Examples

Compare the results of the various methods:

```
long = [25 45]; lat = [30 60];
newlon = intrplon(lat,long,45,'linear')

newlon =
    35

newlon = intrplon(lat,long,45,'rh')

newlon =
   33.6515

newlon = intrplon(lat,long,45,'gc')

newlon =
   32.0526
```

## More About

### Tips

There are separate functions for interpolating latitudes and longitudes, for although the cases are identical when using those methods supported by `interp1`, when latitudes and longitudes are treated like the spherical angles they are (using `'rh'` or `'gc'`), the results are different. Compare the previous example to the example under `intrplat`, which reverses the values of latitude and longitude.

## See Also
`interpm` | `intrplat`

# map.geodesy.isDegree

True if string matchesdegree and false if `radian`

## Syntax

```
tf = map.geodesy.isDegree(angleUnit)
```

## Description

`tf = map.geodesy.isDegree(angleUnit)` returns `true` if `angleUnit` is a partial match for `'degree'` (or `'degrees'`) and `false` if `angleUnit` is a partial match for `'radian'` (or `'radians'`). If `angleUnit` matches neither `'degrees'` or `'radians'`, `map.geodesy.isDegree` returns an error.

## Examples

**Test Inputs to a Function for Validity Before Processing**

Create a function to calculate a cosine. In the function, use `map.geodesy.isDegree` to check the validity of the inputs.

```
function y = cosine(x, angleUnit)
% X can be in either degrees or radians

if map.geodesy.isDegree(angleUnit)
    y = cosd(x);
else
    y = cos(x);
end
```

## Input Arguments

**angleUnit — Angle unit value**
`'degree'` | `'radian'`

Angle unit value, specified as the character strings `'degree'` or `'radian'`.

Data Types: `char`

## Output Arguments

**`tf` — True/false flag indicating if a match was found**
logical scalar

True/false flag indicating if a match was found, returned as a logical scalar.

# ismap

True for axes with map projection

## Syntax

```
mflag = ismap
mflag = ismap(hndl)
[mflag,msg] = ismap(hndl)
```

## Description

`mflag = ismap` returns a 1 if the current axes is a map axes, and 0 otherwise.

`mflag = ismap(hndl)` specifies the handle of the axes to be tested.

`[mflag,msg] = ismap(hndl)` returns a string message if the axes is not a map axes, specifying why not.

The `ismap` function tests an axes object to determine whether it is a map axes.

## See Also
gcm | ismapped

# ismapped

True, if object is projected on map axes

## Syntax

```
mflag = ismapped
mflag = ismapped(hndl)
[mflag,msg] = ismapped(hndl)
```

## Description

`mflag = ismapped` returns a 1 if the current object is projected on a map axes, and 0 otherwise.

`mflag = ismapped(hndl)` specifies the handle of the object to be tested.

`[mflag,msg] = ismapped(hndl)` returns a string message if the axes is not projected on a map axes, specifying why not.

The `ismapped` function tests an object to determine whether it is projected on map axes.

## See Also
gcm | ismap

# ispolycw

True if polygon vertices are in clockwise order

## Syntax

```
tf = ispolycw(x, y)
```

## Description

`tf = ispolycw(x, y)` returns true if the polygonal contour vertices represented by `x` and `y` are ordered in the clockwise direction. `x` and `y` are numeric vectors with the same number of elements.

Alternatively, `x` and `y` can contain multiple contours, either in `NaN`-separated vector form or in cell array form. In that case, `ispolycw` returns a logical array containing one true or false value per contour.

`ispolycw` always returns true for polygonal contours containing two or fewer vertices.

Vertex ordering is not well defined for self-intersecting polygonal contours. For such contours, `ispolycw` returns a result based on the order or vertices immediately before and after the left-most of the lowest vertices. In other words, of the vertices with the lowest `y` value, find the vertex with the lowest `x` value. For a few special cases of self-intersecting contours, the vertex ordering cannot be determined using only the left-most of the lowest vertices; for these cases, `ispolycw` uses a signed area test to determine the ordering.

## Class Support

`x` and `y` may be any numeric class.

## Examples

Orientation of a square:

```
x = [0 1 1 0 0];
y = [0 0 1 1 0];
ispolycw(x, y)                  % Returns 0
ispolycw(fliplr(x), fliplr(y))  % Returns 1
```

## See Also
poly2cw | poly2ccw | polybool

# isShapeMultipart

True if polygon or line has multiple parts

## Syntax

```
tf = isShapeMultipart(xdata, ydata)
```

## Description

`tf = isShapeMultipart(xdata, ydata)` returns `1` (`true`) if the polygon or line shape specified by `xdata` and `ydata` consists of multiple `NaN`-separated parts (i.e. has inner or multiple polygon rings or multiple line segments). The coordinate arrays `xdata` and `ydata` must match in size and have identical `NaN` locations.

## Examples

### Check If Datasets Are Multipart

Create a simple data set and check if it's multipart. If a data set contains NaN separators, `isShapeMultipart` returns `1`, otherwise `0`.

```
sample_xdata = [0 0 1];
sample_ydata = [0 1 0];
isShapeMultipart(sample_xdata,sample_ydata)
```

```
ans =

    0
```

Create simple multipart data sets.

```
multi_xdata = [0 0 1 NaN 2 2 3 3];
multi_ydata = [0 1 0 NaN 2 3 3 2];
isShapeMultipart(multi_xdata,multi_ydata)
```

```
ans =

     1
```

Check a real data set.

```
load coastlines
isShapeMultipart(coastlat,coastlon)
```

```
ans =

     1
```

Check the data in a shapefile.

```
S = shaperead('concord_hydro_area');
isShapeMultipart(S(1).X, S(1).Y)
```

```
ans =

     0
```

Check another dataset in the shapefile.

```
isShapeMultipart(S(14).X, S(14).Y)
```

```
ans =

     1
```

## See Also
```
polysplit
```

# km2deg

Convert distance from kilometers to degrees

## Syntax

```
deg = km2deg(km)
deg = km2deg(km,radius)
deg = km2deg(km,sphere)
```

## Description

`deg = km2deg(km)` converts distances from kilometers to degrees as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`deg = km2deg(km,radius)` converts distances from kilometers to degrees as measured along a great circle on a sphere having the specified radius. `radius` must be in units of kilometers.

`deg = km2deg(km,sphere)` converts distances from kilometers to degrees, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

## Examples

Two cities are 340 km apart. How many degrees of arc is that? How many degrees would it be if the cities were on Mars?

```
deg = km2deg(340)

deg =
    3.0577

deg = km2deg(340,'mars')
```

```
deg =
    5.7465
```

## See Also

deg2rad | deg2km | km2rad | km2nm | km2sm | deg2nm | nm2deg | nm2km | nm2sm |
deg2sm | sm2deg | sm2km | sm2nm | rad2deg

# km2nm

Convert kilometers to nautical miles

## Syntax

```
nm = km2nm(km)
```

## Description

`nm = km2nm(km)` converts distances from kilometers to nautical miles.

### See Also

deg2km | km2deg | km2rad | rad2km | deg2nm | nm2deg | nm2rad | rad2nm | deg2sm | sm2deg | deg2sm | sm2rad | rad2sm

# km2rad

Convert distance from kilometers to radians

## Syntax

```
rad = km2rad(km)
rad = km2rad(km,radius)
rad = km2rad(km,sphere)
```

## Description

`rad = km2rad(km)` converts distances from kilometers to radians as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`rad = km2rad(km,radius)` converts distances from kilometers to radians as measured along a great circle on a sphere having the specified radius. `radius` must be in units of kilometers.

`rad = km2rad(km,sphere)` converts distances from kilometers to radians , as measured along a great circle on a sphere approximating an object in the Solar System. `sphere` may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

## Examples

How many radians does 1,000 km span on the Earth and on the Moon?

```
rad = km2rad(1000)

rad =
    0.1570

rad = km2rad(1000,'moon')

rad =
```

```
0.5754
```

## See Also

deg2rad | rad2km | km2deg | km2nm | km2sm | rad2nm | nm2deg | nm2km | nm2sm | rad2sm | sm2deg | sm2km | sm2nm | rad2deg

# km2sm

Convert kilometers to statute miles

## Syntax

```
sm = km2sm(km)
```

## Description

`sm = km2sm(km)` converts distances from kilometers to statute miles.

## Examples

How many statute miles is a 10k run?

```
sm = km2sm(10)

sm =
    6.2137
```

## See Also
deg2km | km2deg | km2rad | rad2km | deg2nm | nm2deg | nm2rad | rad2nm |
deg2sm | sm2deg | deg2sm | sm2rad | rad2sm

# kmlwrite

Write geographic data to KML file

## Syntax

```
kmlwrite(filename,S)

kmlwrite(filename,latitude,longitude)
kmlwrite(filename,latitude,longitude,altitude)

kmlwrite(filename,address)

kmlwrite( ___ ,Name,Value)
```

## Description

`kmlwrite(filename,S)` writes the geographic point or line data stored in `S` to the file specified by `filename` in Keyhole Markup Language (KML) format. `S` is a geopoint vector, a geoshape vector, or geostruct. `kmlwrite` creates a KML Placemark in the file and populates the tags in the placemark with the data in `S`.

`kmlwrite(filename,latitude,longitude)` writes the geographic point data, `latitude` and `longitude`, to the file specified by `filename` in KML format. `kmlwrite` sets the third coordinate value (altitude) to `0`.

`kmlwrite(filename,latitude,longitude,altitude)` uses the values of `latitude`,`longitude`, and `altitude` to set the three values in the Placemark coordinates tag. When you specify an altitude value, `kmlwrite` sets the `AltitudeMode` attribute to `relativeToSeaLevel` .

`kmlwrite(filename,address)` writes `address` to the file specified by `filename` in KML format. `address` is a text string containing freeform address data, that can include street, city, state, country, and/or postal code. To specify multiple addresses, use a cell array of strings. `kmlwrite` creates a KML Placemark in the file, setting the address tag to the value specified by `address`. An address is an alternative way to specify a point, instead of using latitude and longitude.

kmlwrite( ___ ,Name,Value) specifies name-value pairs that set additional KML feature properties. Parameter names can be abbreviated and are case-insensitive.

# Examples

### Write Point Data to KML File Using geopoint Vector

Read point data into a geopoint vector.

```
placenames = gpxread('boston_placenames');
```

Define the name of the KML file you want to create.

```
filename = 'Boston_Placenames.kml';
```

Write the point data to the file, using optional parameters to specify names for the points and define the colors used for the icons.

```
kmlwrite(filename, placenames, 'Name', placenames.Name, ...
         'Color', jet(length(placenames)));
```

### Write Line Data to KML File Using geoshape Vector

Read line features into a geoshape vector.

```
tracks = gpxread('sample_tracks', 'Index', 1:2);
```

Define the name of the KML file you want to create.

```
filename = 'tracks.kml';
```

Write the line data to the file, using several optional parameters to specify the color and width of the lines, and their names and descriptions.

```
color = {'red', 'green'};
description = tracks.Metadata.Name;
name = {'track1', 'track2'};

kmlwrite(filename, tracks, 'Color', color, 'Width', 2, ...
         'Description', description, 'Name', name);
```

### Write Geographic Data to KML File Using geostruct

Read geographic data (locations of major European cities) from a shape file, including the names of the cities, and remove the default description table.

```
latlim = [ 30; 75];
lonlim = [-25; 45];
cities = shaperead('worldcities.shp','UseGeoCoords', true, ...
                    'BoundingBox', [lonlim, latlim]);
```

Define the name of the KML file you want to create.

```
filename = 'European_Cities.kml';
```

Write the geographic data to a KML file, using several optional parameters to include the names of the cities in the placemarks.

```
kmlwrite(filename, cities, 'Name', {cities.Name}, 'Description',{});
```

### Write Unstructured Address to KML File

Create a cell array of unstructured addresses (the names of several Australian cities).

```
address = {'Perth, Australia', ...
           'Melbourne, Australia', ...
           'Sydney, Australia'};
```

Define the name of the KML file you want to create.

```
filename = 'Australian_Cities.kml';
```

Write the unstructured address data to a KML file, using the optional Name parameter to include the names of the cities in the placemarks.

```
kmlwrite(filename, address, 'Name', address);
```

### Include HTML Format Tags in Description

Define a point by latitude and longitude.

```
lat =  42.299827;
lon = -71.350273;
```

Specify the description text used with the placemark, including HTML tags for formatting.

```
description = sprintf('%s<br>%s</br><br;>%s</br>', ...
      '3 Apple Hill Drive', 'Natick, MA. 01760', ...
      'http://www.mathworks.com');
name = 'The MathWorks, Inc.';
iconDir = fullfile(matlabroot,'toolbox','matlab','icons');
iconFilename = fullfile(iconDir, 'matlabicon.gif');
```

Define the name of the KML file you want to create.

```
filename = 'MathWorks.kml';
```

Write the data to a KML file, using the `Description` parameter to include the names of the cities in the placemarks.

```
kmlwrite(filename, lat, lon, ...
        'Description', description, 'Name', name, 'Icon', iconFilename);
```

**Write Point Data to a KML file Using Camera to Specify the View**

Specify the latitude and longitude values that define a point. In this example, the location is Mount Ranier.

```
lat_rainier = 46.8533
lon_rainier = -121.7599
```

Create a geopoint vector to specify the position of the virtual camera (eye) using the `Camera` parameter.

```
myview = geopoint(46.7, -121.7,'Altitude',2500,'Tilt',85,'Heading',345)
```

Define the name of the KML file you want to create.

```
filename = 'Mt_Ranier.kml';
```

Write the point data to the file, specifying a name and a custom color for the icon

```
kmlwrite(filename,lat_rainier,lon_rainier,'Name','Mt Rainier',...
        'Color','red','IconScale',2,'Camera',myview)
```

## Input Arguments

### `filename` — Name of KML file to create
character string

Name of KML file to create, specified as a character string. `kmlwrite` writes the file in the current folder, unless you specify a full or relative path name. If the file name includes an extension, it must be `.kml`.

Data Types: `char`

### S — Geographic features to write to file
geopoint vector, geoshape vector, or geostruct

Geographic features to write to file, specified as a geopoint vector, a geoshape vector, or geostruct.

- If a geoshape vector, the `Geometry` field must be set to `'point'` or `'line'`.
- If a geostruct (with `Lat` and `Lon` fields), the `Geometry` field must be set to either `'Point'`, `'MultiPoint'`, or `'Line'`.
- If you include additional attribute fields in `S`, `kmlwrite` writes the data to the description tag of the placemark for each feature, formatted as an HTML table. The attribute fields appear in the table in the same order as they occur in `S`.
- If `S` contains a field named either `Elevation`, `Altitude`, or `Height`, `kmlwrite` writes the field values to the file as KML altitudes and sets the altitude interpretation to `'relativeToSeaLevel'`. If `S` contains fields with more than one of these names, `kmlwrite` issues a warning and ignores the altitude fields.

### `latitude` — Latitude of points
numeric vector in the range [`-90 90`]

Latitude of points, specified as a numeric vector in the range [`-90 90`].

Data Types: `single` | `double`

### `longitude` — Longitude of points
numeric vector automatically wrapped to the range [`-180, 180`]

Longitude of points, specified as a numeric vector in the range [`-180, 180`].

Data Types: `single` | `double`

### `altitude` — Altitude of points
0 (default) | numeric vector or scalar.

Altitude of points, specified as a numeric vector or scalar. Unit of measure is meters.

- If a scalar, `kmlwrite` applies the value to each point.
- If a vector, you must specify an altitude value for each point. That is, the vector must have the same length as `latitude` and `longitude`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

### `address` — Location of points
character string or cell array

Location of points, specified as a character string or cell array of strings containing freeform address data, such as street, city, state, and/or postal code. If `address` is a cell array, each cell represents a unique location.

Example: `'MathWorks, Inc, 3 Apple Hill Drive, Natick, MA 01760-2098'`

Data Types: `char` | `cell`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Name','Point Reyes'

### `'Name'` — Label of object displayed in viewer
`'Address N'`, `'Point N'`, or `'Line N'`, where N is the index of the address, point, or line (default) | character string or cell array of strings

Label for an object displayed in viewer, specified as a character string or cell array of strings.

- If a string, `kmlwrite` applies the name to all objects
- If a cell array, you must specify one string for each point, line, or address. That is, the cell array must be the same length as `latitude` and `longitude`, `S`, or `address`.
- If you are writing multipoint data, `kmlwrite` places the points in a folder labeled with the `name` value you specified, or, if no name is specified, `'Multipoint N'`. In the folder, each point is labeled `'Point N'`, where N is the index of the point.
- If you are writing a line and the lines contains `NaN` values, `kmlwrite` places the line segments in a folder labeled with the corresponding `Name` value or, if no name is specified, `'Line N'`. In the folder, each line segments is labeled `'Segment N'`, where N is the segment number.

Data Types: `char` | `cell`

### `'Description'` — Content to be displayed in the placemark's description balloon
character string or cell array of strings | attribute specification

Content to be displayed in the placemark's description balloon, specified as a text string, cell array of strings, or attribute specification. `kmlwrite` uses this data to sets the values

of the feature's description tag(s). The description appears in the description balloon when the user clicks on either the feature name in the Google Earth Places panel or clicks the placemark icon in the viewer window.

- If a string, `kmlwrite` applies the description to all objects.

- If a cell array, there must be one label for each point, that is, it must be the same length as `latitude` and `longitude`, `S`, or `address`.

You can include basic HTML mark up, however, Google Earth applies some HTML formatting automatically. For example, Google Earth replaces newlines with line break tags and encloses valid URLs in anchor tags to make them hyperlinks. To see examples of HTML tags that are recognized by Google Earth, view http://earth.google.com.

If you provide an attribute specification, the attribute fields of `S` display as a table in the description tag of the placemark for each element of `S`, in the order in which the fields appear in the specification. To construct an attribute spec, call `makeattribspec` and then modify the output to remove attributes or change the `Format` field for one or more attributes. The `latitude` and `longitude` coordinates of `S` are not considered to be attributes. If included in an attribute spec, they are ignored. If you specify `latitude` and `longitude`, `kmlwrite` ignores the attribute specification.

Example: `'Description','My Description'`

Data Types: `char` | `cell`

### `'Icon'` — File name of a custom icon
defined by viewer (default) | character string or cell array of strings

File name of a custom icon, specified as character string or cell array of strings.

- If a string, `kmlwrite` applies the value to all objects.

- If a cell array, you must specify an icon for each point or line. That is, it must be the same length as `latitude` and `longitude`, `S`, or `address`.

- If the string is an Internet URL, the URL must include the protocol type.

- If the icon filename is not in the current folder, or in a folder on the MATLAB path, specify a full or relative pathname.

Data Types: `char` | `cell`

### `'IconScale'` — Scaling factor for icon
positive numeric scalar or vector

**1-695**

Scaling factor for the icon, specified as a positive numeric scalar or vector.

- If a scalar, `kmlwrite` applies the value to all objects

- If a vector, you must specify a value for each icon. That is, the vector must be the same length as `latitude` and `longitude`, `S`, or `address`.

Example: `'Iconscale', 2`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

### `'Color'` — Color of lines or icons
defined by viewer (default) | ColorSpec

Color of lines or icons, specified as a MATLAB Color Specification (`ColorSpec`).

Example: `'Color','red'`

### `'Width'` — Width of the line, in pixels
1 (default) | positive numeric scalar or vector

Width of the line in pixels, specified as a positive numeric scalar or vector.

- If a scalar, `kmlwrite` applies the value to all lines.

- If a vector, you must specify the width of each line. That is, the vector must be the same length as `S`.

Example: `'Width',2`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

### `'AltitudeMode'` — Interpretation of altitude values
`'clampToGround'` (default) | `'relativeToGround'` | `'relativeToSeaLevel'`

Interpretation of altitude values, specified as one of the following text strings.

| Value | Description |
|---|---|
| `'clampToGround'` | Ignore altitude values and set the feature on the ground. This is the default value when you do not specify `altitude` values. |
| `'relativeToGround'` | Set altitude values relative to the actual ground elevation of a particular feature |

| Value | Description |
|---|---|
| `'relativeToSeaLevel'` | Set altitude values relative to sea level, regardless of the actual elevation values of the terrain beneath the feature. (Named `'absolute'` in the KML specification.) This is the default when you specify `altitude` values. |

Example: `'AltitudeMode','relativeToGround'`

Data Types: `char`

### `'LookAt'` — Position of virtual camera (eye) relative to object being viewed
geopoint vector

Position of the virtual camera (eye) relative to the object being viewed, specified as a geopoint vector. The fields of the geopoint vector, listed below, define the view. `LookAt` is limited to looking down at a feature. To tilt the virtual camera to look above the horizon into the sky, use the `Camera` parameter.

| Property Name | Description | Data Type |
|---|---|---|
| `Latitude` | Latitude of the object the camera is looking at, in degrees | Scalar double, from -90 to 90 |
| `Longitude` | Longitude of the object the camera is looking at, in degrees | Scalar double, from −180 to 180 |
| `Altitude` | Altitude of the object the camera is looking at from the Earth's surface, in meters | Scalar numeric |
| `Heading` | Camera direction (azimuth), in degrees (optional) | Scalar numeric `[0 360]`, default `0` (true North) |
| `Tilt` | Angle between the direction of the `LookAt` position and the normal to the surface of the Earth, in degrees (optional) | Scalar numeric `[0 90]`, default `0` (directly above) |
| `Range` | Distance in meters from the object specified by `latitude`, `longitude`, and `altitude` to the location of the camera. | Scalar numeric, default `0` |

| Property Name | Description | Data Type |
|---|---|---|
| AltitudeMode | Interpretation of the camera altitude value (optional) | 'relativeToSeaLevel', 'clampToGround', (default) 'relativeToGround' |

Example: 'LookAt',
geopoint(lat,lon,'Range',14794.88,'Heading',71.13,'Tilt',66.77)

### 'Camera' — Position of virtual camera relative to surface of Earth
geopoint vector

Position of virtual camera (eye) relative to Earth's surface, specified as a geopoint vector. The fields of the geopoint vector, listed below, define the view. Camera provides full six degrees of freedom control over the view, so you can position the camera in space and then rotate it around the *x*-, *y*-, and *z*-axes. You can tilt the camera view so that you're looking above the horizon into the sky.

| Property Name | Description | Data Type |
|---|---|---|
| Latitude | Latitude of the virtual camera (eye), in degrees | Scalar double, in the range [-90 90] |
| Longitude | Longitude of the virtual camera, in degrees | Scalar double, in the range [-180 180]. |
| Altitude | Distance of the virtual camera from the Earth's surface, in meters | Scalar numeric |
| Heading | Direction (azimuth) in degrees (optional) | Scalar numeric [0 360], default 0 (true North) |
| Tilt | Camera rotation around the X-axis, in degrees (optional) | Scalar numeric [0 180], default 0 (directly above) |
| Roll | Camera rotation in degrees around the Z-axis (optional) | Scalar numeric, in the range [-180 180], default 0 |
| AltitudeMode | Specifies how camera altitude is interpreted (optional) | 'relativeToSeaLevel', 'clampToGround', 'relativeToGround' (default) |

Example:
'Camera',geopoint(lat,lon,'Altitude',2500,'Tilt',85,'Heading',345)

## More About

### Tips

- You can view KML files with the Google Earth™ browser, which must be installed on your computer.

  For Windows, use the `winopen` function:

  ```
  winopen(filename)
  ```

  For Linux, if the filename is a partial path, use the following commands:

  ```
  cmd = 'googleearth ';
  fullfilename = fullfile(pwd, filename);
  system([cmd fullfilename])
  ```

  For Mac, if the filename is a partial path, use the following commands:

  ```
  cmd = 'open -a Google\ Earth '
  fullfilename = fullfile(pwd, filename);
  system([cmd fullfilename])
  ```

- You can also view KML files with a Google Maps™ browser. The file must be located on a web server that is accessible from the Internet. A private intranet server will not suffice, because Google's server must be able to access the URL that you provide. A template for using Google Maps is listed below:

  ```
  GMAPS_URL = 'http://maps.google.com/maps?q=';
  KML_URL = 'http://your-web-server-path';
  web([GMAPS_URL_KML_URL])
  ```

## See Also

kmlwriteline | kmlwritepoint | makeattribspec | shapewrite

# kmlwriteline

Write geographic line data to KML file

## Syntax

```
kmlwriteline(filename,latitude,longitude)
kmlwriteline(filename,latitude,longitude,altitude)
kmlwriteline( ___ ,Name,Value)
```

## Description

kmlwriteline(filename,latitude,longitude) writes the geographic data, latitude and longitude, to the file specified by filename in Keyhole Markup Language (KML) format. kmlwriteline creates a KML Placemark element for each line, using the latitude and longitude values as the coordinates of the points that define the line. kmlwriteline sets the third coordinate value (altitude) to 0.

kmlwriteline(filename,latitude,longitude,altitude) uses the values of latitude, longitude, and altitude to set the coordinates of the points that define the line. When you specify an altitude value, kmlwriteline sets the AltitudeMode attribute to 'relativeToSeaLevel'.

kmlwriteline( ___ ,Name,Value) specifies name-value pairs that set additional KML feature properties. Parameter names can be abbreviated and are case-insensitive.

## Examples

### Write Line Data to KML File

Load geographic data describing coast lines.

```
load coastlines
```

Define the name of the KML file you want to create.

```
 filename = 'coast.kml';
```

Write the line data to the file, specifying the color and width of the line.

```
kmlwriteline(filename, coastlat, coastlon, 'Color','black', ...
        'Width', 3);
```

### Retrieve GPS Track Log from GPX File and Write Data to KML File

Read the track log from a GPX file. gpxread returns the data as a geopoint vector.

```
S = gpxread('sample_tracks');
```

Define the name of the KML file you want to create.

```
filename = 'track.kml';
```

Write the geographic line data to the file, specifying a description and a name.

```
kmlwriteline(filename, S.Latitude, S.Longitude, S.Elevation, ...
        'Description', S.Metadata.Name, 'Name', 'Track Log');
```

## Input Arguments

### `filename` — Name of KML file to create
character string

Name of KML file to create, specified as a character string. kmlwriteline creates the file in the current folder, unless you specify a full or relative path name. If the file name includes an extension, it must be .kml.

Data Types: char

### `latitude` — Latitude of points
numeric vector in the range [-90 90]

Latitude of points, specified as a numeric vector in the range [-90 90].

Data Types: single | double

### `longitude` — Longitude of points
numeric vector in the range [-180, 180]

Longitude of points, specified as a numeric vector in the range [-180, 180].

Data Types: single | double

**`altitude`** — **Altitude of points**
0 (default) | numeric vector or scalar

Altitude of points, specified as a numeric vector or scalar. Unit of measure is meters.

- If a scalar, `kmlwriteline` applies the value to each point.
- If a vector, you must specify an altitude value for each point. That is, the vector must have the same length as `latitude` and `longitude`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'  '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Name','Point Reyes'`

**`'Name'`** — **Label of line displayed in viewer**
`'Line N'` where `N` is the index of the line (default) | character string

Label of line displayed in viewer, specified as a character string.

If the line contains `NaN` values, `kmlwriteline` places the line segments in a folder labeled `'Line 1'` and labels the line segments `'Segment N'`, where `N` is the index value of the line segment.

Example: `'Name','Point Reyes'`

Data Types: `char`

**`'Description'`** — **Content to be displayed in the line's description balloon**
character string

Content to be displayed in the line's description balloon, specified as a character string. The description appears in the description balloon when the user clicks on either the feature name in the Google Earth Places panel or clicks the line in the viewer window.

You can include basic HTML mark up, however, Google Earth applies some HTML formatting automatically. For example, Google Earth replaces newlines with line break

tags and encloses valid URLs in anchor tags to make them hyperlinks. To see examples of HTML tags that are recognized by Google Earth, view http://earth.google.com.

Example: `'Description','My Description'`

Data Types: `char`

### `'Color'` — Color of line
defined by viewer (default) | ColorSpec

Color of line, specified as a MATLAB Color Specification (`ColorSpec`).

Example: `'Color','red'`

### `'Width'` — Width of line in pixels
1 (default) | positive numeric scalar

Width of line in pixels, specified as a positive numeric scalar.

Example: 'Width',2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

### `'AltitudeMode'` — Interpretation of altitude values
`'clampToGround'` (default) | `'relativeToGround'` | `'relativeToSeaLevel'`

Interpretation of altitude values, specified as one of the following character strings.

| Value | Description |
|---|---|
| `'clampToGround'` | Ignore altitude values and set the feature on the ground. This is the default value when you do not specify `altitude` values. |
| `'relativeToGround'` | Set altitude values relative to the actual ground elevation of a particular feature. |
| `'relativeToSeaLevel'` | Set altitude values relative to sea level, regardless of the actual elevation values of the terrain beneath the feature. Called '`absolute`' in KML terminology. This is the default when you specify `altitude` values. |

Example: `'AltitudeMode','relativeToGround'`

Data Types: `char`

### `'LookAt'` — Position of virtual camera (eye) relative to object being viewed
geopoint vector

Position of the virtual camera (eye) relative to the object being viewed, specified as a geopoint vector. The fields of the geopoint vector, listed below, define the view. `LookAt` is limited to looking down at a feature. To tilt the virtual camera to look above the horizon into the sky, use the `Camera` parameter.

| Property Name | Description | Data Type |
|---|---|---|
| Latitude | Latitude of the line the camera is looking at, in degrees | Scalar double, from -90 to 90 |
| Longitude | Longitude of the line the camera is looking at, in degrees | Scalar double, from −180 to 180 |
| Altitude | Altitude of the line the camera is looking at from the Earth's surface, in meters | Scalar numeric |
| Heading | Camera direction (azimuth), in degrees (optional) | Scalar numeric [0 360], default 0 (true North) |
| Tilt | Angle between the direction of the `LookAt` position and the normal to the surface of the Earth, in degrees (optional) | Scalar numeric [0 90], default 0 (directly above) |
| Range | Distance in meters from the point specified by `latitude`, `longitude`, and `altitude` to the position of the camera. | Scalar numeric, default 0 |
| AltitudeMode | Interpretation of the camera altitude value (optional) | `'relativeToSeaLevel'`, `'clampToGround'`, (default) `'relativeToGround'` |

Example:
`'LookAt',geopoint(lat,lon,'Range',14794.88,'Heading',71.13,'Tilt',66.77)`

### `'Camera'` — Position of virtual camera relative to surface of Earth
geopoint vector

Position of virtual camera (eye) relative to Earth's surface, specified as a geopoint vector. The fields of the geopoint vector, listed below, define the view. `Camera` provides full six degrees of freedom control over the view, so you can position the camera in space and then rotate it around the *x*-, *y*-, and *z*-axes. You can tilt the camera view so that you're looking above the horizon into the sky.

| Property Name | Description | Data Type |
|---|---|---|
| `Latitude` | Latitude of the virtual camera (eye), in degrees | Scalar double, in the range `[-90 90]` |
| `Longitude` | Longitude of the virtual camera, in degrees, | Scalar double, in the range `[-180 180]`. |
| `Altitude` | Distance of the virtual camera from the Earth's surface, in meters | Scalar numeric |
| `Heading` | Direction (azimuth) in degrees (optional) | Scalar numeric `[0 360]`, default `0` (true North) |
| `Tilt` | Camera rotation around the X-axis, in degrees (optional) | Scalar numeric `[0 180]`, default `0` (directly above) |
| `Roll` | Camera rotation in degrees around the Z-axis (optional) | Scalar numeric, in the range `[-180 180]`, default `0` |
| `AltitudeMode` | Specifies how camera altitude is interpreted (optional) | `'relativeToSeaLevel'`, `'clampToGround'`, `'relativeToGround'` (default) |

Example:
```
'Camera',geopoint(lat,lon,'Altitude',2500,'Tilt',85,'Heading',345)
```

# More About

## Tips

- If you do not see your line, set `AltitudeMode` to `'clampToGround'`. If the line appears, then you may have a problem with your altitude value.
- You can view KML files with the Google Earth browser, which must be installed on your computer.

For Windows, use the `winopen` function:

```
winopen(filename)
```

For Linux, if the filename is a partial path, use the following commands:

```
cmd = 'googleearth ';
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

For Mac, if the filename is a partial path, use the following commands:

```
cmd = 'open -a Google\ Earth '
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

· You can also view KML files with a Google Maps browser. The file must be located on a web server that is accessible from the Internet. A private intranet server will not suffice, because Google's server must be able to access the URL that you provide. A template for using Google Maps is listed below:

```
GMAPS_URL = 'http://maps.google.com/maps?q=';
KML_URL = 'http://your-web-server-path';
web([GMAPS_URL_KML_URL])
```

## See Also
kmlwrite | kmlwritepoint | shapewrite

# kmlwritepoint

Write geographic point data to KML file

## Syntax

```
kmlwritepoint(filename,latitude,longitude)
kmlwritepoint(filename,latitude,longitude,altitude)
kmlwritepoint( ___ ,Name,Value)
```

## Description

kmlwritepoint(filename,latitude,longitude) writes the geographic data, latitude and longitude, to the file specified by filename in Keyhole Markup Language (KML) format. kmlwritepoint creates a KML Placemark element for each point, using the latitude and longitude values as coordinates of the points. kmlwritepoint sets the third coordinate value (altitude) to 0.

kmlwritepoint(filename,latitude,longitude,altitude) writes latitude,longitude, and altitude data as point coordinates. When you specify an altitude value, kmlwritepoint sets the AltitudeMode attribute to relativeToSeaLevel.

kmlwritepoint( ___ ,Name,Value) specifies name-value pairs that set additional KML feature properties. Parameter names can be abbreviated and are case-insensitive.

## Examples

### Write a Point Data to KML File with Description and Custom Icon

Specify the latitude and longitude values that define a point.

```
lat =  42.299827;
lon = -71.350273;
```

Define the description, name, and icon for the point.

```
description = sprintf('%s<br>%s</br><br;>%s</br>', ...
       '3 Apple Hill Drive', 'Natick, MA. 01760', ...
       'http://www.mathworks.com');
name = 'The MathWorks, Inc.';
iconDir = fullfile(matlabroot,'toolbox','matlab','icons');
iconFilename = fullfile(iconDir, 'matlabicon.gif');
```

Define the name of the KML file you want to create.

```
 filename = 'MathWorks.kml';
```

Write the point data to the file, with the description, name, and icon.

```
kmlwritepoint(filename, lat, lon, ...
       'Description', description, 'Name', name, 'Icon', iconFilename);
```

### Retrieve Point Data from Shape File and Write Data to KML File

Read the locations of major cities from a shape file into a geostruct.

```
latlim = [ 30; 75];
lonlim = [-25; 45];
cities = shaperead('worldcities.shp','UseGeoCoords', true, ...
       'BoundingBox', [lonlim, latlim]);
```

Get the latitudes, longitudes, and names of the cities from the geostruct.

```
lat = [cities.Lat];
lon = [cities.Lon];
name = {cities.Name};
```

Define the name of the KML file you want to create.

```
filename = 'European_Cities.kml';
```

Write the geographic data to the file, specifying the names of the cities and the size of the icon.

```
kmlwritepoint(filename, lat, lon, 'Name', name, 'IconScale', 2);
```

### Retrieve Point Data from GPX File and Write Data to KML File

Read point data from a GPX file into a geopoint object.

```
S = gpxread('boston_placenames');
```

Define the name of the KML file you want to create.

```
filename = 'Boston_Placenames.kml';
```

Write the point data to the file, specifying a name for each location.

```
kmlwritepoint(filename, S.Latitude, S.Longitude, 'Name', S.Name);
```

### Write Point Data to KML File Using LookAt to Specify View

Specify the latitude and longitude values that define a point. In this example, the location is the Machu Picchu ruins in Peru.

```
lat =  42.299827;
lon = -71.350273;
```

Create a geopoint object to specify the viewing options available through the `LookAt` parameter.

```
lookAt = geopoint(-13.209676, -72.503364, 'Range', 14794.88, ...
                  'Heading', 71.13, 'Tilt', 66.77);
```

Define the name of the KML file you want to create.

```
filename = 'Machu_Picchu.kml';
```

Write the point data to the file, specifying a name.

```
kmlwritepoint(filename, lat, lon, 'LookAt', lookAt, ...
              'Name', 'Machu Picchu');
```

### Write Point Data to KML File Using Camera to Specify View

Specify the latitude and longitude values that define a point. In this example, the location is Mount Ranier.

```
lat_rainier = 46.8533
lon_rainier = -121.7599
```

Create a geopoint vector to specify the position of the virtual camera (eye) using the `Camera` parameter.

```
myview = geopoint(46.7, -121.7,'Altitude',2500,'Tilt',85,'Heading',345)
```

Define the name of the KML file you want to create.

```
filename = 'Mt_Ranier.kml';
```

Write the point data to the file, specifying a name and a custom color for the icon.

```
kmlwritepoint(filename,lat_rainier,lon_rainier,'Name','Mt Rainier',...
        'Color','red','IconScale',2,'Camera',myview)
```

# Input Arguments

### `filename` — Name of KML file to create
character string

Name of KML file to create, specified as a character string. `kmlwritepoint` creates the file in the current folder, unless you specify a full or relative path name. If the file name includes an extension, it must be `.kml`.

Data Types: `char`

### `latitude` — Latitude of points
numeric vector in the range `[-90 90]`

Latitude of points, specified as a numeric vector in the range `[-90 90]`.

Data Types: `single` | `double`

### `longitude` — Longitude of point
numeric vector in the range `[-180, 180]`

Longitude of points, specified as a numeric vector in the range `[-180, 180]`.

Data Types: `single` | `double`

### `altitude` — Altitude of points
0 (default) | numeric vector or scalar

Altitude of points, specified as a numeric vector or scalar. Unit of measure is meters.

- If a scalar, `kmlwritepoint` applies the value to each point.
- If a vector, you must specify an altitude value for each point. That is, the vector must have the same length as `latitude` and `longitude`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Name','Point Reyes'

### 'Name' — Label of point displayed in viewer
'Point N' where N is the index value of the point (default) | character string or cell array of strings

Label of point displayed in viewer, specified as a character string or cell array of strings.

- If a character string, `kmlwritepoint` applies the name to all points.
- If a cell array, you must include a label for each point; that is, the cell array must have the same length as `latitude` and `longitude`.

Example: 'Name', 'Mount Rainier'

Data Types: char | cell

### 'Description' — Content to be displayed in the point's description balloon
character string or cell array of strings

Content to be displayed in the point's description balloon, specified as a character string or a cell array of strings. The description appears in the description balloon when the user clicks on either the feature name in the Google Earth Places panel or clicks the placemark icon in the viewer window.

- If a string, `kmlwritepoint` applies the description to all points.
- If a cell array, you must include description information for each point; that is, the cell array must be the same length as `latitude` and `longitude`.

You can include basic HTML mark up in description, however, Google Earth applies some HTML formatting automatically. For example, Google Earth replaces newlines with line break tags and encloses valid URLs in anchor tags to make them hyperlinks. To see examples of HTML tags that are recognized by Google Earth, view http://earth.google.com.

Example: 'Description', sprintf('%s<br>%s</br><br>%s</br>', '3 Apple Hill Drive', 'Natick, MA. 01760', 'http://www.mathworks.com');

Data Types: `char` | `cell`

### `'Icon'` — File name of a custom icon
defined by viewer, for example, Google Earth uses an image of a push pin. (default) | character string or cell array of strings

File name of a custom icon, specified as character string or cell array of strings.

- If a string, `kmlwritepoint` uses the icon for all points.
- If a cell array, you must specify an icon for each point. That is, the cell array must be the same length as `latitude` and `longitude`.

If the icon filename is not in the current folder, or in a folder on the MATLAB path, you must specify a full or relative pathname. If the string is an Internet URL, the URL must include the protocol type.

Example: 'Icon', 'my_icon.jpg'

Data Types: `char` | `cell`

### `'IconScale'` — Scaling factor for icon
(default) | positive numeric scalar or vector

Scaling factor for the icon, specified as a positive numeric scalar or vector.

- If a scalar, `kmlwritepoint` applies the scaling factor to the icon for all points.
- If a vector, you must specify a scaling factor for each icon. That is, the vector must be the same length as `latitude` and `longitude`.

Example: 'Iconscale', 2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32`

### `'Color'` — Color of icons
defined by viewer (default) | ColorSpec

Color of icons, specified as a MATLAB Color Specification (`ColorSpec`).

- If a character string, `kmlwritepoint` applies the color to all points.
- If a cell array, you must specify a color for each point. That is, the cell array must be the same length as `latitude` and `longitude`.

- If a numeric array, it must be an *m*-by-3 array where *m* is the length of `latitiude` and `longitude`.

Example: `'Color', 'red'`

### `'AltitudeMode'` — Interpretation of altitude values
`'clampToGround'`, if you do not specify altitude values, otherwise,`'relativeToSeaLevel'` (default) | `'clampToGround'`, `'relativeToGround'`, or `'relativeToSeaLevel'`

Interpretation of altitude values, specified as one of the following character strings.

| Value | Description |
|---|---|
| `'clampToGround'` | Ignore altitude values and set the feature on the ground |
| `'relativeToGround'` | Sets altitude values relative to the actual ground elevation of a particular feature |
| `'relativeToSeaLevel'` | Sets altitude values relative to sea level, regardless of the actual elevation values of the terrain beneath the feature. Called 'absolute' in KML terminology. |

Example: `'altitudeMode', 'relativeToGround'`

Data Types: `char`

### `'LookAt'` — Position of the virtual camera (eye) relative to the object being viewed
geopoint vector

Position of the virtual camera (eye) relative to the object being viewed, specified as a geopoint vector. The view is defined by the fields of the geopoint vector, listed in the table below. `LookAt` is limited to looking down at a feature, you cannot tilt the virtual camera to look above the horizon into the sky. To tilt the virtual camera to look above the horizon into the sky, use the `Camera` parameter.

| Property Name | Description | Data Type |
|---|---|---|
| Latitude | Latitude of the point the camera is looking at, in degrees north or south of the Equator (0 degrees) | Scalar double, from -90 to 90 |

| Property Name | Description | Data Type |
|---|---|---|
| Longitude | Longitude of the point the camera is looking at, in degrees, specifying angular distance relative to the Prime Meridian | Scalar double, in the range `[-180 180]`. Values west of the Meridian range from -180 to 0 degrees. Values east of the Meridian range from 0 to 180 degrees |
| Altitude | Altitude of the point the camera is looking at from the Earth's surface, in meters | Scalar numeric, default 0 |
| Heading | Camera direction (azimuth), in degrees (optional) | Scalar numeric `[0 360]`, default 0 (true North) |
| Tilt | Angle between the direction of the LookAt position and the normal to the surface of the earth, in degrees (optional) | Scalar numeric `[0 90]`, default: 0, directly above. |
| Range | Distance in meters from the point specified by `latitude`, `longitude`, and `altitude` to the point where the camera is positioned—the `LookAt` position. | Scalar numeric, default: 0 |
| AltitudeMode | Interpretation of camera altitude value (optional) | String with value: `'relativeToSeaLevel'`, `'clampToGround'`, (default) `'relativeToGround'` |

Example: `'LookAt',geopoint(-13.209676, -72.503364,'Range', 14794.88,'Heading', 71.13, 'Tilt',66.77)`

### `'Camera'` — Position of the virtual camera relative to the surface of the Earth
geopoint vector

Position of the camera relative to the Earth's surface, specified as a geopoint vector. The fields of the geopoint vector, listed below, define the view. Camera provides full six-degrees-of-freedom control over the view, so you can position the camera in space and then rotate it around the *X*, *Y*, and *Z* axes. You can tilt the camera view so that you're looking above the horizon into the sky.

| Property Name | Description | Data Type |
|---|---|---|
| Latitude | Latitude of the virtual camera (eye), in degrees north or south of the Equator (0 degrees) | Scalar double in the range [-90 90] |
| Longitude | Longitude of the virtual camera, in degrees, specifying angular distance relative to the Prime Meridian | Scalar double, in the range [-180 180]. Values west of the Meridian range from -180 to 0 degrees. Values east of the Meridian range from 0 to 180 degrees |
| Altitude | Distance of the virtual camera from the Earth's surface, in meters | Scalar numeric |
| Heading | Direction (azimuth) in degrees (optional) | Scalar numeric [0 360], default 0 (true North) |
| Tilt | Camera rotation around the X axis, in degrees (optional) | Scalar numeric [0 180], default: 0, directly above |
| Roll | Camera rotation in degrees around the Z axis (optional) | Scalar numeric, in the range [-180 180] default: 0 |
| AltitudeMode | Specifies how camera altitude is interpreted. (optional) | 'relativeToSeaLevel', 'clampToGround' , (default) 'relativeToGround' |

- If a scalar, kmlwritepoint applies the value to all the points.
- If a vector, you must include an item for each point; that is, the length must be the same length as latitude and longitude.

Example: 'Camera', geopoint(46.7, -121.7,'Altitude',2500,'Tilt',85,'Heading',345)

## More About

### Tips

- You can view KML files with the Google Earth browser, which must be installed on your computer.

For Windows, use the `winopen` function:

```
winopen(filename)
```

For Linux, if the filename is a partial path, use the following commands:

```
cmd = 'googleearth ';
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

For Mac, if the filename is a partial path, use the following commands:

```
cmd = 'open -a Google\ Earth '
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

- You can also view KML files with a Google Maps browser. The file must be located on a web server that is accessible from the Internet. A private intranet server will not suffice, because Google's server must be able to access the URL that you provide. A template for using Google Maps is listed below:

```
GMAPS_URL = 'http://maps.google.com/maps?q=';
KML_URL = 'http://your-web-server-path';
web([GMAPS_URL_KML_URL])
```

## See Also
kmlwrite | kmlwriteline | shapewrite

# latlon2pix

Convert latitude-longitude coordinates to pixel coordinates

## Syntax

```
[row, col ] = latlon2pix(R,lat,lon)
```

## Description

`[row, col ] = latlon2pix(R,lat,lon)` calculates pixel coordinates `row`, `col` from latitude-longitude coordinates `lat`, `lon`. `R` is either a 3-by-2 referencing matrix that transforms intrinsic pixel coordinates to geographic coordinates, or a geographic raster reference object. `lat` and `lon` are vectors or arrays of matching size. The outputs `row` and `col` have the same size as `lat` and `lon`. `lat` and `lon` must be in degrees.

Longitude wrapping is handled in the following way: Results are invariant under the substitution `lon = lon +/- n * 360` where n is an integer. Any point on the Earth that is included in the image or gridded data set corresponding to `r` will yield row/column values between 0.5 and 0.5 + the image height/width, regardless of what longitude convention is used.

## Examples

Find the pixel coordinates of the upper left and lower right outer corners of a 2-by-2 degree gridded data set.

```
R = makerefmat(1, 89, 2, 2);
[UL_row, UL_col] = latlon2pix(R,  90, 0)      % Upper left
[LR_row, LR_col] = latlon2pix(R, -90, 360)    % Lower right
[LL_row, LL_col] = latlon2pix(R, -90, 0)      % Lower left
```

Note that the in both the 2nd case and 3rd case we get a column value of 0.5, because the left and right edges are on the same meridian and (-90, 360) is the same point as (-90, 0).

## See Also
map2pix | makerefmat | pix2latlon

# lcolorbar

Colorbar with text labels

## Syntax

```
lcolorbar(labels)
lcolorbar(labels,'property',value,...)
hcb = lcolorbar(...)
```

## Description

`lcolorbar(labels)` appends a colorbar with text labels. The labels input is a cell array of label strings. The colorbar is constructed using the current colormap with the label strings applied at the centers of the color bands.

`lcolorbar(labels,'property',value,...)` controls the colorbar's properties. The location of the colorbar is controlled by the `Location` property. Valid entries for `Location` are `'vertical'` (the default) or `'horizontal'`. Properties `TitleString`, `XLabelString`, `YLabelString` and `ZLabelString` set the respective strings. Property `ColorAlignment` controls whether the colorbar labels are centered on the color bands or the color breaks. Valid values for `ColorAlignment` are `'center'` and `'ends'`.

Other valid property-value pairs are any properties and values that can be applied to the title and labels of the colorbar axes.

`hcb = lcolorbar(...)` returns a handle to the colorbar axes.

## Examples

```
figure; colormap(jet(5))
labels = {'apples','oranges','grapes','peaches','melons'};
lcolorbar(labels,'fontweight','bold');
```

## See Also

contourcmap | colormapeditor

# legs

Courses and distances between navigational waypoints

## Syntax

```
[course,dist] = legs(lat,lon)
[course,dist] = legs(lat,lon,method)
[course,dist] = legs(pts,___ )
mat = legs(___ )
```

## Description

`[course,dist] = legs(lat,lon)` returns the azimuths (`course`) and distances (`dist`) between navigational waypoints, which are specified by the column vectors `lat` and `lon`.

`[course,dist] = legs(lat,lon,method)` specifies the logic for the leg characteristics. If the string *method* is `'rh'` (the default), `course` and `dist` are calculated in a rhumb line sense. If *method* is `'gc'`, great circle calculations are used.

`[course,dist] = legs(pts,___ )` specifies waypoints in a single two-column matrix, `pts`.

`mat = legs(___ )` packs up the outputs into a single two-column matrix, `mat`.

This is a navigation function. All angles are in degrees, and all distances are in nautical miles. Track legs are the courses and distances traveled between navigational waypoints.

## Examples

Imagine an airplane taking off from Logan International Airport in Boston (42.3ºN,71ºW) and traveling to LAX in Los Angeles (34ºN,118ºW). The pilot wants to file a flight plan that takes the plane over O'Hare Airport in Chicago (42ºN,88ºW) for a navigational update, while maintaining a constant heading on each of the two legs of the trip.

What are those headings and how long are the legs?

```
lat = [42.3; 42; 34];
long = [-71; -88; -118];
[course,dist] = legs(lat,long,'rh')

course =
  268.6365
  251.2724
dist =
  1.0e+003 *
    0.7569
    1.4960
```

Upon takeoff, the plane should proceed on a heading of about 269º for 756 nautical miles, then alter course to 251º for another 1495 miles.

How much farther is it traveling by not following a great circle path between waypoints? Using rhumb lines, it is traveling

```
totalrh = sum(dist)

totalrh =
    2.2530e+003
```

For a great circle route,

```
[coursegc,distgc] = legs(lat,long,'gc'); totalgc = sum(distgc)

totalgc =
    2.2451e+003
```

The great circle path is less than one-half of one percent shorter.

## See Also
dreckon | gcwaypts | navfix | track

# lightm

Project light objects on map axes

## Syntax

```
h = lightm(lat,lon)
h = lightm(lat,lon,PropertyName,PropertyValue,...)
h = lightm(lat,lon,alt)
```

## Description

h = lightm(lat,lon) projects a light object at the coordinates lat and lon. The handle, h, of the object can be returned.

h = lightm(lat,lon,*PropertyName*,PropertyValue,...) allows the specification of any property name/property value pair supported by the standard MATLAB light function.

h = lightm(lat,lon,alt) allows the specification of an altitude, alt, for the light object. When omitted, the default is an infinite light source altitude.

## Examples

### Add Light to Map

Load data.

```
load topo
```

Create map.

```
axesm globe; view(120,30)
meshm(topo,topolegend); demcmap(topo)
```
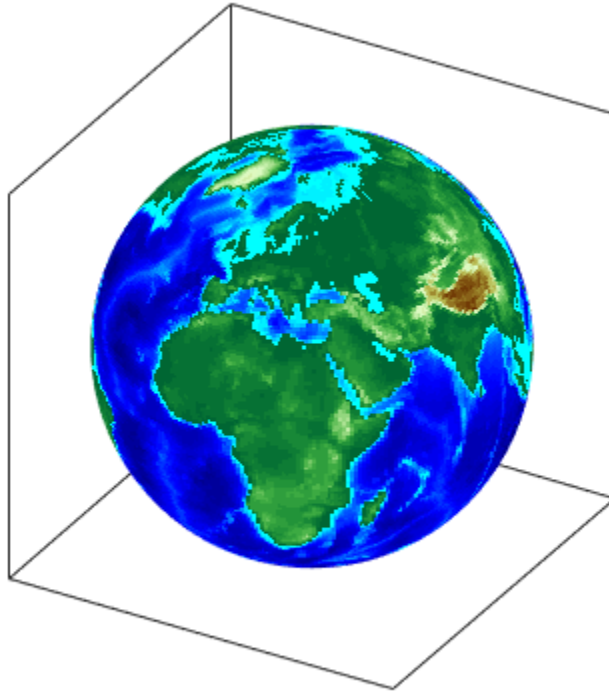
Add light, specifying surface reflectance properties and lighting mode.

```
lightm(0,90,'color','yellow')
material([.5 .5 1]); lighting gouraud
```
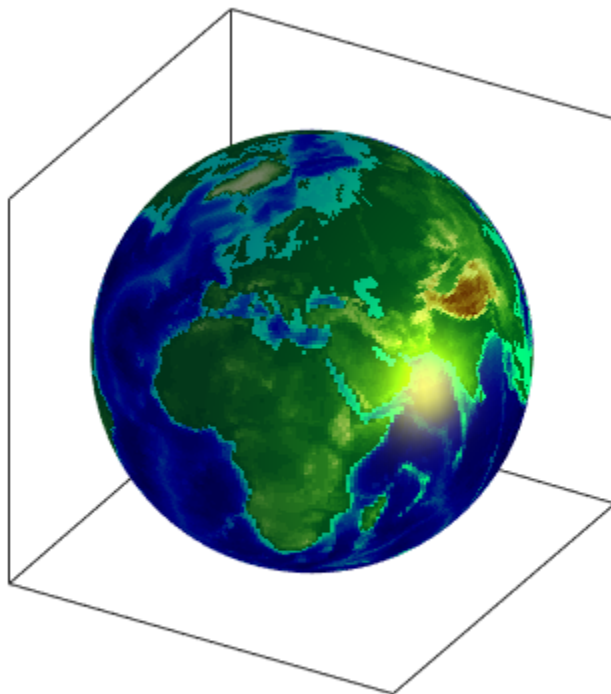
**See Also**
light

# limitm

Determine latitude and longitude limits of regular data grid

## Syntax

```
[latlim,lonlim] = limitm(Z,R)
latlonlim = limitm(Z,R)
```

## Description

`[latlim,lonlim] = limitm(Z,R)` computes the latitude and longitude limits of the geographic quadrangle bounding the regular data grid `Z` spatially referenced by `R`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```
If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```
If `R` is a referencing matrix, it must also define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. The output `latlim` is a vector of the form `[southern_limit northern_limit]` and `lonlim` is a vector of the form `[western_limit eastern_limit]`. All angles are in units of degrees.

`latlonlim = limitm(Z,R)` concatenates `latlim` and `lonlim` into a 1-by-4 row vector of the form:

```
[southern_limit northern_limit western_limit eastern_limit]
```

## Examples

Use a familiar data grid:

```
load topo
[latlimits,lonlimits] = limitm(topo,topolegend)

latlimits =
   -90     90
lonlimits =
     0    360
```

The result is expected, because topo covers the whole globe.

## See Also

makerefmat

# linecirc

Intersections of circles and lines in Cartesian plane

## Syntax

```
[xout,yout] = linecirc(slope,intercpt,centerx,centery,radius)
```

## Description

`[xout,yout] = linecirc(slope,intercpt,centerx,centery,radius)` finds the points of intersection given a circle defined by a center and radius in *x-y* coordinates, and a line defined by slope and *y*-intercept, or a slope of "inf" and an *x*-intercept. Two points are returned. When the objects do not intersect, `NaN`s are returned.

When the line is tangent to the circle, two identical points are returned. All inputs must be scalars.

### See Also

`circcirc`

# linem

Project line object on map axes

## Syntax

```
h = linem(lat,lon)
h = linem(lat,lon,linetype)
h = linem(lat,lon,PropertyName,PropertyValue,...)
h = linem(lat,lon,z)
```

## Description

`h = linem(lat,lon)` displays projected line objects on the current map axes. `lat` and `lon` are the latitude and longitude coordinates, respectively, of the line object to be projected. Note that this ordering is conceptually reversed from the MATLAB `line` function, because the *vertical* (*y*) coordinate comes first. However, the ordering latitude, then longitude, is standard geographic usage. `lat` and `lon` must be the same size and in the `AngleUnits` of the map axes. The object handle for the displayed line can be returned in `h`.

`h = linem(lat,lon,linetype)` allows the specification of the line style, where `linetype` is any string recognized by the MATLAB `line` function.

`h = linem(lat,lon,PropertyName,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB `line` function except for `XData`, `YData`, and `ZData`.

`h = linem(lat,lon,z)` displays a line object in three dimensions, where `z` is the same size as `lat` and `lon` and contains the desired altitude data. `z` is independent of `AngleUnits`. If omitted, all points are assigned a *z*-value of 0 by default.

The units of `z` are arbitrary, except when using the `Globe` projection. In the case of `globe`, `z` should have the same units as the radius of the earth or semimajor axis specified in the `'geoid'` (reference ellipsoid) property of the map axes. This implies that when the reference ellipsoid is a unit sphere, the units of `z` are earth radii.

`linem` is the mapping equivalent of the MATLAB `line` function. It is a low-level graphics function for displaying line objects in map projections. Ordinarily, it is not used directly. Use `plotm` or `plot3m` instead.

## Examples

```
axesm sinusoid; framem
linem([15; 0; -45; 15],[-100; 0; 100; 170],'r-')
```
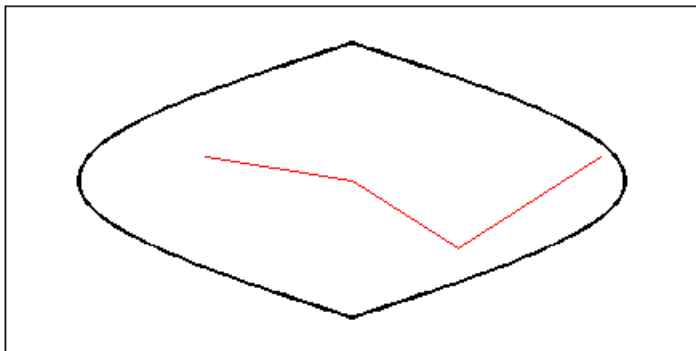


## See Also

`line` | `plot3m` | `plotm`

# los2

Line-of-sight visibility between two points in terrain

## Syntax

```
vis = los2(Z,R,lat1,lon1,lat2,lon2)
vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1)
vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2)
vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt)
vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt,alt2opt)
vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt, ...
  alt2opt,actualradius)
vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt, ...
  alt2opt,actualradius,effectiveradius)
[vis,visprofile,dist,H,lattrk,lontrk] = los2(...)
los2(...)
```

## Description

`los2` computes the mutual visibility between two points on a displayed digital elevation map. `los2` uses the current object if it is a regular data grid, or the first regular data grid found on the current axes. The grid's `zdata` is used for the profile. The color data is used in the absence of data in `z`. The two points are selected by clicking on the map. The result is displayed in a new figure. Markers indicate visible and obscured points along the profile. The profile is shown in a Cartesian coordinate system with the origin at the observer's location. The displayed `z` coordinate accounts for the elevation of the terrain and the curvature of the body.

`vis = los2(Z,R,lat1,lon1,lat2,lon2)` computes the mutual visibility between pairs of points on a digital elevation map. The elevations are provided as a regular data grid `Z` containing elevations in units of meters. The two points are provided as vectors of latitudes and longitudes in units of degrees. The resulting logical variable `vis` is equal to one when the two points are visible to each other, and zero when the line of sight is obscured by terrain. If any of the input arguments are empty, `los2` attempts to gather the data from the current axes. With one or more output arguments, no figures are created and only the data is returned.

R can be a geographic raster reference object, a referencing vector, or a referencing matrix. If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

`vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1)` places the first point at the specified altitude in meters above the surface (on a tower, for instance). This is equivalent to putting the point on a tower. If omitted, point 1 is assumed to be on the surface. `alt1` may be either a scalar or a vector with the same length as `lat1`, `lon1`, `lat2`, and `lon2`.

`vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2)` places both points at a specified altitudes in meters above the surface. `alt2` may be either a scalar or a vector with the same length as `lat1`, `lon1`, `lat2`, and `lon2`. If `alt2` is omitted, point 2 is assumed to be on the surface.

`vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt)` controls the interpretation of `alt1` as either a relative altitude (`alt1opt` equals `'AGL'`, the default) or an absolute altitude (`alt1opt` equals `'MSL'`). If the altitude option is `'AGL'`, `alt1` is interpreted as the altitude of point 1 in meters above the terrain ("above ground level"). If `alt1opt` is `'MSL'`, `alt1` is interpreted as altitude above zero ("mean sea level").

`vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt,alt2opt)` controls the interpretation ALT2.

`vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt, ...`
  `alt2opt,actualradius)` does the visibility calculation on a sphere with the specified radius. If omitted, the radius of the earth in meters is assumed. The altitudes,

elevations and the radius should be in the same units. This calling form is most useful for computations on bodies other than the earth.

vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt, ...
  alt2opt,actualradius,effectiveradius) assumes a larger radius for propagation of the line of sight. This can account for the curvature of the signal path due to refraction in the atmosphere. For example, radio propagation in the atmosphere is commonly treated as straight line propagation on a sphere with 4/3 the radius of the earth. In that case the last two arguments would be R_e and 4/3*R_e, where R_e is the radius of the earth. Use Inf as the effective radius for flat earth visibility calculations. The altitudes, elevations and radii should be in the same units.

[vis,visprofile,dist,H,lattrk,lontrk] = los2(...), for scalar inputs (lat1, lon1, etc.), returns vectors of points along the path between the two points. visprofile is a logical vector containing true (logical(1)) where the intermediate points are visible and false (logical(0)) otherwise. dist is the distance along the path (in meters or the units of the radius). H contains the terrain profile relative to the vertical datum along the path. lattrk and lontrk are the latitudes and longitudes of the points along the path. For vector inputs los2 returns visprofile, dist, H, lattrk, and lontrk as cell arrays, with one cell per element of lat1,lon1, etc.

los2(...), with no output arguments, displays the visibility profile between the two points in a new figure.

## Examples

```
Z = 500*peaks(100);
refvec = [1000 0 0];
[lat1, lon1, lat2, lon2] = deal(-0.027, 0.05, -0.093, 0.042);
los2(Z,refvec,lat1,lon1,lat2,lon2,100);
```
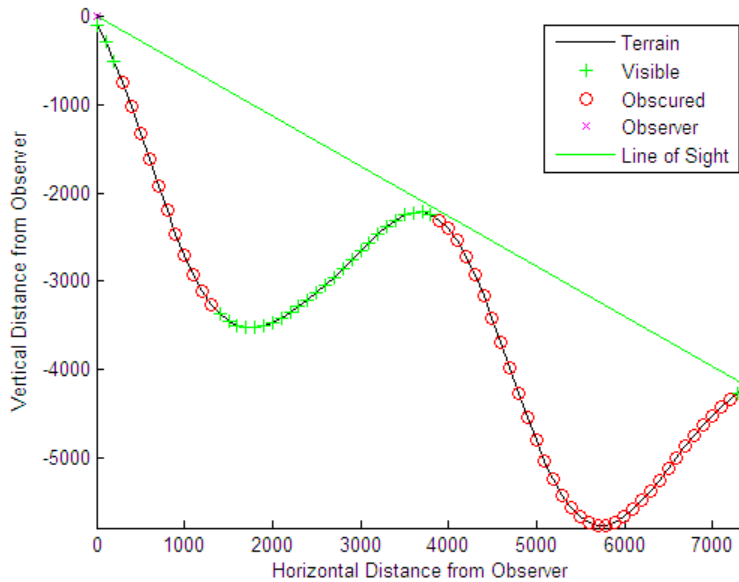
```
figure;
axesm('globe','geoid',earthRadius('meters'))
meshm(Z, refvec, size(Z), Z); axis tight
camposm(-10,-10,1e6); camupm(0,0)
demcmap('inc', Z, 1000); shading interp; camlight
[vis,visprofile,dist,h,lattrk,lontrk] = ...
los2(Z,refvec,lat1,lon1,lat2,lon2,100);
plot3m(lattrk([1;end]),lontrk([1; end]),...
h([1; end])+[100; 0],'r','linewidth',2)
plotm(lattrk(~visprofile),lontrk(~visprofile),...
h(~visprofile),'r.','markersize',10)
plotm(lattrk(visprofile),lontrk(visprofile),...
h(visprofile),'g.','markersize',10)
```
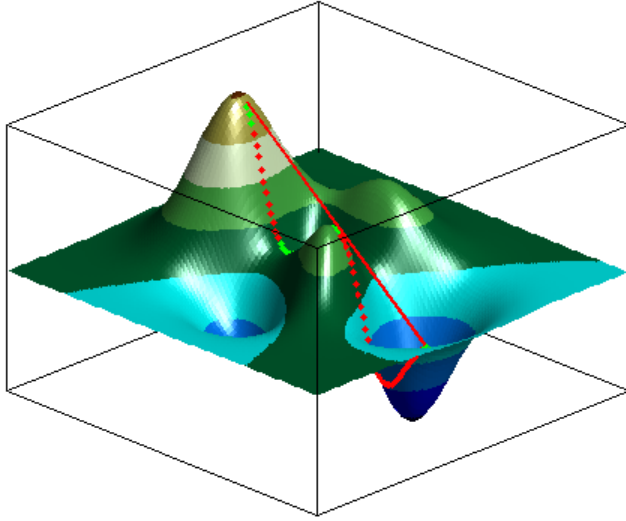
## See Also

viewshed | mapprofile

# ltln2val

Extract data grid values for specified locations

## Syntax

```
val = ltln2val(Z, R, lat, lon)
val = ltln2val(Z, R, lat, lon, method)
```

## Description

`val = ltln2val(Z, R, lat, lon)` interpolates a regular data grid `Z` with referencing vector `R` at the points specified by vectors of latitude and longitude, `lat` and `lon`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

`val = ltln2val(Z, R, lat, lon, method)` accepts a `method` string to specify the type of interpolation: `'bilinear'` for linear interpolation, `'bicubic'` for cubic interpolation, or `'nearest'` for nearest neighbor interpolation.

## Examples

Find the elevations in `topo` associated with three European cities—Milan, Bern, and Prague (`topo` elevations are in meters):

```
load topo
% The city locations, [Milan Bern Prague]
lats = [45.45; 46.95; 50.1];
longs = [9.2; 7.4; 14.45];
elevations = ltln2val(topo,topolegend,lats,longs)

elevations =
 313
 1660
 297
```

## See Also

`findm` | `imbedm`

# lv2ecef

Convert local vertical to geocentric (ECEF) coordinates

---

**Note:** `lv2ecef` will be removed in a future release. Use `enu2ecef` instead. In `enu2ecef`, the latitude and longitude of the local origin are in degrees by default, so the optional `angleUnit` input should be included, with the value `'radians'`.

---

## Syntax

```
[x,y,z] = lv2ecef(xl,yl,zl,phi0,lambda0,h0,ellipsoid)
```

## Description

`[x,y,z] = lv2ecef(xl,yl,zl,phi0,lambda0,h0,ellipsoid)` converts arrays `xl`, `yl`, and `zl` in the local vertical coordinate system to arrays `x`, `y`, and `z` in the geocentric coordinate system. The origin of the local vertical system is at geodetic latitude `phi0`, geodetic longitude `lambda0`, and ellipsoidal height `h0`. The arrays `xl`, `yl`, and `zl` may have any shape, as long as they are all the same size. They are measured in the same length units as the semimajor axis. `phi0` and `lambda0` are scalars measured in radians; `h0` is a scalar with the same length units as the semimajor axis; and `ellipsoid` is a referenceEllipsoid (oblateSpheroid) object, a referenceSphere object, or a vector of the form [`semimajor axis, eccentricity`]). The coordinates `x`, `y`, and `z` also have the same units as the semimajor axis.

## Definitions

For a definition of the local vertical system, also known as East-North-Up (ENU), see the help for `ecef2lv`. For a definition of the geocentric system, also known as Earth-Centered, Earth-Fixed (ECEF), see the help for `geodetic2ecef`.

### See Also
`ecef2enu`

# majaxis

Semimajor axis of ellipse

## Syntax

```
a = majaxis(semiminor,e)
a = majaxis(vec)
```

## Description

`a = majaxis(semiminor,e)` computes the semimajor axis of an ellipse (or ellipsoid of revolution) given the semiminor axis and eccentricity. The input data can be scalar or matrices of equal dimensions.

`a = majaxis(vec)` assumes a 2 element vector (`vec`) is supplied, where `vec` = [`semiminor`, `e`].

## See Also
axes2ecc | flat2ecc | minaxis | n2ecc

# makeattribspec

Attribute specification from geographic data structure

## Syntax

```
attribspec = makeattribspec(S)
```

## Description

`attribspec = makeattribspec(S)` creates an attribute specification from `S` suitable for use with `kmlwrite`. `S` can be any of the following:

- geopoint vector
- geoshape vector, with `'point'` Geometry and no dynamic vertex properties
- geostruct with `'Lat'` and `'Lon'` coordinate fields

The return value, `attribspec`, is a scalar MATLAB structure with two levels. The top level consists of a field for each attribute in `S`. Each of these fields contains a scalar structure with a fixed pair of fields:

| | |
|---|---|
| AttributeLabel | A string that corresponds to the name of the attribute field in `S`. With `kmlwrite`, the string is used to label the attribute in the first column of the HTML table. The string may be modified prior to calling `kmlwrite`. You might modify an attribute label, for example, because you want to use spaces in your HTML table, but the attribute field names in `S` must be valid MATLAB variable names and cannot have spaces themselves. |
| Format | The `sprintf` format character string that converts the attribute value to a string. |

## Examples

1. Import a shapefile representing *tsunami* (tidal wave) events reported between 1950 and 2006 and tagged geographically by source location, and construct a default attribute specification (which includes all the shapefile attributes):

```
s = shaperead('tsunamis', 'UseGeoCoords', true);
attribspec = makeattribspec(s)
attribspec =

          Year: [1x1 struct]
         Month: [1x1 struct]
           Day: [1x1 struct]
          Hour: [1x1 struct]
        Minute: [1x1 struct]
        Second: [1x1 struct]
      Val_Code: [1x1 struct]
      Validity: [1x1 struct]
    Cause_Code: [1x1 struct]
         Cause: [1x1 struct]
        Eq_Mag: [1x1 struct]
       Country: [1x1 struct]
      Location: [1x1 struct]
    Max_Height: [1x1 struct]
      Iida_Mag: [1x1 struct]
     Intensity: [1x1 struct]
    Num_Deaths: [1x1 struct]
   Desc_Deaths: [1x1 struct]
```

**2** Modify the attribute specification to

- Display just the attributes `Max_Height`, `Cause`, `Year`, `Location`, and `Country`

- Rename the `Max_Height` field to `Maximum Height`

- Display each attribute's label in bold type

- Set to zero the number of decimal places used to display `Year`

- Add "Meters" to the `Height` format, given independent knowledge of these units

```
desiredAttributes = ...
      {'Max_Height', 'Cause', 'Year', 'Location', 'Country'};
allAttributes = fieldnames(attribspec);
attributes = setdiff(allAttributes, desiredAttributes);
attribspec = rmfield(attribspec, attributes);
attribspec.Max_Height.AttributeLabel = '<b>Maximum Height</b>';
attribspec.Max_Height.Format = '%.1f Meters';
attribspec.Cause.AttributeLabel = '<b>Cause</b>';
attribspec.Year.AttributeLabel = '<b>Year</b>';
attribspec.Year.Format = '%.0f';
attribspec.Location.AttributeLabel = '<b>Location</b>';
```

```
attribspec.Country.AttributeLabel = '<b>Country</b>';
```

**3**   Use the attribute specification to export the selected attributes and source locations to a KML file as a Description:

```
filename = 'tsunami.kml';
kmlwrite(filename, s, 'Description', attribspec, ...
    'Name', {s.Location})
```

## See also

kmlwrite, makedbfspec, shapewrite

## More About

**Tips**

- The easiest way to construct an attribute specification is to create one, using makeattribspec, and then modify the output, removing attributes or changing the Format field for one or more attributes.

- You can use an attribute specification with kmlwrite as the value of the Description parameter. kmlwrite constructs an HTML table that consists of a label for the attribute in the first column and the string value of the attribute in the second column. You can modify the attribute specification to control which attribute fields are written to the HTML table and the format of the string conversion.

# makedbfspec

DBF specification from geographic data structure

## Syntax

```
dbfspec = makedbfspec(S)
```

## Description

dbfspec = makedbfspec(S) analyzes a geographic data structure, S, and constructs a DBF specification suitable for use with shapewrite. You can modify dbfspec, then pass it to shapewrite to exert control over which geostruct attribute fields are written to the DBF component of the shapefile, the field-widths, and the precision used for numerical values.

dbfspec is a scalar MATLAB structure with two levels. The top level consists of a field for each attribute in S. Each of these fields, in turn, contains a scalar structure with a fixed set of four fields:

| dbfspec field | Contents |
|---|---|
| FieldName | The field name to be used within the DBF file. This will be identical to the name of the corresponding attibute, but may modified prior to calling shapewrite. This might be necessary, for example, because you want to use spaces your DBF field names, but the attribute fieldnames in S must be valid MATLAB variable names and cannot have spaces themselves. |
| *FieldType* | The field type to be used in the file, either 'N' (numeric) or 'C' (character). |
| FieldLength | The number of bytes that each instance of the field will occupy in the file. |
| FieldDecimalCount | The number of digits to the right of the decimal place that are kept in a numeric field. Zero for integer-valued fields and |

| dbfspec field | Contents |
|---|---|
|  | character fields. The default value for noninteger numeric fields is 6. |

## Examples

Import a shapefile representing a small network of road segments, and construct a DBF specification.

```
s = shaperead('concord_roads')

s =
609x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH

dbfspec = makedbfspec(s)

dbfspec =
    STREETNAME: [1x1 struct]
     RT_NUMBER: [1x1 struct]
         CLASS: [1x1 struct]
    ADMIN_TYPE: [1x1 struct]
        LENGTH: [1x1 struct]
```

Modify the DBF spec to (a) eliminate the 'ADMIN_TYPE' attribute, (b) rename the 'STREETNAME' field to 'Street Name', and (c) reduce the number of decimal places used to store road lengths.

```
dbfspec = rmfield(dbfspec,'ADMIN_TYPE')

dbfspec =
    STREETNAME: [1x1 struct]
     RT_NUMBER: [1x1 struct]
         CLASS: [1x1 struct]
```

```
        LENGTH: [1x1 struct]

dbfspec.STREETNAME.FieldName = 'Street Name';
dbfspec.LENGTH.FieldDecimalCount = 1;
```

Export the road network back to a modified shapefile. (Actually, only the DBF component will be different.)

```
shapewrite(s, 'concord_roads_modified', 'DbfSpec', dbfspec)
```

Verify the changes you made. Notice the appearance of `'Street Name'` in the field names reported by shapeinfo, the absence of the `'ADMIN_TYPE'` field, and the reduction in the precision of the road lengths.

```
info = shapeinfo('concord_roads_modified')
info =
        Filename: [3x28 char]
       ShapeType: 'PolyLine'
     BoundingBox: [2x2 double]
     NumFeatures: 609
      Attributes: [4x1 struct]

{info.Attributes.Name}

ans =
    'Street Name'    'RT_NUMBER'    'CLASS'    'LENGTH'

r = shaperead('concord_roads_modified')

r =
609x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    StreetName
    RT_NUMBER
    CLASS
    LENGTH

s(33).LENGTH

ans =
    3.492817400000000e+002
```

```
r(33).LENGTH

ans =
    3.493000000000000e+002
```

## See also

shapeinfo, shapewrite

# makemapped

Convert ordinary graphics object to mapped object

## Compatibility

`makemapped` will be removed in a future release.

## Syntax

```
makemapped(h)
```

## Description

`makemapped(h)` modifies the graphic object(s) associated with `h` such that upon subsequent modification of map axes properties, they are automatically reprojected appropriately. The object's coordinates are not changed by `makemapped`, but will change should you modify the map projection. `h` can be a handle, vector of handles, or any name string recognized by `handlem`. The objects are then considered to be geographic data. You should first trim objects extending outside the map frame to the map frame using `trimcart`.

## Examples

```
axesm('miller','geoid',[25 O])
framem
plot(humps,'b+-')

h = plot(humps,'r+-');
trimcart(h)
makemapped(h)

setm(gca,'MapProjection','sinusoid')
```
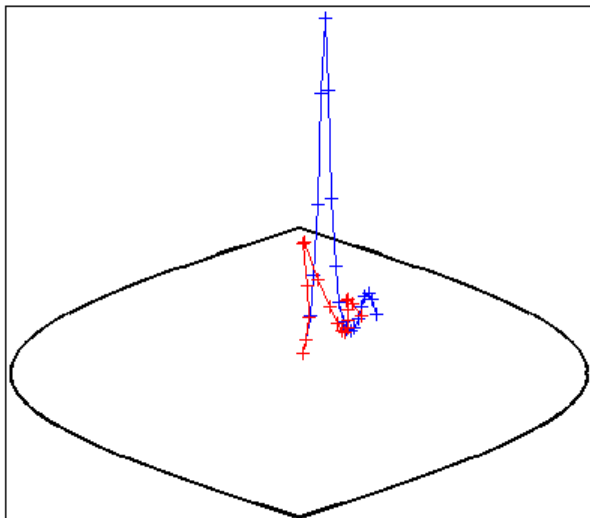
## More About

### Tips

Objects should first be trimmed to the map frame using `trimcart`. This avoids problems in taking inverse map projections with out-of-range data.

# makerefmat

Construct affine spatial-referencing matrix

## Syntax

```
R = makerefmat(x11, y11, dx, dy)
R = makerefmat(lon11, lat11, dlon, dlat)
R = makerefmat(param1, val1, param2, val2, ...)
```

## Description

R = makerefmat(x11, y11, dx, dy), with scalars dx and dy, constructs a referencing matrix that aligns image or data grid rows to map *x* and columns to map *y*. Scalars x11 and y11 specify the map location of the center of the first (1,1) pixel in the image or the first element of the data grid, so that

```
[x11 y11] = pix2map(R,1,1)
```

dx is the difference in *x* (or longitude) between pixels in successive columns, and dy is the difference in *y* (or latitude) between pixels in successive rows. More abstractly, R is defined such that

```
[x11 + (col-1) * dx, y11 + (row-1) * dy] = pix2map(R, row, col)
```

Pixels cover squares on the map when abs(dx) = abs(dy). To achieve the most typical kind of alignment, where *x* increases from column to column and *y* decreases from row to row, make dx positive and dy negative. In order to specify such an alignment along with square pixels, make dx positive and make dy equal to -dx:

```
R = makerefmat(x11, y11, dx, -dx)
```

R = makerefmat(x11, y11, dx, dy), with two-element vectors dx and dy, constructs the most general possible kind of referencing matrix, for which

```
[x11 + ([row col]-1) * dx(:), y11 + ([row col]-1) * dy(:)] ...

 = pix2map(R, row, col)
```

In this general case, each pixel can become a parallelogram on the map, with neither edge necessarily aligned to map *x* or *y*. The vector [dx(1) dy(1)] is the difference in

map location between a pixel in one row and its neighbor in the preceding row. Likewise, `[dx(2) dy(2)]` is the difference in map location between a pixel in one column and its neighbor in the preceding column.

To specify pixels that are rectangular or square (but possibly rotated), choose `dx` and `dy` such that `prod(dx) + prod(dy) = 0`. To specify square (but possibly rotated) pixels, choose `dx` and `dy` such that the 2-by-2 matrix `[dx(:) dy(:)]` is a scalar multiple of an orthogonal matrix (that is, its two eigenvalues are real, nonzero, and equal in absolute value). This amounts to either rotation, a mirror image, or a combination of both. Note that for scalars `dx` and `dy`,

```
R = makerefmat(x11, y11, [0 dx], [dy 0])
```

is equivalent to

```
R = makerefmat(x11, y11, dx, dy)
```

`R = makerefmat(lon11, lat11, dlon, dlat)`, with longitude preceding latitude, constructs a referencing matrix for use with geographic coordinates. In this case,

```
[lat11,lon11] = pix2latlon(R,1,1),
[lat11+(row-1)*dlat,lon11+(col-1)*dlon] = pix2latlon(R,row,col)
```

for scalar `dlat` and `dlon`, and

```
[lat11+[row col]-1)*dlat,lon11+([row col]-1)*dlon] = ...
pix2latlon(R, row,col)
```

for vector `dlat` and `dlon`. Images or data grids aligned with latitude and longitude might already have referencing vectors. In this case you can use function `refvec2mat` to convert to a referencing matrix.

`R = makerefmat(param1, val1, param2, val2, ...)` uses parameter name-value pairs to construct a referencing matrix for an image or raster grid that is referenced to and aligned with a geographic coordinate system. There can be no rotation or skew: each column must fall along a meridian, and each row must fall along a parallel. Each parameter name must be specified exactly as shown, including case.

| Parameter Name | Data Type | Value |
|---|---|---|
| RasterSize | Two-element size vector [M N] | The number of rows (M) and columns (N) of the raster or image to be used with the referencing matrix. |

| Parameter Name | Data Type | Value |
|---|---|---|
| | | With `'RasterSize'`, you may also provide a size vector having more than two elements. This enables usage such as:<br><br>```R = makerefmat('RasterSize', ...<br>    size(RGB), ...)```<br>where `RGB` is `M-by-N-by-3`. However, in cases like this, only the first two elements of the size vector will actually be used. The higher (non-spatial) dimensions will be ignored. The default value is [1 1]. |
| LatitudeLimits | Two-element row vector of the form: `[southern_limit, northern_limit]`, in units of degrees. | The limits in latitude of the geographic quadrangle bounding the georeferenced raster. The default value is [0 1]. |
| LongitudeLimits | Two-element row vector of the form: `[western_limit, eastern_limit]`, in units of degrees. | The limits in longitude of the geographic quadrangle bounding the georeferenced raster. The elements of the `'LongitudeLimits'` vector must be ascending in value. In other words, the limits must be unwrapped. The default value is [0 1]. |
| ColumnsStartFrom | String | Indicates the column direction of the raster (south-to-north vs. north-to-south) in terms of the edge from which row indexing starts. The input string can have the value `'south'` or `'north'`, can be shortened, and is case-insensitive. In a typical terrain grid, row indexing starts at southern edge. In images, row indexing starts at northern edge. The default value is `'south'`. |

| Parameter Name | Data Type | Value |
|---|---|---|
| `RowsStartFrom` | String | Indicates the row direction of the raster (west-to-east vs. east-to-west) in terms of the edge from which column indexing starts. The input string can have the value `'west'` or `'east'`, can be shortened, and is case-insensitive. Rows almost always run from west to east. The default value is `'west'`. |

## Examples

Create a referencing matrix for an image with square, four-meter pixels and with its upper left corner (in a map coordinate system) at $x = 207000$ meters, $y = 913000$ meters. The image follows the typical orientation: $x$ increasing from column to column and $y$ decreasing from row to row.

```
x11 = 207002;  % Two meters east of the upper left corner
y11 = 912998;  % Two meters south of the upper left corner
dx =  4;
dy = -4;
R = makerefmat(x11, y11, dx, dy)
```

Create a referencing matrix for a global geoid grid.

```
% Add array 'geoid' to the workspace:
load geoid

%'geoid' contains a model of the Earth's geoid sampled in
% one-degree-by-one-degree cells. Each column of 'geoid'
% contains geoid heights in meters for 180 cells starting
% at latitude -90 degrees and extending to +90 degrees, for
% a given longitude. Each row contains geoid heights for 360
% cells starting at longitude 0 and extending 360 degrees.
geoidR = makerefmat('RasterSize', size(geoid), ...
    'Latlim', [-90 90], 'Lonlim', [0 360])

% At its most extreme, the geoid reaches a minimum of slightly
% less than -100 meters. This minimum occurs in the Indian Ocean
% at approximately 4.5 degrees latitude, 78.5 degrees longitude.
% Check the geoid height at its most extreme by using latlon2pix
% with the referencing matrix.
```

```
[row, col] = latlon2pix(geoidR, 4.5, 78.5)
geoid(round(row),round(col))
```

# More About

### Spatial Referencing Matrix

A spatial referencing matrix R ties the row and column subscripts of an image or regular data grid to 2-D map coordinates or to geographic coordinates (longitude and geodetic latitude). R is a 3-by-2 affine transformation matrix. R either transforms pixel subscripts (row, column) to/from map coordinates (x,y) according to

```
[x y] = [row col 1] * R
```

or transforms pixel subscripts to/from geographic coordinates according to

```
[lon lat] = [row col 1] * R
```

To construct a referencing matrix for use with geographic coordinates, use longitude in place of X and latitude in place of Y, as shown in the R = makerefmat(X11, Y11, dx, dy) syntax. This is one of the few places where longitude precedes latitude in a function call.

.

## See Also

latlon2pix | map2pix | pix2latlon | pix2map | refvec2mat | worldfileread | worldfilewrite

# makesymbolspec

Construct vector layer symbolization specification

## Syntax

```
symbolspec = makesymbolspec(geometry,rule1,rule2,...ruleN)
```

## Description

`symbolspec = makesymbolspec(geometry,rule1,rule2,...ruleN)` constructs a symbol specification structure (`symbolspec`) for symbolizing a (vector) shape layer in the Map Viewer or when using `mapshow`. `geometry` is one of `'Point'`, `'Line'`, `'PolyLine'`, `'Polygon'`, or `'Patch'`. Rules, defined in detail below, specify the graphics properties for each feature of the layer. A rule can be a default rule that is applied to all features in the layer or it may limit the symbolization to only those features that have a particular value for a specified attribute. Features that do not match any rules are displayed using the default graphics properties.

To create a rule that applies to all features, a default rule, use the following syntax:

```
{'Default',Property1,Value1,Property2,Value2,...
            PropertyN,ValueN}
```

To create a rule that applies only to features that have a particular value or range of values for a specified attribute, use the following syntax:

```
{AttributeName,AttributeValue,
Property1,Value1,Property2,Value2,...,PropertyN,ValueN}
```

`AttributeValue` and `ValueN` can each be a two-element vector, `[low high]`, specifying a range. If `AttributeValue` is a range, `ValueN` might or might not be a range.

The following is a list of allowable values for `PropertyN`.

- Points or Multipoints: `'Marker'`, `'Color'`, `'MarkerEdgeColor'`, `'MarkerFaceColor'`, `'MarkerSize'`, and `'Visible'`

- Lines or PolyLines: `'Color'`, `'LineStyle'`, `'LineWidth'`, and `'Visible'`
- Polygons: `'FaceColor'`, `'FaceAlpha'`, `'LineStyle'`, `'LineWidth'`, `'EdgeColor'`, `'EdgeAlpha'`, and `'Visible'`

## Examples

The following examples import a shapefile containing road data and symbolize it in several ways using symbol specifications.
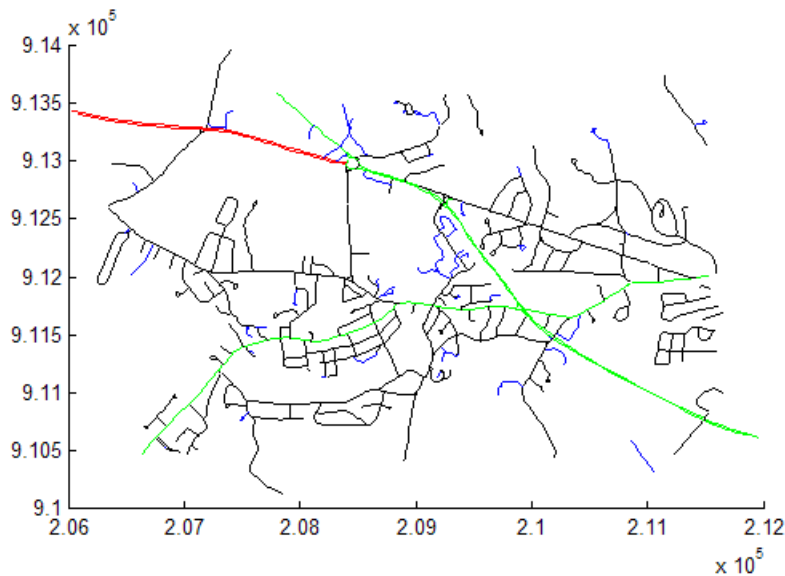
### Example 1 — Default Color

```
roads = shaperead('concord_roads.shp');
blueRoads = makesymbolspec('Line',{'Default','Color',[0 0 1]});
mapshow(roads,'SymbolSpec',blueRoads);
```
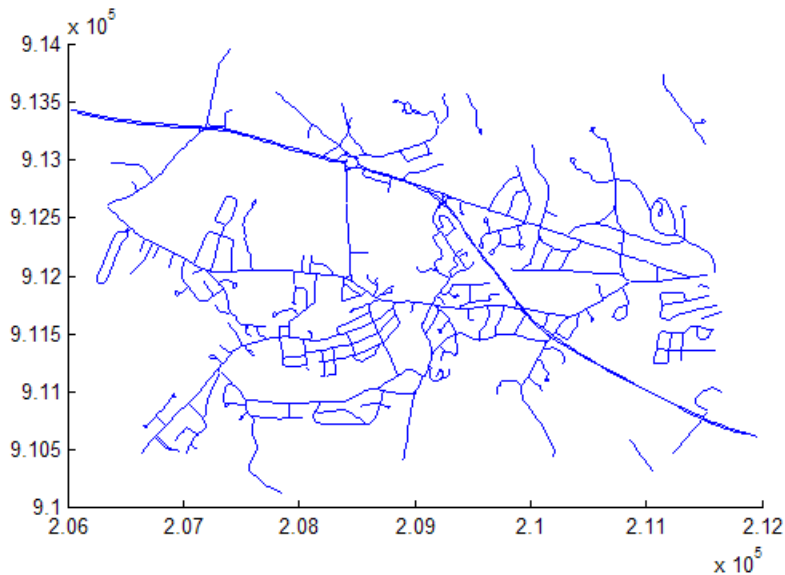


### Example 2 — Discrete Attribute Based

```
roads = shaperead('concord_roads.shp');
```

```
roadColors = ...
makesymbolspec('Line',{'CLASS',2,'Color','r'},...
                     {'CLASS',3,'Color','g'},...
                     {'CLASS',6,'Color','b'},...
                     {'Default','Color','k'});
mapshow(roads,'SymbolSpec',roadColors);
```



## Example 3 — Using a Range of Attribute Values

```
roads = shaperead('concord_roads.shp');
lineStyle = makesymbolspec('Line',...
 {'CLASS',[1 3], 'LineStyle',':'},...
 {'CLASS',[4 6],'LineStyle','-.'});
mapshow(roads,'SymbolSpec',lineStyle);
```
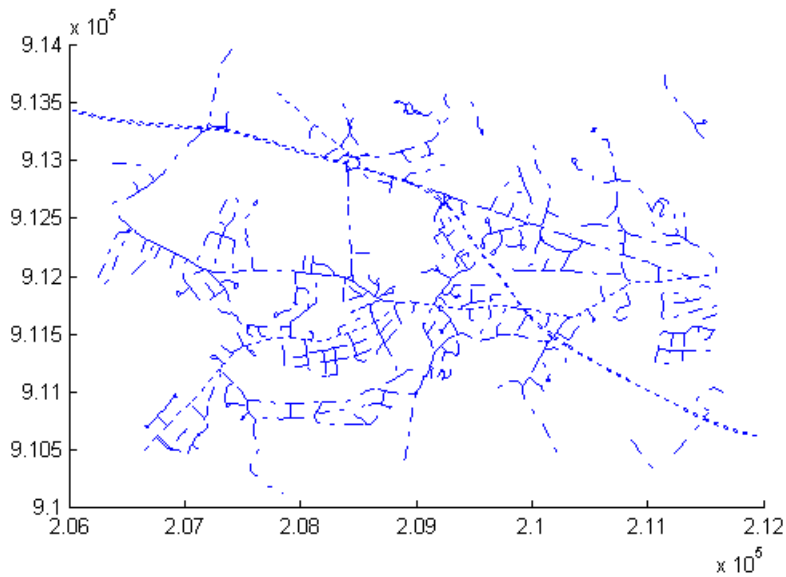
## Example 4 — Using a Range of Attribute Values and a Range of Property Values

```
roads = shaperead('concord_roads.shp');
colorRange = makesymbolspec('Line',...
                            {'CLASS',[1 6],'Color',summer(10)});
mapshow(roads,'SymbolSpec',colorRange);
```
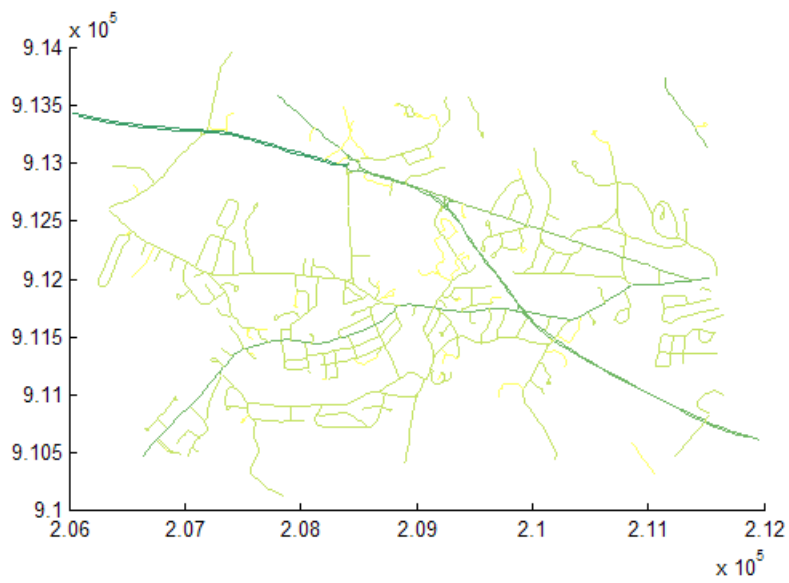
## See Also

`mapshow` | `geoshow` | `mapview`

# map.geodesy.AuthalicLatitudeConverter class

**Package:** map.geodesy

Convert between geodetic and authalic latitudes

## Description

The authalic latitude maps an ellipsoid (oblate spheroid) to a sphere while preserving surface area. Use authalic latitudes when implementing equal area map projections on the ellipsoid. A map.geodesy.AuthalicLatitudeConverter object provides conversion methods between geodetic and authalic latitudes for an ellipsoid with a given eccentricity.

## Construction

`converter = map.geodesy.AuthalicLatitudeConverter` creates an authalic latitude converter object for a sphere (with Eccentricity 0).

`converter = map.geodesy.AuthalicLatitudeConverter(spheroid)` creates an authalic latitude converter object with Eccentricity matching the specified spheroid object.

### Input Arguments

**`spheroid`**

Reference spheroid, specified as a scalar referenceEllipsoid, oblateSpheroid, and referenceSphere object.

## Properties

**`Eccentricity`**

Scalar double falling in the interval [`0` `0.5`]. (Eccentricities larger than `0.5` are possible in theory, but do not occur in practice and are not supported.)

## Methods

| | |
|---|---|
| forward | Geodetic latitude to authalic latitude |
| inverse | Authalic latitude to geodetic latitude |

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

## Examples

### Create an Authalic Converter

Create a Geodetic Reference System 1980 (grs80) reference ellipsoid.

```
grs80 = referenceEllipsoid('GRS 80');
```

Create an authalaic converter object and set the value of the Eccentricity property.

```
conv1 = map.geodesy.AuthalicLatitudeConverter;
conv1.Eccentricity = grs80.Eccentricity
```

```
conv1 =

  AuthalicLatitudeConverter with properties:

    Eccentricity: 0.0818
```

### Create an Authalic Converter Specifying a Spheroid

Create a reference ellipsoid.

```
grs80 = referenceEllipsoid('GRS 80');
```

Create an authalic latitude converter object based on the ellipsoid.

```
conv2 = map.geodesy.AuthalicLatitudeConverter(grs80)
```

```
conv1 =
```

```
AuthalicLatitudeConverter with properties:

  Eccentricity: 0.0818
```

# References

# forward

**Class:** map.geodesy.AuthalicLatitudeConverter
**Package:** map.geodesy

Geodetic latitude to authalic latitude

## Syntax

```
beta = forward(converter,phi)
beta = forward(converter,phi,angleUnit)
```

## Description

`beta = forward(converter,phi)` returns the authalic latitude corresponding to geodetic latitude `phi`.

`beta = forward(converter,phi,angleUnit)` specifies the units of input `phi` and output `beta`.

## Input Arguments

**`converter`**

Authalic latitude converter, specified as a scalar `map.geodesy.AuthalicLatitudeConverter` object.

**`phi`**

Geodetic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

**`angleUnit`**

Units of angles, specified as `'degrees'` or `'radians'`.

**Default:** 'degrees'

## Output Arguments

**beta**

Authalic latitude of each element in phi, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument angleUnit, if supplied; values are in degrees, otherwise.

## Examples

### Convert Geodetic Latitude Values to Authalic Values

```
phi = [-90 -67.5 -45 -22.5 0 22.5 45 67.5 90];
conv = map.geodesy.AuthalicLatitudeConverter(wgs84Ellipsoid);
beta = forward(conv,phi)

beta =

  -90.0000  -67.4092  -44.8717  -22.4094        0   22.4094   44.8717   67.4092   90.
```

# inverse

**Class:** map.geodesy.AuthalicLatitudeConverter
**Package:** map.geodesy

Authalic latitude to geodetic latitude

## Syntax

```
phi = inverse(converter,beta)
phi = inverse(converter,beta,angleUnit)
```

## Description

`phi = inverse(converter,beta)` returns the geodetic latitude corresponding to authalic latitude `beta`.

`phi = inverse(converter,beta,angleUnit)` where `angleUnit` specifies the units of input `beta` and output `phi`.

## Input Arguments

**converter**

Authalic latitude converter, specified as a scalar `map.geodesy.AuthalicLatitudeConverter` object.

**beta**

Authalic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of `double` or `single`. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

**angleUnit**

Units of angles, specified as `'degrees'` or `'radians'`.

**Default:** `'degrees'`

## Output Arguments

**phi**

Geodetic latitude of each element in `beta`, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise.

## Examples

### Convert Authalic Latitude Values to Geodetic Values

```
beta = [-90 -67.4092 -44.8717 -22.4094 0 22.4094 44.8717 67.4092 90];
conv = map.geodesy.AuthalicLatitudeConverter(wgs84Ellipsoid);
phi = inverse(conv,beta)

phi =

  -90.0000  -67.5000  -45.0000  -22.5000        0   22.5000   45.0000   67.5000   90.
```

# map.geodesy.ConformalLatitudeConverter class

**Package:** map.geodesy

Convert between geodetic and conformal latitudes

## Description

The conformal latitude maps an ellipsoid (oblate spheroid) to a sphere while preserving shapes and angles locally. (Curves that meet at a given angle on the ellipsoid meet at the same angle on the sphere.) Use conformal latitudes when implementing conformal map projections on the ellipsoid. A `map.geodesy.ConformalLatitudeConverter` object provides conversion methods between geodetic and conformal latitudes for an ellipsoid with a given eccentricity.

## Construction

`converter = map.geodesy.ConformalLatitudeConverter` creates a conformal latitude converter object for a sphere (with `Eccentricity` 0).

`converter = map.geodesy.ConformalLatitudeConverter(spheroid)` creates a conformal latitude converter object with `Eccentricity` matching the specified spheroid object.

### Input Arguments

**spheriod**

Reference spheroid, specified as a scalar referenceEllipsoid, oblateSpheroid, and referenceSphere object.

## Properties

**Eccentricity**

Scalar double falling in the interval `[0 0.5]`. (Eccentricities larger than 0.5 are possible in theory, but do not occur in practice and are not supported.)

# Methods

| | |
|---|---|
| forward | Geodetic latitude to conformal latitude |
| inverse | Conformal latitude to geodetic latitude |

# Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

# Examples

### Create a Conformal Latitude Convertor Object and Set Property

```
grs80 = referenceEllipsoid('GRS 80');

conv1 = map.geodesy.ConformalLatitudeConverter;
conv1.Eccentricity = grs80.Eccentricity

conv1 =

  ConformalLatitudeConverter with properties:

    Eccentricity: 0.0818
```

### Create a Conformal Latitude Convertor Object Specifying a Spheriod

```
grs80 = referenceEllipsoid('GRS 80');

conv2 = map.geodesy.ConformalLatitudeConverter(grs80)

conv2 =

  ConformalLatitudeConverter with properties:
```

```
Eccentricity: 0.0818
```

## See Also

map.geodesy.AuthalicLatitudeConverter | map.geodesy.IsometricLatitudeConverter | map.geodesy.RectifyingLatitudeConverter | `geocentricLatitude` | `parametricLatitude`

# forward

**Class:** map.geodesy.ConformalLatitudeConverter
**Package:** map.geodesy

Geodetic latitude to conformal latitude

## Syntax

```
chi = forward(converter,phi)
chi = forward(converter,phi,angleUnit)
```

## Description

`chi = forward(converter,phi)` returns the conformal latitude corresponding to the geodetic latitude `phi`.

`chi = forward(converter,phi,angleUnit)` specifies the units of input `phi` and output `chi`.

## Input Arguments

**converter**

Conformal latitude converter, specified as a scalar `map.geodesy.ConformalLatitudeConverter` object.

**phi**

Geodetic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of singles or doubles. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

**angleUnit**

Units of angles, specified as `'degrees'` or `'radians'`.

**Default:** `'degrees'`

# Output Arguments

**chi**

Conformal latitude of each element in `phi`, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise.

# Examples

### Convert Geodetic Latitude Values to Conformal Values

```
phi = [-90 -67.5 -45 -22.5 0 22.5 45 67.5 90];
conv = map.geodesy.ConformalLatitudeConverter(wgs84Ellipsoid);
chi = forward(conv,phi)

chi =

  -90.0000  -67.3637  -44.8077  -22.3643         0   22.3643   44.8077   67.3637   90.
```

# inverse

**Class:** map.geodesy.ConformalLatitudeConverter
**Package:** map.geodesy

Conformal latitude to geodetic latitude

## Syntax

```
phi = inverse(converter,chi)
phi = inverse(converter,chi,angleUnit)
```

## Description

`phi = inverse(converter,chi)` returns the geodetic latitude corresponding to the conformal latitude `chi`.

`phi = inverse(converter,chi,angleUnit)` specifies the units of input `chi` and output `phi`.

## Input Arguments

**converter**

Conformal latitude converter, specified as a scalar `map.geodesy.ConformalLatitudeConverter` object.

**chi**

Conformal latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of `singles` or `doubles`. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise. .

**angleUnit**

Units of angles, specified as `'degrees'` or `'radians'`. Data type: char.

**Default:** `'degrees'`

# Output Arguments

**phi**

Geodetic latitude of each element in `chi`, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise.

# Examples

### Convert Conformal Latitude Values to Geodetic Values

```
chi = [-90 -67.3637 -44.8077 -22.3643 0 22.3643 44.8077 67.3637 90];
conv = map.geodesy.ConformalLatitudeConverter(wgs84Ellipsoid);
phi = inverse(conv,chi)

phi =

  -90.0000  -67.5000  -45.0000  -22.5000         0   22.5000   45.0000   67.5000   90.
```

# map.geodesy.IsometricLatitudeConverter class

**Package:** map.geodesy

Convert between geodetic and isometric latitudes

## Description

The isometric latitude is a nonlinear function of the geodetic latitude that is directly proportional to the spacing of parallels, relative to the Equator, in an ellipsoidal Mercator projection. It is a dimensionless quantity and, unlike other types of auxiliary latitude, the isometric latitude is not angle-valued. It equals `Inf` at the north pole and `-Inf` at the south pole. A `map.geodesy.IsometricLatitudeConverter` object provides conversion methods between geodetic and isometric latitudes for an ellipsoid with a given eccentricity.

## Construction

`converter = map.geodesy.IsometricLatitudeConverter` returns an isometric latitude converter object for a sphere (with `Eccentricity` 0).

`converter = map.geodesy.IsometricLatitudeConverter(spheroid)` returns an isometric latitude converter object with `Eccentricity` matching the specified spheroid object.

### Input Arguments

**spheroid**

Reference spheroid, specified as a scalar referenceEllipsoid, oblateSpheroid, and referenceSphere object.

## Properties

**Eccentricity**

`Eccentricity` is a scalar double falling in the interval [0 0.5]. (Eccentricities larger than 0.5 are possible in theory, but do not occur in practice and are not supported.)

## Methods

forward                                      Convert between geodetic and isometric
                                             latitudes

inverse                                      Isometric latitude to geodetic latitude

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the
MATLAB documentation.

## Examples

**Create an Isometric Latitude Converter Object and Set Property**

```
grs80 = referenceEllipsoid('GRS 80');

conv1 = map.geodesy.IsometricLatitudeConverter;
conv1.Eccentricity = grs80.Eccentricity

conv1 =

  IsometricLatitudeConverter with properties:

    Eccentricity: 0.0818
```

**Create an Isometric Latitude Converter Object Specifying Spheroid**

```
grs80 = referenceEllipsoid('GRS 80');

conv2 = map.geodesy.IsometricLatitudeConverter(grs80)

conv2 =

  IsometricLatitudeConverter with properties:
```

```
Eccentricity: 0.0818
```

## See Also

map.geodesy.AuthalicLatitudeConverter | map.geodesy.ConformalLatitudeConverter | map.geodesy.RectifyingLatitudeConverter | `geocentricLatitude` | `parametricLatitude`

# forward

**Class:** map.geodesy.IsometricLatitudeConverter
**Package:** map.geodesy

Convert between geodetic and isometric latitudes

## Syntax

```
psi = forward(converter,phi)
psi = forward(converter,phi,angleUnit)
```

## Description

`psi = forward(converter,phi)` returns the isometric latitude corresponding to geodetic latitude `phi`.

`psi = forward(converter,phi,angleUnit)` specifies the units of input `phi`.

## Input Arguments

**converter**

Isometric latitude converter, specified as a scalar `map.geodesy.IsometricLatitudeConverter` object.

**phi**

Geodetic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of `singles` or `doubles`. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

**angleUnit**

Units of angles, specified as `'degrees'` or `'radians'`.

**Default:** 'degrees'

## Output Arguments

**psi**

Isometric latitude of each element in phi, returned as a scalar value, vector, matrix, or N-D array. Unlike phi, isometric latitude is a dimensionless number and does not have an angle unit.

## Examples

### Convert Geodetic Latitude Values to Isometric Values

```
phi = [-90 -67.5 -45 -22.5 0 22.5 45 67.5 90];
conv = map.geodesy.IsometricLatitudeConverter(wgs84Ellipsoid);
psi = forward(conv,phi)

psi =

      -Inf   -1.6087   -0.8766   -0.4006        0   0.4006   0.8766   1.6087
```

# inverse

**Class:** map.geodesy.IsometricLatitudeConverter
**Package:** map.geodesy

Isometric latitude to geodetic latitude

## Syntax

```
phi = inverse(converter,psi)
phi = inverse(converter,psi,angleUnit)
```

## Description

`phi = inverse(converter,psi)` returns the geodetic latitude corresponding to the isometric latitude `psi`.

`phi = inverse(converter,psi,angleUnit)` where `angleUnit` specifies the units of output `phi`.

## Input Arguments

**`converter`**

Isometric latitude converter, specified as a scalar `map.geodesy.IsometricLatitudeConverter` object.

**`psi`**

Isometric latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of `singles` or `doubles`. Isometric latitude is a dimensionless number and does not have an angle unit.

**`angleUnit`**

Units of angles, specified as `'degrees'` or `'radians'`.

**Default:** 'degrees'

# Output Arguments

**phi**

Geodetic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of `singles` or `doubles`. Units are determined by the input argument `angleUnit`, if supplied, and in degrees, otherwise.

# Examples

### Convert Isometric Latitude Values to Geodetic Values

```
psi = [-Inf -1.6087 -0.87663 -0.40064 0 0.40064 0.87663 1.6087 Inf];
conv = map.geodesy.IsometricLatitudeConverter(wgs84Ellipsoid);
phi = inverse(conv,psi)

phi =

  Columns 1 through 8

  -90.0000   -67.5001   -44.9998   -22.5002         0   22.5002   44.9998   67.5001

  Column 9

   90.0000
```

# map.geodesy.RectifyingLatitudeConverter class

**Package:** map.geodesy

Convert between geodetic and rectifying latitudes

## Description

The rectifying latitude maps an ellipsoid (oblate spheroid) to a sphere while preserving the distances along the meridians. Rectifying latitudes are used when implementing map projections, such as Equidistant Cylindrical, that preserve such distances. A `map.geodesy.RectifyingLatitudeConverter` object provides conversion methods between geodetic and rectifying latitudes for an ellipsoid with a given third flattening.

## Construction

`converter = map.geodesy.RectifyingLatitudeConverter` returns a rectifying latitude converter object for a sphere (with `ThirdFlattening` 0).

`converter = map.geodesy.RectifyingLatitudeConverter(spheroid)` returns a rectifying latitude converter object with `ThirdFlattening` matching the specified spheroid object.

### Input Arguments

**`spheroid`**

Reference spheroid, specified as a scalar referenceEllipsoid, oblateSpheroid, and referenceSphere object.

## Properties

**`ThirdFlattening`**

Scalar double falling in the interval `[0, ecc2n(0.5)]`, or approximately `[0 0.071797]`. (Flatter spheroids are possible in theory, but do not occur in practice and are not supported.)

# Methods

| | |
|---|---|
| forward | Geodetic latitude to rectifying latitude |
| inverse | Rectifying latitude to geodetic latitude |

# Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

# Examples

### Create a Rectifying Latitude Converter Object and Set Property

```
grs80 = referenceEllipsoid('GRS 80');

conv1 = map.geodesy.RectifyingLatitudeConverter;
conv1.ThirdFlattening = grs80.ThirdFlattening

conv1 =

  RectifyingLatitudeConverter with properties:

    ThirdFlattening: 0.0017
```

### Create a Rectifying Latitude Converter Object, Specifying Spheroid

```
grs80 = referenceEllipsoid('GRS 80');

conv2 = map.geodesy.RectifyingLatitudeConverter(grs80)

conv2 =

  RectifyingLatitudeConverter with properties:
```

```
ThirdFlattening: 0.0017
```

## See Also

map.geodesy.AuthalicLatitudeConverter | map.geodesy.ConformalLatitudeConverter | map.geodesy.IsometricLatitudeConverter | `geocentricLatitude` | `parametricLatitude`

# forward

**Class:** map.geodesy.RectifyingLatitudeConverter
**Package:** map.geodesy

Geodetic latitude to rectifying latitude

## Syntax

```
mu = forward(converter,phi)
mu = forward(converter,phi,angleUnit)
```

## Description

`mu = forward(converter,phi)` returns the rectifying latitude corresponding to geodetic latitude `phi`.

`mu = forward(converter,phi,angleUnit)` specifies the units of input `phi` and output `mu`.

## Input Arguments

**converter**

Rectifying latitude converter, specified as a scalar `map.geodesy.IsometricLatitudeConverter` object.

**phi**

Geodetic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of `singles` or `doubles`. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

**angleUnit**

Units of angles, specified as `'degrees'` or `'radians'`.

**Default:** 'degrees'

## Output Arguments

**mu**

Rectifying latitude of each element in phi, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument angleUnit, if supplied; values are in degrees, otherwise.

## Examples

### Convert Geodetic Latitude Values to Rectifying Latitudes

```
phi = [-90 -67.5 -45 -22.5 0 22.5 45 67.5 90];
conv = map.geodesy.RectifyingLatitudeConverter(wgs84Ellipsoid);
mu = forward(conv,phi)

mu =

  -90.0000  -67.3978  -44.8557  -22.3981        0   22.3981   44.8557   67.3978   90.
```

# inverse

**Class:** map.geodesy.RectifyingLatitudeConverter
**Package:** map.geodesy

Rectifying latitude to geodetic latitude

## Syntax

```
phi = inverse(converter,mu)
phi = inverse(converter,mu,angleUnit)
```

## Description

`phi = inverse(converter,mu)` returns the geodetic latitude corresponding to rectifying latitude `mu`.

`phi = inverse(converter,mu,angleUnit)` specifies the units of input `mu` and output `phi`.

## Input Arguments

**`converter`**

Rectifying latitude converter, specified as a scalar `map.geodesy.RectifyingLatitudeConverter` object.

**`mu`**

Rectifying latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array of `singles` or `doubles`. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

**`angleUnit`**

Units of angles, specified as `'degrees'` or `'radians'`.

**Default:** 'degrees'

## Output Arguments

**phi**

Geodetic latitude of each element in mu, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument angleUnit, if supplied; values are in degrees, otherwise.

## Examples

### Convert Rectifying Latitude Values to Geodetic Latitudes

```
mu = [-90  -67.3978 -44.8557 -22.3981   0 22.3981 44.8557 67.3978  90];
conv = map.geodesy.RectifyingLatitudeConverter(wgs84Ellipsoid);
phi = inverse(conv,mu)

phi =

  -90.0000  -67.5000  -45.0000  -22.5000         0   22.5000   45.0000   67.5000  90.
```

# map2pix

Convert map coordinates to pixel coordinates

## Syntax

```
[row,col] = map2pix(R,x,y)
p = map2pix(R,x,y)
[...] = map2pix(R,s)
```

## Description

`[row,col] = map2pix(R,x,y)` calculates pixel coordinates `row,col` from map coordinates `x,y`. `R` is either a 3-by-2 referencing matrix defining a 2-dimensional affine transformation from intrinsic pixel coordinates to map coordinates, or a map raster reference object. `x` and `y` are vectors or arrays of matching size. The outputs `row` and `col` have the same size as `x` and `y`.

`p = map2pix(R,x,y)` combines `row` and `col` into a single array `p`. If `x` and `y` are column vectors of length `n`, then `p` is an `n`-by-2 matrix and each `p(k,:)` specifies the pixel coordinates of a single point. Otherwise, `p` has size `[size(row) 2]`, and `p(k1,k2,...,kn,:)` contains the pixel coordinates of a single point.

`[...] = map2pix(R,s)` combines `x` and `y` into a single array `s`. If `x` and `y` are column vectors of length `n`, the `s` should be an `n`-by-2 matrix such that each row (`s(k,:)`) specifies the map coordinates of a single point. Otherwise, `s` should have size `[size(X) 2]`, and `s(k1,k2,...,kn,:)` should contain the map coordinates of a single point.

## Examples

```
% Find the pixel coordinates for the spatial coordinates
% (207050, 912900)
R = worldfileread('concord_ortho_w.tfw');
[r,c] = map2pix(R, 207050, 912900);
```

## See Also

latlon2pix | pix2map | makerefmat | worldfileread

# mapbbox

Compute bounding box of georeferenced image or data grid

## Syntax

```
bbox = mapbbox(R,height,width)
bbox = mapbbox(R, sizea)
BBOX = mapbbox(info)
```

## Description

`bbox = mapbbox(R,height,width)` computes the 2-by-2 bounding box of a georeferenced image or regular gridded data set. `R` is either a 3-by-2 referencing matrix defining a 2-dimensional affine transformation from intrinsic pixel coordinates to map coordinates, or a map.rasterref.MapCellsReference object. (If you are using a raster reference object, you can calculate the bounding box from the object limit properties of the object rather than using this function.) `height` and `width` are the image dimensions. `bbox` bounds the outer edges of the image in map coordinates:

```
[minX minY
 maxX maxY]
```

`bbox = mapbbox(R, sizea)` accepts `sizea = [height, width, ...]` instead of `height` and `width`.

`BBOX = mapbbox(info)` accepts a scalar struct array with the fields

| | |
|---|---|
| `'RefMatrix'` | 3-by-2 referencing matrix |
| `'Height'` | Scalar number |
| `'Width'` | Scalar number |

## See Also

geotiffinfo | makerefmat | mapoutline | pixcenters | pix2map

# maplist

Available Mapping Toolbox map projections

## Syntax

```
list = maplist
[list,defproj] = maplist
```

## Description

`list = maplist` returns a structure that lists all the available Mapping Toolbox map projections. The list structure is `list.Name`, `list.IdString`, `list.Classification`, `list.ClassCode`. This list structure is used by the functions `maps` and `axesmui` when processing map projection identifiers during operation of the toolbox functions.

`[list,defproj] = maplist` also returns the default projection's `IdString`.

`list.Name` defines the full name of the projection. This entry is used in the command-line table display and in the Projection Control Box.

`list.IdString` provides the name of the MATLAB function that computes the projection.

`list.Classification` defines the projection classification that is used in the command-line table display.

`list.ClassCode` defines the character string that is used to label the classes of projections in the Projection Control Box. The eight class codes are

- `Azim` — Azimuthal
- `Coni` — Conic
- `Cyln` — Cylindrical
- `Mazi` — Modified azimuthal
- `Pazi` — Pseudoazimuthal
- `Pcon` — Pseudoconic

- `Pcy` — Pseudocylindrical
- `Poly` — Polyconic

## See Also
maps | axesmui

# mapoutline

Compute outline of georeferenced image or data grid

## Syntax

```
[x,y] = mapoutline(R, height, width)
[x,y] = mapoutline(R, sizea)
[x,y] = mapoutline(info)
[x,y] = mapoutline(...,'close')
[lon,lat] = mapoutline(R,...)
outline = mapoutline(...)
```

## Description

`[x,y] = mapoutline(R, height, width)` computes the outline of a georeferenced image or regular gridded data set in map coordinates. `R` is either a 3-by-2 referencing matrix defining a 2-dimensional affine transformation from intrinsic pixel coordinates to map coordinates, or a map.rasterref.MapCellsReference ormap.rasterref.MapPostingsReference object. `height` and `width` are the image dimensions. `x` and `y` are 4-by-1 column vectors containing the map coordinates of the outer corners of the corner pixels, in the following order:

```
(1,1), (height,1), (height, width), (1, width).
```

`[x,y] = mapoutline(R, sizea)` accepts `sizea = [height, width, ...]` instead of `height` and `width`.

`[x,y] = mapoutline(info)` accepts a scalar struct array with the fields

| | |
|---|---|
| `'RefMatrix'` | 3-by-2 referencing matrix |
| `'Height'` | Scalar number |
| `'Width'` | Scalar number |

`[x,y] = mapoutline(...,'close')` returns `x` and `y` as 5-by-1 vectors, appending the coordinates of the first of the four corners to the end.

`[lon,lat] = mapoutline(R,...)`, where `R` georeferences pixels to longitude and latitude rather than map coordinates, returns the outline in geographic coordinates. Longitude must precede latitude in the output argument list.

`outline = mapoutline(...)` returns the corner coordinates in a 4-by-2 or 5-by-2 array.

## Examples

Draw a red outline delineating the Boston GeoTIFF image, which is referenced to the Massachusetts Mainland State Plane coordinate system with units of survey feet.

```
figure
info  = geotiffinfo('boston.tif');
[x,y] = mapoutline(info, 'close');
hold on
plot(x,y,'r')
xlabel('MA Mainland State Plane easting, survey feet')
ylabel('MA Mainland State Plane northing, survey feet')
```

Draw a black outline delineating a TIFF image of Concord, Massachusetts, while lies roughly 25 km north west of Boston. Convert world file units to survey feet from meters to be consistent with the Boston image.

```
info  = imfinfo('concord_ortho_w.tif');
R = worldfileread('concord_ortho_w.tfw');
R = R * unitsratio('sf','meter');
[x,y] = mapoutline(R, info.Height, info.Width, 'close');
plot(x,y,'k')
```
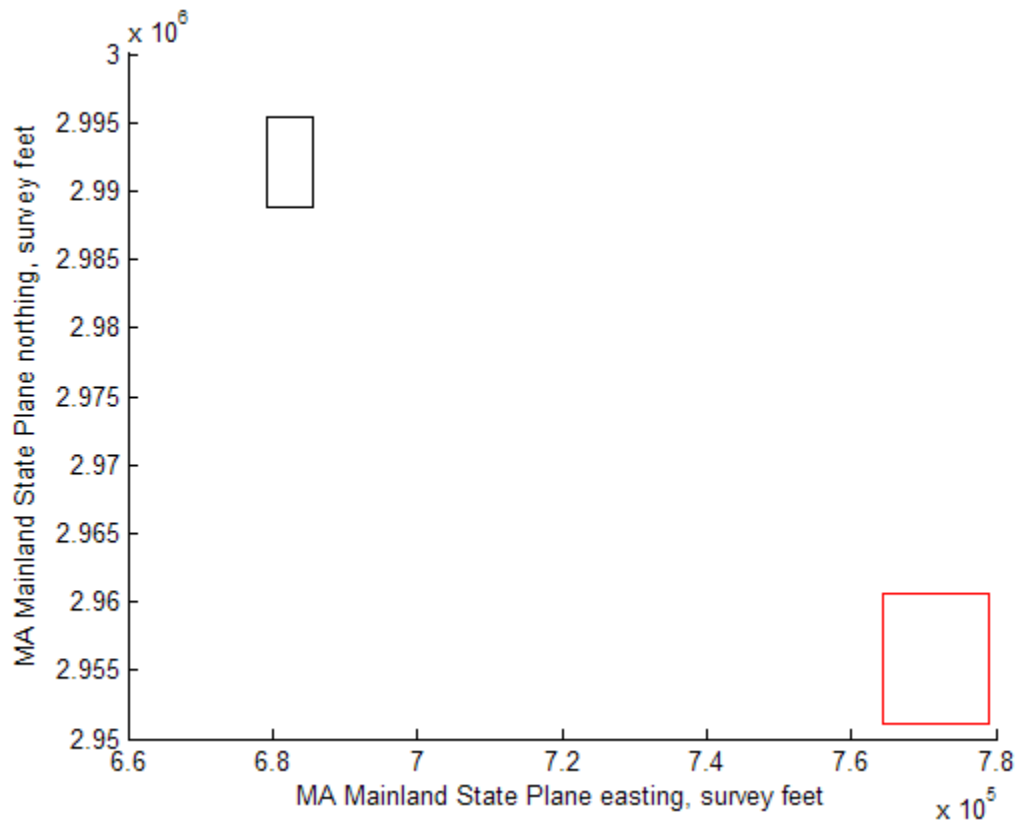
## See Also

map.rasterref.MapCellsReference | map.rasterref.MapPostingsReference | `makerefmat` | `mapbbox` | `pix2map` | `pixcenters`

# mappoint class

Planar point vector

## Syntax

```
p = mappoint()
p = mappoint(x,y)
p = mappoint(x,y,Name,Value)
p = mappoint(structArray)
p = mappoint(x,y,structArray)
```

## Description

A mappoint vector is a container object that holds planar point coordinates and attributes. The points are coupled, such that the size of the X and Y coordinate arrays are always equal and match the size of any dynamically added attribute arrays. Each entry of a coordinate pair and associated attributes, if any, represent a discrete element in the mappoint vector.

## Construction

`p = mappoint()` constructs an empty mappoint vector, `p`, with these default property settings:

```
p =

 0x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: []
           Y: []
```

For an additional example, see "Construct a Default mappoint Vector" on page 1-798.

`p = mappoint(x,y)` constructs a new mappoint vector and assigns the `X` and `Y` properties to the numeric array inputs, `lat` and `lon`. For an example, see "Construct a mappoint Vector from x and y Values" on page 1-798.

`p = mappoint(x,y,Name,Value)` constructs a mappoint vector from input arrays `x` and `y`, and then adds dynamic properties to the mappoint vector using the `Name`, `Value` argument pairs.

• If a specified name is `Metadata` and the corresponding `Value` is a scalar structure, then the value is copied to the `Metadata` property. Otherwise, an error is issued.

For an example, see "Construct a mappoint Vector from x, y, and Temperature Values" on page 1-799.

`p = mappoint(structArray)` constructs a new `mappoint` vector from the fields of the structure, `structArray`. See for an example.

• If `structArray` is a scalar structure containing the field `Metadata` and the field value is a scalar structure, then the `Metadata` field is copied to the `Metadata` property. Otherwise, an error is issued if the `Metadata` field is not a structure, or ignored if `structArray` is not scalar.

• Other fields of `structArray` are assigned to `p` and become dynamic properties. Field values in `structArray` that are not numeric or strings or cell arrays of numeric or string values are ignored.

For an example, see "Construct a mappoint Vector from a Structure Array" on page 1-799.

`p = mappoint(x,y,structArray)` constructs a new `mappoint` vector and sets the `X` and `Y` properties equal to the numeric arrays, `x` and `y`, and sets dynamic properties from the field values of the structure, `structArray`.

• If `structArray` is a scalar structure containing the field `Metadata`, and the field value is a scalar structure, then it is copied to the `Metadata` property. Otherwise, an error is issued if the `Metadata` field is not a structure, or ignored if `structArray` is not scalar.

**1-793**

For an example, see "Construct a mappoint Vector from x and y Numeric Arrays and a Structure Array" on page 1-800.

## Input Arguments

### x — Vector of latitude coordinates
numeric array

Vector of latitude coordinates, specified as a numeric array.

Data Types: `double` | `single`

### y — Vector of longitude coordinates
numeric array

Vector of longitude coordinates, specified as a numeric array.

Data Types: `double` | `single`

### `structArray` — Dynamic properties
structure

Dynamic properties, specified as an array of structures containing fields to be assigned.

### `Name`
string

Name of dynamic property, specified as a string.

Data Types: `char`

### `Value` — Value associated with a dynamic property specified in `Name`
numeric, logical, char, or a cell array of strings

Property value associated with dynamic property `Name`. Values may be numeric, logical, char, or a cell array of strings.

## Output Arguments

### `p`

mappoint vector.

# Properties

Each element in a mappoint vector is considered a feature. Feature properties contain one value (a scalar number or a string) for each element in the mappoint vector. The `Latitude` and `Longitude` coordinate properties are feature properties as there is one value for each feature.

`Geometry` and `Metadata` are collection properties. These properties contain only one value per class instance. The term *collection* is used to distinguish these two properties from other feature properties which have values associated with each feature (element in a mappoint vector).

You can attach new dynamic feature properties to the object by using dot '.' notation. This is similar to adding dynamic fields to a structure. Dynamic feature properties apply to each individual feature in the mappoint vector. See "Construct a mappoint Vector for Two Features and Add Features Dynamically" on page 1-801 for an example.

**Geometry**

String defining the type of geometry.

For `mappoint`, string is always `'point'`.

**Attributes:**

| | |
|---|---|
| Geometry | string |

**Metadata**

Metadata is a scalar structure containing information for the entire set of mappoint vector elements. You can add any data type to the structure.

**Attributes:**

| | |
|---|---|
| Metadata | Scalar struct |

**X**

Vector of planar X coordinates. The values can be either a row or column vector.

**Attributes:**

```
X                                    single | double vector
```

**Y**

Vector of planar Y coordinates. The values can be either a row or column vector.

**Attributes:**

```
Y                                    single | double vector
```

**Dynamic properties**

You can attach new properties to the object using dot '.' notation. The class type of the values for the dynamic properties must be either numeric, logical, char, or a cell array of strings.

# Methods

| | |
|---|---|
| append | Append features to mappoint vector |
| cat | Concatenate mappoint vectors |
| disp | Display mappoint vector |
| fieldnames | Dynamic properties of mappoint vector |
| isempty | True if `mappoint` vector is empty |
| isfield | Returns true if dynamic property exists |
| isprop | Returns true if property exists |
| length | Number of elements in mappoint vector |
| properties | properties of a mappoint vector |
| rmfield | Remove dynamic property from mappoint vector |
| rmprop | Remove properties from mappoint vector |
| size | Size of `mappoint` vector |
| struct | Convert mappoint vector to scalar structure |

| vertcat | Vertical concatenation for mappoint vectors |
|---|---|

# Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

# Class Behaviors

- This example builds a default mappoint object and then dynamically adds a single feature.

  "Construct a mappoint Vector for One Feature" on page 1-800

- mappoint vectors autoresize all properties lengths to ensure they are equal in size when a new dynamic property is added or an existing property appended or shortened.

  "Construct a mappoint Vector for Two Features and Add Features Dynamically" on page 1-801

- The class syntax containing the `Name-Value` pair allows features to be included during object construction.

  "Construct a mappoint Specifying Two Features" on page 1-804

- The data for the following example resides in a MAT file containing oceanic depths.

  "Create a mappoint Vector from a MAT File" on page 1-805

- This example also accesses data from a file and loads it into a structure array. Features are then added to the mappoint illustrating a number of behaviors.

  "Assign Dynamic Features to mappoint Vector from a Structure Array" on page 1-807

- In this example, a feature is added to the mappoint vector using linear indexing.

  "Append a Point by Indexing" on page 1-809

- Features can be sorted by using the indexing behavior of the mappoint class.

  "Sort Dynamic Properties" on page 1-810

- This example demonstrates that input arguments x and y can be either row or column vectors.

  "Row and Column Input Arguments" on page 1-812

## Examples

### Construct a Default mappoint Vector

Dynamically set the X and Y property values, and dynamically add Vertex property Z.

```
p = mappoint();
p.X = 1:3;
p.Y = 1:3;
p.Z = [10 10 10]

p =

 3x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1 2 3]
           Y: [1 2 3]
           Z: [10 10 10]
```

### Construct a mappoint Vector from x and y Values

Create a mappoint.

```
x = [40 50 60];
y = [10, 11, 12];
p = mappoint(x, y)

p =

 3x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
```

```
    Metadata: [1x1 struct]
 Feature properties:
            X: [40 50 60]
            Y: [10 11 12]
```

### Construct a mappoint Vector from x, y, and Temperature Values

```
x = 41:43;
y = 1:3;
temperature = 61:63;
p = mappoint(x, y, 'Temperature', temperature)

p =

 3x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
                X: [41 42 43]
                Y: [1 2 3]
     Temperature: [61 62 63]
```

### Construct a mappoint Vector from a Structure Array

Create a structure array and then create a mappoint vector, specifying the array as input.

```
structArray = shaperead('boston_placenames')
p = mappoint(structArray)

structArray =

13x1 struct array with fields:
    Geometry
    X
    Y
    NAME
    FEATURE
    COORD


p =
```

```
 13x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1x13 double]
           Y: [1x13 double]
        NAME: {1x13 cell}
     FEATURE: {1x13 cell}
       COORD: {1x13 cell
```

### Construct a mappoint Vector from x and y Numeric Arrays and a Structure Array

```
[structArray, A] = shaperead('boston_placenames');
x = [structArray.X];
y = [structArray.Y];
p = mappoint(x, y, A)

p =

 13x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1x13 double]
           Y: [1x13 double]
        NAME: {1x13 cell}
     FEATURE: {1x13 cell}
       COORD: {1x13 cell}
```

### Construct a mappoint Vector for One Feature

```
x = 1;
y = 1;
p = mappoint(x, y)

p =

 1x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
```

```
        Metadata: [1x1 struct]
 Feature properties:
             X: 1
             Y: 1
```

Add a feature dynamic property with a string value.

```
p.FeatureName = 'My Feature'

p =

 1x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
                 X: 1
                 Y: 1
     FeatureName: 'My Feature'
```

**Construct a mappoint Vector for Two Features and Add Features Dynamically**

Create a mappoint vector.

```
x = [1 2];
y = [10 10];
p = mappoint(x, y)

p =

 2x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1 2]
           Y: [10 10]
```

Add a feature dynamic property.

```
p.FeatureName = {'Feature 1', 'Feature 2'}

p =
```

```
2x1 mappoint vector with properties:

Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
Feature properties:
             X: [1 2]
             Y: [10 10]
   FeatureName: {'Feature 1' 'Feature 2'}
```

Add a numeric feature dynamic property.

```
p.ID = [1 2]

p =

 2x1 mappoint vector with properties:

 Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
 Feature properties:
             X: [1 2]
             Y: [10 10]
   FeatureName: {'Feature 1' 'Feature 2'}
            ID: [1 2]
```

Add a third feature. All properties are autosized so that all vector lengths match.

```
p(3).X = 3
p(3).Y = 10

p =

 3x1 mappoint vector with properties:

 Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
 Feature properties:
             X: [1 2 3]
             Y: [10 10 10]
   FeatureName: {'Feature 1' 'Feature 2' ''}
            ID: [1 2 0]
```

Set the values for the ID feature dynamic property with more values than contained in X or Y. All properties are expanded to match in size.

```
p.ID = 1:4

p =

 4x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [1 2 3 0]
              Y: [10 10 10 0]
    FeatureName: {'Feature 1' 'Feature 2' '' ''}
             ID: [1 2 3 4]
```

Set the values for the ID feature dynamic property with less values than contained in X or Y. The ID property values expand to match the length of X and Y.

```
p.ID = 1:2

p =

 4x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [1 2 3 0]
              Y: [10 10 10 0]
    FeatureName: {'Feature 1' 'Feature 2' '' ''}
             ID: [1 2 0 0]
```

Set the values of either coordinate property (X or Y) with fewer values. All properties shrink in size to match the new length.

```
p.X = 1:2

p =

 2x1 mappoint vector with properties:
```

```
Collection properties:
      Geometry: 'point'
      Metadata: [1x1 struct]
Feature properties:
              X: [1 2]
              Y: [10 10]
    FeatureName: {'Feature 1' 'Feature 2'}
             ID: [1 2]
```

Remove the FeatureName (or ID) property by setting its value to [ ].

```
p.FeatureName = []

p =

 2x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
          X: [1 2]
          Y: [10 10]
         ID: [1 2]
```

Remove all dynamic properties and set the object to empty by setting a coordinate property value to [ ].

```
p.X = []

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
          X: []
          Y: []
```

### Construct a mappoint Specifying Two Features

```
point = mappoint([42 44], [10, 11], 'Temperature', [63 65])

point =

 2x1 mappoint vector with properties:
```

```
Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
               X: [42 44]
               Y: [10 11]
     Temperature: [63 65]
```

### Create a mappoint Vector from a MAT File

Load data from the seamount MAT-file and construct a mappoint vector to hold the coordinates.

```
seamount = load('seamount');
p = mappoint(seamount.x, seamount.y, 'Z', seamount.z);
```

Create a level list to use to bin the z values and create a list of color values for each level.

```
levels = [unique(floor(seamount.z/1000)) * 1000; 0];
colors = {'red', 'green', 'blue', 'cyan', 'black'};
```

Add a `MinLevel` and `MaxLevel` feature property to indicate the lowest and highest binned level. Add a dynamic feature property to indicate the Z coordinate. Add a dynamic feature property to indicate a binned level value and a color value for a given level. Include metadata information from the MAT-file.

```
for k = 1:length(levels) - 1
   index = levels(k) <= p.Z & p.Z < levels(k+1);
   p(index).MinLevel = levels(k);
   p(index).MaxLevel = levels(k+1) - 1;
   p(index).Color = colors{k};
end
```

Add metadata information. `Metadata` is a scalar structure containing information for the entire set of properties. Any type of data may be added to the structure.

```
p.Metadata.Caption = seamount.caption;
p.Metadata

ans =

    Caption: [1x229 char]
```

Display the point data as a 2-D plot.

```
figure
minLevels = unique(p.MinLevel);
for k=1:length(minLevels)
  index = p.MinLevel == minLevels(k);
  mapshow(p(index).X, p(index).Y, ...
    'MarkerEdgeColor', p(find(index,1)).Color, ...
    'Marker', 'o', ...
    'DisplayType', 'point')
end
legend(num2str(minLevels'))
```
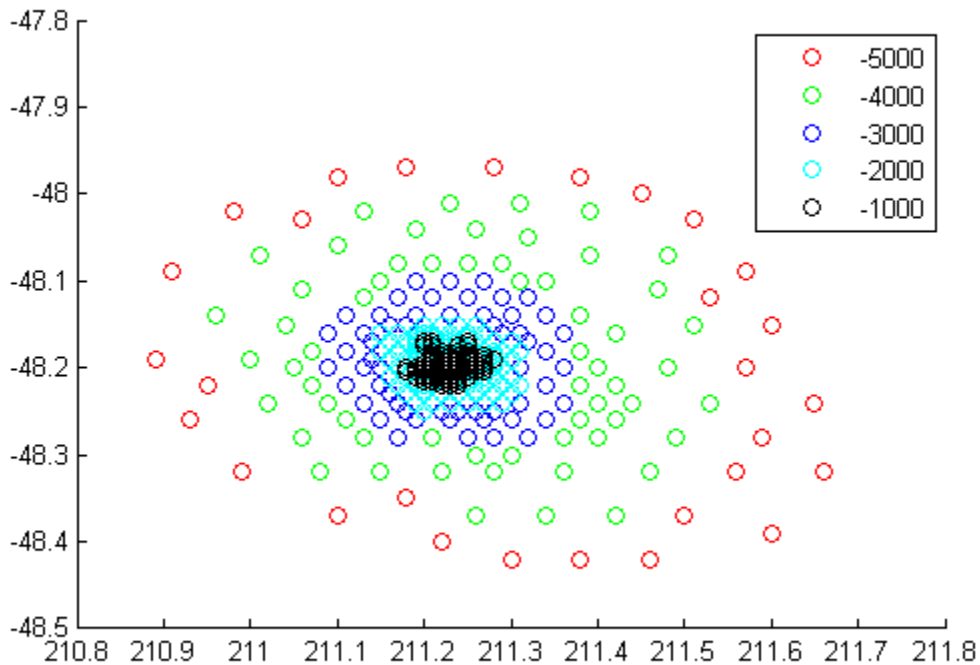


Display the point data as a 3-D scatter plot.

```
figure
```

```
scatter3(p.X, p.Y, p.Z)
```
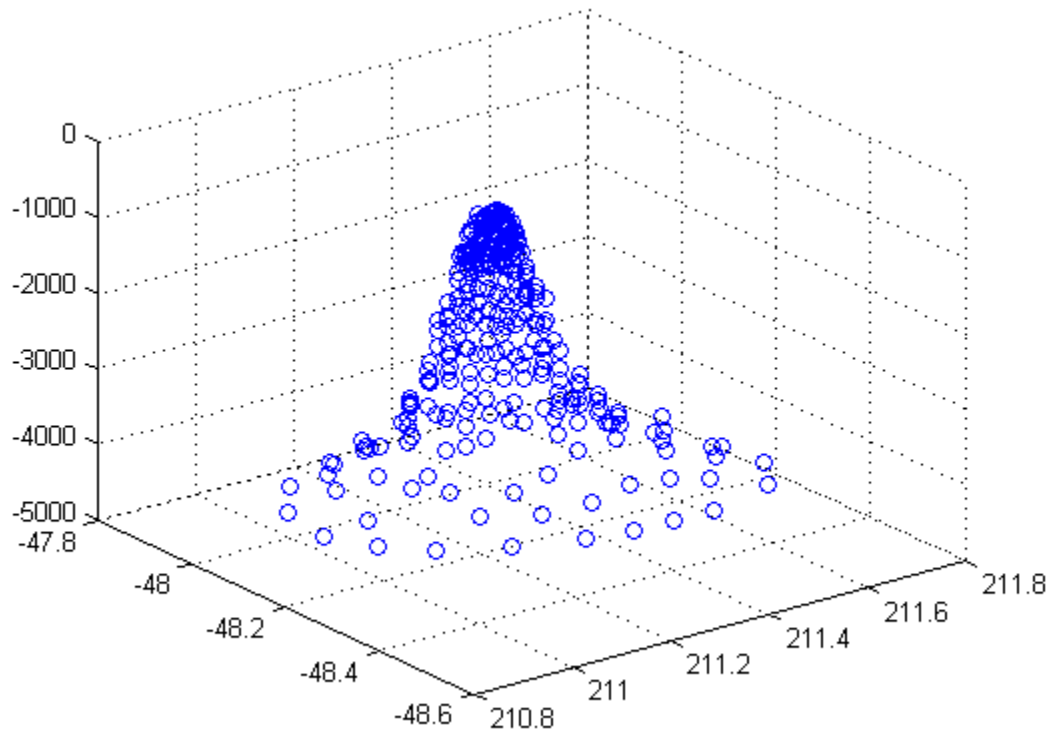


### Assign Dynamic Features to mappoint Vector from a Structure Array

```
structArray = shaperead('boston_placenames');
p = mappoint();
p.X = [structArray.X];
p.Y = [structArray.Y];
p.Name = {structArray.NAME}

p =

  13x1 mappoint vector with properties:
```

```
Collection properties:
   Geometry: 'point'
   Metadata: [1x1 struct]
Feature properties:
         X: [1x13 double]
         Y: [1x13 double]
      Name: {1x13 cell}
```

Construct a mappoint vector from a structure array using the constructor syntax.

```
filename = 'boston_placenames.shp';
structArray = shaperead(filename);
p = mappoint(structArray)

p =

 13x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
          X: [1x13 double]
          Y: [1x13 double]
       NAME: {1x13 cell}
    FEATURE: {1x13 cell}
      COORD: {1x13 cell}
```

Add a Filename field to the Metadata structure. Display the first five points and the Metadata structure.

```
p.Metadata.Filename = filename;
p(1:5)
p.Metadata

ans =

 5x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
          X: [2.3403e+05 2.3357e+05 2.3574e+05 2.3627e+05 2.3574e+05]
          Y: [900038 9.0019e+05 9.0113e+05 9.0097e+05 9.0036e+05]
```

```
      NAME: {1x5 cell}
   FEATURE: {'PPL-SUBDVSN' ' MARSH' ' HILL' ' PPL' ' PENINSULA'}
     COORD: {1x5 cell}

ans =

   Filename: 'boston_placenames.shp'
```

**Append a Point by Indexing**

Append Paderborn Germany to the vector of world cities.

```
p = mappoint(shaperead('worldcities.shp'));
x = 51.715254;
y = 8.75213;
p = append(p, x, y, 'Name', 'Paderborn');
p(end)

ans =

 1x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: 51.7153
           Y: 8.7521
        Name: 'Paderborn'
```

You can also add a point to the end of the vector using linear indexing. Add Arlington, Virginia to the end of the vector.

```
p(end+1).X = 38.880043;
p(end).Y = -77.196676;
p(end).Name = 'Arlington';
p(end-1:end)

ans =

 2x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
```

```
  Feature properties:
            X: [51.7153 38.8800]
            Y: [8.7521 -77.1967]
         Name: {'Paderborn' 'Arlington'}

% Plot the points
figure
mapshow(p.X, p.Y, 'DisplayType', 'point')
```
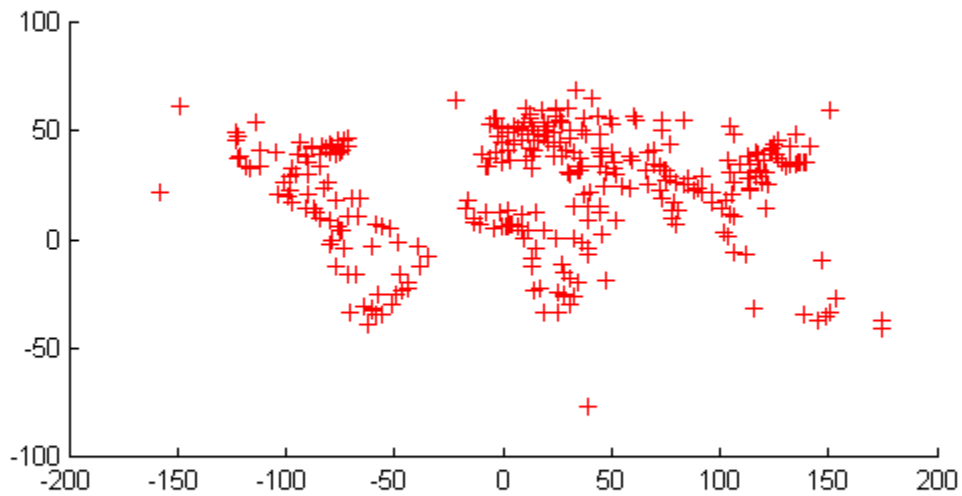


### Sort Dynamic Properties

Construct a mappoint vector and sort the dynamic properties.

```
p = mappoint(shaperead('tsunamis'));
```

```
p = p(:, sort(fieldnames(p)))

p =

 162x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [1x162 double]
              Y: [1x162 double]
          Cause: {1x162 cell}
     Cause_Code: [1x162 double]
        Country: {1x162 cell}
            Day: [1x162 double]
    Desc_Deaths: [1x162 double]
         Eq_Mag: [1x162 double]
           Hour: [1x162 double]
       Iida_Mag: [1x162 double]
      Intensity: [1x162 double]
       Location: {1x162 cell}
     Max_Height: [1x162 double]
         Minute: [1x162 double]
          Month: [1x162 double]
     Num_Deaths: [1x162 double]
         Second: [1x162 double]
       Val_Code: [1x162 double]
       Validity: {1x162 cell}
           Year: [1x162 double]
```

Modify the mappoint vector to contain only the dynamic properties, 'Year', 'Month', 'Day', 'Hour', 'Minute'.

```
p = p(:, {'Year', 'Month', 'Day', 'Hour', 'Minute'})

p =

 162x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1x162 double]
```

```
         Y: [1x162 double]
      Year: [1x162 double]
     Month: [1x162 double]
       Day: [1x162 double]
      Hour: [1x162 double]
    Minute: [1x162 double]
```

Display the first 5 elements.

```
p(1:5)

ans =

 5x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
         X: [128.3000 -156 157.9500 143.8500 -155]
         Y: [-3.8000 19.5000 -9.0200 42.1500 19.1000]
      Year: [1950 1951 1951 1952 1952]
     Month: [10 8 12 3 3]
       Day: [8 21 22 4 17]
      Hour: [3 10 NaN 1 3]
    Minute: [23 57 NaN 22 58]
```

**Row and Column Input Arguments**

Input argument x and y values can be either Nx2 or 2xM arrays.

If you typically store x and y coordinate values in a N-by-2 or 2-by-M array, you can assign a mappoint object to these numeric values. If the values are stored in a N-by-2 array, then the X property values are assigned to the first column and the Y property values are assigned to the second column.

```
x = 1:10;
y = 21:30;
pts = [x' y'];
p = mappoint;
p(1:length(pts)) = pts

p =

 10x1 mappoint vector with properties:
```

```
Collection properties:
   Geometry: 'point'
   Metadata: [1x1 struct]
Feature properties:
         X: [1 2 3 4 5 6 7 8 9 10]
         Y: [21 22 23 24 25 26 27 28 29 30]
```

If the values are stored in a 2-by-M array, then the X property values are assigned to the first row and the Y property values are assigned to the second row. pts = [x; y];

```
pts = [x; y];
p(1:length(pts)) = pts

p =

 10x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
          X: [1 2 3 4 5 6 7 8 9 10]
          Y: [21 22 23 24 25 26 27 28 29 30]
```

## See Also

geopoint | geoshape | mapshape | gpxread | shaperead

# append

**Class:** mappoint

Append features to mappoint vector

## Syntax

```
outPoints = append(inPoints,x,y)
outPoints = append( ___ ,field,value,...)
```

## Description

`outPoints = append(inPoints,x,y)` appends the vector, `x` to the `X` property of the mappoint vector, `inPoints`, and the vector, `y`, to the `Y` property of `inPoints`.

`outPoints = append( ___ ,field,value,...)` appends the values specified in `value` to the corresponding dynamic property, `field`. If the property does not exist, `append` adds the dynamic property to the object using the value of `field` for the name and assigning the field the value specified in `value`.

## Input Arguments

**`inPoints`**

mappoint vector.

**`x`**

Numeric vector of `X` values, of class `single` or `double`.

**`y`**

Numeric vector of `Y` values, of class `single` or `double`.

**`field`**

Field name, specified as a string. Field can specify the name of an existing property in the mappoint vector or the name you want assigned to a new property that you want to add to the mappoint vector.

**value**

Value you want to assign to the property specified by `field`, specified as a cell array, scalar, or vector of any numeric class or logical.

- When `value` is a cell array, `append` adds the property as a Vertex property.
- When `value` is a numeric array, `append` adds the property as a Feature property.
- When `value` is empty, `append` removes the property.

# Output Arguments

**outPoints**

Modified mappoint vector with additional entries in X and Y fields along with any new fields for dynamic properties that you added.

# Examples

### Append Values to Fields

Append values to existing fields of a mappoint vector.

```
p = mappoint(42,-110, 'Temperature', 65);
p = append(p, 42.1, -110.4, 'Temperature', 65.5)

p =

 2x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [42 42.1000]
              Y: [-110 -110.4000]
    Temperature: [65 65.5000]
```

### Append Dynamic Property

Create mappoint vector and append dynamic property to it.

```
p = mappoint(42,-110, 'Temperature', 65);
p = append(p, 42.2, -110.5, 'Temperature', 65.6, 'Pressure', 100.0)

p =

 2x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [42 42.2000]
              Y: [-110 -110.5000]
    Temperature: [65 65.6000]
       Pressure: [0 100]
```

## See Also

mappoint.vertcat

# cat

**Class:** mappoint

Concatenate mappoint vectors

# Syntax

```
p= cat(dim,p1, p2, ...)
```

# Description

p= cat(dim,p1, p2, ...) concatenates the mappoint vectors p1,p2 and so on along dimensions dim. dim must be 1.

# Input Arguments

**p1, p2, ...**

mappoint vectors to be concatenated.

# Output Arguments

**p**

Concatenated mappoint vector.

# Examples

### Concatenate two mappoint vectors

Create two mappoint vectors and concatenate them to a single vector.

```
pt1 = mappoint(42,-110, 'Temperature', 65);
```

```
pt2 = mappoint(42.2, -110.5, 'Temperature', 65.6);
p = cat(1,pt1,pt2)

p =

 2x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [42 42.2000]
              Y: [-110 -110.5000]
    Temperature: [65 65.6000]
```

## See Also

mappoint.vertcat

# disp

**Class:** mappoint

Display mappoint vector

## Syntax

```
disp(p)
```

## Description

`disp(p)` prints the size of the mappoint vector, `p`, and its properties and dynamic properties, if they exist. If the command window is large enough, the values of the properties are also shown, otherwise only their size is shown. You can control the display of the numerical values by using the format command.

## Input Arguments

**p**

mappoint vector.

## Examples

### Display a mappoint vector

Create mappoint vector and display it.

```
p = mappoint(shaperead('worldcities'));
disp(p)
disp(p(1:2))
```

```
  318x1 mappoint vector with properties:

 Collection properties:
```

```
      Geometry: 'point'
      Metadata: [1x1 struct]
Feature properties:
          X: [1x318 double]
          Y: [1x318 double]
       Name: {1x318 cell}

2x1 mappoint vector with properties:

Collection properties:
   Geometry: 'point'
   Metadata: [1x1 struct]
Feature properties:
          X: [-3.9509 54.7589]
          Y: [5.2985 24.6525]
       Name: {'Abidjan' 'Abu Dhabi'}
```

## See Also

mappoint | format

# fieldnames

**Class:** mappoint

Dynamic properties of mappoint vector

## Syntax

```
names = fieldnames(p)
```

## Description

`names = fieldnames(p)` returns the names of the dynamic properties of the mappoint vector, `p`.

## Input Arguments

**p**

mappoint vector for which the properties are to be queried.

## Output Arguments

**names**

Names of the dynamic properties in the mappoint vector `p`

## Examples

### Find dynamic properties

Create mappoint vector and view its properties.

```
p = mappoint(shaperead('worldcities'))
```

```
fieldnames(p)

p =

 318x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1x318 double]
           Y: [1x318 double]
        Name: {1x318 cell}

ans =

    'Name'
```

## See Also

mappoint.properties

# isempty

**Class:** mappoint

True if `mappoint` vector is empty

## Syntax

```
tf = isempty(p)
```

## Description

`tf = isempty(p)` returns true if the mappoint vector, `p`, is empty and false otherwise.

## Input Arguments

**p**

mappoint vector.

## Examples

### Check if a mappoint vector is empty

Create mappoint vector and check if it's empty.

```
p = mappoint();
isempty(p)

ans =

     1
```

# isfield

**Class:** mappoint

Returns true if dynamic property exists

## Syntax

```
tf = isfield(p,name)
tf = isfield(p,names)
```

## Description

`tf = isfield(p,name)` returns true if the value specified by the string `name` is a dynamic property of the mappoint vector, `p`.

`tf = isfield(p,names)` return true for each element of the cell array, `names`, that is a dynamic property of `p`. `tf` is a logical array of the same size as `names`.

## Input Arguments

**p**

mappoint vector.

**name**

Name of the dynamic property.

**names**

Cell array of names of dynamic properties.

## Output Arguments

**tf**

Boolean. 1 if `p` contains the specified fields or 0 otherwise.

# Examples

### Check for fieldname

Create mappoint vector and check is a particular fieldname exists in the vector.

```
p = mappoint(-33.961, 18.484, 'Name', 'Cape Town');
isfield(p, 'X')
isfield(p, 'Name')

ans =

     0


ans =

     1
```

## See Also

mappoint.isprop | mappoint.fieldnames

# isprop

**Class:** mappoint

Returns true if property exists

## Syntax

```
tf = isprop(p,name)
tf = isprop(p,names)
```

## Description

`tf = isprop(p,name)` returns true if the value specified by the string, `name` is a property of the mappoint vector, `p`.

`tf = isprop(p,names)` returns true for each element of the cell array of strings, `names`, that is a property of `p`. `tf` is a logical array the same size as `names`.

## Input Arguments

**p**

mappoint vector.

**name**

String specifying the property of the mappoint vector, `p`.

**names**

Cell array of strings specifying the property of the mappoint vector, `p`.

## Output Arguments

**tf**

Boolean. 1 if the property exists with `p` ,0 otherwise.

# Examples

### Check if property exists

Create a mappoint vector and check if certain properties exist.

```
p = mappoint(-33.961, 18.484, 'Name', 'Cape Town');
isprop(p, 'X')
isprop(p, 'Name')

ans =

     1


ans =

     1
```

## See Also

mappoint.isfield | mappoint.properties

# length

**Class:** mappoint

Number of elements in mappoint vector

## Syntax

```
N = length(p)
```

## Description

`N = length(p)` returns the number of elements contained in the mappoint vector, `p`. The result is equivalent to `size(p,1)`.

## Input Arguments

**p**

mappoint vector.

## Output Arguments

**N**

Length of the mappoint vector, p.

## Examples

### Create a mappoint vector and Get Its Length

Create mappoint vector and get its length.

```
coast = load('coast');
```

```
p = mappoint(coast.lat, coast.long);
length(p)
length(coast.lat)
```

```
ans =

       9865
```

```
ans =

       9865
```

## See Also
mappoint.size

# properties

**Class:** mappoint

properties of a mappoint vector

## Syntax

```
prop = properties(p)
properties(p)
```

## Description

`prop = properties(p)` returns a cell of the property names of the mappoint vector, `p`.

`properties(p)` displays the names of the properties of `p`.

## Input Arguments

**p**

mappoint vector.

## Output Arguments

**prop**

Cell variable consisting of property names of the mappoint vector, `p`.

## Examples

### properties of a mappoint vector

Create a mappoint vector and display its properties.

```
p = mappoint(shaperead('tsunamis'));
properties(p)

Properties for class mappoint:

    Geometry
    Metadata
    X
    Y
    Year
    Month
    Day
    Hour
    Minute
    Second
    Val_Code
    Validity
    Cause_Code
    Cause
    Eq_Mag
    Country
    Location
    Max_Height
    Iida_Mag
    Intensity
    Num_Deaths
    Desc_Deaths
```

## See Also
mappoint.fieldnames

# rmfield

**Class:** mappoint

Remove dynamic property from mappoint vector

## Syntax

```
p = rmfield(p, fieldname)
p = rmfield(p, fields)
```

## Description

`p = rmfield(p, fieldname)` removes the field specified by the string, `fieldname`, from the mappoint vector, `p`.

`p = rmfield(p, fields)` removes all the fields specified by the cell array, `fields`.

---

**Note:** `rmfield` cannot remove X, Y and `Metadata` fields and the string specified is case sensitive.

---

## Input Arguments

**p**

mappoint vector.

**fieldname**

Exact string representing the name of the property.

**fields**

Cell array of strings specifying the names of the properties.

# Output Arguments

**p**

Updated mappoint vector with the field(s) removed.

# Examples

### Remove fields from a mappoint vector

Create a mappoint vector and delete a field from it.

```
p = mappoint(shaperead('tsunamis'));
p2 = rmfield(p, 'Geometry')

p2 =

 162x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [1x162 double]
              Y: [1x162 double]
           Year: [1x162 double]
          Month: [1x162 double]
            Day: [1x162 double]
           Hour: [1x162 double]
         Minute: [1x162 double]
         Second: [1x162 double]
       Val_Code: [1x162 double]
       Validity: {1x162 cell}
     Cause_Code: [1x162 double]
          Cause: {1x162 cell}
         Eq_Mag: [1x162 double]
        Country: {1x162 cell}
       Location: {1x162 cell}
     Max_Height: [1x162 double]
       Iida_Mag: [1x162 double]
      Intensity: [1x162 double]
     Num_Deaths: [1x162 double]
```

```
Desc_Deaths: [1x162 double]
```

## See Also

mappoint.fieldnames | mappoint.rmprop

# rmprop

**Class:** mappoint

Remove properties from mappoint vector

## Syntax

```
pF = rmprop(p,propname)
pF = rmprop(p,propnames)
```

## Description

`pF = rmprop(p,propname)` removes the property specified by the string, `propname` from the mappoint vector, `p`.

`pF = rmprop(p,propnames)` removes all the properties specified in the cell array, `propnames`, from the mappoint vector, `p`. If `propnames` contains a coordinate property an error is issued.

---

**Note:** `rmprop` cannot remove `X`, `Y` and `Metadata` fields and the string specified is case sensitive.

---

## Input Arguments

**p**

mappoint vector.

## Output Arguments

**pF**

Modified mappoint vector with the specified property(s) removed.

## Examples

**Remove a property of a mappoint vector**

Create a mappoint vector and remove a property from the vector.

```
p = mappoint(shaperead('tsunamis'));
p2 = rmprop(p, 'Validity')

p2 =

 162x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
             X: [1x162 double]
             Y: [1x162 double]
          Year: [1x162 double]
         Month: [1x162 double]
           Day: [1x162 double]
          Hour: [1x162 double]
        Minute: [1x162 double]
        Second: [1x162 double]
      Val_Code: [1x162 double]
    Cause_Code: [1x162 double]
         Cause: {1x162 cell}
        Eq_Mag: [1x162 double]
       Country: {1x162 cell}
      Location: {1x162 cell}
    Max_Height: [1x162 double]
      Iida_Mag: [1x162 double]
     Intensity: [1x162 double]
    Num_Deaths: [1x162 double]
   Desc_Deaths: [1x162 double]
```

## See Also
mappoint.fieldnames

# size

**Class:** mappoint

Size of `mappoint` vector

## Syntax

```
SZ = size(p)
SZ = size(p,1)
SZ = size(p, n)
[m,k] = size(p)
```

## Description

`SZ = size(p)` returns the vector `[length(p), 1]`.

`SZ = size(p,1)` returns the length of `p`.

`SZ = size(p, n)` returns 1 for `n >= 2`.

`[m,k] = size(p)` returns `length(p)` for `m` and 1 for `k`.

## Input Arguments

**p**

mappoint vector.

**n**

Number of the dimension at which size of `p` is required.

## Output Arguments

**sz**

Vector of the form `[length(p), 1]`.

**m**

Length of `p`.

**k**

Length of second dimension of `p`. `k` is always `1`.

# Examples

### Size of a mappoint vector

Create a mappoint vector and get its size.

```
coast = load('coast');
p = mappoint(coast.lat, coast.long);
size(p)

ans =

       9865             1
```

The second dimension is always 1.

## See Also

mappoint.length | `size`

# struct

**Class:** mappoint

Convert mappoint vector to scalar structure

## Syntax

```
S = struct(p)
```

## Description

`S = struct(p)` converts the mappoint vector, `p`, to a scalar structure, `S`.

## Input Arguments

**p**

mappoint vector.

## Output Arguments

**S**

Scalar structure of the mappoint vector `p`.

## Examples

### Converting a mappoint vector into struct

Create a mappoint vactor and convert it into a structure.

```
S = shaperead('worldcities');
p = mappoint(S)
```

```
S2 = struct(p)
class(S2)

p =

 318x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1x318 double]
           Y: [1x318 double]
        Name: {1x318 cell}


S2 =

    Geometry: 'point'
    Metadata: [1x1 struct]
           X: [1x318 double]
           Y: [1x318 double]
        Name: {1x318 cell}


ans =

struct
```

## See Also

mappoint.properties

# vertcat

**Class:** geopoint

Vertical concatenation for mappoint vectors

## Syntax

```
p = vertcat(p1,p2, ...)
```

## Description

`p = vertcat(p1,p2, ...)` vertically concatenates the mappoint vector, `p1`, `p2`, and so on. If the class type of any property is a cell array, then the resultant field in the output `p` will also be a cell array.

## Input Arguments

**p1, p2, ...**

mappoint vectors that need to be concatenated.

## Output Arguments

**p**

Concatenated mappoint vector.

## Examples

**Concatenate mappoint vectors**

Create two mappoint vectors and concatenate them.

```
pt1 = mappoint(42,   -110,   'Temperature', 65,   'Name', 'point1');
pt2 = mappoint(42.1, -110.4, 'Temperature', 65.5, 'Name', 'point2');
pts = vertcat(pt1, pt2)

pts =

 2x1 mappoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1x1 struct]
 Feature properties:
              X: [42 42.1000]
              Y: [-110 -110.4000]
    Temperature: [65 65.5000]
           Name: {'point1' 'point2'}
```

## See Also

mappoint.cat

# mapprofile

Interpolate between waypoints on regular data grid

## Syntax

```
[zi,ri,lat,lon] = mapprofile
[zi,ri,lat,lon] = mapprofile(Z,R,lat,lon)
[zi,ri,lat,lon] = mapprofile(Z,R,lat,lon,units)
[zi,ri,lat,lon] = mapprofile(Z,R,lat,lon,ellipsoid)
[zi,ri,lat,lon] = ...
mapprofile(Z,R,lat,lon,units,'trackmethod','interpmethod')
[zi,ri,lat,lon] = ...
mapprofile(Z,R,lat,lon,ellipsoid,'trackmethod','interpmethod')
```

## Description

`mapprofile` plots a profile of values between waypoints on a displayed regular data grid. `mapprofile` uses the current object if it is a regular data grid, or the first regular data grid found on the current axes. The grid's `zdata` is used for the profile. The color data is used in the absence of `zdata`. The result is displayed in a new figure.

`[zi,ri,lat,lon] = mapprofile` returns the values of the profile without displaying them. The output `zi` contains interpolated values along great circles between the waypoints. `ri` is a vector of associated distances from the first waypoint in units of degrees of arc along the surface. `lat` and `lon` are the corresponding latitudes and longitudes.

`[zi,ri,lat,lon] = mapprofile(Z,R,lat,lon)` accepts as input a regular data grid and waypoint vectors. No displayed grid is required. Sets of waypoints may be separated by NaNs into line sequences. The output ranges are measured from the first waypoint within a sequence. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel.

`[zi,ri,lat,lon] = mapprofile(Z,R,lat,lon,units)` specifies the units of the output ranges along the profile. Valid range units inputs are any distance string recognized by `unitsratio`. Surface distances are computed using the default radius of the earth. If omitted, `'degrees'` are assumed.

`[zi,ri,lat,lon] = mapprofile(Z,R,lat,lon,ellipsoid)` uses the provided ellipsoid definition in computing the range along the profile. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The output range is reported in the same distance units as the semimajor axes of the ellipsoid. If omitted, the range vector is for a sphere.

```
[zi,ri,lat,lon] = ...
mapprofile(Z,R,lat,lon,units,'trackmethod','interpmethod')
```
and
```
[zi,ri,lat,lon] = ...
mapprofile(Z,R,lat,lon,ellipsoid,'trackmethod','interpmethod')
```
control the interpolation methods used. Valid track methods are `'gc'` for great circle tracks between waypoints, and `'rh'` for rhumb lines. Valid methods for interpolation within the matrix are `'bilinear'` for linear interpolation, `'bicubic'` for cubic interpolation, and `'nearest'` for nearest neighbor interpolation. If omitted, `'gc'` and `'bilinear'` are assumed.

## Examples

### Example 1

Create a map axes for the Korean peninsula. Specify an elevation profile across the sample Korean digital elevation data and plot it, combined with a coastline and city markers:

```
load korea
h = worldmap(map, refvec); % The figure has no map content.
plat = [ 43  43  41  38];
plon = [116 120 126 128];
mapprofile(map, refvec, plat, plon)
load coast
plotm(lat, long)
geoshow('worldcities.shp', 'Marker', '.', 'Color', 'red')
```
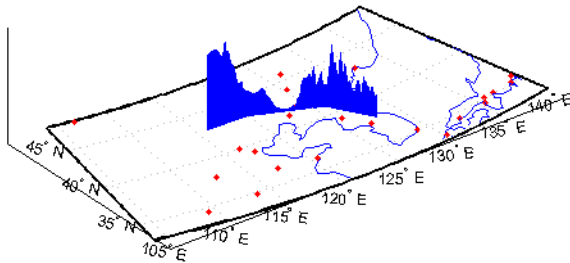


When you select more than two waypoints, the automatically generated figure displays the result in three dimensions. The following example shows the relative sizes of the mountains in northern China compared to the depths of the Sea of Japan. The call to mapprofile without input arguments requires you to interactively pick waypoints on the figure using the mouse, and press **Enter** after you select the final point:

```
axes(h);
meshm(map, refvec, size(map))
demcmap(map)
[zi,ri,lat,lon] = mapprofile
```

Adding output arguments suppresses the display of the results in a new figure. You can then use the results in further calculations or display the results yourself. Here the profile from the upper left to lower right is computed from waypoints interactively picked on the map (your profile will not be identical to what is shown below). The example converts ranges and elevations to kilometers and displays them in a new figure, setting the vertical exaggeration factor to 20. With no vertical exaggeration, the changes in elevation would be almost too small to see.

**1-845**
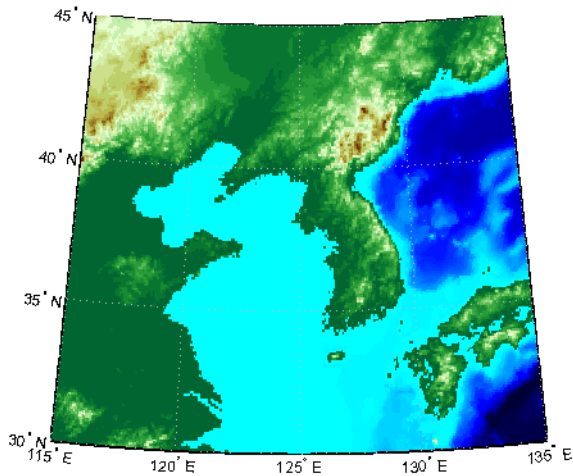
```
figure
plot(deg2km(ri),zi/1000)
daspect([ 1 1/20 1 ]);
xlabel 'Range (km)'
ylabel 'Elevation (km)'
```
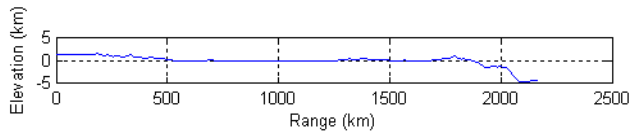


Naturally, the profile you get depends on the transect locations you pick.

## Example 2

You can compute values along a path without reference to an existing figure by providing a regular data grid and vectors of waypoint coordinates. Optional arguments allow control over the units of the range output and interpolation methods between waypoints and data grid elements.

Show what land and ocean areas lie under a great circle track from Frankfurt to Seattle:

```
cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
Seattle = strcmp('Seattle', {cities(:).Name});
```

```
Frankfurt = strcmp('Frankfurt', {cities(:).Name});
lat = [cities(Seattle).Lat cities(Frankfurt).Lat]
lon = [cities(Seattle).Lon cities(Frankfurt).Lon]
load topo
[valp,rp,latp,lonp] = ...
   mapprofile(double(topo),topolegend, ...
   lat,lon,'km','gc','nearest');
figure
worldmap([40 80],[-135 20])
land = shaperead('landareas.shp', 'UseGeoCoords', true);
faceColors = makesymbolspec('Polygon',...
   {'INDEX', [1 numel(land)], 'FaceColor', ...
   polcmap(numel(land))});
geoshow(land,'SymbolSpec',faceColors)
plotm(latp,lonp,'r')
plotm(lat,lon,'ro')
axis off
```
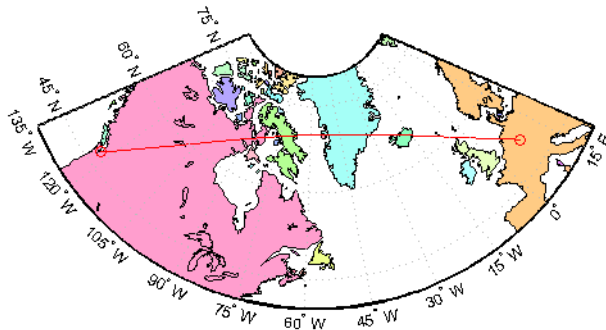


## See Also
```
ltln2val | los2
```

# maprasterref

Construct map raster reference object

## Compatibility

Use the `maprefcells` function or the `maprefpostings` function instead, except when constructing a raster reference object from world file input.

## Syntax

```
R = maprasterref()
R = maprasterref(Name,Value)
R = maprasterref(W,rasterSize,rasterInterpretation)
```

## Description

`R = maprasterref()` constructs a map raster reference object with default property values.

`R = maprasterref(Name,Value)` accepts a list of name-value pairs that are used to assign selected properties when initializing a map raster reference object.

`R = maprasterref(W,rasterSize,rasterInterpretation)` constructs a map raster reference object with the specified raster size and interpretation properties, and with remaining properties defined by a 2-by-3 world file matrix, `W`.

## Input Arguments

**W**

2-by-3 world file matrix

**rasterSize**

Number of cells or samples in each spatial dimension. Two-element vector [M N] specifying the number of rows (*M*) and columns (*N*) of the raster or image associated with

the referencing object. For convenience, you may assign a size vector having more than two elements to `RasterSize`. This flexibility enables assignments like `R.RasterSize = size(RGB)`, for example, where `RGB` is *M*-by-*N*-by-3. However, in such cases, only the first two elements of the size vector are actually stored. The higher (non-spatial) dimensions are ignored

**`rasterInterpretation`**

Controls handling of raster edges. This argument is optional and can equal either `'cells'` or `'postings'`.

**Default:** `'cells'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

You can include any of the following properties, overriding their default values as needed. Alternatively, you may omit any or all properties when constructing your map raster reference object. Then, you can customize the result by resetting properties from this list one at a time. This name-value syntax always results in an object with a `'rectilinear'` `TransformationType`. If your image is rotated with respect to the world coordinate axes, you need an object with a `TransformationType` of `'affine'`. Alternately, you can provide an appropriate world file matrix as input, as shown in the third syntax. You cannot do it by resetting properties of an existing rectilinear map raster reference object.

**`'XLimWorld'`**

Limits of raster in world *x*

Two-element row vector of the form `[xMin xMax]`.

**Default:** [0.5 2.5]

**`'YLimWorld'`**

Limits of raster in world *y*

Two-element row vector of the form `[yMin yMax]`.

**Default:** [0.5 2.5]

**'RasterSize'**

Two-element vector [M N] specifying the number of rows (*M*) and columns (*N*) of the raster or image associated with the referencing object. For convenience, you may assign a size vector having more than two elements to RasterSize. This flexibility enables assignments like R.RasterSize = size(RGB), for example, where RGB is *M*-by-*N*-by-3. However, in such cases, only the first two elements of the size vector are actually stored. The higher (non-spatial) dimensions are ignored.

**Default:** [2 2]

**'RasterInterpretation'**

Controls handling of raster edges. A string that equals either 'cells' or 'postings'.

**Default:** 'cells'

**'ColumnsStartFrom'**

Edge where column indexing starts. A string that equals either 'south' or 'north'.

**Default:** 'south'

**'RowsStartFrom'**

Edge from which row indexing starts. A string that equals either 'west' or 'east'.

**Default:** 'west'

## Output Arguments

**R**

Map raster reference object

## Examples

Construct a referencing object for an 1000-by-2000 image with square, half-meter pixels referenced to a planar map coordinate system (the "world" system). The *X*-limits in the

world system are 207000 and 208000. The *Y*-limits are 912500 and 913000. The image follows the popular convention in which world *X* increases from column to column and world *Y* decreases from row to row.

```
R = maprasterref('RasterSize', [1000 2000], ...
          'YWorldLimits', [912500 913000], 'ColumnsStartFrom','north', ...
          'XWorldLimits', [207000 208000])

R =

  MapCellsReference with properties:

             XWorldLimits: [207000 208000]
             YWorldLimits: [912500 913000]
               RasterSize: [1000 2000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1/2
      CellExtentInWorldY: -1/2
    RasterExtentInWorldX: 1000
    RasterExtentInWorldY: 500
         XIntrinsicLimits: [0.5 2000.5]
         YIntrinsicLimits: [0.5 1000.5]
       TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
```

Repeat the first example with a different strategy: Create a default object, and then modify that object's property settings as needed.

```
R = maprasterref;
R.XWorldLimits = [207000 208000];
R.YWorldLimits = [912500 913000];
R.ColumnsStartFrom = 'north';
R.RasterSize = [1000 2000]

R =

  MapCellsReference with properties:

             XWorldLimits: [207000 208000]
             YWorldLimits: [912500 913000]
               RasterSize: [1000 2000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
```

```
              RowsStartFrom: 'west'
          CellExtentInWorldX: 1/2
          CellExtentInWorldY: -1/2
        RasterExtentInWorldX: 1000
        RasterExtentInWorldY: 500
            XIntrinsicLimits: [0.5 2000.5]
            YIntrinsicLimits: [0.5 1000.5]
          TransformationType: 'rectilinear'
        CoordinateSystemType: 'planar'
```

Repeat the first example again, this time using a world file matrix.

```
W = [0.5   0.0   207000.25; ...
       0.0  -0.5   912999.75];
rasterSize = [1000 2000];
R = maprasterref(W, rasterSize)

R =

  MapCellsReference with properties:

            XWorldLimits: [207000 208000]
            YWorldLimits: [912500 913000]
              RasterSize: [1000 2000]
      RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
       CellExtentInWorldX: 1/2
       CellExtentInWorldY: -1/2
     RasterExtentInWorldX: 1000
     RasterExtentInWorldY: 500
         XIntrinsicLimits: [0.5 2000.5]
         YIntrinsicLimits: [0.5 1000.5]
       TransformationType: 'rectilinear'
     CoordinateSystemType: 'planar'
```

## See Also

map.rasterref.MapPostingsReference | map.rasterref.MapCellsReference |
georasterref | maprefcells | maprefpostings

# maprefcells

Reference raster cells to map coordinates

## Syntax

```
R = maprefcells()
R = maprefcells(xWorldLimits,yWorldLimits,rasterSize)
R = maprefcells(xWorldLimits,yWorldLimits,cellExtentInWorldX,
cellExtentInWorldY)
R = maprefcells(xWorldLimits,yWorldLimits, ___ ,Name,Value)
```

## Description

`R = maprefcells()` returns a default referencing object for a regular raster of cells in planar (map) coordinates.

`R = maprefcells(xWorldLimits,yWorldLimits,rasterSize)` constructs a referencing object for a raster of cells spanning the specified limits in planar coordinates, with the numbers of rows and columns specified by `rasterSize`.

`R = maprefcells(xWorldLimits,yWorldLimits,cellExtentInWorldX, cellExtentInWorldY)` allows the cell extents to be set precisely. If necessary, `maprefcells` adjusts the limits of the raster slightly to ensure an integer number of cells in each dimension.

`R = maprefcells(xWorldLimits,yWorldLimits, ___ ,Name,Value)` allows the directions of the columns and rows to be specified via name-value pairs.

## Examples

### Construct Referencing Object with Raster Interpration of Cells

Define latitude and longitude limits and dimensions of the image. The image follows the popular convention in which world *x* coordinates increase from column to column and world *y* coordinates decrease from row to row.

```
xWorldLimits = [207000 208000];
yWorldLimits = [912500 913000];
rasterSize = [1000 2000]


rasterSize =

        1000        2000
```

Create the referencing object specifying the raster size.

```
R = maprefcells(xWorldLimits,yWorldLimits,rasterSize,...
    'ColumnsStartFrom','north')


R =

  MapCellsReference with properties:

             XWorldLimits: [207000 208000]
             YWorldLimits: [912500 913000]
               RasterSize: [1000 2000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1/2
      CellExtentInWorldY: 1/2
     RasterExtentInWorldX: 1000
     RasterExtentInWorldY: 500
         XIntrinsicLimits: [0.5 2000.5]
         YIntrinsicLimits: [0.5 1000.5]
       TransformationType: 'rectilinear'
      CoordinateSystemType: 'planar'
```

Obtain the same result by specifying the cell extents. For this example, the pixels are 1/2 meter square, referenced to a planar map coordinate system (the "world" system).

```
extent = 1/2;

R = maprefcells(xWorldLimits,yWorldLimits,extent,extent,...
    'ColumnsStartFrom','north')
```

```
R =

  MapCellsReference with properties:

            XWorldLimits: [207000 208000]
            YWorldLimits: [912500 913000]
              RasterSize: [1000 2000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1/2
      CellExtentInWorldY: 1/2
    RasterExtentInWorldX: 1000
    RasterExtentInWorldY: 500
        XIntrinsicLimits: [0.5 2000.5]
        YIntrinsicLimits: [0.5 1000.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
```

## Input Arguments

### xWorldLimits — Limits in the x direction
[0.5 2.5] (default) | 1-by-2 numeric vector

Limits in the *x* direction, specified as a 1-by-2 numeric vector.

Example: xWorldLimits = [207000 208000];

Data Types: double

### yWorldLimits — Limits in the y direction
[0.5 2.5] (default) | 1-by-2 numeric vector

Limits in the *y* direction, specified as a 1-by-2 numeric vector.

Example: yWorldLimits = [912500 913000];

Data Types: double

### rasterSize — Size of the raster
[2 2] (default) | 1-by-2 numeric vector

Size of the raster, specified as a 1-by-2 numeric vector.

Example: `rasterSize = [180 360];`

Data Types: `double`

### `cellExtentInWorldX` — Height of cells
1 (default) | numeric scalar

Height of cells, specified as a numeric scalar.

Example: `extent = 1.5`

Data Types: `double`

### `cellExtentInWorldY` — Width of cells
1 (default) | numeric scalar

Width of cells, specified as a numeric scalar.

Example: `extent = 1.5`

Data Types: `double`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `R = maprefcells(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')`

### `'ColumnsStartFrom'` — Edge from which column indexing starts
`'south'` (default) | `'north'`

Edge from which column indexing starts, specified as the string `'north'` or `'south'`.

Example: `R = maprefcells(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')`

Data Types: `char`

### `'RowsStartFrom'` — Edge from which row indexing starts
`'west'` (default) | `'east'`

Edge from which column indexing starts, specified as the string `'north'` or `'south'`.

Example: `R = maprefcells(latlim,lonlim,rasterSize,'RowsStartFrom','east')`

Data Types: `char`

## Output Arguments

### R — Object that references raster cells to map coordinates
Raster reference object

Object that references raster cells to map coordinates, returned as a raster reference object (map.rasterref.MapCellsReference).

## More About

### Tips

*   To construct a map raster reference object from a world file matrix, use the `maprasterref` function.

## See Also

map.rasterref.MapCellsReference | `georefcells` | `maprefpostings`

### Introduced in R2015b

# map.rasterref.MapCellsReference class

**Package:** map.rasterref

Reference raster cells to map coordinates

## Description

A map cells reference object encapsulates the relationship between a planar map coordinate system and a system of intrinsic coordinates anchored to the columns and rows of a 2-D spatially referenced raster grid or image. Typically, the raster is sampled regularly in the planar world $x$ and world $y$ coordinates of the map system, such that the intrinsic $x$ and world $x$ axes align and the intrinsic $y$ and world $y$ axes align. When this is true, the relationship between the two systems is rectilinear. More generally, and much more rarely, their relationship is affine. The affine relationship allows for a possible rotation (and skew). In either case, rectilinear or affine, the sample spacing from row to row need not equal the sample spacing from column to column.The cells or pixels need not be square. In the most general case, they could conceivably be parallelograms, but in practice they are always rectangular. For more information about coordinate systems, see "Intrinsic Coordinate System" on page 1-863.

## Construction

You can use any of the following functions to construct a `map.rasterref.MapCellsReference` object to reference a regular raster of cells to planar (map) coordinates.

- `maprefcells`—Create a map raster reference object.
- `maprasterref`—Convert a world file to a map raster reference object.
- `refmatToMapRasterReference`—Convert a referencing matrix to a map raster reference object.

For example, this syntax constructs a `MapCellsReference` object with default property settings:

```
R = maprefcells()

R =
```

```
MapCellsReference with properties:

          XWorldLimits: [0.5 2.5]
          YWorldLimits: [0.5 2.5]
            RasterSize: [2 2]
  RasterInterpretation: 'cells'
     ColumnsStartFrom: 'south'
        RowsStartFrom: 'west'
    CellExtentInWorldX: 1
    CellExtentInWorldY: 1
  RasterExtentInWorldX: 2
  RasterExtentInWorldY: 2
      XIntrinsicLimits: [0.5 2.5]
      YIntrinsicLimits: [0.5 2.5]
    TransformationType: 'rectilinear'
  CoordinateSystemType: 'planar'
```

# Properties

### **XWorldLimits** — **Limits of raster in world _x_ coordinates**
[0.5 2.5] (default) | two-element row vector

Limits of raster in world _x_ coordinates, specified as a two-element row vector of the form [xMin xMax].

Example: [207000 209000]

Data Types: double

### **YWorldLimits** — **Limits of raster in world _y_ coordinates**
[0.5 2.5] (default) | two-element row vector

Limits of raster in world _y_ coordinates, specified as a two-element row vector of the form [yMin yMax].

Example: [911000 913000]

Data Types: double

### **RasterSize** — **Number of rows and columns of the raster or image associated with the referencing object**
[2 2] (default) | two-element vector of positive integers

Number of rows and columns of the raster or image associated with the referencing object, specified as a two-element vector, `[M N]`, where *M* represents the number of rows and *N* the number of columns. For convenience, you can assign a size vector having more than two elements. This enables assignments like `R.RasterSize = size(RGB)`, where `RGB` is *M*-by-*N*-by-3. In cases like this, the object stores only the first two elements of the size vector and ignores the higher (nonspatial) dimensions.

Example: `[200 300]`

Data Types: `double`

### `RasterInterpretation` — Geometric nature of the raster
`'cells'` (default)

Geometric nature of the raster, specified as the string `'cells'`. The value `'cells'` indicates that the raster comprises a grid of quadrangular cells, and is bounded on all sides by cell edges. For an *M*-by-*N* raster, points with an intrinsic *X*-coordinate of 1 or *N* or an intrinsic *Y*-coordinate of 1 or *M* fall within the raster, not on its edges.

Cannot be set.

Data Types: `char`

### `ColumnsStartFrom` — Edge from which column indexing starts
`'south'` (default) | `'north'`

Edge from which column indexing starts, specified as the string `'south'` or `'north'`.

Example: `'south'`

Data Types: `char`

### `RowsStartFrom` — Edge from which row indexing starts
`'west'` (default) | `'east'`

Edge from which row indexing starts, specified as the string `'west'` or `'east'`.

Example: `'east'`

Data Types: `char`

### `CellExtentInWorldX` — Extent in world *x* coordinates of individual cells
1 (default) | positive numeric scalar

Extent in world *x* of individual cells, specified as a positive numeric scalar. Distance between the eastern and western limits of a single raster cell. The value is the same for all cells in the raster.

Example: `2.5`

Data Types: `double`

### `CellExtentInWorldY` — Extent in world *y* coordinates of individual cells
1 (default) | positive numeric scalar

Extent in world *y* coordinates of individual cells, specified as a positive numeric scalar. Distance between the northern and southern limits of a single raster cell. The value is the same for all cells in the raster.

Example: `2.5`

Data Types: `double`

### `RasterExtentInWorldX` — Extent of the full raster or image as measured in the world system in a direction parallel to its rows
2 (default) | positive numeric scalar

Extent of the full raster or image as measured in the world system in a direction parallel to its rows, specified as a positive numeric scalar. In the case of a rectilinear geometry, which is most typical, this is the horizontal direction (east-west).

Cannot be set.

Data Types: `double`

### `RasterExtentInWorldY` — Extent of the full raster or image as measured in the world system in a direction parallel to its columns
2 (default) | positive numeric scalar

Extent of the full raster or image as measured in the world system in a direction parallel to its columns, specified as a positive numeric scalar. In the case of a rectilinear geometry, which is most typical, this is the vertical direction (north-south).

Cannot be set.

Data Types: `double`

### `XIntrinsicLimits` — Raster limits in intrinsic *x* coordinates
[0.5 2.5] (default) | two-element row vector of positive integers

Raster limits in intrinsic *x* coordinates, specified as a two-element row vector of positive integers, `[xMin xMax]`. For an M-by-N raster, `XIntrinsicLimits` equals `[0.5, M +0.5]`, because the `RasterInterpretation` is 'cells'.

Cannot be set.

Data Types: `double`

### `YIntrinsicLimits` — Raster limits in intrinsic *y* coordinates
`[0.5 2.5]` (default) | two-element row vector of positive integers

Raster limits in intrinsic *y* coordinates, specified as a two-element row vector of positive integers, `[yMin yMax]`. For an M-by-N raster, `YIntrinsicLimits` equals `[0.5, M +0.5]`, because the `RasterInterpretation` is 'cells'.

Cannot be set.

Data Types: `double`

### `TransformationType` — Type of geometric relationship between intrinsic and world systems
`'rectilinear'` (default) | `'affine'`

Type of geometric relationship between the intrinsic coordinate system and the world coordinate system, specified as either the string `'rectilinear'` or `'affine'`. Its value is `'rectilinear'` when world *x* depends only on intrinsic *x* and vice versa, and world *y* depends only on intrinsic *y* and vice versa. When the value is `'rectilinear'`, the image displays without rotation in the world system, although it might be flipped. Otherwise, the value is `'affine'`.

Cannot be set.

Data Types: `char`

### `CoordinateSystemType` — Type of coordinate system to which the image or raster is referenced
`'planar'` (default)

Type of coordinate system to which the image or raster is referenced, specified as the string `'planar'`.

Cannot be set.

Data Types: `char`

# Methods

| | |
|---|---|
| worldToIntrinsic | Convert from world to intrinsic coordinates |
| worldFileMatrix | World file parameters for transformation |
| sizesMatch | True if object and raster or image are size compatible |
| firstCornerX | World $x$ coordinate of the (1,1) corner of the raster |
| firstCornerY | World $y$ coordinate of the (1,1) corner of the raster |
| worldToDiscrete | Transform map to discrete coordinates |
| intrinsicToWorld | Convert from intrinsic to world coordinates |
| contains | True if raster contains points in world coordinate system |

# Definitions

## Intrinsic Coordinate System

A 2-D Cartesian system with its $x$-axis running parallel to the rows of a raster or image and its $y$-axis running parallel to the columns. $x$ increases by 1 from column to column, and $y$ increases by 1 from row to row.

The Mapping Toolbox and Image Processing Toolbox use the convention for the location of the origin relative to the raster cells or sampling points such that, at a sample location or at the center of a cell, $x$ has an integer value equal to the column index. Likewise, at a sample location or at the center of a cell, $y$ has an integer value equal to the row index. For details, see Image Coordinate Systems in the Image Processing Toolbox documentation.

## See Also

map.rasterref.GeographicCellsReference | map.rasterref.MapPostingsReference | maprasterref | maprefcells

## More About

- Class Attributes
- Property Attributes

**Introduced in R2013b**

# worldToIntrinsic

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

Convert from world to intrinsic coordinates

## Syntax

```
[xIntrinsic, yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)
```

## Description

`[xIntrinsic, yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)` maps point locations from the world system (`xWorld`, `yWorld`) to the intrinsic system (`xIntrinsic`, `yIntrinsic`) based on the relationship defined by the referencing object `R`. The input can include values outside limits of the raster (or image) in the world system. In this case, world *x* and *y* are extrapolated outside the bounds of the image in the intrinsic system.

# worldFileMatrix

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

World file parameters for transformation

## Syntax

```
W = worldFileMatrix(R)
```

## Description

`W = worldFileMatrix(R)` returns a 2-by-3 world file matrix. Each of the six elements in `W` matches one of the lines in a world file corresponding to the rectilinear or affine transformation defined by referencing object `R`.

Given `W` with the form:

```
W = [A B C;
     D E F]
```

a point (`xi, yi`) in intrinsic coordinates maps to a point (`xw, yw`) in planar world coordinates like this:

```
xw = A * (xi - 1) + B * (yi - 1) + C
yw = D * (xi - 1) + E * (yi - 1) + F
```

More compactly:

```
[xw yw]' = W * [xi - 1  yi - 1  1]'
```

The `-1`s are needed to maintain the Mapping Toolbox convention for intrinsic coordinates, which is consistent with the 1-based indexing used throughout MATLAB.

`W` is stored in a world file with one term per line in column-major order: A, D, B, E, C, F. That is, a world file contains the elements of `W` in this order:

```
W(1,1)
W(2,1)
```

```
W(1,2)
W(2,2)
W(1,3)
W(2,3)
```

The previous expressions hold for both affine and rectilinear transformations. But whenever `R.TransformationType` is `'rectilinear'`, B, D, `W(2,1)` and `W(1,2)` are identically 0.

## See Also

`worldfileread` | `worldfilewrite`

# sizesMatch

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

True if object and raster or image are size compatible

## Syntax

```
TF = sizesMatch(R,A)
```

## Description

TF = sizesMatch(R,A) returns true if the size of the raster or image A is consistent with the RasterSize property of referencing object R. That is:

```
R.RasterSize == [size(A,1) size(A,2)]
```

# firstCornerX

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

World *x* coordinate of the (1,1) corner of the raster

## Syntax

```
firstCornerX(R)
```

## Description

`firstCornerX(R)` returns the world *x* coordinate of the outermost corner of the first cell (1,1) of the raster associated with referencing object `R`.

# firstCornerY

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

World *y* coordinate of the (1,1) corner of the raster

## Syntax

```
firstCornerY(R)
```

## Description

`firstCornerY(R)` returns the world *y* coordinate of the outermost corner of the first cell (1,1) of the raster associated with referencing object `R`.

# worldToDiscrete

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

Transform map to discrete coordinates

## Syntax

```
[I,J] = worldToDiscrete(R,xWorld,yWorld)
```

## Description

`[I,J] = worldToDiscrete(R,xWorld,yWorld)` returns subscript arrays `I` and `J`.

When referencing object `R` has `RasterInterpretation` `'cells'`, `I` and `J` are the row and column subscripts of the raster cells (or image pixels) containing each element of a set of points given their world coordinates (`xWorld`, `yWorld`).

`xWorld` and `yWorld` must have the same size. `I` and `J` have the same size as `xWorld` and `yWorld`. For an $M$-by-$N$ raster, `1 <= I <= M` and `1 <= J <= N`, except when a point `xWorld(k)`, `yWorld(k)` falls outside the image, as defined by `contains(R,xWorld,yWorld)`. Then both `I(k)` and `J(k)` are NaN.

# intrinsicToWorld

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

Convert from intrinsic to world coordinates

## Syntax

```
[xWorld, yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)
```

## Description

`[xWorld, yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)` maps
point locations from the intrinsic system (`xIntrinsic, yIntrinsic`) to the world
system (`xWorld, yWorld`) based on the relationship defined by the referencing object
`R`. If your input includes values outside the limits of the raster or image in the intrinsic
system, `intrinsicToWorld` extrapolates world *x* and *y* outside the bounds of the image
in the world system.

# contains

**Class:** map.rasterref.MapCellsReference
**Package:** map.rasterref

True if raster contains points in world coordinate system

## Syntax

```
TF = contains(R,xWorld,yWorld)
```

## Description

`TF = contains(R,xWorld,yWorld)` returns a logical array `TF` having the same size as `xWorld` and `yWorld` such that `TF(k)` is `true` if and only if the point (`xWorld(k)`, `yWorld(k)`) falls within the bounds of the raster associated with referencing object `R`.

# maprefpostings

Reference raster postings to map coordinates

## Syntax

```
R = maprefpostings()
R = maprefpostings(xWorldLimits,yWorldLimits,rasterSize)
R = maprefpostings(xWorldLimits,yWorldLimits,sampleSpacingInWorldX,
sampleSpacingInWorldY)
R = maprefpostings(xWorldLimits,yWorldLimits, ___ , Name,Value)
```

## Description

`R = maprefpostings()` returns a default referencing object for a raster of regularly posted samples in planar (map) coordinates.

`R = maprefpostings(xWorldLimits,yWorldLimits,rasterSize)` constructs a referencing object for a raster spanning the specified limits in planar coordinates, with the numbers of rows and columns specified by `rasterSize`.

`R = maprefpostings(xWorldLimits,yWorldLimits,sampleSpacingInWorldX, sampleSpacingInWorldY)` allows the sample spacings to be set precisely. If necessary, `maprefpostings` adjusts the limits of the raster slightly to ensure an integer number of samples in each dimension.

`R = maprefpostings(xWorldLimits,yWorldLimits, ___ , Name,Value)` allows the directions of the columns and rows to be specified via name-value pairs.

## Examples

### Construct Referencing Object for Grid

Define latitude and longitude limits and dimension of a grid. The example uses postings separated by 1/2 meter, referenced to a planar map coordinate system (the "world" system).

```
xWorldLimits = [207000 208000];
yWorldLimits = [912500 913000];
rasterSize = [1001 2001]


rasterSize =

        1001        2001
```

Create the referencing object specifying the raster size.

```
R = maprefpostings(xWorldLimits,yWorldLimits,rasterSize)


R =

  MapPostingsReference with properties:

              XWorldLimits: [207000 208000]
              YWorldLimits: [912500 913000]
                RasterSize: [1001 2001]
      RasterInterpretation: 'postings'
          ColumnsStartFrom: 'south'
             RowsStartFrom: 'west'
      SampleSpacingInWorldX: 1/2
      SampleSpacingInWorldY: 1/2
       RasterExtentInWorldX: 1000
       RasterExtentInWorldY: 500
           XIntrinsicLimits: [1 2001]
           YIntrinsicLimits: [1 1001]
         TransformationType: 'rectilinear'
       CoordinateSystemType: 'planar'
```

Obtain the same result by specifying the sample spacing.

```
spacing = 1/2;

R = maprefpostings(xWorldLimits,yWorldLimits,spacing,spacing)


R =

  MapPostingsReference with properties:
```

```
           XWorldLimits: [207000 208000]
           YWorldLimits: [912500 913000]
             RasterSize: [1001 2001]
   RasterInterpretation: 'postings'
       ColumnsStartFrom: 'south'
          RowsStartFrom: 'west'
  SampleSpacingInWorldX: 1/2
  SampleSpacingInWorldY: 1/2
   RasterExtentInWorldX: 1000
   RasterExtentInWorldY: 500
       XIntrinsicLimits: [1 2001]
       YIntrinsicLimits: [1 1001]
     TransformationType: 'rectilinear'
   CoordinateSystemType: 'planar'
```

## Input Arguments

**xWorldLimits — Limits in the *x* direction**
[0.5 2.5] (default) | 1-by-2 numeric vector

Limits in the *x* direction, specified as a 1-by-2 numeric vector.

Example: xWorldLimits = [207000 208000];

Data Types: double

**yWorldLimits — Limits in the *y* direction**
[0.5 2.5] (default) | 1-by-2 numeric vector

Limits in the *y* direction, specified as a 1-by-2 numeric vector.

Example: yWorldLimits = [912500 913000];

Data Types: double

**rasterSize — Size of the raster**
[2 2] (default) | 1-by-2 numeric vector

Size of the raster, specified as a 1-by-2 numeric vector.

Example: rasterSize = [180 360];

Data Types: double

### sampleSpacingInWorldX — Vertical spacing of posting
1 (default) | numeric scalar

Vertical spacing of posting, specified as a numeric scalar.

Example: spacing = 1.5

Data Types: double

### sampleSpacingInWorldY — Horizontal spacing of postings
1 (default) | numeric scalar

Horizontal spacing of postings, specified as a numeric scalar.

Example: spacing = 1.5

Data Types: double

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: R = maprefpostings(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')

### 'ColumnsStartFrom' — Edge from which column indexing starts
'south' (default) | 'north'

Edge from which column indexing starts, specified as the string 'north' or 'south'.

Example: R = maprefpostings(latlim,lonlim,rasterSize,'ColumnsStartFrom','north')

Data Types: char

### 'RowsStartFrom' — Edge from which row indexing starts
'west' (default) | 'east'

Edge from which row indexing starts, specified as the string 'east' or 'west'.

Example: R =
maprefpostings(latlim,lonlim,rasterSize,'RowsStartFrom','east')

Data Types: char

## Output Arguments

### R — Object that references raster postings to map coordinates
Raster reference object

Object that references raster postings to map coordinates, returned as a raster reference object (map.rasterref.MapPostingsReference).

## More About

### Tips

- To construct a map raster reference object from a world file matrix, use the maprasterref function.

## See Also

map.rasterref.MapPostingsReference | georefpostings | maprefcells

**Introduced in R2015b**

# map.rasterref.MapPostingsReference class

**Package:** map.rasterref

Reference raster postings to map coordinates

## Description

A map postings raster reference object encapsulates the relationship between a planar map coordinate system and a system of intrinsic coordinates anchored to the columns and rows of a 2-D spatially referenced grid of point samples (or "postings"). Typically, the raster is sampled regularly in the planar world *x* and world *y* coordinates of the map system, such that the intrinsic *x* and world *x* axes align and the intrinsic *y* and world *y* axes align. When this is true, the relationship between the two systems is rectilinear. More generally, and much more rarely, their relationship is affine. The affine relationship allows for a possible rotation (and skew). In either case, rectilinear or affine, the sample spacing from row to row need not equal the sample spacing from column to column. For more information about coordinate systems, see "Intrinsic Coordinate System" on page 1-884.

## Construction

You can use any of the following functions to construct a `map.rasterref.MapPostingsReference` object to reference a regular raster of posted samples to planar (map) coordinates.

- `maprefpostings`—Create a map raster reference object.
- `maprasterref`—Convert a world file to a map raster reference object.
- `refmatToMapRasterReference`—Convert a referencing matrix to a map raster reference object.

For example, to construct a map raster reference object with default property settings, use this command:

```
R = maprefpostings()

R =

  MapPostingsReference with properties:
```

```
            XWorldLimits: [0.5 2.5]
            YWorldLimits: [0.5 2.5]
              RasterSize: [2 2]
    RasterInterpretation: 'postings'
       ColumnsStartFrom: 'south'
          RowsStartFrom: 'west'
   SampleSpacingInWorldX: 2
   SampleSpacingInWorldY: 2
    RasterExtentInWorldX: 2
    RasterExtentInWorldY: 2
        XIntrinsicLimits: [1 2]
        YIntrinsicLimits: [1 2]
      TransformationType: 'rectilinear'
     CoordinateSystemType: 'planar''
```

## Properties

### `XWorldLimits` — Limits of raster in world *x* coordinates
`[0.5 2.5]` (default) | two-element row vector

Limits of raster in world *x* coordinates, specified as a two-element row vector of the form
`[xMin xMax]`.

Example: `[207000 209000]`

Data Types: `double`

### `YWorldLimits` — Limits of raster in world *y* coordinates
`[0.5 2.5]` (default) | two-element row vector

Limits of raster in world *y* coordinates, specified as a two-element row vector of the form
`[yMin yMax]`.

Example: `[911000 913000]`

Data Types: `double`

### `RasterSize` — Number of rows and columns of the raster or image associated with the referencing object
`[2 2]` (default) | two-element vector of positive integers

Number of rows and columns of the raster or image associated with the referencing
object, specified as a two-element vector, `[M N]`, where *M* represents the number of rows

and *N* the number of columns. For convenience, you can assign a size vector having more than two elements. This enables assignments like `R.RasterSize = size(RGB)`, where `RGB` is *M*-by-*N*-by-3. In cases like this, the object stores only the first two elements of the size vector and ignores the higher (nonspatial) dimensions. *M* and *N* must be positive in all cases and must be 2 or greater.

Example: `[200 300]`

Data Types: `double`

### RasterInterpretation — Geometric nature of the raster
`'postings'` (default)

Geometric nature of the raster, specified as the string `'postings'`. The value `'postings'` indicates that the raster comprises a grid of sample points, where rows or columns of samples run along the edge of the grid. For an *M*-by-*N* raster, points with an intrinsic *x*-coordinate of 1 or *N* or an intrinsic *y*-coordinate of 1 or *M* fall right on an edge (or corner) of the raster.

Cannot be set.

Data Types: `char`

### ColumnsStartFrom — Edge from which column indexing starts
`'south'` (default) | `'north'`

Edge from which column indexing starts, specified as the string `'south'` or `'north'`.

Example: `ColumnsStartFrom: 'south'`

Data Types: `char`

### RowsStartFrom — Edge from which row indexing starts
`'west'` (default) | `'east'`

Edge from which row indexing starts, specified as the string `'west'` or `'east'`.

Example: `RowsStartFrom: 'east'`

Data Types: `char`

### SampleSpacingInWorldX — East-west distance between adjacent postings
2 (default) | positive numeric scalar

East-west distance between adjacent postings, specified as a positive numeric scalar. The value is constant throughout the raster.

Example: 2.5

Data Types: `double`

### SampleSpacingInWorldY — North-south distance between adjacent postings
2 (default) | positive numeric scalar

North-south distance between adjacent postings, specified as a positive numeric scalar. The value is constant throughout the raster.

Example: 2.5

Data Types: `double`

### RasterExtentInWorldX — Extent of the full raster or image as measured in the world system in a direction parallel to its rows
2 (default) | positive numeric scalar

Extent of the full raster or image as measured in the world system in a direction parallel to its rows, specified as a positive numeric scalar. In the case of a rectilinear geometry, which is most typical, this is the horizontal direction (east-west).

Cannot be set.

Data Types: `double`

### RasterExtentInWorldY — Extent of the full raster or image as measured in the world system in a direction parallel to its columns
2 (default) | positive numeric scalar

Extent of the full raster or image as measured in the world system in a direction parallel to its columns. In the case of a rectilinear geometry, which is most typical, this is the vertical direction (north-south).

Cannot be set.

Data Types: `double`

### XIntrinsicLimits — Raster limits in intrinsic *x* coordinates
[1  2] (default) | two-element row vector of positive integers

Raster limits in intrinsic $x$ coordinates, specified as a two-element row vector of positive integers, [xMin xMax]. For an $M$-by-$N$ raster, XIntrinsicLimits equals [1 M], because the RasterInterpretation is 'postings'.

Cannot be set.

Example: `[2 4]`

Data Types: `double`

### `YIntrinsicLimits` — Raster limits in intrinsic *y* coordinates

`[1 2]` (default) | two-element row vector of positive integers

Raster limits in intrinsic *y* coordinates, specified as a two-element row vector of positive integers, `[yMin yMax]`. For an *M*-by-*N* raster with `RasterInterpretation` equal to `'postings'`, `YIntrinsicLimits` equals `[1 M]`.

Cannot be set.

Data Types: `double`

### `TransformationType` — Type of geometric relationship between intrinsic and world systems

`'rectilinear'` (default) | `'affine'`

Type of geometric relationship between the intrinsic coordinate system and the world coordinate system, specified as either the string `'rectilinear'` or `'affine'`. Its value is `'rectilinear'` when world *x* depends only on intrinsic *x* and vice versa, and world *y* depends only on intrinsic *y* and vice versa. When the value is `'rectilinear'`, the image displays without rotation in the world system, although it might be flipped. Otherwise, the value is `'affine'`.

Cannot be set.

Data Types: `char`

### `CoordinateSystemType` — Type of coordinate system to which the image or raster is referenced

`'planar'` (default)

Type of coordinate system to which the image or raster is referenced, specified as the string `'planar'`.

Cannot be set.

Data Types: `char`

## Methods

| | |
|---|---|
| worldToIntrinsic | Convert from world to intrinsic coordinates |

| worldFileMatrix | World file parameters for transformation |
| sizesMatch | True if object and raster or image are size compatible |
| firstCornerX | World $x$ coordinate of the (1,1) corner of the raster |
| firstCornerY | World $y$ coordinate of the (1,1) corner of the raster |
| worldToDiscrete | Transform map to discrete coordinates |
| intrinsicToWorld | Convert from intrinsic to world coordinates |
| contains | True if raster contains points in world coordinate system |

## Definitions

### Intrinsic Coordinate System

A 2-D Cartesian system with its $x$-axis running parallel to the rows of a raster or image and its $y$-axis running parallel to the columns. $x$ increases by 1 from column to column, and $y$ increases by 1 from row to row.

The Mapping Toolbox and Image Processing Toolbox use the convention for the location of the origin relative to the raster cells or sampling points such that, at a sample location or at the center of a cell, $x$ has an integer value equal to the column index. Likewise, at a sample location or at the center of a cell, $y$ has an integer value equal to the row index. For details, see Image Coordinate Systems in the Image Processing Toolbox documentation.

### See Also
map.rasterref.MapCellsReference | map.rasterref.GeographicPostingsReference | maprasterref | maprefpostings

### More About
- Class Attributes
- Property Attributes

**Introduced in R2013b**

# worldToIntrinsic

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

Convert from world to intrinsic coordinates

## Syntax

```
[xIntrinsic, yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)
```

## Description

`[xIntrinsic, yIntrinsic] = worldToIntrinsic(R,xWorld,yWorld)` maps point locations from the world system (`xWorld`, `yWorld`) to the intrinsic system (`xIntrinsic`, `yIntrinsic`) based on the relationship defined by the referencing object `R`. The input can include values outside limits of the raster (or image) in the world system. In this case, world $x$ and $y$ are extrapolated outside the bounds of the image in the intrinsic system.

# worldFileMatrix

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

World file parameters for transformation

## Syntax

```
W = worldFileMatrix(R)
```

## Description

`W = worldFileMatrix(R)` returns a 2-by-3 world file matrix. Each of the six elements in `W` matches one of the lines in a world file corresponding to the rectilinear or affine transformation defined by referencing object `R`.

Given `W` with the form:

```
W = [A B C;
     D E F]
```

a point `(xi, yi)` in intrinsic coordinates maps to a point `(xw, yw)` in planar world coordinates like this:

```
xw = A * (xi - 1) + B * (yi - 1) + C
yw = D * (xi - 1) + E * (yi - 1) + F
```

More compactly:

```
[xw yw]' = W * [xi - 1  yi - 1  1]'
```

The `-1`s are needed to maintain the Mapping Toolbox convention for intrinsic coordinates, which is consistent with the 1-based indexing used throughout MATLAB.

`W` is stored in a world file with one term per line in column-major order: A, D, B, E, C, F. That is, a world file contains the elements of `W` in this order:

```
W(1,1)
W(2,1)
```

```
W(1,2)
W(2,2)
W(1,3)
W(2,3)
```

The previous expressions hold for both affine and rectilinear transformations. But whenever `R.TransformationType` is `'rectilinear'`, B, D, `W(2,1)` and `W(1,2)` are identically 0.

## See Also
`worldfileread` | `worldfilewrite`

# sizesMatch

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

True if object and raster or image are size compatible

## Syntax

```
TF = sizesMatch(R,A)
```

## Description

TF = sizesMatch(R,A) returns true if the size of the raster or image A is consistent with the RasterSize property of referencing object R. That is:

```
R.RasterSize == [size(A,1) size(A,2)]
```

# firstCornerX

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

World *x* coordinate of the (1,1) corner of the raster

## Syntax

```
firstCornerX(R)
```

## Description

`firstCornerX(R)` returns the world *x* coordinate of the first sample point of the raster associated with referencing object `R`.

# firstCornerY

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

World *y* coordinate of the (1,1) corner of the raster

## Syntax

```
firstCornerY(R)
```

## Description

`firstCornerY(R)` returns the world *y* coordinate of the first sample point of the raster associated with referencing object `R`.

# worldToDiscrete

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

Transform map to discrete coordinates

## Syntax

```
[I,J] = worldToDiscrete(R,xWorld,yWorld)
```

## Description

`[I,J] = worldToDiscrete(R,xWorld,yWorld)` returns subscript arrays `I` and `J`.

If `R.RasterInterpretation` is `'postings'`, `I` and `J` refer to the nearest sample point (posting).

`xWorld` and `yWorld` must have the same size. `I` and `J` have the same size as `xWorld` and `yWorld`. For an $M$-by-$N$ raster, `1 <= I <= M` and `1 <= J <= N`, except when a point `xWorld(k)`, `yWorld(k)` falls outside the image, as defined by `contains(R,xWorld,yWorld)`. Then both `I(k)` and `J(k)` are NaN.

# intrinsicToWorld

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

Convert from intrinsic to world coordinates

## Syntax

```
[xWorld, yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)
```

## Description

`[xWorld, yWorld] = intrinsicToWorld(R,xIntrinsic,yIntrinsic)` maps point locations from the intrinsic system `(xIntrinsic, yIntrinsic)` to the world system `(xWorld, yWorld)` based on the relationship defined by the referencing object `R`. If your input includes values outside the limits of the raster or image in the intrinsic system, `intrinsicToWorld` extrapolates world *x* and *y* outside the bounds of the image in the world system.

# contains

**Class:** map.rasterref.MapPostingsReference
**Package:** map.rasterref

True if raster contains points in world coordinate system

## Syntax

```
TF = contains(R,xWorld,yWorld)
```

## Description

TF = contains(R,xWorld,yWorld) returns a logical array TF having the same size as xWorld and yWorld such that TF(k) is true if and only if the point (xWorld(k), yWorld(k)) falls within the bounds of the raster associated with referencing object R.

# maps

List available map projections and verify names

## Syntax

```
strmat = maps('namelist')
strmat = maps('idlist')
stdstr = maps(id_string)
```

## Description

`maps` displays in the Command Window a table describing all projections available for use.

`strmat = maps('namelist')` returns the English names for the available projections as a matrix of strings.

`strmat = maps('idlist')` returns the standard projection identification strings for the available projections as a matrix of strings.

`stdstr = maps(id_string)` returns the specific standard projection identification string associated with a unique truncation abbreviation.

## Examples

To show the first five entries of the projections name list,

```
str1 = maps('namelist');
str1(1:5,:)
ans =
Balthasart Cylindrical
Behrmann Cylindrical
Bolshoi Sovietskii Atlas Mira
Braun Perspective Cylindrical
Cassini Cylindrical
```

The corresponding shorthand names are

```
str2 = maps('idlist');
str2(1:5,:)
ans =
balthsrt
behrmann
bsam
braun
cassini
```

These are the strings used, for example, when setting the `axesm` property
`MapProjection`.

The functions `setm` and `axesm` recognize unique abbreviations (truncations) of these
strings. The `maps` function can be used to convert such an abbreviation to the standard
ID string:

```
stdstr = maps('merc')
stdstr =
mercator
```

When the function name alone is used,

```
maps

MapTools Projections
CLASS              NAME                           ID STRING
Cylindrical        Balthasart Cylindrical         balthsrt
Cylindrical        Behrmann Cylindrical           behrmann
Cylindrical        Bolshoi Sovietskii Atlas Mira* bsam
Cylindrical        Braun Perspective Cylindrical* braun
Cylindrical        Cassini Cylindrical            cassini
Cylindrical        Central Cylindrical*           ccylin
Cylindrical        Equal Area Cylindrical         eqacylin
Cylindrical        Equidistant Cylindrical        eqdcylin
Cylindrical        Gall Isographic                giso...
```

The actual result contains all defined projections.

## See Also
axesm | setm

# mapshape class

Planar shape vector

## Syntax

```
s = mapshape()
s = mapshape(x,y)
s = mapshape(x,y,Name,Value)
s = mapshape(structArray)
s = mapshape(x,y,structArray)
```

## Description

A mapshape vector is an object that represents planar vector features with either point, line, or polygon topology. The features consist of X and Y coordinates and associated attributes. If these attributes vary spatially they are termed Vertex properties. These elements of the mapshape vector are coupled such that the length of the X and Y coordinate property values are always equal in length to any additional dynamic Vertex properties. Attributes which only pertain to the overall feature (point, line, polygon) are termed Feature properties. Feature properties are not linked to the auto-sizing mechanism of the Vertex properties. Both of the property types can be dynamically added to a mapshape vector using the standard dot notation.

A mapshape vector is always a column vector.

## Construction

`s = mapshape()` constructs an empty mapshape vector, `s`, with the following default property settings.

```
s =

 0x1 mapshape vector with properties:
```

```
Collection properties:
   Geometry: 'line'
   Metadata: [1x1 struct]
Vertex properties:
         X: []
         Y: []
```

For an additional example see: "Construct a Default mapshape Vector and Set and Add Properties" on page 1-903

`s = mapshape(x,y)` constructs a mapshape vector and sets the X and Y property values equal to vectors `x` and `y`. `x` and `y` may be either numeric vectors of class single or double, or cell arrays containing numeric vectors of class single or double. For an example, see "Construct a mapshape Vector Specifying x and y Values" on page 1-904.

`s = mapshape(x,y,Name,Value)` constructs a mapshape vector from the input `x` and `y` vectors, and then adds dynamic properties to the mapshape vector using the `Name`, `Value` argument pairs.

- If `Value` is in a cell array containing numeric, logical or cell array of strings, then this property is designated as a Vertex property. Otherwise, this property is designated as a Feature property.
- If the specified `Name` is `Metadata` and the corresponding `Value` is a scalar structure, then `Value` is copied to the `Metadata` property. Otherwise, an error is issued.

For an example, see: "Construct a mapshape vector Specifying a Name-Value Pair" on page 1-905.

`s = mapshape(structArray)` constructs a `mapshape` vector from the fields of the structure array, `structArray`.

- If `structArray` is a scalar structure which contains the field `Metadata` and the field value is a scalar structure, then the `Metadata` field is copied to the `Metadata` property. If `structArray` is a scalar structure and the `Metadata` field is present and is not a scalar structure, then an error is issued. If `structArray` is not scalar then the `Metadata` field is ignored.
- Other fields of `structArray` are assigned to `s` and become dynamic properties. Field values in `structArray` that are not numeric, strings, logical, or cell arrays of numeric, logical, or string values are ignored.

For an example, see "Construct a mapshape Vector from a Structure Array" on page 1-905.

`s = mapshape(x,y,structArray)` constructs a new `mapshape` vector and sets the X and Y properties equal to the numeric vectors, `x` and `y`, and sets the field values of struct `structArray` as dynamic properties.

- If `structArray` is a scalar structure and contains the field `Metadata`, and the field value is a scalar structure, then it is copied to the `Metadata` property value. Otherwise, an error is issued if the `Metadata` field is not a structure, or ignored if `structArray` is not scalar.

For an example, see "Construct a mapshape Vector Using Cell Arrays and Structures to Define Multiple Features and Properties" on page 1-906.

## Input Arguments

### x — Vector of latitude coordinates
numeric array

Vector of latitude coordinates, specified as a numeric array.

Data Types: `double` | `single` | `cell`

### y — Vector of longitude coordinates
numeric array

Vector of longitude coordinates, specified as a numeric array.

Data Types: `double` | `single` | `cell`

### `structArray`

An array of structures containing fields to be assigned as dynamic properties.

### `Name` — Name of dynamic property
string

Name of dynamic property, specified as a string.

Data Types: `char`

**Value** — Value associated with a dynamic property specified in **Name**
numeric, logical, char, or a cell array of strings

Value associated with a dynamic property specified in `Name`, specified as numeric, logical, cell array of strings, or a cell array of numeric, logical, or cell array of strings, for Vertex dynamic properties. The class type of the values for the Feature dynamic properties may be either numeric, logical, char, or a cell array of strings.

## Output Arguments

**s**

mapshape vector.

# Properties

`mapshape` class is a general class that represents a variety of planar features. The class permits features to have more than one vertex and can thus represent lines and polygons in addition to multipoints. The class has the following property types.

| Types of Properties | Description |
|---|---|
| Collection Properties | Collection properties contain only one value per class instance. This is in contrast to the other two property types which can have attribute values associated with each feature or with each vertex in a set that defines a feature. Geometry and Metadata are the only two Collection properties. |
| Vertex Properties | Vertex properties provide a scalar number or a string for each vertex in a mapshape object. Vertex properties are suitable for attributes that vary spatially from point to point (vertex to vertex) along a line. Examples of such spatially varying attributes could be elevation, speed, temperature, or time. X and Y are vertex properties since they contain a scalar number for each vertex in a mapshape vector. Attribute values can be dynamically associated with each vertex by using dot notation. This is similar to adding dynamic fields to a structure. The dynamically added |

| Types of Properties | Description |
|---|---|
| | vertex property values of an individual feature match its X and Y values in length. |
| Feature Properties | Feature properties provide one value (a scalar number or a string) for each feature in a mapshape vector. They are suitable for properties, such as name, owner, serial number, age, etc., that describe a given feature (an element of a mapshape vector) as a whole. Like Vertex properties, Feature properties can be added dynamically. |

### Geometry

The `Geometry` property is a string that denotes the shape type for all the features in the mapshape vector. As a Collection Property there is only one value per object instance. Its purpose is purely informational; the three allowable string values for `Geometry` do not change class behavior. Additionally, the class does not provide validation for line or polygon topologies.

Default value for `Geometry` is 'line'.

```
Geometry                              'point', 'line', 'polygon'
```

### Metadata

Metadata is a scalar structure containing information for all the features. You can add any data type to the structure. As a Collection Property type, only one instance per object is allowed.

```
Metadata                    Scalar struct
```

### X

Vector of X coordinates. The values can be either a row or column vector, but are stored as a row vector.

**Attributes:**

```
X                           single | double vector
```

**Y**

Vector of Y coordinates. The values can be either a row or column vector, but are stored as a row vector.

**Attributes:**

| | |
|---|---|
| Y | `single | double vector` |

# Methods

| | |
|---|---|
| append | Append features to `mapshape` vector |
| append | Append features to `mapshape` vector |
| cat | Concatenate mapshape vectors |
| disp | Display mapshape vector |
| fieldnames | Dynamic properties of mapshape vector |
| isempty | True if mapshape vector is empty |
| isfield | True if dynamic property exists |
| isprop | True if property exists |
| length | Number of elements in mapshape vector |
| properties | Properties of a mapshape vector |
| rmfield | Remove dynamic property from mapshape vector |
| rmprop | Remove properties from mapshape vector |
| size | Size of `mapshape` vector |
| struct | Convert mapshape vector to scalar structure |
| vertcat | Vertical concatenation for mapshape vectors |

# Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects in the MATLAB documentation.

# Class Behaviors

- The following examples show how to build a mapshape vector by dynamically adding a single features after construction using dot (.) notation.

  "Construct a mapshape Vector and Add a Feature Property" on page 1-907

- The following extended example adds multiple features that are both Vertex and Feature properties. It also demonstrates property behaviors when vector lengths are either changed or set to [ ].

  "Construct a mapshape Vector and Manipulate Features" on page 1-908

- This example uses the Name-Value constructor to build a mapshape vector and define two dynamic features.

  "Construct a mapshape Vector Specifying Several Name-Value Pairs" on page 1-912

- The mapshape vector can be indexed like any MATLAB vector. You can access any element of the vector to obtain a specific feature. The following example demonstrates this behavior.

  "Construct a mapshape vector Containing Multiple Features and Indexing Behaviors" on page 1-913

- This example builds a mapshape vector from a structure array; adds a Metadata property and demonstrates selective indexing behavior.

  "Construct a mapshape Vector and Add Metadata and Indexing" on page 1-916

- The following example shows a variety of indexing behaviors.

  "Construct a mapshape Vector and Sort the Dynamic Properties" on page 1-917

- If either X or Y is set to [ ], then both coordinate properties are set to [ ] and all dynamic Vertex or Feature properties are removed.

- If a Vertex or Feature property is set to [ ], then it is removed from the object.

# Examples

### Construct a Default mapshape Vector and Set and Add Properties

Create default mapshape vector.

```
s = mapshape()

0x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
 Vertex properties:
           X: []
           Y: []
```

Set the values of the existing X and Y properties and dynamically add the Vertex property Z.

```
s(1).X = 0:45:90;
s(1).Y= [10 10 10];
s(1).Z = [10 20 30]

s =

 1x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
 Vertex properties:
           X: [0 45 90]
           Y: [10 10 10]
           Z: [10 20 30]
```

### Construct a mapshape Vector Specifying x and y Values

Create a mapshape vector specifying x and y.

```
x = [40, 50, 60];
y = [10, 20, 30];
shape = mapshape(x, y)

shape =

 1x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
```

```
    Metadata: [1x1 struct]
 Vertex properties:
            X: [40 50 60]
            Y: [10 20 30]
```

### Construct a mapshape vector Specifying a Name-Value Pair

Create mapshape vector specifying a name-value pair.

```
x = 1:10;
y = 21:30;
temperature = {61:70};
shape = mapshape(x, y, 'Temperature', temperature)

shape =

 1x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
              X: [1 2 3 4 5 6 7 8 9 10]
              Y: [21 22 23 24 25 26 27 28 29 30]
    Temperature: [61 62 63 64 65 66 67 68 69 70]
```

When `Value` is a cell array containing numeric, logical, or cell array of strings, it is construed as a Vertex property. Otherwise the `Name-Value` pair is designated as being a Feature property.

### Construct a mapshape Vector from a Structure Array

Create structure array and then create mapshape vector with array.

```
    structArray = shaperead('concord_roads');
    shape = mapshape(structArray)

shape =

 609x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
```

```
Vertex properties:
  (609 features concatenated with 608 delimiters)
            X: [1x5422 double]
            Y: [1x5422 double]
Feature properties:
    STREETNAME: {1x609 cell}
     RT_NUMBER: {1x609 cell}
         CLASS: [1x609 double]
    ADMIN_TYPE: [1x609 double]
        LENGTH: [1x609 double]
```

**Construct a mapshape Vector Using Cell Arrays and Structures to Define Multiple Features and Properties**

Read data from a shape file into a structure.

```
[structArray, A] = shaperead('concord_hydro_area');

structArray =

98x1 struct array with fields:

    Geometry
    BoundingBox
    X
    Y


A =

98x1 struct array with fields:

    AREA
    PERIMETER
```

Create a mapshape vector specifying the structure.

```
shape = mapshape({structArray.X}, {structArray.Y}, A);
shape.Geometry = structArray(1).Geometry

shape =

 98x1 mapshape vector with properties:
```

```
Collection properties:
     Geometry: 'polygon'
     Metadata: [1x1 struct]
Vertex properties:
  (98 features concatenated with 97 delimiters)
            X: [1x4902 double]
            Y: [1x4902 double]
Feature properties:
         AREA: [1x98 double]
    PERIMETER: [1x98 double]
```

**Construct a mapshape Vector and Add a Feature Property**

Create a mapshape vector.

```
x = 0:10:100;
y = 0:10:100;
shape = mapshape(x, y)

shape =

 1x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
 Vertex properties:
            X: [0 10 20 30 40 50 60 70 80 90 100]
            Y: [0 10 20 30 40 50 60 70 80 90 100]
```

Add a feature dynamic property.

```
shape.FeatureName = 'My Feature'

shape =

 1x1 mapshape vector with properties:

 Collection properties:
      Geometry: 'line'
      Metadata: [1x1 struct]
 Vertex properties:
              X: [0 10 20 30 40 50 60 70 80 90 100]
              Y: [0 10 20 30 40 50 60 70 80 90 100]
 Feature properties:
```

```
      FeatureName: 'My Feature'
```

Add a vertex dynamic property to the first feature.

```
shape(1).Temperature = 65 + rand(1, length(shape.X))

shape =

 1x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
              X: [0 10 20 30 40 50 60 70 80 90 100]
              Y: [0 10 20 30 40 50 60 70 80 90 100]
    Temperature: [1x11 double]
 Feature properties:
    FeatureName: 'My Feature
```

### Construct a mapshape Vector and Manipulate Features

Create a mapshape vector.

```
x = {1:3, 4:6};
y = {[0 0 0], [1 1 1]};
shape = mapshape(x, y)


shape =

 2x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
          X: [1 2 3 NaN 4 5 6]
          Y: [0 0 0 NaN 1 1 1]
```

Add a two element feature dynamic property.

```
shape.FeatureName = {'Feature 1', 'Feature 2'}
```

```
shape =

 2x1 mapshape vector with properties:

 Collection properties:
        Geometry: 'line'
        Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
               X: [1 2 3 NaN 4 5 6]
               Y: [0 0 0 NaN 1 1 1]
 Feature properties:
    FeatureName: {'Feature 1' 'Feature 2'}
```

Add a vertex dynamic property.

```
z = {101:103, [115, 114, 110]}
shape.Z = z

z =

    [1x3 double]    [1x3 double]


shape =

 2x1 mapshape vector with properties:

 Collection properties:
        Geometry: 'line'
        Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
               X: [1 2 3 NaN 4 5 6]
               Y: [0 0 0 NaN 1 1 1]
               Z: [101 102 103 NaN 115 114 110]
 Feature properties:
    FeatureName: {'Feature 1' 'Feature 2'}
```

Display the second feature.

```
shape(2)

ans =
```

```
1x1 mapshape vector with properties:

Collection properties:
      Geometry: 'line'
      Metadata: [1x1 struct]
Vertex properties:
             X: [4 5 6]
             Y: [1 1 1]
             Z: [115 114 110]
Feature properties:
   FeatureName: 'Feature 2'
```

Add a third feature. The lengths of all the properties are synchronized.

```
shape(3).X = 5:9

shape =

 3x1 mapshape vector with properties:

Collection properties:
      Geometry: 'line'
      Metadata: [1x1 struct]
Vertex properties:
 (3 features concatenated with 2 delimiters)
             X: [1 2 3 NaN 4 5 6 NaN 5 6 7 8 9]
             Y: [0 0 0 NaN 1 1 1 NaN 0 0 0 0 0]
             Z: [101 102 103 NaN 115 114 110 NaN 0 0 0 0 0]
Feature properties:
   FeatureName: {'Feature 1' 'Feature 2' ''}
```

Set the values for the Z vertex property with fewer values than contained in X or Y. The Z values expand to match the length of X and Y.

```
shape(3).Z = 1:3

shape =

 3x1 mapshape vector with properties:

Collection properties:
      Geometry: 'line'
      Metadata: [1x1 struct]
Vertex properties:
 (3 features concatenated with 2 delimiters)
```

```
              X: [1 2 3 NaN 4 5 6 NaN 5 6 7 8 9]
              Y: [0 0 0 NaN 1 1 1 NaN 0 0 0 0 0]
              Z: [101 102 103 NaN 115 114 110 NaN 1 2 3 0 0]
 Feature properties:
    FeatureName: {'Feature 1' 'Feature 2' ''}
```

Set the values for either coordinate property (X or Y) and all properties shrink in size to match the new vertex length of that feature.

```
shape(3).Y = 1

shape =

 3x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
  (3 features concatenated with 2 delimiters)
              X: [1 2 3 NaN 4 5 6 NaN 5]
              Y: [0 0 0 NaN 1 1 1 NaN 1]
              Z: [101 102 103 NaN 115 114 110 NaN 1]
 Feature properties:
    FeatureName: {'Feature 1' 'Feature 2' ''}
```

Set the values for the Z vertex property with more values % than contained in X or Y. All properties expand in length % to match Z.

```
shape(3).Z = 1:6

shape =

 3x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
  (3 features concatenated with 2 delimiters)
              X: [1 2 3 NaN 4 5 6 NaN 5 0 0 0 0 0]
              Y: [0 0 0 NaN 1 1 1 NaN 1 0 0 0 0 0]
              Z: [101 102 103 NaN 115 114 110 NaN 1 2 3 4 5 6]
 Feature properties:
    FeatureName: {'Feature 1' 'Feature 2' ''}
```

Remove the FeatureName property.

```
shape.FeatureName = []

shape =

 3x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
 Vertex properties:
  (3 features concatenated with 2 delimiters)
           X: [1 2 3 NaN 4 5 6 NaN 5 0 0 0 0 0]
           Y: [0 0 0 NaN 1 1 1 NaN 1 0 0 0 0 0]
           Z: [101 102 103 NaN 115 114 110 NaN 1 2 3 4 5 6]
```

Remove all dynamic properties and set the object to empty.

```
shape.X = []

shape =

 0x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
 Vertex properties:
           X: []
           Y: []
```

**Construct a mapshape Vector Specifying Several Name-Value Pairs**

Create a mapshape vector specifying several name-value pairs.

```
x = {1:3, 4:6};
y = {[0 0 0], [1 1 1]};
z = {41:43, [56 50 59]};
name = {'Feature 1', 'Feature 2'};
id = [1 2];
shape = mapshape(x, y, 'Z', z, 'Name', name, 'ID', id)

shape =
```

```
2x1 mapshape vector with properties:

Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
Vertex properties:
 (2 features concatenated with 1 delimiter)
          X: [1 2 3 NaN 4 5 6]
          Y: [0 0 0 NaN 1 1 1]
          Z: [41 42 43 NaN 56 50 59]
Feature properties:
       Name: {'Feature 1' 'Feature 2'}
         ID: [1 2]
```

### Construct a mapshape vector Containing Multiple Features and Indexing Behaviors

Load the data and create x, y, and z arrays. Create a level list to use to bin the z values.

```
seamount = load('seamount');
x = seamount.x; y = seamount.y; z = seamount.z;

levels = [unique(floor(seamount.z/1000)) * 1000; 0];
```

Construct a mapshape object and assign the X and Y vertex properties to the binned x and y values. Create a new Z vertex property to contain the binned z values. Add a Levels feature property to contain the lowest level value per feature.

```
shape = mapshape;
for k = 1:length(levels) - 1
   index = z >= levels(k) & z < levels(k+1);
   shape(k).X = x(index);
   shape(k).Y = y(index);
   shape(k).Z = z(index);
   shape(k).Level = levels(k);
end
```

Add a Color feature property to denote a color for that feature, and specify that the geometry is 'point'

```
shape.Color = {'red', 'green', 'blue', 'cyan', 'black'};
shape.Geometry = 'point'

shape =
```

```
5x1 mapshape vector with properties:

Collection properties:
   Geometry: 'point'
   Metadata: [1x1 struct]
Vertex properties:
 (5 features concatenated with 4 delimiters)
         X: [1x298 double]
         Y: [1x298 double]
         Z: [1x298 double]
Feature properties:
      Level: [-5000 -4000 -3000 -2000 -1000]
      Color: {'red' 'green' 'blue' 'cyan' 'black'}
```

Add metadata information. Metadata is a scalar structure containing information for the entire set of properties. Any type of data may be added to the structure.

```
shape.Metadata.Caption = seamount.caption;
shape.Metadata

ans =

    Caption: [1x229 char]
```

Display the point data in 2D.

```
figure
for k=1:length(shape)
  mapshow(shape(k).X, shape(k).Y, ...
    'MarkerEdgeColor', shape(k).Color, ...
    'Marker', 'o', ...
    'DisplayType', shape.Geometry)
end
legend(num2str(shape.Level'))
```
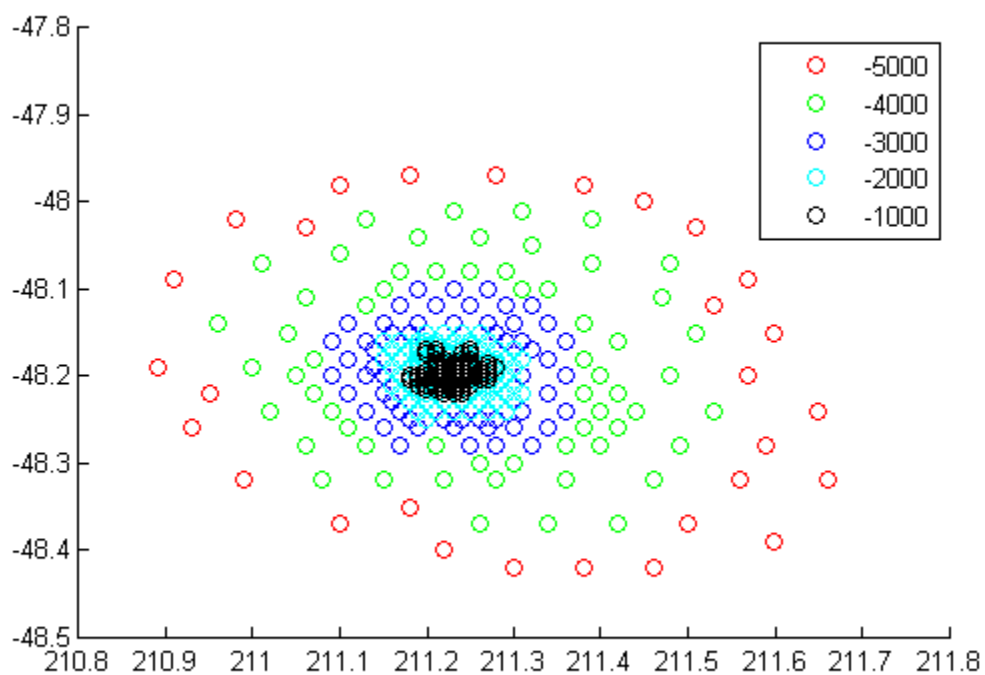
Display data as a 3-D scatter plot.

```
figure
scatter3(shape.X, shape.Y, shape.Z)
```
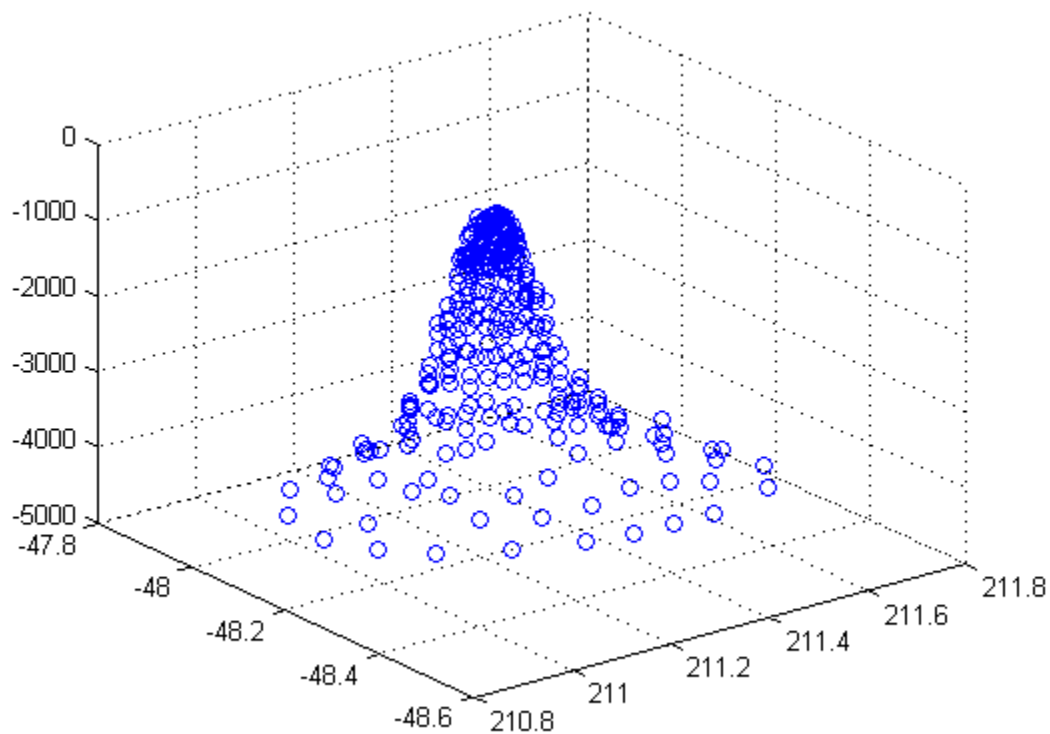
### Construct a mapshape Vector and Add Metadata and Indexing

Construct a mapshape vector from a structure array

```
filename = 'concord_roads.shp';
S = shaperead(filename);
shape = mapshape(S)

shape =

 609x1 mapshape vector with properties:

 Collection properties:
     Geometry: 'line'
```

```
      Metadata: [1x1 struct]
 Vertex properties:
  (609 features concatenated with 608 delimiters)
              X: [1x5422 double]
              Y: [1x5422 double]
 Feature properties:
    STREETNAME: {1x609 cell}
     RT_NUMBER: {1x609 cell}
         CLASS: [1x609 double]
    ADMIN_TYPE: [1x609 double]
        LENGTH: [1x609 double]
```

Add a Filename to the Metadata structure and then construct a new mapshape object with only CLASS 4 (major road) designation.

```
shape.Metadata.Filename = filename;
class4 = shape(shape.CLASS == 4)

class4 =

 26x1 mapshape vector with properties:

 Collection properties:
      Geometry: 'line'
      Metadata: [1x1 struct]
 Vertex properties:
  (26 features concatenated with 25 delimiters)
              X: [1x171 double]
              Y: [1x171 double]
 Feature properties:
    STREETNAME: {1x26 cell}
     RT_NUMBER: {1x26 cell}
         CLASS: [4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4]
    ADMIN_TYPE: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
        LENGTH: [1x26 double]
```

### Construct a mapshape Vector and Sort the Dynamic Properties

You can create a new mapshape vector that contains a subset of dynamic properties by adding the name of a property or a cell array of property names to the last index in the () operator.

Read data from file directly in mapshape constructor.

```
shape = mapshape(shaperead('tsunamis'))
```

```
shape =

 162x1 mapshape vector with properties:

 Collection properties:
        Geometry: 'point'
        Metadata: [1x1 struct]
 Vertex properties:
  (162 features concatenated with 161 delimiters)
               X: [1x323 double]
               Y: [1x323 double]
 Feature properties:
           Cause: {1x162 cell}
      Cause_Code: [1x162 double]
         Country: {1x162 cell}
             Day: [1x162 double]
     Desc_Deaths: [1x162 double]
          Eq_Mag: [1x162 double]
            Hour: [1x162 double]
        Iida_Mag: [1x162 double]
       Intensity: [1x162 double]
        Location: {1x162 cell}
      Max_Height: [1x162 double]
          Minute: [1x162 double]
           Month: [1x162 double]
      Num_Deaths: [1x162 double]
          Second: [1x162 double]
        Val_Code: [1x162 double]
        Validity: {1x162 cell}
            Year: [1x162 double]
```

Alphabetize the Feature properties.

```
shape = shape(:, sort(fieldnames(shape)))

shape =

 162x1 mapshape vector with properties:

 Collection properties:
        Geometry: 'point'
        Metadata: [1x1 struct]
 Vertex properties:
  (162 features concatenated with 161 delimiters)
               X: [1x323 double]
```

```
                     Y: [1x323 double]
Feature properties:
          Cause: {1x162 cell}
     Cause_Code: [1x162 double]
        Country: {1x162 cell}
            Day: [1x162 double]
    Desc_Deaths: [1x162 double]
         Eq_Mag: [1x162 double]
           Hour: [1x162 double]
       Iida_Mag: [1x162 double]
      Intensity: [1x162 double]
       Location: {1x162 cell}
     Max_Height: [1x162 double]
         Minute: [1x162 double]
          Month: [1x162 double]
     Num_Deaths: [1x162 double]
         Second: [1x162 double]
       Val_Code: [1x162 double]
       Validity: {1x162 cell}
           Year: [1x162 double]
```

Modify the mapshape vector to contain only the specified dynamic properties.

```
shape = shape(:, {'Year', 'Month', 'Day', 'Hour', 'Minute'})

shape =

 162x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Vertex properties:
  (162 features concatenated with 161 delimiters)
          X: [1x323 double]
          Y: [1x323 double]
 Feature properties:
       Year: [1x162 double]
      Month: [1x162 double]
        Day: [1x162 double]
       Hour: [1x162 double]
     Minute: [1x162 double]
```

Create a new mapshape vector in which each feature contains the points for the same year. Copy the data from a mappoint vector to ensure that NaN feature separators are

not included. Create a subsection of data to include only Year and Country dynamic properties.

```
points = mappoint(shaperead('tsunamis'));
points = points(:, {'Year', 'Country'});
years = unique(points.Year);
multipoint = mapshape();
multipoint.Geometry = 'point';
for k = 1:length(years)
   index = points.Year == years(k);
   multipoint(k).X = points(index).X;
   multipoint(k).Y = points(index).Y;
   multipoint(k).Year = years(k);
   multipoint(k).Country = points(index).Country;
end
multipoint    % Display

multipoint =

 53x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Vertex properties:
  (53 features concatenated with 52 delimiters)
          X: [1x214 double]
          Y: [1x214 double]
     Country: {1x214 cell}
 Feature properties:
        Year: [1x53 double]
```

Display the third from the end feature.

```
multipoint(end-3)

ans =

 1x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Vertex properties:
          X: [3.6340 -62.1800 143.9100]
```

```
        Y: [36.9640 16.7220 41.8150]
  Country: {'ALGERIA' 'MONTSERRAT' 'JAPAN'}
Feature properties:
      Year: 2003
```

## See Also

geopoint | geoshape | mappoint | gpxread | shaperead

# append

**Class:** mapshape

Append features to `mapshape` vector

## Syntax

```
outShape = append(inShape,X,Y)
outShape = append( ___ ,field,value,...)
```

## Description

`outShape = append(inShape,X,Y)` appends the vector, X, to the X property of the mapshape vector, `inShape`, and the vector, Y, to the Y property of `inShape`. X and Y are either vectors of class single or double, or cell arrays containing numeric arrays of class single or double.

`outShape = append( ___ ,field,value,...)` appends the values specified in `value` to the corresponding dynamic property, `field`. If the property does not exist, `append` adds the dynamic property to the object using the value of `field` for the name and assigning the field the value specified in `value`.

## Input Arguments

**inShape**

mapshape vector

**X**

Numeric vector of planar X values

**Y**

Numeric vector of planar Y values

**field**

Field name, specified as a string. Field can specify the name of an existing property in the mapshape vector or the name you want assigned to a new property that you want to add to the mapshape vector.

**value**

Value you want to assign to the property specified by `field`, specified as a cell array, scalar, or vector of any numeric class or logical.

- When `value` is a cell array, `append` adds the property as a Vertex property.
- When `value` is a numeric array, `append` adds the property as a Feature property.
- When `value` is empty, `append` removes the property.

# Output Arguments

**outShape**

Modified mapshape vector with additional entries in X and Y fields along with any new fields for dynamic properties that you added.

# Examples

### Append Values to Field

Create a mapshape vector.

```
shape = mapshape(42:44,30:32, 'Temperature', {65:67})

shape =

 1x1 mapshape vector with properties:

 Collection properties:
        Geometry: 'line'
        Metadata: [1x1 struct]
 Vertex properties:
               X: [42 43 44]
               Y: [30 31 32]
```

```
      Temperature: [65 66 67]
```

Append data representing another feature.

```
shape = append(shape, 42.1, 33, 'Temperature', 65.5)

shape =

 2x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
               X: [42 43 44 NaN 42.1000]
               Y: [30 31 32 NaN 33]
     Temperature: [65 66 67 NaN 65.5000]
```

Note that the mapshape vector grew from 1x1 to 2x1 in length.

## See Also

mapshape | mapshape.vertcat

# cat

**Class:** mapshape

Concatenate mapshape vectors

# Syntax

```
s= cat(dim,s1, s2, ...)
```

# Description

s= cat(dim,s1, s2, ...) concatenates the mapshape vectors s1,s2 and so on along dimensions dim. dim must be 1.

# Input Arguments

**s1, s2, ...**

mapshape vectors to be concatenated.

# Output Arguments

**s**

Concatenated mapshape vector.

# Examples

### Concatenate two mapshape vectors

Create two mapshape vectors and concatenate them into a single vector.

```
s1 = mapshape(42,-110, 'Temperature', 65);
```

```
s2 = mapshape(42.2, -110.5, 'Temperature', 65.6);
s1s2 = cat(1,s1,s2)

s1s2 =

 2x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
        X: [42 NaN 42.2000]
        Y: [-110 NaN -110.5000]
 Feature properties:
    Temperature: [65 65.6000]
```

## See Also

mapshape.vertcat

# disp

**Class:** mapshape

Display mapshape vector

## Syntax

```
disp(s)
```

## Description

disp(s) prints the size of the mapshape vector, s, and its properties and dynamic properties, if they exist. If the command window is large enough, the values of the properties are also shown, otherwise only their size is shown. You can control the display of the numerical values by using the format command.

## Input Arguments

**s**

mapshape vector.

## Examples

### Display a mapshape vector

Create a mapshape vector and display it.

```
s = mapshape(shaperead('worldcities', 'UseGeo', true));
disp(s)
disp(s(1:2))
```

```
 318x1 mapshape vector with properties:

 Collection properties:
```

```
   Geometry: 'point'
   Metadata: [1x1 struct]
Vertex properties:
 (318 features concatenated with 317 delimiters)
          X: [1x635 double]
          Y: [1x635 double]
Feature properties:
        Lon: [1x318 double]
        Lat: [1x318 double]
       Name: {1x318 cell}

2x1 mapshape vector with properties:

Collection properties:
   Geometry: 'point'
   Metadata: [1x1 struct]
Vertex properties:
 (2 features concatenated with 1 delimiter)
          X: [0 NaN 0]
          Y: [0 NaN 0]
Feature properties:
        Lon: [-3.9509 54.7589]
        Lat: [5.2985 24.6525]
       Name: {'Abidjan' 'Abu Dhabi'}
```

## See Also

mapshape | format

# fieldnames

**Class:** mapshape

Dynamic properties of mapshape vector

## Syntax

```
names = fieldnames(s)
```

## Description

`names = fieldnames(s)` returns the names of the dynamic properties of the mapshape vector, `s`.

## Input Arguments

**s**

mapshape vector for which the properties are to be queried.

## Output Arguments

**names**

Names of the dynamic properties in the mapshape vector `s`

## Examples

### Find Dynamic Properties

Create mapshape vector and view names of properties.

```
s = mapshape(shaperead('worldcities'))
```

```
fieldnames(s)

s =

 318x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Vertex properties:
  (318 features concatenated with 317 delimiters)
          X: [1x635 double]
          Y: [1x635 double]
 Feature properties:
       Name: {1x318 cell}


ans =

    'Name'
```

## See Also

mapshape.properties

# isempty

**Class:** mapshape

True if mapshape vector is empty

## Syntax

```
tf = isempty(s)
```

## Description

`tf = isempty(s)` returns true if the mapshape vector, `s`, is empty and false otherwise.

## Input Arguments

**s**

mapshape vector.

## Output Arguments

**tf**

`Boolean`. 1 if `s` is empty or 0 if not.

## Examples

### Check if a mapshape vector is empty

Create a mapshape vector and see if it's empty.

```
s = mapshape();
isempty(s)
```

```
ans =

     1
```

# isfield

**Class:** mapshape

True if dynamic property exists

## Syntax

```
tf = isfield(s,name)
tf = isfield(s,names)
```

## Description

`tf = isfield(s,name)` returns true if the value specified by the string `name` is a dynamic property of the mapshape vector, `s`.

`tf = isfield(s,names)` return true for each element of the cell array, `names`, that is a dynamic property of `s`. `tf` is a logical array of the same size as `names`.

## Input Arguments

**s**

mapshape vector.

**name**

Name of the dynamic property.

**names**

Cell array of names of dynamic properties.

## Output Arguments

**tf**

Boolean. 1 if `s` contains the specified fields or 0 otherwise.

## Examples

### Check for Field By Name

Create a mapshape vector and check for a field by name.

```
s = mapshape(-33.961, 18.484, 'Name', 'Cape Town')
isfield(s, 'X')
isfield(s, 'Name')

s =

 1x1 mapshape vector with properties:

 Collection properties:
    Geometry: 'line'
    Metadata: [1x1 struct]
 Vertex properties:
           X: -33.9610
           Y: 18.4840
 Feature properties:
        Name: 'Cape Town'


ans =

     0


ans =

     1
```

Note that X returns 0 because it is not a dynamically added property.

## See Also

mapshape.isprop | mapshape.fieldnames

# isprop

**Class:** mapshape

True if property exists

## Syntax

```
tf = isprop(s,name)
tf = isprop(s,names)
```

## Description

`tf = isprop(s,name)` returns true if the value specified by the string, `name` is a property of the mapshape vector, `s`.

`tf = isprop(s,names)` returns true for each element of the cell array of strings, `names`, that is a property of `s`. `tf` is a logical array the same size as `names`.

## Input Arguments

**s**

mapshape vector.

**name**

String specifying the property of the mapshape vector, `s`.

**names**

Cell array of strings specifying the property of the mapshape vector, `s`.

## Output Arguments

**tf**

Boolean. 1 if the property exists with `s` ,0 otherwise.

## Examples

### Check if Property Exists

Create a mapshape vector and check for properties by name.

```
s = mapshape(-33.961, 18.484, 'Name', 'Cape Town');
isprop(s, 'X')
isprop(s, 'Name')

ans =

     1


ans =

     1
```

## See Also

mapshape.isfield | mapshape.properties

# length

**Class:** mapshape

Number of elements in mapshape vector

## Syntax

```
N = length(s)
```

## Description

`N = length(s)` returns the number of elements contained in the mapshape vector, `s`. The result is equivalent to `size(s,1)`.

## Input Arguments

**s**

mapshape vector.

## Output Arguments

**N**

Length of the mapshape vector, `s`.

## Examples

### Find Length of mapshape Vector

Create a mapshape vector and get its length.

```
coast = load('coast');
```

```
s = mapshape(coast.lat, coast.long);
length(s)
length(coast.lat)

ans =

     1


ans =

       9865
```

## See Also
mapshape.size

# properties

**Class:** mapshape

Properties of a mapshape vector

## Syntax

```
prop = properties(s)
properties(s)
```

## Description

`prop = properties(s)` returns a cell of the property names of the mapshape vector, `s`.

`properties(s)` displays the names of the properties of `s`.

## Input Arguments

**s**

mapshape vector.

## Output Arguments

**prop**

Cell variable consisting of property names of the mapshape vector, `s`.

## Examples

### View Properties of a mapshape Vector

Create mapshape vector and view its properties.

```
s = mapshape(shaperead('tsunamis', 'UseGeo', true));
properties(s)

Properties for class mapshape:

    Geometry
    Metadata
    X
    Y
    Lon
    Lat
    Year
    Month
    Day
    Hour
    Minute
    Second
    Val_Code
    Validity
    Cause_Code
    Cause
    Eq_Mag
    Country
    Location
    Max_Height
    Iida_Mag
    Intensity
    Num_Deaths
    Desc_Deaths
```

## See Also
mapshape.fieldnames

# rmfield

**Class:** mapshape

Remove dynamic property from mapshape vector

## Syntax

```
s = rmfield(s, fieldname)
s = rmfield(s, fields)
```

## Description

`s = rmfield(s, fieldname)` removes the field specified by the string, `fieldname`, from the mapshape vector, `s`.

`s = rmfield(s, fields)` removes all the fields specified by the cell array, `fields`.

---

**Note:** `rmfield` cannot remove X, Y, `Metadata` and `Geometry` fields. The specified string, '`fieldname`, is case sensitive.

---

## Input Arguments

**s**

mapshape vector.

**fieldname**

Exact string representing the name of the property.

**fields**

Cell array of strings specifying the names of the properties.

## Output Arguments

**s**

Updated mapshape vector with the field(s) removed.

## Examples

### Remove Fields from a mapshape Vector

Create mapshape vector and remove a field.

```
s = mapshape(shaperead('tsunamis'));
tf = isfield(s,'Intensity')
s2 = rmfield(s,'Intensity');
tf = isfield(s2,'Intensity')

tf =

     1

tf =

     0
```

## See Also

mapshape.fieldnames | mapshape.rmprop

# rmprop

**Class:** mapshape

Remove properties from mapshape vector

# Syntax

```
sf = rmprop(s,propname)
sf = rmprop(s,propnames)
```

# Description

`sf = rmprop(s,propname)` removes the property specified by the string, `propname` from the mapshape vector, `s`.

`sf = rmprop(s,propnames)` removes all the properties specified in the cell array, `propnames`, from the mapshape vector, `s`. If `propnames` contains a coordinate property an error is issued.

**Note:** `rmprop` cannot remove `X`, `Y`, `Metadata` and `Geometry` fields. The specified string, `propname`, is case sensitive.

# Input Arguments

**s**

mapshape vector.

# Output Arguments

**sf**

Modified mapshape vector with the specified property(s) removed.

## Examples

### Remove a property of a mapshape vector

Create a mapshape vector and remove a property.

```
s = mapshape(shaperead('tsunamis'));
tf = isfield(s,'Validity')
s2 = rmprop(s, 'Validity');
tf = isfield(s2,'Validity')

tf =

     1

tf =

     0
```

## See Also
mapshape.fieldnames

# size

**Class:** mapshape

Size of `mapshape` vector

## Syntax

```
val = size(s)
val = size(s,1)
val = size(s, n)
[m,k] = size(s)
```

## Description

`val = size(s)` returns the vector `[length(val), 1]`.

`val = size(s,1)` returns the length of `s`.

`val = size(s, n)` returns 1 for `n >= 2`.

`[m,k] = size(s)` returns `length(s)` for `m` and 1 for `k`.

## Input Arguments

**s**

mapshape vector.

**n**

Number of the dimension at which size of `s` is required.

## Output Arguments

**val**

Vector of the form `[length(s), 1]`.

**m**

Length of `s`.

**k**

Length of second dimension of `s`. `k` is always `1`.

# Examples

### Size of a mapshape vector

Create a mapshape vector and get its size.

```
structArray = shaperead('worldrivers');
s= mapshape(structArray);
structSize = size(structArray)
sSize = size(s)

structSize =

   128     1


sSize =

   128     1
```

The second dimension is always 1.

## See Also

geoshape.length | `size`

# struct

**Class:** mapshape

Convert mapshape vector to scalar structure

## Syntax

```
structArray = struct(s)
```

## Description

`structArray = struct(s)` converts the mapshape vector, `s`, to a scalar structure array, `structArray`.

## Input Arguments

**s**

mapshape vector.

## Output Arguments

**structArray**

Scalar structure of the mapshape vector `s`.

## Examples

### Convert a mapshape vector into struct

Create a mapshape vector and then convert it into a struct.

```
structArray = shaperead('worldcities')
```

```
s= mapshape(structArray);
structArray2 = struct(s)

structArray =

318x1 struct array with fields:
    Geometry
    X
    Y
    Name


structArray2 =

    Geometry: 'point'
    Metadata: [1x1 struct]
          X: [1x635 double]
          Y: [1x635 double]
       Name: {1x318 cell}
```

## See Also

geoshape.properties

# vertcat

**Class:** mapshape

Vertical concatenation for mapshape vectors

## Syntax

```
s = vertcat(s1,s2, ...)
```

## Description

`s = vertcat(s1,s2, ...)` vertically concatenates the mapshape vector, `s1`, `s2`, and so on. If the class type of any property is a cell array, then the resultant field in the output `s` will also be a cell array.

## Input Arguments

**s1, s2, ...**

mapshape vectors to be concatenated.

## Output Arguments

**s**

Concatenated mapshape vector.

## Examples

### Concatenate mapshape vectors

Create two mapshape vectors and concatenate them.

```
s1 = mapshape(42, -110, 'Temperature', 65, 'Name', 'point1');
s2 = mapshape( 42.1, -110.4, 'Temperature', 65.5, 'Name', 'point2');
s = vertcat(s1, s2)

s =

 2x1 mapshape vector with properties:

 Collection properties:
       Geometry: 'line'
       Metadata: [1x1 struct]
 Vertex properties:
  (2 features concatenated with 1 delimiter)
              X: [42 NaN 42.1000]
              Y: [-110 NaN -110.4000]
 Feature properties:
    Temperature: [65 65.5000]
           Name: {'point1' 'point2'}
```

## See Also

mapshape.cat

# mapshow

Display map data without projection

## Syntax

```
mapshow(x,y)
mapshow(s)
mapshow(x,y,z)
mapshow(Z,R)
mapshow(x,y,image)
mapshow(x,y,A,cmap)
mapshow(image,R)
mapshow(A,cmap,R)
mapshow(ax,...)
mapshow('Parent',ax,...)
mapshow(filename)
mapshow( ___ ,name,value,...)
h = mapshow(...)
```

## Description

mapshow(x,y) displays the coordinate vectors x and y. The default behavior for mapshow is to display x and y as lines.

mapshow(s) displays the vector geographic features stored in the geographic data structure s as points, multipoints, lines, or polygons according to the Geometry field of s. If s includes X and Y fields, then these fields are used directly to plot features in map coordinates. If s includes Lat and Lon fields instead, the coordinates are projected with the Plate Carrée projection and a warning is issued.

mapshow(x,y,z) displays a geolocated data grid. x and y are *M*-by-*N* coordinate arrays, and z is an *M*-by-*N* array of class double.

mapshow(Z,R) displays a regular data grid, Z.

mapshow(x,y,image) or mapshow(x,y,A,cmap) displays a geolocated image as a texturemap on a zero-elevation surface. x and y are geolocation arrays in map

coordinates. `x`, `y`, and the image array must match in size. Examples of geolocated images include a color composite from a satellite swath or an image originally referenced to a different coordinate system.

`mapshow(image,R)` or `mapshow(A,cmap,R)` displays an image georeferenced to map coordinates through `R`. The `mapshow` function constructs an image object if the display geometry permits. Otherwise, `mapshow` displays the image as a texturemap on a zero-elevation surface.

`mapshow(ax,...)` and `mapshow('Parent',ax,...)` set the axes parent to `ax`.

`mapshow(filename)` displays data from the file specified according to the type of file format.

`mapshow( ___ ,name,value,...)` specifies parameters and corresponding values that modify the type of display or set MATLAB graphics properties. Parameter names can be abbreviated, and case does not matter.

`h = mapshow(...)` returns a handle to a MATLAB graphics object.

## Input Arguments

**x**

Coordinate vector, M-by-N coordinate array, or geolocation array in map coordinates, depending on the syntax. `x` can contain embedded NaNs to delimit individual lines or polygon parts.

**y**

Coordinate vector, M-by-N coordinate array, or geolocation array in map coordinates, depending on the syntax. `y` can contain embedded NaNs to delimit individual lines or polygon parts.

**z**

M-by-N array. `z` can contain NaN values.

**s**

Geographic data structure.

**Z**

Regular data grid.

**R**

Referencing matrix or an object of class map.rasterref.MapCellsReference or map.rasterref.MapPostingsReference that relates the subscripts of Z to map coordinates. If R is a `map.rasterref.MapRasterReference` object with raster interpretation `'postings'`, then `mapshow` does not accept the `'image'` and `'texturemap'` display types.

**image**

Grayscale, logical, or truecolor image.

**A**

Indexed image.

**cmap**

Colormap.

**ax**

Axes object.

**filename**

Name of file.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'SymbolSpec'**

A structure returned by `makesymbolspec` that specifies the symbolization rules used for vector data. When both `SymbolSpec` and one or more graphics properties are specified, the graphics properties will override any settings in the symbolspec structure.

To change the default symbolization rule for a Name,Value pair in the symbolspec, prefix the word `'Default'` to the graphics property name.

**`'DisplayType'`**

Type of graphic display for the data. You can set any MATLAB Graphics line, patch, image, surface, and contour properties. See the table for links to the MATLAB documentation on these properties.

| Data Type | DisplayType | Type of Property |
|---|---|---|
| Vector | `'point'` | *line marker* |
| | `'multipoint'` | *line marker* |
| | `'line'` (default) | *line* |
| | `'polygon'` | *patch* |
| Image | `'image'` or `'surface'` | *image* or *surface* |
| Grid | `'surface'` | *surface* |
| | `'mesh'` | *surface* |
| | `'texturemap'` | *surface* |
| | `'contour'` | *contour* |

If `DisplayType` is `'texturemap'`, `mapshow` constructs a surface with `ZData` values set to 0.

Set the `DisplayType` to `'image'` if you are using the syntax `mapshow(image, R, ...)`.

When using the `filename` argument, the `DisplayType` parameter is automatically set, according to the following table:

| Format | DisplayType |
|---|---|
| Shapefile | `'point'`, `'line'`, or `'polygon'` |
| GeoTIFF | `'image'` |
| TIFF/JPEG/PNG with a world file | `'image'` |

| Format | DisplayType |
|---|---|
| ARC ASCII GRID | `'surface'` (can be overridden) |
| SDTS raster | `'surface'` (can be overridden) |

## Output Arguments

**h**

Handle to a MATLAB graphics object or, in the case of polygons, a modified patch object. If a mapstruct or shapefile name is input, `mapshow` returns the handle to an hggroup object with one child per feature in the mapstruct or shapefile. In the case of a polygon mapstruct or shapefile, each child is a modified patch object; otherwise it is a line object.

## Class Support

| Display Type | Supported Class Types |
|---|---|
| Image | `logical`, `uint8`, `uint16`, and `double` |
| Surface | `single` and `double` |
| Texture map | All numeric types and `logical` |

## Examples

**Overlay Vector on Orthophoto**

Overlay Boston roads on an orthophoto. Note that `mapshow` draws a new layer in the axes rather than replacing its contents.

Display image.

```
figure
mapshow boston.tif
axis image off manual
```

Convert Boston roads to units of survey feet and overlay on orthophoto.

```
S = shaperead('boston_roads.shp');
surveyFeetPerMeter = unitsratio('sf','meter');
x = surveyFeetPerMeter * [S.X];
y = surveyFeetPerMeter * [S.Y];
mapshow(x,y)
```

**Display Vector Data Customizing Line Style**

Read the vector data and display it using a dotted line.

```
roads = shaperead('boston_roads.shp');
figure
mapshow(roads,'LineStyle',':');
```
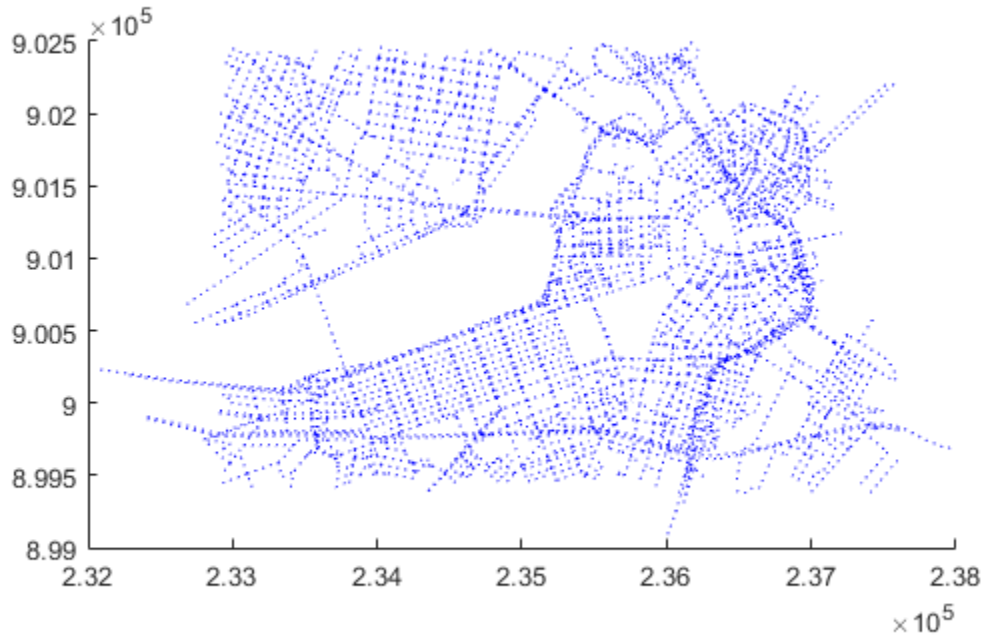
**Display Vector Data Using Symbol Specification**

Create a symbol specification to distinguish between different types of roads. For example, you can hide very minor roads (CLASS=6) by turning off their visibility and make major roads (CLASS=1-4) more visible by increasing their line widths. This symbol specification also uses color to distinguish between types of roads.

```
roadspec = makesymbolspec('Line',...
                          {'ADMIN_TYPE',0,'Color','cyan'}, ...
                          {'ADMIN_TYPE',3,'Color','red'},...
                          {'CLASS',6,'Visible','off'},...
                          {'CLASS',[1 4],'LineWidth',2});
```

Display the vector data using the symbol specification.

```
figure
mapshow('boston_roads.shp','SymbolSpec',roadspec);
```
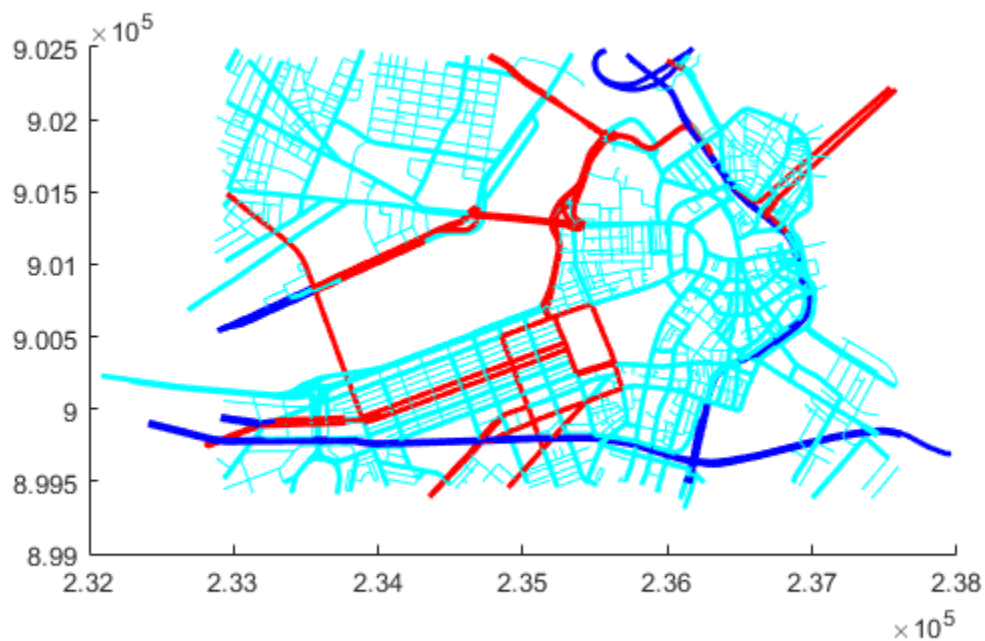


### Override Default Properties of the Line

Create a symbol specification and specify the default color used for lines. As seen in the previous example, the default is blue. This example sets the default to black.

```
roadspec = makesymbolspec('Line',...
                         {'Default', 'Color', 'black'}, ...
                         {'ADMIN_TYPE',0,'Color','c'}, ...
                         {'ADMIN_TYPE',3,'Color','r'},...
                         {'CLASS',6,'Visible','off'},...
                         {'CLASS',[1 4],'LineWidth',2});
```

Display the vector data, using the symbol specification. Note how the major roads displayed in blue in the previous example are now black.

```
figure
mapshow('boston_roads.shp','SymbolSpec',roadspec);
```
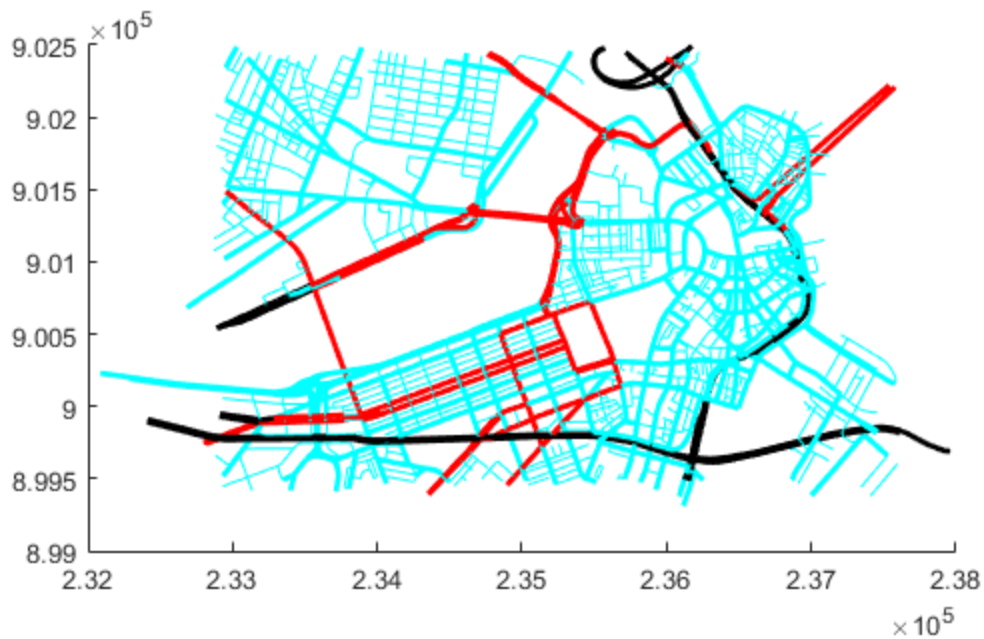


### Override Symbol Specification on Command Line

Create a symbol specification, setting various properties.

```
roadspec = makesymbolspec('Line',...
                          {'Default', 'Color', 'yellow'}, ...
                          {'ADMIN_TYPE',0,'Color','c'}, ...
                          {'ADMIN_TYPE',3,'Color','r'},...
```

```
                                    {'CLASS',6,'Visible','off'},...
                                    {'CLASS',[1 4],'LineWidth',2});
```

Display the vector data, specifying the color on the command line.

```
figure
mapshow('boston_roads.shp', 'Color', 'black', 'SymbolSpec', roadspec);
```
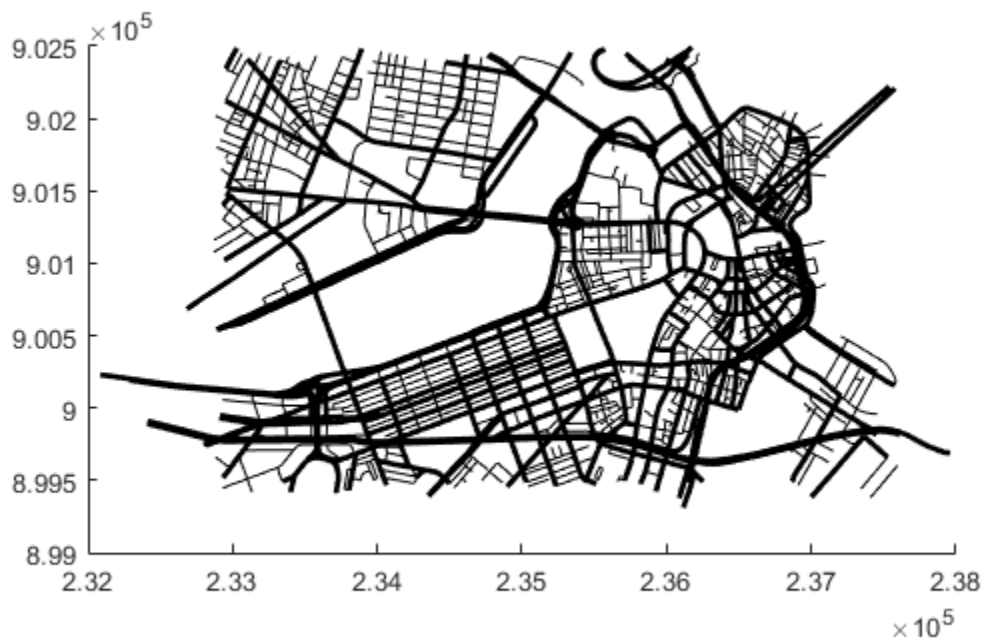


### Overlay Polygon and Lines on Orthophoto

Read orthophoto.

```
[ortho, cmap] = imread('concord_ortho_w.tif');
```

Read vector data.

```
R = worldfileread('concord_ortho_w.tfw', 'planar', size(ortho));
```

Display the orthophoto with the vector data.

```
figure
mapshow(ortho,cmap,R)
```
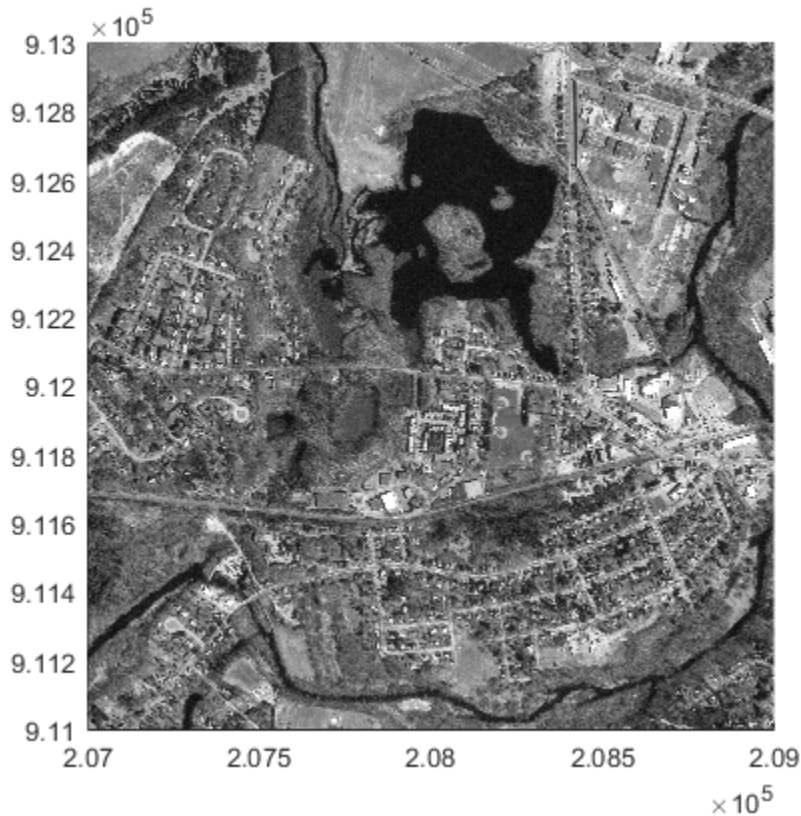


Read polygon vector data representing a pond with three islands (feature 14 in the file). Note that the islands are visible in the orthophoto throught three "holes" in the pond polygon.

```
pond = shaperead('concord_hydro_area.shp', 'RecordNumbers', 14);
mapshow(pond, 'FaceColor', [0.3 0.5 1], 'EdgeColor', 'black')
```
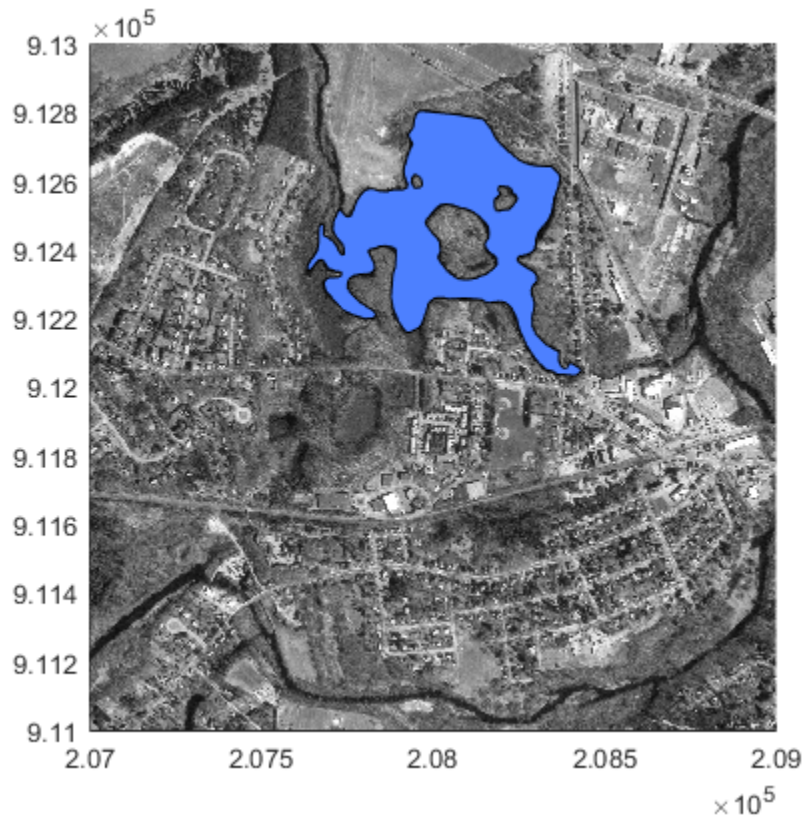
Overlay road vector data in the same figure.

```
mapshow('concord_roads.shp', 'Color', 'red', 'LineWidth', 1);
```
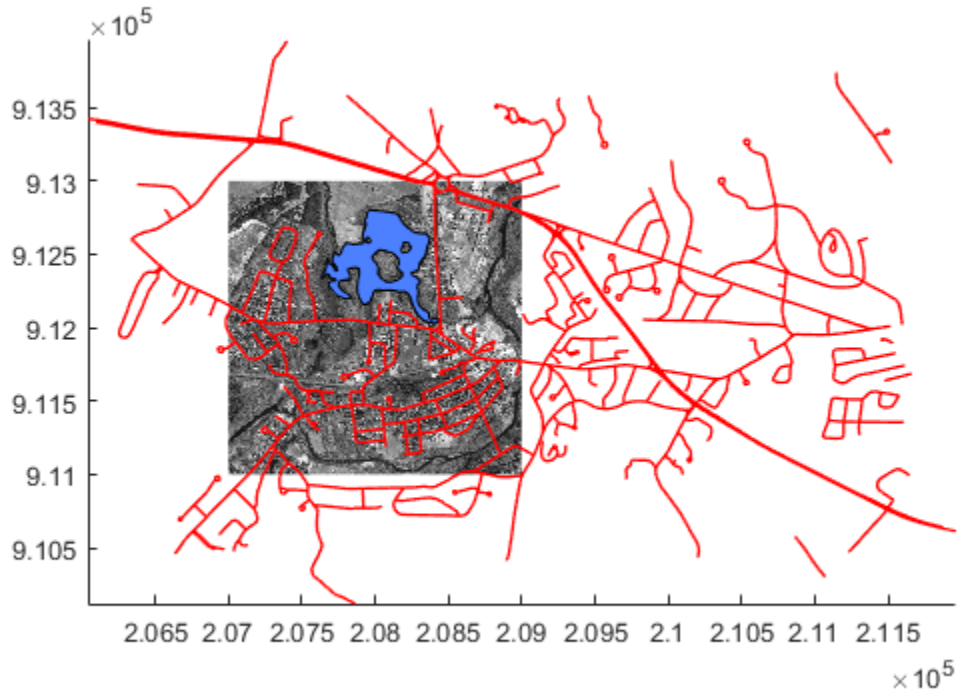
**Display Terrain Data as Mesh, Surface, and 3-D Surface**

Read SDTS DEM terrain data for Mount Washington.

```
[Z, R] = sdtsdemread('9129CATD.DDF');
```

View the terrain data as a mesh.

```
figure
mapshow(Z, R, 'DisplayType', 'mesh');
demcmap(Z)
```
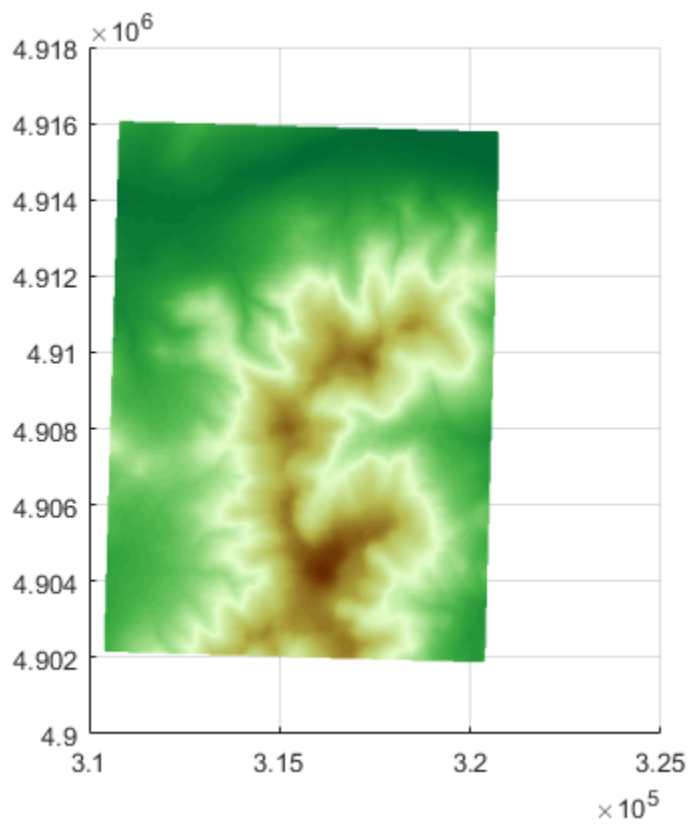
View the terrain data as a surface.

```
figure
mapshow(Z, R, 'DisplayType', 'surface');
demcmap(Z)
```
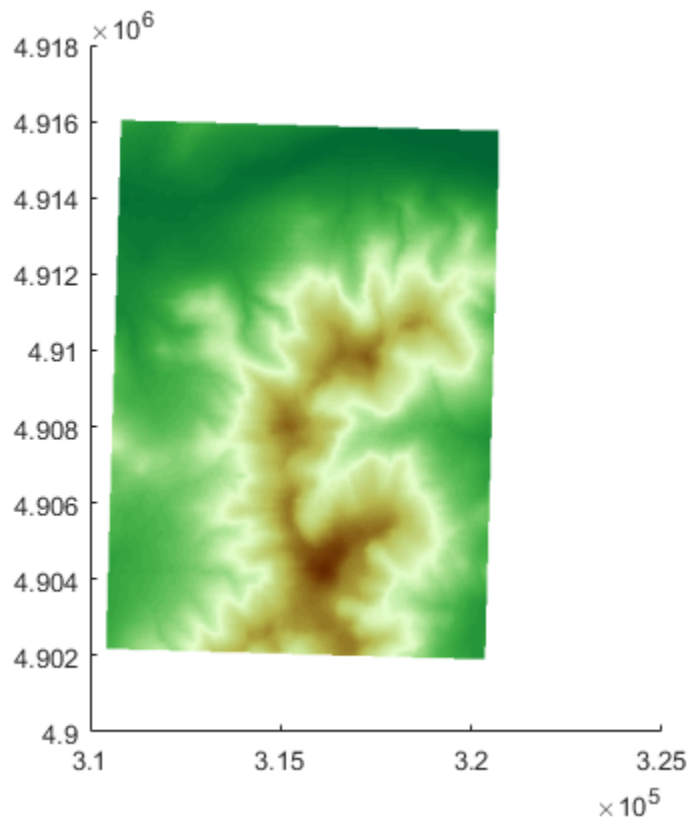
View the terrain data as a 3-D surface.

```
view(3);
axis normal
```
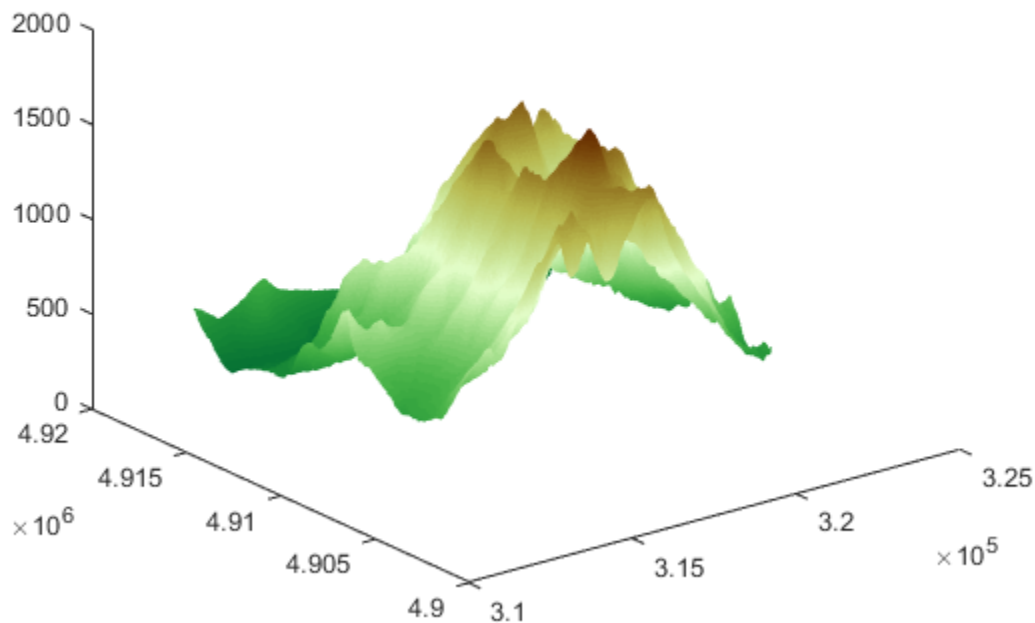
### Display Grid and Contour Lines

Read the terrain data files for Mount Washington and Mount Dartmouth.

```
[Z_W, R_W] = arcgridread('MtWashington-ft.grd');
[Z_D, R_D] = arcgridread('MountDartmouth-ft.grd');
```

Display the terrain data as a surface in the z == 0 plane, so that overlying contour lines and labels are visible.

```
figure
hold on
mapshow(zeros(size(Z_W)),R_W,'CData',Z_W,'DisplayType','surface')
mapshow(zeros(size(Z_D)),R_D,'CData',Z_D,'DisplayType','surface')
```

```
axis equal
```
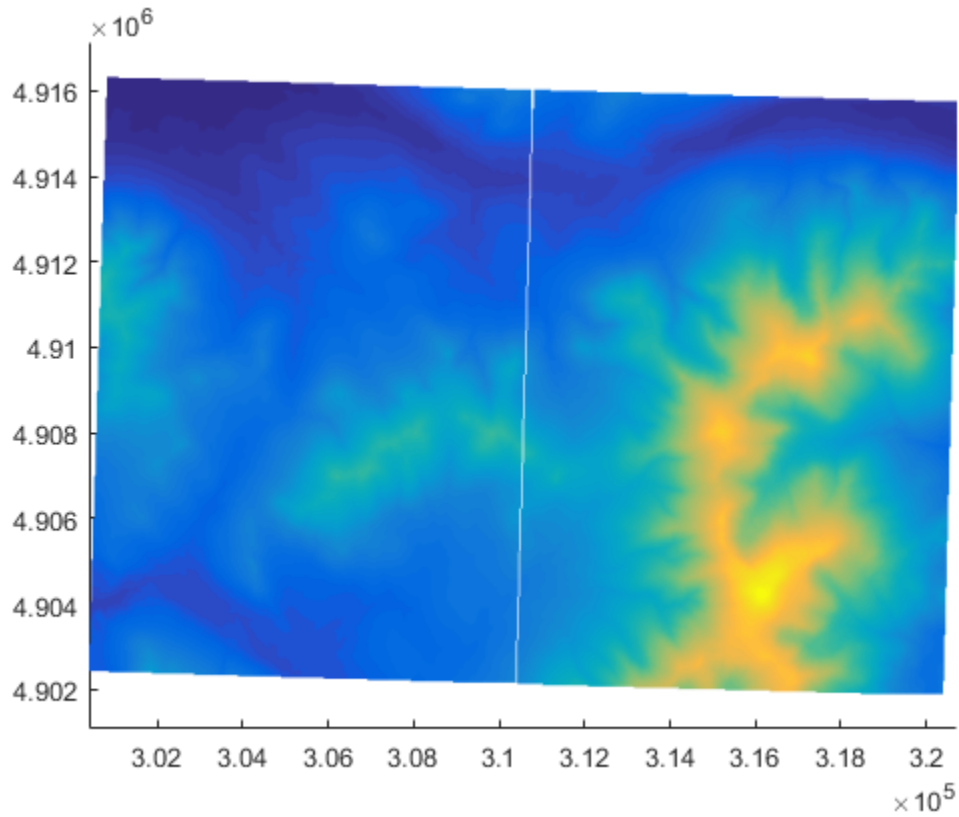


Overlay black contour lines and labels.

```
cW = mapshow(Z_W, R_W, 'DisplayType', 'contour', ...
    'LineColor','black', 'ShowText', 'on');
cD = mapshow(Z_D, R_D, 'DisplayType', 'contour', ...
    'LineColor','black', 'ShowText', 'on');
```
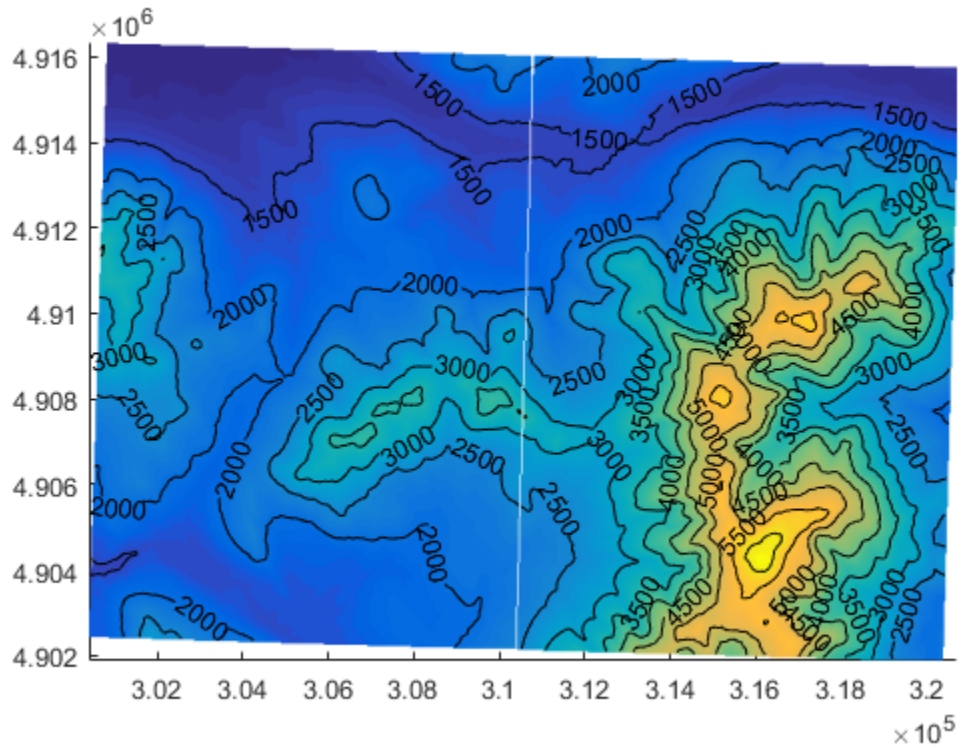
Set the colormap appropriate to terrain elevation.

```
demcmap(Z_W)
```
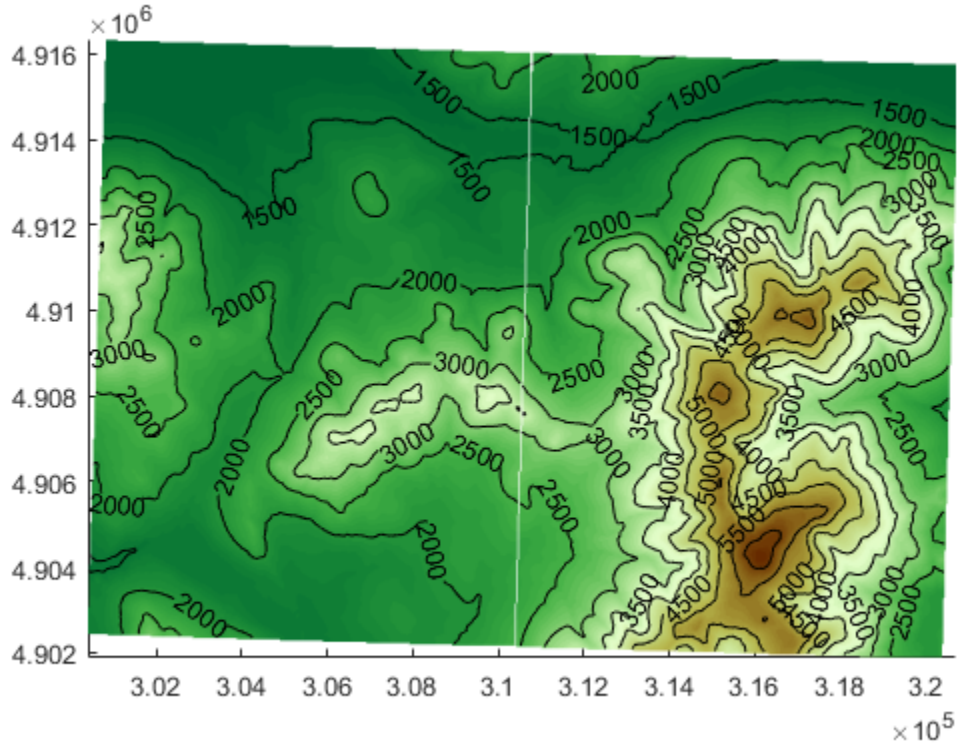
## More About

**Tips**

- If you do not want `mapshow` to draw on top of an existing map, create a new figure or subplot before calling it.

- You can use `mapshow` to display vector data in an `axesm` figure. However, you should not subsequently change the map projection using `setm`.

- If you display a polygon, do not set `'EdgeColor'` to either `'flat'` or `'interp'`. This combination may result in a warning.

- If `s` is a geostruct (has `Lat` and `Lon` fields), it may be more appropriate to use `geoshow` to display them. You can project latitude and longitude coordinate values to map coordinates by displaying with `geoshow` on a map axes.

## See Also

map.rasterref.MapCellsReference | map.rasterref.MapPostingsReference | `geoshow` | `makesymbolspec` | `mapview` | `shaperead`

# maptriml

Trim lines to latitude-longitude quadrangle

## Syntax

```
[lat,lon] = maptriml(lat0,lon0,latlim,lonlim)
```

## Description

`[lat,lon] = maptriml(lat0,lon0,latlim,lonlim)` returns *filtered* `NaN`-delimited vector map data sets from which all points lying outside the desired latitude and longitude limits have been discarded. These limits are specified by the two-element vectors `latlim` and `lonlim`, which have the form `[south-limit north-limit]` and `[west-limit east-limit]`, respectively.

## Examples

Following is a simple example:

```
lat0 = [1:10,9:-1:0]; lon0 = -30:-11;
[lat,lon] = maptriml(lat0,lon0,[3 7],[-29 -12]);
[lat lon]

ans =
    NaN    NaN
      3    -28
      4    -27
      5    -26
      6    -25
      7    -24
    NaN    NaN
      7    -18
      6    -17
      5    -16
      4    -15
      3    -14
    NaN    NaN
```

Notice that trimmed line segment ends have NaNs inserted at trim points.

## See Also
maptrimp | maptrims

# maptrimp

Trim polygons to latitude-longitude quadrangle

## Syntax

```
[latTrimmed,lonTrimmed] = maptrimp(lat,lon,latlim,lonlim)
```

## Description

[latTrimmed,lonTrimmed] = maptrimp(lat,lon,latlim,lonlim) trims the polygons in lat and lon to the quadrangle specified by latlim and lonlim. latlim and lonlim are two-element vectors, defining the latitude and longitude limits respectively. lat and lon must be vectors that represent valid polygons.

## Examples

### Trim Dataset to Specific Geographic Area

Read coastline data and display it on a map.

```
coast = load('coast.mat');
figure
mapshow(coast.long, coast.lat, 'DisplayType', 'polygon');
```
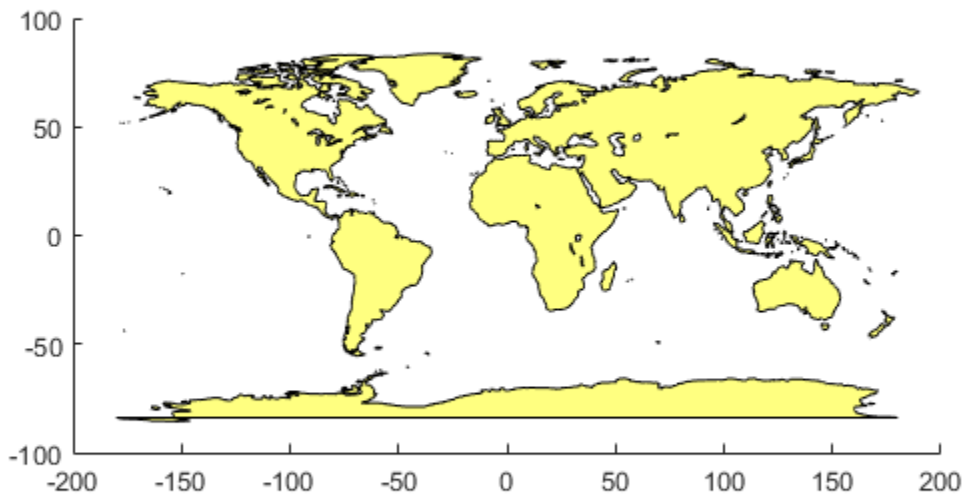
Trim the dataset.

```
latlim = [-50 50];
lonlim = [-100 50];
[latTrimmed, lonTrimmed] = maptrimp(coast.lat, coast.long, ...
    latlim, lonlim);
```

Display the trimmed dataset.

```
figure
mapshow(lonTrimmed, latTrimmed, 'DisplayType', 'polygon');
```
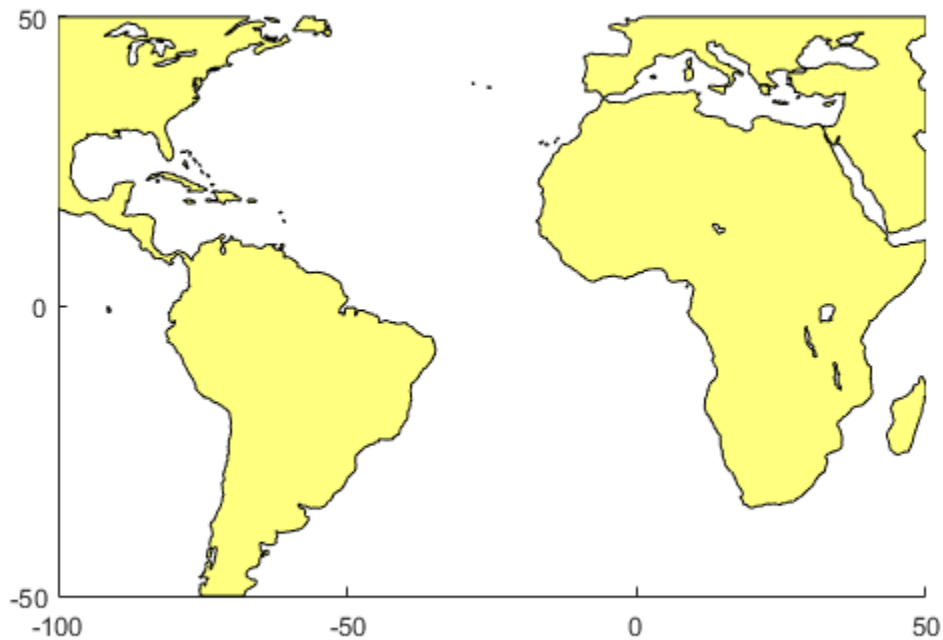
## More About

**Tips**

`maptrimp` conditions the longitude limits such that:

- `lonlim(2)` always exceeds `lonlim(1)`
- `lonlim(2)` never exceeds `lonlim(1)` by more than `360`
- `lonlim(1) < 180` or `lonlim(2) > -180`
- Ensure that if the quadrangle span the Greenwich meridian, then that meridian appears at longitude `0`.

## See Also

maptriml | maptrims

# maptrims

Trim regular data grid to latitude-longitude quadrangle

## Syntax

```
[Z_trimmed] = maptrims(Z,R,latlim,lonlim)
[Z_trimmed] = maptrims(Z,R,latlim,lonlim,cellDensity)
[Z_trimmed, R_trimmed] = maptrims(...)
```

## Description

`[Z_trimmed] = maptrims(Z,R,latlim,lonlim)` trims a regular data grid `Z` to the region specified by `latlim` and `lonlim`. By default, the output grid `Z_trimmed` has the same sample size as the input. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix. If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)` and its `RasterInterpretation` must be `'cells'`.

If `R` is a referencing vector, it must be a 1-by-3 vector with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```
If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```
If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. `latlim` and `lonlim` are two-element vectors, defining the latitude and longitude limits, respectively. The `latlim` vector has the form:

```
[southern_limit northern_limit]
```

Likewise, the `lonlim` vector has the form:

```
[western_limit eastern_limit]
```

When an individual value in `latlim` or `lonlim` corresponds to a parallel or meridian that runs precisely along cell boundaries, the output grid will extend all the way to

that limit. But if a limiting parallel or meridian cuts through a column or row of input cells, then the limit will be adjusted inward. In other words, the requested limits will be truncated as necessary to avoid partial cells.

`[Z_trimmed] = maptrims(Z,R,latlim,lonlim,cellDensity)` uses the scalar `cellDensity` to reduce the size of the output. If `R` is a referencing vector, then `R(1)` must be evenly divisible by `cellDensity`. If `R` is a referencing matrix, then the inverse of each element in the first two rows (containing `"deltaLat"` and `"deltaLon"`) must be evenly divisible by `cellDensity`.

`[Z_trimmed, R_trimmed] = maptrims(...)` returns a referencing vector, matrix, or object for the trimmed data grid. If `R` is a referencing vector, then `R_trimmed` is a referencing vector. If `R` is a referencing matrix, then `R_trimmed` is a referencing matrix. If `R` is a geographic raster reference object, then `R_trimmed` is either a geographic raster reference object (when `Z_trimmed` is non-empty) or `[]` (when `Z_trimmed` is empty).

# Examples

```
load topo
[subgrid,subrefvec] = maptrims(topo,topolegend,...
                               [80.25 85.3],[165.2 170.7])

subgrid =
       -2826        -2810        -2802        -2793
       -2915        -2913        -2905        -2884
       -3192        -3186        -3165        -3122
       -3399        -3324        -3273        -3214

subrefvec =
     1    85    166
```

The upper left corner of the grid might differ slightly from that of the requested region. `maptrims` uses the corner coordinates of the first cell inside the limits.

# See Also

`maptriml` | `maptrimp` | `resizem`

# mapview

Interactive map viewer

## Syntax

```
mapview
```

## Description

`mapview` opens the Map Viewer app in an empty state. Use the Map Viewer to view geospatial data in map (*x-y*) coordinates. The Map Viewer works with vector, image, and raster data grids in a map coordinate system. You can load data, pan and zoom on the map, control the map scale of your screen display, control the order, visibility, and symbolization of map layers, annotate your map, and click to learn more about individual vector features. `mapview` complements `mapshow` and `geoshow`, which are for constructing maps in ordinary figure windows in a less interactive, script-oriented way.

For usage information, see the following sections. You can also work through the Map Viewer tutorial, "Tour Boston with the Map Viewer App".

## Importing Data

The Map Viewer opens with no data loaded and an empty map display window. The first step is to import a data set. Use the options in the **File** menu to select data from a file or from the MATLAB workspace:

### Import From File

Use the file browsing dialog to open a file in one of the following formats: Shapefile, GeoTIFF, SDTS DEM, Arc ASCII Grid, TIFF, JPEG, or PNG with world file. This option imports the data into the viewer but does not add it to your workspace.

To view standard-format geodata files provided with the toolbox, set your working folder or navigate the Map Viewer Open dialog to

*matlabroot*/toolbox/map/mapdata

## Import From Workspace

**Images.**    Use the **Raster Data** > **Image** import dialog to select a **Referencing matrix or object name** and **Raster data name** for the image from the list of workspace variables. If the image type is truecolor (RGB), specify which band represents the red, green, and blue intensities. (The RasterInterpretation of the referencing object must be 'cells'.)

**Data grids.**    Use the **Raster Data** > **Grid** import dialog to select X and Y geolocation and data grid array names from the list of workspace variables.

**Vector data.**    Use the **Vector Data** > **Map coordinates** import dialog to select X and Y variables for map coordinates from the list of workspace variables and identify the type of geometry to be displayed (**Point**, **Line**, or **Polygon**). The X and Y variables can specify multiple line segments or multiple polygons if they contain NaNs at matching locations in the coordinate vectors.

**Vector geographic data structure.**    Use the **Vector Data** > **Geographic data structure** import dialog to select the struct that contains vector map data from the list of workspace variables.

Once you import your first data set, the Map Viewer automatically sets the limits of its map display window to the spatial extent of the imported data.

## Working in Map Coordinates

As you move any of the Map Viewer cursors across the map display area, the coordinate readout in the lower left corners shows you the cursor position in map X and Y coordinates.

The Map Viewer requires that all currently viewed data sets possess the same coordinate system and length units. This is likely to be the case for data sets that originated from a common source. If it is not the case, you will need to adjust coordinates before importing data into the Map Viewer.

If some or all of your data is in geographic coordinates, use projfwd or mfwdtran to project latitudes and longitudes to your desired map coordinate system before you import

it. When starting from a different projection, you must first unproject to latitude and longitude using `projinv` or `minvtran`, then reproject with `projfwd` or `mfwdtran`. You might also need to adjust the horizontal datum of your data using, for example, the free GEOTRANS (Geographic Translator) application from the Geospatial Sciences Division of the U.S. National Geospatial-Intelligence Agency (NGA). If you simply need a change of units, multiply by the appropriate conversion factor obtained from `unitsratio`.

`mapview` can also display data in unprojected geographic coordinates, if you consistently substitute longitude for map X and latitude for map Y. Geographic coordinates must be consistently expressed in either degrees or radians (not both at once). When using geographic coordinates, do not specify the viewer's map units (see below); you can only use the Map Viewer's map scale display when working in linear units of length.

## Setting Map Units and Scale

If you tell the Map Viewer which length unit you are using, it can calculate an approximate map scale for your onscreen display. Set the map units with either the drop-down menu at the bottom of the display or the **Set Map Units** item in the **Tools** menu.

The scale computed by the Map Viewer is displayed in the window just above the map units drop-down. To change your display scale while keeping the center of the map display fixed, simply edit this text box.

Make sure to format your text in the standard way (1:*N*, where *N* is a positive number such that a distance on the ground is *N* times the same distance on your screen, e.g., `1:24000`).

The scale is approximate because it depends on the MATLAB estimate of the size of your screen pixels. It is also approximate if your projection introduces significant distortion. If your data falls in a fairly small area and you use a conformal projection (e.g., UTM with all data in a single zone), the scale will be very consistent across your entire map.

## Navigating Your Map

By default, the Map Viewer sets the limits of your map window to match the extent of the first data set that you load. You will probably want to adjust this to see some areas in greater detail.

The Map Viewer provides several tools to control the limits of your map window and the map scale of the data display. Some are familiar from standard MATLAB figure windows.

- **Zoom in**: Drag a box to zoom in on a specific area or click a point to zoom in with that point centered in the map display.

- **Zoom out**: Click a point to zoom out with that point centered in the map display.

- **Pan tool**: Click, hold, and drag to reposition the selected point in the display window, while holding the map scale fixed. Release when you are satisfied with new display limits.

- **Fit to window**: Set the map display to enclose all currently loaded data layers. This is equivalent to selecting **Fit to Window** in the **View** menu.

- **Back to previous view**: Click this button once to return the map scale and display center to their values prior to the most recent zoom, pan, or scale change. Click repeatedly to undo earlier changes. This is equivalent to selecting **Previous View** in the **View** menu.

Another way to zoom in or out while keeping the center of the view fixed at the same map coordinates is to directly edit the map scale box at the bottom of the screen.

## Managing Map Layers

Each time you import a set of vectors, an image, or a data grid into the Map Viewer, the new data is stored in a new map layer. The layers form an ordered stack. Each layer is listed as an item in the **Layers** menu, with its position in the menu indicating its position in the stack.

When you import a new layer, the Map Viewer automatically places it at the top of the layer stack. To reposition a layer in the stack, select it in the **Layers** menu, slide right, and select **To Top**, **To Bottom**, **Move Up**, or **Move Down** from the pop-up submenu.

The vector features or raster in a given layer obscure coincident elements of any underlying layers. To control layers that are obscuring one another, you can also toggle layer visibility on and off. Use the item **Visible** in the slide-right menu. Or, simply remove a layer from the Map Viewer via the **Remove** item in the slide-right menu. Remember that even if a layer's visibility is *on*, the layer does not appear if its contents are located completely outside the current display limits or are obscured by another layer.

# Symbolizing Vector Features

When point, line, and polygon layers are loaded, the Map Viewer initializes their graphics properties as follows:

| Geometry | Properties |
|---|---|
| Point (line objects) | `LineStyle = 'none`<br>`Marker = 'x'`<br>`MarkerEdgeColor` = \<randomly generated value\><br>`MarkerFaceColor = 'none'` |
| Line (line objects) | `Color` = \<randomly generated value\><br>`LineStyle = '-'`<br>`Marker = 'none'` |
| Polygon (patch objects) | `EdgeColor = [0 0 0] FaceColor` = \<randomly generated value\> |

To override symbolism defaults for a vector layer, use `makesymbolspec` to create a symbol specification in the workspace. A symbolspec contains a set of rules for setting vector graphics properties based on the values of feature attributes. For instance, if you have a line layer representing roads of various classes (e.g., major highway, secondary road, etc.), you can create a symbolspec to use a different color, line width, or line style for each road class. See the `makesymbolspec` help for examples and to learn how to construct a symbolspec. If you regularly work with data sets sharing a common set of feature attributes, you might want to save one or more symbolspecs in a MAT-file (or save calls to `makesymbolspec` in a MATLAB program file).

Once you have a symbolspec in your workspace, select your vector layer in the **Layers** menu, then slide right and click **Set Symbol Spec**, which opens a dialog box. Use the dialog box to select the symbolspec from your workspace.

# Getting Information About Vector Features

The **Datatip** tool and the **Info** tool provide different ways to check the attributes of vector features that you select graphically. Before using either tool you must designate one of your vector layers as *active*. (The default active layer is the first one that you imported.) Either use the **Active Layer** drop-down menu at the bottom of your screen or select the layer in the **Layers** menu, slide right, and select **Active**. Having a designated

active layer ensures that when you click a feature you don't inadvertently select an overlapping feature from a different layer.

- **Datatip tool**: The **Datatip** tool displays a feature attribute in a text label each time you click a vector feature. By default the attribute is the first one in the layer's attribute list. To change which attribute is used, select the layer in the **Layers** menu, slide right, and click `Set Layer Attribute`. In the dialog that follows, select a different attribute, or `Index`. If you choose `Index`, the Map Viewer displays the one-based index value corresponding to a given feature—based on its position in the input file or workspace array. To remove a text label, right-click it and choose **Delete datatip** from the context menu. Or choose **Delete all datatips** from the context menu or the **Tools** menu.

- **Info tool**: The **Info** tool opens a separate text window each time you click a vector feature. The window displays all the attribute names and values for that feature, in contrast to the **Datatip** tool, which displays only the value of a single attribute. If you need to compare two or more features, simply click each one and view the info windows together. Use its close button to close an info window when you're done with it, or choose **Close All Info Windows** from the **Tools** menu.

## Annotating Your Map

Use the **text**, **line**, or **arrow** annotation tools to mark and highlight points of interest on your map, or select the corresponding items in the **Insert** menu. Note that to insert an additional object of the same type, you must reselect the appropriate tool. Use the **Select annotations** tool and **Edit** menu to modify or remove your annotations. The Map Viewer manages annotations separately from data layers; annotations always stay on top. Note that annotations cannot be saved as graphic objects, although you can export maps containing annotations to an image format as described below.

## Creating and Using Additional Views

Use **New View** on the **File** menu to create an additional Map Viewer window linked to an existing window. Consider using an additional window when you want to see your map at different scales at the same time (e.g., a detailed view plus an overview), or when you want to simultaneously see different areas of the map at large scale. You can create as many additional windows as you need, and close them when you want. Your `mapview` session ends when you close the last window.

Options for creating a new viewer window include: **Duplicate Current View**, **Full Extent**, **Full Extent of Active Layer**, and **Selected Area**. Click and drag with the **Select area** tool to define a selected area.

A new viewer window differs from existing windows mainly in terms of the visible map extent and scale (it also omits annotations and any labels you added with the datatip tool). You will see the same layers in the same order with the same settings (including the active layer). Updates to layers (insertion/removal, order, visibility, label attribute, and symbolization) in one viewer window are propagated automatically to all the windows with which it is linked. Updates to annotations and datatip labels are not propagated between viewers. If you need two different layer configurations in different windows, launch a second `mapview` from the command line instead of creating an additional window. The views it contains will not be linked to previous ones.

## Exporting Your Map

The Map Viewer allows you to export all or part of your map for use in a publication or on a Web page. Use **File** > **Save As Raster Map** to export an image of either the current display extent or an area outlined with the **Select area** tool. Select a format (PNG, TIFF, JPEG) from the drop-down menu in the export dialog. For maps including vector layers, PNG (Portable Network Graphics) is often the best choice. This format provides excellent quality, good compression, and is well supported by modern Web browsers. The export process automatically creates a world file (ending with suffix `tfw`, `jgw`, or `pgw`) as well; the pair of files constitute a georeferenced image that itself can be displayed with `mapview`, `mapshow`, and many external GIS packages.

### See Also
arcgridread | geoshow | geotiffread | makesymbolspec | mapshow | sdtsdemread | shaperead | updategeostruct | worldfileread

# mdistort

Display contours of constant map distortion

## Syntax

```
mdistort
mdistort off
mdistort(parameter)
mdistort parameter
mdistort(parameter,levels)
mdistort(parameter,levels,gsize)
h = mdistort(...)
```

## Description

mdistort, with no input arguments, toggles the display of contours of projection-induced distortion on the current map axes. The magnitude of the distortion is reported in percent.

mdistort off removes the contours.

mdistort(*parameter*) or mdistort *parameter* displays contours of distortion for the specified parameter. Recognized *parameter* strings are 'area', 'angles' for the maximum angular distortion of right angles, 'scale' or 'maxscale' for the maximum scale, 'minscale' for the minimum scale, 'parscale' for scale along the parallels, 'merscale' for scale along the meridians, and 'scaleratio' for the ratio of maximum and minimum scale. If omitted, the 'maxscale' parameter is displayed. All parameters are displayed as percent distortion except angles, which are displayed in degrees.

mdistort(*parameter*,levels) specifies the levels for which the contours are drawn. levels is a vector of values as used by contour. If empty, the default levels are used.

mdistort(*parameter*,levels,gsize) controls the size of the underlying graticule matrix used to compute the contours. gsize is a two-element vector containing the number of rows and columns. If omitted, the default Mapping Toolbox graticule size of [50 100] is assumed.

`h = mdistort(...)` returns a handle to the contourgroup object containing the contours and text.

## Background

Map projections inevitably introduce distortions in the shape and size of objects as they are transformed from three-dimensional spherical coordinates to two-dimensional Cartesian coordinates. The amount and type of distortion vary between projections, over the projection, and with the selection of projection parameters such as standard parallels. This function provides a quantitative graphical display of distortion parameters.

`mdistort` is not intended for use with UTM. Distortion is minimal within a given UTM zone. `mdistort` issues a warning if a UTM projection is encountered.

## Examples

### Example 1

Note the extreme area distortion of the Mercator projection. This makes it ill-suited for global displays.

```
figure
axesm mercator
load coast
framem;
plotm(lat, long,'color',.5*[1 1 1])
mdistort('area', [1 30 100 200 500 1000])
```
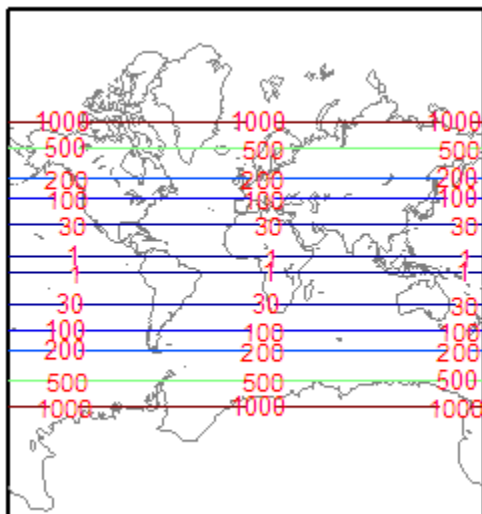
## Example 2

The lines of zero distortion for the Bonne projection follow the central meridian and the standard parallel.

```
figure
axesm bonne
load coast
framem;plotm(lat, long,'color',.5*[1 1 1])
mdistort('angles', 0:15:90)
parallelui
```
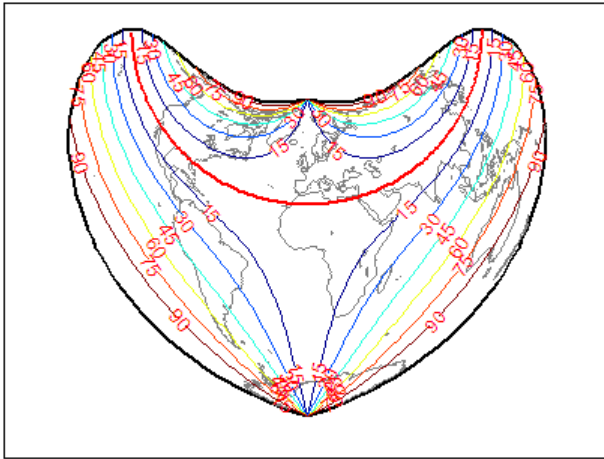
## Example 3

An equidistant conic projection with properly chosen parallels can map the conterminous United States with less than 1.5% distortion.

```
figure
usamap conus
load conus
patchm(uslat, uslon, [1 0.7 0])
plotm(statelat, statelon)
patchm(gtlakelat, gtlakelon, 'cyan')
framem off; gridm off; mlabel off; plabel off
mdistort('parscale', -2:.5:2)
parallelui
```
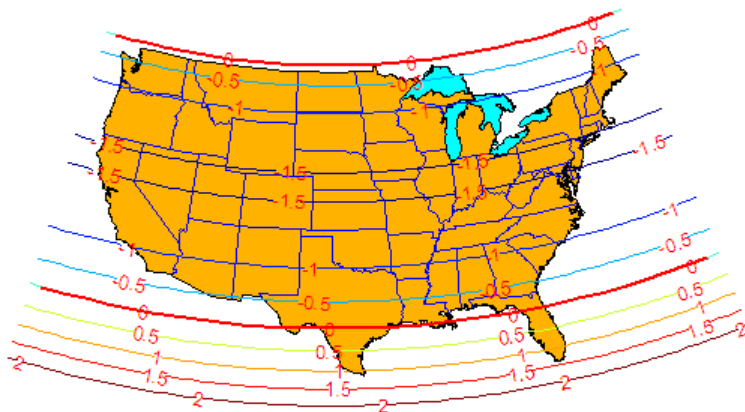
## More About

**Tips**

`mdistort` can help in the placement of standard parallels for projections. Standard parallels are generally placed to minimize distortion over the region of interest. The default parallel locations might not be appropriate for maps of smaller regions. By using `mdistort` and `parallelui`, you can immediately see how the movement of parallels reduces distortion.

## See Also
`tissot` | `distortcalc` | `vfwdtran`

# meanm

Mean location of geographic coordinates

## Syntax

```
[latmean,lonmean] = meanm(lat,lon)
[latmean,lonmean] = meanm(lat,lon,units)
[latmean,lonmean] = meanm(lat,lon,ellipsoid)
```

## Description

`[latmean,lonmean] = meanm(lat,lon)` returns row vectors of the geographic mean positions of the columns of the input latitude and longitude points.

`[latmean,lonmean] = meanm(lat,lon,units)` indicates the angular units of the data. When the standard angle string *units* is omitted, `'degrees'` is assumed.

`[latmean,lonmean] = meanm(lat,lon,ellipsoid)` specifies the shape of the Earth using `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The default ellipsoid model is a spherical Earth, which is sufficient for most applications.

If a single output argument is used, then `geomeans = [latmean,longmean]`. This is particularly useful if the original `lat` and `lon` inputs are column vectors.

## Background

Finding the mean position of geographic points is more complicated than simply averaging the latitudes and longitudes. `meanm` determines mean position through three-dimensional vector addition. See "Geographic Statistics" in the *Mapping Toolbox User's Guide*.

## Examples

Create random latitude and longitude matrices:

```
lat = rand(3)

lat =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

lon = rand(3)

lon =
    0.4447    0.9218    0.4057
    0.6154    0.7382    0.9355
    0.7919    0.1763    0.9169

[latmean,lonmean] = meanm(lat,lon,'radians')

latmean =
    0.6004    0.7395    0.4448
lonmean =
    0.6347    0.6324    0.7478
```

## See Also
filterm | hista | histr | stdist | stdm

# meridianarc

Ellipsoidal distance along meridian

## Syntax

```
s = meridianarc(phi1,phi2,ellipsoid)
```

## Description

`s = meridianarc(phi1,phi2,ellipsoid)` calculates the (signed) distance `s` between latitudes `phi1` and `phi2` along a meridian on the ellipsoid defined by `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form [`semimajor_axis eccentricity`]. Latitudes `phi1` and `phi2` are in radians. The distance `s` has the same units as the semimajor axis of the ellipsoid. If `phi2` is less than `phi1`, `s` is negative.

## See Also
`meridianfwd`

# meridianfwd

Reckon position along meridian

## Syntax

```
phi2 = meridianfwd(phi1,s,ellipsoid)
```

## Description

`phi2 = meridianfwd(phi1,s,ellipsoid)` determines the geodetic latitude `phi2` reached by starting at geodetic latitude `phi1` and traveling distance `s` north (positive `s`) or south (negative `s`) along a meridian on the specified `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. Latitudes `phi1` and `phi2` are in radians, and `s` has the same units as the semimajor axis of the ellipsoid.

### See Also
`meridianarc`

# meshgrat

Construct map graticule for surface object display

## Syntax

```
[lat, lon] = meshgrat(Z, R)
[lat, lon] = meshgrat(Z, R, gratsize)
[lat, lon] = meshgrat(lat, lon)
[lat, lon] = meshgrat(latlim, lonlim, gratsize)
[lat, lon] = meshgrat(lat, lon, angleunits)
[lat, lon] = meshgrat(latlim, lonlim, angleunits)
[lat, lon] = meshgrat(latlim, lonlim, gratsize, angleunits)
```

## Description

`[lat, lon] = meshgrat(Z, R)` constructs a graticule for use in displaying a regular data grid, `Z`. In typical usage, a latitude-longitude graticule is projected, and the grid is warped to the graticule using MATLAB graphics functions. In this two-argument calling form, the graticule size is equal to the size of `Z`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```
If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```
If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel.

`[lat, lon] = meshgrat(Z, R, gratsize)` produces a graticule of size `gratsize`. `gratsize` is a two-element vector of the form `[number_of_parallels`

number_of_meridians]. If `gratsize = []`, then the graticule returned has the default size 50-by-100. (But if `gratsize` is omitted, a graticule of the same size as `Z` is returned.) A finer graticule uses larger arrays and takes more memory and time but produces a higher fidelity map.

`[lat, lon] = meshgrat(lat, lon)` takes the vectors `lat` and `lon` and returns graticule arrays of size `numel(lat)-by-numel(lon)`. In this form, `meshgrat` is similar to the MATLAB function `meshgrid`.

`[lat, lon] = meshgrat(latlim, lonlim, gratsize)` returns a graticule mesh of size `gratsize` that covers the geographic limits defined by the two-element vectors `latlim` and `lonlim`.

`[lat, lon] = meshgrat(lat, lon, `*angleunits*`)`, `[lat, lon] = meshgrat(latlim, lonlim, `*angleunits*`)`, and `[lat, lon] = meshgrat(latlim, lonlim, gratsize, `*angleunits*`)` use the string *angleunits* to specify the angle units of the inputs and outputs. The string *angleunits* can be either `'degrees'` (the default) or `'radians'`.

The graticule mesh is a grid of points that are projected on a map axes and to which surface map objects are warped. The fineness, or resolution, of this grid determines the quality of the projection and the speed of plotting. There is no hard and fast rule for sufficient graticule resolution, but in general, cylindrical projections need very few graticules in the longitudinal direction, while complex curve-generating projections require more.

## Examples

Make a (coarse) graticule for the entire world:

```
latlim = [-90 90];
lonlim = [-180 180];
[lat,lon] = meshgrat(latlim,lonlim,[3 6])

lat =
  -90.0000   -90.0000   -90.0000   -90.0000   -90.0000   -90.0000
         0          0          0          0          0          0
   90.0000    90.0000    90.0000    90.0000    90.0000    90.0000
lon =
 -180.0000 -108.0000   -36.0000    36.0000   108.0000   180.0000
 -180.0000 -108.0000   -36.0000    36.0000   108.0000   180.0000
```

```
-180.0000 -108.0000  -36.0000   36.0000  108.0000  180.0000
```

These paired coordinates are the graticule vertices, which are projected according to the requirements of the desired map projection. Then a surface object like the `topo` map can be warped to the grid.

## See Also
`meshgrid` | `meshm` | `surfacem` | `surfm`

# meshlsrm

3-D lighted shaded relief of regular data grid

## Syntax

```
meshlsrm(Z,R)
meshlsrm(Z,R,[azim elev])
meshlsrm(Z,R,[azim elev],cmap)
meshlsrm(Z,R,[azim elev],cmap,clim)
h = meshlsrm(...)
```

## Description

`meshlsrm(Z,R)` displays the regular data grid `Z` colored according to elevation and surface slopes. `R` can be a referencing vector, a referencing matrix, or a geographic raster reference object.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. By default, shading is based on a light to the east (90 deg.) at an elevation of 45 degrees. Also by default, the colormap is constructed from 16 colors and 16 grays. Lighting is applied before the data is projected. The current axes must have a valid map projection definition.

`meshlsrm(Z,R,[azim elev])` displays the regular data grid `Z` with the light coming from the specified azimuth and elevation. Angles are specified in degrees, with the

azimuth measured clockwise from North, and elevation up from the zero plane of the surface.

`meshlsrm(Z,R,[azim elev],cmap)` displays the regular data grid Z using the specified colormap. The number of grayscales is chosen to keep the size of the shaded colormap below 256. If the vector of `azimuth` and `elevation` is empty, the default locations are used. Color axis limits are computed from the data.

`meshlsrm(Z,R,[azim elev],cmap,clim)` uses the provided color axis limits, which by default are computed from the data.

`h = meshlsrm(...)` returns the handle to the surface drawn.

## Examples

Create a new colormap using `demcmap`, with white colors for the sea and default colors for land. Use this colormap for a lighted shaded relief map of the world.

```
korea = load('korea.mat');
Z = korea.map;
R = georasterref('RasterSize', size(Z), ...
  'Latlim', [30 45], 'Lonlim', [115 135]);
worldmap(Z, R)
meshlsrm(Z, R, [45, 65])
```
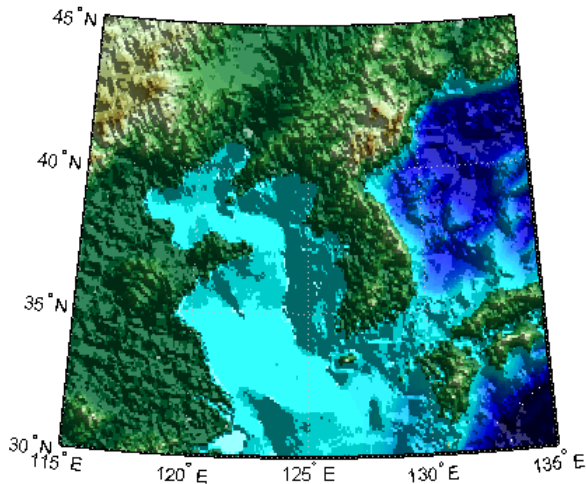
## More About

### Tips

This function effectively multiplies two colormaps, one with color based on elevation, the other with a grayscale based on the slope of the surface, to create a new colormap. This produces an effect similar to using a light on a surface, but with all of the visible colors actually in the colormap. Lighting calculations are performed on the unprojected data.

### See Also

meshgrat | meshm | pcolorm | surfacem | surflm | surflsrm

# meshm

Project regular data grid on map axes

## Syntax

```
meshm(Z, R)
meshm(Z, R, gratsize)
meshm(Z, R, gratsize, alt)
meshm(..., param1, val1, param2, val2, ...)
H = meshm(...)
```

## Description

`meshm(Z, R)` will display the regular data grid Z warped to the default projection graticule. R can be a referencing vector, a referencing matrix, or a geographic raster reference object.

If R is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If R is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If R is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If R is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. The current axes must have a valid map projection definition.

`meshm(Z, R, gratsize)` displays a regular data grid warped to a graticule mesh defined by the 1-by-2 vector `gratsize`. `gratsize(1)` indicates the number of lines of constant latitude (parallels) in the graticule, and `gratsize(2)` indicates the number of lines of constant longitude (meridians).

meshm(Z, R, gratsize, alt) displays the regular surface map at the altitude specified by alt. If alt is a scalar, then the grid is drawn in the z = alt plane. If alt is a matrix, then size(alt) must equal gratsize, and the graticule mesh is drawn at the altitudes specified by alt. If the default graticule is desired, set gratsize = [].

meshm(..., param1, val1, param2, val2, ...) uses optional parameter name-value pairs to control the properties of the surface object constructed by meshm. (If data is placed in the UserData property of the surface, then the projection of this object can not be altered once displayed.)

H = meshm(...) returns the handle to the surface drawn.

## Examples

```
korea = load('korea.mat');
Z = korea.map;
R = georasterref('RasterSize', size(Z), ...
  'Latlim', [30 45], 'Lonlim', [115 135]);
worldmap(Z, R)
meshm(Z, R)
demcmap(Z)
```
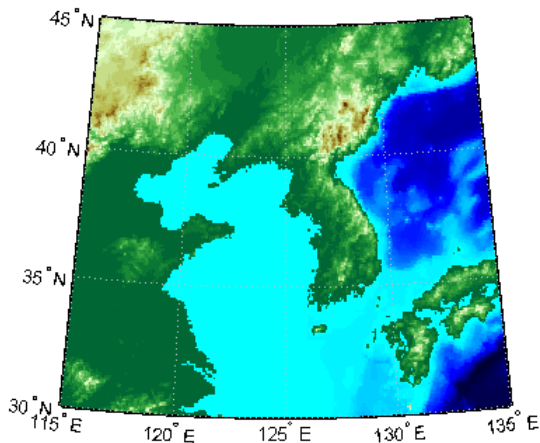


## See Also
geoshow | mapshow | meshgrat | pcolorm | surfacem | surfm

# mfwdtran

Project geographic features to map coordinates

## Syntax

```
[x,y] = mfwdtran(lat,lon)
[x,y,z] = mfwdtran(lat,lon,alt)
[...] = mfwdtran(mstruct,...)
```

## Description

`[x,y] = mfwdtran(lat,lon)` applies the forward transformation defined by the map projection in the current map axes. You can use this function to convert point locations and line and polygon vertices given in latitudes and longitudes to a planar, projected map coordinate system.

`[x,y,z] = mfwdtran(lat,lon,alt)` applies the forward projection to 3-D input, resulting in 3-D output. If the input `alt` is empty or omitted, then `alt` = 0 is assumed.

`[...] = mfwdtran(mstruct,...)` requires a valid map projection structure as the first argument. In this case, no map axes is needed.

## Examples

The following latitude and longitude data for the District of Columbia is obtained from the `usastatelo` shapefile:

```
dc = shaperead('usastatelo', 'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'District of Columbia'),...
    'Name'});
lat = [dc.Lat]';
lon = [dc.Lon]';
[lat lon]

ans =
    38.9000  -77.0700
```

```
38.9500   -77.1200
39.0000   -77.0300
38.9000   -76.9000
38.7800   -77.0300
38.8000   -77.0200
38.8700   -77.0200
38.9000   -77.0700
38.9000   -77.0500
38.9000   -77.0700
     NaN       NaN
```

Before projecting the data, it is necessary to define projection parameters. You can do this with the `axesm` function or with the `defaultm` function:

```
mstruct = defaultm('mercator');
mstruct.origin = [38.89 -77.04 0];
mstruct = defaultm(mstruct);
```

Now that the projection parameters have been set, transform the District of Columbia data into map coordinates using the Mercator projection:

```
[x,y] = mfwdtran(mstruct,lat,lon);
[x y]

ans =
   -0.0004    0.0002
   -0.0011    0.0010
    0.0001    0.0019
    0.0019    0.0002
    0.0001   -0.0019
    0.0003   -0.0016
    0.0003   -0.0003
   -0.0004    0.0002
   -0.0001    0.0002
   -0.0004    0.0002
       NaN       NaN
```

## See Also

defaultm | gcm | minvtran | projfwd | projinv | vfwdtran | vinvtran

# minaxis

Semiminor axis of ellipse

## Syntax

```
b = minaxis(semimajor,e)
b = minaxis(vec)
```

## Description

`b = minaxis(semimajor,e)` computes the semiminor axis of an ellipse (or ellipsoid of revolution) given the semimajor axis and eccentricity. The input data can be scalar or matrices of equal dimensions.

`b = minaxis(vec)` assumes a 2 element vector (`vec`) is supplied, where `vec` = [`semimajor`, `e`].

### See Also
`axes2ecc` | `flat2ecc` | `majaxis` | `n2ecc`

# minvtran

Unproject features from map to geographic coordinates

## Syntax

```
[lat,lon] = minvtran(x,y)
[lat,lon,alt] = minvtran(x,y,z)
[...] = minvtran(mstruct,...)
```

## Description

`[lat,lon] = minvtran(x,y)` applies the inverse transformation defined by the map projection in the current map axes. Using `minvtran`, you can convert point locations and line and polygon vertices in a planar, projected map coordinate system to latitudes and longitudes.

`[lat,lon,alt] = minvtran(x,y,z)` applies the inverse projection to 3-D input, resulting in 3-D output. If the input Z is empty or omitted, then `Z = 0` is assumed.

`[...] = minvtran(mstruct,...)` takes a valid map projection structure as the first argument. In this case, no map axes is needed.

## Examples

Before using any transformation functions, it is necessary to create a map projection structure. You can do this with `axesm` or the `defaultm` function:

```
mstruct = defaultm('mercator');
mstruct.origin = [38.89 -77.04 0];
mstruct = defaultm(mstruct);
```

The following latitude and longitude data for the District of Columbia is obtained from the `usastatelo` shapefile:

```
dc = shaperead('usastatelo', 'UseGeoCoords', true,...
     'Selector',{@(name) strcmpi(name,'District of Columbia'),...
```

```
     'Name'});
lat = [dc.Lat]';
lon = [dc.Lon]';
[lat lon]

ans =
   38.9000   -77.0700
   38.9500   -77.1200
   39.0000   -77.0300
   38.9000   -76.9000
   38.7800   -77.0300
   38.8000   -77.0200
   38.8700   -77.0200
   38.9000   -77.0700
   38.9000   -77.0500
   38.9000   -77.0700
       NaN       NaN
```

This data can be projected into Cartesian coordinates of the Mercator projection using the `mfwdtran` function:

```
[x,y] = mfwdtran(mstruct,lat,lon);
[x y]

ans =
   -0.0004    0.0002
   -0.0011    0.0010
    0.0001    0.0019
    0.0019    0.0002
    0.0001   -0.0019
    0.0003   -0.0016
    0.0003   -0.0003
   -0.0004    0.0002
   -0.0001    0.0002
   -0.0004    0.0002
      NaN       NaN
```

To transform the projected *x-y* data back into the unprojected geographic system, use the `minvtran` function:

```
[lat2,lon2] = minvtran(mstruct,x,y);
[lat2 lon2]

ans =
   38.9000   -77.0700
```

```
38.9500   -77.1200
39.0000   -77.0300
38.9000   -76.9000
38.7800   -77.0300
38.8000   -77.0200
38.8700   -77.0200
38.9000   -77.0700
38.9000   -77.0500
38.9000   -77.0700
   NaN       NaN
```

## See Also

axesm | defaultm | gcm | mfwdtran | projfwd | projinv | vfwdtran | vinvtran

# mlabel

Toggle and control display of meridian labels

## Syntax

```
mlabel
mlabel('on')
mlabel('off')
mlabel('reset')
mlabel(parallel)
mlabel(MapAxesPropertyName,PropertyValue,...)
```

## Description

`mlabel` toggles the visibility of meridian labeling on the current map axes.

`mlabel('on')` sets the visibility of meridian labels to `'on'`.

`mlabel('off')` sets the visibility of meridian labels to `'off'`.

`mlabel('reset')` resets the displayed meridian labels using the currently defined meridian label properties.

`mlabel(parallel)` sets the value of the `MLabelParallel` property of the map axes to the value of `parallel`. This determines the parallel upon which the labels are placed (see `axesm`). The options for `parallel` are a scalar latitude or the strings `'north'`, `'south'`, or `'equator'`.

`mlabel(MapAxesPropertyName,PropertyValue,...)` allows paired map axes' property names and property values to be passed in. For a complete description of map axes properties, see the `axesm` reference page in this guide.

Meridian label handles can be returned in `h` if desired.

### See Also
axesm | mlabelzero22pi | plabel | setm

# mlabelzero22pi

Convert meridian labels to 0-360 degree range

## Syntax

```
mlabelzero22pi
```

## Description

`mlabelzero22pi` displays longitude labels in the range of 0 to 360 degrees east of the prime meridian.

## Examples

```
% create a map
figure('color','w'); axesm('miller','grid','on'); tightmap;
mlabel on; plabel on
```
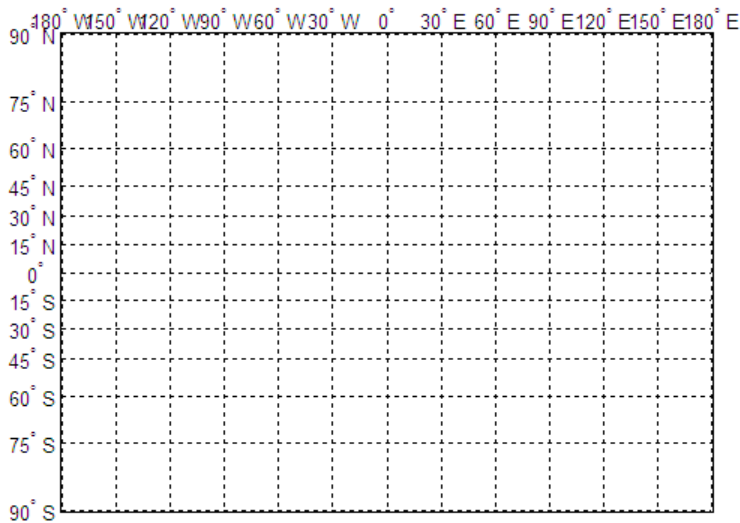
```
% Display longitude labels in the range of 0 to 360 degrees
mlabelzero22pi
```
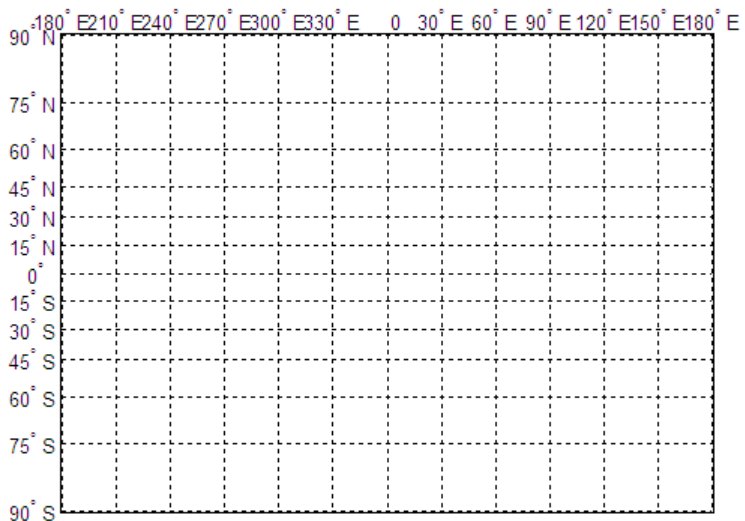


## See Also
```
mlabel
```

# n2ecc

Eccentricity of ellipse from third flattening

## Syntax

## Description

ecc = n2ecc(n) computes the eccentricity of an ellipse (or ellipsoid of revolution) given the parameter n (the "third flattening"). n is defined as (a-b)/(a+b), where a is the semimajor axis and b is the semiminor axis. Except when the input has 2 columns (or is a row vector), each element is assumed to be a third flattening and the output ecc has the same size as n.

ecc = n2ecc(n), where n has two columns (or is a row vector), assumes that the second column is a third flattening, and a column vector is returned.

## See Also
axes2ecc | ecc2n

# namem

Names of graphics objects

## Syntax

```
obj_names = namem
obj_names = namem(h)
```

## Description

`obj_names = namem` returns the names of all the objects on the current axes. The name of an object is the value of its `Tag` property, if specified. Otherwise, the name of the object is the value of its `Type` property. The values of these properties are either set at object creation or defined using the `tagm` function.

`obj_names = namem(h)` returns the name of the object (or objects) specified by the array of Handle Graphics object handles, `h`.

## Examples

### Get List of Handles to Handle Graphics Objects in Map Display

Display a map in a figure window.

```
load korea
h = worldmap(map, refvec);
set(h, 'Visible', 'off')
geoshow(h, map, refvec, 'DisplayType', 'texturemap')
demcmap(map)
```

Get a list of all the names of Handle Graphics objects in the display.

```
obj_names = namem

obj_names =

surface
PLabel
```

```
MLabel
Meridian
Parallel
Frame
```

Convert the array of names into a cell array.

```
obj_names_cell = cellstr(obj_names)

obj_names_cell =

    'surface'
    'PLabel'
    'MLabel'
    'Meridian'
    'Parallel'
    'Frame'
```

# Input Arguments

### h — Handle Graphics objects
array of handles

Handle Graphics objects, specified as an array of handles.

Example:

Data Types: `double`

# Output Arguments

### `obj_names` — Names of Handle Graphics objects
character matrix

Names of Handle Graphics objects, returned as a character matrix. `namem` removes duplicate object names from the output string matrix.

Note: Use `cellstr(obj_names)` to convert `obj_names` to a cell array of strings.

## See Also
`clma` | `clmo` | `handlem` | `hidem` | `showm` | `tagm`

# nanclip

Clip vector data with NaNs at specified pen-down locations

## Syntax

```
dataout = nanclip(datain)
dataout = nanclip(datain,pendowncmd)
```

## Description

`dataout = nanclip(datain)` and `dataout = nanclip(datain,pendowncmd)` return the pen-down delimited data in the matrix `datain` as NaN-delimited data in `dataout`. When the first column of `datain` equals `pendowncmd`, a segment is started and a NaN is inserted in all columns of `dataout`. The default `pendowncmd` is `-1`.

Pen-down delimited data is a matrix with a first column consisting of pen commands. At the beginning of each segment in the data, this first column has an entry corresponding to a pen-down command. Other entries indicate that the segment is continuing. NaN-delimited data consists of columns of data, each segment of which ends in a NaN in every data column. Since there is no pen command column, the NaN-delimited format can represent the same data in one fewer columns; the remaining columns have more entries, one for each NaN (that is, for each segment).

## Examples

```
datain = [-1 45 67; 0 23 54; 0 28 97; -1 47 89; 0 56 12]

datain =
    -1    45    67         % Begin first segment
     0    23    54
     0    28    97
    -1    47    89         % Begin second segment
     0    56    12

dataout = nanclip(datain)
```

```
dataout =
    45    67
    23    54
    28    97
   NaN   NaN                  % End first segment
    47    89
    56    12
   NaN   NaN                  % End second segment
```

## See Also
spcread

# nanm

Construct regular data grid of NaNs

## Compatibility

nanm is not recommended and may be removed in a future release. Instead, use georefcells to construct a geographic raster reference object and then use nan to initialize a data grid of the appropriate size:

```
R = georefcells(latlim,lonlim,1/scale,1/scale);
Z = nan(R.RasterSize);
```

## Syntax

```
[Z,refvec] = nanm(latlim,lonlim,scale)
```

## Description

[Z,refvec] = nanm(latlim,lonlim,scale) returns a regular data grid consisting entirely of NaNs and a three-element referencing vector for the returned Z. The two-element vectors latlim and lonlim define the latitude and longitude limits of the geographic region. They should be of the form [south north] and [west east], respectively. The scalar scale specifies the number of rows and columns per degree of latitude and longitude.

## Examples

```
[Z,refvec] = nanm([46,51],[-79,-75],1)

Z =
    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN
    NaN    NaN    NaN    NaN
```

```
refvec =
     1    51    -79
```

## See Also

limitm | onem | sizem | spzerom | zerom

# navfix

Mercator-based navigational fix

## Syntax

```
[latfix,lonfix] = navfix(lat,long,az)
[latfix,lonfix] = navfix(lat,long,range,casetype)
[latfix,lonfix] = navfix(lat,long,az_range,casetype)
[latfix,lonfix] = navfix(lat,long,az_range,casetype,drlat,drlon)
```

## Description

`[latfix,lonfix] = navfix(lat,long,az)` returns the intersection points of rhumb lines drawn parallel to the observed bearings, `az`, of the landmarks located at the points `lat` and `long` and passing through these points. One bearing is required for each landmark. Each possible pairing of the n landmarks generates one intersection, so the total number of resulting intersection points is the combinatorial *n choose 2*. The calculation time therefore grows rapidly with n.

`[latfix,lonfix] = navfix(lat,long,range,casetype)` returns the intersection points of Mercator projection circles with radii defined by `range`, centered on the landmarks located at the points `lat` and `long`. One range value is required for each landmark. Each possible pairing of the $n$ landmarks generates up to two intersections (circles can intersect twice), so the total number of resulting intersection points is the combinatorial *2 times (n choose 2)*. The calculation time therefore grows rapidly with $n$. In this case, the variable `casetype` is a vector of 0s the same size as the variable `range`.

`[latfix,lonfix] = navfix(lat,long,az_range,casetype)` combines ranges and bearings. For each element of `casetype` equal to 1, the corresponding element of `az_range` represents an azimuth to the associated landmark. Where `casetype` is a 0, `az_range` is a range.

`[latfix,lonfix] = navfix(lat,long,az_range,casetype,drlat,drlon)` returns for each possible pairing of landmarks only the intersection that lies closest to the dead reckoning position indicated by `drlat` and `drlon`. When this syntax is used, all

included landmarks' bearing lines or range arcs must intersect. If any possible pairing fails, the warning `No Fix` is displayed.

# Background

This is a navigational function. It assumes that all latitudes and longitudes are in degrees and all distances are in nautical miles. In navigation, piloting is the practice of fixing one's position based on the observed bearing and ranges *to* fixed landmarks (points of land, lighthouses, smokestacks, etc.) *from* the navigator's vessel. In conformance with navigational practice, bearings are treated as rhumb lines and ranges are treated as the radii of circles on a Mercator projection.

In practice, at least three azimuths (bearings) and/or ranges are required for a usable fix. The resulting intersections are unlikely to coincide exactly. Refer to "Navigation" in the *Mapping Toolbox User's Guide* for a more complete description of the use of this function.

# Examples

For a fully illustrated example of the application of this function, refer to the "Navigation" section in the *Mapping Toolbox User's Guide*.

Imagine you have two landmarks, at (15ºN,30.4ºW) and (14.8ºN,30.1ºW). You have a visual bearing to the first of 280º and to the second of 160º. Additionally, you have a range to the second of 12 nm. Find the intersection points:

```
[latfix,lonfix] = navfix([15 14.8 14.8],[-30.4 -30.1 -30.1],...
                          [280 160 12],[1 1 0])

latfix =
    14.9591       NaN
    14.9680    14.9208
    14.9879       NaN
lonfix =
   -30.1599       NaN
   -30.2121    -29.9352
   -30.1708       NaN
```

Here is an illustration of the geometry:

Small dots are intersection points.
A dead reckoning position could be
used to eliminate the inconsistent
intersection.
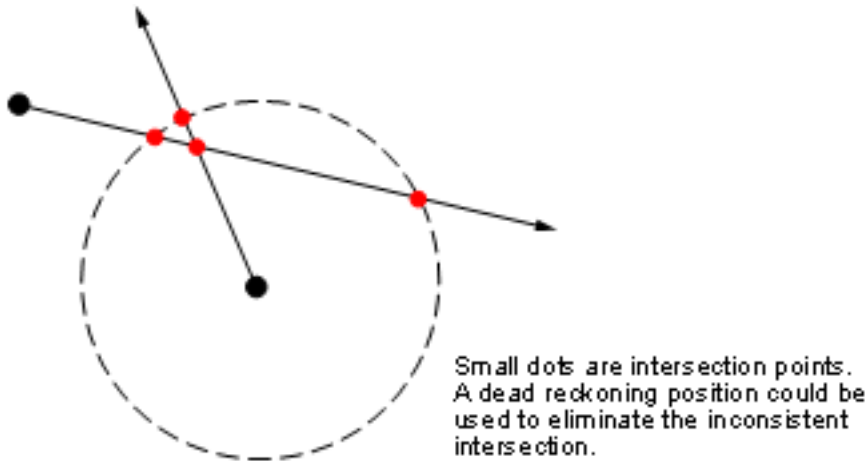
## Limitations

Traditional plotting and the `navfix` function are limited to relatively short distances. Visual bearings are in fact great circle azimuths, not rhumb lines, and range arcs are actually arcs of small circles, not of the planar circles plotted on the chart. However, the mechanical ease of the process and the practical limits of visual bearing ranges and navigational radar ranges (~30 nm) make this limitation moot in practice. The error contributed because of these assumptions is minuscule at that scale.

## More About

### Tips

The outputs of this function are matrices providing the locations of the intersections for all possible pairings of the n entered lines of bearing and range arcs. These matrices therefore have *n-choose-2* rows. In order to allow for two intersections per combination, these matrices have two columns. Whenever there are fewer than two intersections for that combination, one or two NaNs are returned in that row.

When a dead reckoning position is included, these matrices are column vectors.

navfix

## See Also

crossfix | gcxgc | gcxsc | scxsc | rhxrh | polyxpoly | dreckon | gcwaypts |
legs | track

# ned2aer

Local Cartesian NED to local spherical AER

## Syntax

```
[az,elev,slantRange] = ned2aer(xNorth,yEast,zDown)
[ ___ ] = ned2aer( ___ ,angleUnit)
```

## Description

`[az,elev,slantRange] = ned2aer(xNorth,yEast,zDown)` returns coordinates in a local spherical system corresponding to coordinates `xNorth`, `yEast`, `zDown` in a local north-east-down (NED) Cartesian system having the same local origin. Any of the three numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ned2aer( ___ ,angleUnit)` adds `angleUnit` which specifies the units outputs `az`, `elev`.

## Input Arguments

### xNorth — Local NED x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local NED system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### yEast — Local NED y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local NED system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**`zDown` — Local NED z-coordinates**
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local NED system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

**`angleUnit` — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

# Output Arguments

**`az` — Azimuth angles**
scalar value | vector | matrix | N-D array

Azimuth angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Azimuths are measured clockwise from north. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the half-open interval [0 360).

**`elev` — Elevation angles**
scalar value | vector | matrix | N-D array

Elevation angles in the local spherical system, returned as a scalar value, vector, matrix, or N-D array. Elevations are with respect to a plane perpendicular to the spheroid surface normal. Units determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the closed interval [-90 90].

**`slantRange` — Distances from local origin**
scalar value | vector | matrix | N-D array

Distances from origin in the local spherical system, returned as a scalar, vector, matrix, or N-D array. The straight-line, 3-D Cartesian distance is computed.

# See Also
aer2ned | enu2aer

# ned2ecef

Local Cartesian NED to geocentric ECEF

## Syntax

```
[X,Y,Z] = ned2ecef(xNorth,yEast,zDown,lat0,lon0,h0,spheroid)
[ ___ ] = ned2ecef( ___ ,angleUnit)
```

## Description

`[X,Y,Z] = ned2ecef(xNorth,yEast,zDown,lat0,lon0,h0,spheroid)` returns Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian coordinates corresponding to coordinates xNorth, yEast, zDown in a local north-east-down (NED) Cartesian system. Any of the first six numeric input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ned2ecef( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0` and `lon0`.

## Input Arguments

### `xNorth` — Local NED x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local NED system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `yEast` — Local NED y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local NED system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `zDown` — Local NED z-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local NED system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

### `angleUnit` — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### X — ECEF x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

### Y — ECEF y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

### Z — ECEF y-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object.

## See Also
aer2ecef | ecef2ned | enu2ecef | ned2geodetic

# ned2ecefv

Rotate vector from local NED to geocentric ECEF

## Syntax

```
[U,V,W] = ned2ecefv(uNorth,vEast,wDown,lat0,lon0)
[ ___ ] = ned2ecefv( ___ ,angleUnit)
```

## Description

`[U,V,W] = ned2ecefv(uNorth,vEast,wDown,lat0,lon0)` returns Cartesian 3-vector components in an Earth-Centered Earth-Fixed (ECEF) spheroid-centric Cartesian system corresponding to the 3-vector with components `uNorth`, `vEast`, `wDown` in a local north-east-down (NED) system. Any of the five numerical input arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric arguments must match in size.

`[ ___ ] = ned2ecefv( ___ ,angleUnit)` adds `angleUnit` which specifies the units of inputs `lat0` and `lon0`.

## Input Arguments

### `uNorth` — Vector x-components in NED system
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the local NED system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### `vEast` — Vector y-components in NED system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the local ENU system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### **wDown** — Vector z-components in NED system
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the local NED system, specified as a scalar value, vector, matrix, or N-D array.

Data Types: `single` | `double`

### **lat0** — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### **lon0** — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### **angleUnit** — Units of angles
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

## Output Arguments

### **U** — Vector x-components in ECEF system
scalar value | vector | matrix | N-D array

x-components of one or more Cartesian vectors in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array.

### V — Vector y-components in ECEF system
scalar value | vector | matrix | N-D array

y-components of one or more Cartesian vectors in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array.

### W — Vector z-components in ECEF system
scalar value | vector | matrix | N-D array

z-components of one or more Cartesian vectors in the spheroid-centric ECEF system, returned as a scalar value, vector, matrix, or N-D array.

## See Also
ecef2ned | enu2ecefv | ned2ecef

# ned2geodetic

Local Cartesian NED to geodetic

## Syntax

```
[lat,lon,h] = ned2geodetic(xNorth,yEast,zDown,lat0,lon0,h0,spheroid)
[ ___ ] = ned2geodetic( ___ ,angleUnit)
```

## Description

`[lat,lon,h] = ned2geodetic(xNorth,yEast,zDown,lat0,lon0,h0,spheroid))`
returns geodetic coordinates corresponding to coordinates `xNorth`, `yEast`, `zDown` in
a local north-east-down (NED) Cartesian system. Any of the first six numeric input
arguments can be scalar, even when the others are nonscalar; but all nonscalar numeric
arguments must match in size.

`[ ___ ] = ned2geodetic( ___ ,angleUnit)` adds `angleUnit` which specifies the
units of inputs `lat0`, `lon0`, and outputs `lat`, `lon`.

## Input Arguments

### `xNorth` — Local NED x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the local NED system, specified as a scalar value,
vector, matrix, or N-D array. Values must be in units that match the `LengthUnit`
property of the `spheroid` input.

Data Types: `single` | `double`

### `yEast` — Local NED y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the local NED system, specified as a scalar value,
vector, matrix, or N-D array. Values must be in units that match the `LengthUnit`
property of the `spheroid` input.

Data Types: `single` | `double`

### `zDown` — Local NED z-coordinates
scalar value | vector | matrix | N-D array

z-coordinates of one or more points in the local NED system, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the `LengthUnit` property of the `spheroid` input.

Data Types: `single` | `double`

### `lat0` — Geodetic latitude of local origin
scalar value | vector | matrix | N-D array

Geodetic latitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lat0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `lon0` — Longitude of local origin
scalar value | vector | matrix | N-D array

Longitude of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `lon0` is scalar, but it need not be. (It may refer to a moving platform, for example). Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `h0` — Ellipsoidal height of local origin
scalar value | vector | matrix | N-D array

Ellipsoidal height of local origin (reference) point(s), specified as a scalar value, vector, matrix, or N-D array. In many cases there is one origin (reference) point, and the value of `h0` is scalar, but it need not be. (It may refer to a moving platform, for example). Units are determined by the `LengthUnit` property of the spheroid input.

Data Types: `single` | `double`

### `spheroid` — Reference spheroid
referenceEllipsoid | oblateSpheroid | referenceSphere

Reference spheroid, specified as a referenceEllipsoid, oblateSpheroid, or referenceSphere object. Use the constructor for one of these three classes, or the `wgs84Ellipsoid` function, to construct a Mapping Toolbox spheroid object. You can not directly pass in a string that names your spheroid. Instead, pass that string to `referenceEllipsoid` or `referenceSphere` and use the resulting object.

**angleUnit — Units of angles**
`'degrees'` (default) | `'radians'`

Units of angles, specified as 'degrees' (default), or 'radians'.

Data Types: `char`

# Output Arguments

**lat — Geodetic latitudes**
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the closed interval [-90 90].

**lon — Longitudes**
scalar value | vector | matrix | N-D array

Longitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the input argument `angleUnit`, if supplied; values are in degrees, otherwise. When in degrees, they lie in the interval [-180 180].

**h — Ellipsoidal heights**
scalar value | vector | matrix | N-D array

Ellipsoidal heights of one or more points, returned as a scalar value, vector, matrix, or N-D array. Units are determined by the `LengthUnit` property of the `spheroid` object

# See Also
aer2geodetic | enu2geodetic | geodetic2ned | ned2ecef

# neworig

Orient regular data grid to oblique aspect

## Syntax

```
[Z,lat,lon] = neworig(Z0,R,origin)
[Z,lat,lon] = neworig(Z0,R,origin,'forward')
[Z,lat,lon] = neworig(Z0,R,origin,'inverse')
```

## Description

`[Z,lat,lon] = neworig(Z0,R,origin)` and `[Z,lat,lon] = neworig(Z0,R,origin,'forward')` will transform regular data grid `Z0` into an oblique aspect, while preserving the matrix storage format. In other words, the oblique map origin is not necessarily at (0,0) in the Greenwich coordinate frame. This allows operations to be performed on the matrix representing the oblique map. For example, azimuthal calculations for a point in a data grid become row and column operations if the data grid is transformed so that the north pole of the oblique map represents the desired point on the globe.

`R` can be a geographic raster reference object, a referencing vector, or a referencing matrix. If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

`[cells/degree northern_latitude_limit western_longitude_limit]`

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

`[lon lat] = [row col 1] * R`

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points

outside the grid limits or for which `lat` or `lon` contain NaN. All angles are in units of degrees.

`[Z,lat,lon] = neworig(ZO,R,origin,'inverse')` transforms the regular data grid from the oblique frame to the Greenwich coordinate frame.

The `neworig` function transforms a regular data grid into a new matrix in an altered coordinate system. An analytical use of the new matrix can be realized in conjunction with the `newpole` function. If a selected point is made the *north pole* of the new system, then when a new matrix is created with `neworig`, each row of the new matrix is a constant distance from the selected point, and each column is a constant azimuth from that point.

## Limitations

`neworig` only supports data grids that cover the entire globe.

## Examples

This is the `topo` map transformed to put Sri Lanka at the North Pole:

```
load topo
origin = newpole(7,80)
origin =
    83.0000 -100.0000         O
[Z,lat,lon] = neworig(topo,topolegend,origin);
axesm miller
latlim = [ -90  90];
lonlim = [-180 180];
gratsize = [90 180];
[lat,lon] = meshgrat(latlim,lonlim,gratsize);
surfm(lat,lon,Z)
demcmap(topo)
tightmap
```
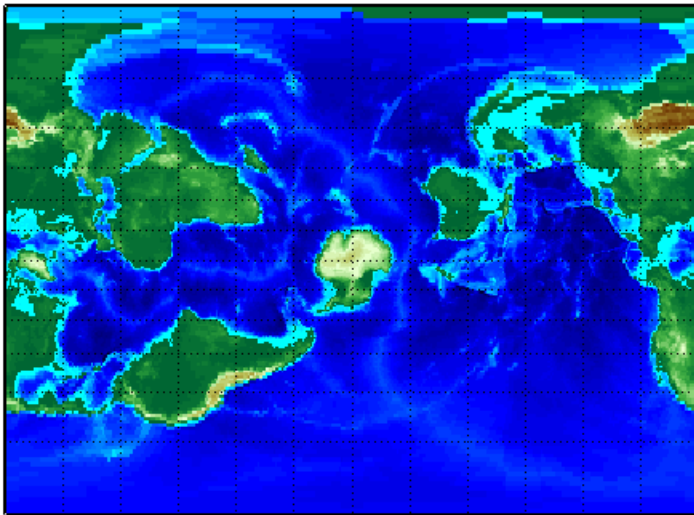
## See Also
org2pol | rotatem | setpostn

# newpole

Origin vector to place specific point at pole

## Syntax

```
origin = newpole(polelat,polelon)
origin = newpole(polelat,polelon,units)
```

## Description

`origin = newpole(polelat,polelon)` provides the `origin` vector for a transformed coordinate system based upon moving the point (`polelat`, `polelon`) to become the north pole singularity in the new system. The origin is a three-element vector of the form `[latitude longitude orientation]`, where the latitude and longitude are the coordinates the new center (`origin`) had in the untransformed system, and the orientation is the azimuth of the true North Pole from the new origin point. For the `newpole` calculation, this orientation is constrained to be always 0º.

`origin = newpole(polelat,polelon,units)` specifies the units of the inputs and output, where *units* is any valid angle units string. The default is `'degrees'`.

When developing transverse or oblique projections, you need transformed coordinate systems. One way to define these systems is to establish the point in the original (untransformed) system that will become the new (transformed) *north pole*.

## Examples

Take a point and make it the new North Pole:

```
origin = newpole(60,180)

origin =
   30.0000        0        0
```

This makes sense: as a point 30º beyond the true North Pole on the original origin's meridian is pulled up to become the *pole*, the point originally 30º above the origin is pulled down into the origin spot.

## See Also
neworig | org2pol | putpole

# nm2deg

Convert distance from nautical miles to degrees

## Syntax

```
deg = nm2deg(nm)
deg = nm2deg(nm,radius)
deg = nm2deg(nm,sphere)
```

## Description

`deg = nm2deg(nm)` converts distances from nautical miles to degrees, as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`deg = nm2deg(nm,radius)` converts distances from nautical miles to degrees, as measured along a great circle on a sphere having the specified radius. `radius` must be in units of nautical miles.

`deg = nm2deg(nm,sphere)` converts distances from nautical miles to degrees, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

### See Also

deg2rad | deg2km | deg2nm | deg2sm | km2deg | km2nm | km2rad | km2sm | nm2km | nm2sm | sm2deg | sm2km | sm2nm | rad2deg

# nm2km

Convert nautical miles to kilometers

## Syntax

```
km = nm2km(nm)
```

## Description

`km = nm2km(nm)` converts distances from nautical miles to kilometers.

## Examples

How fast is 30 knots (nautical miles per hour) in kph?

```
km = nm2km(30)

km =
   55.5600
```

## See Also

deg2km | km2deg | km2rad | rad2km | deg2nm | nm2deg | nm2rad | rad2nm |
deg2sm | sm2deg | deg2sm | sm2rad | rad2sm

# nm2rad

Convert distance from nautical miles to radians

## Syntax

```
rad = nm2rad(nm)
rad = nm2rad(nm,radius)
rad = nm2rad(nm,sphere)
```

## Description

`rad = nm2rad(nm)` converts distances from nautical miles to radians as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`rad = nm2rad(nm,radius)` converts distances from nautical miles to radians as measured along a great circle on a sphere having the specified radius. `radius` must be in units of nautical miles.

`rad = nm2rad(nm,sphere)` converts distances from nautical miles to radians, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

## See Also

deg2rad | rad2km | km2deg | km2nm | km2sm | rad2nm | nm2deg | nm2km | nm2sm | rad2sm | sm2deg | sm2km | sm2nm | rad2deg

# nm2sm

Convert nautical to statute miles

## Syntax

```
sm = nm2sm(nm)
```

## Description

`sm = nm2sm(nm)` converts distances from nautical miles to statute miles.

## See Also

deg2km | km2deg | km2rad | rad2km | deg2nm | nm2deg | nm2rad | rad2nm |
deg2sm | sm2deg | deg2sm | sm2rad | rad2sm

# northarrow

Add graphic element pointing to geographic North Pole

## Syntax

```
northarrow
northarrow('property',value,...)
```

## Description

`northarrow` creates a default north arrow.

`northarrow('property',value,...)` creates a north arrow using the specified property/value pairs. Valid entries for properties are `'latitude'`, `'longitude'`, `'facecolor'`, `'edgecolor'`, `'linewidth'`, and `'scaleratio'`. The `'latitude'` and `'longitude'` properties specify the location of the north arrow. The `'facecolor'`, `'edgecolor'`, and `'linewidth'` properties control the appearance of the north arrow. The `'scaleratio'` property represents the size of the north arrow as a fraction of the size of the axes. A `'scaleratio'` value of 0.10 creates a north arrow one-tenth (1/10) the size of the axes. You can change the appearance (`'facecolor'`, `'edgecolor'`, and `'linewidth'`) of the north arrow using the `set` command.

`northarrow` creates a north arrow symbol at the map origin on the displayed map. You can reposition the north arrow symbol by clicking and dragging its icon. Alternate clicking the icon creates an input dialog box that you can also use to change the location of the north arrow.

Modifying some of the properties of the north arrow results in replacement of the original object. Use `HANDLEM('NorthArrow')` to get the handles associated with the north arrow.

## Examples

Create a map of the South Pole and then add the north arrow in the upper left of the map.

```
Antarctica = shaperead('landareas', 'UseGeoCoords', true, ...
    'Selector',{@(name) strcmpi(name,{'Antarctica'}), 'Name'});
figure;
worldmap('south pole')
geoshow(Antarctica,'FaceColor',[.9 .9 .9])
northarrow('latitude', -57, 'longitude', 135);
```
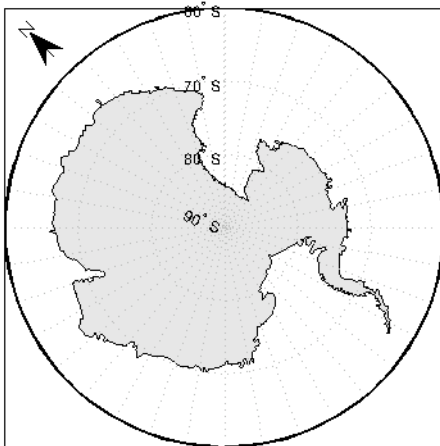


Right-click the north arrow icon to activate the input dialog box. Increase the size of the north arrow symbol by changing the 'ScaleRatio' property.



1-1045
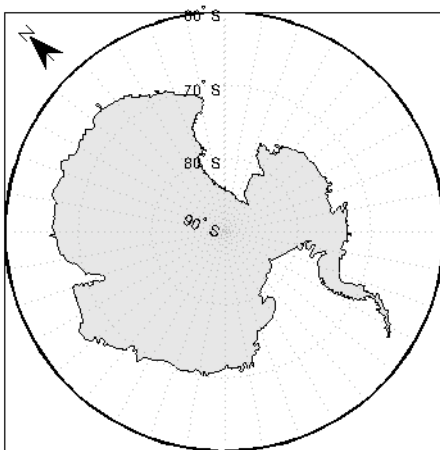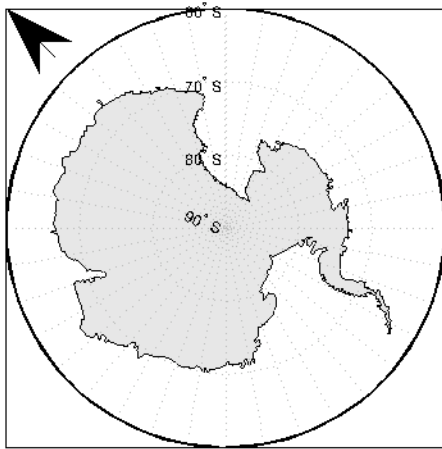
Create a map of Texas and add the north arrow in the lower left of the map.

```
figure; usamap('texas')
states = shaperead('usastatelo.shp','UseGeoCoords',true);
faceColors = makesymbolspec('Polygon',...
        {'INDEX', [1 numel(states)], 'FaceColor', ...
        polcmap(numel(states))});
geoshow(states, 'DisplayType', 'polygon', ...
        'SymbolSpec', faceColors)
northarrow('latitude',25,'longitude',-105,'linewidth',1.5);
```
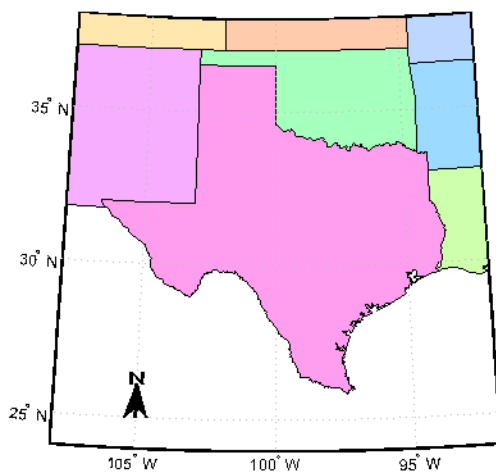
Change the 'FaceColor' and 'EdgeColor' properties of the north arrow.

```
h = handlem('NorthArrow');
set(h,'FaceColor',[1.000 0.8431 0.0000],...
      'EdgeColor',[0.0100 0.0100 0.9000])
```
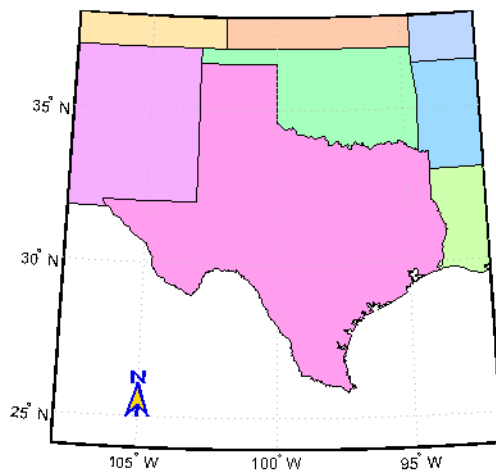
## Limitations

You can draw multiple north arrows on the map. However, the callbacks will only work with the most recently created north arrow. In addition, since it can be displayed outside the map frame limits, the north arrow is not converted into a "mapped" object. Hence, the location and orientation of the north arrow have to be updated manually if the map origin or projection changes.

## See Also

`scaleruler`

# npi2pi

Wrap longitudes to [-180 180] degree interval

---

**Note:** The `npi2pi` function has been replaced by `wrapTo180` and `wrapToPi`.

---

## Syntax

```
anglout = npi2pi(anglin)
anglout = npi2pi(anglin,units)
anglout = npi2pi(anglin,units,method)
```

## Description

`anglout = npi2pi(anglin)` wraps the input angle `anglin` (typically representing a longitude) to lie on the range -180 to 180 (e.g., 270º is renamed -90º).

`anglout = npi2pi(anglin,units)` specifies the angle units with any valid angle units string *units*. The default is `'degrees'`.

`anglout = npi2pi(anglin,units,method)` allows special alternative computations to be used when `npi2pi` is called from within certain Mapping Toolbox functions. *method* can be one of the following strings:

- `'exact'`, for exact wrapping (the default value)
- `'inward'`, where angles are scaled by a factor of `(1 - epsm('radians'))` before wrapping
- `'outward'`, where angles are scaled by a factor of `(1 + epsm('radians'))` before wrapping

## Examples

```
npi2pi(315)
```

```
ans =
   -45

npi2pi(181)

ans =
  -179
```

## See Also
wrapToPi | wrapTo180

# onem

Construct regular data grid of 1s

## Compatibility

onem is not recommended and may be removed in a future release. Instead, use georefcells to construct a geographic raster reference object and then use ones to initialize a data grid of the appropriate size:

```
R = georefcells(latlim,lonlim,1/scale,1/scale);
Z = ones(R.RasterSize);
```

## Syntax

```
[Z,refvec] = onem(latlim,lonlim,scale)
```

## Description

[Z,refvec] = onem(latlim,lonlim,scale) returns a regular data grid consisting entirely of 1s and a three-element referencing vector for the returned data grid, Z.. The two-element vectors latlim and lonlim define the latitude and longitude limits of the geographic region. They should be of the form [south north] and [west east], respectively. The scalar scale specifies the number of rows and columns per degree of latitude and longitude.

## Examples

```
[Z,refvec] = onem([46,51],[-79,-75],1)

Z =
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
refvec =
     1     51    -79
```

## See Also

```
limitm | nanm | sizem | spzerom | zerom
```

# org2pol

Location of north pole in rotated map

## Syntax

```
pole = org2pol(origin)
pole = org2pol(origin,units)
```

## Description

`pole = org2pol(origin)` returns the location of the North Pole in terms of the coordinate system after transformation based on the input `origin`. The `origin` is a three-element vector of the form `[latitude longitude orientation]`, where `latitude` and `longitude` are the coordinates that the new center (origin) had in the untransformed system, and `orientation` is the azimuth of the true North Pole from the new origin point in the transformed system. The output `pole` is a three-element vector of the form `[latitude longitude meridian]`, which gives the latitude and longitude point in terms of the original untransformed system of the new location of the true North Pole. The meridian is the longitude from the original system upon which the new system is centered.

`pole = org2pol(origin,units)` allows the specification of the angular units of the `origin` vector, where `units` is any valid angle units string. The default is `'degrees'`.

When developing transverse or oblique projections, transformed coordinate systems are required. One way to define these systems is to establish the point at which, in terms of the original (untransformed) system, the (transformed) true North Pole will lie.

## Examples

Perhaps you want to make (30ºN,0º) the new origin. Where does the North Pole end up in terms of the original coordinate system?

```
pole = org2pol([30 0 0])
```

```
pole =
   60.0000          0          0
```

This makes sense: pull a point 30° down to the origin, and the North Pole is pulled down 30°. A little less obvious example is the following:

```
pole = org2pol([5 40 30])

pole =
   59.6245   80.0750   40.0000
```

## See Also
neworig | putpole

# outlinegeoquad

Polygon outlining geographic quadrangle

## Syntax

```
[lat,lon] = outlinegeoquad(latlim,lonlim,dlat,dlon)
```

## Description

`[lat,lon] = outlinegeoquad(latlim,lonlim,dlat,dlon)` constructs a polygon that traces the outline of the geographic quadrangle defined by `latlim` and `lonlim`. Such a polygon can be useful for displaying the quadrangle graphically, especially on a projection where the meridians and/or parallels do not project to straight lines. `latlim` is a two-element vector of the form: `[southern-limit northern-limit]` and `lonlim` is a two-element vector of the form: `[western-limit eastern-limit]`. `dlat` is a positive scalar that specifies a minimum vertex spacing in degrees to be applied along the meridians that bound the eastern and western edges of the quadrangle. Likewise, `dlon` is a positive scalar that specifies a minimum vertex spacing in degrees of longitude to be applied along the parallels that bound the northern and southern edges of the quadrangle. The outputs `lat` and `lon` contain the vertices of a simple closed polygon with clockwise vertex ordering.

## Examples

Display the outlines of three geographic quadrangles having very different qualities on top of a simple base map:

```
figure('Color','white')
axesm('ortho','Origin',[-45 110],'frame','on','grid','on')
axis off
load coastlines
geoshow(coastlat, coastlon)

% Quadrangle covering Australia and vicinity
```

```
[lat, lon] = outlinegeoquad([-45 5],[110 175],5,5);
geoshow(lat,lon,'DisplayType','polygon','FaceAlpha',0.5);

% Quadrangle covering Antarctic region
antarcticCircleLat = dms2degrees([-66 33 39]);
[lat, lon] = outlinegeoquad([-90 antarcticCircleLat], ...
   [-180 180],5,5);
geoshow(lat,lon,'DisplayType','polygon', ...
      'FaceColor','cyan','FaceAlpha',0.5);

% Quadrangle covering nominal time zone 9 hours ahead of UTC
[lat, lon] = outlinegeoquad([-90 90], 135 + [-7.5 7.5], 5, 5);
geoshow(lat,lon,'DisplayType','polygon', ...
      'FaceColor','green','FaceAlpha',0.5);
```
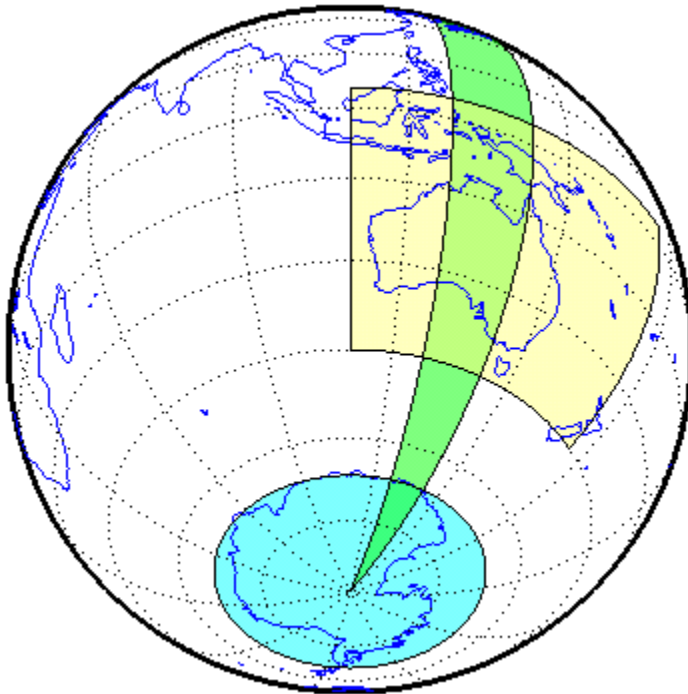
# More About

### Tips

All input and output angles are in units of degrees. Choose a reasonably small value for `dlat` (a few degrees, perhaps) when using a projection with curved meridians or curved parallels.

To avoid interpolating extra vertices along meridians or parallels, set `dlat` or `dlon` to a value of `Inf`.

## Special Cases

The insertion of additional vertices is suppressed at the poles (that is, if `latlim(1) == -90` or `latlim(2) == 90`. If `lonlim` corresponds to a quadrangle width of exactly 360 degrees (`lonlim == [-180 180]`, for example), then it covers a full latitudinal zone and includes two separate, NaN-separated parts, unless either

- `latlim(1) == -90` *or* `latlim(2) == 90`, so that only one part is needed—a polygon that follows a parallel clockwise around one of the poles.
- `latlim(1) == -90` *and* `latlim(2) == 90`, so that the quadrangle encompasses the entire planet. In this case, the quadrangle cannot be represented by a latitude-longitude polygon, and an error results.

## See Also
ingeoquad | intersectgeoquad

# paperscale

Set figure properties for printing at specified map scale

## Syntax

```
paperscale(paperdist,punits,surfdist,sunits)
paperscale(paperdist,punits,surfdist,sunits,lat,long)
paperscale(paperdist,punits,surfdist,sunits,lat,long,az)
paperscale(paperdist,punits,surfdist,sunits,lat,long,az,gunits)
paperscale(paperdist,punits,surfdist,sunits,lat,long,az,gunits,
radius)
paperscale(scale,...)
[paperXdim,paperYdim] = paperscale(...)
```

## Description

paperscale(paperdist,*punits*,surfdist,*sunits*) sets the figure paper position to print the map in the current axes at the desired scale. The scale is described by the geographic distance that corresponds to a paper distance. For example, a scale of 1 inch = 10 kilometers is specified as degrees(1,'inch',10,'km'). See below for an alternate method of specifying the map scale. The surface distance units string *sunits* can be any string recognized by unitsratio. The paper units string *punits* can be any dimensional units string recognized for the figure PaperUnits property.

paperscale(paperdist,*punits*,surfdist,*sunits*,lat,long) sets the paper position so that the scale is correct at the specified geographic location. If omitted, the default is the center of the map limits.

paperscale(paperdist,*punits*,surfdist,*sunits*,lat,long,az) also specifies the direction along which the scale is correct. If omitted, 90 degrees (east) is assumed.

paperscale(paperdist,*punits*,surfdist,*sunits*,lat,long,az,*gunits*) also specifies the units in which the geographic position and direction are given. If omitted, 'degrees' is assumed.

paperscale(paperdist,*punits*,surfdist,*sunits*,lat,long,az,*gunits*,

radius) uses the last input to determine the radius of the sphere. `radius` can be one of the strings supported by `km2deg`, or it can be the (numerical) radius of the desired sphere in `zunits`. If omitted, the default radius of the Earth is used..

`paperscale(scale,...)`, where the numeric scale replaces the two property/value pairs, specifies the scale as a ratio between distance on the sphere and on paper. This is commonly notated on maps as 1:scale (e.g. 1:100 000, or 1:1 000 000). For example, `paperscale(100000)` or `paperscale(100000,lat,long)`.

`[paperXdim,paperYdim] = paperscale(...)` returns the computed paper dimensions. The dimensions are in the paper units specified. For the scale calling form, the returned dimensions are in centimeters.

## Background

Maps are usually printed at a size that allows an easy comparison of distances measured on paper to distances on the Earth. The relationship of geographic distance and paper distance is termed *scale*. It is usually expressed as a ratio, such as 1 to 100,000 or 1:100,000 or 1 cm = 1 km.

## Examples

The small circle measures 10 cm across when printed.

```
axesm mercator
[lat,lon] = scircle1(0,0,km2deg(5));
plotm(lat,lon)
[x,y] = paperscale(1,'centimeter',1,'km'); [x y]

ans =
      13.154        12.509

set(gca,'pos', [ 0 0 1 1])
[x,y] = paperscale(1,'centimeter',1,'km'); [x y]

ans =
      10.195        10.195
```

## Limitations

The relationship between the paper and geographic coordinates holds only as long as there are no changes to the display that affect the axes limits or the relationship between geographic coordinates and projected coordinates. Changes of this type include the ellipsoid or scale factor properties of the map axes, or adding elements to the display that cause MATLAB to modify the axes autoscaling. To be sure that the scale is correct, execute `paperscale` just before printing.

## See Also

`printpreview` | `axesscale` | `daspectm`

# parametricLatitude

Convert geodetic to parametric latitude

## Syntax

```
beta = parametricLatitude(phi,F)
beta = parametricLatitude(phi,F,angleUnit)
```

## Description

`beta = parametricLatitude(phi,F)` returns the parametric latitude corresponding to geodetic latitude `phi` on an ellipsoid with flattening `F`.

`beta = parametricLatitude(phi,F,angleUnit)` specifies the units of input `phi` and output `beta`.

## Examples

### Convert Geodetic Latitude to Parametric Latitude

Create a reference ellipsoid and then convert the geodetic latitude to parametric latitude. The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;

parametricLatitude(45, s.Flattening)

ans =

   44.9038
```

### Convert Geodetic Latitude Expressed in Radians to Parametric Latitude

Create a reference ellipsoid and then convert a parametric latitude expressed in radians to geodetic latitude. The reference ellipsoid contains a flattening factor.

```
s = wgs84Ellipsoid;
```

```
parametricLatitude(pi/3, s.Flattening, 'radians')

ans =

    1.0457
```

## Input Arguments

### `phi` — Geodetic latitude of one or more points
scalar value, vector, matrix, or N-D array

Geodetic latitude of one or more points, specified as a scalar value, vector, matrix, or N-D array. Values must be in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

Data Types: `single` | `double`

### `F` — Flattening of reference spheroid
scalar

Flattening of reference spheroid, specified as a scalar value.

Data Types: `double`

### `angleUnit` — Unit of measurement for angle
'`degrees`' (default) | '`radians`'

Unit of measurement for angle, specified as the text string '`degrees`' or '`radians`'.

Data Types: `char`

## Output Arguments

### `beta` — Parametric latitudes of one or more points
scalar value, vector, matrix, or N-D array

Parametric latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array. Values are in units that match the input argument `angleUnit`, if supplied, and in degrees, otherwise.

### See Also

map.geodesy.AuthalicLatitudeConverter | map.geodesy.ConformalLatitudeConverter | map.geodesy.IsometricLatitudeConverter | map.geodesy.RectifyingLatitudeConverter | geocentricLatitude | geodeticLatitudeFromparametric

# patchesm

Project patches on map axes as individual objects

## Syntax

```
patchesm(lat,lon,cdata)
patchesm(lat,lon,z,cdata)
patchesm(...,'PropertyName',PropertyValue,...)
h = patchesm(...)
```

## Description

`patchesm(lat,lon,cdata)` projects 2-D patch objects onto the current map axes. The input latitude and longitude data must be in the same units as specified in the current map axes. The input cdata defines the patch face color. If the input vectors are `NaN` clipped, then multiple patches are drawn each with a single face. Unlike `fillm` and `fill3m`, `patchesm` will always add the patches to the current map regardless of the current hold state.

`patchesm(lat,lon,z,cdata)` projects 3-D planar patches at the uniform elevation given by scalar `z`.

`patchesm(...,'PropertyName',PropertyValue,...)` uses the patch properties supplied to display the patch. Except for `xdata`, `ydata`, and `zdata`, all patch properties available through `patch` are supported by `patchesm`.

`h = patchesm(...)` returns the handles to the patch objects drawn.

## Examples

```
load coast
axesm sinusoid; framem
h = patchesm(lat,long,'b');
```
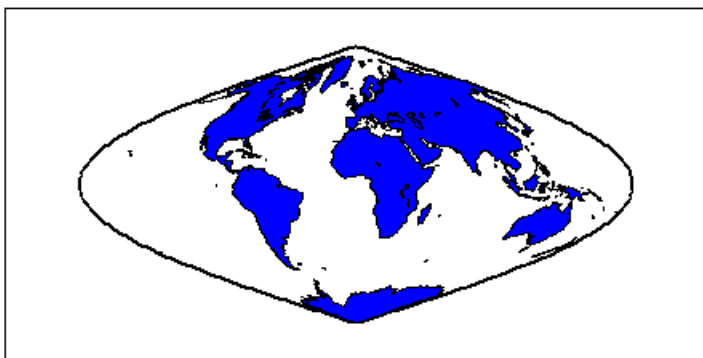
```
length(h)

ans =
   238
```

# More About

**Tips**

## Differences between patchesm and patchm

The `patchesm` function is very similar to the `patchm` function. The significant difference is that in `patchesm`, separate patches (delineated by `NaN`s in the inputs `lat` and `lon`) are separated and plotted as distinct patch objects on the current map axes. The advantage to this is that less memory is required. The disadvantage is that multifaced objects cannot be treated as a single object. For example, the archipelago of the Philippines cannot be treated and handled as a single Handle Graphics object.

## When Patches Are Completely Trimmed Away

Removing graphic objects that fall outside the map frame is called trimming. If, after trimming no polygons remain to be seen within it, `patchesm` creates no patches and returns an empty 1-by-0 list of handles. When this occurs, automatic reprojection of the patch data (by changing the projection or any of its parameters) is not possible. In cases where some polygons are completely trimmed away but not others, handles returned for

the trimmed polygons will be empty. No polygons or rings that have been totally trimmed away can be reprojected; to plot them again, you will need to call `patchesm` again with the original data.

## See Also
`geoshow` | `fill3m` | `fillm` | `patchm`

# patchm

Project patch objects on map axes

## Syntax

```
h = patchm(lat,lon,cdata)
h = patchm(lat,lon,cdata,PropertyName,PropertyValue,...)
h = patchm(lat,lon,PropertyName,PropertyValue,...)
h = patchm(lat,lon,z,cdata)
h = patchm(lat,lon,z,cdata, PropertyName,PropertyValue,...)
```

## Description

`h = patchm(lat,lon,cdata)` and `h = patchm(lat,lon,cdata,PropertyName,PropertyValue,...)` project and display patch (polygon) objects defined by their vertices given in `lat` and `lon` on the current map axes. `lat` and `lon` must be vectors. The color data, `cdata`, can be any color data designation supported by the standard MATLAB `patch` function. The object handle or handles, `h`, can be returned.

`h = patchm(lat,lon,PropertyName,PropertyValue,...)` allows any property name/property value pair supported by `patch` to be assigned to the `patchm` object.

`h = patchm(lat,lon,z,cdata)` and `h = patchm(lat,lon,z,cdata, PropertyName,PropertyValue,...)` allow the assignment of an altitude, `z`, to each patch object. The default altitude is `z = 0`.

## Examples

```
load coast
axesm sinusoid; framem
h = patchm(lat,long,'b');
```
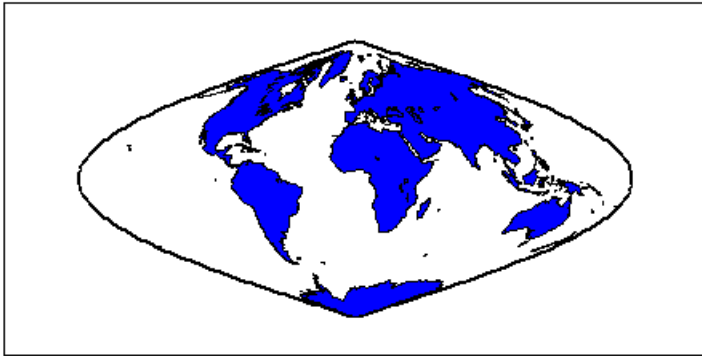
```
length(h)

ans =
      1
```

# More About

**Tips**

## How patchm Works

This Mapping Toolbox function is very similar to the standard MATLAB `patch` function. Like its analog, and unlike higher level functions such as `fillm` and `fill3m`, `patchm` adds patch objects to the current map axes regardless of hold state. Except for `XData`, `YData`, and `ZData`, all line properties and styles available through `patch` are supported by `patchm`.

## When A Patch Is Completely Trimmed Away

Removing graphic objects that fall outside the map frame is called trimming. If, after trimming to the map frame no polygons remain to be seen within it, `patchm` creates no patches and returns an empty 0-by-1 handle. When this occurs, automatic reprojection of the patch data (by changing the projection or any of its parameters) will not be possible. Instead, after changing the projection, call `patchm` again.

## See Also
patchesm | fill3m | fillm

# pcolorm

Project regular data grid on map axes in z = 0 plane

## Syntax

```
pcolorm(lat,lon,Z)
pcolorm(latlim,lonlim,Z)
pcolorm(...,prop1,val1,prop2,val2,...)
h = pcolorm(...)
```

## Description

pcolorm(lat,lon,Z) constructs a surface to represent the data grid Z in the current map axes. The surface lies flat in the horizontal plane with its CData property set to Z. Lat and lon are vectors or 2-D arrays that define the latitude-longitude graticule mesh on which Z is displayed. For a complete description of the various forms that lat and lon can take, see surfm. If the hold state is 'off', pcolorm clears the current map.

pcolorm(latlim,lonlim,Z) defines the graticule using the latitude and longitude limits latlim and lonlim. These limits should match the geographic extent of Z, the data grid. Latlim is a two-element vector of the form:

[southern_limit northern_limit]

Likewise, lonlim has the form:

[western_limit eastern_limit]

A latitude-longitude graticule of size 50-by-100 is constructed. The surface FaceColor property is 'texturemap', except when Z is precisely 50-by-100, in which case it is 'flat'.

pcolorm(...,prop1,val1,prop2,val2,...) applies additional MATLAB graphics properties to the surface via property/value pairs. Any property accepted by the surface may be specified, except for XData, YData, and ZData.

h = pcolorm(...) returns a handle to the surface object.

## Examples

Construct a surface to represent the data grid `topo`.

```
figure('Color','white')
load topo
axesm miller
axis off; framem on; gridm on;
[lat lon] = meshgrat(topo,topolegend,[90 180]);
pcolorm(lat,lon,topo)
demcmap(topo)
tightmap
```
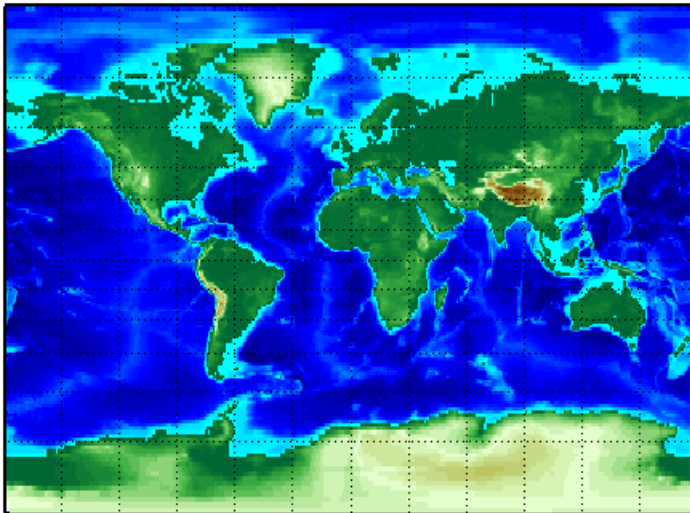


## More About

### Tips

This function warps a data grid to a graticule mesh, which is projected according to the map axes property `MapProjection`. The fineness, or resolution, of this grid determines the quality of the projection and the speed of plotting it. There is no hard and fast rule for sufficient graticule resolution, but in general, cylindrical projections need fewer graticule points in the longitudinal direction than do complex curve-generating projections.

## See Also

geoshow | meshgrat | meshm | surfacem | surfm

# pix2latlon

Convert pixel coordinates to latitude-longitude coordinates

## Syntax

```
[lat, lon] = pix2latlon(R,row,col)
```

## Description

`[lat, lon] = pix2latlon(R,row,col)` calculates latitude-longitude coordinates `lat`, `lon` from pixel coordinates `row`, `col`. `R` is either a 3-by-2 referencing matrix that transforms intrinsic pixel coordinates to geographic coordinates, or a geographic raster reference object. `row` and `col` are vectors or arrays of matching size. The outputs `lat` and `lon` have the same size as `row` and `col`.

## Examples

Find the latitude and longitude of the upper left outer corner of a 2-by-2 degree gridded data set.

```
R = makerefmat(1, 89, 2, 2);
[UL_lat, UL_lon] = pix2latlon(R, .5, .5)
```

The output appears as follows:

```
UL_lat =
    88
UL_lon =
     0
```

Find the latitude and longitude of the lower right outer corner of a 2-by-2 degree gridded data set.

```
[LR_lat, LR_lon] = pix2latlon(R, 90.5, 180.5)
```

The output appears as follows:

```
LR_lat =
    268
LR_lon =
    360
```

## See Also
```
latlon2pix | pix2map | makerefmat
```

# pix2map

Convert pixel coordinates to map coordinates

## Syntax

```
[x,y] = pix2map(R,row,col)
s = pix2map(R,row,col)
[...] = pix2map(R,p)
```

## Description

`[x,y] = pix2map(R,row,col)` calculates map coordinates `x,y` from pixel coordinates `row,col`. `R` is either a 3-by-2 referencing matrix defining a two-dimensional affine transformation from intrinsic pixel coordinates to map coordinates, or a map raster reference object. `row` and `col` are vectors or arrays of matching size. The outputs `x` and `y` have the same size as `row` and `col`.

`s = pix2map(R,row,col)` combines `x` and `y` into a single array `s`. If `row` and `col` are column vectors of length n, then `s` is an n-by-2 matrix and each row (`s(k,:)`) specifies the map coordinates of a single point. Otherwise, `s` has size `[size(row) 2]`, and `s(k1,k2,...,kn,:)` contains the map coordinates of a single point.

`[...] = pix2map(R,p)` combines `row` and `col` into a single array `p`. If `row` and `col` are column vectors of length n, then `p` should be an n-by-2 matrix such that each row (`p(k,:)`) specifies the pixel coordinates of a single point. Otherwise, `p` should have size `[size(row) 2]`, and `p(k1,k2,...,kn,:)` should contain the pixel coordinates of a single point.

## Examples

```
% Find the map coordinates for the pixel at (100,50).
R = worldfileread('concord_ortho_w.tfw');
[x,y] = pix2map(R,100,50);
```

## See Also
makerefmat | map2pix | pix2latlon | worldfileread

# pixcenters

Compute pixel centers for georeferenced image or data grid

## Syntax

```
[x,y] = pixcenters(R, height, width)
[x,y] = pixcenters(r,sizea)
[x,y] = pixcenters(..., 'makegrid')
```

## Description

`[x,y] = pixcenters(R, height, width)` returns the spatial coordinates of a spatially-referenced image or regular gridded data set. `R` is either a 3-by-2 referencing matrix defining a 2-dimensional affine transformation from intrinsic pixel coordinates to map coordinates, or a map.rasterref.MapCellsReference object. `height` and `width` are the image dimensions. If `r` does not include a rotation (i.e., `r(1,1) = r(2,2) = 0`), then `x` is a 1-by-`width` vector and `y` is a 1-by-`height` vector. In this case, the spatial coordinates of the pixel in row `row` and column `col` are given by `x(col)`, `y(row)`. Otherwise, `x` and `y` are each a `height`-by-`width` matrix such that `x(col,row)`, `y(col,row)` are the coordinates of the pixel with subscripts `(row,col)`.

`[x,y] = pixcenters(r,sizea)` accepts the size vector `sizea = [height, width, ...]` instead of `height` and `width`.

`[x,y] = pixcenters(info)` accepts a scalar struct array with the fields

| | |
|---|---|
| `'RefMatrix'` | 3-by-2 referencing matrix |
| `'Height'` | Scalar number |
| `'Width'` | Scalar number |

`[x,y] = pixcenters(..., 'makegrid')` returns `x` and `y` as `height`-by-`width` matrices even if `r` is irrotational. This syntax can be helpful when you call `pixcenters` from within a function or script.

## Examples

```
[Z,R] = arcgridread('MtWashington-ft.grd');
[x,y] = pixcenters(R, size(Z));
h = surf(x,y,Z); axis equal; demcmap(Z)
set(h,'EdgeColor','none')
xlabel('x (easting in meters)')
ylabel('y (northing in meters)')
zlabel('elevation in feet')
```

## More About

### Tips

For more information on referencing matrices, see the `makerefmat` reference page.

`pixcenters` is useful for working with `surf`, `mesh`, or `surface`, and for coordinate transformations.

## See Also

map.rasterref.MapCellsReference | `arcgridread` | `makerefmat` | `mapbbox` | `mapoutline` | `mapshow` | `pix2map` | `worldfileread`

# plabel

Toggle and control display of parallel labels

## Syntax

```
plabel
plabel('on')
plabel('off')
plabel(meridian)
plabel(MapAxesPropertyName,PropertyValue,...)
```

## Description

`plabel` toggles the visibility of parallel labeling on the current map axes.

`plabel('on')` sets the visibility of parallel labels to `'on'`.

`plabel('off')` sets the visibility of parallel labels to `'off'`.

`plabel('reset')` resets the displayed parallel labels using the currently defined parallel label properties.

`plabel(meridian)` sets the value of the `PLabelMeridian` property of the map axes to the value meridian. This determines the meridian upon which the labels are placed (see `axesm`). The options for `meridian` are a scalar longitude or the strings `'east'`, `'west'`, or `'prime'`.

`plabel(MapAxesPropertyName,PropertyValue,...)` allows paired map axes property names and property values to be passed in. For a complete description of map axes properties, see the `axesm` reference page.

Parallel label handles can be returned in `h` if desired.

## See Also

axesm | setm | mlabel

# plot3m

Project 3-D lines and points on map axes

## Syntax

```
h = plot3m(lat,lon,z)
h = plot3m(lat,lon,linetype)
h = plot3m(lat,lon,PropertyName,PropertyValue,...)
```

## Description

`h = plot3m(lat,lon,z)` displays projected line objects on the current map axes. `lat` and `lon` are the latitude and longitude coordinates, respectively, of the line object to be projected. Note that this ordering is conceptually reversed from the MATLAB `line` function, because the *vertical* (*y*) coordinate comes first. However, the ordering latitude, then longitude, is standard geographic usage. `lat` and `lon` must be the same size, and in the `AngleUnits` of the map axes. `z` is the altitude data associated with each point in `lat` and `lon`. The object handle for the displayed line can be returned in `h`.

The units of `z` are arbitrary, except when using the Globe projection. In the case of `globe`, `z` should have the same units as the radius of the earth or semimajor axis specified in the `'geoid'` (reference ellipsoid) property of the map axes. This implies that when the reference ellipsoid is a unit sphere, the units of `z` are earth radii.

`h = plot3m(lat,lon,linetype)` allows the specification of the line style, where *linetype* is any string recognized by the MATLAB `line` function.

`h = plot3m(lat,lon,PropertyName,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB `line` function except for `XData`, `YData`, and `ZData`.

## Examples

```
axesm sinusoid; framem; view(3)
[lats,longs] = interpm([45 -45 -45 45 45 -45]',...
```

```
                              [-100 -100 100 100 -100 -100]',1);
z = (1:671)'/100;
plot3m(lats,longs,z,'m')
```
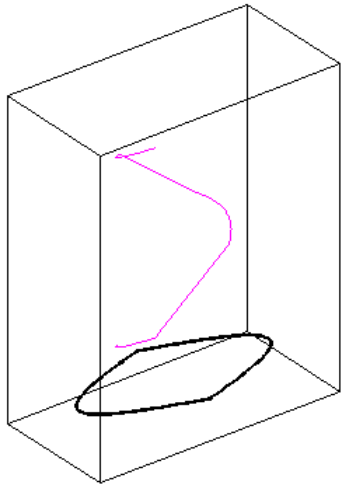


## More About

### Tips

plot3m is the mapping equivalent of the MATLAB plot3 function.

### See Also
```
linem | plot3 | plotm
```

# plotm

Project 2-D lines and points on map axes

## Syntax

```
h = plotm(lat,lon)
h = plotm(lat,lon,linetype)
h = plotm(lat,lon,PropertyName,PropertyValue,...)
h = plotm([lat lon],...)
```

## Description

`h = plotm(lat,lon)` displays projected line objects on the current map axes. `lat` and `lon` are the latitude and longitude coordinates, respectively, of the line object to be projected. Note that this ordering is conceptually reversed from the MATLAB `line` function, because the *vertical* (*y*) coordinate comes first. However, the ordering latitude, then longitude, is standard geographic usage. `lat` and `lon` must be the same size, and in the `AngleUnits` of the map axes. The object handle for the displayed line can be returned in `h`.

`h = plotm(lat,lon,linetype)` allows the specification of the line style, where *linetype* is any string recognized by the MATLAB `line` function.

`h = plotm(lat,lon,PropertyName,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB `line` function except for `XData`, `YData`, and `ZData`.

`h = plotm([lat lon],...)` allows the coordinates to be packed into a single two-column matrix.

`plotm` is the mapping equivalent of the MATLAB `plot` function.

## Examples

```
load coast
```

```
axesm sinusoid; framem
plotm(lat,long,'g')
```
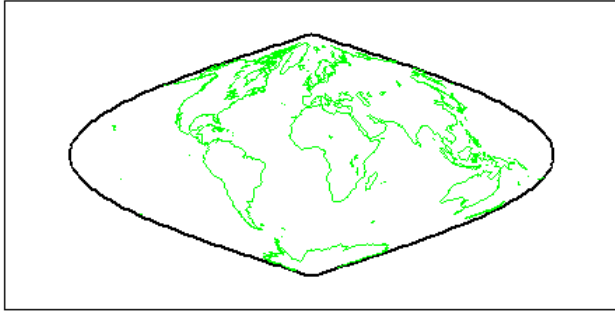


## See Also
`linem` | `plot` | `plot3m`

# polcmap

Colormaps appropriate to political regions

## Syntax

```
polcmap
polcmap(ncolors)
polcmap(ncolors,maxsat)
polcmap(ncolors,huelimits,saturationlimits,valuelimits)
cmap = polcmap(...)
```

## Description

`polcmap` applies a random muted colormap to the current figure. The size of the colormap is the same as the existing colormap.

`polcmap(ncolors)` creates a colormap with the specified number of colors.

`polcmap(ncolors,maxsat)` controls the maximum saturation of the colors. Larger maximum saturation values produce brighter, more saturated colors. If omitted, the default is 0.5.

`polcmap(ncolors,huelimits,saturationlimits,valuelimits)` controls the colors. Hue, saturation, and value are randomly selected values within the limit vectors. These are two-element vectors of the form `[min max]`. Valid values range from 0 to 1. As the hue varies from 0 to 1, the resulting color varies from red, through yellow, green, cyan, blue, and magenta, back to red. When the saturation is 0, the colors are unsaturated; they are simply shades of gray. When the saturation is 1, the colors are fully saturated; they contain no white component. As the value varies from 0 to 1, the brightness increases.

`cmap = polcmap(...)` returns the colormap without applying it to the figure.

## Examples

Draw a map of Texas and surrounding states. Color the patches with a symbolspec constructed using `polcmap`:

```
figure; usamap('texas')
states = shaperead('usastatelo.shp','UseGeoCoords',true);
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
     polcmap(numel(states))});
geoshow(states, 'DisplayType', 'polygon', ...
        'SymbolSpec', faceColors)
```
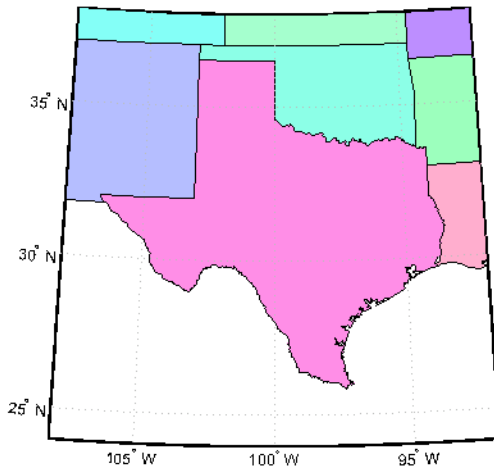


Note that the colors you obtain for this example can vary from what you see above because polcmap computes them randomly.

## More About

### Tips

You cannot use polcmap to alter the colors of displayed patches drawn by geoshow or mapshow. The patches must have been rendered by displaym. However, you can color patches using polcmap when you call geoshow or mapshow, as shown below.

## See Also

demcmap | colormap

# poly2ccw

Convert polygon contour to counterclockwise vertex ordering

## Syntax

```
[x2, y2] = poly2ccw(x1, y1)
```

## Description

`[x2, y2] = poly2ccw(x1, y1)` arranges the vertices in the polygonal contour (`x1, y1`) in counterclockwise order, returning the result in `x2` and `y2`. If `x1` and `y1` can contain multiple contours, represented either as `NaN`-separated vectors or as cell arrays, then each contour is converted to clockwise ordering. `x2` and `y2` have the same format (`NaN`-separated vectors or cell arrays) as `x1` and `y1`.

## Examples

Convert a clockwise-ordered square to counterclockwise ordering.

```
x1 = [0 0 1 1 0];
y1 = [0 1 1 0 0];
ispolycw(x1, y1)

ans =
        1

[x2, y2] = poly2ccw(x1, y1);
ispolycw(x2, y2)
ans =
        0
```

## See also

`ispolycw`, `poly2cw`, `polybool`

# poly2cw

Convert polygon contour to clockwise vertex ordering

## Syntax

```
[x2, y2] = poly2cw(x1, y1)
```

## Description

`[x2, y2] = poly2cw(x1, y1)` arranges the vertices in the polygonal contour (`x1`, `y1`) in clockwise order, returning the result in `x2` and `y2`. If `x1` and `y1` can contain multiple contours, represented either as `NaN`-separated vectors or as cell arrays, then each contour is converted to clockwise ordering. `x2` and `y2` have the same format (`NaN`-separated vectors or cell arrays) as `x1` and `y1`.

## Examples

Convert a counterclockwise-ordered square to clockwise ordering.

```
x1 = [0 1 1 0 0];
y1 = [0 0 1 1 0];
ispolycw(x1, y1)

ans =
        0

[x2, y2] = poly2cw(x1, y1);
ispolycw(x2, y2)

ans =
        1
```

## See also

`ispolycw`, `poly2ccw`, `polybool`

# poly2fv

Convert polygonal region to patch faces and vertices

## Syntax

```
[F,V] = poly2fv(x,y)
```

## Description

`[F,V] = poly2fv(x,y)` converts the polygonal region represented by the contours `(x,y)` into a faces matrix, `F`, and a vertices matrix, `V`, that can be used with the `patch` function to display the region. If the polygon represented by `x` and `y` has multiple parts, either the `NaN`-separated vector format or the cell array format may be used. The `poly2fv` function creates triangular faces.

Most Mapping Toolbox functions adhere to the convention that individual contours with clockwise-ordered vertices are external contours and individual contours with counterclockwise-ordered vertices are internal contours. Although the `poly2fv` function ignores vertex order, you should follow the convention when creating contours to ensure consistency with other functions.

## Examples

Display a rectangular region with two holes using a single patch object.

```
% External contour, rectangle.
x1 = [0 0 6 6 0];
y1 = [0 3 3 0 0];

% First hole contour, square.
x2 = [1 2 2 1 1];
y2 = [1 1 2 2 1];

% Second hole contour, triangle.
x3 = [4 5 4 4];
y3 = [1 1 2 1];
```

```
% Compute face and vertex matrices.
[f, v] = poly2fv({x1, x2, x3}, {y1, y2, y3});

% Display the patch.
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
 'EdgeColor', 'none');
axis off, axis equal
```

See the documentation for the function `polybool` for additional examples illustrating `poly2fv`.

## See also

`ispolycw`, `patch`, `poly2cw`, `poly2ccw`, `polybool`

# polybool

Set operations on polygonal regions

## Syntax

```
[x,y] = polybool(flag,x1,y1,x2,y2)
```

## Description

`[x,y] = polybool(flag,x1,y1,x2,y2)` performs the polygon set operation identified by `flag`. A valid flag string is any one of the following alternatives:

- Region intersection: `'intersection'`, `'and'`, `'&'`
- Region union: `'union'`, `'or'`, `'|'`, `'+'`, `'plus'`
- Region subtraction: `'subtraction'`, `'minus'`, `'-'`
- Region exclusive or: `'exclusiveor'` , `'xor'`

Polygons processed via `polybool` are assumed to be in a Cartesian coordinate system. The polygon inputs are `NaN`-delimited vectors, or cell arrays containing individual polygonal contours. The result is output using the same format as the input. Geographic data that encompasses a pole cannot be used directly. Use `flatearthpoly` to convert polygons that contain a pole to Cartesian coordinates.

Most Mapping Toolbox functions adhere to the convention that individual contours with clockwise-ordered vertices are external contours and individual contours with counterclockwise-ordered vertices are internal contours. Although the `polybool` function ignores vertex order, you should follow the convention when creating contours to ensure consistency with other functions.

## Examples

### Set Operations on Two Overlapping Circular Regions

```
theta = linspace(0, 2*pi, 100);
x1 = cos(theta) - 0.5;
y1 = -sin(theta);    % -sin(theta) to make a clockwise contour
```

```matlab
x2 = x1 + 1;
y2 = y1;
[xa, ya] = polybool('union', x1, y1, x2, y2);
[xb, yb] = polybool('intersection', x1, y1, x2, y2);
[xc, yc] = polybool('xor', x1, y1, x2, y2);
[xd, yd] = polybool('subtraction', x1, y1, x2, y2);

subplot(2, 2, 1)
patch(xa, ya, 1, 'FaceColor', 'r')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Union')

subplot(2, 2, 2)
patch(xb, yb, 1, 'FaceColor', 'r')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Intersection')

subplot(2, 2, 3)
% The output of the exclusive-or operation consists of disjoint
% regions.  It can be plotted as a single patch object using the
% face-vertex form.  Use poly2fv to convert a polygonal region
% to face-vertex form.
[f, v] = poly2fv(xc, yc);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
  'EdgeColor', 'none')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Exclusive Or')

subplot(2, 2, 4)
patch(xd, yd, 1, 'FaceColor', 'r')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Subtraction')
```
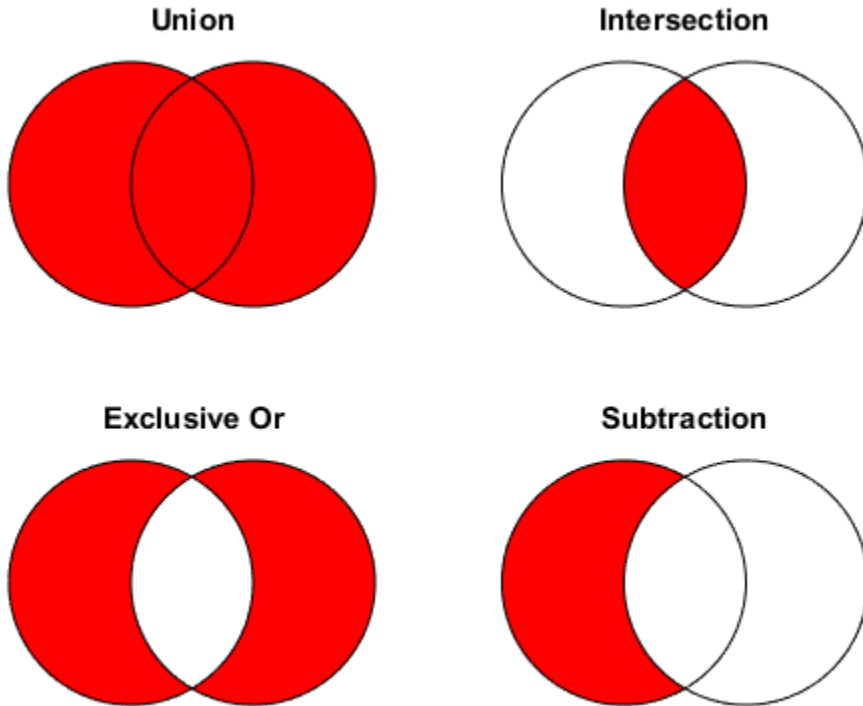
### Set Operations on Regions with Holes

```
Ax = {[1 1 6 6 1], [2 5 5 2 2], [2 5 5 2 2]};
Ay = {[1 6 6 1 1], [2 2 3 3 2], [4 4 5 5 4]};
subplot(2, 3, 1)
[f, v] = poly2fv(Ax, Ay);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
  'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Ax), plot(Ax{k}, Ay{k}, 'Color', 'k'), end
title('A')

Bx = {[0 0 7 7 0], [1 3 3 1 1], [4 6 6 4 4]};
By = {[0 7 7 0 0], [1 1 6 6 1], [1 1 6 6 1]};
```

```matlab
subplot(2, 3, 4);
[f, v] = poly2fv(Bx, By);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
  'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Bx), plot(Bx{k}, By{k}, 'Color', 'k'), end
title('B')

subplot(2, 3, 2)
[Cx, Cy] = polybool('union', Ax, Ay, Bx, By);
[f, v] = poly2fv(Cx, Cy);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
  'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Cx), plot(Cx{k}, Cy{k}, 'Color', 'k'), end
title('A \cup B')

subplot(2, 3, 3)
[Dx, Dy] = polybool('intersection', Ax, Ay, Bx, By);
[f, v] = poly2fv(Dx, Dy);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
  'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Dx), plot(Dx{k}, Dy{k}, 'Color', 'k'), end
title('A \cap B')

subplot(2, 3, 5)
[Ex, Ey] = polybool('subtraction', Ax, Ay, Bx, By);
[f, v] = poly2fv(Ex, Ey);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
  'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Ex), plot(Ex{k}, Ey{k}, 'Color', 'k'), end
title('A - B')

subplot(2, 3, 6)
[Fx, Fy] = polybool('xor', Ax, Ay, Bx, By);
[f, v] = poly2fv(Fx, Fy);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
  'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Fx), plot(Fx{k}, Fy{k}, 'Color', 'k'), end
title('XOR(A, B)')
```
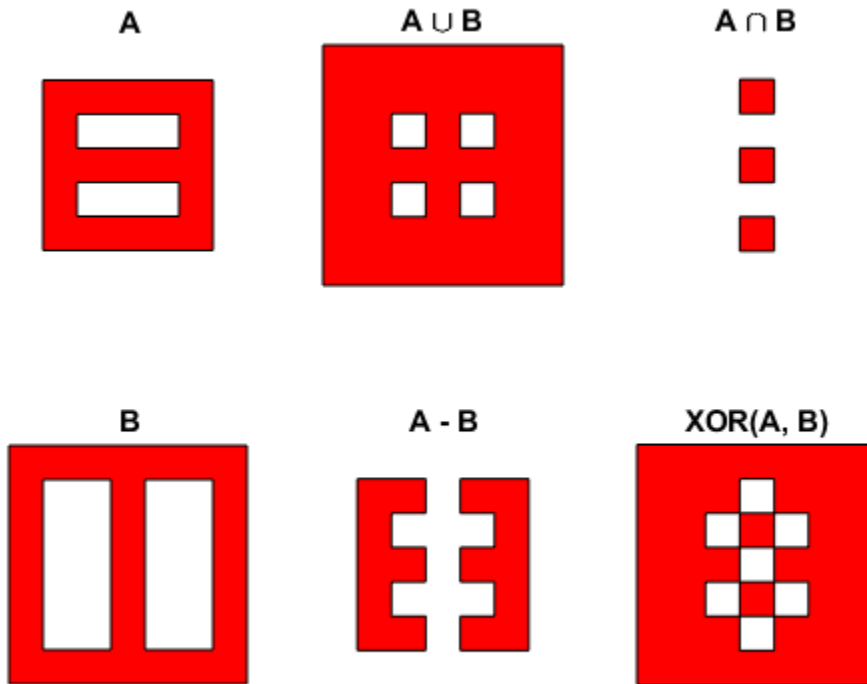
## See Also
bufferm | flatearthpoly | ispolycw | poly2cw | poly2ccw | poly2fv |
polyjoin | polysplit

# polycut

Polygon branch cuts for holes

## Syntax

```
[lat2,long2] = polycut(lat,long)
```

## Description

`[lat2,long2] = polycut(lat,long)` connects the contour and holes of polygons using optimal branch cuts. Polygons are input as `NaN`-delimited vectors, or as cell arrays containing individual polygons in each element with the outer face separated from the subsequent inner faces by `NaN`s. Multiple polygons outputs are separated by `NaN`s.

## See Also

`polybool` | `polysplit` | `polyjoin`

# polyjoin

Convert line or polygon parts from cell arrays to vector form

## Syntax

```
[lat,lon] = polyjoin(latcells,loncells)
```

## Description

`[lat,lon] = polyjoin(latcells,loncells)` converts polygons from cell array format to column vector format. In cell array format, each element of the cell array is a vector that defines a separate polygon.

## Examples

```
latcells = {[1 2 3]'; 4; [5 6 7 8 NaN 9]'};
loncells = {[9 8 7]'; 6; [5 4 3 2 NaN 1]'};
[lat,lon] = polyjoin(latcells,loncells);
[lat lon]

ans =
     1     9
     2     8
     3     7
   NaN   NaN
     4     6
   NaN   NaN
     5     5
     6     4
     7     3
     8     2
   NaN   NaN
     9     1
```

# More About

### Tips

A polygon may consist of an outer contour followed by holes separated with NaNs. In vector format, each vector may contain multiple faces separated by NaNs. There is no structural distinction between outer contours and holes in vector format.

## See Also

`polybool` | `polycut` | `polysplit`

# polymerge

Merge line segments with matching endpoints

## Syntax

```
[latMerged, lonMerged] = polymerge(lat, lon)
[latMerged, lonMerged] = polymerge(lat, lon, tol)
[latMerged, lonMerged] = polymerge(lat, lon, tol, outputFormat)
```

## Description

`[latMerged, lonMerged] = polymerge(lat, lon)` accepts a multipart line in latitude-longitude with vertices stored in arrays `lat` and `lon`, and merges the parts wherever a pair of end points coincide. For this purpose, an end point can be either the first or last vertex in a given part. When a pair of parts are merged, they are combined into a single part and the duplicate common vertex is removed. If two first vertices coincide or two last vertices coincide, then the vertex order of one of the parts will be reversed. A merge is applied anywhere that the end points of exactly two distinct parts coincide, so that an indefinite number of parts can be chained together in a single call to `polymerge`. If three or more distinct parts share a common end point, however, the choice of which parts to merge is ambiguous and therefore none of the corresponding parts are connected at that common point.

The inputs `lat` and `lon` can be column or row vectors with NaN-separated parts (and identical NaN locations in each array), or they can be cell arrays with each part in a separate cell. The form of the output arrays, `latMerged` and `lonMerged`, matches the inputs in this regard.

`[latMerged, lonMerged] = polymerge(lat, lon, tol)` combines line segments whose endpoints are separated by less than the circular tolerance, `tol`. `tol` has the same units as the polygon input.

`[latMerged, lonMerged] = polymerge(lat, lon, tol, outputFormat)` allows you to request either the NaN-separated vector form for the output (set `outputFormat` to `'vector'`), or the cell array form (set `outputFormat` to `'cell'`).

## Examples

```
lat = [1 2 3 NaN 6 7 8 9 NaN 6 5 4 3 NaN 12 13 14 ...
   NaN 9 10 11 12]';
lon = lat;
[lat2, lon2] = polymerge(lat, lon);
[lat2, lon2]

ans =

      1      1
      2      2
      3      3
      4      4
      5      5
      6      6
      7      7
      8      8
      9      9
     10     10
     11     11
     12     12
     13     13
     14     14
    NaN    NaN
```

## See Also

```
polyjoin | polysplit
```

# polysplit

Convert line or polygon parts from vector form to cell arrays

## Syntax

```
[latcells,loncells] = polysplit(lat,lon)
```

## Description

`[latcells,loncells] = polysplit(lat,lon)` returns the `NaN`-delimited segments of the vectors `lat` and `lon` as N-by-1 cell arrays with one polygon segment per cell. `lat` and `lon` must be the same size and have identically-placed `NaN`s. The polygon segments are column vectors if `lat` and `lon` are column vectors, and row vectors otherwise.

## Examples

```
lat = [1 2 3 NaN 4 NaN 5 6 7 8 9]';
lon = [9 8 7 NaN 6 NaN 5 4 3 2 1]';
[latcells,loncells] = polysplit(lat,lon);
[latcells loncells]

ans =
    [3x1 double]    [3x1 double]
    [         4]    [         6]
    [5x1 double]    [5x1 double]
```

## See Also

`isshapemultipart` | `polybool` | `polycut` | `polyjoin`

# polyxpoly

Intersection points for lines or polygon edges

## Syntax

```
[xi,yi] = polyxpoly(x1,y1,x2,y2)
[xi,yi,ii] = polyxpoly(...)
[xi,yi] = polyxpoly(...,'unique')
```

## Description

`[xi,yi] = polyxpoly(x1,y1,x2,y2)` returns the intersection points of two polylines in a planar, Cartesian system. `x1` and `y1` are vectors containing the *x*- and *y*-coordinates of the vertices in the first polyline, and `x2` and `y2` contain the vertices in the second polyline. The output variables, `xi` and `yi`, are column vectors containing the *x*- and *y*-coordinates of each point at which a segment of the first polyline intersects a segment of the second. In the case of overlapping, collinear segments, the intersection is actually a line segment rather than a point, and both endpoints are included in `xi`, `yi`.

`[xi,yi,ii] = polyxpoly(...)` returns a two-column array of line segment indices corresponding to the intersection points. The *k*-th row of `ii` indicates which polyline segments give rise to the intersection point `xi(k)`, `yi(k)`. To remember how these indices work, just think of segments and vertices as fence sections and posts. The *i*-th fence section connects the *i*-th post to the (*i*+1)-th post. In general, letting *i* and *j* denote the scalar values comprised by the *k*-th row of `ii`, the intersection indicated by that row occurs where the *i*-th segment of the first polyline intersects the *j*-th segment of the second polyline. But when an intersection falls precisely on a vertex of the first polyline, then *i* is the index of that vertex. Likewise with the second polyline and the index *j*. In the case of an intersection at the *i*-th vertex of the first line, for example, `xi(k)` equals `x1(i)` and `yi(k)` equals `y1(i)`. In the case of intersections between vertices, *i* and *j* can be interpreted as follows: the segment connecting `x1(i)`, `y1(i)` to `x1(i+1)`, `y1(i+1)` intersects the segment connecting `x2(j)`, `y2(j)` to `x2(j+1)`, `y2(j+1)` at the point `xi(k)`, `yi(k)`.

`[xi,yi] = polyxpoly(...,'unique')` filters out duplicate intersections, which may result if the input polylines are self-intersecting.

# Examples

Use the `polyxpoly` function to find the intersection points between a rectangle and a two-part polyline.

```
% Define and fill a rectangular area in the plane
xlimit = [3 13];
ylimit = [2  8];
xbox = xlimit([1 1 2 2 1]);
ybox = ylimit([1 2 2 1 1]);
mapshow(xbox,ybox,'DisplayType','polygon','LineStyle','none')

% Define and display a two-part polyline
x = [0 6  4  8 8 10 14 10 14 NaN 4 4 6 9 15];
y = [4 6 10 11 7  6 10 10  6 NaN 0 3 4 3  6];
mapshow(x,y,'Marker','+')

% Intersect the polyline with the rectangle
[xi, yi] = polyxpoly(x, y, xbox, ybox);
mapshow(xi,yi,'DisplayType','point','Marker','o')
```
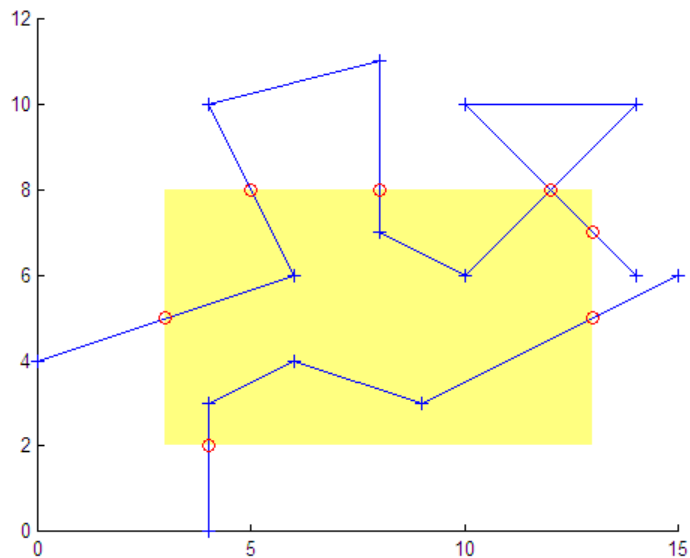
```
% Display the intersection points; note that the point (12, 8)
% appears twice because of a self-intersection near the end of
% the first part of the polyline.
[xi yi]

ans =

      3     5
      5     8
      8     8
     12     8
     12     8
     13     7
      4     2
     13     5

% You could suppress this duplicate point by using the 'unique'
% option.
[xi, yi] = polyxpoly(x, y, xbox, ybox, 'unique');
[xi yi]

ans =

      3     5
      5     8
      8     8
     12     8
     13     7
      4     2
     13     5
```

Use the `polyxpoly` function to find the intersection points between the state of California and a small circle.

```
california = shaperead('usastatehi',...
   'UseGeoCoords', true,...
   'Selector',{@(name) strcmpi(name,'California'), 'Name'});
usamap('california')
geoshow(california, 'FaceColor', 'none')

lat0 = 37; lon0 = -122; rad = 500;
[latc, lonc] = scircle1(lat0, lon0, km2deg(rad));
plotm(lat0, lon0, 'r*')
plotm(latc, lonc, 'r')
```

```
[loni, lati] = polyxpoly(lonc, latc, ...
   california.Lon', california.Lat');
plotm(lati, loni, 'bo')
```
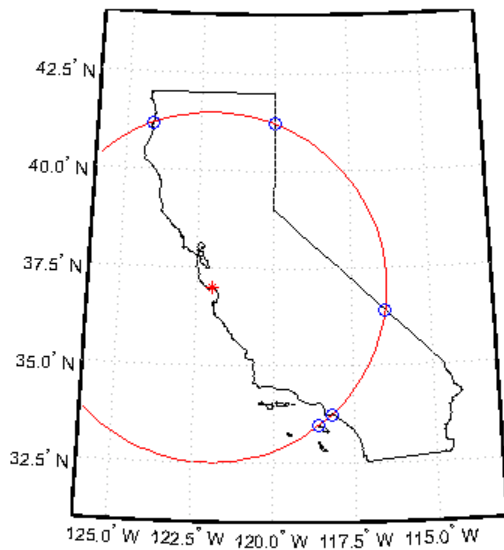


## See Also

crossfix | gcxgc | gcxsc | navfix | rhxrh | scxsc

# previewmap

View map at printed size

## Description

The appearance of a map onscreen can differ from the final printed output. This results from the difference in the size and shape of the figure window and the area the figure occupies on the printed page. A map that appears readable on screen might be cluttered when the printed output is smaller. Likewise, the relative position of multiple axes can appear different when printed. This function resizes the figure to the printed size.

## Examples

Is the text small enough to avoid overlapping in a map of Europe?

```
figure
worldmap europe
land=shaperead('landareas.shp','UseGeoCoords',true);
geoshow([land.Lat],[land.Lon])
m=gcm;
latlim = m.maplatlimit;
lonlim = m.maplonlimit;
BoundingBox = [lonlim(1) latlim(1);lonlim(2) latlim(2)];
cities=shaperead('worldcities.shp', ...
   'BoundingBox',BoundingBox,'UseGeoCoords',true);
for index=1:numel(cities)
   h=textm(cities(index).Lat, cities(index).Lon, ...
           cities(index).Name);
   trimcart(h)
   rotatetext(h)
end
orient landscape
tightmap
axis off
previewmap
```
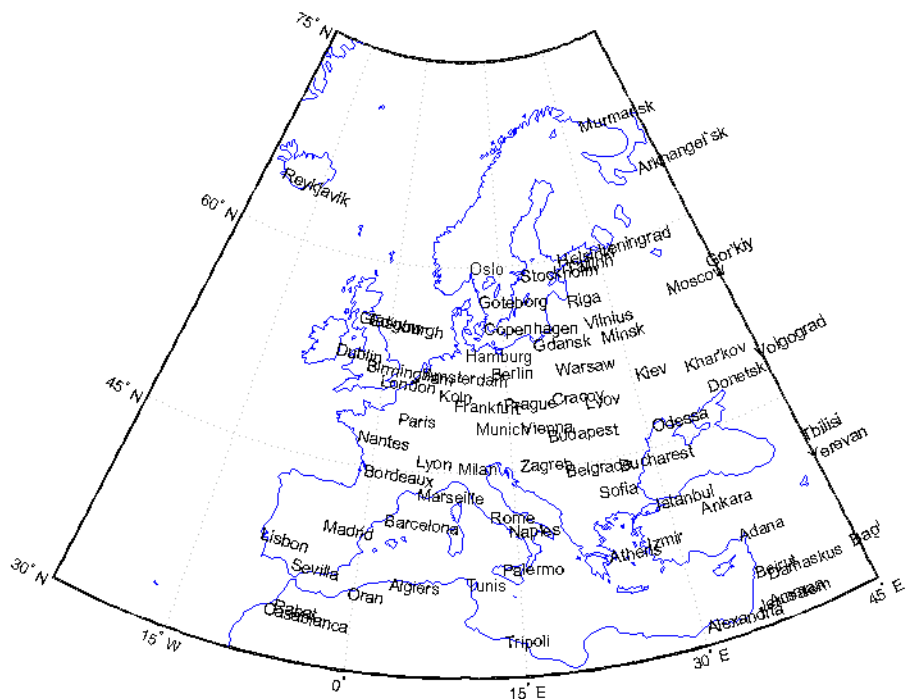
## Limitations

The figure cannot be made larger than the screen.

## More About

### Tips

`previewmap` changes the size of the current figure to match the printed output. If the resulting figure size exceeds the screen size, the figure is enlarged as much as possible.

### See Also

`printpreview` | `axesscale` | `paperscale`

# project

Project displayed map graphics object

## Compatibility

`project` will be removed in a future release.

## Syntax

```
project(h)
project(h,'xy')
project(h,'yx')
```

## Description

`project(h)` takes unprojected objects with handles `h` that are displayed on map axes and projects them. For example, `project` takes a line created on a map axes with the `plot` function and projects it as though it had been created with the `plotm` function. This can be useful if a standard MATLAB function was accidentally executed. The map structure of the existing map axes determines the specifics of the projection. If `h` is the handle of the map axes, then all the children of `h` are projected. Do not attempt this if any children of `h` have already been projected!

`project(h,'xy')` specifies that the `XData` of the unprojected objects corresponds to longitudes and the `YData` to latitudes. This is the default assumption.

`project(h,'yx')` specifies that the `XData` of the unprojected objects corresponds to latitudes and the `YData` to longitudes.

## Examples

Create an axes, plot a line, then project it:

```
axesm('bonne','AngleUnits','radians');framem;
```

```
h = plot([-1 -.5 0 .5 1],[-1 -.5  0 .5 1]);
```
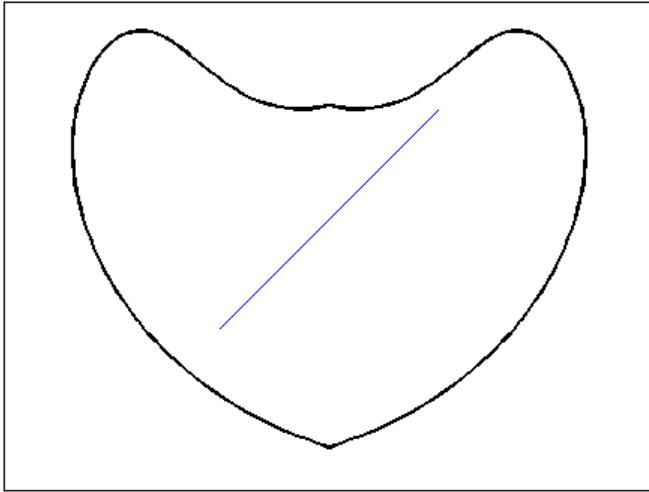


```
project(h)
```
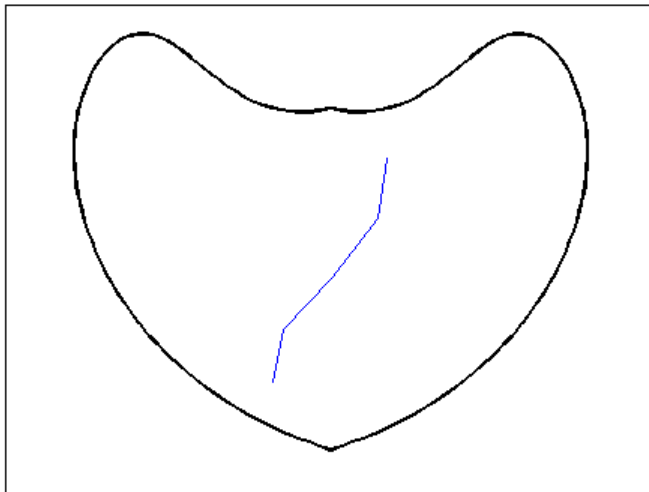


The line is straight in *x-y* space, but when converted to a projected map object, it bends with the projection.

# projfwd

Forward map projection using `PROJ.4` map projection library

## Syntax

```
[x,y] = projfwd(proj,lat,lon)
```

## Description

`[x,y] = projfwd(proj,lat,lon)` returns the x and y map coordinates from the forward projection transformation. `proj` is a structure defining the map projection. `proj` can be an mstruct or a GeoTIFF `info` structure. `lat` and `lon` are arrays of the latitude and longitude coordinates.

For a complete list of GeoTIFF `info` and map projection structures that you can use with `projinv`, see the reference page for `projlist`.

## Examples

### Overlay Boundary of Massachusetts on Orthophoto of Boston

Read vector data for state boundary of Massachusetts (in latitude and longitude).

```
S = shaperead('usastatehi', 'UseGeoCoords', true, ...
    'Selector',{@(name) strcmpi(name,'Massachusetts'), 'Name'});
```

Obtain the projection structure for the orthophoto and project the state boundary vectors to it (Massachusetts State Plane coordinate system, U.S. Survey Feet).

```
proj = geotiffinfo('boston.tif');
lat = [S.Lat];
lon = [S.Lon];
[x, y] = projfwd(proj, lat, lon);
```

Read and display the `'boston.tif'` orthophoto image.

```
[RGB, R, bbox] = geotiffread('boston.tif');
figure
mapshow(RGB, R)
```

```
xlabel('MA Mainland State Plane easting, survey feet')
ylabel('MA Mainland State Plane northing, survey feet')
```

Overlay the state boundary and set map limits to show a little more detail.

```
hold on
mapshow(gca, x, y,'Color','black','LineWidth',2.0)
set(gca,'XLim', [ 645000,  895000], ...
        'YLIm', [2865000, 3040000]);
```
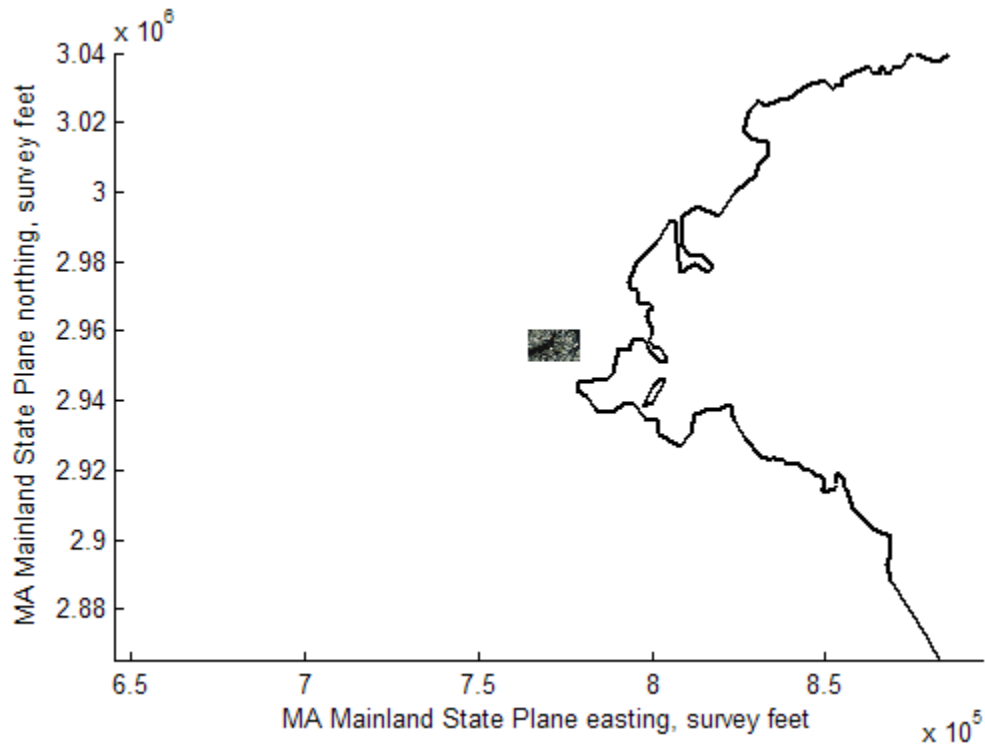


boston.tif image copyright © GeoEye, all rights reserved.

## Input Arguments

### `proj` — Map projection
scalar structure

Map projection, specified as a scalar map projection structure (`mstruct`) or GeoTIFF info structure.

Data Types: `struct`

### `lat` — Geodetic latitudes
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array, in units of degrees. Size must match the size of the `lon` input.

Data Types: `single` | `double`

### `lon` — Longitudes
scalar value | vector | matrix | N-D array

Longitudes of one or more points, specified as a scalar value, vector, matrix, or N-D array, in units of degrees. Size must match the size of the `lat` input.

Data Types: `single` | `double`

## Output Arguments

### x — Projected x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the projected coordinate system, returned as a scalar value, vector, matrix, or N-D array.

### y — Projected y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the projected coordinate system, returned as a scalar value, vector, matrix, or N-D array.

## See Also
`geotiffinfo` | `mfwdtran` | `minvtran` | `projinv` | `projlist`

# projinv

Inverse map projection using `PROJ.4` map projection library

## Syntax

```
[lat,lon] = projinv(proj,x,y)
```

## Description

`[lat,lon] = projinv(proj,x,y)` returns the latitude and longitude values from the inverse projection transformation. `proj` is a structure defining the map projection. `proj` can be a map projection mstruct or a GeoTIFF `info` structure. `x` and `y` are *x-y* map coordinate arrays. For a complete list of GeoTIFF `info` and map projection structures that you can use with `projinv`, see the reference page for `projlist`.

## Examples

### Display Boston Orthophoto on a Mercator Projection

Import the Boston roads from the shapefile and obtain the projection structure from the `'boston.tif'` orthophoto.

```
roads = shaperead('boston_roads.shp');
proj = geotiffinfo('boston.tif');
```

Convert the road coordinates to the projection's length unit. In this example, coordinates in the roads shapefile are in meters. The units of length in the projected coordinate system is `US Survey Feet`, documented in the value of the `UOMLength` field of the projection structure.

```
x = [roads.X] * unitsratio('survey feet','meter');
y = [roads.Y] * unitsratio('survey feet','meter');

[roadsLat, roadsLon] = projinv(proj, x, y);
```

Read the `boston_ovr.jpg` image and worldfile.

```
RGB = imread('boston_ovr.jpg');
R = worldfileread(getworldfilename('boston_ovr.jpg'));
```

Read state boundary vectors for Massachusetts from the `usastatehi` shapefile using a selector to eliminate other states.

```
S = shaperead('usastatehi', 'UseGeoCoords', true, ...
    'Selector',{@(name) strcmpi(name,'Massachusetts'), 'Name'});
```

Open a figure with a Mercator projection and display the image, state boundary, and roads.

```
figure
axesm('mercator')

geoshow(RGB, R)
geoshow(S.Lat, S.Lon, 'Color','red')
geoshow(roadsLat, roadsLon, 'Color', 'green')
```

Set the map boundary to the image's northern, western, and southern limits, and the eastern limit of the state boandary within the image latitude bounding box:.

```
[lon, lat] = mapoutline(R, size(RGB(:,:,1)));
ltvals = find((S.Lat>=min(lat(:))) & (S.Lat<=max(lat(:))));
setm(gca,'maplonlimit',[min(lon(:)) max(S.Lon(ltvals))], ...
         'maplatlimit',[min(lat(:)) max(lat(:))])
tightmap
```
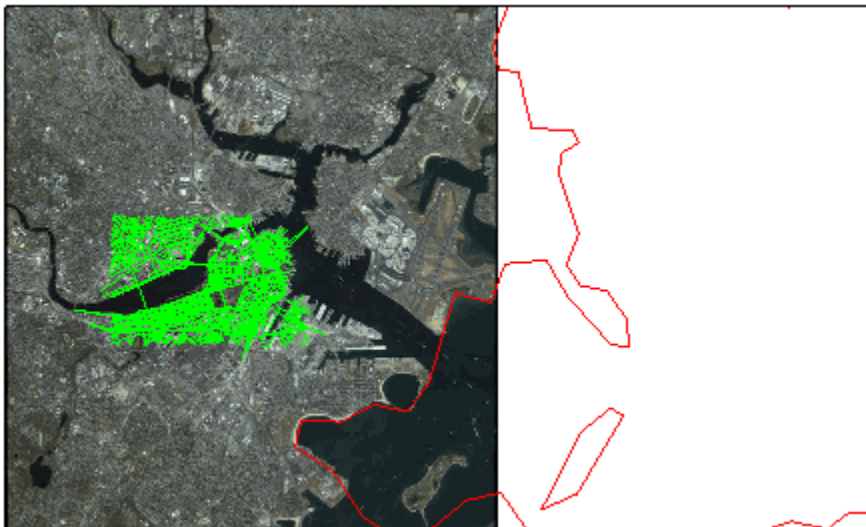
# Input Arguments

### `proj` — Map projection
scalar structure

Map projection, specified as a scalar map projection structure (`mstruct`) or GeoTIFF info structure.

Data Types: `struct`

### x — Projected x-coordinates
scalar value | vector | matrix | N-D array

x-coordinates of one or more points in the projected coordinate system, specified as a scalar value, vector, matrix, or N-D array. Size must match the size of the `y` input.

Data Types: `single` | `double`

### y — Projected y-coordinates
scalar value | vector | matrix | N-D array

y-coordinates of one or more points in the projected coordinate system, specified as a scalar value, vector, matrix, or N-D array. Size must match the size of the `x` input.

Data Types: `single` | `double`

# Output Arguments

### `lat` — Geodetic latitudes
scalar value | vector | matrix | N-D array

Geodetic latitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array, in units of degrees.

### `lon` — Longitude
scalar value | vector | matrix | N-D array

Longitudes of one or more points, returned as a scalar value, vector, matrix, or N-D array, in units of degrees.

## See Also

geotiffinfo | mfwdtran | minvtran | projfwd | projlist

# projlist

Map projections supported by `projfwd` and `projinv`

## Syntax

```
projlist(listmode)
S = projlist(listmode)
```

## Description

`projlist(`*`listmode`*`)` displays a table of projection names, IDs, and availability. *`listmode`* is a string with value `'mapprojection'`, `'geotiff'`, `'geotiff2mstruct'`, or `'all'`. The default value is `'mapprojection'`.

`S = projlist(`*`listmode`*`)` returns a structure array containing projection names, IDs, and availability. The output of `projlist` for each *`listmode`* is described below:

- `mapprojection` — Lists the map projection IDs that are available for use with `projfwd` and `projinv`. The output structure contains the fields

  - `Name` — Projection name
  - `MapProjection` — Projection ID string

- `geotiff` — Lists the GeoTIFF projection IDs that are available for use with `projfwd` and `projinv`. The output structure contains the fields

  - `GeoTIFF` — GeoTIFF projection ID string.
  - `Available` — Logical array with values 1 or 0

- `geotiff2mstruct` — Lists the GeoTIFF projection IDs that are available for use with `geotiff2mstruct`. The output structure contains the fields

  - `GeoTIFF` — GeoTIFF projection ID string
  - `MapProjection` — Projection ID string

- `all` — Lists the map and GeoTIFF projection IDs that are available for use with `projfwd` and `projinv`. The output structure contains the fields

- `GeoTIFF` — GeoTIFF projection ID string
- `MapProjection` — Projection ID string
- `info` — Logical array with values 1 or 0
- `mstruct` — Logical array with values 1 or 0

## Examples

```
s=projlist

s =
1x19 struct array with fields:
    Name
    MapProjection

s=projlist('geotiff2mstruct')

s =
1x19 struct array with fields:
    GeoTIFF
    MapProjection
```

## More About

### Tips

`projfwd` and `projinv` can be used to process certain forward or inverse map projections. These functions are implemented in C using the `PROJ.4` library. `projlist` provides a convenient list of the projections that can be used with `projfwd` or `projinv`. Because `projfwd` and `projinv` accept either a map projection structure (mstruct) or a GeoTIFF info structure, `projlist` provides separate lists for each case. It can also list the projections for which a GeoTIFF info structure can be converted to an mstruct.

## See Also

geotiff2mstruct | projfwd | projinv | maplist | maps

# putpole

Origin vector to place north pole at specified point

## Syntax

```
origin = putpole(pole)
origin = putpole(pole,units)
```

## Description

`origin = putpole(pole)` returns an `origin` vector required to transform a coordinate system in such a way as to put the true North Pole at a point specified by the three- (or two-) element vector `pole`. This vector is of the form `[latitude longitude meridian]`, specifying the coordinates in the original system at which the true North Pole is to be placed in the transformed system. The meridian is the longitude upon which the new system is to be centered, which is the new pole longitude if omitted. The output is a three-element vector of the form `[latitude longitude orientation]`, where the latitude and longitude are the coordinates in the untransformed system of the new origin, and the orientation is the azimuth of the true North Pole in the transformed system.

`origin = putpole(pole,units)` allows the specification of the angular units of the `origin` vector, where *units* is any valid angle units string. The default is `'degrees'`.

## Examples

Pull the North Pole down the 0º meridian by 30º to 60ºN. What is the resulting origin vector?

```
origin = putpole([60 0])

origin =
   30.0000         0         0
```

This makes sense: when the pole slid down 30º, the point that was 30º north of the origin slid down to become the origin. Following is a less obvious transformation:

```
origin = putpole([60 80 0])    % constrain to original central
                                % meridian

origin =
    4.9809         0   29.6217

origin = putpole([60 80 40])   % constrain to arbitrary meridian

origin =
    4.9809   40.0000   29.6217
```

## More About

### Tips

When developing transverse or oblique projections, you need transformed coordinate systems. One way to define these systems is to establish the point in the original (untransformed) system that will become the new (transformed) origin.

## See Also

neworig | org2pol

# quiver3m

Project 3-D quiver plot on map axes

## Syntax

```
h = quiver3m(lat,lon,alt,u,v,w)
h = quiver3m(lat,lon,alt,u,v,w,linespec)
h = quiver3m(lat,lon,alt,u,v,w,linespec,'filled')
h = quiver3m(lat,lon,alt,u,v,w,scale)
h = quiver3m(lat,lon,alt,u,v,w,linespec,scale)
h = quiver3m(lat,lon,alt,u,v,w,linespec,scale,'filled')
```

## Description

h = quiver3m(lat,lon,alt,u,v,w) displays *velocity* vectors with components (u,v,w) at the geographic points (lat,lon) and altitude alt on a displayed map axes. The inputs u, v, and w determine the direction of the vectors in latitude, longitude, and altitude, respectively. The function automatically determines the length of these vectors to make them as long as possible without overlap. The object handles of the displayed vectors can be returned in h.

h = quiver3m(lat,lon,alt,u,v,w,*linespec*) allows the control of the line specification of the displayed vectors with a *linespec* string recognized by the MATLAB line function. If symbols are indicated in *linespec*, they are plotted at the start points of the vectors, i.e., the input points (lat,lon,alt).

h = quiver3m(lat,lon,alt,u,v,w,*linespec*,'filled') results in the filling in of any symbols specified by *linespec*.

h = quiver3m(lat,lon,alt,u,v,w,scale), h = quiver3m(lat,lon,alt,u,v,w,linespec,scale) and h = quiver3m(lat,lon,alt,u,v,w,linespec,scale,'filled') alter the automatically calculated vector lengths by multiplying them by the scalar value scale. For example, if scale is 2, the displayed vectors are twice as long as they would be if scale were 1 (the default). When scale is set to 0, the automatic scaling is suppressed and the length of the vectors is determined by the inputs. In this case, the vectors are plotted from (lat,lon,alt) to (lat+u,lon+v,alt+w).

## Examples

Plot 3-D quiver vectors from London (51.5ºN,0º) and New Delhi (29ºN,77.5ºE), both at an altitude of 0. Suppress the automatic scaling. Terminate both vectors at an altitude of 1; the London vector should terminate 100º southward and 70º eastward, while the New Delhi vector should terminate 50º northward and 10º eastward.

```
load coast
axesm miller; view(3)
plotm(lat,long)
lat0 = [51.5,29]; lon0 = [0 77.5]; alt = [0 0];
u = [-40 50]; v = [-70 10]; w = [1 1];
quiver3m(lat0,lon0,alt,u,v,w,'m')
tightmap
```
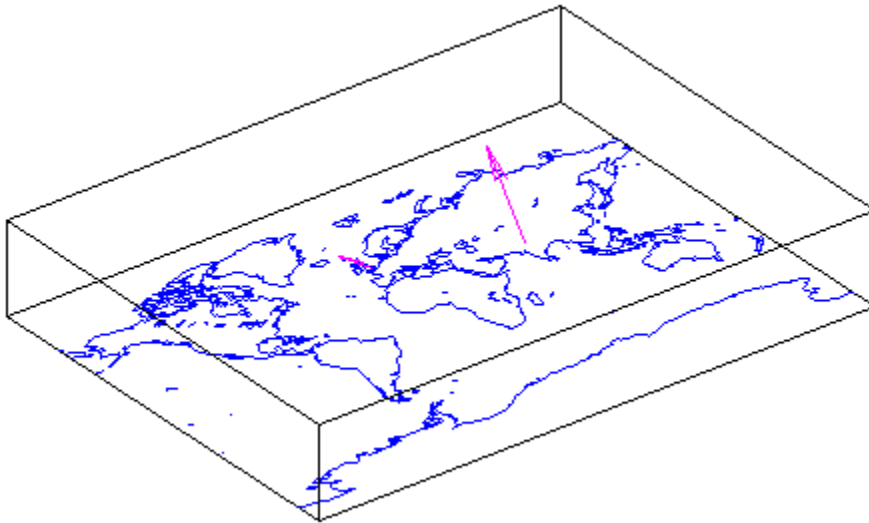


## See Also

```
quiverm | quiver3
```

# quiverm

Project 2-D quiver plot on map axes

## Syntax

```
h = quiverm(lat,lon,u,v)
h = quiverm(lat,lon,u,v,linespec)
h = quiverm(lat,lon,u,v,linespec,'filled')
h = quiverm(lat,lon,u,v,scale)
h = quiverm(lat,lon,u,v,...linespec,scale,'filled')
```

## Description

`h = quiverm(lat,lon,u,v)` displays *velocity* vectors with components (`u`,`v`) at the geographic points (`lat`,`lon`) on displayed map axes. All four inputs should be in the `AngleUnits` of the map axes. The inputs `u` and `v` determine the direction of the vectors in latitude and longitude, respectively. The function automatically determines the length of these vectors to make them as long as possible without overlap. The object handles of the displayed vectors can be returned in `h`.

`h = quiverm(lat,lon,u,v,linespec)` allows the control of the line specification of the displayed vectors with a *linespec* string recognized by the MATLAB `line` function. If symbols are indicated in *linespec*, they are plotted at the start points of the vectors, i.e., the input points (`lat`,`lon`).

`h = quiverm(lat,lon,u,v,linespec,'filled')` results in the filling in of any symbols specified by *linespec*.

`h = quiverm(lat,lon,u,v,scale)` and `h = quiverm(lat,lon,u,v,...linespec,scale,'filled')` alter the automatically calculated vector lengths by multiplying them by the scalar value `scale`. For example, if `scale` is 2, the displayed vectors are twice as long as they would be if `scale` were 1 (the default). When scale is set to 0, the automatic scaling is suppressed, and the length of the vectors is determined by the inputs. In this case, the vectors are plotted from (`lat`,`lon`) to (`lat+u`,`lon+v`).

## Examples

Plot quiver vectors from Land's End (50ºN,5.4ºW) and Majorca (39.7ºN,2.9ºE) in a direction corresponding to +5º latitude and +3º longitude. Use automatic scaling.

```
load coast
axesm('eqaconic','MapLatLimit',[30 60],'MapLonLimit',[-10 10])
framem; plotm(lat,long)
lat0 = [50 39.7]; lon0 = [-5.4 2.9];
u = [5 5]; v = [3 3];
quiverm(lat0,lon0,u,v,'r')
```
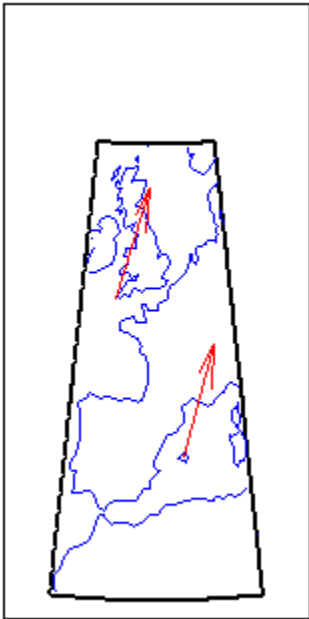


## See Also

```
quiver3m | quiver
```

# rad2km

Convert distance from radians to kilometers

## Syntax

```
km = rad2km(rad)
km = rad2km(rad,radius)
km = rad2km(rad,sphere)
```

## Description

km = rad2km(rad) converts distances from radians to kilometers as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

km = rad2km(rad,radius) converts distances from radians to kilometers as measured along a great circle on a sphere having the specified radius. radius must be in units of kilometers.

km = rad2km(rad,*sphere*) converts distances from radians to kilometers, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive.

## See Also

km2rad | deg2rad | deg2km | km2deg | km2nm | km2sm | deg2nm | nm2rad | nm2km | nm2sm | deg2sm | sm2rad | sm2km | sm2nm | rad2deg

# radtodeg

Convert angles from radians to degrees

## Syntax

```
angleInDegrees = radtodeg(angleInRadians)
```

## Description

`angleInDegrees = radtodeg(angleInRadians)` converts angle units from radians to degrees. This is both an angle conversion function and a distance conversion function, because arc length can be a measure of distance in either radians or degrees (provided the radius is known).

## Examples

There are 180° in π radians:

```
anglout = radtodeg(pi)

anglout =
    180
```

## See Also

degtorad | fromDegrees | fromRadians | toDegrees | toRadians

# rad2nm

Convert distance from radians to nautical miles

## Syntax

```
nm = rad2nm(rad)
nm = rad2nm(rad,radius)
nm = rad2nm(rad,sphere)
```

## Description

nm = rad2nm(rad) converts distances from radians to nautical miles as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

nm = rad2nm(rad,radius) converts distances from radians to nautical miles as measured along a great circle on a sphere having the specified radius. radius must be in units of nautical miles.

nm = rad2nm(rad,*sphere*) converts distances from radians to nautical miles, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive.

## See Also

km2rad | deg2rad | deg2km | km2deg | km2nm | km2sm | deg2nm | nm2rad | nm2km | nm2sm | deg2sm | sm2rad | sm2km | sm2nm | rad2deg

# rad2sm

Convert distance from radians to statute miles

## Syntax

```
sm = rad2sm(rad)
sm = rad2sm(rad,radius)
sm = rad2sm(rad,sphere)
```

## Description

`sm = rad2sm(rad)` converts distances from radians to statute miles as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`sm = rad2sm(rad,radius)` converts distances from radians to statute miles as measured along a great circle on a sphere having the specified radius. `radius` must be in units of statute miles.

`sm = rad2sm(rad,sphere)` converts distances from radians to statute miles, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

## Examples

How long is a trip around the equator in statute miles?

```
sm = rad2sm(2*pi)

sm =
   2.4874e+04
```

How about on Jupiter?

```
sm = rad2sm(2*pi,'jupiter')
```

```
sm =
   2.7283e+005
```

## See Also

km2rad | deg2rad | deg2km | km2deg | km2nm | km2sm | deg2nm | nm2rad | nm2km
| nm2sm | deg2sm | sm2rad | sm2km | sm2nm | rad2deg

# rcurve

Ellipsoidal radii of curvature

## Syntax

```
r = rcurve(ellipsoid,lat)
r = rcurve('parallel',ellipsoid,lat)
r = rcurve('meridian',ellipsoid,lat)
r = rcurve('transverse',ellipsoid,lat)
r = rcurve(..., angleunits)
```

## Description

`r = rcurve(ellipsoid,lat)` and `r = rcurve('parallel',ellipsoid,lat)` return the parallel radius of curvature at the latitude `lat` for a reference ellipsoid defined by `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. `r` is in units of length consistent with those used for the semimajor axis. `lat` is in 'degrees'.

`r = rcurve('meridian',ellipsoid,lat)` returns the meridional radius of curvature, which is the radius of curvature in the plane of a meridian at the latitude `lat`.

`r = rcurve('transverse',ellipsoid,lat)` returns the transverse radius of curvature, which is the radius of a curvature in a plane normal to the surface of the ellipsoid and normal to a meridian, at the latitude `lat`.

`r = rcurve(..., angleunits)` specifies the units of the input `lat`. `angleunits` can be 'degrees' or 'radians'.

## Examples

The radii of curvature of the default ellipsoid at 45º, in kilometers:

```
r = rcurve('transverse',referenceEllipsoid('earth','km'),...
```

```
           45,'degrees')

r =
   6.3888e+03

r = rcurve('meridian',referenceEllipsoid('earth','km'),...
           45,'degrees')

r =
   6.3674e+03

r = rcurve('parallel',referenceEllipsoid('earth','km'),...
           45,'degrees')

r =
   4.5024e+03
```

## See Also
rsphere

# readfields

Read fields or records from fixed-format files

## Syntax

```
struc = readfields(fname,fstruc)
struc = readfields(fname,fstruc,recordIDs)
struc = readfields(fname,fstruc,fieldIDs)
struc = readfields(fname,fstruc,recordIDs,mformat)
struc = readfields(fname,fstruc,recordIDs,mformat,fid)
struc = readfields(fname,fstruc,recordIDs,mformat,fid,'sparse')
```

## Description

`struc = readfields(fname,fstruc)` reads all the records from a fixed format file. `fname` is a string containing the name of the file. If it is empty, the file is selected interactively. `fstruc` is a structure defining the format of the file. The contents of `fstruc` are described below. The result is returned in a structure.

`struc = readfields(fname,fstruc,recordIDs)` reads only the records specified in the vector `recordIDs`. For example, `recordIDs = [1 2 3 4]`. All the fields in the selected records are read.

`struc = readfields(fname,fstruc,fieldIDs)` reads only the fields specified in the cell array `fieldIDs`. For example, `fieldIDs = {1 2 4}`. The selected fields are read from all the records. `fieldIDs` can be used in place of `recordIDs` in all calling forms.

`struc = readfields(fname,fstruc,recordIDs,mformat)` opens the file with the specified machine format. `mformat` must be recognized by `fopen`.

`struc = readfields(fname,fstruc,recordIDs,mformat,fid)` reads from a file that is already open. `fid` is the file identifier returned by `fopen`. The records are read starting from the current location in the file.

`struc = readfields(fname,fstruc,recordIDs,mformat,fid,'sparse')` disables error messages when the number of elements read does not agree with the stated format of the file. This is useful for formatted files with empty fields. Use `fid`

= [ ] for files that are not already open. This option is only compatible with reading selected records.

## Background

Map data is often provided as binary or ASCII files with a fixed format. Writing your own functions to read the data into the MATLAB workspace can be difficult and time-consuming, particularly for binary files. This function allows you to read the data by simply specifying the format of the file.

## Examples

Write a binary file and read it.

```
fid = fopen('testbin','wb');
for i = 1:3
 fwrite(fid,['character' num2str(i) ],'char');
 fwrite(fid,i,'int8');
 fwrite(fid,[i i],'int16');
 fwrite(fid,i,'integer*4');
 fwrite(fid,i,'real*8');
end
fclose(fid);

fs(1).length = 10;fs(1).type = 'char';fs(1).name = 'field 1';
fs(2).length = 1;fs(2).type = 'int8'; fs(2).name = 'field 2';
fs(3).length = 2;fs(3).type = 'int16';fs(3).name = 'field 3';
fs(4).length = 1;fs(4).type = 'integer*4';fs(4).name = 'field 4';
fs(5).length = 1;fs(5).type = 'float64'; fs(5).name = 'field 5';

s = readfields('testbin',fs);

s(1)
ans =
    field1: 'character1'
    field2: 1
    field3: [1 1]
    field4: 1
    field5: 1
```

## Limitations

Formatted numbers must stay within the width specified for them. Files must have a size that is an integer multiple of the computed record length. This is potentially a

problem for formatted files on DOS platforms that use a carriage return/linefeed line ending everywhere except the last record. File sizes are not checked when an open file is provided.

## More About

### Tips

The format of the file is described in the input argument `fstruc`. `fstruc` is a structure with one entry for every field in the file. `fstruc` has three required fields: `length`, `name`, and `type`. For fields containing binary data of the type that would be read by `fread`, `length` is the number of elements to be read, `name` is a string containing the field name under which the read data is stored in the output structure, and `type` is a format string recognized by `fread`. Repetition modifiers such as `'40*char'` are *not* supported. Fields with empty field names are omitted from the output.

The following `fstruc` definition is for a file with a 40-character field, a field containing two integers, and a field with a single-precision floating-point number.

```
fstruc(1).length = 40;
fstruc(1).name = 'character Field'; % spaces will be suppressed
filestruc(1).type = 'char';

fstruc(2).length = 2;
fstruc(2).name = 'integer Field';    % spaces will be suppressed
fstruc(2).type = 'int16';

fstruc(3).length = 1;
fstruc(3).name = 'float Field';      % spaces will be suppressed
fstruc(3).type = 'real*4';
```

The type can also be a `fscanf` and `sscanf`-style format string of the form `'%nX'`, where `n` is the number of characters within which the formatted data is found, and `X` is the conversion character such as `'g'` or `'d'`. For formatted fields, the length entry in `fstruc` is the number of elements, each of which has the width specified in the type string. Fortran-style double-precision output such as `'0.0D00'` can be read using a type string such as `'%nD'`, where `n` is the number of characters per element. This is an extension to the C-style format strings accepted by `sscanf`. Users unfamiliar with C should note that `'%d'` is preferred over `'%i'` for formatted integers. MATLAB syntax follows C in interpreting `'%i'` integers with leading zeros as octal. Line-ending characters in ASCII files must also be counted in the `fstruc` specification. Note that the number of line-ending characters differs across platforms.

A field specification for a formatted field with two integers each six characters wide would be of the form

```
fstruc(4).length = 2;
fstruc(4).name = 'Elevation Units';
fstruc(4).type = '%6d'
```

To summarize, `length` is the number of elements for binary numbers, the number of characters for strings, and the number of elements for formatted data.

You can omit fields from all output by providing an empty string for the `fstruc` name field.

## See Also

`grepfields` | `spcread` | `readmtx` | `textread` | `dlmread`

# readfk5

Read Fifth Fundamental Catalog of Stars

## Syntax

```
struc = readfk5(filename)
struc = readfk5(filename,struc)
```

## Compatibility

readfk5 will be removed in a future release.

## Description

struc = readfk5(*filename*) reads the FK5 file and returns the contents in a structure. Each star is an element in the structure, with the different data items stored in appropriately named fields.

struc = readfk5(*filename*,struc) appends the data in the file to the existing structure struc.

## Background

The Fifth Fundamental Catalog of Stars (FK5), Parts I and II, is a compilation of data on more than 4500 stars. The catalog contains positions, errors in positions, proper motions, and characteristics such as magnitudes, spectral types, parallaxes, and radial velocities. There are also cross-references to the identities of stars in other catalogs. It was compiled by researchers at the Astronomisches Rechen-Institut in Heidelberg.

## Examples

```
FK5 = readfk5('FK5.dat');
FK5e = readfk5('FK5_ext.dat');
whos
```

```
  Name          Size          Bytes  Class
  FK5          1x1535       5042752  struct array
  FK5e         1x3117      10226424  struct array
FK5e(1)

ans =

          FK5: 2003
          RAh: 0
          RAm: 5
          RAs: 1.1940
         pmRA: 0.6230
          DEd: 27
          DEm: 40
          DEs: 29.0100
         pmDE: -1.1100
      RAh1950: 0
      RAm1950: 2
      RAs1950: 26.5900
     pmRA1950: 0.6210
      DEd1950: 27
      DEm1950: 23
      DEs1950: 47.4400
     pmDE1950: -1.1100
     EpRA1900: 51.7200
        e_RAs: 2
       e_pmRA: 9
     EpDE1900: 46.8200
        e_DEs: 3.4000
       e_pmDE: 14
         Vmag: 6.4700
       n_Vmag: ''
       SpType: 'G5'
          plx: []
           RV: 12
        AGK3R: '38'
          SRS: ''
           HD: '225292'
           DM: 'BD+26 4744'
           GC: '48'
```

## More About

### Tips

Positions are given in terms of right ascension and declination. "Projections and Parameters" in the Concepts section of the Mapping Toolbox documentation shows how to convert these to latitude and longitude for display by the toolbox.

The Fifth Fundamental Catalog of Stars (FK5), Parts I and II data and documentation are available over the Internet by anonymous ftp.

## References

See references [5] and [6] in the Bibliography located at the end of this chapter.

### See Also
dms2degrees | scatterm

# readmtx

Read matrix stored in file

## Syntax

```
mtx = readmtx(fname,nrows,ncols,precision)
mtx = readmtx(fname,nrows,ncols,precision,readrows,readcols)
mtx = readmtx(fname,nrows,ncols,precision,...
readrows,readcols,mformat)
mtx = readmtx(fname,nrows,ncols,precision,...
readrows,readcols,mformat,nheadbytes)
mtx = readmtx(fname,nrows,ncols,precision,...
readrows,readcols,mformat,nheadbytes,nRowHeadBytes)
mtx = readmtx(fname,nrows,ncols,precision,...
readrows,readcols,mformat,nheadbytes,nRowHeadBytes,nRowTrailBytes)
mtx = readmtx(fname,nrows,ncols,precision,...
readrows,readcols,mformat,nheadbytes,nRowHeadBytes,...
nRowTrailBytes,nFileTrailBytes)
mtx = readmtx(fname,nrows,ncols,precision,...
readrows,readcols,mformat,nheadbytes,nRowHeadBytes,...
nRowTrailBytes,nFileTrailBytes,recordlen)
```

## Description

mtx = readmtx(*fname*,nrows,ncols,*precision*) reads a matrix stored in a file. The file contains only a matrix of numbers with the dimensions nrows by ncols stored with the specified *precision*. Recognized *precision* strings are described below.

mtx = readmtx(*fname*,nrows,ncols,*precision*,readrows,readcols) reads a subset of the matrix. readrows and readcols specify which rows and columns are to be read. They can be vectors containing the row or column numbers, or two-element vectors of the form [start end], which are expanded using the colon operator to start:end. To read just two rows or columns, without expansion by the colon operator, provide the indices as a column matrix.

mtx = readmtx(*fname*,nrows,ncols,*precision*,...
readrows,readcols,*mformat*) specifies the machine format used to write the file.

*mformat* can be any string recognized by `fopen`. This option is used to automatically swap bytes for files written on platforms with a different byte ordering.

mtx = readmtx(*fname*,nrows,ncols,*precision*,...
readrows,readcols,*mformat*,nheadbytes) skips the file header, whose length is specified in bytes.

mtx = readmtx(*fname*,nrows,ncols,*precision*,...
readrows,readcols,*mformat*,nheadbytes,nRowHeadBytes) also skips a header that precedes every row of the matrix. The length of the header is specified in bytes.

mtx = readmtx(*fname*,nrows,ncols,*precision*,...
readrows,readcols,*mformat*,nheadbytes,nRowHeadBytes,nRowTrailBytes) also skips a trailer that follows every row of the matrix. The length of the trailer is specified in bytes.

mtx = readmtx(*fname*,nrows,ncols,*precision*,...
readrows,readcols,*mformat*,nheadbytes,nRowHeadBytes,...
nRowTrailBytes,nFileTrailBytes)   accounts for the length of data following the matrix. The sizes of the components of the matrix are used to compute an expected file size, which is compared to the actual file size.

mtx = readmtx(*fname*,nrows,ncols,*precision*,...
readrows,readcols,*mformat*,nheadbytes,nRowHeadBytes,...
nRowTrailBytes,nFileTrailBytes,recordlen) overrides the record length calculated from the precision and number of columns, and instead uses the record length given in bytes. This is used for formatted data with extra spaces or line breaks in the matrix.

## Background

Map data is often provided as binary or ASCII files with a fixed format. Writing your own functions to read the data into the MATLAB workspace can be difficult and time-consuming, particularly for binary files. This function allows you to read the data by simply specifying the format of the file.

## Examples

Write and read a binary matrix file:

```
fid = fopen('binmat','w');
fwrite(fid,1:100,'int16');
fclose(fid);
mtx = readmtx('binmat',10,10,'int16')

mtx =
     1     2     3     4     5     6     7     8     9    10
    11    12    13    14    15    16    17    18    19    20
    21    22    23    24    25    26    27    28    29    30
    31    32    33    34    35    36    37    38    39    40
    41    42    43    44    45    46    47    48    49    50
    51    52    53    54    55    56    57    58    59    60
    61    62    63    64    65    66    67    68    69    70
    71    72    73    74    75    76    77    78    79    80
    81    82    83    84    85    86    87    88    89    90
    91    92    93    94    95    96    97    98    99   100

mtx = readmtx('binmat',10,10,'int16',[2 5],3:2:9)

mtx =
    13    15    17    19
    23    25    27    29
    33    35    37    39
    43    45    47    49
```

## Limitations

Every row of the matrix must have the same number of elements.

## More About

### Tips

This function reads files that have a general format consisting of a header, a matrix, and a trailer. Each row of the matrix can have a certain number of bytes of extraneous information preceding or following the matrix data.

Both binary and formatted data files can be read. If the file is binary, the precision argument is a format string recognized by `fread`. Repetition modifiers such as `'40*char'` are *not* supported. If the file is formatted, precision is a `fscanf` and `sscanf`-

style format string of the form `'%nX'`, where n is the number of characters within which the formatted data is found, and X is the conversion character such as `'g'` or `'d'`. Fortran-style double-precision output such as `'0.0D00'` can be read using a precision string such as `'%nD'`, where n is the number of characters per element. This is an extension to the C-style format strings accepted by sscanf. Users unfamiliar with C should note that `'%d'` is preferred over `'%i'` for formatted integers. MATLAB syntax follows C in interpreting `'%i'` integers with leading zeros as octal. Formatted files with line endings need to provide the number of trailing bytes per row, which can be 1 for platforms with carriage returns *or* linefeed (Macintosh, UNIX), or 2 for platforms with carriage returns *and* linefeeds (DOS).

## See Also

readfields | spcread | textread | dlmread

# reckon

Point at specified azimuth, range on sphere or ellipsoid

## Syntax

```
[latout,lonout] = reckon(lat,lon,arclen,az)
[latout,lonout] = reckon(lat,lon,arclen,az,units)
[latout,lonout] = reckon(lat,lon,arclen,az,ellipsoid)
[latout,lonout] = reckon(lat,lon,arclen,az,ellipsoid,units)
[latout,lonout] = reckon(track,...)
```

## Description

`[latout,lonout] = reckon(lat,lon,arclen,az)`, for scalar inputs, calculates a position (`latout,lonout`) at a given range, `arclen`, and azimuth, `az`, along a great circle from a starting point defined by `lat` and `lon`. `lat` and `lon` are in degrees. `arclen` must be expressed as degrees of arc on a sphere, and equals the length of a great circle arc connecting the point (`lat`, `lon`) to the point (`latout`, `lonout`). `az`, also in degrees, is measured clockwise from north. `reckon` calculates multiple positions when given four arrays of matching size. When given a combination of scalar and array inputs, the scalar inputs are automatically expanded to match the size of the arrays.

`[latout,lonout] = reckon(lat,lon,arclen,az,units)`, where *units* is either `'degrees'` or `'radians'`, specifies the units of the inputs and outputs, including `arclen`. The default value is `'degrees'`.

`[latout,lonout] = reckon(lat,lon,arclen,az,ellipsoid)` calculates positions along a geodesic on an ellipsoid, as specified by `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The range, `arclen`, must be expressed same unit of length as the semimajor axis of the `ellipsoid`.

`[latout,lonout] = reckon(lat,lon,arclen,az,ellipsoid,units)` calculates positions on the specified ellipsoid with `lat`, `lon`, `az`, `latout`, and `lonout` in the specified angle units.

[latout,lonout] = reckon(*track*,...) calculates positions on great circles (or geodesics) if *track* is 'gc' and along rhumb lines if *track* is 'rh'. The default value is 'gc'.

## Examples

Find the coordinates of the point 600 nautical miles northwest of London, UK (51.5ºN,0º) in a great circle sense:

```
% Convert nm distance to degrees.
dist = nm2deg(600)
dist =
    9.9933

% Northwest is 315 degrees.
pt1 = reckon(51.5,0,dist,315)
pt1 =
   57.8999  -13.3507
```

Now, determine where a plane from London traveling on a constant northwesterly course for 600 nautical miles would end up:

```
pt2 = reckon('rh',51.5,0,dist,315)

pt2 =
   58.5663  -12.3699
```

How far apart are the points above (distance in great circle sense)?

```
separation = distance('gc',pt1,pt2)

separation =
    0.8430

% Convert answer to nautical miles.
nmsep = deg2nm(separation)
nmsep =
   50.6156
```

Over 50 nautical miles separate the two points.

## See Also

azimuth | distance | km2deg | dreckon | track | track1 | track2

# reducem

Reduce density of points in vector data

## Syntax

```
[latout,lonout] = reducem(latin,lonin)
[latout,lonout] = reducem(latin,lonin,tol)
[latout,lonout,cerr] = reducem(...)
[latout,lonout,cerr,tol] = reducem(...)
```

## Description

`[latout,lonout] = reducem(latin,lonin)` reduces the number of points in vector map data. In this case the tolerance is computed automatically.

`[latout,lonout] = reducem(latin,lonin,tol)` uses the provided tolerance. The units of the tolerance are degrees of arc on the surface of a sphere.

`[latout,lonout,cerr] = reducem(...)` in addition returns a measure of the error introduced by the simplification. The output `cerr` is the difference in the arc length of the original and reduced data, normalized by the original length.

`[latout,lonout,cerr,tol] = reducem(...)` also returns the tolerance used in the reduction, which is useful when the tolerance is computed automatically.

## Examples

Compare the original and reduced outlines of the District of Columbia from the `usastatehi` state outline data:

```
dc = shaperead('usastatehi',...
    'UseGeoCoords', true,...
    'Selector',{@(name) ...
     strcmpi(name,'district of columbia'), 'Name'});
lat = extractfield(dc, 'Lat')';
lon = extractfield(dc, 'Lon')';
[latreduced, lonreduced] = reducem(lat, lon);
```

```
lonlim = dc.BoundingBox(:,1)' + [-0.02 0.02];
latlim = dc.BoundingBox(:,2)' + [-0.02 0.02];

subplot(1,2,1)
usamap(latlim, lonlim); axis off
geoshow(lat, lon,...
    'DisplayType', 'polygon', 'FaceColor', 'blue')

subplot(1,2,2)
usamap(latlim, lonlim); axis off
geoshow(latreduced, lonreduced,...
    'DisplayType', 'polygon', 'FaceColor', 'yellow')
```
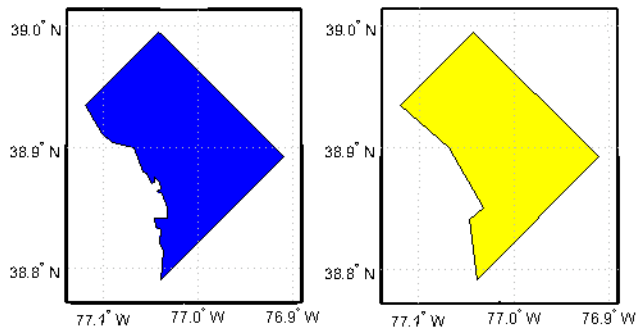


## More About

### Tips

Vector data is reduced using the Douglas-Peucker line simplification algorithm. This method recursively subdivides a polygon until a run of points can be replaced by a straight line segment, with no point in that run deviating from the straight line by more than the tolerance. The distances used to decide on which runs of points to eliminate are computed in a Plate Carrée projection.

Reduced geographic data might not always be appropriate for display. If all intermediate points in a data set are reduced, then lines appearing straight in one projection are incorrectly displayed as straight lines in others.

## See Also
interpm | resizem

# referenceEllipsoid class

Reference ellipsoid

## Description

A `referenceEllipsoid` is an `oblateSpheroid` object with three additional properties:

- A name string (`Name`).
- A string (`LengthUnit`) indicating the units of the semimajor and semiminor axes.
- A numeric code (`Code`) that matches an entry in the ellipsoid table of the EPSG/OGP Geodetic Parameter Dataset.

## Construction

`E = referenceEllipsoid` returns a reference ellipsoid object that represents the unit sphere.

`E = referenceEllipsoid(name)` returns a reference ellipsoid object corresponding to the string, `name`. The values of the `SemimajorAxis` and `SeminimorAxis` properties are in `meters`.

`E = referenceEllipsoid(code)` returns a reference ellipsoid object corresponding to the numerical EPSG code, `code`. All of the nearly 60 codes in the EPSG ellipsoid table are supported. The unit of length used for the `SemimajorAxis` and `SeminimorAxis` properties depends on the ellipsoid selected, and is indicated in the property, `E.LengthUnit`.

`E = referenceEllipsoid(name,lengthUnit)` and `E = referenceEllipsoid(code,lengthUnit)` return the ellipsoid object with the `SemimajorAxis` and `SeminimorAxis` properties in the specified unit of length, `lengthUnit`. The `lengthUnit` can be any length unit string supported by `validateLengthUnit`.

### Input Arguments

**name**

One of the short or long names listed in table.

**DataType:** String.

| EPSG Code | Short Name | Long Name |
|---|---|---|
| — | `'unitsphere'` | `'Unit Sphere'` |
| 7019 | `'grs80'` | `'Geodetic Reference System 1980'` |
| 7030 | `'wgs84'` | `'World Geodetic System 1984'` |
| 7015 | `'everest'` | `'Everest 1830'` |
| 7004 | `'bessel'` | `'Bessel 1841'` |
| 7001 | `'airy1830'` | `'Airy 1830'` |
| 7002 | `'airy1849'` | `'Airy Modified 1849'` |
| 7008 | `'clarke66'` | `'Clarke 1866'` |
| 7012 | `'clarke80'` | `'Clarke 1880'` |
| 7022 | `'international'` | `'International 1924'` |
| 7024 | `'krasovsky'` | `'Krasovsky 1940'` |
| 7043 | `'wgs72'` | `'World Geodetic System 1972'` |
| — | `'wgs60'` | `'World Geodetic System 1960'` |
| — | `'iau65'` | `'International Astronomical Union 1965'` |
| — | `'wgs66'` | `'World Geodetic System 1966'` |
| — | `'iau68'` | `'International Astronomical Union 1968'` |
| 7030 | `'earth'` | `'World Geodetic System 1984'` |
| — | `'sun'` | `'Sun'` |
| — | `'moon'` | `'Moon'` |

| EPSG Code | Short Name | Long Name |
|---|---|---|
| — | `'mercury'` | `'Mercury'` |
| — | `'venus'` | `'Venus'` |
| — | `'mars'` | `'Mars'` |
| — | `'jupiter'` | `'Jupiter'` |
| — | `'saturn'` | `'Saturn'` |
| — | `'uranus'` | `'Uranus'` |
| — | `'neptune'` | `'Neptune'` |
| — | `'pluto'` | `'Pluto'` |

**code**

Numerical EPSG code

A numerical code between 7000 and 8000 indicating a row in the EPSG ellipsoid table. All of the nearly 60 codes in the EPSG ellipsoid table are supported, in addition to the ones listed in the above table.

**lengthUnit**

Unit of length used for semimajor and semiminor axes.

## Properties

**Code**

Numerical EPSG code

A numerical code between 7000 and 8000 indicating a row in the EPSG ellipsoid table.

**Name**

Name of the reference ellipsoid

A string naming or describing the ellipsoid, for example, 'World Geodetic System 1984'.

**LengthUnit**

Unit of length string for ellipsoid axes

The empty string, or any unit of length string accepted by the `validateLengthUnit` function.

**SemimajorAxis**

Equatorial radius of spheroid, `a`

When set to a new value, the `SeminorAxis` property scales as needed to preserve the shape of the spheroid and the values of shape-related properties including `InverseFlattening` and `Eccentricity`.

The only way to change the `SemimajorAxis` property is to set it directly.

**DataType:** Positive, finite scalar.

**Default:** 1

**SeminorAxis**

Distance from center of spheroid to pole, `b`

The value is always less than or equal to `SemimajorAxis` property. When set to a new value, the `SemimajorAxis` property remains unchanged, but the shape of the spheroid changes, which is reflected in changes in the values of `InverseFlattening`, `Eccentricity`, and other shape-related properties.

**DataType:** Nonnegative, finite scalar.

**Default:** 1

**InverseFlattening**

Reciprocal of flattening

`1/f = a/(a-b)`, where `a` and `b` are semimajor and seminor axes. A value of `1/f = Inf` designates a perfect sphere. As `1/f` value approaches 1, the spheroid approaches a flattened disk. When set to a new value, other shape-related properties update, including `Eccentricity`. The `SemimajorAxis` value is unaffected by changes to `1/f`, but the value of the `SeminorAxis` property adjusts to reflect the new shape.

**DataType:** Positive scalar in the interval `[1 Inf]`.

**Default:** `Inf`

**Eccentricity**

First eccentricity of spheroid

`ecc = sqrt(a^2-b^2)/a`. A value of 0 designates a perfect sphere. When set to a new value, other shape-related properties update, including `InverseFlattening`. The `SemimajorAxis` value is unaffected by changes to `ecc`, but the value of the `SemiminorAxis` property adjusts to reflect the new shape.

**DataType:** Nonnegative scalar less than or equal to 1.

**Default:** 0

**Flattening**

Flattening of spheroid

`f = (a-b)/a`, where `a` and `b` are semimajor and semiminor axes of the spheroid.

**Access:** Read only

**ThirdFlattening**

Third flattening of spheroid

`n = (a-b)/(a+b)`, where `a` and `b` are semimajor and semiminor axes of spheroid.

**Access:** Read only

**MeanRadius**

Mean radius of spheroid, `(2*a+b)/3`

The `MeanRadius` property uses the same unit of length as the `SemimajorAxis` and `SemiminorAxis` properties.

**Access:** Read only

**SurfaceArea**

Surface area of spheroid

The `SurfaceArea` is expressed in units of area consistent with the unit of length used for the `SemimajorAxis` and `SemiminorAxis` properties.

**Access:** Read only

**Volume**

Volume of spheroid

The `Volume` is expressed in units of volume consistent with the unit of length used for the `SemimajorAxis` and `SemiminorAxis` properties.

**Access:** Read only

---

**Note:** When you define a spheroid in terms of semimajor and semiminor axes (rather than semimajor axis and inverse flattening or semimajor axis and eccentricity), a small loss of precision in the last few digits of `f`, `ecc`, and `n` is possible. This is unavoidable, but does not affect the results of practical computation.

---

# Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects

# Examples

### Construct a GRS80 reference ellipsoid

Create a reference ellipsoid object by specifying the name of the ellipsoid..

```
e = referenceEllipsoid('Geodetic Reference System 1980')

e =

referenceEllipsoid with defining properties:

              Code: 7019
              Name: 'Geodetic Reference System 1980'
        LengthUnit: 'meter'
      SemimajorAxis: 6378137
      SemiminorAxis: 6356752.31414036
   InverseFlattening: 298.257222101
       Eccentricity: 0.0818191910428158
```

```
  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Create the same reference ellipsoid object by specifying the EPSG code 7019.

```
e = referenceEllipsoid(7019)

e =

referenceEllipsoid with defining properties:

                 Code: 7019
                 Name: 'GRS 1980'
            LengthUnit: 'meter'
         SemimajorAxis: 6378137
         SemiminorAxis: 6356752.31414036
     InverseFlattening: 298.257222101
          Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

### Specify Units for GRS80 ellipsoid

Create a reference ellipsoid, specifying the unit of length. For lengh unit, you can specify any any string accepted by the `validateLengthUnit` function.

```
e = referenceEllipsoid('GRS80','km')

e =

referenceEllipsoid with defining properties:

                 Code: 7019
```

```
                  Name: 'Geodetic Reference System 1980'
             LengthUnit: 'kilometer'
          SemimajorAxis: 6378.137
          SemiminorAxis: 6356.75231414036
      InverseFlattening: 298.257222101
            Eccentricity: 0.0818191910428158

  and additional properties:

     Flattening
     ThirdFlattening
     MeanRadius
     SurfaceArea
     Volume
```

### Construct an Ellipsoid from an Input File

Create a reference ellipsoid, specifying the GeoTIFFCodes.Ellipsoid field from a file.

```
info = geotiffinfo('boston.tif');
e = referenceEllipsoid(info.GeoTIFFCodes.Ellipsoid)

e =

referenceEllipsoid with defining properties:

                  Code: 7019
                  Name: 'GRS 1980'
             LengthUnit: 'meter'
          SemimajorAxis: 6378137
          SemiminorAxis: 6356752.31414036
      InverseFlattening: 298.257222101
            Eccentricity: 0.0818191910428158

  and additional properties:

     Flattening
     ThirdFlattening
     MeanRadius
     SurfaceArea
     Volume
```

## See Also
oblateSpheroid | referenceSphere | validateLengthUnit | wgs84Ellipsoid

# referenceSphere class

Reference sphere

## Description

A `referenceSphere` object represents a sphere with a specific name and radius that you can use in map projections and other geodetic operations.

## Construction

`S = referenceSphere` returns a reference sphere object representing a unit sphere.

`S = referenceSphere(name)` returns a reference sphere object corresponding to the string, `name`, which specifies an approximately spherical body. The radius of the reference sphere is given in meters.

`S = referenceSphere(name, lengthUnit)` returns a reference sphere with radius given in the specified unit of length, `lengthUnit`.

### Input Arguments

**name**

Name of reference sphere

One of the following values: `'unit sphere'`, `'earth'`, `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, `'pluto'`. Name is case-insensitive.

**DataType:** String.

**lengthUnit**

Unit of length for radius

lengthUnit can be any length unit string supported by `validateLengthUnit`.

# Properties

**Name**

Name of reference sphere

A string naming or describing the reference sphere.

**Default:** `'Unit Sphere'`.

**LengthUnit**

Unit of length for radius

The empty string, or any unit of length string accepted by the `validateLengthUnit` function.

**Default:** `''`.

**Radius**

Radius of sphere

**DataType:** Positive, finite scalar.

**Default:** `1`.

**SemimajorAxis**

Equatorial radius of sphere `a = Radius`

The SemimajorAxis property is equal to `Radius`.

**Access:** Read only

**SemiminorAxis**

Distance from center of sphere to pole, `b = Radius`

Its value is equal to `Radius`.

**Access:** Read only

### `InverseFlattening`

Reciprocal of flattening, `1/f = Inf`

The `InverseFlattening` property provides consistency with the oblateSpheroid class. Its value is always `Inf`.

**Access:** Read only

### `Eccentricity`

First eccentricity of sphere, `ecc = 0`

The `Eccentricity` property provides consistency with the `oblateSpheroid` class. Its value is always `0`.

**Access:** Read only

### `Flattening`

Flattening of sphere, `f = 0`

The `Flattening` property provides consistency with the `oblateSpheroid` class. Its value is always `0`.

**Access:** Read only

### `ThirdFlattening`

Third flattening of sphere, `n = 0`

The `ThirdFlattening` property provides consistency with the `oblateSpheroid` class. Its value is always 0.

**Access:** Read only

### `MeanRadius`

Mean radius of sphere

The `MeanRadius` property provides consistency with the `oblateSpheroid` class. Its value is always equal to `Radius`.

**Access:** Read only

**SurfaceArea**

Surface area of sphere

The surface area of the sphere has units consistent with the `LengthUnit` property. For example, if `LengthUnit` is `'kilometer'` then `SurfaceArea` is in `square kilometers`.

**Access:** Read only

**Volume**

Volume of sphere

The volume of the sphere has units consistent with the `LengthUnit` property. For example, if `LengthUnit` is `'kilometer'` then `Volume` is in `cubic kilometers`.

**Access:** Read only

# Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects.

# Examples

### Create a Model of Earth in Kilometers

Construct a reference sphere that models the Earth as a sphere with a radius of 6371000 meters. Note that the unit of length is meters.

```
s = referenceSphere('Earth')

s =

referenceSphere with defining properties:

         Name: 'Earth'
    LengthUnit: 'meter'
        Radius: 6371000
```

```
  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Switch the unit of length in the reference sphere to kilometers.

```
s.LengthUnit = 'kilometer'

s =

referenceSphere with defining properties:

         Name: 'Earth'
    LengthUnit: 'kilometer'
       Radius: 6371

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Determine the surface area of the sphere in square kilometers.

```
s.SurfaceArea

ans =

   5.1006e+08
```

Find the volume of the sphere in cubic kilometers.

```
s.Volume
```

```
ans =
```

```
   1.0832e+12
```

## See Also

referenceEllipsoid | `validateLengthUnit`

# refmat2vec

Convert referencing matrix to referencing vector

## Compatibility

`refmat2vec` will be removed in a future release. Use `refmatToGeoRasterReference` instead, which will construct a geographic raster reference object.

## Syntax

```
refvec = refmat2vec(R,s)
```

## Description

`refvec = refmat2vec(R,s)` converts a referencing matrix, `R`, to the three-element referencing vector `refvec`. `R` is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. `s` is the size of the array (data grid) that is being referenced. `refvec` is a 1-by-3 referencing vector having elements `[cells/degree north-latitude west-longitude]` with latitude and longitude limits specified in degrees.

## Examples

```
% Verify the conversion of the geoid referencing vector to a
% referencing matrix.
load geoid;
R = refvec2mat(geoidlegend, size(geoid));
V = refmat2vec(R, size(geoid));
```

## See Also
makerefmat | refmatToGeoRasterReference

# refvec2mat

Convert referencing vector to referencing matrix

## Compatibility

`refvec2mat` will be removed in a future release. Use `refvecToGeoRasterReference` instead, which will construct a geographic raster reference object.

## Syntax

```
R = refvec2mat(refvec,s)
```

## Description

`R = refvec2mat(refvec,s)` converts a referencing vector, `refvec`, to the referencing matrix `R`. `refvec` is a 1-by-3 referencing vector having elements `[cells/degree north-latitude west-longitude]` with latitude and longitude limits specified in degrees. `s` is the size of the array (data grid) that is being referenced. `R` is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates.

## Examples

```
% Convert the geoid referencing vector to a referencing matrix
 load geoid;
R = refvec2mat(geoidlegend, size(geoid));
```

## See Also
makerefmat | refvecToGeoRasterReference

# refmatToGeoRasterReference

Referencing matrix to geographic raster reference object

## Syntax

```
R = refmatToGeoRasterReference(refmat,rasterSize)
R = refmatToGeoRasterReference( ___ ,rasterInterpretation)
R = refmatToGeoRasterReference( ___ ,funcName,varName,argIndex)
R = refmatToGeoRasterReference(Rin,rasterSize, ___ )
```

## Description

`R = refmatToGeoRasterReference(refmat,rasterSize)` constructs a cell-oriented geographic raster reference object, `R`, from a referencing matrix, `refmat`, and a size vector, `rasterSize`.

`R = refmatToGeoRasterReference( ___ ,rasterInterpretation)` uses the `rasterInterpretation` input to determine which type of geographic raster reference object to construct. The `rasterInterpretation` input indicates basic geometric nature of the raster, and can equal either `'cells'` or `'postings'`.

`R = refmatToGeoRasterReference( ___ ,funcName,varName,argIndex)` uses up to three optional arguments to provide additional information. This information is used to construct error messages if either the `refmat` or `rasterSize` inputs turn out to be invalid. Thus, you can use `refmatToGeoRasterReference` for both validating and converting a referencing matrix. The optional inputs work just like their counterparts in the MATLAB function `validateattributes`.

`R = refmatToGeoRasterReference(Rin,rasterSize, ___ )` verifies that size of the geographic raster reference object, `Rin` is consistent with the size specified by `rasterSize`, and then copies `Rin` to `R`. `refmatToGeoRasterReference` gets size information from the `Rin.RasterSize` property.

# Input Arguments

**refmat**

Any valid referencing matrix. The matrix must lead to valid latitude and longitude limits when combined with `rasterSize`, and the matrix columns and rows must be aligned with meridians and parallels, respectively.

**rasterSize**

Size vector `[M N ...]` specifying the number of rows (M) and columns (N) in the raster or image to be associated with the geographic raster reference object, `R`. For convenience, `rasterSize` may be a row vector with more than two elements. This flexibility allows you to specify the size in the following way:

```
R = refmatToGeoRasterReference(refmat, size(RGB))
```

where `RGB` is *M*-by-*N*-by-3. However, in such cases, only the first two elements of the size vector are actually used. The higher (non-spatial) dimensions are ignored.

**rasterInterpretation**

Text string that indicates basic geometric nature of the raster, either `'cells'` or `'postings'`.

**funcName**

String that specifies the name used in the formatted error message to identify the function checking the input.

**varName**

String that specifies the name used in the formatted error message to identify the referencing matrix.

**argIndex**

Positive integer that indicates the position of the referencing matrix checked in the function argument list. `refmatToGeoRasterReference` includes this information in the formatted error message.

**Rin**

Geographic raster reference object.

## Output Arguments

**R**

Geographic raster reference object.

## Examples

Convert a referencing matrix to a geographic raster reference object:

```
% Construct a referencing matrix for a regular grid that covers the
% entire globe with 1-degree cells.
rasterSize = [180 360];
refmat = makerefmat( ...
    'RasterSize', rasterSize, 'Latlim', [-90 90], ...
    'Lonlim', [0 360])

% Convert to a geographic raster reference object.
R = refmatToGeoRasterReference(refmat, rasterSize)

% For comparison, construct a referencing object directly.
georasterref( ...
    'RasterSize', rasterSize, 'Latlim', [-90 90], 'Lonlim', [0 360])
```

## See Also
```
georasterref | refvecToGeoRasterReference
```

# refmatToMapRasterReference

Referencing matrix to map raster reference object

## Syntax

```
R = refmatToMapRasterReference(refmat,rasterSize)
R = refmatToMapRasterReference( ___ ,rasterInterpretation)
R = refmatToMapRasterReference( ___ ,func_name, var_name, arg_pos)
R = refmatToMapRasterReference(Rin,rasterSize, ___ )
```

## Description

`R = refmatToMapRasterReference(refmat,rasterSize)` constructs a map raster reference object, `R`, from a referencing matrix, `refmat`, and a size vector, `rasterSize`.

`R = refmatToMapRasterReference( ___ ,rasterInterpretation)` uses the `rasterInterpretation` input to determine which type of map raster reference object to construct. The `rasterInterpretation` input indicates basic geometric nature of the raster, and can equal either `'cells'` or `'postings'`.

`R = refmatToMapRasterReference( ___ ,func_name, var_name, arg_pos)` uses up to three optional arguments to provide additional information. This information is used to construct error messages if either the `refmat` or `rasterSize` inputs turn out to be invalid. Thus, you can use `refmatToMapRasterReference` for both validating and converting a referencing matrix. The optional inputs work just `like` their counterparts in the MATLAB function `validateattributes`.

`R = refmatToMapRasterReference(Rin,rasterSize, ___ )` verifies that `Rin.RasterSize` is consistent with `rasterSize`, then copies `Rin` to `R`.

## Input Arguments

**refmat**

Referencing matrix

**rasterSize**

Size vector [M N ...] specifying the number of rows (*M*) and columns (*N*) in the raster or image to be associated with the MapRasterReference object, R. For convenience, rasterSize may be a row vector with more than two elements. This flexibility allows you to specify the size in the following way:

```
R = refmatToMapRasterReference(refmat, size(RGB))
```

where RGB is *M*-by-*N*-by-3. However, in such cases, only the first two elements of the size vector are actually used. The higher (non-spatial) dimensions are ignored.

**rasterInterpretation**

Text string that specifies basic geometric nature of the raster, either 'cells' or 'postings'.

**func_name**

String that specifies the name used in the formatted error message to identify the function checking the input.

**var_name**

String that specifies the name used in the formatted error message to identify the referencing matrix.

**arg_pos**

Positive integer that indicates the position of the referencing matrix checked in the function argument list. refmatToMapRasterReference includes this information in the formatted error message.

**Rin**

Map raster reference object.

## Output Arguments

**R**

Map raster reference object.

## Examples

Convert a referencing matrix manually versus using the `maprasterref` function.

```
% Import a referencing matrix from a world file for a
% 2000-by-2000 orthoimage referenced to the Massachusetts
% State Plane Mainland coordinate system.
refmat = worldfileread('concord_ortho_e.tfw')

% Import the corresponding TIFF image and use its size to
% help convert the referencing matrix to a referencing object.
[X, cmap] = imread('concord_ortho_e.tif');
R = refmatToMapRasterReference(refmat, size(X))

% Use the mapbbox function to obtain the map limits independently
% of the referencing object.
bbox = mapbbox(refmat, size(X))
xLimWorld = bbox(:,1)';  % Transpose the first column
yLimWorld = bbox(:,2)';  % Transpose the second column

% Construct a referencing object directly, for comparison.
maprasterref('RasterSize', size(X), 'ColumnsStartFrom', 'north', ...
  'XLimWorld', xLimWorld, 'YLimWorld', yLimWorld)
```

## See Also

maprasterref | refmatToGeoRasterReference

# refmatToWorldFileMatrix

Convert referencing matrix to world file matrix

## Syntax

```
W = refmatToWorldFileMatrix(refmat)
```

## Description

`W = refmatToWorldFileMatrix(refmat)` converts the 3-by-2 referencing matrix `refmat` to a 2-by-3 world file matrix `W`.

For a definition of a referencing matrix, see `makerefmat`.

For a definition of a world file matrix, see the `worldFileMatrix` method of the map raster reference and geographic raster reference classes.

### See Also
`makerefmat` | `worldFileMatrixToRefmat`

# refvecToGeoRasterReference

Referencing vector to geographic raster reference object

## Syntax

```
R = refvecToGeoRasterReference(refvec,rasterSize)
R = refvecToGeoRasterReference(refvec,rasterSize,funcName,varName,
argIndex)
R = refvecToGeoRasterReference(Rin,rasterSize, ___ )
```

## Description

`R = refvecToGeoRasterReference(refvec,rasterSize)` constructs a geographic raster reference object, `R`, from a referencing vector, `refvec`, and a size vector, `rasterSize`.

`R = refvecToGeoRasterReference(refvec,rasterSize,funcName,varName, argIndex)` uses up to three optional arguments to provide additional information. This information is used to construct error messages if either the `refvec` or `rasterSize` inputs turn out to be invalid. Thus, you can use `refvecToGeoRasterReference` for both validating and converting a referencing vector. The optional inputs work just like their counterparts in the MATLAB function `validateattributes`.

`R = refvecToGeoRasterReference(Rin,rasterSize, ___ )` verifies that `Rin.RasterSize` is consistent with `rasterSize`, then copies `Rin` to `R`.

## Input Arguments

**refvec**

Any valid 1-by-3 referencing vector, as long as the cell size `1/refvec(1)`, northwest corner latitude `refvec(2)`, and northwest corner longitude `refvec(3)` lead to valid latitude and longitude limits when combined with the `rasterSize` vector.

**rasterSize**

Size vector [M N ...] specifying the number of rows (M) and columns (N) in the raster or image to be associated with the GeoRasterReference object, R.

**funcName**

String that specifies the name used in the formatted error message to identify the function checking the input.

**varName**

String that specifies the name used in the formatted error message to identify the referencing vector.

**argIndex**

Positive integer that indicates the position of the referencing vector checked in the function argument list. refvecToGeoRasterReference includes this information in the formatted error message.

**Rin**

Geographic raster reference object.

## Output Arguments

**R**

Geographic raster reference object.

## Examples

Convert a referencing vector manually versus using the georasterref function.

```
% Construct a referencing vector for a regular 180-by-240 grid
% covering an area that includes the Korean Peninsula, with 12 cells
% per degree.
refvec = [12  45  115];
```

```
% Convert to a geographic raster reference object:
rasterSize = [180 240];
R = refvecToGeoRasterReference(refvec, rasterSize)

% For comparison, construct a referencing object directly:
[latlim, lonlim] = limitm(zeros(rasterSize), refvec);
georasterref('RasterSize', rasterSize, 'Latlim', latlim, ...
   'Lonlim', lonlim)
```

## See Also
georasterref | refmatToGeoRasterReference

# removeExtraNanSeparators

Clean up `NaN` separators in polygons and lines

## Syntax

```
[xdata, ydata] = removeExtraNanSeparators(xdata,ydata)
[xdata, ydata, zdata] = removeExtraNanSeparators(xdata,ydata,zdata)
```

## Description

`[xdata, ydata] = removeExtraNanSeparators(xdata,ydata)` removes `NaN`s from the vectors `xdata` and `ydata`, leaving only isolated `NaN` separators. If present, one or more leading `NaN`s are removed entirely. If present, a single trailing `NaN` is preserved. `NaN`s are removed, but never added, so if the input lacks a trailing `NaN`, so will the output. `xdata` and `ydata` must match in size and have identical `NaN` locations.

`[xdata, ydata, zdata] = removeExtraNanSeparators(xdata,ydata,zdata)` removes `NaN`s from the vectors `xdata`, `ydata`, and `zdata`, leaving only isolated `NaN` separators and optionally, if consistent with the input, a single trailing `NaN`.

## Examples

```
xin = [NaN NaN 1:3 NaN 4:5 NaN NaN NaN 6:9 NaN NaN];
yin = xin;
[xout, yout] = removeExtraNanSeparators(xin, yin);
xout

xout =
  1  2  3  NaN  4  5 NaN  6  7  8  9  NaN

xin = [NaN 1:3 NaN NaN 4:5 NaN NaN NaN 6:9]'
yin = xin;
zin = xin;
[xout, yout, zout] = removeExtraNanSeparators(xin, yin, zin);
xout
```

```
xout =
      1
      2
      3
    NaN
      4
      5
    NaN
      6
      7
      8
      9
```

# resizem

Resize regular data grid

## Syntax

```
Z = resizem(Z1,scale)
Z = resizem(Z1,[numrows numcols])
[Z,R] = resizem(Z1,scale,R1)
[Z,R] = resizem(Z1,[numrows numcols],R1)
[...] = resizem(..., method)
```

## Description

`Z = resizem(Z1,scale)` returns a regular data grid `Z` that is `scale` times the size of the input, `Z1`. `resizem` uses interpolation to resample to a new sample density/cell size. If `scale` is between 0 and 1, the size of `Z` is smaller than the size of `Z1`. If `scale` is greater than 1, the size of `Z` is larger. For example, if `scale` is 0.5, the number of rows and the number of columns will be halved. By default, `resizem` uses nearest neighbor interpolation.

`Z = resizem(Z1,[numrows numcols])` resizes `Z1` to have `numrows` rows and `numcols` columns. `numrows` and `numcols` must be positive whole numbers.

`[Z,R] = resizem(Z1,scale,R1)` or `[Z,R] = resizem(Z1,[numrows numcols],R1)` resizes a regular data grid that is spatially referenced by `R1`. `R1` can be a referencing vector, a referencing matrix, or a geographic raster reference object.

If `R1` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z1)` and its `RasterInterpretation` must be `'cells'`.

If `R1` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R1` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R1
```

If `R1` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. The output `R` will be the same type as `R1` (referencing object, vector, or matrix). If `R1` is a referencing vector, the form `[numrows numcols]` is not supported and scale must be a scalar resizing factor.

`[...] = resizem(..., method)` resizes a regular data grid using one of the following three interpolation methods:

| Method | Description |
|---|---|
| `'nearest'` | nearest neighbor interpolation (default) |
| `'bilinear'` | bilinear interpolation |
| `'bicubic'` | bicubic interpolation |

If the grid size is being reduced (`scale` is less than 1 or `[numrows numcols]` is less than the size of the input grid) and `method` is `'bilinear'` or `'bicubic'`, `resizem` applies a low-pass filter before interpolation, to reduce aliasing. The default filter size is 11-by-11. You can specify a different length for the default filter using:

`[...] = resizem(..., method, n)`
`n` is an integer scalar specifying the size of the filter, which is n-by-n. If `n` is 0 or `method` is `'nearest'`, `resizem` omits the filtering step. You can also specify your own filter `h` using:

`[...] = resizem(..., method, h)`
`h` is any two-dimensional FIR filter (such as those returned by Image Processing Toolbox functions `ftrans2`, `fwind1`, `fwind2`, or `fsamp2`). If `H` is specified, filtering is applied even when `method` is `'nearest'`.

## Examples

Double the size of a grid then reduce it using different methods:

```
Z = [1 2; 3 4]

Z =
     1   2
     3   4
```

```
neargrid = resizem(Z,2)

neargrid =
     1  1  2  2
     1  1  2  2
     3  3  4  4
     3  3  4  4

bilingrid = resizem(Z,2,'bilinear')

bilingrid =
    1.0000    1.3333    1.6667    2.0000
    1.6667    2.0000    2.3333    2.6667
    2.3333    2.6667    3.0000    3.3333
    3.0000    3.3333    3.6667    4.0000

bicubgrid = resizem(bilingrid,[3 2],'bicubic')

bicubgrid =
    0.7406    1.2994
    1.6616    2.3462
    1.9718    2.5306
```

## See Also
```
filter2
```

# restack

Restack objects within map axes

## Compatibility

restack will be removed in a future release. Use uistack instead.

## Syntax

restack(h,*position*)

## Description

restack(h,*position*) changes the stacking position of the object h within the axes. h can be a handle, a vector of handles to graphics objects, or a name string recognized by handlem. Recognized *position* strings are 'top', 'bottom', 'bot', 'up', or 'down'.

## See Also
uistack

# rhxrh

Intersection points for pairs of rhumb lines

## Syntax

```
[newlat,newlong] = rhxrh(lat1,lon1,az1,lat2,lon2,az2)
[newlat,newlon] = rhxrh(lat1,lon1,az1,lat2,lon2,az2,units)
```

## Description

`[newlat,newlong] = rhxrh(lat1,lon1,az1,lat2,lon2,az2)` returns in `newlat` and `newlon` the location of the intersection point for each pair of rhumb lines input in *rhumb line notation*. For example, the first line in the pair passes through the point (`lat1,lon1`) and has a constant azimuth of `az1`. When the two rhumb lines are identical or do not intersect (conditions that are not, in general, apparent by inspection), two `NaNs` are returned instead and a warning is displayed. The inputs must be column vectors.

`[newlat,newlon] = rhxrh(lat1,lon1,az1,lat2,lon2,az2,units)` specifies the units used, where *units* is any valid units string. The default units are `'degrees'`.

For any pair of rhumb lines, there are three possible intersection conditions: the lines are identical, they intersect once, or they do not intersect at all (except at the poles, where all nonequatorial rhumb lines meet—this is not considered an intersection). `rhxrh` does not allow multiple rhumb line intersections, although it is possible to construct cases in which such a condition occurs. See the following discussion of Limitations.

*Rhumb line notation* consists of a point on the line and the constant azimuth of the line.

## Examples

Given a starting point at (10ºN,56ºW), a plane maintains a constant heading of 35º. Another plane starts at (0º,10ºW) and proceeds at a constant heading of 310º (–50º). Where would their two paths cross each other?

```
[newlat,newlong] = rhxrh(10,-56,35,0,-10,310)
```

```
newlat =
   26.9774
newlong =
  -43.4088
```

## Limitations

Rhumb lines are specifically helpful in navigation because they represent lines of constant heading, whereas great circles have, in general, continuously changing heading. In fact, the Mercator projection was originally designed so that rhumb lines plot as straight lines, which facilitates both manual plotting with a straightedge and numerical calculations using a Cartesian planar representation. When a rhumb line proceeds off the left or right *edge* of this representation at some latitude, it reappears on the other edge at the same latitude and continues on the same slope. For rhumb lines where this occurs—for example, one with a heading of 85º—it is easy to imagine another rhumb line, say one with a heading of 0º, repeatedly intersecting the first. The real-world uses of rhumb lines make this merely an intellectual exercise, however, for in practice it is always clear which *crossing* line segment is relevant. The function rhxrh returns at most one intersection, selecting in each case that line segment containing the input starting point for its computation.

## See Also

gcxgc | gcxsc | scxsc | crossfix | polyxpoly | navfix

# rootlayr

Construct cell array of workspace variables for `mlayers` tool

## Compatibility

`rootlayr` will be removed in a future release.

## Syntax

```
rootlayr
```

## Description

`rootlayr` allows the `mlayers` tool to be used with workspace variables. It constructs a cell array that contains all the structure variables in the current workspace. This cell array is returned in the variable `ans`, which can then be an input to `mlayers`. If there is an existing variable named `ans`, it is overwritten.

The recommended calling procedure is `rootlayr;mlayers(ans);`

## Examples

`rootlayr` creates a cell array named `ans`, consisting of the three structure variables in the following workspace.

```
whos
  Name           Size          Bytes  Class
  borders        1x1           38390  struct array
  lats        2345x1           18760  double array
  lons        2345x1           18760  double array
  nation         1x1           70224  struct array
  states         1x51         254970  struct array

rootlayr
ans
```

```
ans =
   [1x1  struct]     'borders'
   [1x1  struct]     'nation'
   [1x51 struct]     'states'
```

The function mlayers(ans) can now be used to activate the mlayers tool for the
structures contained in ans.

## See Also
mlayers

# rotatem

Transform vector map data to new origin and orientation

## Syntax

```
[lat1,lon1] = rotatem(lat,lon,origin,'forward')
[lat1,lon1] = rotatem(lat,lon,origin,'inverse')
[lat1,lon1] = rotatem(lat,lon,origin,'forward',units)
[lat1,lon1] = rotatem(lat,lon,origin,'forward',units)
```

## Description

`[lat1,lon1] = rotatem(lat,lon,origin,'forward')` transforms latitude and longitude data (`lat` and `lon`) to their new coordinates (`lat1` and `lon1`) in a coordinate system resulting from Euler angle rotations as specified by `origin`. The input `origin` is a three- (or two-) element vector having the form `[latitude longitude orientation]`. The latitude and longitude are the coordinates of the point in the original system, which is the center of the output system. The orientation is the azimuth from the new origin point to the original North Pole in the new system. If `origin` has only two elements, the orientation is assumed to be 0°. This `origin` vector might be the output of `putpole` or `newpole`.

`[lat1,lon1] = rotatem(lat,lon,origin,'inverse')` transforms latitude and longitude data (`lat` and `lon`) in a coordinate system *that has been transformed* by Euler angle rotations specified by `origin` to their coordinates (`lat1` and `lon1`) in the coordinate system *from which they were originally transformed*. In a sense, this *undoes* the `'forward'` process. Be warned, however, that if data is rotated forward and then inverted, the final data might not be identical to the original. This is because of roundoff and *data collapse* at the original and intermediate singularities (the poles).

`[lat1,lon1] = rotatem(lat,lon,origin,'forward',units)` and `[lat1,lon1] = rotatem(lat,lon,origin,'forward',units)` specify the angle units of the data, where *units* is any recognized angle units string. The default is `'radians'`. Note that this default is different from that of most functions.

The `rotatem` function transforms vector map data to a new coordinate system.

An analytical use of the new data can be realized in conjunction with the `newpole` function. If a selected point is made the *north pole* of the new system, then when new vector data is created with `rotatem`, the distance of every data point from this new north pole is its new colatitude (90º minus latitude). The absolute difference in the great circle azimuths between every pair of points from their new *pole* is the same as the difference in their new longitudes.

## Examples

What are the coordinates of Rio de Janeiro (23ºS,43ºW) in a coordinate system in which New York (41ºN,74ºW) is made the North Pole? Use the `newpole` function to get the origin vector associated with putting New York at the pole:

```
nylat = 41; nylon = -74;
riolat = -23; riolon = -43;
origin = newpole(nylat,nylon);
[riolat1,riolon1] = rotatem(riolat,riolon,origin,...
                            'forward','degrees')

riolat1 =
   19.8247
riolon1 =
 -149.7375
```

What does this mean? For one thing, the colatitude of Rio in this new system is its distance from New York. Compare the distance between the original points and the new colatitude:

```
dist = distance(nylat,nylon,riolat,riolon)

dist =
   70.1753

90-riolat1

ans =
   70.1753
```

## See Also
`neworig` | `newpole` | `org2pol` | `putpole`

# rotatetext

Rotate text to projected graticule

## Syntax

```
rotatetext
rotatetext(objects)
rotatetext(objects,'inverse')
```

## Description

`rotatetext` rotates displayed text objects to account for the curvature of the graticule. The objects are selected interactively from a graphical user interface.

`rotatetext(objects)` rotates the selected objects. `objects` can be a name string recognized by `handlem` or a vector of handles to displayed text objects.

`rotatetext(objects,'inverse')` removes the rotation added by an earlier use of `rotatetext`. If omitted, `'forward'` is assumed.

## Examples

Add text to a map and rotate the text to the graticule.

```
figure
worldmap('south america')
geoshow('landareas.shp','facecolor','yellow')
cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
Santiago = strcmp('Santiago',{cities(:).Name});
h=textm(cities(Santiago).Lat, cities(Santiago).Lon, ...
     'Santiago');
rotatetext(h)
```
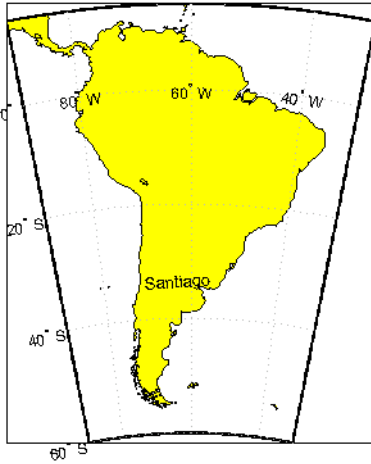
# More About

**Tips**

You can rotate meridian and parallel labels automatically by setting the map axes `LabelRotation` property to `'on'`.

# See Also

`vfwdtran` | `vinvtran`

# roundn

Round to multiple of $10^n$

## Compatibility

`roundn` is not recommended. Use `round` instead.

## Syntax

```
roundn(x,n)
```

## Description

`roundn(x,n)` rounds each element of `x` to the nearest multiple of $10^n$. `n` must be scalar, and integer-valued. For complex `x`, the imaginary and real parts are rounded independently. For `n = 0`, `roundn` gives the same result as `round`. That is, `roundn(x,0) == round(x)`.

## Examples

Round pi to the nearest hundredth.

```
roundn(pi, -2)

ans =

    3.1400
```

Round the equatorial radius of the Earth, 6378137 meters, to the nearest kilometer.

```
roundn(6378137, 3)

ans =

     6378000
```

## See Also

round

# rsphere

Radii of auxiliary spheres

## Syntax

```
r = rsphere('biaxial',ellipsoid)
r = rsphere('biaxial',ellipsoid,method)
r = rsphere('triaxial',ellipsoid)
r = rsphere('triaxial',ellipsoid,method)
r = rsphere('eqavol',ellipsoid)
r = rsphere('authalic',ellipsoid)
r = rsphere('rectifying',ellipsoid)
r = rsphere('curve',ellipsoid,lat)
r = rsphere('curve',ellipsoid,lat,method)
r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid)
r = rsphere('curve', ..., angleUnits)
r = rsphere('euler', ..., angleUnits)
```

## Description

`r = rsphere('biaxial',ellipsoid)` computes the arithmetic mean i.e., `(a
+b)/2` where `a` and `b` are the semimajor and semiminor axes of the specified ellipsoid.
`ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector
of the form `[semimajor_axis eccentricity]`.

`r = rsphere('biaxial',ellipsoid,method)` computes the arithmetic mean if the
string *method* is `'mean'` and the geometric mean, `sqrt(a*b)`, if *method* is `'norm'`.

`r = rsphere('triaxial',ellipsoid)` computes the triaxial arithmetic mean of the
semimajor axes, `a`, and semiminor axes, `b` of the ellipsoid, `(2*a+b)/3`.

`r = rsphere('triaxial',ellipsoid,method)` computes the arithmetic mean if the
string *method* is `'mean'` and the triaxial geometric mean, `(a^2*b)^(1/3)`, if *method* is
'norm'.

`r = rsphere('eqavol',ellipsoid)` returns the radius of a sphere with a volume
equal to that of the ellipsoid.

`r = rsphere('authalic',ellipsoid)` returns the radius of a sphere with a surface area equal to that of the ellipsoid.

`r = rsphere('rectifying',ellipsoid)` returns the radius of a sphere with meridional distances equal to those of the ellipsoid.

`r = rsphere('curve',ellipsoid,lat)` computes the arithmetic mean of the transverse and meridional radii of curvature at the latitude, `lat`. `lat` is in degrees.

`r = rsphere('curve',ellipsoid,lat,`*method*`)` computes an arithmetic mean if the string *'method'* is 'mean' and a geometric mean if *'method'* is 'norm'.

`r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid)` computes the Euler radius of curvature at the midpoint of the geodesic arc defined by the endpoints (`lat1,lon1`) and (`lat2,lon2`). `lat1`, `lon1`, `lat2`, and `lon2` are in degrees.

`r = rsphere('curve', ..., angleUnits)` and `r = rsphere('euler', ..., angleUnits)` use the *angleUnits* string to specify the units of the `latitude` and `longitude` inputs. *angleUnits* can be 'degrees' or 'radians'.

## Examples

Different criteria result in different spheres:

```
r = rsphere('biaxial',referenceEllipsoid('earth','km'))

r =
   6.3674e+03

r = rsphere('triaxial',referenceEllipsoid('earth','km'))

r =
   6.3710e+03

r = rsphere('curve',referenceEllipsoid('earth','km'))

r =
   6.3781e+03
```

## See Also

`rcurve`

# satbath

Read 2-minute terrain/bathymetry from Smith and Sandwell

## Syntax

```
[latgrat,longrat,z] = satbath
[latgrat,longrat,z] = satbath(scalefactor)
[latgrat,longrat,z] = satbath(scalefactor,latlim,lonlim)
[latgrat,longrat,z] = satbath(scalefactor,latlim,lonlim,gsize)
```

## Description

`[latgrat,longrat,z] = satbath` reads the global topography file for the entire world (`topo_8.2.img`), returning every $50^{th}$ point. The result is returned as a geolocated data grid. If you use a different version of the global topography file, you need to rename it to "`topo_8.2.img`". If the file is not found on the MATLAB path, a dialog opens to request the file.

`[latgrat,longrat,z] = satbath(scalefactor)` returns the data for the entire world, subsampled by the integer `scalefactor`. A `scalefactor` of 10 returns every 10th point. The matrix at full resolution has 6336 by 10800 points.

`[latgrat,longrat,z] = satbath(scalefactor,latlim,lonlim)` returns data for the specified region. The returned data extends slightly beyond the requested area. If omitted, the entire area covered by the data file is returned. The limits are two-element vectors in units of degrees, with `latlim` in the range `[-90 90]` and `lonlim` in the range `[-180 180]`.

`[latgrat,longrat,z] = satbath(scalefactor,latlim,lonlim,gsize)` controls the size of the graticule matrices. `gsize` is a two-element vector containing the number of rows and columns desired. If omitted, a graticule the size of the data grid is returned.

## Background

This is a global bathymetric model derived from ship soundings and satellite altimetry by W.H.F. Smith and D.T. Sandwell. The model was developed by iteratively adjusting

gravity anomaly data from Geosat and ERS-1 against historical track line soundings. This technique takes advantage of the fact that gravity mirrors the large variations in the ocean floor as small variations in the height of the ocean's surface. The computational procedure uses the ship track line data to calibrate the scaling between the observed surface undulations and the inferred bathymetry. Land elevations are reduced-resolution versions of GTOPO30 data.

## Examples

Read the data for the Falklands Islands (Islas Malvinas) at full resolution.

```
[latgrat,longrat,mat] = satbath(1,[-55 -50],[-65 -55]);
whos
```

```
  Name          Size         Bytes  Class

  latgrat     247x301       594776  double array
  longrat     247x301       594776  double array
  mat         247x301       594776  double array
```

## More About

### Tips

Land elevations are given in meters above mean sea level. The data is stored in a Mercator projection grid. As a result, spatial resolution varies with latitude. The grid spacing is 2 minutes (about 4 kilometers) at the equator.

This data is available over the Internet, but subject to copyright. The data file is binary, and should be transferred with no line-ending conversion or byte swapping. This function carries out any byte swapping that might be required. The data requires about 133 MB uncompressed.

The data and documentation are available over the Internet via http and anonymous ftp. Download the latest version of file `topo_x.2.img`, where x is the version number, and rename it `topo_8.w.img` for compatibility with the `satbath` function.

`satbath` returns a geolocated data grid rather than a regular data grid and a referencing vector or matrix. This is because the data is in a Mercator projection, with columns

evenly spaced in longitude, but with decreasing spacing for rows at higher latitudes. Referencing vectors and matrices assume that the number of cells per degrees of latitude and longitude are both constant across a data grid.

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

## See Also
tbase | gtopo30 | egm96geoid

# scaleruler

Add or modify graphic scale on map axes

## Syntax

```
scaleruler
scaleruler on
scaleruler off
scaleruler(property,value,...)
h = scaleruler(...)
```

## Description

`scaleruler` toggles the display of a graphic scale. If no graphic scale is currently displayed in the current map axes, one is added. If any graphic scales are currently displayed, they are removed.

`scaleruler on` adds a graphic scale to the current map axes. Multiple graphic scales can be added to the same map axes.

`scaleruler off` removes any currently displayed graphic scales.

`scaleruler(property,value,...)` adds a graphic scale and sets the properties to the values specified. You can display a list of graphic scale properties using the command `setm(h)`, where `h` is the handle to a graphic scale object. The current values for a displayed graphic scale object can be retrieved using `getm`. The properties of a displayed graphic scale object can be modified using `setm`.

`h = scaleruler(...)` returns the `hggroup` handle to the graphic scale object.

## Background

Cartographers often add graphic elements to the map to indicate its scale. Perhaps the most commonly used is the graphic scale, a ruler-like object that shows distances on the ground at the correct size for the projection.

# Examples

Create a map, add a graphic scale with the default settings, and shift it up a bit. Add a second scale showing nautical miles, and change the tick marks and direction.

```
figure
usamap('Texas')
geoshow('usastatelo.shp', 'FaceColor', [0.9 0.9 0])
scaleruler on
setm(handlem('scaleruler1'), ...
    'XLoc',-6.2e5,'YLoc',3.1e6,...
    'MajorTick',0:200:600)
scaleruler('units','nm')
setm(handlem('scaleruler2'), ...
    'YLoc', 3.0e6, ...
    'MajorTick', 0:100:300,...
    'MinorTick', 0:25:50, ...
    'TickDir', 'down', ...
    'MajorTickLength', km2nm(25),...
    'MinorTickLength', km2nm(12.5))
```
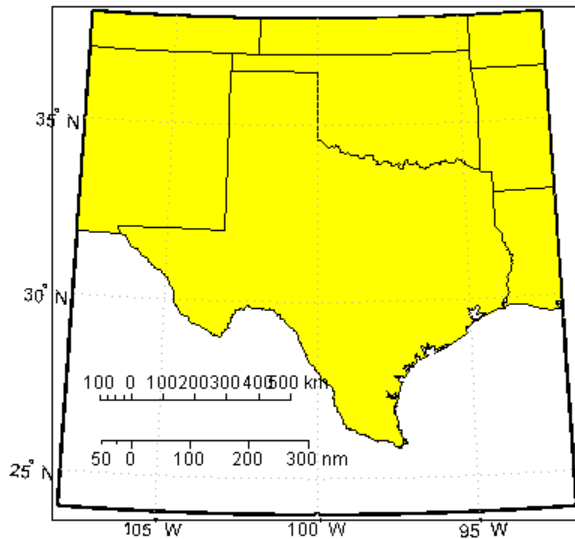
# Object Properties

## Properties That Control Appearance

Color

    `ColorSpec {no default}`

*Color of the displayed graphic scale* — Controls the color of the graphic scale lines and text. You can specify a color using a vector of RGB values or one of the MATLAB predefined names. By default, the graphic scale is displayed in black (`[0 0 0]`).

FontAngle

    `{normal} | italic | oblique`

*Angle of the graphic scale label text* — Controls the appearance of the graphic scale text components. Use any MATLAB font angle string.

FontName

    `courier | {helvetica} | symbol | times`

*Font family name for all graphic scale labels* — Sets the font for all displayed graphic scale labels. To display and print properly `FontName` must be a font that your system supports.

FontSize

    `scalar in units specified in FontUnits {9}`

*Font size* — Specifies the font size to use for all displayed graphic scale labels, in units specified by the `FontUnits` property. The default point size is 9.

FontUnits

    `inches | centimeters | normalized | {points} | pixels`

*Units used to interpret the FontSize property* — When set to normalized, the toolbox interprets the value of `FontSize` as a fraction of the height of the axes. For example, a normalized `FontSize` of 0.16 sets the text characters to a font whose height is one-tenth of the axes' height. The default units, points, are equal to 1/72 of an inch.

FontWeight

    `light | {normal} | demi | bold`

*Select bold or normal font* — The character weight for all displayed graphic scale labels.

Label

> string

> *Label text for the graphic scale* — Contains a string used to label the graphic scale. The text is displayed centered on the scale. The label is often used to indicate the scale of the map, for example "1:50,000,000."

LineWidth

> scalar {0.5}

> *Graphic scale line width* — Sets the line width of the displayed scale. The value is a scalar representing points, which is 0.5 by default.

MajorTick

> vector

> *Graphic scale major tick locations* — Sets the major tick locations for the graphic scale. The default values are chosen to give a reasonably sized scale. You can specify the locations of the tick marks by providing a vector of locations. These are usually equally spaced values as generated by `start:step:end`. The values are distances in the units of the `Units` property.

MajorTickLabel

> Cell array of strings

> *Graphic scale major tick labels* — Sets the text labels associated with the major tick locations. By default, the labels are identical to the major tick locations. You can override these by providing a cell array of strings. There must be as many strings as tick locations.

MajorTickLength

> scalar

> *Length of the major tick lines* — Controls the length of the major tick lines. The length is a distance in the units of the `Units` property.

MinorTick

> vector

> *Graphic scale minor tick locations* — Sets the minor tick locations for the graphic scale. The default values are chosen to give a reasonably sized scale. You can specify the locations of the tick marks by providing a vector of locations. These are usually

equally spaced values as generated by `start:step:end`. The values are distances in the units of the `Units` property.

MinorTickLabel

strings

*Graphic scale minor tick labels* — Sets the text labels associated with the minor tick locations. By default, the label is identical to the last minor tick location. You can override this by providing a string label.

MinorTickLength

scalar

*Length of the minor tick lines* — Controls the length of the minor tick lines. The length is a distance in the units of the `Units` property.

RulerStyle

{ruler} | lines | patches

*Style of the graphic scale* — Selects among three different kinds of graphic scale displays. The default `ruler` style looks like n axes' *x*-axis. The `lines` style has three horizontal lines across the tick marks. This type of graphic scale is often used on maps from the U.S. Geological Survey. The `patches` style has alternating black and white rectangles in place of lines and tick marks.

TickDir

{up} | down

*Direction of the tick marks and text* — Controls the direction in which the tick marks and text labels are drawn. In the default up direction, the tick marks and text labels are placed above the baseline, which is placed at the location given in the `XLoc` property. In the down position, the tick marks and labels are drawn below the baseline.

TickMode

{auto} | manual

*Tick locations mode* — Controls whether the tick locations and labels are computed automatically or are user-specified. Explicitly setting the tick labels or locations results in a `'manual'` tick mode. Setting any of the tick labels or locations to an empty matrix resets the tick mode to `'auto'`. Setting the tick mode to `'auto'` clears any explicitly specified tick locations and labels, which are then replaced by default values.

XLoc

> scalar

> *X-location of the graphic scale* — Controls the horizontal location of the graphic scale within the axes. The location is specified in the axes Cartesian projected coordinates. Use showaxes to make the Cartesian grid labels visible. You can also move the graphic scale by dragging the baseline with the mouse.

YLoc

> scalar

> *Y-location of the graphic scale* — Controls the vertical location of the graphic scale within the axes. The location is specified in the axes Cartesian projected coordinates. Use showaxes to make the Cartesian grid labels visible. You can also move the graphic scale by dragging the baseline with the mouse.

## Properties That Control Scaling

Azimuth

> scalar

> *Azimuth of scale computation* — The scale of a map varies, within the projection, with geographic location and azimuth. This property controls the azimuth along which the scaling between geographic and projected coordinates is computed. The azimuth is given in the current angle units of the map axes. The default azimuth is 0.

Lat

> scalar

> *Latitude of scale computation* — The scale of a map varies, within the projection, with geographic location and azimuth. This property controls the geographic location at which the scaling between geographic and projected coordinates is computed. The latitude is given in the current angle units of the map axes. The default location is the center of the displayed map.

Long

> scalar

> *Longitude of scale computation* — The scale of a map varies, within the projection, with geographic location and azimuth. This property controls the geographic location

at which the scaling between geographic and projected coordinates is computed. The longitude is given in the current angle units of the map axes. The default location is the center of the displayed map.

Radius

Name or radius of reference sphere

*Reference sphere name or radius* — The `radius` property controls the scaling between angular and surface distances. `radius` can be one of the strings supported by `km2deg`, or it can be the (numerical) radius of the desired sphere in the same units as the `Units` property. The default is `'earth'`.

Units

(valid distance unit strings)

*Surface distance units* — Defines the distance units displayed in the graphic scale. `Units` can be any distance unit string recognized by `unitsratio`. The distance string is also used in the last graphic scale text label.

## Other Properties

Children

(read-only)

*Name string of graphic scale elements* — Contains the tag string assigned to the graphic elements that compose the graphic scale. All elements of the graphic scale have hidden handles except the baseline. You do not normally need to access the elements directly.

# More About

### Tips

You can reposition graphic scale objects by dragging them with the mouse. You can also change their positions by modifying the `XLoc` and `YLoc` properties using `setm`.

Use `scaleruler off`, `clmo scaleruler`, or `clmo scalerulerN` to remove the scale rulers. You can also remove a graphic scale object with `delete(h)`, or `delete(handlem(`scalerulerN'))`, where `N` is the corresponding integer.

## See Also

distance | surfdist | axesscale | paperscale | mdistort | distortcalc

# scatterm

Project point markers with variable color and area

## Syntax

```
scatterm(lat,lon,s,c)
scatterm(lat,lon)
scatterm(lat,lon,s)
scatterm(...,m)
scatterm(...,'filled')
scatterm(ax,...)
h = scatterm(...)
```

## Description

scatterm(lat,lon,s,c) displays colored circles at the locations specified by the vectors lat and lon (which must be the same size). The area of each marker is determined by the values in the vector s (in points$^2$) and the colors of each marker are based on the values in c. s can be a scalar, in which case all the markers are drawn the same size, or a vector the same length as lat and lon.

When c is a vector the same length as lat and lon, the values in c are linearly mapped to the colors in the current colormap. When c is a length(lat)-by-3 matrix, the values in c specify the colors of the markers as RGB values. c can also be a color string.

scatterm(lat,lon) draws the markers in the default size and color.

scatterm(lat,lon,s) draws the markers with a single color.

scatterm(...,m) uses the marker m instead of 'o'.

scatterm(...,'filled') fills the markers.

scatterm(ax,...) plots into axes ax instead of gca. ax is a handle to a map axes.

h = scatterm(...) returns a handle to an hggroup.

## Examples

Plot the `seamount` MATLAB data as symbols with the color proportional to the height.

```
load seamount
worldmap([-49 -47.5],[-150 -147.5])
scatterm(y,x,5,z)
scaleruler
set(gca,'Visible','off')
```
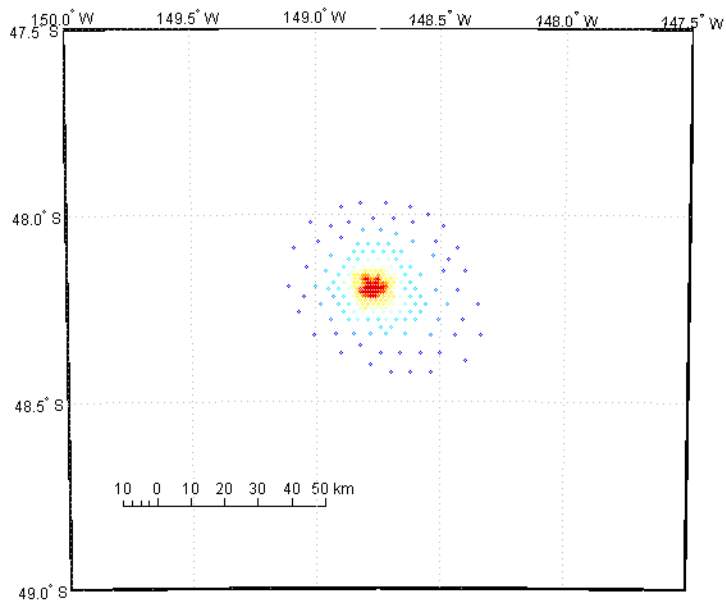


## See Also

`stem3m`

# scircle1

Small circles from center, range, and azimuth

## Syntax

```
[lat,lon] = scircle1(lat0,lon0,rad)
[lat,lon] = scircle1(lat0,lon0,rad,az)
[lat,lon] = scircle1(lat0,lon0,rad,az,ellipsoid)
[lat,lon] = scircle1(lat0,lon0,rad,units)
[lat,lon] = scircle1(lat0,lon0,rad,az,units)
[lat,lon] = scircle1(lat0,lon0,rad,az,ellipsoid,units)
[lat,lon] = scircle1(lat0,lon0,rad,az,ellipsoid,units,npts)
[lat,lon] = scircle1(track,...)
```

## Description

`[lat,lon] = scircle1(lat0,lon0,rad)` computes small circles (on a sphere) with a center at the point `lat0,lon0` and radius `rad`. The inputs can be scalar or column vectors. The input radius is in degrees of arc length on a sphere.

`[lat,lon] = scircle1(lat0,lon0,rad,az)` uses the input `az` to define the small circle arcs computed. The arc azimuths are measured clockwise from due north. If `az` is a column vector, then the arc length is computed from due north. If `az` is a two-column matrix, then the small circle arcs are computed starting at the azimuth in the first column and ending at the azimuth in the second column. If `az = []`, then a complete small circle is computed.

`[lat,lon] = scircle1(lat0,lon0,rad,az,ellipsoid)` computes small circles on the ellipsoid defined by the input `ellipsoid`, rather than by assuming a sphere. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. If the semimajor axis is non-zero, `rad` is assumed to be in distance units matching the units of the semimajor axis. However, if `ellipsoid = []`, or if the semimajor axis is zero, then `rad` is interpreted as an angle and the small circles are computed on a sphere as in the preceding syntax.

`[lat,lon] = scircle1(lat0,lon0,rad,units)`,
`[lat,lon] = scircle1(lat0,lon0,rad,az,units)`, and

`[lat,lon] = scircle1(lat0,lon0,rad,az,ellipsoid,`*`units`*`)` are all valid calling forms, which use the input string *units* to define the angle units of the inputs and outputs. If the *units* string is omitted, `'degrees'` is assumed.

`[lat,lon] = scircle1(lat0,lon0,rad,az,ellipsoid,`*`units`*`,npts)` uses the scalar input `npts` to determine the number of points per small circle computed. The default value of `npts` is 100.

`[lat,lon] = scircle1(`*`track`*`,...)` uses the *track* string to define either a great circle or rhumb line radius. If *track* = `'gc'`, then small circles are computed. If *track* = `'rh'`, then the circles with radii of constant rhumb line distance are computed. If the *track* string is omitted, `'gc'` is assumed.

`mat = scircle1(...)` returns a single output argument where `mat = [lat lon]`. This is useful if a single small circle is computed.

Multiple circles can be defined from a single starting point by providing scalar `lat0,lon0` inputs and column vectors for `rad` and `az` if desired.

## Definitions

A *small circle* is the locus of all points an equal surface distance from a given center. For true small circles, this distance is always calculated in a great circle sense; however, the `scircle1` function allows a locus to be calculated using distances in a rhumb line sense as well. An example of a small circle is *all points exactly 100 miles from the Washington Monument*. Parallels on the globe are all small circles. Great circles are a subset of small circles, specifically those with a radius of 90º or its angular equivalent, so all meridians on the globe are small circles as well.

*Small circle notation* consists of a center point and a radius in units of angular arc length.

## Examples

Create and plot a small circle centered at (0º,0º) with a radius of 10º.

```
axesm('mercator','MapLatLimit',[-30 30],'MapLonLimit',[-30 30]);
[latc,longc] = scircle1(0,0,10);
plotm(latc,longc,'g')
```

If the desired radius is known in some nonangular distance unit, use the radius returned by the `earthRadius` function as the ellipsoid input to set the range units. (Use an empty azimuth entry to indicate a full circle.)

```
[latc,longc] = scircle1(0,0,550,[],earthRadius('nm'));
plotm(latc,longc,'r')
```

For just an arc of the circle, enter an azimuth range.

```
[latc,longc] = scircle1(0,0,5,[-30 70]);
plotm(latc,longc,'m')
```
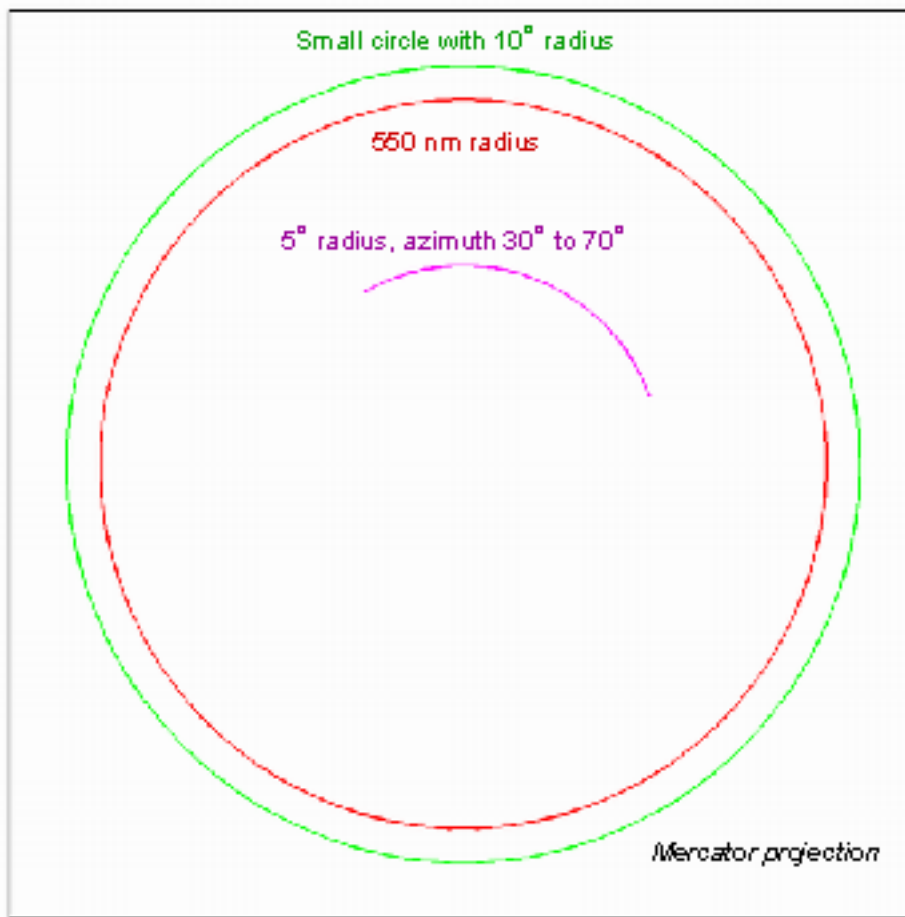
### See Also

`scircle2 | scircleg | track | trackg | track1 | track2`

# scircle2

Small circles from center and perimeter

## Syntax

```
[lat,lon] = scircle2(lat1,lon1,lat2,lon2)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,ellipsoid)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,units)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,ellipsoid,units)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,ellipsoid,units,npts)
[lat,lon] = scircle2(track,...)
mat = scircle2(...)
mat = [lat lon]
```

## Description

`[lat,lon] = scircle2(lat1,lon1,lat2,lon2)` computes small circles (on a sphere) with centers at the points `lat1,lon1` and points on the circles at `lat2,lon2`. The inputs can be scalar or column vectors.

`[lat,lon] = scircle2(lat1,lon1,lat2,lon2,ellipsoid)` computes the small circle on the ellipsoid defined by the input `ellipsoid`, rather than by assuming a sphere. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. Default is a unit sphere.

`[lat,lon] = scircle2(lat1,lon1,lat2,lon2,units)` and
`[lat,lon] = scircle2(lat1,lon1,lat2,lon2,ellipsoid,units)` are valid calling forms, which use the input string *units* to define the angle units of the inputs and outputs. If the input string *units* is omitted, `'degrees'` is assumed.

`[lat,lon] = scircle2(lat1,lon1,lat2,lon2,ellipsoid,units,npts)` uses the scalar input `npts` to determine the number of points per track computed. The default value of `npts` is 100.

`[lat,lon] = scircle2(track,...)` uses the *track* string to define either a great circle or a rhumb line radius. If *track*' = `'gc'`, then small circles are computed. If

*track* = 'rh', then circles with radii of constant rhumb line distance are computed. If the *track* string is omitted, 'gc' is assumed.

mat = scircle2(...) returns a single output argument where mat = [lat lon]. This is useful if a single circle is computed.

Multiple circles can be defined from a single center point by providing scalar lat1,lon1 inputs and column vectors for the points on the circumference, lat2,lon2.

## Definitions

A *small circle* is the locus of all points an equal surface distance from a given center. For true small circles, this distance is always calculated in a great circle sense. However, the scircle2 function allows a locus to be calculated using distances in a rhumb line sense as well. An example of a small circle is *all points exactly 100 miles from the Washington Monument*.

## Examples

Plot the locus of all points the same distance from New Delhi as Kathmandu:

```
axesm('mercator','MapLatlimit',[0 40],'MapLonLimit',[60 110]);
load coast

% For reference
plotm(lat,long,'k');

% New Delhi
lat1 = 29; lon1 = 77.5;

% Kathmandu
lat2 = 27.6; lon2 = 85.5;

% Plot the cities
plotm([lat1 lat2],[lon1 lon2],'b*')
[latc,lonc] = scircle2(lat1,lon1,lat2,lon2);
plotm(latc,lonc,'b')
```
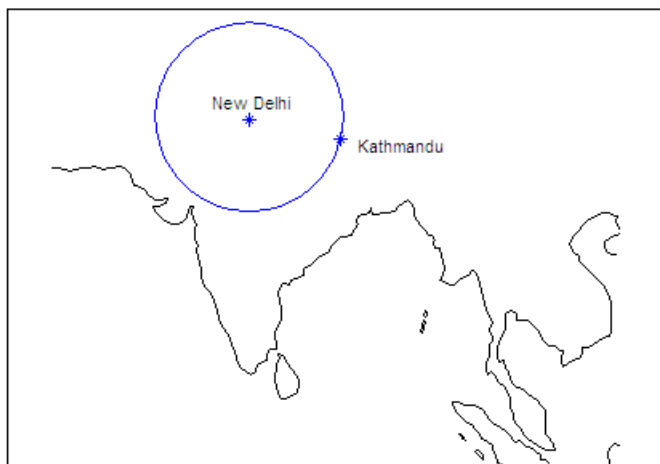
## See Also

scircle1 | track | track1 | track2

# scircleg

Small circle defined via mouse input

## Syntax

```
h = scircleg(ncirc)
h = scircleg(ncirc,npts)
h = scircleg(ncirc,linestyle)
h = scircleg(ncirc,PropertyName,PropertyValue,...)
[lat,lon] = scircleg(ncirc,npts,...)
h = scircleg(track,ncirc,...)
```

## Description

`h = scircleg(ncirc)` brings forward the current map axes and waits for the user to make (2 * `ncirc`) mouse clicks. The output `h` is a vector of handles for the `ncirc` small circles, which are then displayed.

`h = scircleg(ncirc,npts)` specifies the number of plotting points to be used for each small circle. `npts` is 100 by default.

`h = scircleg(ncirc,linestyle)` specifies the line style for the displayed small circles, where `linestyle` is any line style string recognized by the standard MATLAB function `line`.

`h = scircleg(ncirc,PropertyName,PropertyValue,...)` allows property name/property value pairs to be set, where `PropertyName` and `PropertyValue` are recognized by the `line` function.

`[lat,lon] = scircleg(ncirc,npts,...)` returns the coordinates of the plotted points rather than the handles of the small circles. Successive circles are stored in separate columns of `lat` and `lon`.

`h = scircleg(track,ncirc,...)` specifies the logic with which ranges are calculated. If the string track is `'gc'` (the default), great circle distance is used. If track is `'rh'`, rhumb line distance is used.

This function is used to define small circles for display using mouse clicks. For each circle, two clicks are required: one to mark the center of the circle and one to mark any point on the circle itself, thereby defining the radius.

## Background

A small circle is the locus of all points an equal surface distance from a given center. For true small circles, this distance is always calculated in a great circle sense; however, the scircleg function allows a locus to be calculated using distances in a rhumb line sense as well. You can modify the circle after creation by **shift**+clicking it. The circle is then in edit mode, during which you can change the size and position by dragging control points, or by entering values into a control panel. **Shift**+clicking again exits edit mode.

## See Also
scircle1 | scircle2

## scxsc

Intersection points for pairs of small circles

## Syntax

```
[newlat,newlon] = scxsc(lat1,lon1,range1,lat2,lon2,range2)
[newlat,newlon]=scxsc(lat1,lon1,range1,lat2,lon2,range2,units)
```

## Description

`[newlat,newlon] = scxsc(lat1,lon1,range1,lat2,lon2,range2)` returns in `newlat` and `newlon` the locations of the points of intersection of two small circles in *small circle notation*. For example, the first small circle in a pair would be centered on the point (`lat1`,`lon1`) with a radius of `range1` (in angle units). The inputs must be column vectors. If the circles do not intersect, or are identical, two `NaN`s are returned and a warning is displayed. If the two circles are tangent, the single intersection point is returned twice.

`[newlat,newlon]=scxsc(lat1,lon1,range1,lat2,lon2,range2,units)` specifies the angle units used for all inputs, where *units* is any valid angle units string. The default units are `'degrees'`.

For any pair of small circles, there are four possible intersection conditions: the circles are identical, they do not intersect, they are tangent to each other and hence they intersect once, or they intersect twice.

*Small circle notation* consists of a center point and a radius in units of angular arc length.

## Examples

Given a small circle centered at (10ºS,170ºW) with a radius of 20º (~1200 nautical miles), where does it intersect with a small circle centered at (3ºN, 179ºE), with a radius of 15º (~900 nautical miles)?

```
[newlat,newlon] = scxsc(-10,-170,20,3,179,15)
```

```
newlat =
    -8.8368    9.8526

newlon =
    169.7578 -167.5637
```

Note that in this example, the two small circles cross the date line.

# More About

### Tips

Great circles are a subset of small circles—a great circle is just a small circle with a radius of 90°. This provides two methods of notation for defining great circles. *Great circle notation* consists of a point on the circle and an azimuth at that point. *Small circle notation* for a great circle consists of a center point and a radius of 90° (or its equivalent in radians).

## See Also

gc2sc | gcxgc | gcxsc | rhxrh | crossfix | polyxpoly

# sdtsdemread

Read data from SDTS raster/DEM data set

## Syntax

```
[Z, R] = sdtsdemread(filename)
```

## Description

`[Z, R] = sdtsdemread(filename)` reads data from an SDTS DEM data set. `Z` is a matrix containing the elevation values. `R` is a referencing matrix (see `makerefmat`). `NaNs` are assigned to elements of `Z` corresponding to null data values or fill data values in the cell module.

`filename` can be the name of the SDTS catalog directory file (`*CATD.DDF`) or the name of any of the other files in the data set. `filename` can include the directory name; otherwise `filename` is searched for in the current directory and the MATLAB path. If any of the files specified in the catalog directory are missing, `sdtsdemread` fails.

## Examples

```
[Z, R] = sdtsdemread('9129CATD.ddf');
mapshow(Z,R,'DisplayType','contour')
```

## More About

### Tips

Elevation values can be imported with `sdtsdemread` from DEMs that use the SPRE Raster Profile (in use since January, 2001) as well as from older SDTS DEM data sets. Under this profile, elevations can be encoded either as 32-bit floating-point numbers (when their units are "decimal meters"), or as 16-bit integers (when units are "feet" or "meters"). The output class from sdtsdemread for both types of elevation encoding is `double`.

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

## See Also
arcgridread | mapshow | sdtsinfo | makerefmat

# sdtsinfo

Information about SDTS data set

## Syntax

```
info = sdtsinfo(filename)
```

## Description

`info = sdtsinfo(filename)` returns a structure whose fields contain information about the contents of a SDTS data set.

`filename` is a string that specifies the name of the SDTS catalog directory file, such as `7783CATD.DDF`. The filename can also include the directory name. If `filename` does not include the directory, then it must be in the current directory or in a directory on the MATLAB path. If `sdtsinfo` cannot find the SDTS catalog file, it returns an error.

If any of the other files in the data set as specified by the catalog file is missing, a warning message is returned. Subsequent calls to read data from the file might also fail.

## Field Descriptions

The `info` structure contains the following fields:

| | |
|---|---|
| Filename | String containing the name of the catalog directory file of the SDTS transfer set |
| Title | String containing the name of the data set |
| ProfileID | String containing the Profile Identifier, e.g., `'SRPE: SDTS RASTER PROFILE and EXTENSIONS'` |
| ProfileVersion | String containing the Profile Version Identifier, e.g., `'VER 1.1 1998 01'` |
| MapDate | String specifying the date associated with the cartographic information contained in the data set |

| DataCreationDate | String specifying the creation date of the data set |
|---|---|
| HorizontalDatum | String representing the horizontal datum to which the data is referenced |
| MapRefSystem | String describing the projection and reference system used: `'GEO'`, `'SPCS'`, `'UTM'`, `'UPS'`, or `' '` |
| ZoneNumber | Scalar value representing the zone number |
| XResolution | Scalar value representing the X component of the horizontal coordinate resolution |
| YResolution | Scalar value representing the Y component of the horizontal coordinate resolution |
| NumberOfRows | Scalar value representing the number of rows of the DEM |
| NumberOfCols | Scalar value representing the number of columns of the DEM |
| HorizontalUnits | String specifying the units used for the horizontal coordinate values |
| VerticalUnits | String specifying the units used for the vertical coordinate values |
| MinElevation | Scalar value of the minimum elevation value for the data set |
| MaxElevation | Scalar value of the maximum elevation value for the data set |

## Examples

```
info = sdtsinfo('9129CATD.DDF');
```

## See Also
```
sdtsdemread | makerefmat
```

# sectorg

Sector of small circle defined via mouse input

## Syntax

```
sectorg
```

## Description

`sectorg` prompts the user to indicate by two successive mouse clicks two points that define the center and radius of a small circle arc. By default, the angular width of the sector is 60º. The sector is constructed using the vector defined by the mouse clicks as the reference azimuth (defined to run through the center of the sector).

Once a sector has been drawn, **Shift**+clicking on it displays four control points (center point, arc resize, radial resize, and rotation controls), and the associated **Sector** control window. You can graphically interact with sectors as follows:

- To translate the circle, click and drag the center (o) control.
- To change the arc size, click and drag the resize control (square).
- To change the radial size of the sector, click and drag the radial control (down triangle).
- To rotate the arc, click and drag the rotation control (x).

You can also modify a selected sector by entering the appropriate values in the **Sector** control window and then pressing **Enter** or clicking the **Close** button. Display of the control panel is toggled by **Shift**+clicking the sector. If you select multiple sectors, a separate **Sector** control window will appear for each one.

## More About

### Tips

**Sector** control windows are superimposed at the same location. A valid map axes must exist prior to running this function.

## See Also
`scircleg | trackg`

# setltln

Convert data grid rows and columns to latitude-longitude

## Syntax

```
[lat, lon] = setltln(Z, R, row, col)
[lat, lon, indxPointOutsideGrid] = setltln(Z, R, row, col)
latlon = setltln(Z, R, row, col)
```

## Description

`[lat, lon] = setltln(Z, R, row, col)` returns the latitude and longitudes associated with the input row and column coordinates of the regular data grid `Z`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```
If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```
If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Points falling outside the grid are ignored in `row` and `col`. All input angles are in degrees.

`[lat, lon, indxPointOutsideGrid] = setltln(Z, R, row, col)` returns the indices of the elements of the `row` and `col` vectors that lie outside the input grid. The outputs `lat` and `lon` always ignore these points; the third output accounts for them.

`latlon = setltln(Z, R, row, col)` returns the coordinates in a single two-column matrix of the form `[latitude longitude]`.

## Examples

Find the coordinates of row 45, column 65 of `topo`:

```
load topo
[lat,lon,indxPointOutsideGrid] = setltln(topo,topolegend,45,65)

lat =
  -45.5000
lon =
   64.5000
indxPointOutsideGrid = [] % Empty because the point is valid
```

## See Also
`ltln2val` | `pix2latlon` | `setpostn`

# setm

Set properties of map axes and graphics objects

# Syntax

```
setm(h,MapAxesPropertyName,PropertyValue,...)
setm(texthndl,'MapPosition',position)
setm(surfhndl,'Graticule',lat,lon,alt)
setm(surfhndl,'MeshGrat',npts,alt)
```

# Description

setm(h,*MapAxesPropertyName*,PropertyValue,...), where h is a valid map axes handle, sets the map axes properties specified in the input list. The map axes properties must be recognized by axesm.

setm(texthndl,'MapPosition',position) alters the position of the projected text object specified by its handle to the [latitude longitude] or the [latitude longitude altitude] specified by the position vector.

setm(surfhndl,'Graticule',lat,lon,alt) alters the graticule of the projected surface object specified by its handle. The graticule is specified by the latitude and longitude matrices, specifying locations of the graticule vertices. The altitude can be specified by a scalar, or by a matrix providing a value for each vertex.

setm(surfhndl,'MeshGrat',npts,alt) alters the mesh graticule of projected surface objects displayed using the meshm function. In this case, the two-element vector npts specifies the graticule size in the manner described under meshm. The altitude can be a scalar or a matrix with a size corresponding to npts.

# Examples

Display a map axes and alter it:

```
axesm('bonne','Frame','on','Grid','on')
```

The standard Bonne projection has a standard parallel at 30ºN.
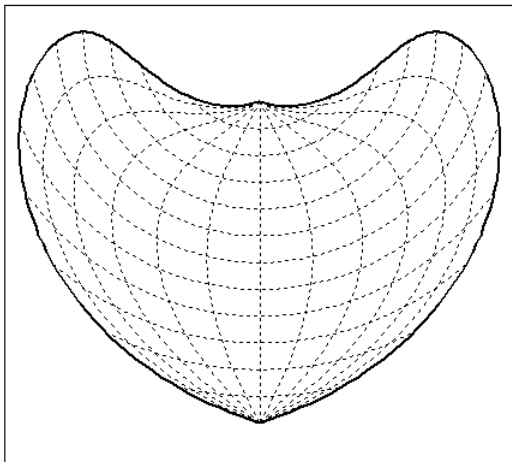


Setting this standard parallel to 0º results in a Sinusoidal projection:

```
setm(gca,'MapParallels',0)
```
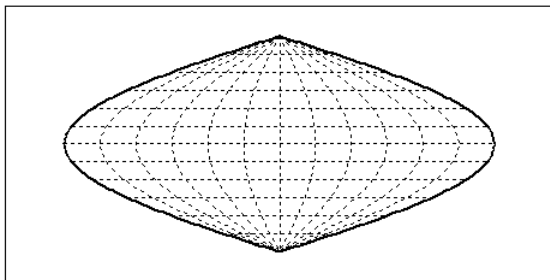


## See Also

axesm | getm

# setpostn

Convert latitude-longitude to data grid rows and columns

## Syntax

```
[row, col] = setpostn(Z, R, lat, lon)
indx = setpostn(...)
[row, col, indxPointOutsideGrid] = setpostn(...)
```

## Description

`[row, col] = setpostn(Z, R, lat, lon)` returns the row and column indices of the regular data grid `Z` for the points specified by the vectors `lat` and `lon`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```
If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```
If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Points falling outside the grid are ignored in `row` and `col`. All input angles are in degrees.

`indx = setpostn(...)` returns the indices of `Z` corresponding to the points in `lat` and `lon`. Points falling outside the grid are ignored in `indx`.

`[row, col, indxPointOutsideGrid] = setpostn(...)` returns the indices of `lat` and `lon` corresponding to points outside the grid. These points are ignored in `row` and `col`.

## Examples

What are the matrix coordinates in topo of Denver, Colorado, at (39.7ºN,105ºW)?

```
load topo
[row,col] = setpostn(topo,topolegend,39.7,105)

row =
   130
col =
   105
```

## See Also
latlon2pix | ltln2val | setltln

# shaderel

Construct cdata and colormap for shaded relief

## Syntax

```
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap)
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev])
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmapl)
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmapl,clim)
```

## Description

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap)` constructs the colormap and color indices to allow a surface to be displayed in colored shaded relief. The colors are proportional to the magnitude of `Z`, but modified by shades of gray based on the surface normals to simulate surface lighting. This representation allows both large and small-scale differences to be seen. `X`, `Y`, and `Z` define the surface. `cmap` is the colormap used to create the new shaded colormap `cimap`. `cindx` is a matrix of color indices to `cimap`, based on the elevation and surface normal of the `Z` matrix element. `clim` contains the color axis limits.

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev])` places the light at the specified azimuth and elevation. By default, the direction of the light is East (90° azimuth) at an elevation of 45°.

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmapl)` chooses the number of grays to give a `cimap` of length `cmapl`. By default, the number of grayscales is chosen to keep the shaded colormap below 256. If the vector of azimuth and elevation is empty, the default locations are used.

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmapl,clim)` uses the color limits to index `Z` into `cmap`.

## Examples

Display the `peaks` surface with a shaded colormap:

```
[X,Y,Z] = peaks(100);
cmap = hot(16);
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap);
surf(X,Y,Z,cindx); colormap(cimap); caxis(clim)
shading flat
```
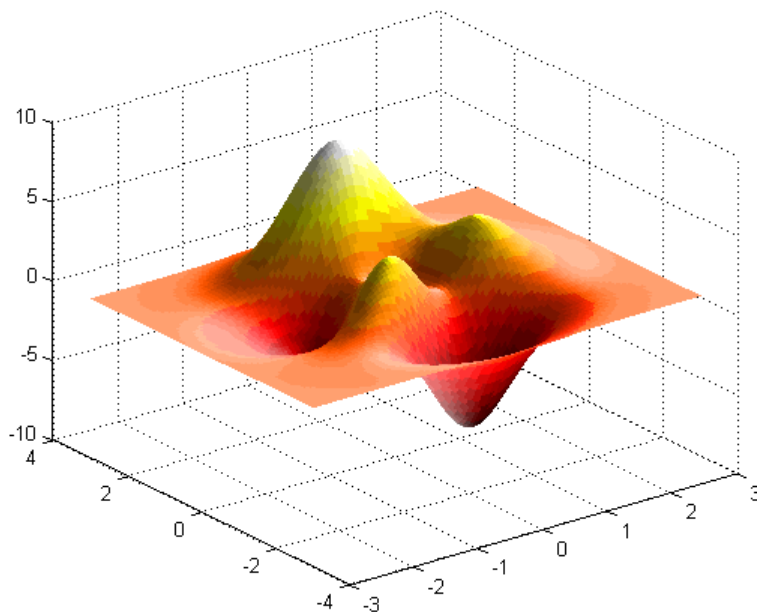


## More About

### Tips

This function effectively multiplies two colormaps, one with color based on elevation, the other with a grayscale based on the slope of the surface, to create a new colormap. This produces an effect similar to using a light on a surface, but with all of the visible colors actually in the colormap. Lighting calculations are performed on the unprojected data.

## See Also

caxis | colormap | light | meshlsrm | surf | surflsrm

# shapeinfo

Information about shapefile

## Syntax

```
info = shapeinfo(filename)
```

## Description

`info = shapeinfo(filename)` returns a structure, `info`, whose fields contain information about the contents of a shapefile. `filename` can be the base name or the full name of any one of the component files. `shapeinfo` reads all three files as long as they exist in the same folder and the unit of length or angle is not specified. If the main file (with extension `.SHP`) is missing, `shapeinfo` returns an error. If either of the other files is missing, `shapeinfo` returns a warning.

## Output Arguments

The `info` structure contains the following fields:

| | |
|---|---|
| Filename | Char array containing the names of the files that were read |
| ShapeType | String containing the shape type |
| BoundingBox | Numerical array of size 2-by-N that specifies the minimum (row 1) and maximum (row 2) values for each dimension of the spatial data in the shapefile |
| Attributes | Structure array of size 1-by-`numAttributes` that describes the attributes of the data |
| NumFeatures | The number of spatial features in the shapefile |

The `Attributes` structure contains these fields:

| | |
|---|---|
| Name | String containing the attribute name as given in the xBASE table |

| Type | String specifying the MATLAB class of the attribute data returned by `shaperead`. The following attribute (xBASE) types are supported: Numeric, Floating, Character, and Date. |
|------|------|

# More About

**Tips**

`shapeinfo` cannot tell you which coordinate system the data in a shapefile uses. Coordinates can be either planar (*x*, *y*) or geographic (*lat*, *lon*) and have a variety of units. This information can be critical to the proper display of shapefile vector data. For more information on this topic, see "Mapstructs and Geostructs".

· "Mapping Toolbox Geographic Data Structures"

# See Also
`shaperead` | `shapewrite`

# shaperead

Read vector features and attributes from shapefile

## Syntax

```
S = shaperead(filename)
S = shaperead(filename,Name,Value, ...)
[S, A] = shaperead(...)
```

## Description

`S = shaperead(filename)` reads the shapefile, `filename`, and returns an *N*-by-1 geographic data structure array in projected map coordinates (a mapstruct). The geographic data structure combines geometric and feature attribute information. `shaperead` supports the ordinary 2-D shape types: `'Point'`, `'Multipoint'`, `'PolyLine'`, and `'Polygon'`.

`S = shaperead(filename,Name,Value, ...)` returns a subset of the shapefile contents in `S`, as determined by the parameters. The geographic data structure, `S`, is a mapstruct unless `UseGeoCoords` is `true`. If you do not specify any parameters, `shaperead` returns an entry for every non-null feature and creates a field for every attribute.

`[S, A] = shaperead(...)` returns an *N*-by-1 geographic data structure array, `S`, containing geometric information, and a parallel *N*-by-1 attribute structure array, `A`, containing feature attribute information.

## Input Arguments

**`filename`**

Refers to the base name or full name of one of the component files in a shapefile. If the main file (with extension `.shp`) is missing, `shaperead` throws an error. If either of the other files is missing, `shaperead` issues a warning.

Make sure that your machine is set to the same character encoding scheme as the data you are importing. For example, if you are trying to import a shapefile that contains Japanese characters, configure your machine to support the `Shift-JIS` encoding scheme.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**`'RecordNumbers'`**

Integer-valued vector of class `double`. Use the parameter `RecordNumbers` to import only features with listed record numbers.

Use the parameters `RecordNumbers`, `BoundingBox`, and `Selector` to select which features to read. If you use more than one of these parameters in the same call, you receive the intersection of the records that match the individual specifications. For instance, if you specify values for both `RecordNumbers` and `BoundingBox`, you import only those features with record numbers that appear in your list and that also have bounding boxes intersecting the specified bounding box.

**`'BoundingBox'`**

2-by-2 array of class `double`. Use the parameter `BoundingBox` to import only features whose bounding boxes intersect the specified box. The `shaperead` function does not trim features that partially intersect the box.

**`'Selector'`**

Cell array containing a function handle and one or more attribute names. (The function must return a `logical` scalar.) Use the `Selector` parameter to import only features for which the function, when applied to the corresponding attribute values, returns `true`.

**`'Attributes'`**

Cell array of attribute names. Use the parameter `Attributes` to include listed attributes and set the order of attributes in the structure array. Use `{}` to omit all attributes.

**'UseGeoCoords'**

`Logical` scalar that specifies returning shapefile contents in a geostruct, if set to `true`. Use this parameter when you know that the *x-* and *y-* coordinates in the shapefile actually represent longitude and latitude data. (If you do not know whether you are working with geographic or map coordinates, see "Mapstructs and Geostructs" in the User's Guide.)

**Default:** `false`

## Output Arguments

**S**

An *N*-by-1 geographic data structure array containing an element for each non-null, spatial feature in the shapefile.

**A**

An *N*-by-1 attribute structure array, `A`, parallel to array `S`.

The fields in the output structure arrays `S` and `A` depend on (1) the type of shape contained in the file and (2) the names and types of attributes included in the file. The `shaperead` function supports the following four attribute types: `numeric` and `floating` (stored as type `double` in MATLAB) and `character` and `date` (stored as `char array`).

## Examples

### Read entire contents of shapefile

Read the entire `concord_roads.shp` shapefile, including the attributes in `concord_roads.dbf`. The `shaperead` function returns a mapstruct with X and Y coordinate vectors.

```
S = shaperead('concord_roads.shp')
```

```
S =
```

```
609x1 struct array with fields:

    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

### Read portion of shapefile based on bounding box

Specify a bounding box to limit the data returned by shaperead . In addition, specify the names of the attributes you want to read.

```
bbox = [2.08 9.11; 2.09 9.12] * 1e5;
S = shaperead('concord_roads','BoundingBox',bbox,...
              'Attributes',{'STREETNAME','CLASS'})


S =

87x1 struct array with fields:

    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    CLASS
```

### Read road data by class from shapefile

Read road data only for class 4 and higher road segments that are at least 200 meters in length. Note the use of an anonymous function in the selector.

```
S = shaperead('concord_roads.shp','Selector',...
          {@(v1,v2) (v1 >= 4) && (v2 >= 200),'CLASS','LENGTH'})
```

```
S =

115x1 struct array with fields:

    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Determine the number of roads of each class.

```
n = histcounts([S.CLASS],'BinLimits',[1 7],'BinMethod','integer')
```

```
n =

     0     0     0     7    93    15     0
```

Display a histogram of the number of roads that fall in each category of length.

```
figure
histogram([S.LENGTH])
```
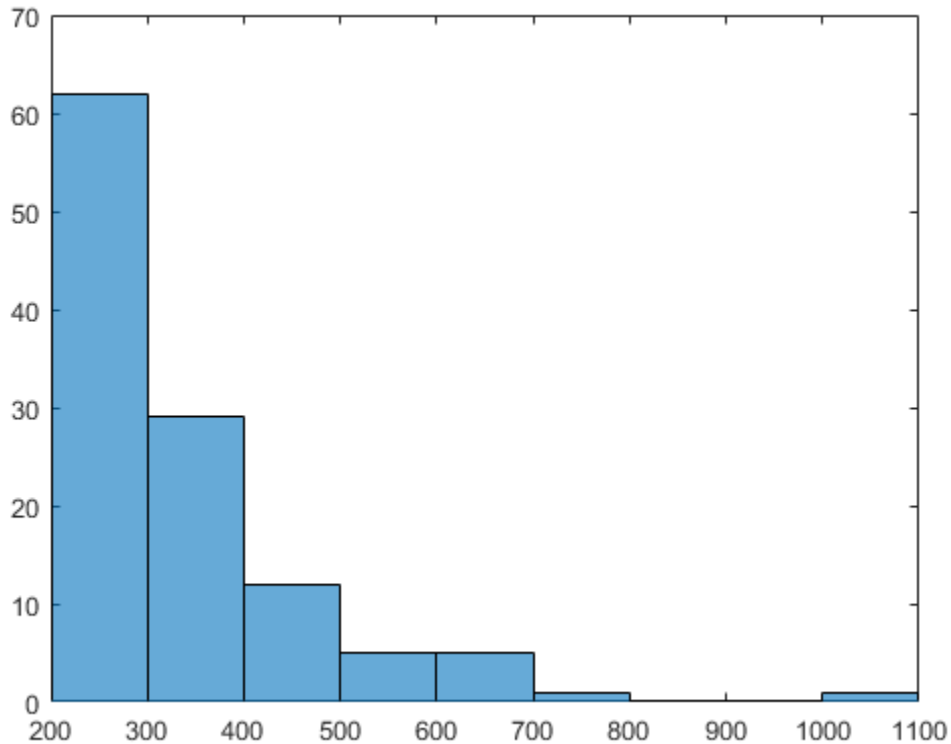
### Read shapefile and return geostruct

Read a shapefile of worldwide city names and locations given in latitude and longitude. Note how shaperead returns a geostruct because you specified the 'UseGeoCoords' parameter.

```
S = shaperead('worldcities.shp', 'UseGeoCoords', true)


S =

318x1 struct array with fields:

    Geometry
```

```
Lon
Lat
Name
```

# More About

- "Mapping Toolbox Geographic Data Structures"
- "Finding Geospatial Data"

# See Also

shapeinfo | shapewrite

# shapewrite

Write geographic vector data structure to shapefile

## Syntax

```
shapewrite(S,filename)
shapewrite(S,filename,'DbfSpec',dbfspec)
```

## Description

`shapewrite(S,filename)` writes the vector geographic features stored in `S` to the file specified by `filename` in shapefile format.

`shapewrite(S,filename,'DbfSpec',dbfspec)` writes a shapefile in which the content and layout of the DBF file is controlled by a DBF specification, indicated here by the parameter value `dbfspec`.

## Input Arguments

### S

Vector geographic features, specified as a mappoint vector, mapshape vector, mapstruct (with `X` and `Y` coordinate fields), geopoint vector, geoshape vector, or a geostruct (with `'Lat'` and `'Lon'` fields) with the following restrictions on its attribute fields:

- Each attribute field value must be either a real, finite, scalar `double` or a character string.
- The type of a given attribute must be consistent across all features.
- If `S` is a geopoint vector, geoshape vector, or a geostruct, `shapewrite` writes the latitude and longitude values as *Y* and *X* coordinates, respectively.
- If a given attribute is integer-valued for all features, `shapewrite` writes it to the `[basename '.dbf']` file as an integer. If an attribute is non-integer-valued for any feature,`shapewrite` writes it as a fixed point decimal value with six digits to the right of the decimal place.

**filename**

Name and location of the shapefile to create, specified as a character string. If the file name includes a file extension, it must be `'.shp'` or `'.SHP'`. `shapewrite` creates three output files: `[basename '.shp']`, `[basename '.shx']`, and `[basename '.dbf']`, where `basename` is `filename` without its extension.

**`'DbfSpec'`**

Feature attributes to be included in the shapefile, specified as a scalar MATLAB structure containing one field for each feature attribute. Assign to that field a scalar structure with the following four fields:

- `FieldName` — The field name to be used in the file
- `FieldType` — The field type to be used in the file: `'N'` (numeric) or `'C'` (character)
- `FieldLength` — The field length in the file, in bytes
- `FieldDecimalCount` — For numeric fields, the number of digits to the right of the decimal place

Call `makedbfspec` to construct a DBF spec. Modify the output to remove attributes or change the `FieldName`, `FieldLength`, or `FieldDecimalCount` for one or more attributes.

To include an attribute in the output file, specify a field in `dbfspec` with the same name as the attribute is specified in `S`.

## Examples

### Write Feature Data to Shapefile

Derive a shapefile from `concord_roads.shp` in which roads of `CLASS 5` and greater are omitted.

Get information about the contents of a shapefile. Note that it contains 609 features (`NumFeatures`).

```
shapeinfo('concord_roads')
ans =
```

```
        Filename: [3x67 char]
       ShapeType: 'PolyLine'
     BoundingBox: [2x2 double]
     NumFeatures: 609
      Attributes: [5x1 struct]
```

Read a selection of the data in the file into a mapstruct. Note the use of the `'Selector'` option in `shaperead`, together with an anonymous function, to read only the main roads from the original shapefile.

```
S = shaperead('concord_roads', 'Selector', ...
              {@(roadclass) roadclass < 4, 'CLASS'});
```

Write the data to a new shapefile.

```
shapewrite(S, 'main_concord_roads.shp')
```

Get information about the contents of the new shapefile.

```
shapeinfo('main_concord_roads')  % 107 features

ans =
        Filename: [3x24 char]
       ShapeType: 'PolyLine'
     BoundingBox: [2x2 double]
     NumFeatures: 107
      Attributes: [5x1 struct]
```

### Write Data Stored in mappoint to Shapefile

Read a shapefile containing a vector of world cities and store the data in a mappoint vector.

```
p = mappoint(shaperead('worldcities.shp'))

p =

 318x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1x318 double]
```

```
           Y: [1x318 double]
        Name: {1x318 cell}
```

Append Paderborn Germany to the mappoint vector. Note that the size of `p` has increased by 1.

```
x = 51.715254;
y = 8.75213;
p = append(p, x, y, 'Name', 'Paderborn')

p =

 319x1 mappoint vector with properties:

 Collection properties:
    Geometry: 'point'
    Metadata: [1x1 struct]
 Feature properties:
           X: [1x319 double]
           Y: [1x319 double]
        Name: {1x319 cell}
```

Write the updated mappoint vector to a shapefile.

```
shapewrite(p,'worldcities_updated')
```

## More About

### Tips

- The easiest way to create a DBF spec is to call `makedbfspec` and then modify the output to remove attributes or change the FieldName, FieldLength, or FieldDecimalCount for one or more attributes. See the help for `makedbfspec` for more details and an example.

- The xBASE (`.dbf`) file specifications require that geostruct and mapstruct attribute names are truncated to 11 characters when copied as DBF field names. Consider shortening long field names before calling `shapewrite`. By doing this, you make field names in the DBF file more readable and avoid introducing duplicate names as a result of truncation.

- Remember to set your character encoding scheme to match that of the geographic data structure you are exporting. For instance, if you are exporting a map that

displays Japanese text, configure your machine to support `Shift-JIS` character encoding.

- "Mapping Toolbox Geographic Data Structures"

## See Also

`makedbfspec` | `shapeinfo` | `shaperead`

# showaxes

Toggle display of map coordinate axes

## Syntax

```
showaxes(action)
showaxes
```

## Description

showaxes(*action*) modifies the Cartesian axes based on the value of *action*, as defined in the Inputs section below.

showaxes toggles between displaying the default axes ticks on the MATLAB Cartesian axes and removing the axes ticks.

## Input Arguments

**action**

A string or RGB triple that specifies how to modify the Cartesian axes

**Default:**

| Value | Data Type | Action |
|---|---|---|
| 'on' | string | Displays the MATLAB Cartesian axes and default axes ticks |
| 'off' | string | Removes the axes ticks from the MATLAB Cartesian axes |
| 'hide' | string | Hides the Cartesian axes |
| 'show' | string | Shows the Cartesian axes |
| 'reset' | string | Sets the Cartesian axes to the default settings |
| 'boxoff' | string | Removes axes ticks, color, and box from the Cartesian axes |

| Value | Data Type | Action |
|---|---|---|
| *colorstr* | string | Sets the Cartesian axes to the color specified by *colorstr* |
| *colorvec* | RGB triple | Uses *colorvec* to set the Cartesian axes color |

## See Also

axesm

# showm

Specify graphic objects to display on map axes

## Syntax

```
showm
showm(handle)
showm(object)
```

## Description

showm brings up a dialog box for selecting the objects to show (set their `Visible` property to `'on'`).

showm(handle) shows the objects specified by a vector of handles.

showm(*object*) shows those objects specified by the *object* string, which can be any string recognized by the handlem function.

## See Also
clma | clmo | handlem | hidem | namem | tagm

# sizem

Row and column dimensions needed for regular data grid

## Syntax

```
[r,c] = sizem(latlim,lonlim,scale)
rc = sizem(latlim,lonlim,scale)
[r,c,refvec] = sizem(latlim,lonlim,scale)
```

## Description

`[r,c] = sizem(latlim,lonlim,scale)` returns the required size for a regular data grid lying between the latitude and longitude limits specified by the two-element input vectors `latlim` and `lonlim`, which are of the form `[south-limit north-limit]` and `[west-limit and east-limit]`, respectively. The `scale` is the desired cells-per-degree measure of the desired data grid.

`rc = sizem(latlim,lonlim,scale)` returns the size of the matrix in one two-element vector.

`[r,c,refvec] = sizem(latlim,lonlim,scale)` also returns the three-element referencing vector geolocating the desired regular data grid.

## Examples

How large a matrix would be required for a map of the world at a scale of 25 matrix cells per degree? (That's 25x25=625 cells per "square" degree.)

```
[r,c] = sizem([90,-90],[-180,180],25)

r =
        4500
c =
        9000
```

Bear in mind for memory purposes — 9000 x 4500 = 4.05 x $10^7$ entries!

## See Also
findm | limitm | nanm | onem | spzerom | zerom

# sm2deg

Convert distance from statute miles to degrees

## Syntax

```
deg = sm2deg(sm)
deg = sm2deg(sm,radius)
deg = sm2deg(sm,sphere)
```

## Description

deg = sm2deg(sm) converts distances from statute miles to degrees as measured along a great circle on a sphere with a radius of 6371 km (3958.748 sm), the mean radius of the Earth.

deg = sm2deg(sm,radius) converts distances from statute miles to degrees as measured along a great circle on a sphere having the specified radius. radius must be in units of statute miles.

deg = sm2deg(sm,sphere) converts distances from statute miles to degrees, as measured along a great circle on a sphere approximating an object in the Solar System. sphere may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive.

## See Also

deg2rad | deg2km | km2rad | km2nm | km2sm | deg2nm | nm2deg | nm2km | nm2sm | deg2sm | sm2km | sm2nm | rad2deg

# sm2km

Convert statute miles to kilometers

## Syntax

```
km = sm2km(sm)
```

## Description

`km = sm2km(sm)` converts distances from statute miles to kilometers.

### See Also
deg2km | km2deg | km2rad | rad2km | deg2nm | nm2deg | nm2rad | rad2nm |
deg2sm | sm2deg | deg2sm | sm2rad | rad2sm

# sm2nm

Convert statute to nautical miles

## Syntax

nm = sm2nm(sm)

## Description

nm = sm2nm(sm) converts distances from statute to nautical miles.

### See Also

deg2km | km2deg | km2rad | rad2km | deg2nm | nm2deg | nm2rad | rad2nm |
deg2sm | sm2deg | deg2sm | sm2rad | rad2sm

# sm2rad

Convert distance from statute miles to radians

## Syntax

```
rad = sm2rad(sm)
rad = sm2rad(sm,radius)
rad = sm2rad(sm,sphere)
```

## Description

`rad = sm2rad(sm)` converts distances from statute miles to radians as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`rad = sm2rad(sm,radius)` converts distances from statue miles to radians as measured along a great circle on a sphere having the specified radius. `radius` must be in units of statute miles.

`rad = sm2rad(sm,sphere)` converts distances from statute miles to radians, as measured along a great circle on a sphere approximating an object in the Solar System. *sphere* may be one of the following strings: `'sun'`, `'moon'`, `'mercury'`, `'venus'`, `'earth'`, `'mars'`, `'jupiter'`, `'saturn'`, `'uranus'`, `'neptune'`, or `'pluto'`, and is case-insensitive.

## See Also

deg2rad | rad2km | km2deg | km2nm | km2sm | rad2nm | nm2deg | nm2km | nm2sm | rad2sm | sm2deg | sm2km | sm2nm | rad2deg

# smoothlong

Remove discontinuities in longitude data

---

**Note:** The `smoothlong` function is obsolete and has been replaced by `unwrapMultipart`, which requires input to be in radians. When working in degrees, use `rad2deg(unwrapMultipart(deg2rad(lon)))`.

---

## Syntax

```
ang = smoothlong(angin)
ang = smoothlong(angin,angleunits)
```

## Description

`ang = smoothlong(angin)` removes discontinuities in longitude data. The resulting angles can cover more than one revolution.

`ang = smoothlong(angin,angleunits)` uses the units defined by the input string *angleunits*. If omitted, default units of `'degrees'` are assumed. Valid *angleunits* are:

- `'degrees'` — decimal degrees
- `'radians'`

## See Also

unwrapMultipart

# spcread

Read columns of data from ASCII text file

## Syntax

```
mat = spcread
mat = spcread(filename)
mat = spcread(cols)
```

## Description

`mat = spcread` reads an ASCII file of space-delimited data in two columns and returns the data in a matrix, `mat`. The file is selected by dialog box.

`mat = spcread(filename)` specifies the file from which to read by its name, given as the string *filename*.

`mat = spcread(cols)` specifies the number of columns of space-delimited data in the file with the integer `cols`. The default value of `cols` is 2.

## More About

### Tips

The `spcread` function is similar to the standard MATLAB function `dlmread`. `spcread`, however, is much faster at reading large data sets of the type common for geographic purposes.

## See Also
`nanclip`

# spzerom

Construct sparse regular data grid of 0s

## Compatibility

spzerom is not recommended and may be removed in a future release. Instead, use georefcells to construct a geographic raster reference object and then use sparse to initialize a data grid of the appropriate size:

```
R = georefcells(latlim,lonlim,1/scale,1/scale);
Z = sparse(R.RasterSize(1),R.RasterSize(2));
```

## Syntax

```
[Z,refvec] = spzerom(latlim,lonlim,scale)
```

## Description

[Z,refvec] = spzerom(latlim,lonlim,scale) returns a sparse regular data grid consisting entirely of 0s and a three-element referencing vector for the returned Z. The two-element vectors latlim and lonlim define the latitude and longitude limits of the geographic region. They should be of the form [south north] and [west east], respectively. The scalar scale specifies the number of rows and columns per degree of latitude and longitude.

## Examples

```
[Z,refvec] = spzerom([46,51],[-79,-75],1)

Z =
   All zero sparse: 5-by-4
refvec =
     1    51    -79
```

## See Also

```
limitm | nanm | onem | sizem | zerom
```

# stdist

Standard distance for geographic points

## Syntax

```
dist = stdist(lat,lon)
dist = stdist(lat,lon,units)
dist = stdist(lat,lon,ellipsoid)
dist = stdist(lat,lon,ellipsoid,units,method)
```

## Description

`dist = stdist(lat,lon)` computes the average standard distance for geographic data. This function assumes that the data is distributed on a sphere. In contrast, `std` assumes that the data is distributed on a Cartesian plane. The result is a single value based on the great-circle distance of the data points from their geographic mean point. When `lat` and `lon` are vectors, a single distance is returned. When `lat` and `lon` are matrices, a row vector of distances is given, providing the distances for each column of `lat` and `lon`. N-dimensional arrays are not allowed. Distances are returned in degrees of angle units.

`dist = stdist(lat,lon,units)` indicates the angular units of the data. When the standard angle string *units* is omitted, `'degrees'` is assumed. Output measurements are in terms of these *units* (as arc length distance).

`dist = stdist(lat,lon,ellipsoid)` specifies the shape of the Earth to be used with `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form [`semimajor_axis eccentricity`]. The default is a unit sphere. Output measurements are in terms of the distance units of the semimajor axis of the `ellipsoid`.

`dist = stdist(lat,lon,ellipsoid,units,method)` specifies the method of calculating the standard distance of the data. The default, `'linear'`, is simply the average great circle distance of the data points from the centroid. Using `'quadratic'` results in the square root of the average of the squared distances, and `'cubic'` results in the cube root of the average of the cubed distances.

## Background

The function `stdm` provides independent standard deviations in latitude and longitude of data points. `stdist` provides a means of examining data scatter that does not separate these components. The result is a *standard distance*, which can be interpreted as a measure of the scatter in the great circle distance of the data points from the centroid as returned by `meanm`.

The output distance can be thought of as the radius of a circle centered on the geographic mean position, which gives a measure of the spread of the data.

## Examples

Create latitude and longitude lists using the `worldcities` data set and obtain standard distance deviation for group (compare with the example for `stdm`):

```
cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
Paris = strcmp('Paris',{cities(:).Name});
London = strcmp('London',{cities(:).Name});
Rome = strcmp('Rome',{cities(:).Name});
Madrid = strcmp('Madrid',{cities(:).Name});
Berlin = strcmp('Berlin',{cities(:).Name});
Athens = strcmp('Athens',{cities(:).Name});
lat = [cities(Paris).Lat cities(London).Lat...
       cities(Rome).Lat cities(Madrid).Lat...
       cities(Berlin).Lat cities(Athens).Lat]
lon = [cities(Paris).Lon cities(London).Lon...
       cities(Rome).Lon cities(Madrid).Lon...
       cities(Berlin).Lon cities(Athens).Lon]

dist = stdist(lat,lon)

lat =
   48.8708   51.5188   41.9260   40.4312   52.4257   38.0164
lon =
    2.4131   -0.1300   12.4951   -3.6788   13.0802   23.5183
dist =
    8.1827
```

## See Also

`meanm` | `stdm`

# stdm

Standard deviation for geographic points

## Syntax

```
[latdev,londev] = stdm(lat,lon)
[latdev,londev] = stdm(lat,lon,ellipsoid)
[latdev,londev] = stdm(lat,lon,units)
```

## Description

`[latdev,londev] = stdm(lat,lon)` returns row vectors of the latitude and longitude geographic standard deviations for the data points specified by the columns of `lat` and `lon`.

`[latdev,londev] = stdm(lat,lon,ellipsoid)` specifies the shape of the Earth to be used by the `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. The default ellipsoid is a unit sphere. Output measurements are in terms of the distance units of the `ellipsoid` vector.

`[latdev,londev] = stdm(lat,lon,units)` indicates the angular units of the data. When the standard angle string *units* is omitted, `'degrees'` is assumed. Output measurements are in terms of these *units* (as arc length distance).

If a single output argument is used, then `geodevs = [latdev longdev]`. This is particularly useful if the original `lat` and `lon` inputs are column vectors.

## Background

Determining the deviations of geographic data in latitude and longitude is more complicated than simple sum-of-squares deviations from the data averages. For latitude deviation, a straightforward angular standard deviation calculation is performed from the *geographic mean* as calculated by `meanm`. For longitudes, a similar calculation is

performed based on data *departure* rather than on angular deviation. See "Geographic Statistics" in the *Mapping Toolbox User's Guide*.

## Examples

Create latitude and longitude lists using the `worldcities` data set and obtain standard distance deviation for group (compare with the example for `stdist`):

```
cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
Paris = strcmp('Paris',{cities(:).Name});
London = strcmp('London',{cities(:).Name});
Rome = strcmp('Rome',{cities(:).Name});
Madrid = strcmp('Madrid',{cities(:).Name});
Berlin = strcmp('Berlin',{cities(:).Name});
Athens = strcmp('Athens',{cities(:).Name});
lat = [cities(Paris).Lat cities(London).Lat...
       cities(Rome).Lat cities(Madrid).Lat...
       cities(Berlin).Lat cities(Athens).Lat]
lon = [cities(Paris).Lon cities(London).Lon...
       cities(Rome).Lon cities(Madrid).Lon...
       cities(Berlin).Lon cities(Athens).Lon]
[latstd,lonstd]=stdm(lat,lon)

lat =
   48.8708    51.5188    41.9260    40.4312    52.4257    38.0164
lon =
    2.4131    -0.1300    12.4951    -3.6788    13.0802    23.5183
latstd =
    2.7640
lonstd =
   68.7772
```

## See Also

`departure | filterm | hista | histr | meanm | stdist`

# stem3m

Project stem plot map on map axes

## Syntax

```
h = stem3m(lat,lon,z)
h = stem3m(lat,lon,z,LineType)
h = stem3m(lat,lon,z,PropertyName,PropertyValue,...)
```

## Description

`h = stem3m(lat,lon,z)` displays a stem plot on the current map axes. Stems are located at the points (`lat`,`lon`) and extend from an altitude of 0 to the values of `z`. The coordinate inputs should be in the same `AngleUnits` as the map axes. It is important to note that the selection of *z*-values will greatly affect the 3-D look of the plot. Regardless of `AngleUnits`, the *x* and *y* limits of the map axes are at most -π to +π and -π/2 to +π/2, respectively. This means that for most purposes, appropriate `z` values would be on the order of 1 to 3, not 10 to 30. The axes `DataAspectRatio` property can be used to adjust the appearance of the graphic. The handles of the displayed stem lines can be returned in `h`.

`h = stem3m(lat,lon,z,LineType)` allows the style of the stem plot's lines to be specified with any string *LineType* recognized by the MATLAB `line` function.

`h = stem3m(lat,lon,z,PropertyName,PropertyValue,...)` allows any property/value pair recognized by the MATLAB `line` function to be specified for the stems.

A stem plot displays data as lines extending normal to the *xy*-plane, in this case, on a map.

## Examples

```
load coast
axesm sinusoid; view(3)
h = framem; set(h,'zdata',zeros(size(lat)))
```

```
plotm(lat,long)
ptlat = [0 30 30 -50 -78]';
ptlon = [0 30 -70 65 -35]';
ptz = [1 1.5 2 .5 1]';
stem3m(ptlat,ptlon,ptz,'r-')
```
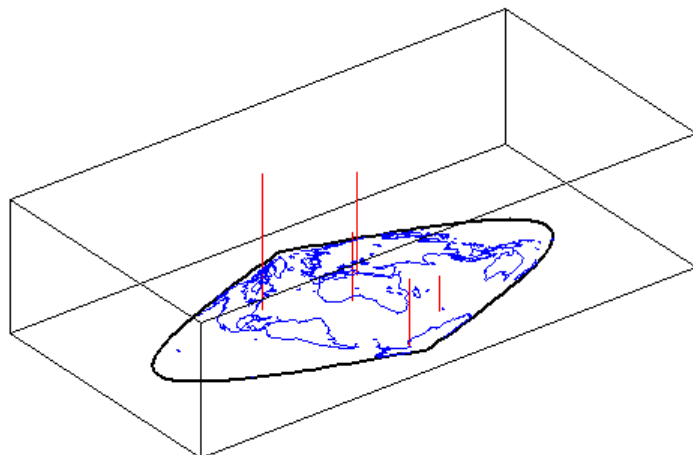


## See Also

```
scatterm
```

# str2angle

Convert strings to angles in degrees

## Syntax

```
angles = str2angle(strings)
```

## Description

`angles = str2angle(strings)` converts strings containing latitudes and/or longitudes, expressed in one of four different formats of degree-minutes-seconds, to numeric angles in units of degrees.

| Format Description | Example |
| --- | --- |
| Degree Symbol, Single/Double Quotes | `'123°30''00"W'` |
| 'd', 'm', 's' Separators | `'123d30m00sW'` |
| Minus Signs as Separators | `'123-30-00W'` |
| "Packed DMS" | `'1233000W'` |

Input must conform closely to the examples provided; in particular, the seconds field must be included, even if it is not significant. Except in Packed DMS format, the seconds field can contain a fractional component. Sign characters are not supported; terminate each string with `'N'` for positive latitude, `'S'` for negative latitude, `'E'` for positive longitude, or `'W'` for negative longitude. `strings` is string or a cell array of strings. For backward compatibility, `strings` can also be a character matrix. If more than one angle is represented, `strings` can either contain homogeneous or heterogeneous formatting (see example). `angles` is a column vector of class double.

## Examples

```
strs = {'23°30''00"N', '23-30-00S', '123d30m00sE', '1233000W'}

strs =
```

```
     '23°30'00"N'    '23-30-00S'    '123d30m00sE'    '1233000W'

str2angle(strs)

ans =
         23.5
        -23.5
         123.5
        -123.5
```

## See Also
angl2str

# surfacem

Project and add geolocated data grid to current map axes

## Syntax

```
surfacem(lat,lon,Z)
surfacem(latlim,lonlim,Z)
surfacem(lat,lon,Z,alt)
surfacem(...,prop1,val1,prop2,val2,...)
h = surfacem(...)
```

## Description

surfacem(lat,lon,Z) constructs a surface to represent the data grid Z in the current map axes. The surface lies flat in the horizontal plane with its CData property set to Z. The vectors or 2-D arrays lat and lon define the latitude-longitude graticule mesh on which Z is displayed. For a complete description of the various forms that lat and lon can take, see surfm.

surfacem(latlim,lonlim,Z) defines the graticule using the latitude and longitude limits latlim and lonlim. These limits should match the geographic extent of the data grid Z. The two-element vector latlim has the form:

```
[southern_limit northern_limit]
```

Likewise, lonlim has the form:

```
[western_limit eastern_limit]
```

A latitude-longitude graticule of size 50-by-100 is constructed. The surface FaceColor property is 'texturemap', except when Z is precisely 50-by-100, in which case it is 'flat'.

surfacem(lat,lon,Z,alt) sets the ZData property of the surface to 'alt', resulting in a 3-D surface. Lat and lon must result in a graticule mesh that matches alt in size.

CData is set to Z. Facecolor is `'texturemap'`, unless Z matches `alt` in size, in which case it is `'flat'`.

surfacem(...,prop1,val1,prop2,val2,...) applies additional MATLAB graphics properties to the surface via property/value pairs. You can specify any property accepted by the `surface` function, except XData, YData, and ZData.

h = surfacem(...) returns a handle to the surface object.

---

**Note** Unlike `meshm` and `surfm`, `surfacem` always adds a surface to the current axes, regardless of hold state.

---

## Examples

Construct a surface to represent the data grid `topo`.

```
  figure('Color','white')
load topo
  latlim = [-90  90];
  lonlim = [  0 360];
  gratsize = 1 + [diff(latlim), diff(wrapTo360(lonlim))]/6;
  [lat, lon] = meshgrat(latlim, lonlim, gratsize);
  worldmap world
  surfacem(lat, lon, topo)
  demcmap(topo)
```
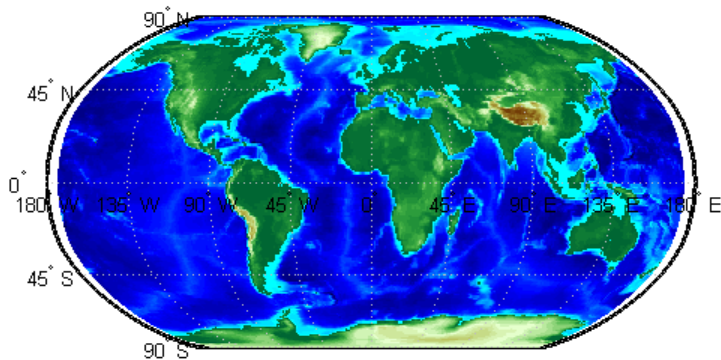
## See Also

geoshow | meshm | pcolorm | surfm

# surflm

3-D shaded surface with lighting on map axes

## Syntax

```
surflm(lat,lon,Z)
surflm(latlim,lonlim,Z)
surflm(...,s)
surflm(...,s,k)
h = surflm(...)
```

## Description

surflm(lat,lon,Z) and surflm(latlim,lonlim,Z) are the same as surfm(...) except that they highlight the surface with a light source. The default light source (45 degrees counterclockwise from the current view) and reflectance constants are the same as in surfl.

surflm(...,s) and surflm(...,s,k) use a light source vector, s, and a vector of reflectance constants, k. For more information on s and k, see the help for surfl.

h = surflm(...) returns a handle to the surface object.

## Examples

Project a 3-D shaded surface with lighting on the current map axes. Note that in the following example, the graticule is the size of topo (180 x 360) and is rendered in 3-D, so it might take a while. It is also memory intensive:

```
figure('Color','white')
load topo
axesm miller
axis off; framem on; gridm on;
[lat,lon] = meshgrat(topo,topolegend);
surflm(lat,lon,topo)
colormap(gray)
```

```
coast = load('coast');
plotm(coast.lat,coast.long,max(topo(:)),...
  'LineWidth',1.5,'Color','yellow')
```
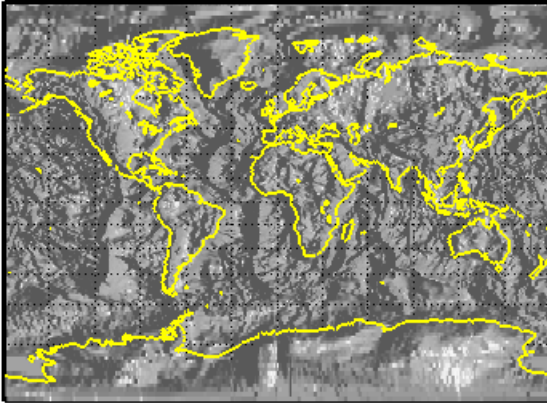


# More About

### Tips

surflm is like surfm, except that it shades the monochrome map surface with a light source, and the only allowed graticule is the size of the data matrix.

## See Also
surfm

# surflsrm

3-D lighted shaded relief of geolocated data grid

## Syntax

```
surflsrm(lat,long,Z)
surflsrm(lat,long,Z,[azim elev])
surflsrm(lat,long,Z,[azim elev],cmap)
surflsrm(lat,long,Z,[azim elev],cmap,clim)
h = surflsrm(...)
```

## Description

surflsrm(lat,long,Z) displays the geolocated data grid, colored according to elevation and surface slopes. The current axes must have a valid map projection definition.

surflsrm(lat,long,Z,[azim elev]) displays the geolocated data grid with the light coming from the specified azimuth and elevation. Lighting is applied before the data is projected. Angles are in degrees, with the azimuth measured clockwise from North, and elevation up from the zero plane of the surface. By default, the direction of the light source is east (90º azimuth) at an elevation of 45º.

surflsrm(lat,long,Z,[azim elev],cmap) displays the geolocated data grid using the provided colormap. The number of grayscales is chosen to keep the size of the shaded colormap below 256. By default, the colormap is constructed from 16 colors and 16 grays. If the vector of azimuth and elevation is empty, the default locations are used.

surflsrm(lat,long,Z,[azim elev],cmap,clim) uses the provided color axis limits, which are, by default, automatically computed from the data.

h = surflsrm(...) returns the handle to the surface drawn.

## Examples

Create a new colormap using demcmap with white colors for the sea and default colors for land. Use this colormap for the lighted shaded relief map of the Middle East region:

```
load mapmtx
[cmap,clim] = demcmap(map1,[],[1 1 1],[]);
axesm loximuth
surflsrm(lt1,lg1,map1,[],cmap,clim)
```
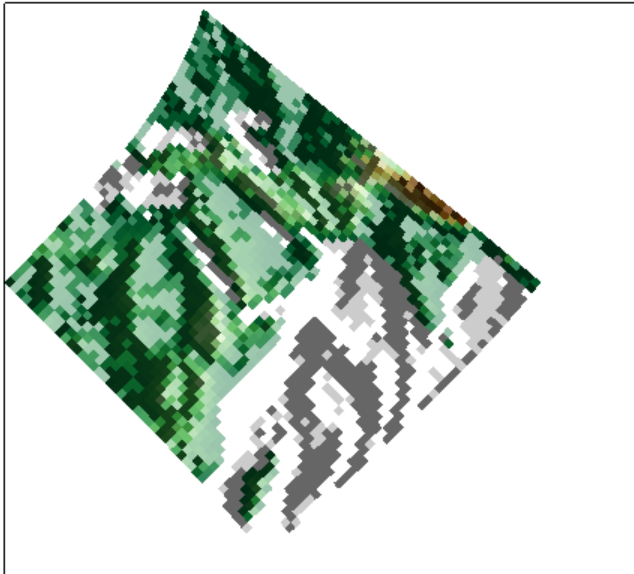


## More About

### Tips

This function effectively multiplies two colormaps, one with color based on elevation, the other with a grayscale based on the slope of the surface, to create a new colormap. This produces an effect similar to using a light on a surface, but with all of the visible colors actually in the colormap. Lighting calculations are performed on the unprojected data.

## See Also

meshlsrm | meshm | pcolorm | shaderel | surfacem | surflm | surfm

# surfm

Project geolocated data grid on map axes

## Syntax

```
surfm(lat,lon,Z)
surfm(latlim,lonlim,Z)
surfm(lat,lon,Z,alt)
surfm(...,prop1,val1,prop2,val2,...)
h = surfm(...)
```

## Description

`surfm(lat,lon,Z)` constructs a surface to represent the data grid `Z` in the current map axes. The surface lies flat in the horizontal plane with its `CData` property set to `Z`. The 2-D arrays or vectors `lat` and `lon` define the latitude-longitude graticule mesh on which `Z` is displayed. The sizes and shapes of `lat` and `lon` affect their interpretation, and also determine whether the default `FaceColor` property of the surface is `'flat'` or `'texturemap'`. There are three options:

- 2-D arrays (matrices) having the same size as `Z`. Lat and `lon` are treated as geolocation arrays specifying the precise location of each vertex. `FaceColor` is `'flat'`.

- 2-D arrays having a different size than `Z`. The arrays `lat` and `lon` define a graticule mesh that might be either larger or smaller than `Z`. Lat and `lon` must match each other in size. `FaceColor` is `'texturemap'`.

- Vectors having more than two elements. The elements of `lat` and `lon` are repeated to form a graticule mesh with size equal to `numel(lat)-by-numel(lon)`. `FaceColor` is `'flat'` if the graticule mesh matches `Z` in size. Otherwise, `FaceColor` is `'texturemap'`.

`surfm` clears the current map if the hold state is `'off'`.

`surfm(latlim,lonlim,Z)` defines the graticule using the latitude and longitude limits `latlim` and `lonlim`, which should match the geographic extent of the data grid `Z`. Latlim is a two-element vector of the form:

```
[southern_limit northern_limit]
```

Likewise `lonlim` has the form:

```
[western_limit eastern_limit]
```

A latitude-longitude graticule is constructed to match `Z` in size. The surface `FaceColor` property is `'flat'` by default.

`surfm(lat,lon,Z,alt)` sets the `ZData` property of the surface to `'alt'`, resulting in a 3-D surface. `lat` and `lon` must result in a graticule mesh that matches `alt` in size. `CData` is set to `Z`. The `FaceColor` property is `'texturemap'`, unless `Z` matches `alt` in size, in which case it is `'flat'`.

`surfm(...,prop1,val1,prop2,val2,...)` applies additional MATLAB graphics properties to the surface via property/value pairs. You can specify any property accepted by the `surface` function except `XData`, `YData`, and `ZData`.

`h = surfm(...)` returns a handle to the surface object.

## Examples

Construct a surface to represent the data grid `topo`.

```
figure('Color','white')
load topo
axesm miller
axis off; framem on; gridm on;
[lat,lon] = meshgrat(topo,topolegend,[90 180]);
surfm(lat,lon,topo)
demcmap(topo)
```
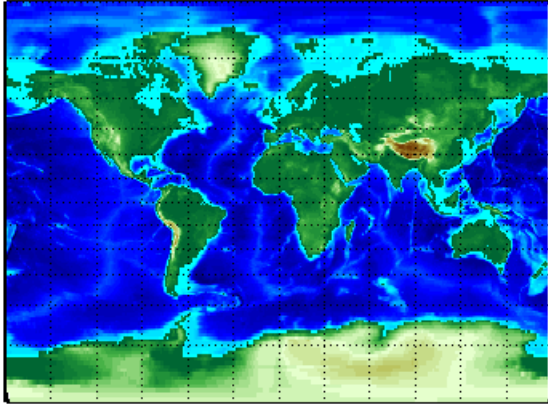
## More About

### Tips

This function warps a data grid to a graticule mesh, which is projected according to the map axes property `MapProjection`. The fineness, or resolution, of this grid determines the quality of the projection and the speed of plotting it. There is no hard and fast rule for sufficient graticule resolution, but in general, cylindrical projections need very few graticule points in the longitudinal direction, while complex curve-generating projections require more.

## See Also
geoshow | meshgrat | meshm | pcolorm | surfacem

# symbolm

Project point markers with variable size

## Syntax

```
symbolm(lat,lon,z,'MarkerType')
symbolm(lat,lon,z,'MarkerType','PropertyName',PropertyValue,...)
h = symbolm(...)
```

## Description

symbolm(lat,lon,z,'MarkerType') constructs a thematic map where the symbol size of each data point (lat, lon) is proportional to it weighting factor (z). The point corresponding to min(z) is drawn at the default marker size, and all other points are plotted with proportionally larger markers. The MarkerType string is a LineSpec string specifying a marker and optionally a color.

symbolm(lat,lon,z,'MarkerType','PropertyName',PropertyValue,...) applies the line properties to all the symbols drawn.

h = symbolm(...) returns a vector of handles to the projected symbols. Each symbol is projected as an individual line object.

## See also

stem3m, plotm, plot

# tagm

Set `Tag` property of map graphics object

## Syntax

```
tagm(hndl,tagstr)
```

## Description

`tagm(hndl,tagstr)` sets the `Tag` property of each object designated in the vector of handles `hndl` to the associated string (row) of the matrix of strings `tagstr`.

This property is recognized by the `namem` and `handlem` functions.

## Examples

Normally, a plotted line has a name of `'line'`:

```
axesm miller
lats = [3 2 1 1 2 3]; longs = [7 8 9 7 8 9];
h=plotm(lats,longs);

untagged = namem(h)
untagged =
line
```

The `tagm` function can rename it:

```
tagm(h,'testpath');
tagged = namem(h)
tagged =
testpath
```

## See Also

`clma` | `clmo` | `handlem` | `hidem` | `namem` | `showm`

# tbase

Read 5-minute global terrain elevations from TerrainBase

## Syntax

```
[Z,refvec] = tbase(scalefactor)
[Z,refvec] = tbase(scalefactor,latlim,lonlim)
```

## Description

`[Z,refvec] = tbase(scalefactor)` reads the data for the entire world, reducing the resolution of the data by the specified scale factor. The result is returned as a regular data grid and an associated three-element referencing vector.

`[Z,refvec] = tbase(scalefactor,latlim,lonlim)` reads the data for the part of the world within the latitude and longitude limits. The limits must be two-element vectors in units of degrees.

## Background

TerrainBase is a global model of terrain and bathymetry on a regular 5-minute grid (approximately 10 km resolution). It is a compilation of the public domain data from almost 20 different sources, including the DCW-DEM and ETOPO5. The data set was created by the National Geophysical Data Center and World Data Center-A for Solid Earth Geophysics in Boulder, Colorado.

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

## Examples

### Read Data for the Entire World But Read Every 10th Point

Set the scale factor to read every 10th point.

```
scalefactor = 10;
```

Read the data and view the size of the returned variables.

```
[Z,refvec] = tbase(scalefactor);
whos Z refvec

Name              Size           Bytes  Class

 Z               216x432       746496  double array
 refvec            1x3              24  double array
```

Determine the latitude and longitude limits of the geographic quadrangle bounding the data grid.

```
limitm(Z,refvec)

ans =

         -90            90          -180           180
```

**Read Data for a Region at Full Resolution**

Set the scale factor to 1 and set the longitude and latitude limits. This example reads the data for Korea and Japan.

```
scalefactor = 1;
latlim = [30 45];
lonlim = [115 145];
```

Read the data and view the size of the returned variables.

```
[Z,refvec] = tbase(scalefactor,latlim,lonlim);
whos

Name      Size          Bytes  Class

 Z        180x360      518400  double array
 refvec     1x3            24  double array
```

## See Also
gtopo30 | etopo | usgsdem

# textm

Project text annotation on map axes

## Syntax

```
textm(lat,lon,string)
textm(lat,lon,z,string)
textm(lat,lon,z,string,PropertyName,PropertyValue,...)
h = textm(...)
```

## Description

`textm(lat,lon,string)` projects the text in `string` onto the current map axes at the locations specified by the `lat` and `lon`. The units of `lat` and `lon` must match the `'angleunits'` property of the map axes. If `lat` and `lon` contain multiple elements, `textm` places a text object at each location. In this case `string` may be a cell array of strings with the same number of elements as `lat` and `lon`. (For backward compatibility, `string` may also be a 2-D character array such that `size(string,1)` matches `numel(lat)`).

`textm(lat,lon,z,string)` draws the text at the altitude(s) specified in `z`, which must be the same size as `lat` and `lon`. The default altitude is 0.

`textm(lat,lon,z,string,PropertyName,PropertyValue,...)` sets the text object properties. All properties supported by the MATLAB `text` function are supported by `textm`.

`h = textm(...)` returns the handles to the text objects drawn.

## Examples

The feature of `textm` that distinguishes it from the standard MATLAB text function is that the `text` object is projected appropriately. Type the following:

```
axesm sinusoid
framem('FEdgeColor','red')
```

```
textm(60,90,'hello')
```
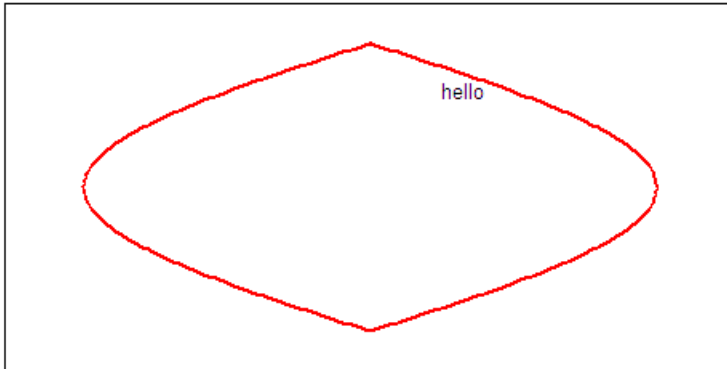


```
figure; axesm miller
framem('FEdgeColor','red')
textm(60,90,'hello')
```
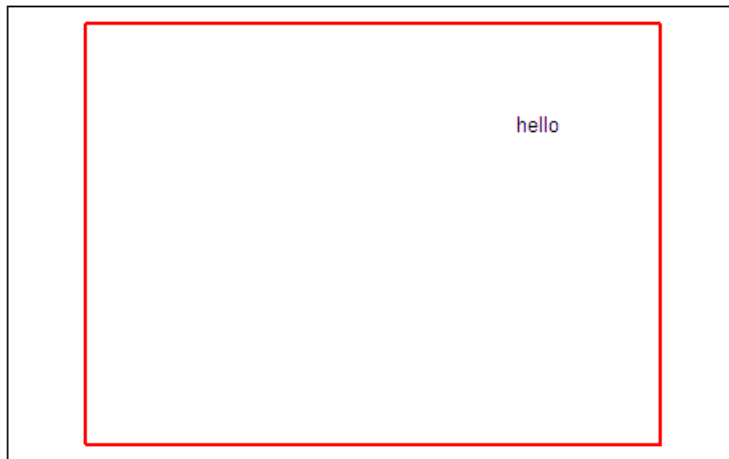


The string `'hello'` is placed at the same geographic point, but it appears to have moved relative to the axes because of the different projections. If you change the projection using

the `setm` function, the text moves as necessary. Use `text` to fix text objects in the axes independent of projection.

## More About

### Tips

You may be working with scalar `lat` and `lon` data or vector `lat` and `lon` data. If you are in scalar mode and you enter a cell array of strings, you will get a text object with a multiline string. Also note that vertical slash characters, rather than producing multiline strings, will yield a single line string containing vertical slashes. On the other hand, if `lat` and `lon` are nonscalar, then the size of the cell array input must match their size exactly.

## See Also

axesm | text

# tgrline

Read TIGER/Line data

---

**Note:** `tgrline` will be removed in a future version. More recent TIGER/Line data sets are available in shapefile format and can be imported using `shaperead`.

---

## Syntax

```
[CL,PR,SR,RR,H,AL,PL] = tgrline(filename)
[CL,PR,SR,RR,H,AL,PL] = tgrline(filename,year)
[CL,PR,SR,RR,H,AL,PL] = tgrline(filename,year,countyname)
```

## Description

`[CL,PR,SR,RR,H,AL,PL] = tgrline(filename)` reads a set of 1994 TIGER/Line files which share the same filename, but different extensions. The results are returned in a set of Mapping Toolbox display structures tagged with feature names and containing:

- county boundaries (CL)
- primary roads (PR)
- secondary roads (SR)
- railroads (RR)
- hydrography (H)
- area landmarks (AL)
- point landmarks (PL)

`[CL,PR,SR,RR,H,AL,PL] = tgrline(filename,year)` reads the TIGER line files in the format from that year. The layout of TIGER/Line files is updated periodically and filename extensions may change from year to year. Valid years are 1990, 1992, 1994, 1995, 1999, 2000, 2002, 2003, and 2004.

`[CL,PR,SR,RR,H,AL,PL] = tgrline(filename,year,countyname)` uses the string `countyname` to tag the county data.

# Background

The United States Census Bureau distributes TIGER/Line data over the Internet and via CD-ROM or DVD.

TIGER/Line files contain vector map data used to support mapping for the U.S. Census Bureau. TIGER is an acronym for Topographically Integrated Geographic Encoding and Referencing. These files contain data for political boundaries, including states, counties, Indian reservations, and census tracts, as well as roads, railroads, hydrography, and landmarks. In addition to the geographically referenced information, the files also contain data to determine the address of an object. The data covers the United States of America and its territories or administrative units: Puerto Rico, the Virgin Islands of the United States, American Samoa, Guam, the Commonwealth of the Northern Marianna Islands, the Republic of Palau, the other Pacific entities that were part of the Trust Territory of the Pacific Islands (the Republic of the Marshall Islands and the Federated States of Micronesia), and the Midway Islands. The most common application of this data is to commercial CD-ROM road atlases.

TIGER/Line is a registered trademark of the United States Census Bureau.

# Examples

Read from the data for Washington, D.C.:

```
[CL,PR,SR,RR,H,AL,PL] = tgrline('TGR11001',1994,'Wash,DC');
```

# More About

### Tips

This function reads only a subset of the data in the TIGER/Line files. For example, the function does not return local roads, zip codes, or census tract numbers.

Data are returned as Mapping Toolbox display structures, which you can then update to geographic data structures. For information about display structure format, see "Version 1 Display Structures" on page 1-200 in the reference page for `displaym`. The `updategeostruct` function performs such conversions.

## See Also

shaperead | updategeostruct

# tightmap

Remove white space around map

## Syntax

```
tightmap
```

## Description

`tightmap` sets the axis limits to be tight around the map in the current axes. This eliminates or reduces the white border between the map frame and the axes box. Use `axis auto` to undo `tightmap`.

## Examples

Display a map of Africa. Notice the white space between the map frame and the edge of the axes box.

```
axesm('miller','maplatlim',[-40 40],'maplonlim',[-20 60])
framem; gridm; mlabel; plabel
load coast
plotm(lat, long)
```

Now use `tightmap` to reduce the wasted space:

```
tightmap
```

## More About

### Tips

The axis limits are fixed. If a change in the projection parameters changes the size or position of the map display within the projected coordinate system, execute `tightmap` again. Also note that `tightmap` needs to be re-applied following any call to `setm` that causes projected map objects to be re-projected.

The `tightmap` function performs no action on a `'globe'` map axes.

## See Also
panzoom | zoom | paperscale | axesscale | previewmap

# timezone

Time zone based on longitude

## Syntax

```
[zd,zltr,zone] = timezone(long)
[zd,zltr,zone] = timezone(long,units)
```

## Description

`[zd,zltr,zone] = timezone(long)` returns an integer zone description, `zd`, an alphabetical string zone indicator, `zltr`, and a string, `zone`, with the complete zone description and alphabetical zone indicator corresponding to the input longitude `long`.

`[zd,zltr,zone] = timezone(long,units)` specifies the angular units with a standard angle *units* string. The default value is `'degrees'`. Valid *units* are:

- `'degrees'` — decimal degrees
- `'radians'`

## Examples

Given that it is locally 1330 (1:30 p.m.) at a longitude of 75ºW, determine GMT:

```
[zd,zltr,zone] = timezone(-75,'degrees')

zd =
     5
zltr =
R
zone =
+5 R
```

Greenwich Mean Time (GMT) is 1330 plus five hours, or 1830 (6:30 p.m.).

# Background

Time is determined by the position of the Sun relative to the prime meridian, the zero longitude line running through Greenwich, England. When this meridian lies directly below the Sun, it is noon GMT. For local times elsewhere, the Earth is divided into 15° longitude bands, each centered on a central meridian. When a central meridian lies directly below the Sun, Local Mean Time (LMT) in that zone is noon. The zone description is an integer that when added to LMT gives GMT. For notational convenience, each zone is also given an alphabetical indicator. The indicator at Greenwich is *Z*, so GMT is often called *ZULU time*.
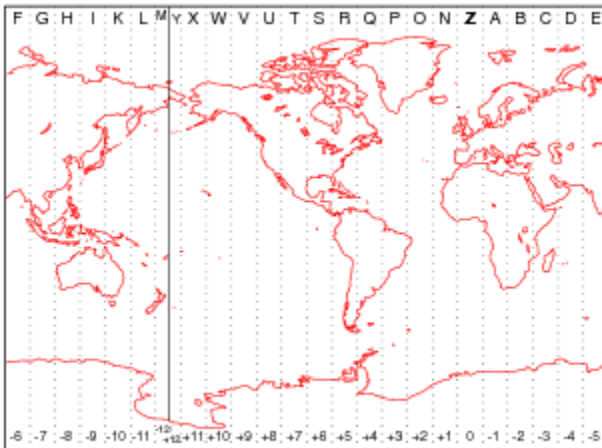


Note that there are actually 25 time zones, because the zone centered on the International Date Line (180° E/W) is split into two: "+12 Y" and "-12 M."

# Limitations

National and local governments set their own time zone boundaries for political or geographic convenience. The `timezone` function does not account for statutory deviations from the meridian-based system.

# tissot

Project Tissot indicatrices on map axes

## Syntax

```
h = tissot
h = tissot(spec)
h = tissot(spec,linestyle)
h = tissot(linestyle)
h = tissot(spec,PropertyName,PropertyValue,...)
h = tissot(linestyle,PropertyName,PropertyValue,...)
```

## Description

`h = tissot` plots the default Tissot diagram, as described above, on the current map axes and returns handles for the displayed indicatrices.

`h = tissot(spec)` allows you to specify plotting parameters of the displayed Tissot diagram as described above.

`h = tissot(spec,linestyle)` and `h = tissot(linestyle)` specify any *linestyle* string recognized by the standard MATLAB `line` function to set the line style of the Tissot indicatrices.

`h = tissot(spec,PropertyName,PropertyValue,...)` and `h = tissot(linestyle,PropertyName,PropertyValue,...)` allow the specification of any property and value recognized by the `line` function.

## Background

Tissot indicatrices are plotting symbols that are useful for understanding the various distortions of a given map projection. The indicatrices are circles of identical true radius on the Earth's surface. When plotted on a map projection, they indicate whether the projection has certain features. If the plotted indicatrices all enclose the same area, the projection is equal area (for example, a Sinusoidal projection would have this feature). If they all remain circular, then conformality is indicated (a Mercator projection has this

property). Distortions in meridianal or parallel distance are exhibited by flattened or stretched indicatrices. Many projections will show very even, circular indicatrices in some regions, often near the center, and wildly distorted indicatrices in others, such as near the edges. The Tissot diagram is therefore very useful in analyzing the appropriateness of a projection to a given purpose or region.

The general layout of the Tissot diagram is defined by the specification vector `spec`.

```
spec = [Radius]
spec = [Latint,Longint]
spec = [Latint,Longint,Radius]
spec = [Latint,Longint,Radius,Points]
```

`Radius` is the small circle radius of each indicatrix circle. If entered, it should be in the same units as the map axes `Geoid`. The default radius is 1/10th the radius of the sphere.

`Latint` is the latitude interval between indicatrix circle centers. If entered it should be in the map axes `AngleUnits`. The default value is one circle every 30º of latitude (that is, 0º, +/-30º, etc.).

`Longint` is the longitude interval between indicatrix circle centers. If entered it should be in the map axes `AngleUnits`. The default value is one circle every 30º of latitude (that is, 0º, +/-30º, etc.).

`Points` is the number of plotting points per circle. The default is 100 points.

## Examples

```
axesm sinusoid; framem
tissot
```
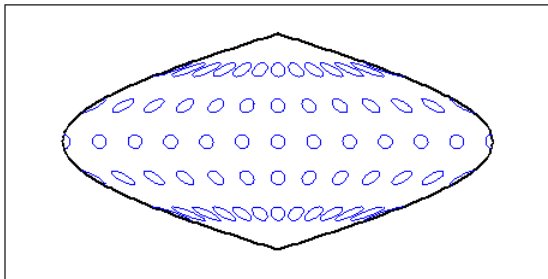
The Sinusoidal projection is equal area.

```
setm(gca,'MapProjection','Mercator')
```
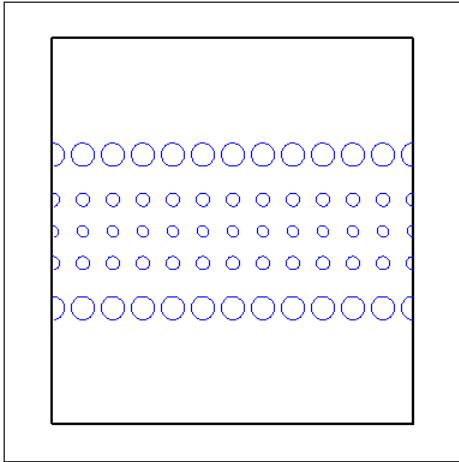


The Mercator projection is conformal.

## More About

· "Supported Map Projections"

## See Also

```
mdistort | distortcalc
```

# toDegrees

Convert angles to degrees

## Syntax

```
[angle1InDegrees, angle2InDegrees, ...] = toDegrees(fromUnits,
angle1, angle2, ...)
```

## Description

`[angle1InDegrees, angle2InDegrees, ...] = toDegrees(`*`fromUnits`*`,
angle1, angle2, ...)` converts `angle1, angle2, ...` to degrees from the
specified angle units. *fromUnits* can be either `'degrees'` or `'radians'` and may
be abbreviated. The inputs `angle1, angle2, ...` and their corresponding outputs
are numeric arrays of various sizes, with `size(angleNinDegrees)` matching
`size(angleN)`.

## See Also

`fromDegrees`, `fromRadians`, `rad2deg`, `toRadians`

# toRadians

Convert angles to radians

## Syntax

```
[angle1InRadians, angle2InRadians, ...] = toRadians(fromUnits,
angle1, angle2, ...)
```

## Description

`[angle1InRadians, angle2InRadians, ...] = toRadians(`*fromUnits*`,
angle1, angle2, ...)` converts `angle1, angle2, ...` to radians from the specified angle units. *fromUnits* can be either `'degrees'` or `'radians'` and may be abbreviated. The inputs `angle1, angle2, ...` and their corresponding outputs are numeric arrays of various sizes, with `size(angleNinRadians)` matching `size(angleN)`.

## See Also

`deg2rad`, `fromDegrees`, `fromRadians`, `toDegrees`

# track

Track segments to connect navigational waypoints

## Syntax

```
[lattrk,lontrk] = track(waypts)
[lattrk,lontrk] = track(waypts,units)
[lattrk,lontrk] = track(lat,lon)
[lattrk,lontrk] = track(lat,lon,ellipsoid)
[lattrk,lontrk] = track(lat,lon,ellipsoid,units,npts)
[lattrk,lontrk] = track(method,lat,...)
trkpts = track(lat,lon...)
```

## Description

[lattrk,lontrk] = track(waypts) returns points in lattrk and lontrk along a track between the waypoints provided in navigational track format in the two-column matrix waypts. The outputs are column vectors in which successive segments are delineated with NaNs.

[lattrk,lontrk] = track(waypts,*units*) specifies the units of the inputs and outputs, where *units* is any valid angle unit string. The default is 'degrees'.

[lattrk,lontrk] = track(lat,lon) allows the user to input the waypoints in two vectors, lat and lon.

[lattrk,lontrk] = track(lat,lon,ellipsoid) specifies the shape of the Earth using ellipsoid, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form [semimajor_axis eccentricity]. The default ellipsoid is a unit sphere

[lattrk,lontrk] = track(lat,lon,ellipsoid,*units*,npts) establishes how many intermediate points are to be calculated for every track segment. By default, npts is 30.

[lattrk,lontrk] = track(*method*,lat,...) establishes the logic to be used to determine the intermediate points along the track between waypoints. Because this is a

navigationally motivated function, the default method is `'rh'`, which results in rhumb line logic. Great circle logic can be specified with `'gc'`.

`trkpts = track(lat,lon...)` compresses the output into one two-column matrix, `trkpts`, in which the first column represents latitudes and the second column, longitudes.

## Examples

The `track` function is useful for generating data in order to display tracks. Lieutenant Sextant is the navigator of the USS Neversail. He is charged with plotting a track to take Neversail from the Straits of Gibraltar to Port Said, Egypt, the northern end of the Suez Canal. He has picked appropriate waypoints and now would like to display the track for his captain's approval.

First, display a chart of the Mediterranean Sea:

```
load coast
axesm('mercator','MapLatLimit',[28 47],'MapLonLimit',[-10 37],...
    'Grid','on','Frame','on','MeridianLabel','on','ParallelLabel','on')
geoshow(lat,long,'DisplayType','line','color','b')
```
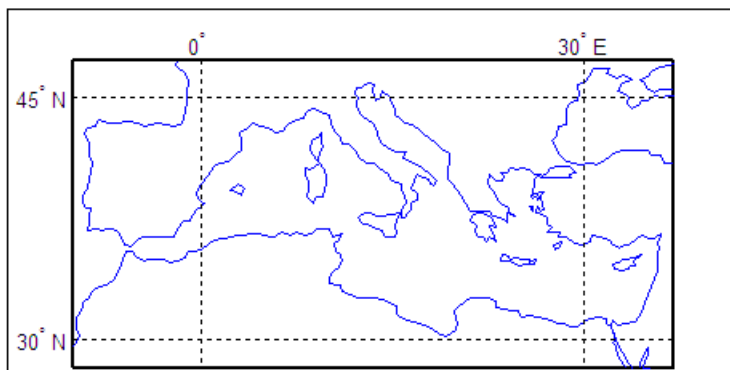


These are the waypoints Lt. Sextant has selected:

```
waypoints = [36,-5; 36,-2; 38,5; 38,11; 35,13; 33,30; 31.5,32]
```

```
waypoints =
    36.0000    -5.0000
    36.0000    -2.0000
    38.0000     5.0000
    38.0000    11.0000
    35.0000    13.0000
    33.0000    30.0000
    31.5000    32.0000
```

Now display the track:

```
[lttrk,lntrk] = track('rh',waypoints,'degrees');
geoshow(lttrk,lntrk,'DisplayType','line','color','r')
```

With a display this clear, the captain gladly approves the plan.
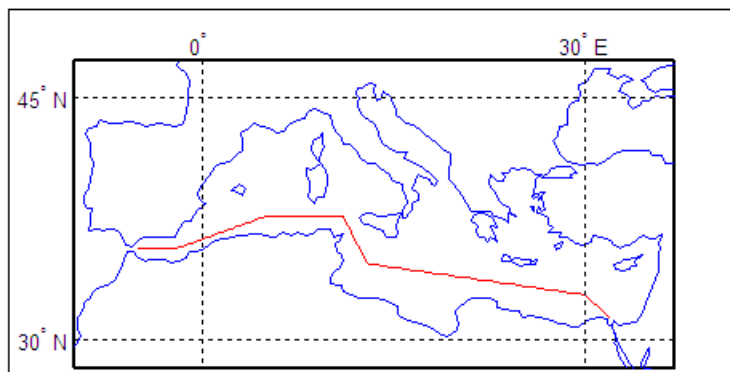


## See Also
dreckon | gcwaypts | legs | navfix

# track1

Geographic tracks from starting point, azimuth, and range

## Syntax

```
[lat,lon] = track1(lat0,lon0,az)
[lat,lon] = track1(lat0,lon0,az,arclen)
[lat,lon] = track1(lat0,lon0,az,arclen,ellipsoid)
[lat,lon] = track1(lat0,lon0,az,angleunits)
[lat,lon] = track1(lat0,lon0,az,arclen,angleunits)
[lat,lon] = track1(lat0,lon0,az,arclen,ellipsoid,angleunits)
[lat,lon] = track1(lat0,lon0,az,arclen,ellipsoid,angleunits,npts)
[lat,lon] = track1(trackstr,...)
mat = track1(...)
```

## Description

`[lat,lon] = track1(lat0,lon0,az)` computes complete great circle tracks on a sphere starting at the point `lat0,lon0` and proceeding along the input azimuth, `az`. The inputs can be scalar or column vectors.

`[lat,lon] = track1(lat0,lon0,az,arclen)` uses the input `arclen` to specify the arc length of the great circle track. `arclen` is specified in units of degrees of arc. If `arclen` is a column vector, then the track is computed from the starting point, with positive distance measured easterly. If `arclen` is a two column matrix, then the track is computed starting at the range in the first column and ending at the range in the second column. If `arclen = []`, then the complete track is computed.

`[lat,lon] = track1(lat0,lon0,az,arclen,ellipsoid)` computes the track along a geodesic arc on the ellipsoid defined by the input `ellipsoid`, which can be a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. `arclen` must be expressed in length units that match the units of the semimajor axis — unless `ellipsoid` is `[]` or the semimajor axis length is zero. In these special cases, `arclen` is assumed to be in degrees of arc and the tracks are computed on a sphere, as in the preceding syntax.

`[lat,lon] = track1(lat0,lon0,az,angleunits)`,

[lat,lon] = track1(lat0,lon0,az,arclen,*angleunits*), and
[lat,lon] = track1(lat0,lon0,az,arclen,ellipsoid,*angleunits*) use the string angleunits to specify the angle units of the inputs and outputs. angleunits can equal 'degrees' or 'radians'.

[lat,lon] = track1(lat0,lon0,az,arclen,ellipsoid,*angleunits*,npts) uses the scalar input npts to specify the number of points per track. The default value of npts is 100.

[lat,lon] = track1(*trackstr*,...) uses the string trackstr to define either a great circle or rhumb line track. trackstr can equal 'gc' or 'rh'. If trackstr is 'gc', then either great circle (given a sphere) or geodesic (given an ellipsoid) tracks are computed. If trackstr is 'rh', then the rhumb line tracks are computed.

mat = track1(...) returns a single output argument mat such that mat = [lat lon]. This is useful if only a single track is computed.

Multiple tracks can be defined from a single starting point by providing scalar lat0 and lon0 and column vectors for az and arclen.

## Definitions

A path along the surface of the Earth connecting two points is a *track*. Two types of track lines are of interest geographically, *great circles* and *rhumb lines*. Great circles represent the shortest possible path between two points. Rhumb lines are paths with constant angular headings. They are not, in general, the shortest path between two points.

Full great circles bisect the Earth; the ends of the track meet to form a complete circle. Rhumb lines with true east or west azimuths are parallels; the ends also meet to form a complete circle. All other rhumb lines terminate at the poles; their ends do not meet.

## Examples

```
% Set up the axes.
axesm('mercator','MapLatLimit',[-60 60],'MapLonLimit',[-60 60])
gridm on; plabel on; mlabel on;

% Plot the great circle track in green.
[lattrkgc,lontrkgc] = track1(0,0,45,[-55 55]);
```

```
plotm(lattrkgc,lontrkgc,'g')

% Plot the rhumb line track in red.
[lattrkrh,lontrkrh] = track1('rh',0,0,45,[-55 55]);
plotm(lattrkrh,lontrkrh,'r')
```
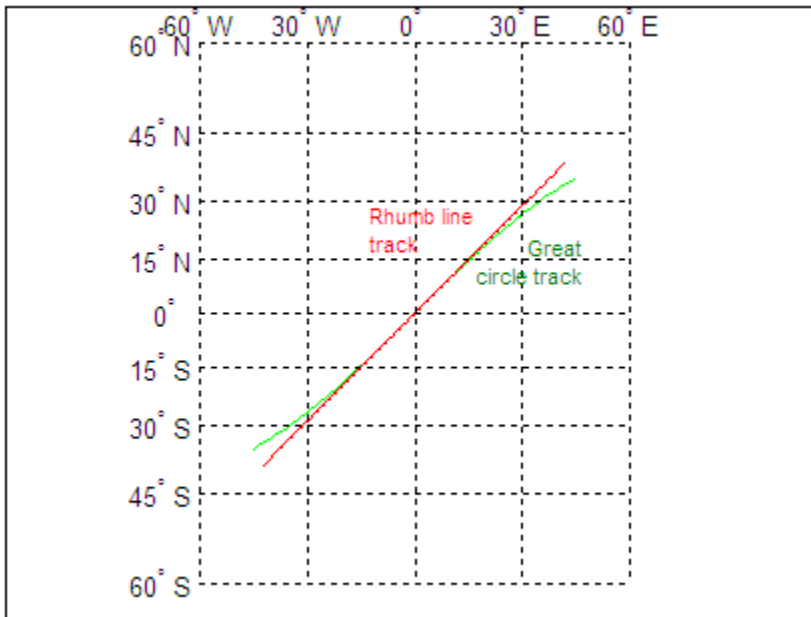


## See Also

azimuth | distance | reckon | scircle1 | scircle2 | track | track2 | trackg

# track2

Geographic tracks from starting and ending points

## Syntax

```
[lat,lon] = track2(lat1,lon1,lat2,lon2)
[lat,lon] = track2(lat1,lon1,lat2,lon2,ellipsoid)
[lat,lon] = track2(lat1,lon1,lat2,lon2,units)
[lat,lon] = track2(lat1,lon1,lat2,lon2,ellipsoid,units)
[lat,lon] = track2(lat1,lon1,lat2,lon2,ellipsoid,units,npts)
[lat,lon] = track2(track,...)
mat = track2(...)
```

## Description

`[lat,lon] = track2(lat1,lon1,lat2,lon2)` computes great circle tracks on a sphere starting at the point `lat1,lon1` and ending at `lat2,lon2`. The inputs can be scalar or column vectors.

`[lat,lon] = track2(lat1,lon1,lat2,lon2,ellipsoid)` computes the great circle track on the ellipsoid defined by the input `ellipsoid`. `ellipsoid` is a referenceSphere, referenceEllipsoid, or oblateSpheroid object, or a vector of the form `[semimajor_axis eccentricity]`. If `ellipsoid = []`, a sphere is assumed.

`[lat,lon] = track2(lat1,lon1,lat2,lon2,units)` and
`[lat,lon] = track2(lat1,lon1,lat2,lon2,ellipsoid,units)` are both valid calling forms, which use the input string *units* to define the angle units of the inputs and outputs. If the input string *units* is omitted, `'degrees'` is assumed.

`[lat,lon] = track2(lat1,lon1,lat2,lon2,ellipsoid,units,npts)` uses the scalar input `npts` to determine the number of points per track computed. The default value of `npts` is 100.

`[lat,lon] = track2(track,...)` uses the *track* string to define either a great circle or a rhumb line track. If *track* = `'gc'`, then great circle tracks are computed. If *track* = `'rh'`, then rhumb line tracks are computed. If the *track* string is omitted, `'gc'` is assumed.

`mat = track2(...)` returns a single output argument where `mat = [lat lon]`. This is useful if a single track is computed. Multiple tracks can be defined from a single starting point by providing scalar inputs for `lat1,lon1` and column vectors for `lat2,lon2`.

## Definitions

A path along the surface of the Earth connecting two points is a *track*. Two types of track lines are of interest geographically, *great circles* and *rhumb lines*. Great circles represent the shortest possible path between two points. Rhumb lines are paths with constant angular headings. They are not, in general, the shortest path between two points.

## Examples

```
% Set up the axes.
axesm('mercator','MapLatLimit',[30 50],'MapLonLimit',[-40 40])

% Calculate the great circle track.
[lattrkgc,lontrkgc] = track2(40,-35,40,35);

% Calculate the rhumb line track.
[lattrkrh,lontrkrh] = track2('rh',40,-35,40,35);

% Plot both tracks.
plotm(lattrkgc,lontrkgc,'g')
plotm(lattrkrh,lontrkrh,'r')
```
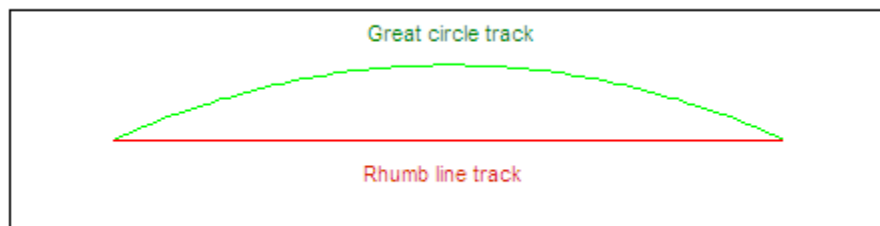
## See Also

azimuth | distance | reckon | scircle1 | scircle2 | track | track1 | trackg

# trackg

Great circle or rhumb line defined via mouse input

## Syntax

```
h = trackg(ntrax)
h = trackg(ntrax,npts)
h = trackg(ntrax,linestyle)
h = trackg(ntrax,PropertyName,PropertyValue,...)
[lat,lon] = trackg(ntrax,npts,...)
h = trackg(track,ntrax,...)
```

## Description

h = trackg(ntrax) brings forward the current map axes and waits for the user to make (2 x ntrax) mouse clicks. The output h is a vector of handles for the ntrax track segments, which are then displayed.

h = trackg(ntrax,npts) specifies the number of plotting points to be used for each track segment. npts is 100 by default.

h = trackg(ntrax,linestyle) specifies the line style for the displayed track segments, where linestyle is any line style string recognized by the standard MATLAB line function.

h = trackg(ntrax,PropertyName,PropertyValue,...) allows property name/property value pairs to be set, where PropertyName and PropertyValue are recognized by the line function.

[lat,lon] = trackg(ntrax,npts,...) returns the coordinates of the plotted points rather than the handles of the track segments. Successive segments are stored in separate columns of lat and lon.

h = trackg(track,ntrax,...) specifies the logic with which tracks are calculated. If the string track is 'gc' (the default), a great circle path is used. If track is 'rh', rhumb line logic is used.

This function is used to define great circles or rhumb lines for display using mouse clicks. For each track, two clicks are required, one for each endpoint of the desired track segment. You can modify the track after creation by **Shift**+clicking it. The track is then in edit mode, during which you can change the length and position by dragging control points, or by entering values into a control panel. **Shift**+clicking again exits edit mode.

## See Also
track1 | track2 | scircleg

# trimcart

Trim graphic objects to map frame

## Syntax

```
trimcart(h)
```

## Description

`trimcart(h)` clips the graphic objects to the map frame. `h` can be a handle or a vector of handles to graphics objects. `h` can also be any object name recognized by `handlem`. `trimcart` clips lines, surfaces, and text objects.

## Examples

```
figure; axesm('miller')
framem
[x, y] = humps(0:.05:1);
h = plot(x, y/25, 'r+-');
load coast
geoshow(lat, long)
trimcart(h)
```
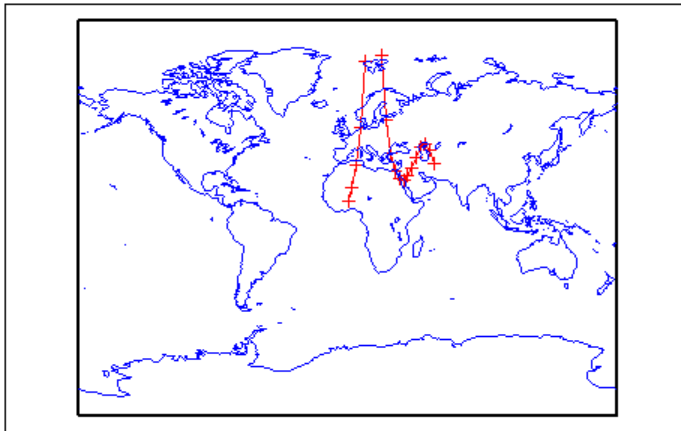
## Limitations

`trimcart` does not trim patch objects.

## See Also

`handlem` | `makemapped`

# trimdata

Trim map data exceeding projection limits

## Syntax

```
[ymat,xmat,trimpts] = trimdata(ymat,ylim,xmat,xlim,'object')
```

## Description

`[ymat,xmat,trimpts] = trimdata(ymat,ylim,xmat,xlim,'object')` identifies points in map data that exceed projection limits. The projection limits are defined by the lower and upper inputs. The particular object to be trimmed is identified by the `'object'` input.

Allowable object strings are

- `'surface'` for trimming graticules
- `'light'` for trimming lights,
- `'line'` for trimming lines
- `'patch'` for trimming patches
- `'text'` for trimming text object location points
- `'none'` to skip all trimming operations

## See Also

`clipdata` | `undotrim` | `undoclip`

# undoclip

Remove object clips introduced by `clipdata`

## Syntax

```
[lat,long] = undoclip(lat,long,clippts,'object')
```

## Description

`[lat,long] = undoclip(lat,long,clippts,'object')` removes the object clips introduced by `clipdata`. This function is necessary to properly invert projected data from the Cartesian space to the original latitude and longitude data points.

The input variable, `clippts`, must be constructed by the function `clipdata`.

Allowable object strings are

- `'surface'` for trimming graticules
- `'light'` for trimming lights
- `'line'` for trimming lines
- `'patch'` for trimming patches
- `'text'` for trimming text object location points
- `'none'` to skip all trimming operations

## See Also
`clipdata` | `trimdata` | `undotrim`

# undotrim

Remove object trims introduced by `trimdata`

## Syntax

```
[ymat,xmat] = undotrim(ymat,xmat,trimpts,'object')
```

## Description

`[ymat,xmat] = undotrim(ymat,xmat,trimpts,'object')` removes the object trims introduced by `trimdata`. This function is necessary to properly invert projected data from the Cartesian space to the original latitude and longitude data points.

The input variable, `trimpts`, must be constructed by the function `trimdata`.

Allowable object strings are

- `'surface'` for trimming graticules
- `'light'` for trimming lights
- `'line'` for trimming lines
- `'patch'` for trimming patches
- `'text'` for trimming text object location points
- `'none'` to skip all trimming operations

## See Also
`clipdata` | `trimdata` | `undoclip`

# unitsratio

Unit conversion factors

## Syntax

```
ratio = unitsratio(to,from)
```

## Description

`ratio = unitsratio(to,from)` returns the number of `to` units per one `from` unit. For example, `unitsratio('cm','m')` returns 100 because there are 100 centimeters per meter. The `unitsratio` function makes it easy to convert from one system of units to another. Specifically, if `x` is in units `from` and

```
y = unitsratio(to, from) * x
```

then `y` is in units `to`.

`to` and `from` can be any strings from the second column of one of the following tables (both must come from the same table). The variables `to` and `from` are case insensitive and can be either singular or plural.

## Units of Length

`unitsratio` recognizes the identifiers listed in the `validateLengthUnit` function.

## Units of Angle

`unitsratio` recognizes the following identifiers for converting units of angle:

| Unit Name | String(s) |
|---|---|
| radian | `'rad'`, `'radian(s)'` |
| degree | `'deg'`, `'degree(s)'` |

# Examples

```
% Approximate mean earth radius in meters
radiusInMeters = 6371000

% Conversion factor
feetPerMeter = unitsratio('feet', 'meter')

% Radius in (international) feet:
radiusInFeet = feetPerMeter * radiusInMeters

% The following prints a true statement for valid TO, FROM pairs:
to   = 'feet';
from = 'mile';
sprintf('There are %g %s per %s.', unitsratio(to,from), to, from)

% The following prints a true statement for valid TO, FROM pairs:
to   = 'degrees';
from = 'radian';
sprintf('One %s is %g %s.', from, unitsratio(to,from), to)
```

# unitstr

Check spatiotemporal unit strings and abbreviations

---

**Note:** The `unitstr` function is obsolete and will be removed in a future release. The syntax `str = unitstr(str,'times')` has already been removed.

---

## Syntax

```
unitstr
str = unitstr(str0,'angles')
str = unitstr(str0,'distances')
```

## Description

`unitstr`, with no arguments, displays a list of strings and abbreviations, recognized by certain Mapping Toolbox functions, for units of angle and length/distance.

`str = unitstr(str0,'angles')` checks for valid angle unit strings or abbreviations. If a valid string or abbreviation is found, it is converted to a standardized, preset string. `'angles'` can be abbreviated.

`str = unitstr(str0,'distances')` checks for valid length unit strings or abbreviations. If a valid string or abbreviation is found, it is converted to a standardized, preset string. `'distances'` can be abbreviated. Note that input strings `'miles'` and `'mi'` are converted to `'statutemiles'`; there is no way to specify international miles in the `unitstr` function.

## Examples

This function recognizes and standardizes certain abbreviations:

```
str = unitstr('sm','distances')
```

```
str =
statutemiles
```

And any unique truncation:

```
str = unitstr('ra','angles')

str =
radians
```

## See Also

```
unitsratio
```

# unwrapMultipart

Unwrap vector of angles with NaN-delimited parts

## Syntax

```
unwrapped = unwrapMultipart(p)
unwrapped = unwrapMultipart(p,angleUnit)
```

## Description

`unwrapped = unwrapMultipart(p)` unwraps a row or column vector of azimuths, longitudes, or phase angles. Input and output units are both radians. If `p` is separated into multiple parts delimited by values of NaN, each part is unwrapped independently. If `p` has only one part, the result is equivalent to `unwrap(p)`. The output is the same size as the input and has NaNs in the same locations.

`unwrapped = unwrapMultipart(p,angleUnit)` unwraps a row or column vector of azimuths, longitudes, or phase angles, where `angleUnit` specifies the unit used for the input and output angles: `'degrees'` or `'radians'`.

## Examples

### Example 1

Compare the behavior `unwrapMultipart` to that of `unwrap`. The output of `unwrapMultipart` starts over again at 6.11 following the NaN, unlike the output of `unwrap`. The output of `unwrapMultipart` is equivalent to a concatenation (with NaN-separator) of separate calls to `unwrap`:

```
p1 = [0.17          5.67          4.89          4.10];
p2 = [6.11          1.05          2.27];
unwrap([p1 NaN p2])

ans =
    0.1700   -0.6132   -1.3932   -2.1832      NaN  -0.1732    1.0500    2.2700

unwrapMultipart([p1 NaN p2])
```

```
ans =
    0.1700   -0.6132   -1.3932   -2.1832         NaN    6.1100    7.3332    8.5532

[unwrap(p1) NaN unwrap(p2)]

ans =
    0.1700   -0.6132   -1.3932   -2.1832         NaN    6.1100    7.3332    8.5532
```

## Example 2

Wrap two revolutions of a sphere to π with `wrapToPi`, and then unwrap it with `unWrapMultipart`:

```
lon = wrapToPi(deg2rad(0:10:720));
unwrappedlon = unwrapMultipart(lon);
figure; hold on
plot(lon,'--')
plot(unwrappedlon)
xlabel 'Point Number'
ylabel 'Longitude in radians'
```
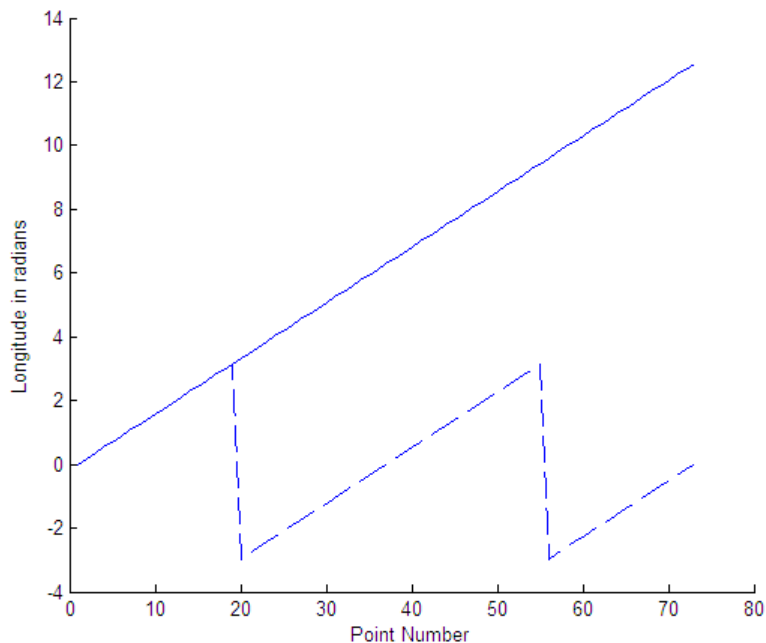
## See Also

`unwrap | wrapTo180 | wrapTo360 | wrapToPi | wrapTo2Pi`

# updategeostruct

Convert line or patch display structure to geostruct

## Syntax

```
geostruct = updategeostruct(displaystruct)
geostruct = updategeostruct(displaystruct, str)
[geostruct,symbolspec] = updategeostruct(displaystruct, ...)
[geostruct,symbolspec] = updategeostruct(displaystruct, ..., cmap)
```

## Description

`geostruct = updategeostruct(displaystruct)` accepts a Mapping Toolbox display structure `displaystruct`. If `displaystruct` is a vector display structure for which the `'type'` field has value `'line'` or `'patch'`, `updategeostruct` restructures its elements to create a geostruct, `geostruct`. If `displaystruct` is a already geographic data structure, it is copied unaltered to `geostruct`. `updategeostruct` does not update display structure arrays of type `'text'`, `'light'`, `'regular'`, or `'surface'`.

`geostruct = updategeostruct(displaystruct, str)` selects only elements whose `tag` field begins with the string `str` (and whose `type` field is either `'line'` or `'patch'`). The selection is case insensitive.

`[geostruct,symbolspec] = updategeostruct(displaystruct, ...)` restructures a display structure and determines a symbolspec based on the graphic properties specified in the `otherproperty` field for each element of `displaystruct` and, if necessary, the `jet` colormap.

`[geostruct,symbolspec] = updategeostruct(displaystruct, ..., cmap)` specifies a colormap, `cmap`, to define the colors used in symbolspec.

## Examples

Update and display the Great Lakes display structure to a geostruct:

```
load greatlakes
cmap = cool(3*numel(greatlakes));
[gtlakes, spec] = updategeostruct(greatlakes, cmap);
lat = extractfield(gtlakes,'Lat');
lon = extractfield(gtlakes,'Lon');
lonlim = [min(lon) max(lon)];
latlim = [min(lat) max(lat)];
figure
usamap(latlim, lonlim);
geoshow(gtlakes, 'SymbolSpec', spec)
```
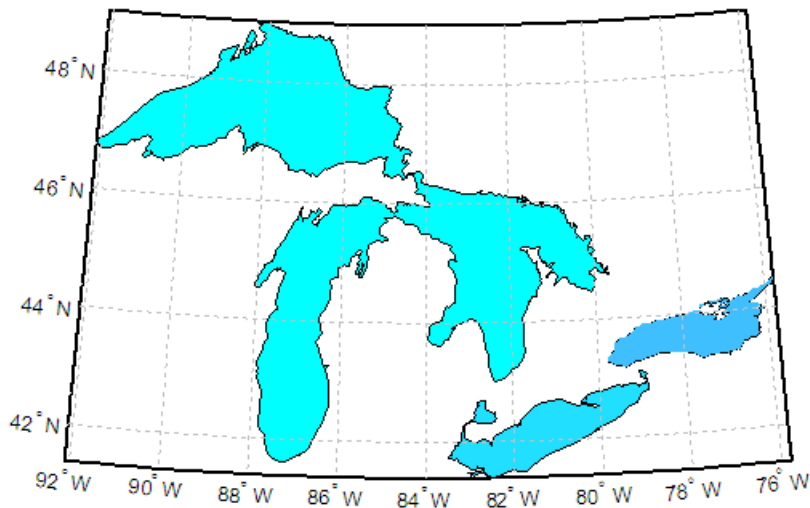


## More About

### Tips

There are two Mapping Toolbox encodings for vector features that use MATLAB structure arrays. In both cases there is one feature per array element, and in both cases a given array's elements all held the same type of feature. Version 1.3.1 and earlier of the Mapping Toolbox software only supported Mapping Toolbox display structures. Version 2.0 introduced a data structure for vector geodata which was less rigidly defined and

more open-ended. The new structures are called *geostructs* (if they contain geographic coordinate data) and *mapstructs* (if they contain projected coordinate data). Over time, display structures are being phased out of the toolbox; the `updategeostruct` function is provided to help users migrate from the old display structure format to the current geostruct/mapstruct format.

A Version 1 Mapping Toolbox display structure is a MATLAB structure that can contain line, patch, text, regular data grid, geolocated data grid, and light objects. The `displaym` function does not accept geostructs produced by Version 2 of the Mapping Toolbox software.

Display structures for lines and patches and Line and Polygon geostructs have the following things in common:

- A field that specifies the type of feature geometry:

  - A `type` field a display structure (value: `'line'` or `'patch'`)
  - A `Geometry` field for a geostruct (value: 'Line' or `'Polygon'`)

- A latitude field:

  - `lat` for a display structure
  - `Lat` for a geostruct

- A longitude field:

  - `long` for a display structure
  - `Lon` for a geostruct

In terms of their differences,

- A geostruct has a `BoundingBox` field; there is no display structure counterpart for this
- A geostruct typically has one or more "attribute" fields, whose values must be either scalar doubles or strings, with arbitrary field names. The presence or absence of a given attribute field—and its value—is dependent on the specific data set that the geostruct represents.
- A (line or patch) display structure has the following fields:

  - A `tag` field that names an individual feature or object
  - An `altitude` coordinate array that extends coordinates to 3-D

1-1317

- An `otherproperty` field in which MATLAB graphics can be specified explicitly, on a per-feature basis

Object properties used in the display are taken from the `otherproperty` field of the structure. If a line or patch object's `otherproperty` field is empty, `displaym` uses default colors. A patch is assigned an index into the current colormap based on the structure's `tag` field. Lines are assigned colors from the current color order according to their tags.

The newer geostruct representation has significant advantages:

- It can represent a much wider range of attributes (display structures essentially can represent only a feature name).
- The geostruct representation (in combination with `geoshow` and `makesymbolspec`) keeps graphics display properties separate from the intrinsic properties of the geographic features themselves.

For example, a road-class attribute can be used to display major highways with a distinctive color and greater line width than secondary roads. The same geographic data structure can be displayed in many different ways, without altering any of its contents, and shapefile data imported from external sources need not be altered to control its graphic display.

For information about the display structure format, see "Version 1 Display Structures" on page 1-200 in the reference page for `displaym`. For a discussion of the characteristics of geographic data structures, see "Mapping Toolbox Geographic Data Structures" in the *Mapping Toolbox User's Guide*.

## See Also

`displaym` | `geoshow` | `makesymbolspec` | `mapshow` | `mapview` | `shaperead`

# usamap

Construct map axes for United States of America

## Syntax

```
usamap state
usamap(state)
usamap 'conus'
usamap('conus')
usamap
usamap(latlim, lonlim)
usamap(Z, R)
h = usamap(...)
h = usamap('all')
```

## Description

`usamap state` or `usamap(state)` constructs an empty map axes with a Lambert Conformal Conic projection and map limits covering a U.S. state or group of states specified by input `state`. `state` may be a string or a cell array of strings, where each string contains the name of a state or `'District of Columbia'`. Alternatively, `state` may be a standard two-letter U.S. Postal Service abbreviation. The map axes is created in the current axes and the axis limits are set tight around the map frame.

`usamap 'conus'` or `usamap('conus')` constructs an empty map axes for the conterminous 48 states (i.e. excluding Alaska and Hawaii).

`usamap` with no arguments asks you to choose from a menu of state names plus `'District of Columbia'`, `'conus'`, and `'all'`.

`usamap(latlim, lonlim)` constructs an empty Lambert Conformal map axes for a region of the U.S. defined by its latitude and longitude limits in degrees. `latlim` and `lonlim` are two-element vectors of the form `[southern_limit northern_limit]` and `[western_limit eastern_limit]`, respectively.

`usamap(Z, R)` derives the map limits from the extent of a regular data grid georeferenced by `R`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel.

`h = usamap(...)` returns the handle of the map axes.

`h = usamap('all')` constructs three empty axes, inset within a single figure, for the conterminous states, Alaska, and Hawaii, with a spherical Earth model and other projection parameters suggested by the U.S. Geological Survey. The maps in the three axes are shown at approximately the same scale. The handles for the three map axes are returned in `h`. `h(1)` is for the conterminous states, `h(2)` is for Alaska, and `h(3)` is for Hawaii. `usamap('allequal')` is the same as `usamap('all')`; usage of `'allequal'` will be removed in a future release.

All axes created with `usamap` are initialized with a spherical Earth model having a radius of 6,371,000 meters.

## Examples

### Example 1

Make a map of Alabama only:

```
usamap('Alabama')
alabamahi = shaperead('usastatehi', 'UseGeoCoords', true,...
          'Selector',{@(name) strcmpi(name,'Alabama'), 'Name'});
geoshow(alabamahi, 'FaceColor', [0.3 1.0, 0.675])
```

```
textm(alabamahi.LabelLat, alabamahi.LabelLon, alabamahi.Name,...
   'HorizontalAlignment', 'center')
```
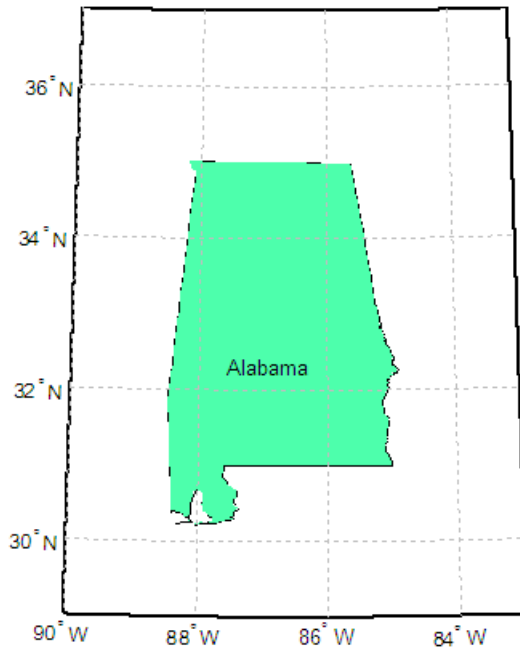


## Example 2

Map a region extending from California to Montana:

```
figure; ax = usamap({'CA','MT'});
set(ax, 'Visible', 'off')
latlim = getm(ax, 'MapLatLimit');
lonlim = getm(ax, 'MapLonLimit');
states = shaperead('usastatehi',...
        'UseGeoCoords', true, 'BoundingBox', [lonlim', latlim']);
geoshow(ax, states, 'FaceColor', [0.5 0.5 1])

lat = [states.LabelLat];
lon = [states.LabelLon];
tf = ingeoquad(lat, lon, latlim, lonlim);
textm(lat(tf), lon(tf), {states(tf).Name}, ...
   'HorizontalAlignment', 'center')
```
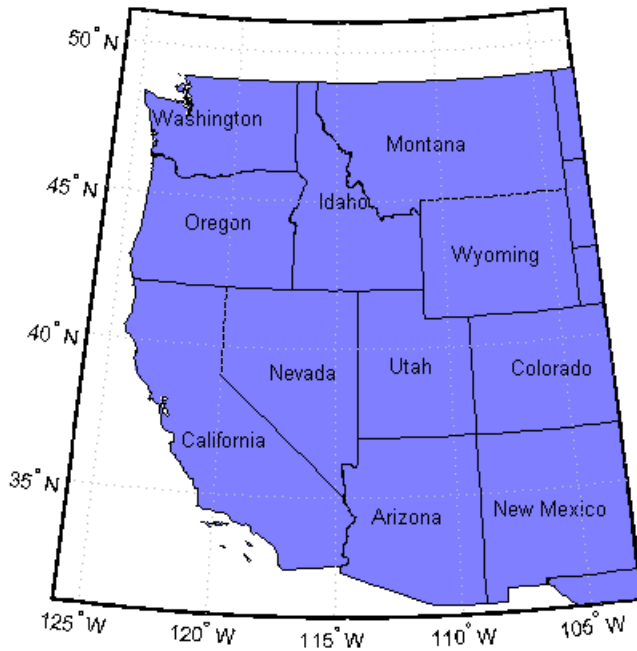
## Example 3

Map the Conterminous United States with a different fill color for each state:

```
figure; ax = usamap('conus');
states = shaperead('usastatelo', 'UseGeoCoords', true,...
  'Selector',...
  {@(name) ~any(strcmp(name,{'Alaska','Hawaii'})), 'Name'});
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))}); %NOTE - colors are random
geoshow(ax, states, 'DisplayType', 'polygon', ...
   'SymbolSpec', faceColors)
framem off; gridm off; mlabel off; plabel off
```
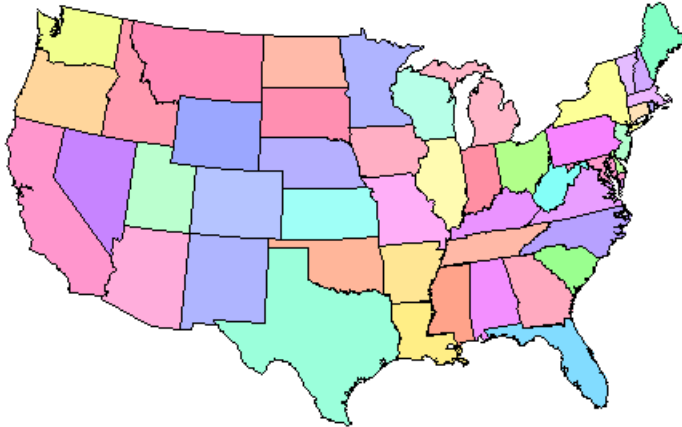
## Example 4

Map of the USA with separate axes for Alaska and Hawaii:

```
figure; ax = usamap('all');
set(ax, 'Visible', 'off')
states = shaperead('usastatelo', 'UseGeoCoords', true);
names = {states.Name};
indexHawaii = strcmp('Hawaii',names);
indexAlaska = strcmp('Alaska',names);
indexConus = 1:numel(states);
indexConus(indexHawaii|indexAlaska) = [];
stateColor = [0.5 1 0.5];
geoshow(ax(1), states(indexConus),  'FaceColor', stateColor)
geoshow(ax(2), states(indexAlaska), 'FaceColor', stateColor)
geoshow(ax(3), states(indexHawaii), 'FaceColor', stateColor)
for k = 1:3
    setm(ax(k), 'Frame', 'off', 'Grid', 'off',...
      'ParallelLabel', 'off', 'MeridianLabel', 'off')
end
```
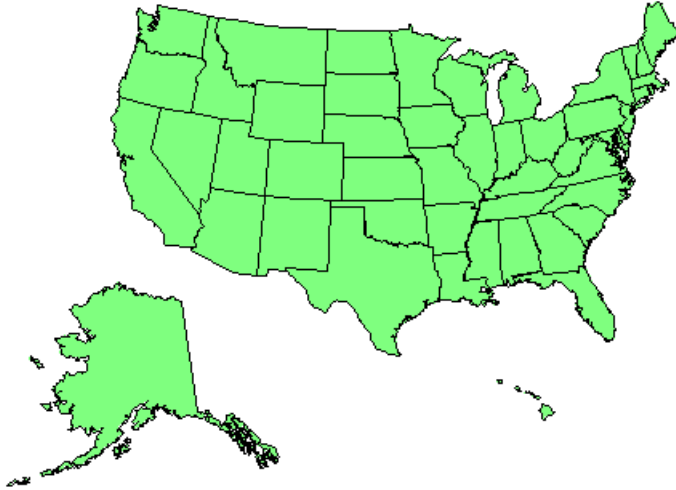
## See also

axesm, axesscale, `geoshow`, `paperscale`, `selectmoveresize`, `tightmap`, `worldmap`

## More About

**Tips**

In some cases, usamap uses `tightmap` to adjust the axis limits tight around the map. If you change the projection, or just want more white space around the map frame, use `tightmap` again or `axis auto`.

`axes(h(n))`, where n = 1, 2, or 3, makes the desired axes current.

`set(h,'Visible','on')` makes the axes visible.

`set(h,'ButtonDownFcn','selectmoveresize')` allows interactive repositioning of the axes. `set(h,'ButtonDownFcn','uimaptbx')` restores the Mapping Toolbox interfaces.

`axesscale(h(1))` resizes the axes containing Alaska and Hawaii to the same scale as the conterminous states.

# usgs24kdem

Read USGS 7.5-minute (30-m or 10-m) Digital Elevation Models

## Syntax

```
[lat,lon,Z] = usgs24kdem
[lat,lon,Z] = usgs24kdem(filename)
[lat,lon,Z] = usgs24kdem(filename,samplefactor)
[lat,lon,Z] = usgs24kdem(filename,samplefactor,latlim,lonlim)
[lat,lon,Z] = usgs24kdem(filename,samplefactor,latlim,lonlim,gsize)
[lat, lon, Z, header, profile] = usgs24kdem(...)
```

## Description

`[lat,lon,Z]` = `usgs24kdem` reads a USGS 1:24,000 digital elevation map (DEM) file in standard format. The file is selected interactively. The entire file is read and subsampled by a factor of 5. A geolocated data grid is returned with a latitude array, `lat`, longitude array, `lon`, and elevation array, `Z`. Horizontal units are in degrees, vertical units may vary. The 1:24,000 series of DEMs are stored as a grid of elevations spaced either at 10 or 30 meters apart. The number of points in a file will vary with the geographic location.

`[lat,lon,Z]` = `usgs24kdem(`*filename*`)` reads the USGS DEM specified by *filename* and returns the result as a geolocated data grid.

`[lat,lon,Z]` = `usgs24kdem(`*filename*`,samplefactor)` reads a subset of the DEM data from *filename*. `samplefactor` is a scalar integer, which when equal to 1 reads the data at its full resolution. When `samplefactor` is an integer `n` greater than one, every `n`th point is read. If `samplefactor` is omitted or empty, it defaults to `5`.

`[lat,lon,Z]` = `usgs24kdem(`*filename*`,samplefactor,latlim,lonlim)` reads a subset of the elevation data from *filename*. The limits of the desired data are specified as two-element vectors of latitude, `latlim`, and longitude, `lonlim`, in degrees. The elements of `latlim` and `lonlim` must be in ascending order. The data may extend somewhat outside the requested area. If limits are omitted, data for the entire area covered by the DEM file is returned.

`[lat,lon,Z] = usgs24kdem(`*`filename`*`,samplefactor,latlim,lonlim,gsize)` specifies the graticule size in `gsize`. `gsize` is a two-element vector specifying the number of rows and columns in the latitude and longitude coordinated grid. If omitted, a graticule the same size as the geolocated data grid is returned. Use empty matrices for `latlim` and `lonlim` to specify the coordinated grid size without specifying the geographic limits.

`[lat, lon, Z, header, profile] = usgs24kdem(...)` also returns the contents of the header and raw profiles of the DEM file. The `header` structure contains descriptions of the data from the file header. The `profile` structure is the raw profile data from which the geolocated data grid is constructed.

# Background

The U.S. Geological Survey has created a series of digital elevation models based on their paper 1:24,000 scale maps. The grid spacing for these elevations models is either 10 or 30 meters on a Universal Transverse Mercator grid. Each file covers a 7.5 minute quadrangle. The map and data series are available for much of the conterminous United States, Hawaii, and Puerto Rico. The data has been released in a number of formats. This function reads the data in the "standard" file format.

# Read USGS 24K DEM file

This example shows how to read a USGS 24K Digital Elevation Model file.

Unzip a USGS 24K DEM file. The toolbox includes a DEM file `sanfranciscos.dem.gz` .

```
filenames = gunzip('sanfranciscos.dem.gz', tempdir);
demFilename = filenames{1};
```

Read every other point of the 1:24,000 DEM file.

```
[lat,lon,Z,header,profile] = usgs24kdem(demFilename,2);
```

Delete the temporary gunzipped file.

```
delete(demFilename);
```

As no negative elevations exist, move all points at sea level to -1 to color them blue.

```
Z(Z==0) = -1;
```

Compute the latitude and longitude limits for the DEM.

```
latlim = [min(lat(:)) max(lat(:))]
lonlim = [min(lon(:)) max(lon(:))]
```

```
latlim =

   37.6249   37.7504


lonlim =

 -122.5008 -122.3740
```

Display the DEM values.

```
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, 'DisplayType','surface')
demcmap(Z)
daspectm('m',1)
```
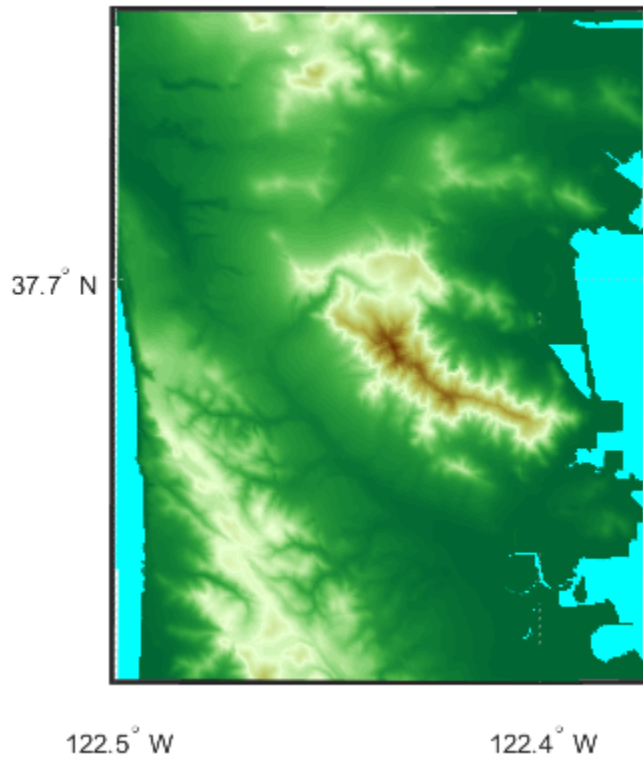
Examine the metadata in the header.

```
header
```

```
header =

              Quadranglename: 'SAN FRANCISCO SOUTH, CA  BIG BASIN DEM  '
                 TextualInfo: 'WMC                                CTOG     '
                      Filler: ''
                 ProcessCode: ''
                     Filler2: ''
           SectionalIndicator: ''
                 MCoriginCode: ''
```

```
                        DEMlevelCode: 2
                ElevationPatternCode: 'regular'
      PlanimetricReferenceSystemCode: 'UTM'
                                Zone: 10
                ProjectionParameters: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
                     HorizontalUnits: 'meters'
                       ElevationUnits: 'feet'
                  NsidesToBoundingBox: 4
                          BoundingBox: [1x8 double]
                      MinMaxElevations: [0 1314]
                        RotationAngle: 0
                         AccuracyCode: 'accuracy information in record C'
                        XYZresolutions: [30 30 1]
                            NrowsCols: [1 371]
                       MaxPcontourInt: NaN
                    SourceMaxCintUnits: 'unknown'
                       SmallestPrimary: NaN
                    SourceMinCintUnits: 'unknown'
                       DataSourceDate: NaN
                       DataInspRevDate: NaN
                           InspRevFlag: ''
                    DataValidationFlag: 'unknown'
                       SuspectVoidFlag: 'unknown'
                        VerticalDatum: 'unknown'
                      HorizontalDatum: 'unknown'
                          DataEdition: NaN
                           PercentVoid: NaN
```

## More About

### Tips

This function reads USGS DEM files stored in the UTM projection. The function
unprojects the grid back to latitude and longitude. Use `usgsdem` for data stored in
geographic grids.

The number of points in a file varies with the geographic location. Unlike the USGS
DEM products, which use an equal-angle grid, the UTM projection grid DEMs cannot
simply be concatenated to cover larger areas. There can be data gaps between DEMs.

You can obtain the data files from the U.S. Geological Survey and from commercial
vendors . Other agencies have made some local area data available online. See "Finding

Geospatial Data". The DEM files are ASCII files, and can be transferred as text. Line-ending conversion is not necessarily required.

## See Also

demdataui | dted | gtopo30 | tbase | etopo | usgsdem | usgsdems

# usgsdem

Read USGS 1-degree (3-arc-second) Digital Elevation Model

## Syntax

```
[Z,refvec] = usgsdem(filename,scalefactor)
[Z,refvec] = usgsdem(filename,scalefactor,latlim,lonlim)
```

## Description

`[Z,refvec] = usgsdem(filename,scalefactor)` reads the specified file and returns the data in a regular data grid along with referencing vector `refvec`, a 1-by-3 vector having elements `[cells/degree north-latitude west-longitude]` with latitude and longitude limits specified in degrees. The data can be read at full resolution (`scalefactor = 1`), or can be downsampled by the `scalefactor`. A `scalefactor` of 3 returns every third point, giving 1/3 of the full resolution.

`[Z,refvec] = usgsdem(filename,scalefactor,latlim,lonlim)` reads data within the latitude and longitude limits. These limits are two-element vectors with the minimum and maximum values specified in units of degrees.

## Background

The U.S. Geological Survey has made available a set of digital elevation maps of 1-degree quadrangles covering the contiguous United States, Hawaii, and limited portions of Alaska. The data is on a regular grid with a spacing of 30 arc-seconds (or about 100-meter resolution). 1-degree DEMs are also referred to as *3-arc-second* or *1:250,000 scale* DEM data.

The data is derived from the U.S. Defense Mapping Agency's DTED-1 digital elevation model, which itself was derived from cartographic and photographic sources. The cartographic sources were maps from the 7.5-minute through 1-degree series (1:24,000 scale through 1:250,000 scale).

# Examples

Read every fifth point in the file containing part of Rhode Island and Cape Cod:

```
[Z,refvec] = usgsdem('providence-e',5);
```

Read the elevation data for Martha's Vineyard at full resolution:

```
[Z,refvec] = usgsdem('providence-e',1,...
 [41.2952 41.4826],[-70.8429 -70.4392]);
whos Z
```

```
  Name        Size          Bytes  Class

  Z         226x485        876880  double array
```

# More About

**Tips**

The grid for the digital elevation maps is based on the 1984 World Geodetic System (WGS84). Older DEMs were based on WGS72. Elevations are in meters relative to National Geodetic Vertical Datum of 1929 (NGVD 29) in the continental U.S. and local mean sea level in Hawaii.

The absolute horizontal accuracy of the DEMs is 130 meters, while the absolute vertical accuracy is ±30 meters. The relative horizontal and vertical accuracy is not specified, but is probably much better than the absolute accuracy.

These DEMs have a grid spacing of 3 arc-seconds in both the latitude and longitude directions. The exception is DEM data in Alaska, where latitudes between 50 and 70 degrees North have grid spacings of 6 arc-seconds, and latitudes greater than 70 degrees North have grid spacings of 9 arc-seconds.

Statistical data in the files is not returned.

You can obtain the data files from the U.S. Geological Survey and from commercial vendors. Other agencies have made some local area data available online.

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

## See Also

`usgs24kdem` | `gtopo30` | `etopo` | `tbase` | `usgsdems`

# usgsdems

USGS 1-degree (3-arc-sec) DEM filenames for latitude-longitude quadrangle

## Syntax

```
[fname,qname] = usgsdems(latlim,lonlim)
```

## Description

`[fname,qname] = usgsdems(latlim,lonlim)` returns cell arrays of the DEM filenames and quadrangle names covering the geographic region. The region is specified by scalar latitude and longitude points or two-element vectors of latitude and longitude limits in units of degrees.

## Background

The U.S. Geological Survey has made available a set of digital elevation maps of 1-degree quadrangles covering the contiguous United States, Hawaii, and limited portions of Alaska. These are referred to as *1-degree*, *3-arc second* or *1:250,000 scale* DEMs. Because the filenames of these 1 degree data sets are taken from the names of cities or features in the quadrangle, determining the files needed to cover a particular region generally requires consulting an index map or other reference. This function takes the place of such a reference by returning the filenames for a given geographic region.

## Examples

Which files are needed to map part of New England?

```
usgsdems([41 44], [-72 -69])

ans =
    'providence-w'
    'providence-e'
    'chatham-w'
```

```
'boston-w'
'boston-e'
'portland-w'
'portland-e'
'bath-w'
```

## More About

### Tips

This function only returns filenames for the contiguous United States.

## See Also
usgsdem

# utmgeoid

Select ellipsoids for given UTM zone

## Syntax

```
ellipsoid = utmgeoid,
ellipsoid = utmgeoid(zone)
[ellipsoid,ellipsoidstr] = utmgeoid(...)
```

## Description

The purpose of this function is to recommend a local ellipsoid for use with a given
UTM zone, depending on the geographic location of that zone. Use it only if you are
not using a global reference ellipsoid, such as the World Geodetic System (WGS) 1984
ellipsoid. In many cases, depending on your application, you should just use the output of
`wgs84Ellipsoid`, or one of the other options available through referenceEllipsoid.

`ellipsoid = utmgeoid,` without any arguments, opens the `utmzoneui` interface
for selecting a UTM zone. This zone is then used to return the recommended ellipsoid
definitions for that particular zone.

`ellipsoid = utmgeoid(zone)` uses the input `zone` to return the recommended
ellipsoid definitions.

`[ellipsoid,ellipsoidstr] = utmgeoid(...)` returns the short name(s) for the
reference ellipsoid(s), as used by referenceEllipsoid, in a `char` array with one name in
each row.

## Background

The Universal Transverse Mercator (UTM) system of projections tiles the world into
quadrangles called zones. Each zone has different projection parameters and commonly
used ellipsoidal models of the Earth. This function returns a list of ellipsoid models
commonly used in a zone.

## Examples

```
zone = utmzone(0,100) % degrees

zone =
47N

[ellipsoid,names] = utmgeoid(zone)

ellipsoid =
        6377.3      0.081473
        6377.4      0.081697
names =
everest
bessel
```

## See Also

referenceEllipsoid | wgs84Ellipsoid

# utmzone

Select UTM zone given latitude and longitude

## Syntax

```
zone = utmzone
zone = utmzone(lat,long)
zone = utmzone(mat),
[latlim,lonlim] = utmzone(zone),
lim = utmzone(zone)
```

## Description

`zone = utmzone` selects a Universal Transverse Mercator (UTM) zone with a graphical user interface. The zone designation is returned as a string.

`zone = utmzone(lat,long)` returns the UTM zone containing the geographic coordinates. If `lat` and `long` are vectors, the zone containing the geographic mean of the data set is returned. The geographic coordinates must be in units of degrees.

`zone = utmzone(mat)`, where `mat` is of the form `[lat long]`.

`[latlim,lonlim] = utmzone(zone)`, where *zone* is a valid UTM zone designation, returns the geographic limits of the zone. Valid UTM zones designations are numbers, or numbers followed by a single letter. For example, `'31'` or `'31N'`. The returned limits are in units of degrees.

`lim = utmzone(zone)` returns the limits in a single vector output.

## Background

The Universal Transverse Mercator (UTM) system of projections tiles the world into quadrangles called zones. This function can be used to identify which zone is used for a geographic area and, conversely, what geographic limits apply to a UTM zone.

## Examples

```
[latlim,lonlim] = utmzone('12F')

latlim =
   -56    -48
lonlim =
  -114  -108

utmzone(latlim,lonlim)

ans =
12F
```

## Limitations

The UTM zone system is based on a regular division of the globe, with the exception of a few zones in northern Europe. utmzone does not account for these deviations.

## See Also

utmgeoid

# validateLengthUnit

Validate and standardize length unit string

## Syntax

```
standardName = validateLengthUnit(unit,funcName,varName,argIndex)
```

## Description

`standardName = validateLengthUnit(unit,funcName,varName,argIndex)` checks that `unit` is a valid length unit string and converts it to a standard unit name. The function is case-insensitive with respect to its input. Spaces, periods, and apostrophes are ignored. Plural forms are accepted in most cases, but the result, `standardName` is always singular. The optional inputs `funcName`, `varName`, and `argIndex` may be included for use in error message formatting, with behavior identical to that provided by the `validateattributes` inputs of the same names.

## Input Arguments

**unit**

String. Any valid length unit string listed in the table.

| Unit Name | String(s) |
|---|---|
| `meter` | 'm', 'meter(s)', 'metre(s)' |
| `centimeter` | 'cm', 'centimeter(s)', 'centimetre(s)' |
| `millimeter` | 'mm', 'millimeter(s)', 'millimetre(s)' |
| `micron` | 'micron(s)' |
| `kilometer` | 'km', 'kilometer(s)', 'kilometre(s)' |

| Unit Name | String(s) |
|---|---|
| nautical mile | 'nm', 'naut mi', 'nautical mile(s)' |
| foot | 'ft', 'international ft', 'foot', 'international foot', 'feet', 'international feet' |
| inch | 'in', 'inch', 'inches' |
| yard | 'yd', 'yds', 'yard(s)' |
| mile | 'mi', 'mile(s)', 'international mile(s)' |
| U.S. survey foot | 'sf', 'survey ft', 'US survey ft', 'U.S. survey ft', 'survey foot', 'US survey foot', 'U.S. survey foot', 'survey feet', 'US survey feet', 'U.S. survey feet' |
| U.S. survey mile (statute mile) | 'sm', 'survey mile(s)', 'statute mile(s)', 'US survey mile(s)', 'U.S. survey mile(s)' |
| Clarke's foot | 'Clarke's foot', 'Clarkes foot' |
| German legal metre | 'German legal metre', 'German legal meter' |
| Indian foot | 'Indian foot' |

**funcName**

String that specifies the name of the function whose input you are validating. If you specify an empty string, '', the validateLengthUnit function ignores the funcName input.

**varName**

String that specifies the name of the input variable. If you specify an empty string, '', the validateLengthUnit function ignores the varName input.

**argIndex**

Positive integer that specifies the position of the input argument.

## Output Arguments

**standardName**

String that specifies the standard unit name for the input string.

## Examples

### Validate 'foot'

This example shows you how `'foot'` is validated when other valid strings for `'foot'` are input.

```
validateLengthUnit('foot')
validateLengthUnit('feet')
validateLengthUnit('international feet')
```

ans =

foot


ans =

foot


ans =

foot

### Validate 'kilometer'

This example shows you how `'kilometer'` is validated when other valid strings for `'kilometer'` are input.

```
validateLengthUnit('kilometer')
```

```
    validateLengthUnit('km')
    validateLengthUnit('kilometre')
    validateLengthUnit('kilometers')
    validateLengthUnit('kilometres')

ans =

kilometer


ans =

kilometer


ans =

kilometer


ans =

kilometer


ans =

kilometer
```

### Create custom error messages

Create custom error messages using the `validateLengthUnit` function. A non-char input to `validateLengthUnit` results in an error message referencing a function name, `FOO`, a variable name, `'unit'` and an argument number, 5.

```
validateLengthUnit(17,'FOO','UNIT',5)
```

```
Error using FOO
Expected input number 5, UNIT, to be one of these types:

  char

Instead its type was double.

Error in validateLengthUnit (line 85)
```

```
validateattributes(unit,{'char'},{'nonempty','row'},varargin{:})
```

## See Also
unitsratio

# vec2mtx

Convert latitude-longitude vectors to regular data grid

## Syntax

```
[Z, R] = vec2mtx(lat, lon, density)
[Z, R] = vec2mtx(lat, lon, density, latlim, lonlim)
[Z, R] = vec2mtx(lat, lon, Z1, R1)
[Z, R] = vec2mtx(..., 'filled')
```

## Description

`[Z, R] = vec2mtx(lat, lon, density)` creates a regular data grid `Z` from vector data, placing ones in grid cells intersected by a vector and zeroes elsewhere. `R` is the referencing vector for the computed grid. `lat` and `lon` are vectors of equal length containing geographic locations in units of degrees. `density` indicates the number of grid cells per unit of latitude and longitude (a value of 10 indicates 10 cells per degree, for example), and must be scalar-valued. Whenever there is space, a buffer of two grid cells is included on each of the four sides of the grid. The buffer is reduced as needed to keep the latitudinal limits within [-90 90] and to keep the difference in longitude limits from exceeding 360 degrees.

`[Z, R] = vec2mtx(lat, lon, density, latlim, lonlim)` uses the two-element vectors `latlim` and `lonlim` to define the latitude and longitude limits of the grid.

`[Z, R] = vec2mtx(lat, lon, Z1, R1)` uses a pre-existing data grid `Z1`, georeferenced by `R1`, to define the limits and density of the output grid. `R1` can be a referencing vector, a referencing matrix, or a geographic raster reference object.

If `R1` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z1)` and its `RasterInterpretation` must be `'cells'`.

If `R1` is a referencing vector, it must be a 1-by-3 vector containing elements:

`[cells/degree northern_latitude_limit western_longitude_limit]`

or a 3-by-2 referencing matrix that transforms raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R1
```
If R1 is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. With this syntax, output R is equal to R1, and may be a referencing object, vector, or matrix.

`[Z, R] = vec2mtx(..., 'filled')`, where `lat` and `lon` form one or more closed polygons (with `NaN`-separators), fills the area outside the polygons with the value two instead of the value zero.

## Notes

Empty `lat,lon` vertex arrays will result in an error unless the grid limits are explicitly provided (via `latlim,lonlim` or `Z1,R1`). In the case of explicit limits, `Z` will be filled entirely with 0s if the `'filled'` parameter is omitted, and 2s if it is included.

It's possible to apply `vec2mtx` to sets of polygons that tile without overlap to cover an area, as in Example 1 below, but using `'filled'` with polygons that actually overlap may lead to confusion as to which areas are inside and which are outside.

## Examples

### Example 1

Convert latitude-longitude polygons to a regular data grid and display as a map.

```
states = shaperead('usastatelo', 'UseGeoCoords', true);
lat = [states.Lat];
lon = [states.Lon];
[Z, R] = vec2mtx(lat, lon, 5, 'filled');
figure; worldmap(Z, R);
geoshow(Z, R, 'DisplayType', 'texturemap')
colormap(flag(3))
```
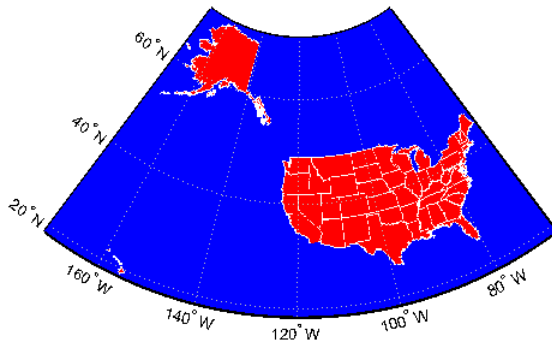
## Example 2

Combine two separate calls to vec2mtx to create a 4-color raster map showing interior land areas, coastlines, oceans, and world rivers.

```
load coastlines
[Z, R] = vec2mtx(coastlat, coastlon, ...
    1, [-90 90], [-90 270], 'filled');
rivers = shaperead('worldrivers.shp','UseGeoCoords',true);
A = vec2mtx([rivers.Lat], [rivers.Lon], Z, R);
Z(A == 1) = 3;
figure; worldmap(Z, R)
geoshow(Z, R, 'DisplayType', 'texturemap')
colormap([.45 .60 .30; 0 0 0; 0 0.5 1; 0 0 1])
```
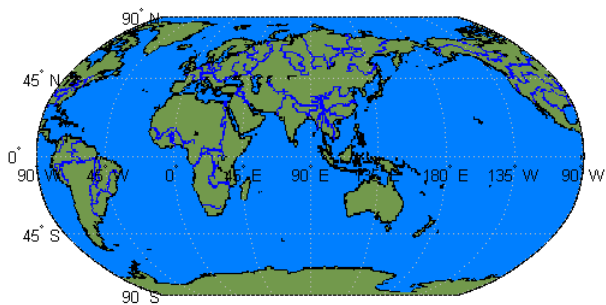
# Example 3

This example illustrates the following syntax in the case where R1 is a spatial referencing object:

```
[Z, R] = vec2mtx(lat, lon, Z1, R1)

% Import US state outlines.
states = shaperead('usastatelo', 'UseGeoCoords', true);
lat = [states.Lat];
lon = [states.Lon];

% Choose geographic limits.
latlim = [  15  75];
lonlim = [-190 -65];

% Specify a grid with 5 cells per degree.
density = 5;

% Compute raster size. (M and N both work out to be integers.)
M = density * diff(latlim);
N = density * diff(lonlim);

% Construct a spatialref.GeoRasterReference object.
R = georasterref('RasterSize', [M N], ...
    'ColumnsStartFrom', 'north', 'Latlim', latlim, ...
  'Lonlim', lonlim);

% Create a blank grid that is consistent with R in
% size -- vec2mtx requires a data grid as input.
Z = zeros(R.RasterSize);

% Overwrite Z with a new grid including state outlines
% and interiors.
Z = vec2mtx(lat, lon, Z, R, 'filled');

% Plot the georeferenced grid.
figure; worldmap(Z, R);
geoshow(Z, R, 'DisplayType', 'texturemap')
colormap(flag(3))
```
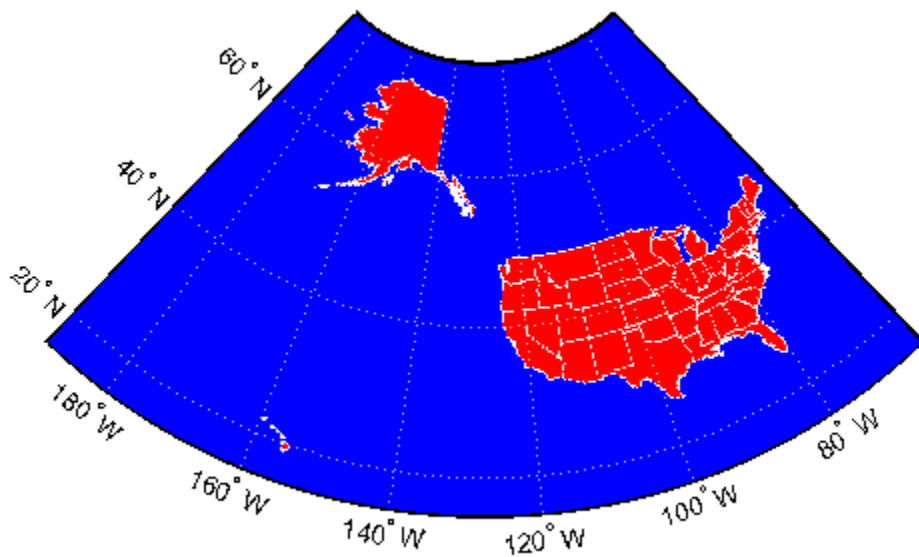
## See Also

imbedm

# vfwdtran

Direction angle in map plane from azimuth on ellipsoid

## Syntax

```
th = vfwdtran(lat,lon,az)
th = vfwdtran(mstruct,lat,lon,az)
[th,len] = vfwdtran(...)
```

## Description

`th = vfwdtran(lat,lon,az)` transforms the azimuth angle at specified latitude and longitude points on the sphere into the projection space. The map projection currently displayed is used to define the projection space. The input angles must be in the same units as specified by the current map projection. The inputs can be scalars or matrices of the equal size. The angle in the projection space is defined as positive counterclockwise from the *x*-axis.

`th = vfwdtran(mstruct,lat,lon,az)` uses the map projection defined by the input `mstruct` to compute the map projection.

`[th,len] = vfwdtran(...)` also returns the vector length in the projected coordinate system. A value of 1 indicates no scale distortion.

## Background

The direction of north is easy to define on the three-dimensional sphere, but more difficult on a two-dimensional map. For cylindrical projections in the normal aspect, north is always in the positive *y*-direction. For conic projections, north can be to the left or right of the *y*-axis. This function transforms any azimuth angle on the sphere to the corresponding angle in the projected paper coordinates.

## Examples

Sample calculations:

```
axesm('eqdconic','maplatlim',[-10 45],'maplonlim',[-55 55])
gridm; framem; mlabel; plabel
quiverm([0 0 0],[-45 0 45],[0 0 0],[10 10 10],0)
quiverm([0 0 0],[-45 0 45],[10 10 10],[0 0 0],0)
```
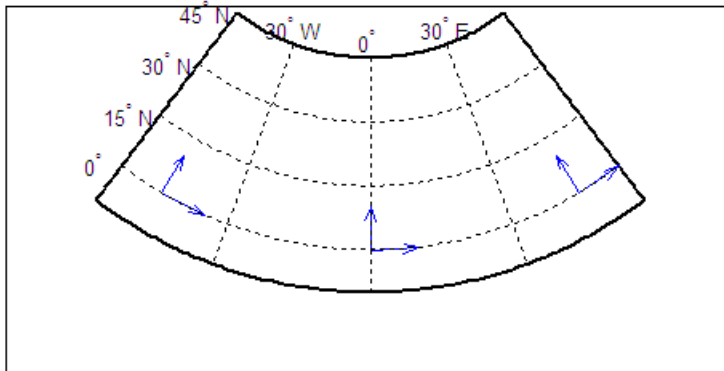


```
vfwdtran([0 0 0],[-45 0 45],[0 0 0])

ans =
      59.614              90         120.39

vfwdtran([0 0 0],[-45 0 45],[90 90 90])

ans =
      -30.385     0.0001931         30.386
```

## Limitations

This transformation is limited to the region specified by the frame limits in the current map definition.

## More About

**Tips**

The geographic azimuth angle is measured clockwise from north. The projection space angle is measured counterclockwise from the *x*-axis.

This function uses a finite difference technique. The geographic coordinates are perturbed slightly in different directions and projected. A small amount of error is introduced by numerical computation of derivatives and the variation of map distortion parameters.

## See Also

vinvtran | mfwdtran | minvtran | defaultm

# viewshed

Areas visible from point on terrain elevation grid

## Syntax

```
[vis,R] = viewshed(Z,R,lat1,lon1)
viewshed(Z,R,lat1,lon1,observerAltitude)
viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude)
viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
    observerAltitudeOption)
viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
    observerAltidueOption,targetAltitudeOption)
viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
    observerAltitudeOption,targetAltitudeOption,actualRadius)
viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
    observerAltitudeOption,targetAltitudeOption, ...
    actualRadius,effectiveRadius)
```

## Description

`[vis,R] = viewshed(Z,R,lat1,lon1)` computes areas visible from a point on a digital elevation grid. `Z` is a regular data grid containing elevations in units of meters. The observer location is provided as scalar latitude and longitude in units of degrees. The visibility grid `vis` contains `1`s at the surface locations visible from the observer location, and `0`s where the line of sight is obscured by terrain. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

```
[cells/degree northern_latitude_limit western_longitude_limit]
```

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to or from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If R is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel. Nearest-neighbor interpolation is used by default. NaN is returned for points outside the grid limits or for which lat or lon contain NaN. All angles are in units of degrees.

viewshed(Z,R,lat1,lon1,observerAltitude) places the observer at the specified altitude in meters above the surface. This is equivalent to putting the observer on a tower. If omitted, the observer is assumed to be on the surface.

viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude) checks for visibility of target points a specified distance above the terrain. This is equivalent to putting the target points on towers that do not obstruct the view. if omitted, the target points are assumed to be on the surface.

viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
  observerAltitudeOption) controls whether the observer is at a relative or absolute altitude. If the observerAltitudeOption is 'AGL', then observerAltitude is in meters above ground level. If observerAltitudeOption is 'MSL', observerAltitude is interpreted as altitude above zero, or mean sea level. If omitted, 'AGL' is assumed.

viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
  observerAltidueOption,targetAltitudeOption) controls whether the target points are at a relative or absolute altitude. If the target altitude option is 'AGL', the targetAltitude is in meters above ground level. If targetAltitudeOption is 'MSL', targetAltitude is interpreted as altitude above zero, or mean sea level. If omitted, 'AGL' is assumed.

viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
  observerAltitudeOption,targetAltitudeOption,actualRadius) does the visibility calculation on a sphere with the specified radius. If omitted, the radius of the earth in meters is assumed. The altitudes, the elevations, and the radius should be in the same units. This calling form is most useful for computations on bodies other than the Earth.

viewshed(Z,R,lat1,lon1,observerAltitude,targetAltitude, ...
  observerAltitudeOption,targetAltitudeOption, ...
  actualRadius,effectiveRadius) assumes a larger radius for propagation of the line of sight. This can account for the curvature of the signal path due to refraction in the

atmosphere. For example, radio propagation in the atmosphere is commonly treated as straight line propagation on a sphere with 4/3 the radius of the Earth. In that case the last two arguments would be `R_e` and `4/3*R_e`, where `R_e` is the radius of the earth. Use `Inf` for flat Earth `viewshed` calculations. The altitudes, the elevations, and the radii should be in the same units.

# Examples

## Compute Visibility for Point on Peaks Map

Compute visibility for a point on the peaks map. Add the detailed information for the line of sight calculation between two points from `los2` .

Create the peak map and plot the visibility.

```
Z = 500*peaks(100);
refvec = [1000 0 0];
[lat1,lon1,lat2,lon2] = deal(-0.027,0.05,-0.093,0.042);

[visgrid,visleg] = viewshed(Z,refvec,lat1,lon1,100);
[vis,visprofile,dist,zi,lattrk,lontrk] ...
   = los2(Z,refvec,lat1,lon1,lat2,lon2,100);

axesm('globe','geoid',earthRadius('meters'))
meshm(visgrid,visleg,size(Z),Z);
axis tight
camposm(-10,-10,1e6)
camupm(0,0)
colormap(flipud(summer(2)))
brighten(0.75)
shading interp
camlight
h = lcolorbar({'obscured','visible'});
h.Position = [.875 .45 .02 .1];

plot3m(lattrk([1;end]),lontrk([1; end]), ...
   zi([1; end])+[100; 0],'r','linewidth',2)
plotm(lattrk(~visprofile),lontrk(~visprofile), ...
   zi(~visprofile),'r.','markersize',10)
plotm(lattrk(visprofile),lontrk(visprofile), ...
   zi(visprofile),'g.','markersize',10)
```
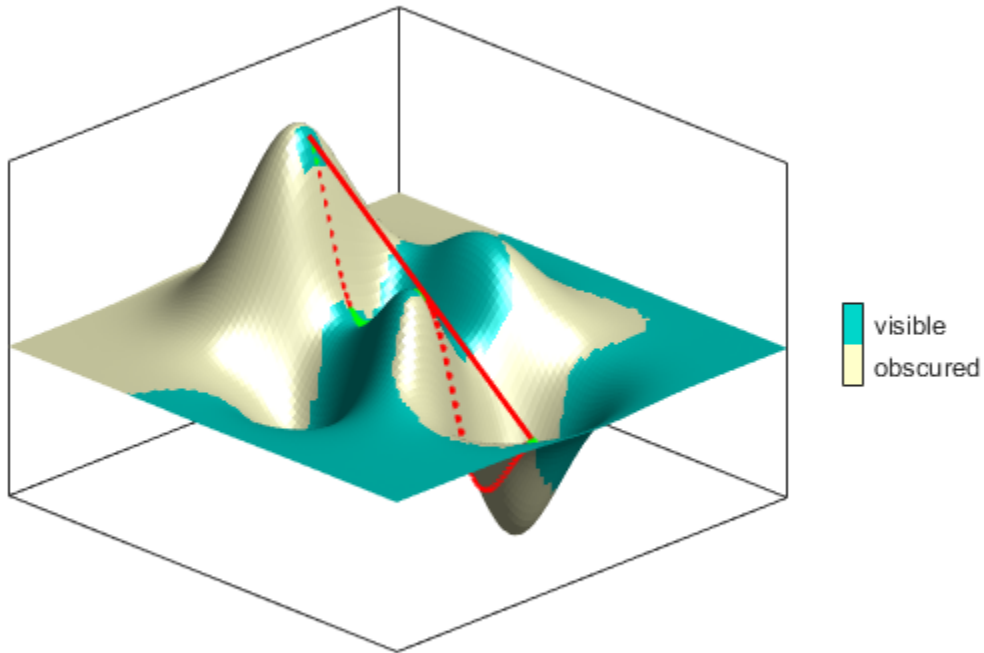
## Compute Surface Area Visible By Radar

Compute the surface area visible by radar from an aircraft flying 3000 meters above the Yellow Sea. Assume that radio wave propagation in the atmosphere can be modeled as straight lines on a 4/3 radius Earth. This example also calculates the area visible to the plane above it at 5000 meters.

Load data from the MAT-file containing information about the Korean peninsula, calculate the viewshed, and display it.

```
load korea
map(map<0) = -1;
figure
```

```
worldmap(map,refvec)
setm(gca,'geoid',[1 0])
da = daspect;
pba = pbaspect;
da(3) = 7.5*pba(3)/da(3);
daspect(da);
demcmap(map)
camlight(90,5);
camlight(0,5);
lighting gouraud
material([0.25 0.8 0])
lat = 34.0931; lon = 125.6578;
altobs = 3000; alttarg = 0;
plotm(lat,lon,'wo')
Re = earthRadius('meters');
[vmap,vmapl] = viewshed( ...
    map,refvec,lat,lon,altobs,alttarg, ...
    'MSL','AGL',Re,4/3*Re);
meshm(vmap,vmapl,size(map),map)
```
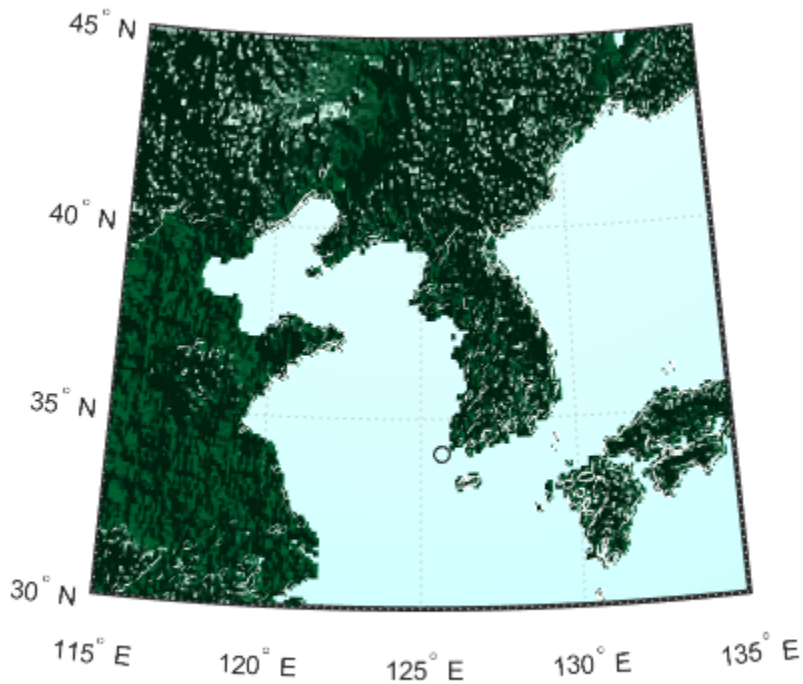
Display the visible areas as blue and the obscured areas as red and drape the visibility colors on an elevation map, using lighting to bring out the surface topography. The aircraft's radar can see out a certain radius on the surface of the ocean, but note how some ocean areas are shadowed by the island of Jeju-Do. Also some mountain valleys closer than the ocean horizon are obscured, while some mountain tops further away are visible.

```
caxis auto; colormap([1 0 0; 0 0 1])
lighting gouraud;
axis off
```
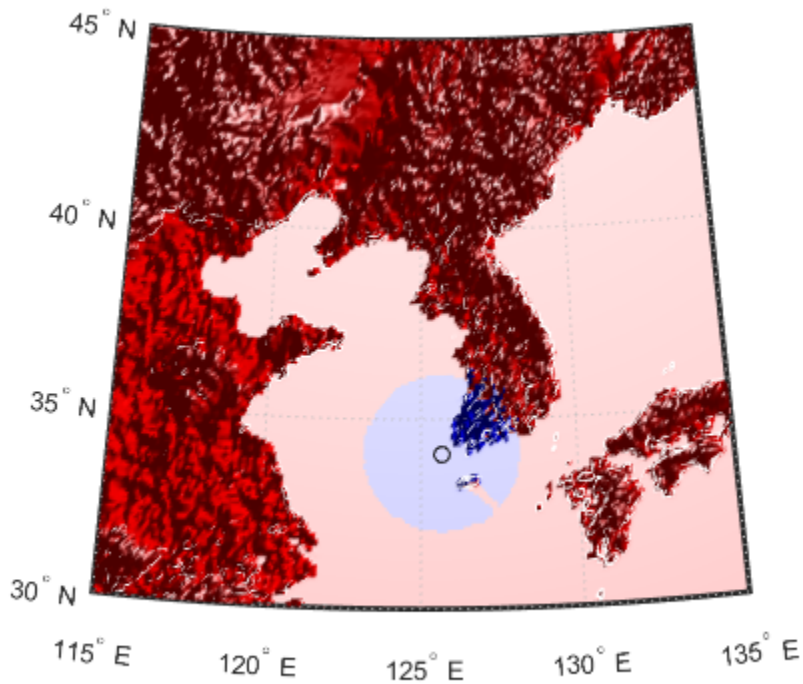
Now calculate the area that the radar plane flying at an altitude of 3000 meters can have line-of-sight to other aircraft flying above it at 5000 meters. Note how the area is much larger but that some edges of the area are reduced by shadowing from Jeju-Do and the mountains on the Korean peninsula.

```
[vmap,vmapl] = viewshed(map,refvec,lat,lon,3000,5000, ...
               'MSL','MSL',Re,4/3*Re);
clmo surface
meshm(vmap,vmapl,size(map),map)
lighting gouraud
```
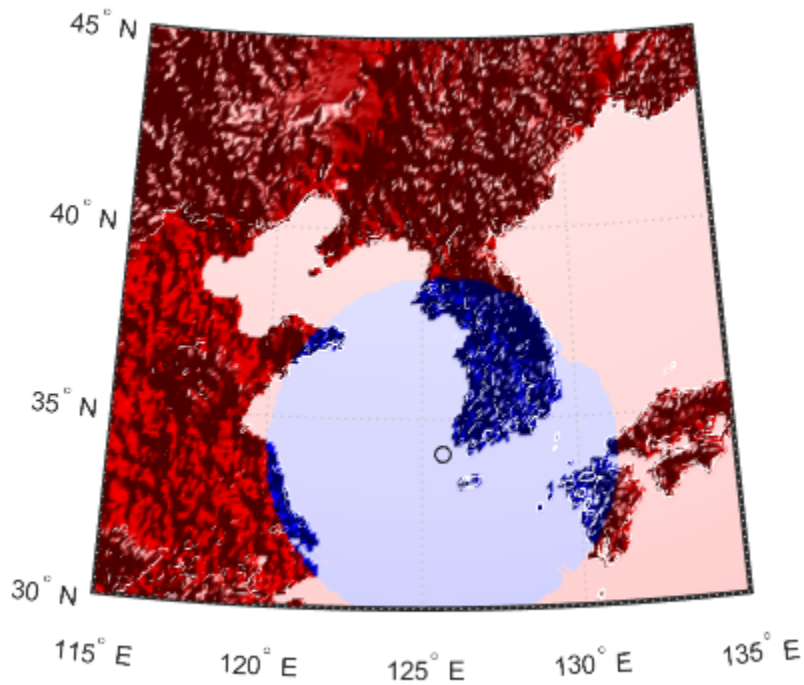
## More About

### Tips

The observer should be located within the latitude-longitude limits of the elevation grid. If the observer is located outside the grid, there is insufficient information to calculate a viewshed. In this case `viewshed` issues a warning and sets all elements of `vis` to zero.

### See Also
los2

# vinvtran

Azimuth on ellipsoid from direction angle in map plane

## Syntax

```
az = vinvtran(x,y,th)
az = vinvtran(mstruct,x,y,th)
[az,len] = vinvtran(...)
```

## Description

`az = vinvtran(x,y,th)` transforms an angle in the projection space at the point specified by x and y into an azimuth angle in geographic coordinates. The map projection currently displayed is used to define the projection space. The input angles must be in the same units as specified by the current map projection. The inputs can be scalars or matrices of equal size. The angle in the projection space angle th is defined as positive counterclockwise from the *x*-axis.

`az = vinvtran(mstruct,x,y,th)` uses the map projection defined by the input struct to compute the map projection.

`[az,len] = vinvtran(...)` also returns the vector length in the geographic coordinate system. A value of 1 indicates no scale distortion for that angle.

## Background

While vectors along the *y*-axis always point to north in a cylindrical projection in the normal aspect, they can point east or west of north on conics, azimuthals, and other projections. This function computes the geographic azimuth for angles in the projected space.

## Examples

Sample calculations:

```
axesm('eqdconic','maplatlim',[-10 45],'maplonlim',[-55 55])
gridm; framem; mlabel; plabel
[x,y] = mfwdtran([0 0 0],[-45 0 45]);
quiver(x,y,[ .2 .2 .2],[0 0 0],0)
quiver(x,y,[0 0 0],[ .2 .2 .2],0)
```
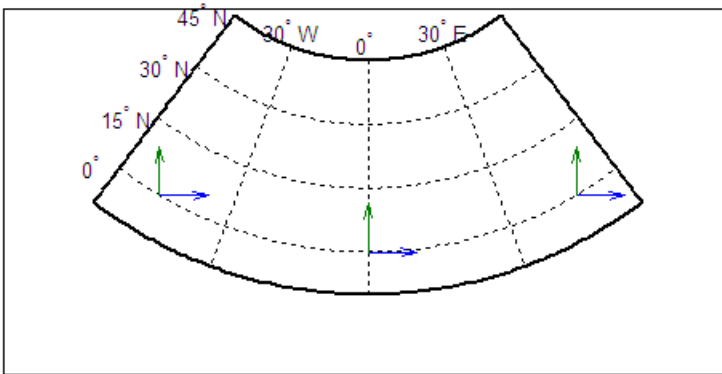


```
vinvtran(x,y,[ 0 0 0])

ans =
        57.345          90.338          124.98

vinvtran(x,y,[ 90 90 90])

ans =
        331.99              0          28.008
```

## Limitations

This transformation is limited to the region specified by the frame limits in the current map definition.

## More About

**Tips**

The geographic azimuth angle is measured clockwise from north. The projection space angle is measured counterclockwise from the $x$-axis.

This function uses a finite difference technique. The geographic coordinates are perturbed slightly in different directions and projected. A small amount of error is introduced by numerical computation of derivatives and the variation of map distortion parameters.

## See Also
vfwdtran | mfwdtran | minvtran | defaultm

# vmap0data

Read selected data from Vector Map Level 0

## Syntax

```
struct = vmap0data(library,latlim,lonlim,theme,topolevel)
struct = vmap0data(devicename,library, ...)
[struct1, struct2, ...] = vmap0data(...,{topolevel1,topolevel2,...})
```

## Description

`struct = vmap0data(`*`library`*`,latlim,lonlim,`*`theme`*`,`*`topolevel`*`)` reads the data for the specified theme and topology level directly from the VMAP0 CD-ROM. There are four CDs, one for each of the libraries: `'NOAMER'` (North America), `'SASAUS'` (Southern Asia and Australia), `'EURNASIA'` (Europe and Northern Asia), and `'SOAMAFR'` (South America and Africa). The desired *theme* is specified by a two-letter code string. A list of valid codes is displayed when an invalid code, such as `'?'`, is entered. *topolevel* defines the type of data returned. It is a string containing `'patch'`, `'line'`, `'point'`, or `'text'`. The region of interest can be given as a point latitude and longitude or as a region with two-element vectors of latitude and longitude limits. The units of latitude and longitude are degrees. The data covering the requested region is returned, but will include data extending to the edges of the tiles. The result is returned as a Mapping Toolbox Version 1 display structure.

`struct = vmap0data(`*`devicename`*`,`*`library`*`, ...)` specifies the logical device name of the CD-ROM for computers that do not automatically name the mounted disk.

`[struct1, struct2, ...] = vmap0data(...,{`*`topolevel1`*`,`*`topolevel2`*`,...})` reads several topology levels. The levels must be specified as a cell array with the entries `'patch'`, `'line'`, `'point'`, or `'text'`. Entering `{'all'}` for the topology level argument is equivalent to `{'patch'`, `'line'`, `'point'`, `'text'}`. Upon output, the data structures are returned in the output arguments by topology level in the same order as they were requested.

## Background

The Vector Map (VMAP) Level 0 database represents the third edition of the *Digital Chart of the World*. The second edition was a limited release item published in 1995. The product is dual named to show its lineage to the original DCW, published in 1992, while positioning the revised product within a broader emerging family of VMAP products. VMAP Level 0 is a comprehensive 1:1,000,000 scale vector base map of the world. It consists of cartographic, attribute, and textual data stored on compact disc read-only memory (CD-ROM). The primary source for the database is the Operational Navigation Chart (ONC) series of the U. S. National Geospatial Intelligence Agency (NGA), formerly the National Imagery and Mapping Agency (NIMA), and before that, the Defense Mapping Agency (DMA). This is the largest scale unclassified map series in existence that provides consistent, continuous global coverage of essential base map features. The database contains more than 1,900 MB of vector data and is organized into 10 thematic layers. The data includes major road and rail networks, major hydrologic drainage systems, major utility networks (cross-country pipelines and communication lines), all major airports, elevation contours (1000 foot (ft), with 500 ft and 250 ft supplemental contours), coastlines, international boundaries, and populated places. The database can be accessed directly from the four optical CD-ROMs that store the database or can be transferred to magnetic media.

## Examples

The *devicename* is platform dependent. On an MS-DOS based operating system it would be something like `'d:'`, depending on the logical device code assigned to the CD-ROM drive. On a UNIX operating system, the CD-ROM might be mounted as `'\cdrom'`, `'\CDROM'`, `'\cdrom1'`, or something similar. Check your computer's documentation for the right *devicename*.

```
s = vmap0data(devicename,'NOAMER',41,-69,'?','patch');

??? Error using ==> vmap0data
Theme not present in library NOAMER

Valid theme identifiers are:
libref : Library Reference
tileref: Tile Reference
bnd    : Boundaries
dq     : Data Quality
```

```
elev   : Elevation
hydro  : Hydrography
ind    : Industry
phys   : Physiography
pop    : Population
trans  : Transportation
util   : Utilities
veg    : Vegetation

BNDpatch = vmap0data(devicename,'NOAMER',...
                    [41 44],[-72 -69],'bnd','patch')
BNDpatch =
1x169 struct array with fields:
    type
    otherproperty
    altitude
    lat
    long
    tag
```

Here are other examples:

```
[TRtext,TRline] = vmap0data(devicename,'SASAUS',...
    [-48 -34],[164 180],'trans',{'text','line'});

[BNDpatch,BNDline,BNDpoint,BNDtext] = vmap0data(devicename,...
    'EURNASIA',-48 ,164,'bnd',{'all'});
```

# More About

## Tips

Data are returned as Mapping Toolbox display structures, which you can then update to geographic data structures. For information about display structure format, see "Version 1 Display Structures" on page 1-200 in the reference page for `displaym`. The `updategeostruct` function performs such conversions.

Latitudes and longitudes use WGS84 as a horizontal datum. Elevations and depths are in meters above mean sea level.

Some VMAP0 themes do not contain all topology levels. In those cases, empty matrices are returned.

Patches are broken at the tile boundaries. Setting the `EdgeColor` to `'none'` and plotting the lines gives the map a normal appearance.

The major differences between VMAP0 and the DCW are the elimination of the gazette layer, addition of bathymetric data, and updated political boundaries.

Vector Map Level 0, created in the 1990s, is still probably the most detailed global database of vector map data available to the public. VMAP0 CD-ROMs are available from through the U.S. Geological Survey (USGS):

USGS Information Services (Map and Book Sales)
Box 25286
Denver Federal Center
Denver, CO 80225
Telephone: (303) 202-4700
Fax: (303) 202-4693

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: "Finding Geospatial Data".

---

## See Also

`vmap0read` | `vmap0rhead` | `geoshow` | `extractm` | `mlayers` | `updategeostruct`

# vmap0read

Read Vector Map Level 0 file

## Syntax

```
vmap0read
vmap0read(filepath,filename)
vmap0read(filepath,filename,recordIDs)
vmap0read(filepath,filename,recordIDs,field,varlen)
struc = vmap0read(...)
[struc,field] = vmap0read(...)
[struc,field,varlen] = vmap0read(...)
[struc,field,varlen,description] = vmap0read(...)
[struc,field,varlen,description,narrativefield] = vmap0read(...)
```

## Description

`vmap0read` reads a VMAP0 file. The user selects the file interactively.

`vmap0read(filepath,filename)` reads the specified file. The combination `[filepath filename]` must form a valid complete filename.

`vmap0read(filepath,filename,recordIDs)` reads selected records or fields from the file. If `recordIDs` is a scalar or a vector of integers, the function returns the selected records. If `recordIDs` is a cell array of integers, all records of the associated fields are returned.

`vmap0read(filepath,filename,recordIDs,field,varlen)` uses previously read field and variable-length record information to skip parsing the file header (see below).

`struc = vmap0read(...)` returns the file contents in a structure.

`[struc,field] = vmap0read(...)` returns the file contents and a structure describing the format of the file.

`[struc,field,varlen] = vmap0read(...)` also returns a vector describing which fields have variable-length records.

[struc,field,varlen,description] = vmap0read(...) also returns a string describing the contents of the file.

[struc,field,varlen,description,narrativefield] = vmap0read(...) also returns the name of the narrative file for the current file.

## Background

The Vector Map Level 0 (VMAP0) uses binary files in a variety of formats. This function determines the format of the file and returns the contents in a structure. The field names of this structure are the same as the field names in the VMAP0 file.

## Examples

The following examples use the UNIX directory system and file separators for the pathname:

```
s = vmap0read('VMAP/VMAPLV0/NOAMER/','GRT')

s =

                   id: 1
            data_type: 'GEO'
                units: 'M'
       ellipsoid_name: 'WGS 84'
     ellipsoid_detail: 'A=6378137 B=6356752 Meters'
       vert_datum_name: 'MEAN SEA LEVEL'
       vert_datum_code: 'O15'
      sound_datum_name: 'N/A'
      sound_datum_code: 'N/A'
        geo_datum_name: 'WGS 84'
        geo_datum_code: 'WGE'
       projection_name: 'Dec. Deg. (unproj.)'

s = vmap0read('VMAP/VMAPLV0/NOAMER/TRANS/','INT.VDT')

s =
34x1 struct array with fields:
    id
    table
    attribute
```

```
        value
        description

s(1)

ans =
               id: 1
            table: 'aerofacp.pft'
        attribute: 'use'
            value: 8
      description: 'Military'
s = vmap0read('VMAP/VMAPLV0/NOAMER/TRANS/','AEROFACP.PFT',1)

s =
          id: 1
      f_code: 'GB005'
         iko: 'BGTL'
         nam: 'THULE AIR BASE'
         na3: 'GL52085'
         use: 8
         zv3: 77
     tile_id: 10
      end_id: 1

s = vmap0read('VMAP/VMAPLV0/NOAMER/TRANS/','AEROFACP.PFT',{1,2})

s =
1x4424 struct array with fields:
    id
    f_code
```

# More About

### Tips

This function reads all VMAP0 files except index files (files with names ending in `'X'`), thematic index files (files with names ending in `'TI'`), and spatial index files (files with names ending in `'SI'`).

File separators are platform dependent. The *filepath* input must use appropriate file separators, which you can determine using the MATLAB `filesep` function.

## See Also

`vmap0data` | `vmap0rhead`

# vmap0rhead

Read Vector Map Level 0 file headers

## Syntax

```
vmap0rhead
vmap0rhead(filepath,filename)
vmap0rhead(filepath,filename,fid)
vmap0rhead(...),
str = vmap0rhead(...)
```

## Description

`vmap0rhead` allows the user to select the header file interactively.

`vmap0rhead(filepath,filename)` reads from the specified file. The combination [*filepath filename*] must form a valid complete *filename*.

`vmap0rhead(filepath,filename,fid)` reads from the already open file associated with `fid`.

`vmap0rhead(...),` with no output arguments, displays the formatted header information on the screen.

`str = vmap0rhead(...)` returns a string containing the VMAP0 header.

## Background

The Vector Map Level 0 (VMAP0) uses header strings in most files to document the contents and format of that file. This function reads the header string and displays a formatted version in the Command Window, or returns it as a string.

## Examples

The following example uses UNIX file separators and pathname:

```
s = vmap0rhead('VMAP/VMAPLV0/NOAMER/','GRT')
s =
L;Geographic Reference Table;-;id=I,1,P,Row
Identifier,-,-,-,:data_type=T,3,N,Data
Type,-,-,-,:units=T,3,N,Units of Measure Code for
Library,-,-,-,:ellipsoid_name=T,15,N,Ellipsoid,-,-,-,:ellipsoid
_detail=T,50,N,Ellipsoid
Details,-,-,-,:vert_datum_name=T,15,N,Datum Vertical
Reference,-,-,-,:vert_datum_code=T,3,N,Vertical Datum
Code,-,-,-,:sound_datum_name=T,15,N,Sounding
Datum,-,-,-,:sound_datum_code=T,3,N,Sounding Datum
Code,-,-,-,:geo_datum_name=T,15,N,Datum Geodetic
Name,-,-,-,:geo_datum_code=T,3,N,Datum Geodetic
Code,-,-,-,:projection_name=T,20,N,Projection Name,-,-,-,:;

vmap0rhead('VMAP/VMAPLV0/NOAMER/TRANS/','AEROFACP.PFT')
L
Airport Point Feature Table
aerofacp.doc
id=I,1,P,Row Identifier,-,-,-,
f_code=T,5,N,FACC Feature Code,char.vdt,-,-,
iko=T,4,N,ICAO Designator,char.vdt,-,-,
nam=T,*,N,Name,char.vdt,-,-,
na3=T,*,N,Name,char.vdt,-,-,
use=S,1,N,Usage,int.vdt,-,-,
zv3=S,1,N,Airfield/Aerodrome Elevation (meters),int.vdt,-,-,
tile_id=S,1,N,Tile Reference ID,-,tile1_id.pti,-,
end_id=I,1,N,Entity Node Primitive ID,-,end1_id.pti,-,
```

## More About

### Tips

This function reads all VMAP0 files except index files (files with names ending in `'X'`), thematic index files (files with names ending in `'TI'`) and spatial index files (files with names ending in `'SI'`).

File separators are platform dependent. The *filepath* input must use appropriate file separators, which you can determine using the MATLAB `filesep` function.

### See Also
vmap0data | vmap0read

# webmap

Open web map

## Syntax

```
webmap
webmap(baseLayer)
webmap( ___ ,Name,Value)
wm = webmap( ___ )
webmap(wm)
```

## Description

webmap opens a new web map in a browser, centering the map at the latitude, longitude point [0 0]. By default, webmap sets the base layer to World Street Map at the maximum spatial extent available. After the web map opens, you can select a different base layer using the Layer Manager available on the right side of the web map. Named base layers are tiled at discrete zoom resolutions.

---

**Note:** The webmap function requires an Internet connection. MathWorks cannot guarantee the stability and accuracy of web maps, as the servers are located on the Internet and are independent from MathWorks. Occasionally, maps may be slow to display, display partially, or fail to display, because web map servers can become unavailable for short periods of time.

---

webmap(baseLayer) opens a new web map with base layer specified by baseLayer. baseLayer can be one of a list of named Web Map Service providers or a WMSLayer array. If baseLayer is a matrix of WMSLayer objects, webmap uses each object's LayerTitle property as the value of items in the Layer Manager.

webmap displays named base layers in the Web Mercator map coordinate reference system. webmap displays WMSLayers in the Web Mercator map coordinate reference system, if that projection (EPSG:900913) is available for all layers in the array. Otherwise, webmap displays WMSLayers in the EPSG:4326 geographic coordinate

reference system. When displaying a WMSLayer in the EPSG:4326 geographic coordinate reference system, `webmap` does not include the named base layers in the Layer Manager, because they are in a different coordinate reference system. These projections include a geographic quadrangle bounded north and south by parallels (which map to horizontal lines) and east and west by meridians (which map to vertical lines).

`webmap( ___ ,Name,Value)` opens a new web map with additional display properties set.

`wm = webmap( ___ )` returns a handle to a web map, `wm`.

`webmap(wm)` makes the web map specified by `wm` the current web map.

# Examples

### Display a Web map

Open a web map centered at `[0 0]`. In the illustration, the Layer Manager is visible.

```
webmap
```
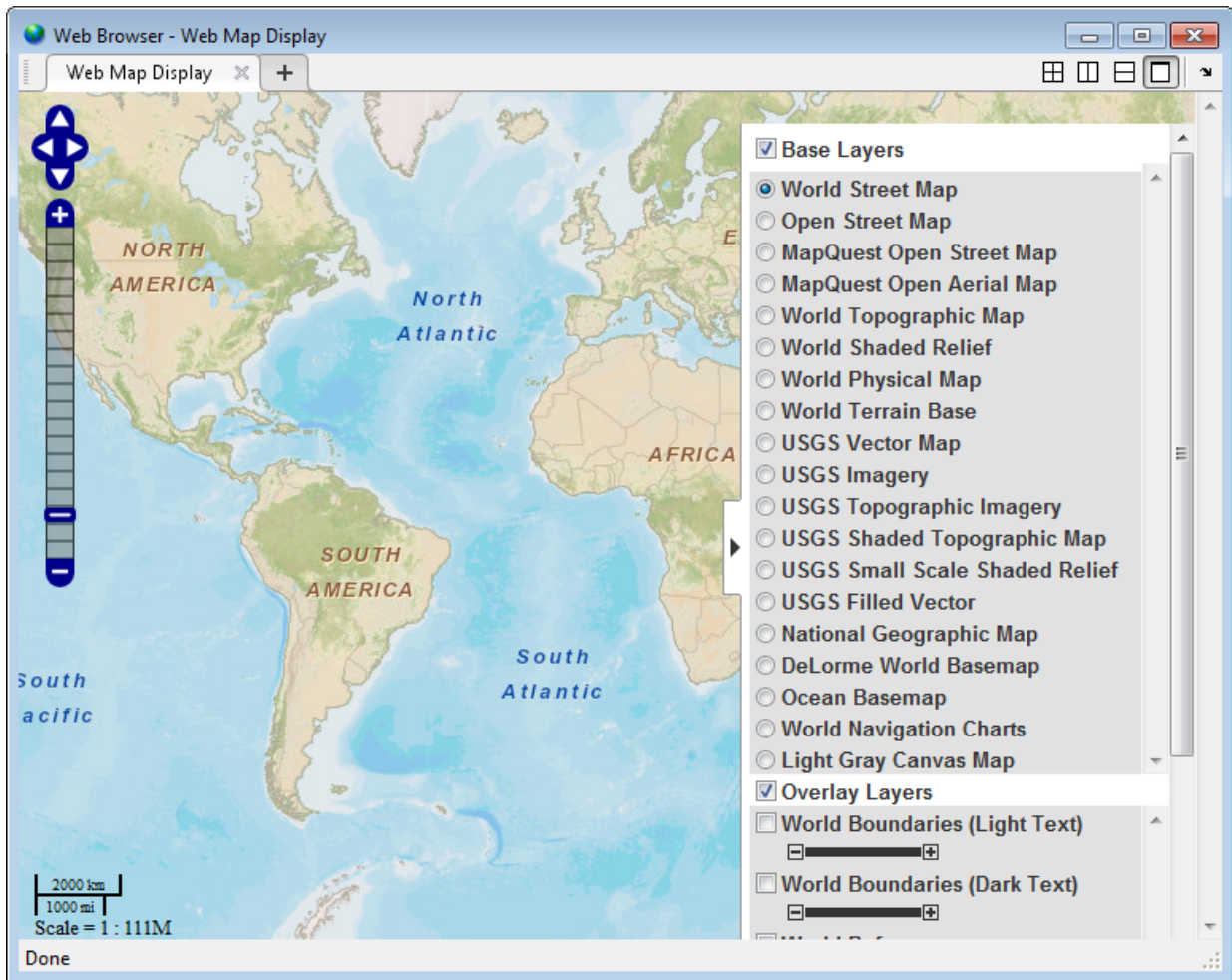
### Display a Web Map Specifying the Base Layer

Open a web map specifying the base layer and show the full extent of the world.

```
webmap('Light Gray Canvas Map', 'WrapAround', false)
```
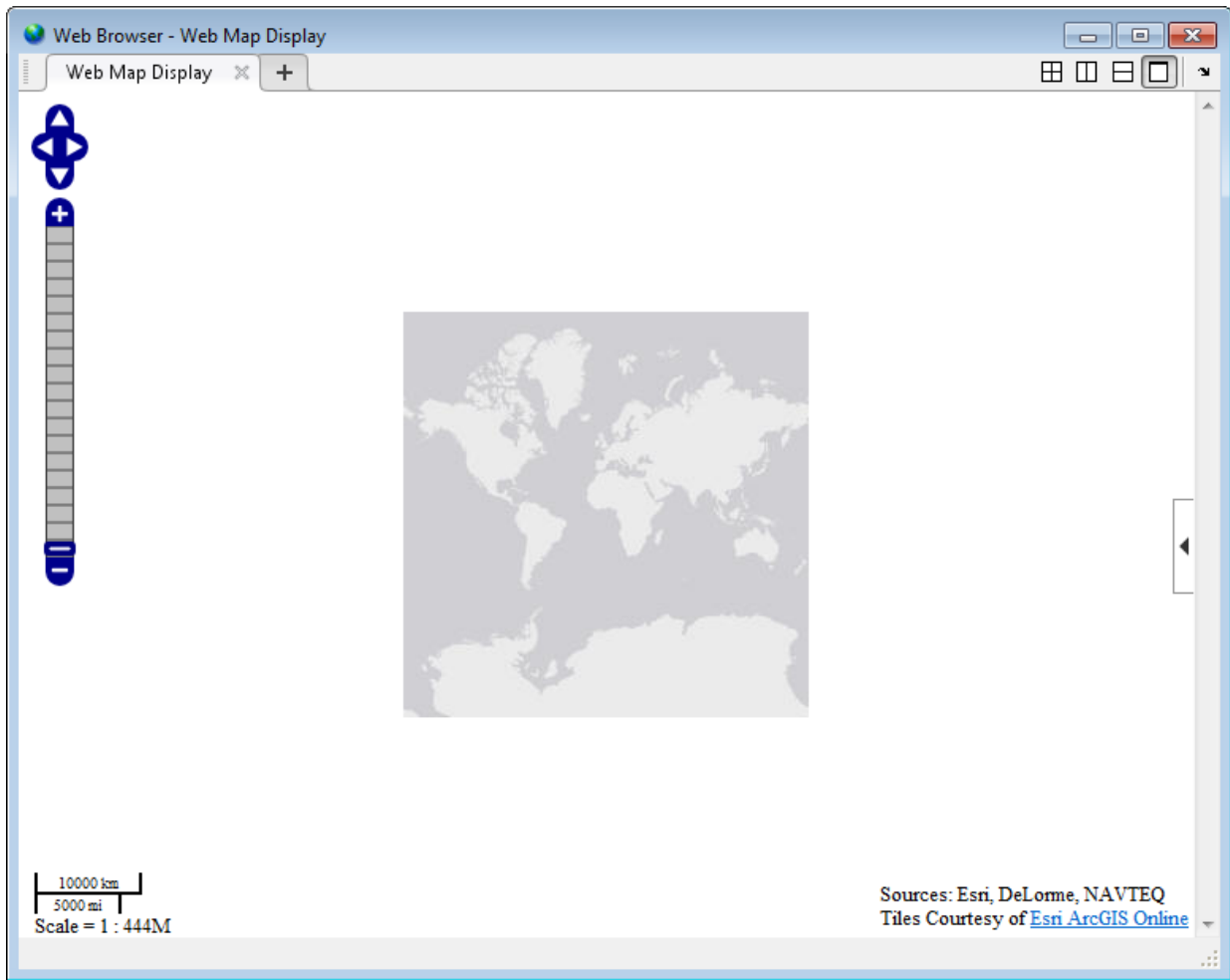
### Display a Web map with a WMS Layer as the Base Layer

Open a web map and retrieve the Blue Marble WMS layer.

```
nasa = wmsfind('nasa', 'SearchField', 'serverurl');
baselayer = refine(nasa, 'bluemarbleng',  ...
                    'SearchField', 'layername',...
                    'MatchType', 'exact');
baselayer = wmsupdate(baselayer);
```

Display a web map with the Blue Marble WMS layer as the base layer.

```
webmap(baselayer)
```
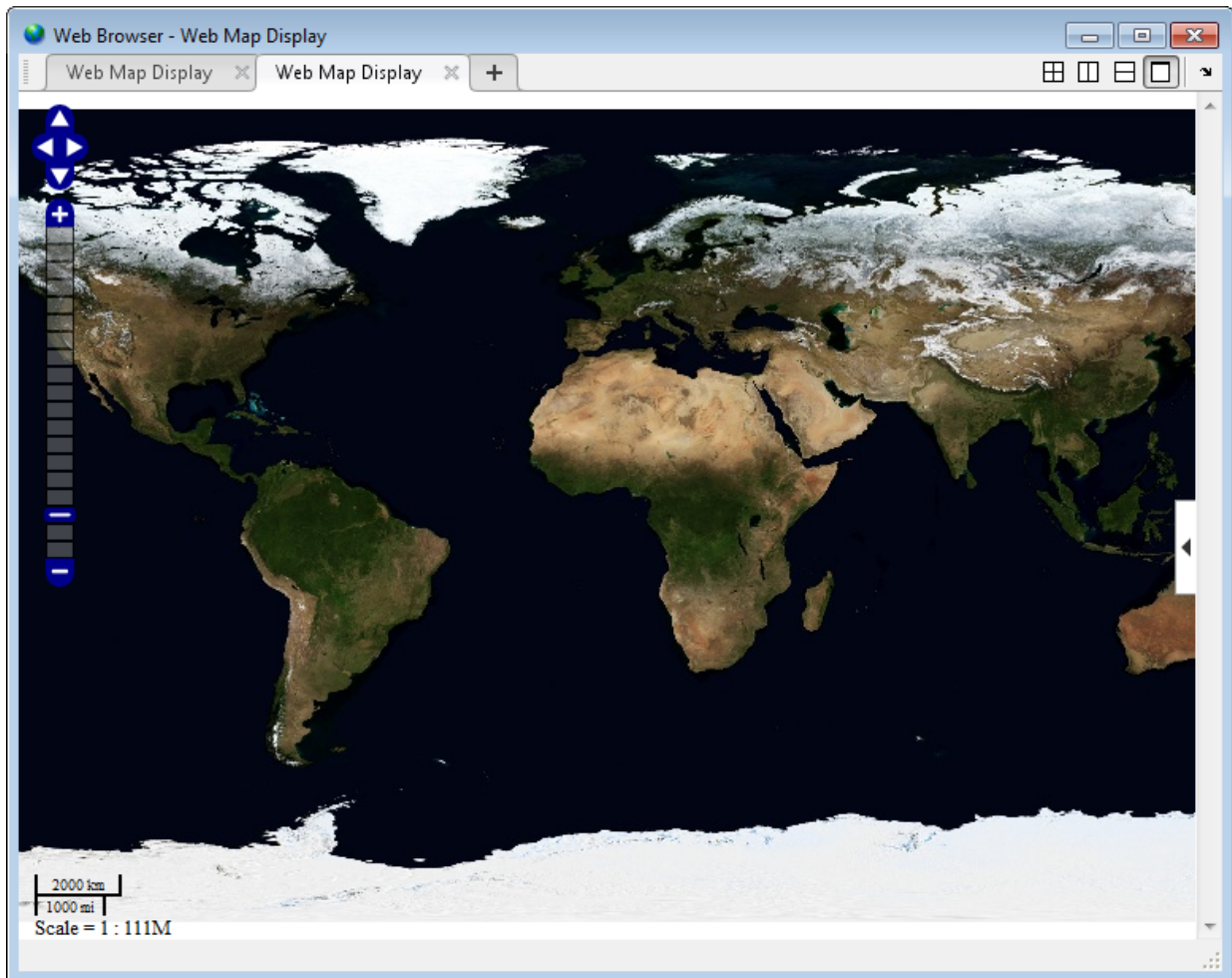


## Input Arguments

**baseLayer — Map displayed in web map browser**
character string | WMSLayer array

Map displayed in web map browser, specified as a character string, listed in the following table, or a WMSLayer array. If specified as string, the value is case insensitive and spaces are optional.

| Name | Description |
| --- | --- |
| `'World Street Map'` | Worldwide street map |
| `'Open Street Map'` | Street map from openstreetmap.org[a] |
| `'MapQuest Open Street Map'` | Open street map tiles provided by MapQuest [b] |
| `'MapQuest Open Aerial Map'` | Open aerial map tiles provided by MapQuest |
| `'World Topographic Map'` | Topographic map for the world from ESRI [c] |
| `'World Shaded Relief'` | Surface elevation as shaded relief provided by ESRI |
| `'World Physical Map'` | Natural Earth map of the world provided by ESRI |
| `'World Terrain Base'` | Shaded relief and bathymetry provided by ESRI |
| `'USGS Vector Map'` | Vector base map for small scales |
| `'USGS Imagery'` | Composite of Blue Marble, NAIP, and Landsat |
| `'USGS Topographic Imagery'` | Topographic map with imagery |
| `'USGS Shaded Topographic Map'` | Composite of contours, shaded relief, and vector layers |
| `'USGS Shaded Relief'` | Shaded relief from National Elevation Dataset [d] |
| `'USGS Filled Vector'` | Polygon area fills tiled for small scales |
| `'National Geographic Map'` | General reference map provided by ESRI |
| `'DeLorme World Basemap'` | Topographic map provided by ESRI |
| `'Ocean Basemap'` | Bathymetry, marine features, depth in meter provided by ESRI |
| `'World Navigation Charts'` | Topographic data with nautical information provided by ESRI |

| Name | Description |
|---|---|
| `'Light Gray Canvas Map'` | Neutral background map with minimal colors provided by ESRI |

a. For information about the Open Street Map layer, visit: http://www.openstreetmap.org.
b. For information about the MapQuest Open layers, visit: http://developer.mapquest.com/web/products/open/map.
c. For information about the Esri ArcGIS Online layers, visit: http://www.esri.com/software/arcgis/arcgisonline/maps/maps-and-map-layers.
d. [e]
de. For information about the USGS layers, visit http://viewer.nationalmap.gov/example/services/serviceList.html

Example: `webmap('Light Gray Canvas Map')`

Data Types: `char`

### `wm` — Web map
web map handle

Web map, specified as a web map handle, returned by the `webmap` function.

Example: `wm = webmap`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `webmap('Light Gray Canvas Map', 'WrapAround', false)`

### `'WrapAround'` — Control map wrapping
true (default) | false

Control map wrapping, specified as a logical value true (`1`) or false (`0`). If `true`, the map supports continuous pan and zoom across the 180-degree meridian, but zoom is constrained to show less than 180 degrees of longitude at a time. If `false`, the map is clipped to the west at -180 and clipped to the east at +180, and zoom can be set to show the full extent of the world.

Example: `webmap('Light Gray Canvas Map', 'WrapAround', false)`

Data Types: `double` | `logical`

# Output Arguments

**wm — Web map**
web map handle

Web map, returned as a web map handle.

# More About

### Web Map

An interactive map accessed through a web page.

In a web map, a user can select different map layers to view and navigate around the map using interactive tools, such as zooming. The web map browser is a window that displays map base layers obtained from Web servers located on the Internet. You can also display overlay layers that contain custom point and line vector data.

### Tips

- Particular maps may not support every available zoom level. If your map displays as completely white, try another zoom level. The map you are displaying may not support the zoom level you have currently selected. You can also select another base layer, which might support the specified zoom level.

# See Also
`wmcenter` | `wmclose` | `wmlimits` | `wmline` | `wmmarker` | `wmprint` | `wmremove` | `wmzoom`

# WebMapServer class

Web map server object

## Description

A `WebMapServer` handle object represents a Web Map Service (WMS) and acts as a proxy to a WMS server. The `WebMapServer` handle object resides physically on the client side. The object can access the capabilities document on the WMS server and perform requests to obtain maps. It supports multiple WMS versions and negotiates with the server automatically to use the highest known version that the server can support.

## Construction

`server = WebMapServer(serverURL)` constructs a `WebMapServer` object from the `serverURL` string parameter. The `serverURL` string parameter must include the protocol `'http://'` or `'https://'`. `WebMapServer` automatically communicates to the server defined by the `serverURL` using the highest known version that the server can support. `serverURL` can contain additional WMS keywords.

## Properties

**`Timeout`**

Indicates the number of milliseconds before a server times out.

**Data Type:** `double`

**Default:** 0 (Indicates that the `WebMapServer` handle object ignores the time-out mechanism.)

**`EnableCache`**

Indicates if the `WebMapServer` handle object allows caching. If `true`, the `WebMapServer` handle object caches the `WMSCapabilites` object, which is returned when you use the

`getCapabilities` method. The cache expires if the `AccessDate` property of the cached `WMSCapabilities` object is not the current day.

**Data Type:** logical

**Default:** `true`

**ServerURL**

Indicates the URL of the server.

**Data Type:** string

**RequestURL**

Indicates the URL of the last request to the server. `RequestURL` specifies a request for either the XML capabilities document or a map. You can insert the `RequestURL` into a browser.

**Data Type:** string

## Methods

| | |
|---|---|
| getCapabilities | Get capabilities document from server |
| getMap | Get raster map from server |
| updateLayers | Update layer properties |

## Examples

### Construct WebMapServer Object and Obtain Server Capabilities Document

Construct a `WebMapServer` object that communicates with one of the Environmental Research Division's Data Access Program (ERDDAP) WMS servers hosted by NOAA and obtains its capabilities document. Search for a server that provides daily, global sea surface temperature (`sst`) data produced by the Jet Propulsion Laboratory's Regional Ocean Modeling System (JPL ROMS) group.

```
layers = wmsfind('coastwatch*jpl*sst','SearchField','serverurl');
serverURL = layers(1).ServerURL;
server = WebMapServer(serverURL);
capabilities = getCapabilities(server);
layers = capabilities.Layer;
```

Obtain data from the server. Show the boundaries of the nations and the global SST data.

```
nations = refine(layers,'nations');
nations = nations(1);
sst = refine(layers,'sst');
sst = sst(1);
layer = [sst nations];
request = WMSMapRequest(layer,server);
A = getMap(server,request.RequestURL);
R = request.RasterReference;
```

Display data from the server.

```
figure
worldmap(A,R)
geoshow(A,R)
title({sst.LayerTitle,sst.Details.Dimension.Default})
```

GHRSST Global 1-km Sea Surface Temperature (G1SST), Global, 0.01 Degree, Daily - SST
2015-06-15T00:00:00Z
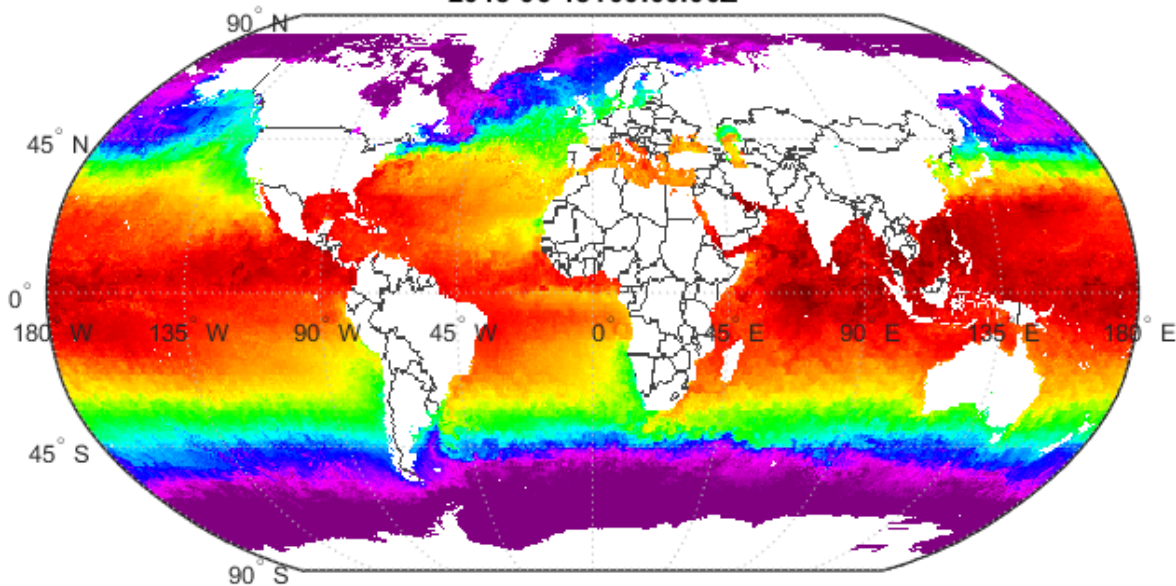
## See Also

WMSCapabilities | wmsfind | wmsinfo | WMSMapRequest | wmsread | wmsupdate

# getCapabilities

**Class:** WebMapServer

Get capabilities document from server

## Syntax

```
capabilities = getCapabilities(server)
```

## Description

`capabilities = getCapabilities(server)` retrieves the capabilities document from the server as a `WMSCapabilities` object and updates the `RequestURL` property.

## Tips

The `getCapabilities` method accesses the Internet to retrieve the document. Periodically, the WMS server is unavailable. Retrieving the document can take several minutes.

## Examples

Retrieve the capabilities document from the NASA SVS Image Server:

```
nasa = wmsfind('NASA SVS Image', 'SearchField', 'servertitle');
serverURL = nasa(1).ServerURL;
server = WebMapServer(serverURL);
capabilities = getCapabilities(server);
```

# getMap

**Class:** WebMapServer

Get raster map from server

## Syntax

```
A = getMap(server,mapRequestURL)
```

## Description

`A = getMap(server,mapRequestURL)` dynamically renders and retrieves a color or grayscale, geographically referenced, raster map from the server and stores it in `A`. `server` is a WebMapServer object that specifies the server. Parameters in the URL, `mapRequestURL`, define the map. The `getMap` method also updates the `WMSMapRequest.RequestURL` property `mapRequestURL`.

## Tips

`getMap` accesses the Internet to retrieve the map. Periodically, the WMS server is unavailable. Retrieving the map can take several minutes.

## Examples

### Retrieve Map from NASA Earth Observations WMS Server

Retrieve a map of the Blue Marble global mosaic layer from the NASA Earth Observations WMS server.

```
neowms = wmsfind('neowms', 'SearchField', 'serverurl');
layer = refine(neowms, 'bluemarbleng', 'MatchType', 'exact');

server = WebMapServer(layer.ServerURL);
mapRequest = WMSMapRequest(layer, server);
```

```
A = getMap(server, mapRequest.RequestURL);
R = mapRequest.RasterReference;
```

Display the map.

```
figure
worldmap world
geoshow(A, R)
setm(gca,'MLabelParallel',-90,'MLabelLocation',90)
title(layer.LayerTitle)
```



**Blue Marble: Next Generation**
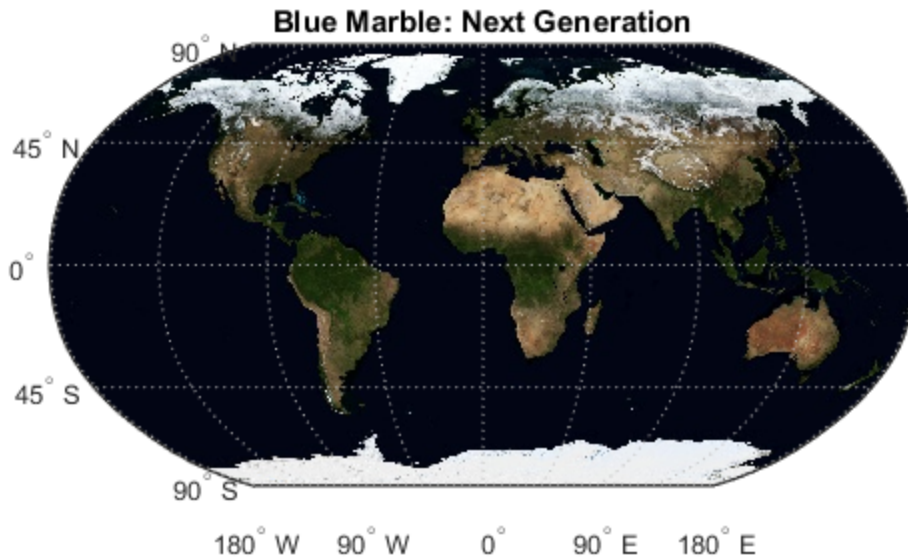
Courtesy NASA Earth Observing System

# updateLayers

**Class:** WebMapServer

Update layer properties

## Syntax

```
[updatedLayer, index] = updateLayers(server,layer)
```

## Description

`[updatedLayer, index] = updateLayers(server,layer)` returns a `WMSLayer` array with properties updated with values from `server`, a WebMapServer object. The `WMSLayer` array `Layer` must contain only one unique `ServerURL`. The `updateLayers` method removes layers no longer available on the server. The logical array `index` contains `true` for each available layer, such that `updatedLayers` has the same size as `layer(index)`.

The `updateLayers` method accesses the Internet to update the properties. Occasionally, a WMS server is unavailable, or several minutes elapse before the properties are updated.

## Examples

### Update Properties of MODIS Global Mosaic Layer

Update the properties of a MODIS global mosaic layer obtained from the NASA Earth Observations WMS server.

```
nasa = wmsfind('NASA Earth Observations','SearchField','any');
modis = refine(nasa,'land*day*month');
modis = modis(1);
```

Create a WebMapServer object.

```
server = WebMapServer(modis.ServerURL);
```

Update the properties of the MODIS layer.

```
updatedLayer = updateLayers(server,modis);
```

Obtain the map and display it..

```
mapRequest = WMSMapRequest(updatedLayer,server);
A = getMap(server,mapRequest.RequestURL);
R = mapRequest.RasterReference;
figure
ax = worldmap('world');
geoshow(A,R)
setm(ax,'MLabelParallel',-90,'MLabelLocation',90)
title({'MODIS Global Mosaic',modis.LayerTitle})
```
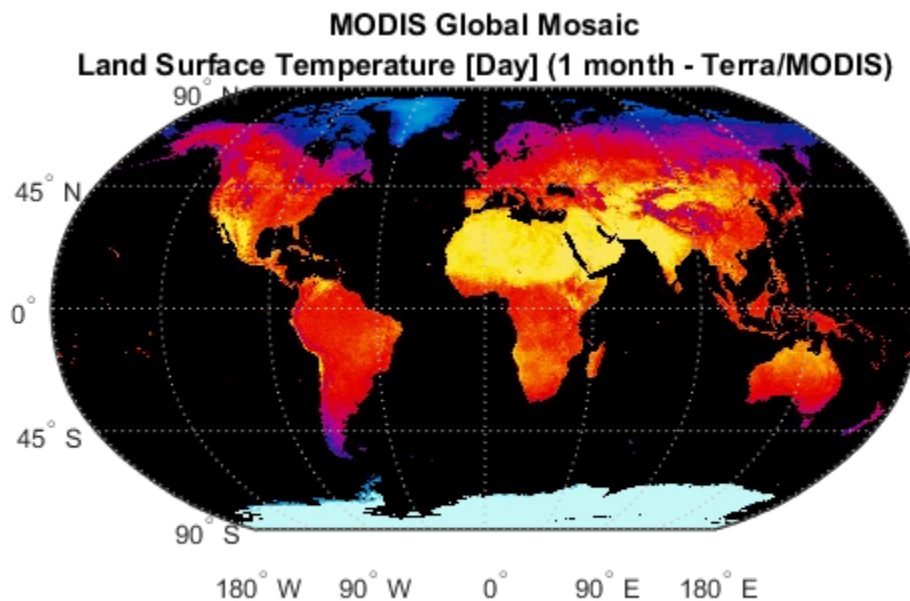


Courtesy of NASA Earth Observing System

View the metadata of the layer.

```
metadata = webread(updatedLayer.Details.MetadataURL);
```

```
disp(metadata)
```

**Update Properties of Layers from Multiple Servers**

Find layers from USGS servers with the word "geology" in the server URL.

```
 usgsLayers = wmsfind('usgs.gov*geology','SearchField','serverurl');
```

Find the layers for an individual server, update their properties, and append them to the `updatedLayers` array.

```
serverURLs = usgsLayers.servers;
updatedLayers = [];
fprintf('Updating layer properties from %d servers.\n', ...
        numel(serverURLs));
for k=1:numel(serverURLs)
   serverLayers = refine(usgsLayers, serverURLs{k}, ...
           'SearchField','serverurl','MatchType','exact');
   serverURL = serverLayers(1).ServerURL;
   fprintf('Updating properties from server %d:\n%s\n', ...
           k, serverURL);
   server = WebMapServer(serverURL);
   try
      layers = updateLayers(server,serverLayers);
      % Grow using concatenation because layers can have any
     % length ranging from O to numel(serverLayers).
      updatedLayers = [updatedLayers; layers];
   catch e
       fprintf('Server %s is not responding.\n', ...
               serverURL);
       fprintf('Error message is %s\n', e.message)
    end
end
```

```
Updating layer properties from 16 servers.
Updating properties from server 1:
http://certmapper.cr.usgs.gov/arcgis/services/geology/afghanistan/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/afghanistan/MapServer/WMSS
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 2:
http://certmapper.cr.usgs.gov/arcgis/services/geology/africa/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/africa/MapServer/WMSServer
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 3:
http://certmapper.cr.usgs.gov/arcgis/services/geology/arabian/MapServer/WMSServer?
```

```
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/arabian/MapServer/WMSServe
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 4:
http://certmapper.cr.usgs.gov/arcgis/services/geology/arctic/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/arctic/MapServer/WMSServer
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 5:
http://certmapper.cr.usgs.gov/arcgis/services/geology/asiapacific/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/asiapacific/MapServer/WMSS
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 6:
http://certmapper.cr.usgs.gov/arcgis/services/geology/bangladesh/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/bangladesh/MapServer/WMSSe
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 7:
http://certmapper.cr.usgs.gov/arcgis/services/geology/caribbean/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/caribbean/MapServer/WMSSer
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 8:
http://certmapper.cr.usgs.gov/arcgis/services/geology/europe/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/europe/MapServer/WMSServer
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 9:
http://certmapper.cr.usgs.gov/arcgis/services/geology/formersovietunion/MapServer/WMSSe
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/formersovietunion/MapServe
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 10:
http://certmapper.cr.usgs.gov/arcgis/services/geology/gulfofmexico/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/gulfofmexico/MapServer/WMS
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 11:
http://certmapper.cr.usgs.gov/arcgis/services/geology/iran/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/iran/MapServer/WMSServer?
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 12:
http://certmapper.cr.usgs.gov/arcgis/services/geology/southamerica/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/southamerica/MapServer/WMS
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 13:
http://certmapper.cr.usgs.gov/arcgis/services/geology/southasia/MapServer/WMSServer?
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/southasia/MapServer/WMSSer
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 14:
http://certmapper.cr.usgs.gov/arcgis/services/geology/wyoming/MapServer/WMSServer?
```

```
Server http://certmapper.cr.usgs.gov/arcgis/services/geology/wyoming/MapServer/WMSServe
Error message is The Capabilities file from the host 'certmapper.cr.usgs.gov' cannot be
Updating properties from server 15:
http://certmapper.cr.usgs.gov/arcgis/services/one_geology_wms/USGS_Geologic_Map_of_Nort
Updating properties from server 16:
http://certmapper.cr.usgs.gov/arcgis/services/one_geology_wms/USGS_Geologic_Map_of_Nort
```

# westof

Wrap longitudes to values west of specified meridian

---

**Note:** The `westof` function is obsolete and will be removed in a future release of the toolbox. Replace it with the following calls, which are also more efficient:

```
westof(lon,meridian,'degrees') ==> meridian-mod(meridian-lon,360)

westof(lon,meridian,'radians') ==> meridian-mod(meridian-lon,2*pi)
```

---

## Syntax

```
lonWrapped = westof(lon,meridian)
lonWrapped = westof(lon,meridian,angleunits)
```

## Description

`lonWrapped = westof(lon,meridian)` wraps angles in `lon` to values in the interval `(meridian-360 meridian]`. `lon` is a scalar longitude or vector of longitude values. All inputs and outputs are in degrees.

`lonWrapped = westof(lon,meridian,angleunits)` specifies the input and output units with the string *angleunits*. *angleunits* can be either `'degrees'` or `'radians'`. It may be abbreviated and is case-insensitive. If *angleunits* is `'radians'`, the input is wrapped to the interval `(meridian-2*pi meridian]`.

# wgs84Ellipsoid

Reference ellipsoid for World Geodetic System 1984

## Syntax

```
E = wgs84Ellipsoid
E = wgs84Ellipsoid(lengthUnit)
```

## Description

`E = wgs84Ellipsoid` returns a `referenceEllipsoid` object representing the World Geodetic System of 1984 (WGS 84) reference ellipsoid. The semimajor axis and semiminor axis are expressed in meters.

`E = wgs84Ellipsoid(lengthUnit)` returns a WGS 84 reference ellipsoid object in which the semimajor axis and semiminor axis are expressed in the specified unit, `lengthUnit`.

## Input Arguments

**`lengthUnit` — Unit of measure**
`'meter'` (default) | character string

Unit of measure, specified as a character string. You can specify any string accepted by the `validateLengthUnit` function.

Example:

Data Types: `char`

## Output Arguments

**E — WGS84 reference ellipsoid**
`referenceEllipsoid` object

WGS84 reference ellipsoid, returned as a referenceEllipsoid object.

# Examples

### Create wgs84 Reference Ellipsoid and View Derived Property

Create a World Geodetic System of 1984 (wgs84) reference ellipsoid and view one of its derived properties.

```
wgs84InMeters = wgs84Ellipsoid

wgs84InMeters =

referenceEllipsoid with defining properties:

                Code: 7030
                Name: 'World Geodetic System 1984'
           LengthUnit: 'meter'
        SemimajorAxis: 6378137
        SemiminorAxis: 6356752.31424518
     InverseFlattening: 298.257223563
         Eccentricity: 0.0818191908426215

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

View value of `SurfaceArea` property.

```
wgs84InMeters.SurfaceArea

ans =

   5.1007e+14
```

### Create wgs84 Reference Ellipsoid Specifying Units

Create a World Geodetic System of 1984 (wgs84) reference ellipsoid, specifying the unit of measure. In the summary information returned, note the value of the `LengthUnit` field.

```
wgs84InKilometers = wgs84Ellipsoid('km')

wgs84InKilometers =

referenceEllipsoid with defining properties:

                Code: 7030
                Name: 'World Geodetic System 1984'
          LengthUnit: 'kilometer'
       SemimajorAxis: 6378.137
       SemiminorAxis: 6356.75231424518
   InverseFlattening: 298.257223563
        Eccentricity: 0.0818191908426215

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

## See Also

referenceEllipsoid | `validateLengthUnit`

# wmclose

Close web map

## Syntax

```
wmclose
wmclose(wm)
wmclose all
```

## Description

`wmclose` closes the current web map.

`wmclose(wm)` closes the web map specified by `wm`.

`wmclose all` closes all web maps.

## Examples

### Close Current Web Map

Open a web map, pause one second, and then close the web map.

```
webmap
pause(1)
wmclose
```

### Close Specified Web Map

Open two web maps, pause for one second, and then close one of the web maps.

```
h1 = webmap;
h2 = webmap('ocean basemap');
pause(1)
wmclose(h1)
```

### Close All Web Maps

Open two web maps, pause for one second, and then close all the web maps.

```
h1 = webmap;
h2 = webmap('ocean basemap');
pause(1)
wmclose all
```

## Input Arguments

### wm — Web map
web map handle

Web map, specified as a web map handle. You use the webmap function to get a web map handle when you create a web map.

## See Also
webmap | wmcenter | wmlimits | wmline | wmmarker | wmprint | wmremove |
wmzoom

# wmprint

Print web map

## Syntax

```
wmprint()
wmprint(wm)
```

## Description

wmprint() prints the contents of the current web map to a printer.

wmprint(wm) prints the contents of the web map specified by wm.

## Examples

### Print Web Map

Create a web map, specifying a base layer.

```
webmap('mapquestopenstreetmap')
```

Position the web map.

```
wmcenter(51.487, 0, 15)
```

Print the contents of the web map.

```
wmprint()
```

## Input Arguments

### wm — Web map
scalar web map handle

Web map, specified as a scalar web map handle.

## See Also

webmap | wmcenter | wmclose | wmlimits | wmline | wmmarker | wmremove |
wmzoom

# wmmarker

Display geographic marker on web map

## Syntax

```
wmmarker(lat,lon)
wmmarker(p)
wmmarker(wm, ___ )
wmmarker(__,Name,Value)
h = wmmarker( ___ )
```

## Description

wmmarker(lat,lon) displays a marker overlay consisting of the points in lat and lon. The overlay appears on the current web map, or if there is no current web map, wmmarker creates one. wmmarker centers the map so that all vector overlays on the web map are visible.

wmmarker(p) displays a marker overlay based on the content of the geopoint vector p. The overlay contains one point for each element of p.

wmmarker(wm, ___ ) displays the overlay in the web map specified by the web map handle, wm.

wmmarker(__,Name,Value) specifies name-value pairs that set additional display properties. Parameter names can be abbreviated and are case-insensitive.

h = wmmarker( ___ ) returns a handle to the overlay.

## Examples

### Draw Marker on Web Map

Add a marker positioned at London, England. In the illustration, note the default marker name and overlay name.

```
lat = 51.5187666404504;
lon = -0.130003487285315;
wmmarker(lat, lon);
```
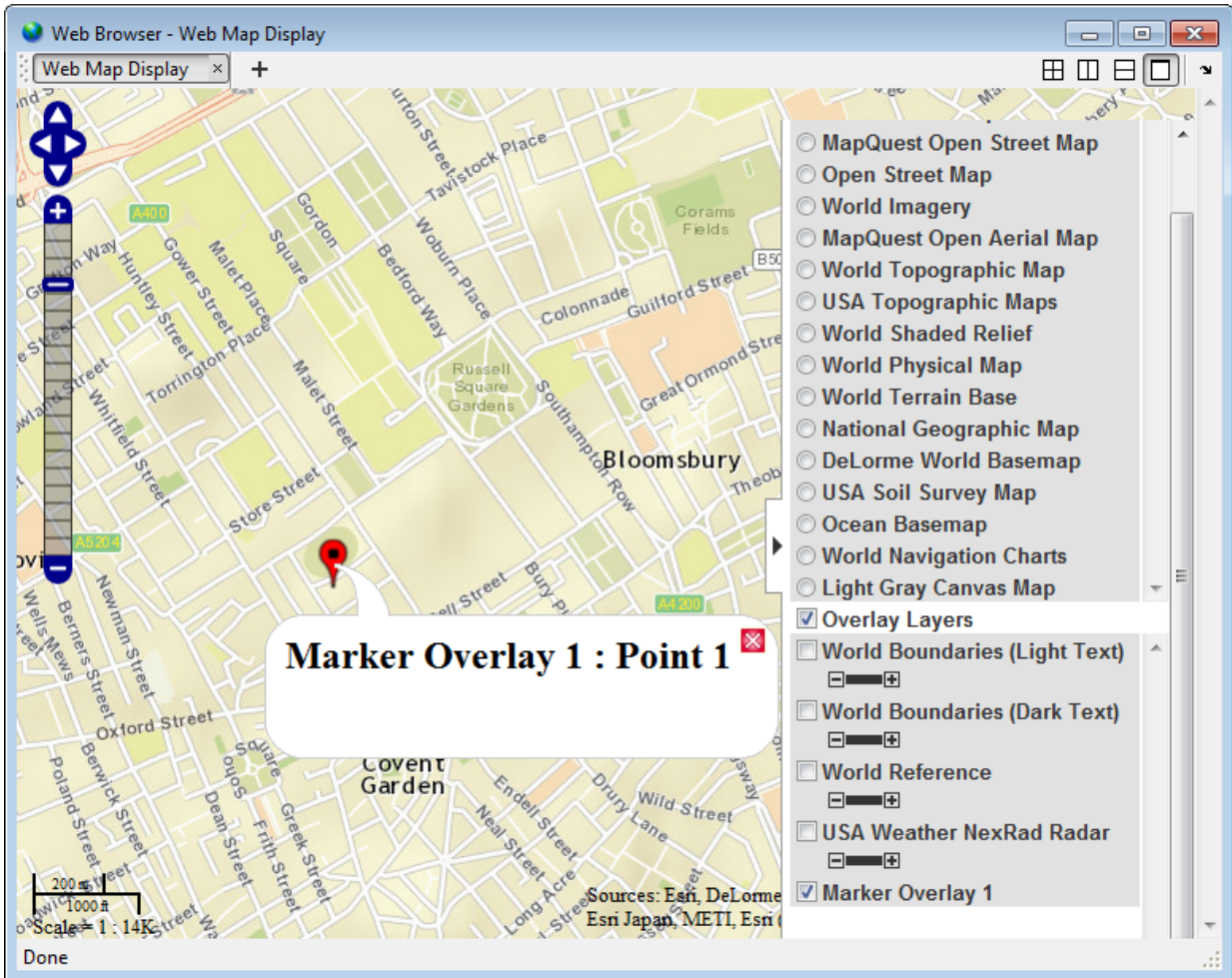


### Draw Markers at Features Defined in a geopoint Vector

Read feature data into geopoint vector.

```
p = gpxread('boston_placenames');
```

Draw markers at features defined in geopoint vector. wmmarker creates the web map. In the illustration, note the feature name in the balloon and the overlay name in the Layer Manager, which were specified in the call to wmmarker.

wmmarker(p, 'FeatureName', p.Name, 'OverlayName', 'Boston Placenames')
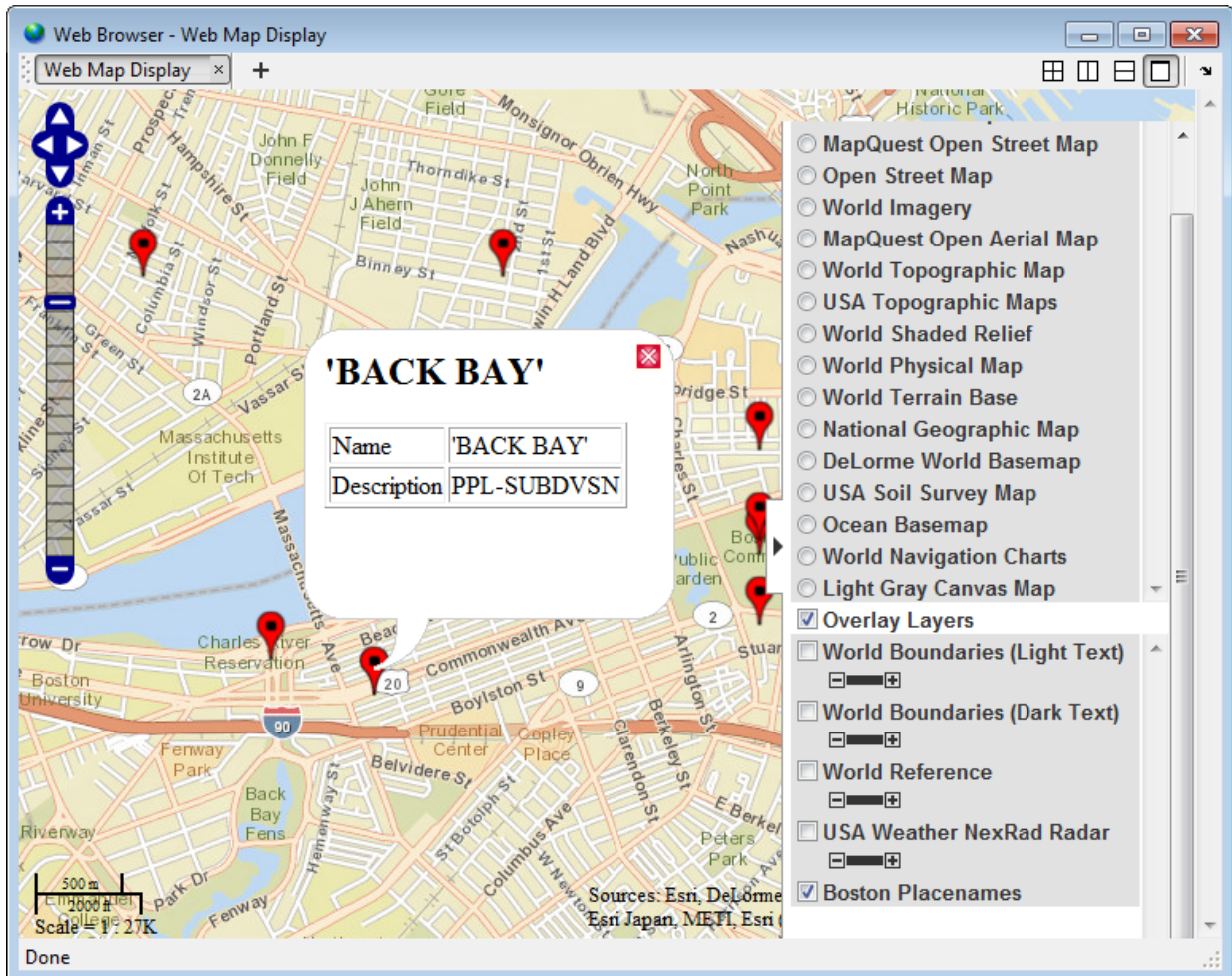


**Draw Marker Using Custom Icon and Description Data**

Define location.

```
lat =  42.299827;
lon = -71.350273;
```

Specify text in description balloon and name.

```
description = sprintf('%s<br>%s</br><br>%s</br>',...
                      '3 Apple Hill Drive', 'Natick, MA. 01760', ...
                      'http://www.mathworks.com');
 name = 'The MathWorks, Inc.';
```

Specify custom icon.

```
iconDir = fullfile(matlabroot,'toolbox','matlab','icons');
iconFilename = fullfile(iconDir, 'matlabicon.gif');
```

Draw marker on web map, specifying `Description`, `FeatureName`, `Icon`, and `OverlayName`. `wmmarker` creates the web map. In the illustration, note the custom icon and the HTML formatting in the description.

```
wmmarker(lat, lon, 'Description', description, ...
                   'FeatureName', name,...
                   'Icon', iconFilename,...
                   'OverlayName', name);
```
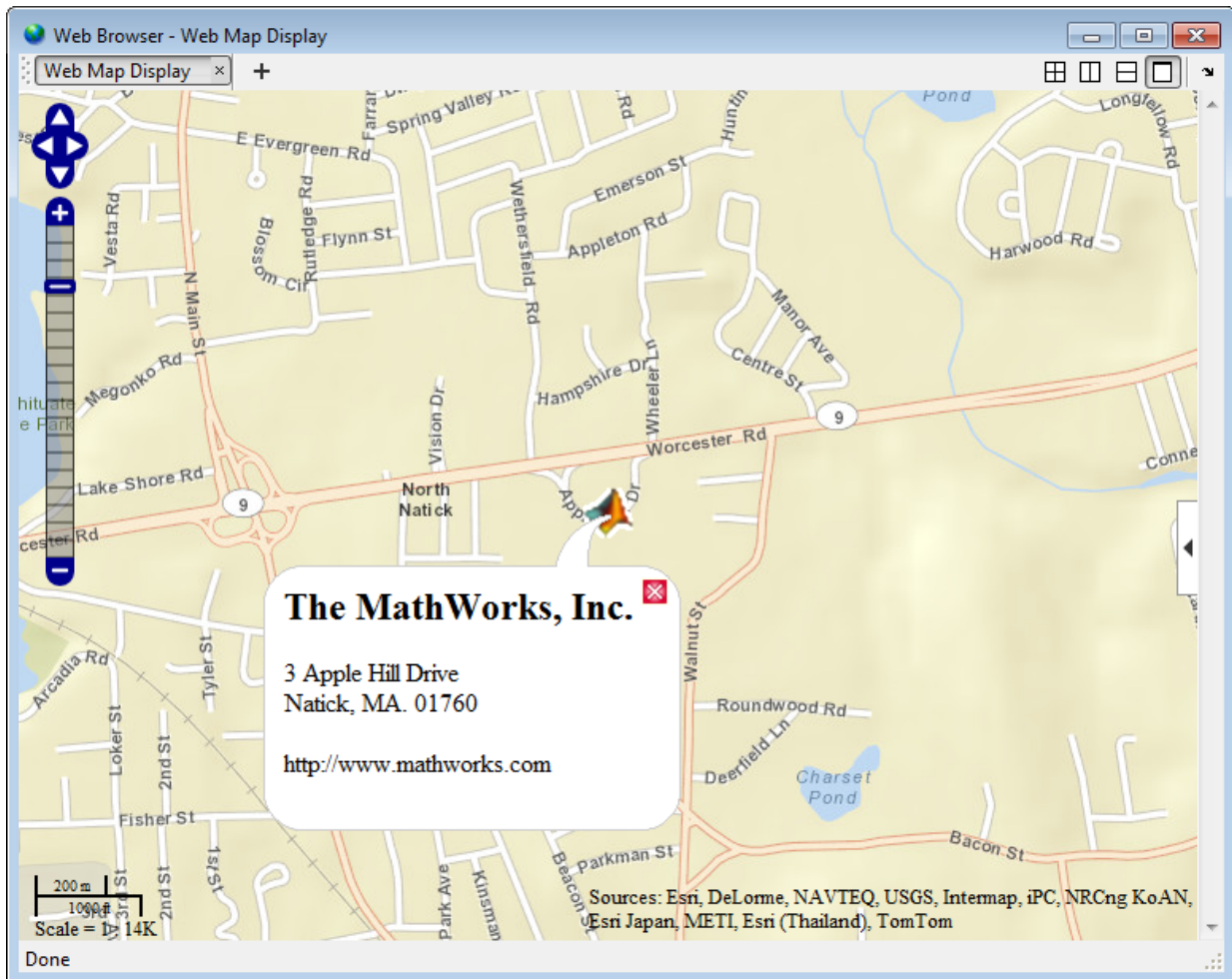
### Draw Marker Overlay Using Attribute Spec

Import a shapefile representing tsunami (tidal wave) events reported over several decades, tagged geographically by source location.

```
S = shaperead('tsunamis', 'UseGeoCoords', true);
```

Convert the geostruct returned by shaperead into a geopoint vector.

```
p = geopoint(S);
```

Create an attribute spec and modify it to define a table of values to display in the feature balloon, including year, cause, country, location, and maximum height. The attribute spec defines the format of the expected value for each field.

```
attribspec = makeattribspec(p);

desiredAttributes = ...
      {'Max_Height', 'Cause', 'Year', 'Location', 'Country'};
allAttributes = fieldnames(attribspec);
attributes = setdiff(allAttributes, desiredAttributes);
attribspec = rmfield(attribspec, attributes);
attribspec.Max_Height.AttributeLabel = '<b>Maximum Height</b>';
attribspec.Max_Height.Format = '%.1f Meters';
attribspec.Cause.AttributeLabel = '<b>Cause</b>';
attribspec.Year.AttributeLabel = '<b>Year</b>';
attribspec.Year.Format = '%.0f';
attribspec.Location.AttributeLabel = '<b>Location</b>';
attribspec.Country.AttributeLabel = '<b>Country</b>';
```

Create a web map, specifying the base layer. Then add the marker overlay. In the illustration, note the table containing the data you specified in the attribute spec.

```
webmap('ocean basemap', 'WrapAround', false);
wmmarker(p, 'Description', attribspec,...
            'OverlayName', 'Tsunami Events')
wmzoom(2)
```
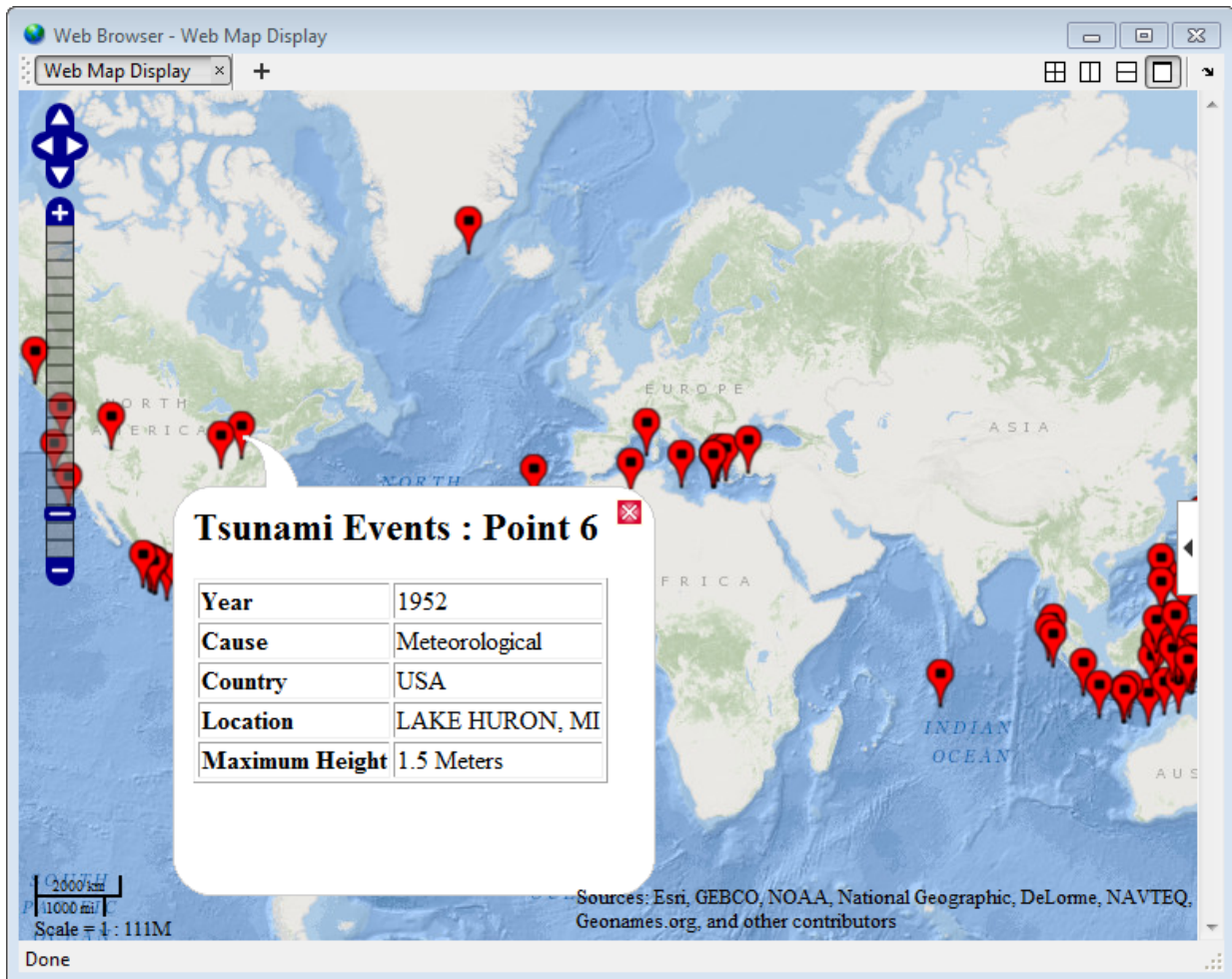
## Input Arguments

**lat — Latitude of points**
matrix of type `single` or `double`

Latitude of points, specified as an array of type `single` or `double`.

Example: `lat = [42.3583 40.7142];`

Data Types: `single` | `double`

### `lon` — Longitude of points
matrix of type `single` or `double`

Longitude of points, specified as an array of type `single` or `double`.

Example: `lon = [-71.0603 −74.0064];`

Data Types: `single` | `double`

### p — Geographic features
geopoint vector

Geographic features, specified as a geopoint vector.

Example: `p = gpxread('boston_placenames');`

### wm — Web map
web map handle

Web map, specified as a web map handle.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `wmmarker(lat,lon,'Autofit',true)`

### `'Autofit'` — Overlay visibility
true (default) | false

Overlay visibility, specified as the logical flag `true` or `false`, or the numeric value 1 or 0. If `true` or 1, `wmmarker` adjusts the spatial extent of the map to ensure that all the vector overlays on the map are visible. If `false`, `wmmarker` does not adjust the spatial extent when the overlay is added to the map.

Overlay visibility, specified as a scalar logical or numeric value `true` (1) or `false` (0).

- If `true`, `wmmarker` adjusts the spatial extent of the map to ensure that all the vector overlays on the map are visible.

- If `false`, `wmmarker` does not adjust the spatial extent when the overlay is added to the map.

Example: `wmmarker(lat,lon,'Autofit',true)`

Data Types: `double` | `logical`

### `'Description'` — Description of feature
empty string (default) | character string | cell array of strings | scalar structure

Description of feature, specified as a character string, cell array of strings, or scalar structure. The description defines the content that is displayed in the feature's description balloon which appears when a user clicks on the feature in the web map. Description elements can be either plain text or marked up with HTML. When you specify an attribute spec, the display in the balloon for the attribute fields of `p` are modified according to the specification. The default value is an empty string (`' '`). If the value is a structure, the attribute spec is applied to the attributes of each feature of `p` and ignored with `lat` and `lon` input.

- If the value is a cell array it is either scalar or the same length as `p`, or `lat` and `lon`, and specifies the description for each marker.
- If the value is a structure, the attribute spec is applied to the attributes of each feature of `p` and ignored with `lat` and `lon` input.

Data Types: `char` | `struct` | `cell`

### `'OverlayName'` — Name of overlay layer
`'Marker Overlay N'`, (default) | text string

Name of overlay layer, specified as a text string. `wmmarker` inserts the name in the Layer Manager under the Overlays item. The Layer Manager is the tool that appears on the right side of the web map frame. The default name is `'Marker Overlay N'` where *N* is the number assigned to this overlay.

Example: `wmmarker(lat,lon,'OverlayName','Route');`

Data Types: `char`

### `'FeatureName'` — Name of feature
`'OverlayName: Point K'` (default) | text string | cell array of strings

Name of feature, specified as a text string or cell array of strings. The name appears in the feature's balloon when a user clicks on the feature in the web map. The default value

is '*OverlayName* : Point *K*', where *OverlayName* is the name of the overlay and *K* is the number assigned to a particular point. If the value is a string, `wmmarker` applies it to all features. If the value is a cell array, it must be a scalar or an array with the same length as `p` or `lat` and `lon`.

Example: `wmmarker(lat,lon,'FeatureName','Mt.Ranier');`

Data Types: `char` | `cell`

### `'Icon'` — File name of custom icon for a marker
text string | cell array of strings

File name of custom icon for a marker, specified as a text string or cell array of strings. If the icon filename is not in the current folder, or in a folder on the MATLAB path, specify a full or relative pathname. The string may be an Internet URL, which must include the protocol type. If the icon filename is not specified, the default icon is displayed. For best results when you want to view a non-default icon, specify a PNG file containing image data with an alpha mask.

- If the value is a string, `wmmarker` applies the value to all markers.
- If you specify a cell array, it must be the same length as `p`, or `lat` and `lon`, and specifies the icon for each marker.

Data Types: `char` | `cell`

### `'IconScale'` — Scaling factor for icon
1 (default) | positive numeric scalar or vector.

Scaling factor for icon, specified as a positive numeric scalar or vector.

- If the value is a scalar, the value is applied to all icons.
- If the value is a vector, it must specify a value for each icon, and it must be the same length as `lat` and `lon` or `p`.

Data Types: `double`

### `'Color'` — Color of icon
`'red'` (default) | ColorSpec | cell array of strings | *M*-by-3 numeric array

Color of icon, specified as a MATLAB Color Specification (ColorSpec), a cell array of color strings, or a numeric array. The color is applied to the icon when a custom icon file has not been specified, otherwise it is ignored. The default value is 'red'. If the value is a cell

array, it must be the same length as LAT and LON, or P. If the value is a numeric array, it must be 1-by-3 or *M*-by-3 where *M* is the length of `lat` and `lon` or `p`.

- If the value is a cell array, it must be scalar or the same length as `p`.
- If the value is a numeric array, it must be an *M*-by-3 where *M* is either 1 or the length of `p`.

# Output Arguments

### h — Handle to marker overlay
scalar

Handle to marker overlay, returned as a scalar.

## See Also
`webmap` | `wmcenter` | `wmclose` | `wmlimits` | `wmline` | `wmprint` | `wmremove` | `wmzoom`

# wmline

Display geographic line on web map

## Syntax

```
wmline(lat,lon)
wmline(p)
wmline(wm, ___ )
wmline( ___ ,Name,Value)
h = wmline( ___ )
```

## Description

wmline(lat,lon) displays a line overlay consisting of the vertices in lat,lon on the current web map. If there is no current web map, wmline creates one. wmline centers the map so that all vector overlays displayed on the web map are visible.

wmline(p) displays a line overlay based on the content of the geopoint or geoshape vector p.

- If p is a geopoint vector, the overlay contains a single line connecting its vertices.
- If p is a geoshape vector, the overlay contains one line feature for each element of p.

wmline(wm, ___ ) displays the line overlay on the web map specified by the web map handle, wm

wmline( ___ ,Name,Value) specifies name-value pairs that set additional display properties

h = wmline( ___ ) returns a handle to line overlay.

## Examples

### Draw Line on Web Map

Find the latitude and longitude for London and Paris.

```
cities = shaperead('worldcities.shp', 'UseGeoCoords', true, ...
    'Selector', ...
    {@(v)(ismember(v, {'London', 'Paris'})), 'Name'});
lat = [cities.Lat];
lon = [cities.Lon];
```

Draw line on web map from London to Paris.

```
wmline(lat, lon)
```
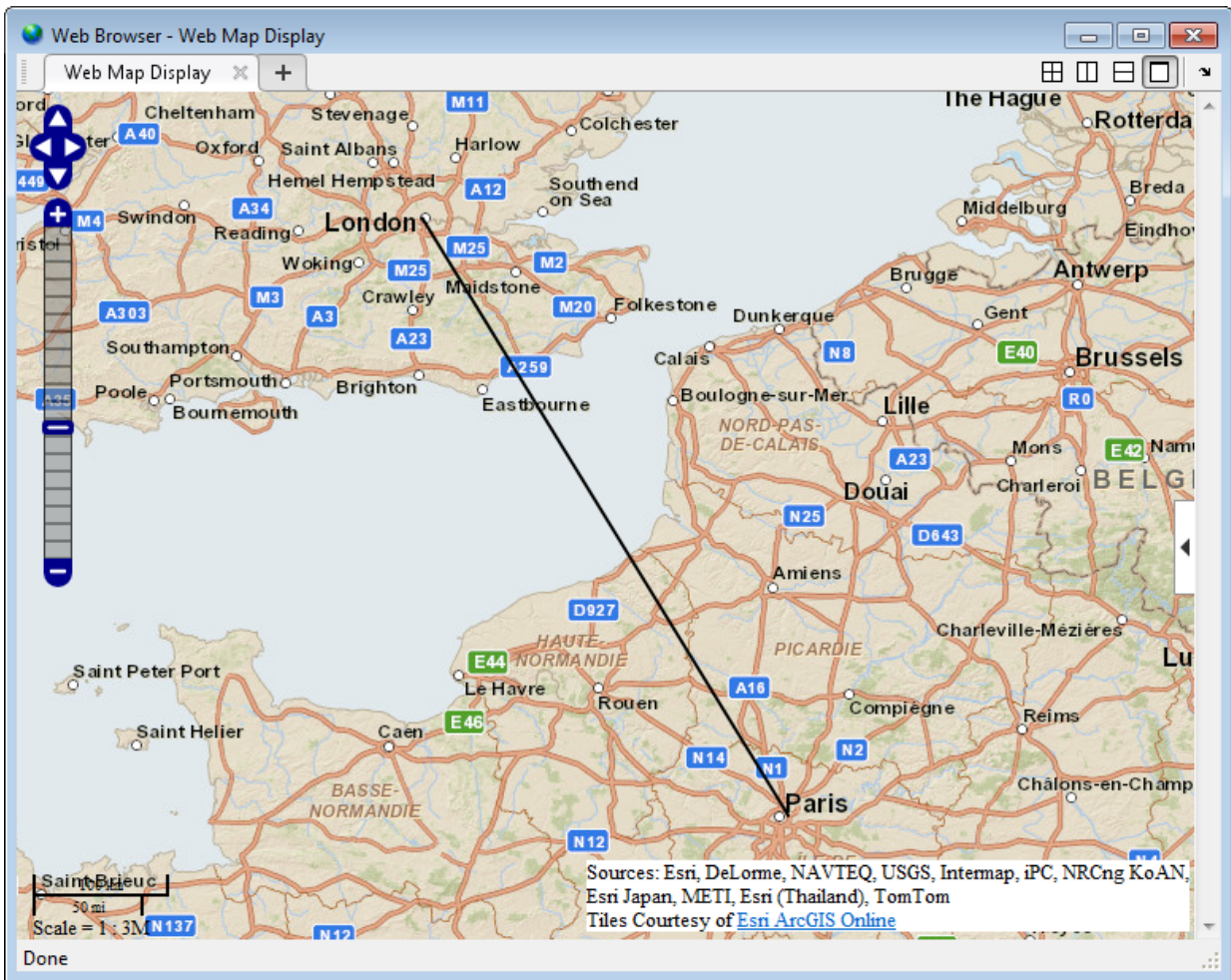
### Draw Reduced Line on Web Map

Large data sets can sometimes be slow to display, making the web map browser appear to hang. To work around this issue, reduce the size of the data set using the `reducem` function before calling `wmline.`.

Load vector data.

```
coast = load('coast');
```

Reduce length of vectors.

```
[lat, lon] = reducem(coast.lat, coast.long);
```

Create a web map that does not wrap since data is of global extent.

```
webmap('ocean basemap', 'WrapAround', false)
```
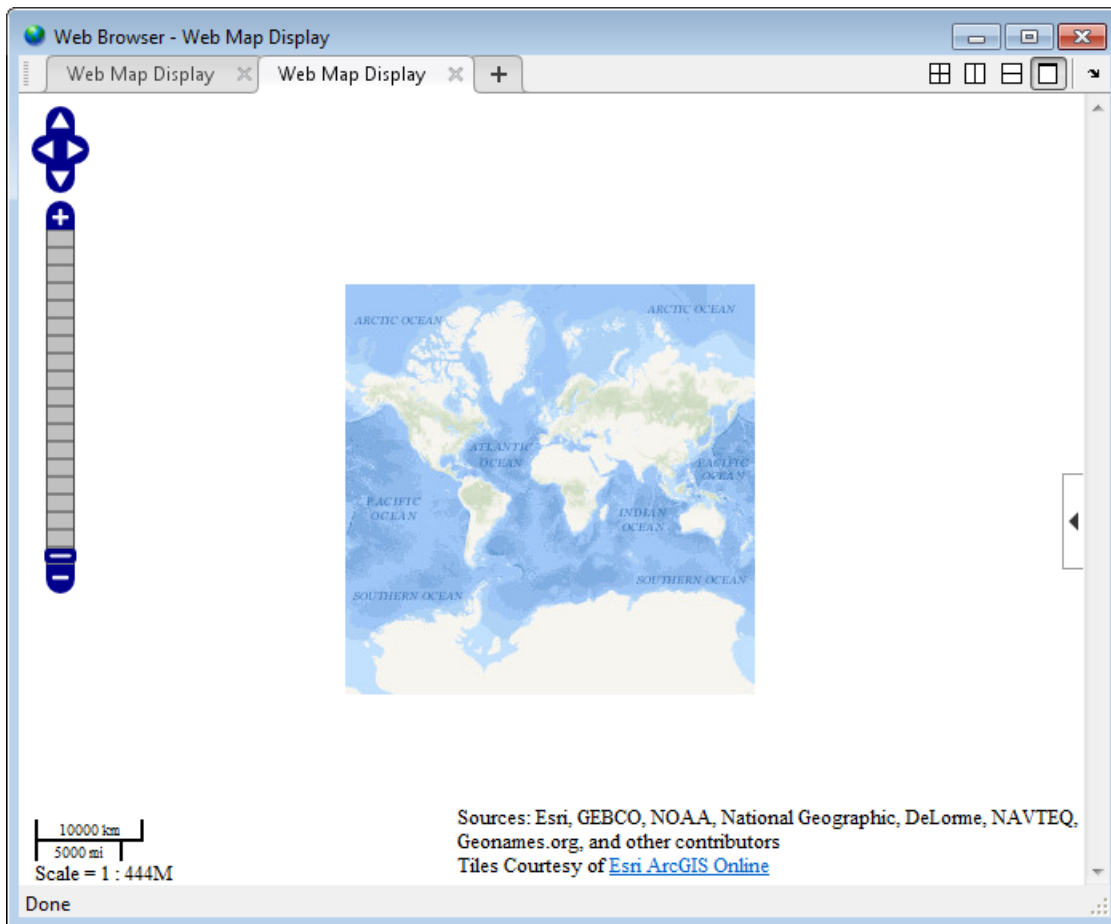


Draw coast line on web map.

```
wmline(lat, lon, 'Width', 3, 'FeatureName', 'coastline')
```

```
wmzoom(2)
```
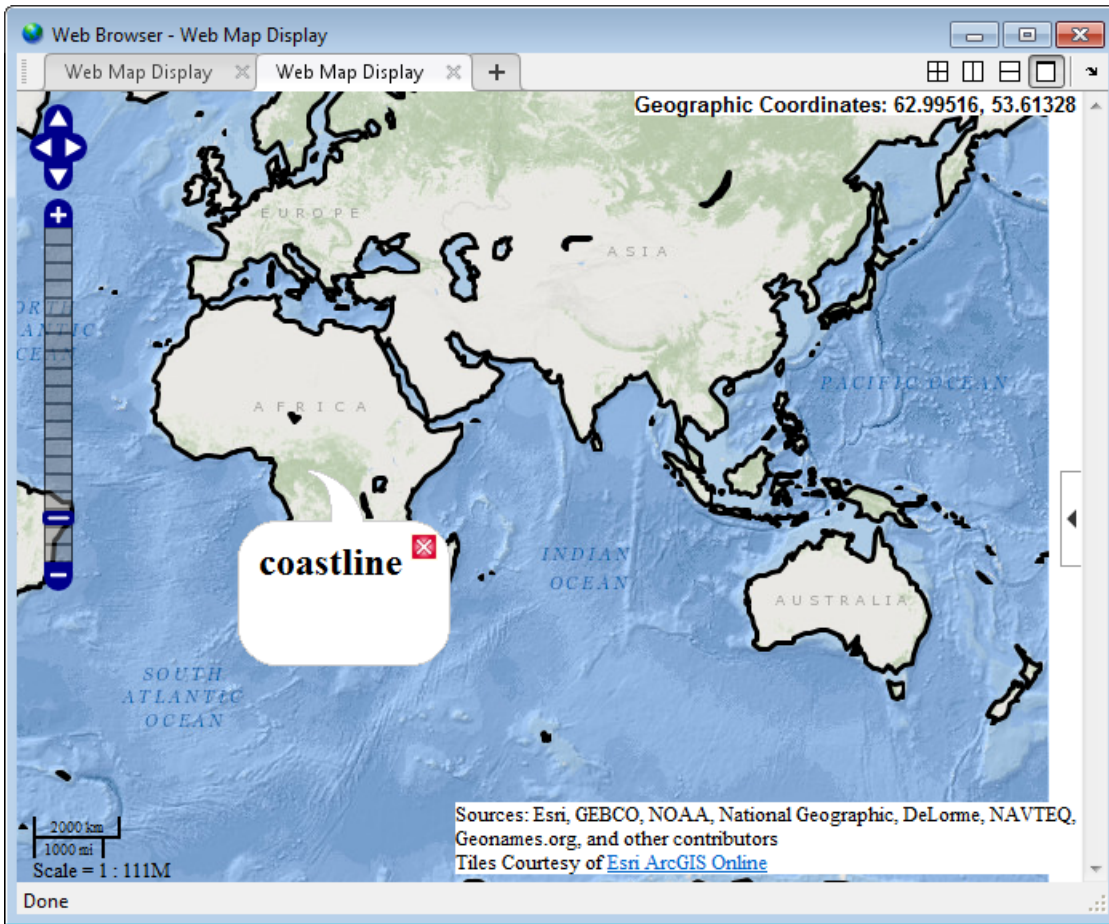


## Draw Circles on Web Map

This example shows how to draw lines on a web map, displaying range data for an airport approach pattern.

Assign latitude and longitude of the feature.

```
lat0 = 51.50487;
```

```
lon0 = .05235;
```

Create a web map using MapQuest OpenStreetMap layer.

```
webmap('MapQuestOpenStreetMap')
```
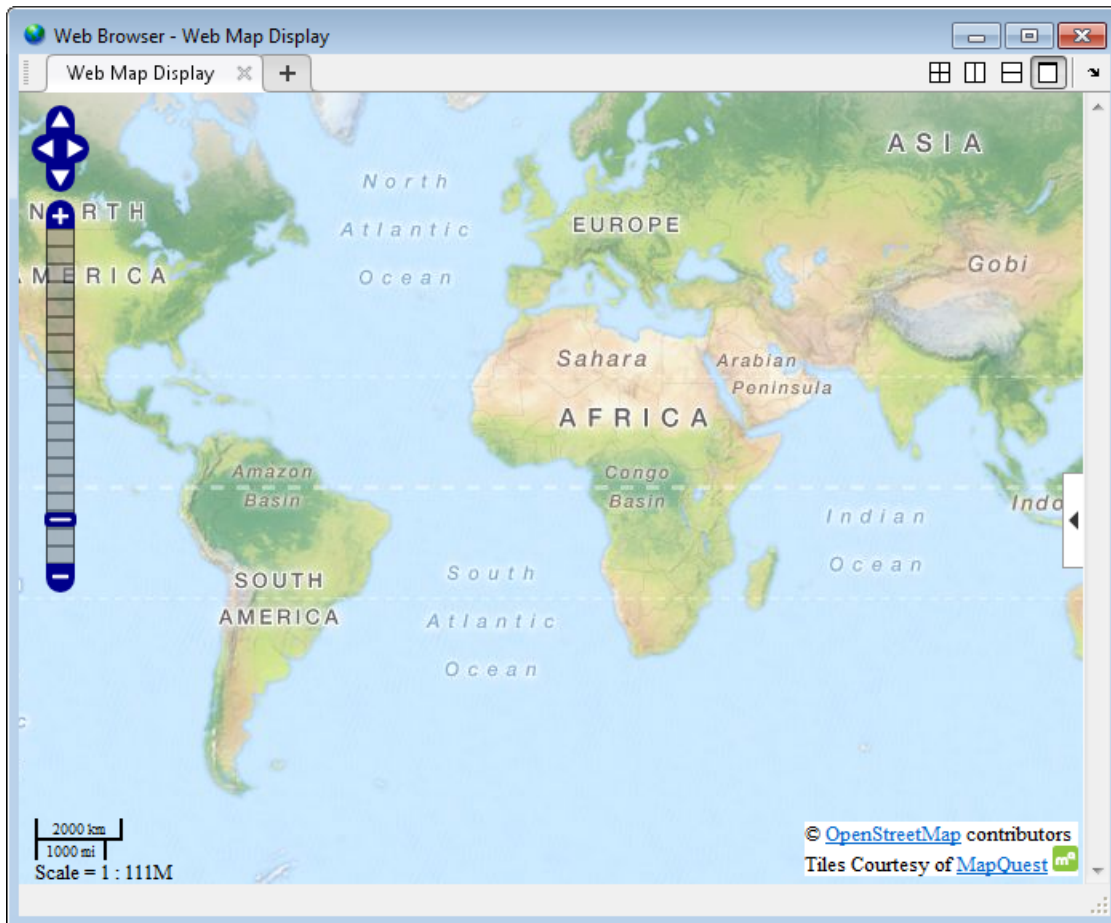


Compute small circle for 1000 meter radius.

```
radius = 1000;
az = [];
e = wgs84Ellipsoid;
[lat, lon] = scircle1(lat0, lon0, radius, az, e);
```

Draw red circle with 1000 meter radius.

```
wmline(lat, lon, 'Color', 'red', 'OverlayName', '1000 Meters');
```
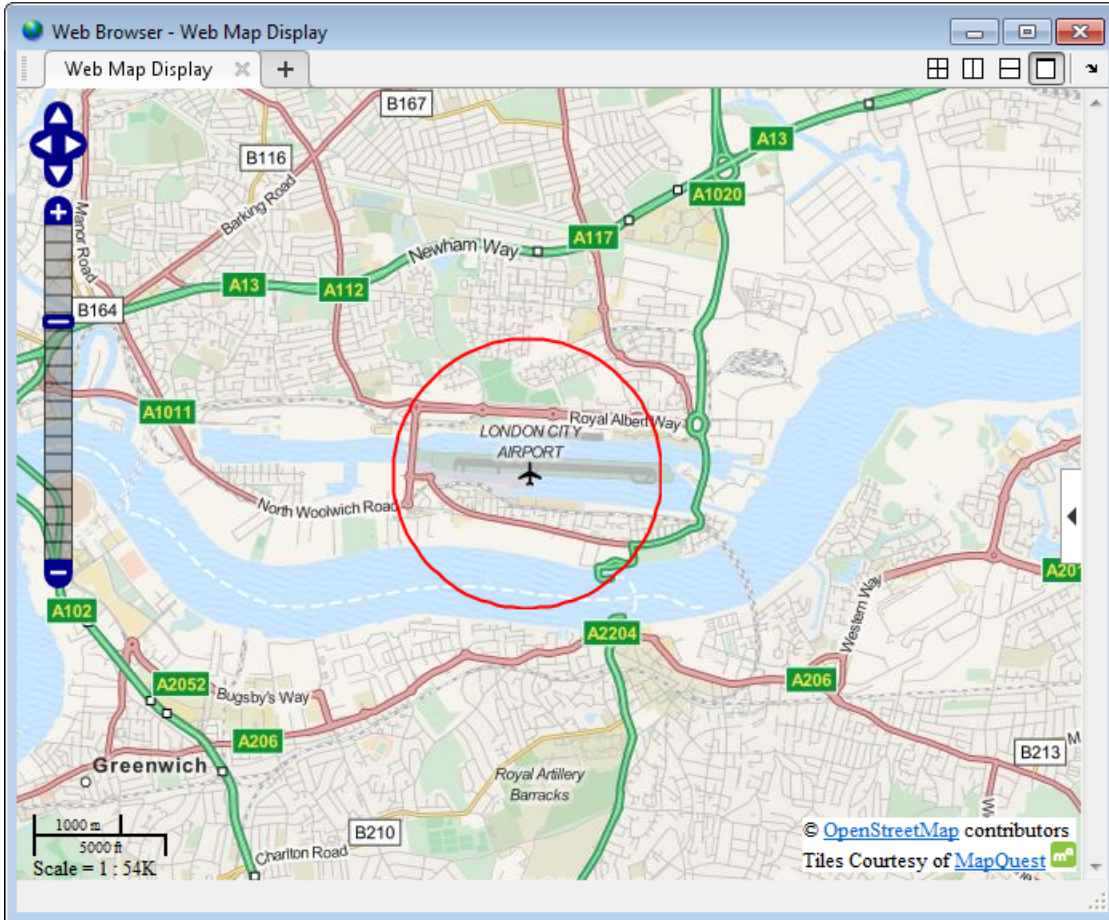


Compute small circle for 2000 meter radius.

```
radius = 2000;
[lat, lon] = scircle1(lat0, lon0, radius, az, e);
```

Draw black circle with 2000 meter radius.

```
wmline(lat, lon, 'Color', 'k', 'OverlayName', '2000 Meters');
```
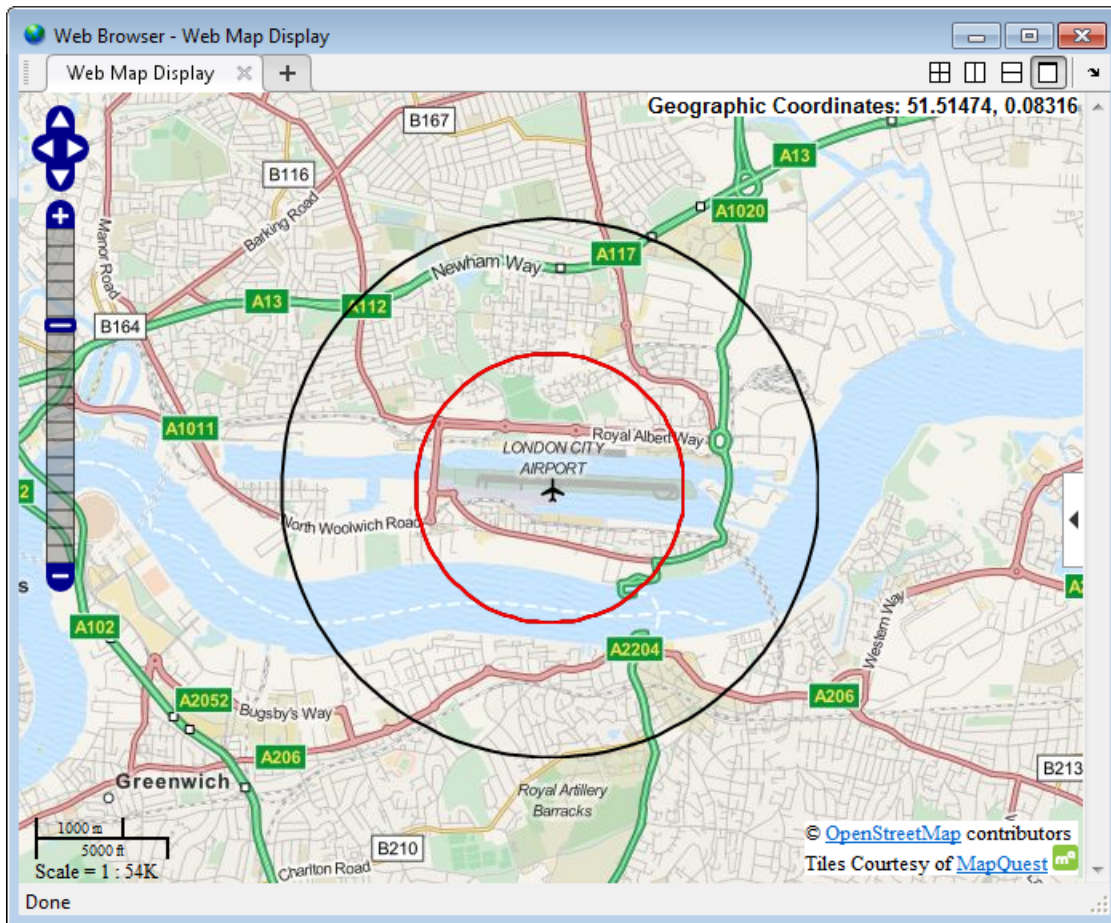
## Input Arguments

### `lat` — Latitude values
array of type `single` or `double`

Latitude values, specified as an array of type `single` or `double`.

Example: `lat = [42.3583 40.7142];`

Data Types: `single` | `double`

### `lon` — Longitude values

array of type `single` or `double`

Longitude values, specified as an array of type `single` or `double`.

Example: `lon = [-71.0603 −74.0064];`

Data Types: `single` | `double`

### p — Geographic features

geopoint vector | geoshape vector

Geographic features, specified as a geopoint or geoshape vector.

Example: `p = gpxread('sample_tracks', 'Index', 1:2);`

### wm — Web map

web map handle

Web map, specified as a web map handle.

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `wmline(lat,lon,'OverlayName','Shortest Route');`

### `'Autofit'` — Line visibility

true (default) | false

Line visibility, specified as the scalar logical or numeric value `true` (1) or `false` (0).

- If `true`, `wmline` adjusts the spatial extent of the map to ensure that all the vector overlays on the map are visible.
- If `false`, `wmline` does not adjust the spatial extent when this vector layer is added to the map.

Example: `wmline(lat,lon,'Autofit',true)`

Data Types: `double` | `logical`

**'Description'** — Description of feature
empty string (default) | character string | cell array of strings | scalar structure

Description of feature, specified as a character string, cell array of strings, or a scalar structure. The description defines the content that is displayed in the feature's description balloon which appears when a user clicks on the feature in the web map. Description elements can be either plain text or marked up with HTML. When an attribute spec is provided, the display in the balloon for the attribute fields of p are modified according to the specification. The default value is an empty string.

- If the value is a cell array, it must be either a scalar or the same length as p and specifies the description for each line.
- If the value is a structure, the attribute spec is applied to each line.

Data Types: char | struct | cell

**'OverlayName'** — Name of overlay layer
'Line Overlay *N*', (default) | text string

Name of overlay layer, specified as a text string. wmline inserts the name in the Layer Manager under the "Overlays" item. The Layer Manager is the tool that appears on the right side of the web map browser. The default name is 'Line Overlay *N*' where *N* is the number assigned to this overlay.

Example: wmline(lat,lon,'OverlayName','Route');

Data Types: char

**'FeatureName'** — Name of feature
'*OverlayName*: Line *K*' (default) | text string | cell array of strings

Name of feature, specified as text string or cell array of strings. The name appears in the feature's balloon when a user clicks on the feature in the web map. The default value is '*OverlayName* : Line *K*', where *OverlayName* is the name of the overlay and *K* is the number assigned to the particular line.

- If the value is a string, it applies to all features.
- If the value is a cell array of strings, it must be either a scalar or the same length as p.

Example: wmline(lat,lon,'FeatureName','Route 66');

Data Types: char | cell

**`'Color'` — Line color**
`'black'` (default) | ColorSpec | cell array of strings | *M*-by-3 numeric array

Line color, specified as a MATLAB Color Specification (ColorSpec), a cell array of color strings, or a numeric array.

- If the value is a cell array, it must be scalar or the same length as `p`.
- If the value is a numeric array, it must be an *M*-by-3 where *M* is either 1 or the length of `p`.

Example: `wmline(lat,lon,'Color','white');`

**`'Width'` — Line width**
1 (default) | positive numeric scalar or vector

Line width, specified as a positive numeric scalar or vector. If you specify a vector, it must include a value for each line, that is, the vector must be the same length as `p`.

Example: `wmline(lat,lon,'Width',2);`

# Output Arguments

**h — Overlay layer**
handle to line overlay

Overlay layer, returned as a handle.

# More About

### Tips

- Under certain conditions, when you zoom in on a line overlay in a web map, parts of the line may become invisible. This can occur if the data is one line segment that is composed of many parts. To workaround this issue, break the long line into a series of line segments by inserting NaNs in the line data.

# See Also
webmap | wmcenter | wmclose | wmlimits | wmmarker | wmprint | wmremove | wmzoom

# wmremove

Remove overlay on web map

## Syntax

```
wmremove()
wmremove(h)
```

## Description

`wmremove()` removes the overlay most recently inserted into the current web map.

`wmremove(h)` removes the overlay or overlays specified by the scalar overlay handle or vector of overlay handles `h`.

## Examples

### Remove a Marker Overlay

Draw a marker on a web map. `wmmarker` creates the web map. Pause, and then remove the marker overlay.

```
wmmarker(42, -73);
pause(1);
wmremove()
```

### Remove Multiple Overlays

Draw several marker overlays on a web map. `wmmarker` creates the web map. Pause, and then remove the marker overlays, specifying a vector of overlay handles.

```
h1 = wmmarker(42, -80);
h2 = wmmarker(42, -78);
pause(1);
wmremove([h1 h2])
```

### Remove Line Overlay

Create a web map.

```
wm = webmap();
```

Load coastline data and display it as an overlay on the webmap.

```
coast = load('coast');
h = wmline(coast.lat, coast.long);
```

Remove the overlay specified by h.

```
wmremove(h)
```

## Input Arguments

### h — Web map overlay
scale overlay handle or vector of overlay handles

Web map overlay, specified as a scalar overlay handle or a vector of overlay handles.

## See Also
webmap | wmcenter | wmclose | wmlimits | wmline | wmmarker | wmprint | wmzoom

# wmcenter

Set or obtain web map center point

## Syntax

```
wmcenter(centerLatitude,centerLongitude)
wmcenter(wm,centerLatitude,centerLongitude)
wmcenter( ___ ,zoomLevel)
[lat,lon] = wmcenter()
[lat,lon] = wmcenter(wm)
```

## Description

`wmcenter(centerLatitude,centerLongitude)` centers the current web map at the specified latitude and longitude. If there is no current web map, `wmcenter` creates a new web map.

`wmcenter(wm,centerLatitude,centerLongitude)` centers the web map, specified by the handle `wm`, at the specified latitude and longitude.

`wmcenter( ___ ,zoomLevel)` centers and zooms the web map to the specified zoom level.

`[lat,lon] = wmcenter()` returns the latitude and longitude of the center point of the current web map.

`[lat,lon] = wmcenter(wm)` returns the latitude and longitude of the center point of the web map specified by the handle `wm`.

## Examples

### Display web map and center it

Display a map a find it's center point. `wmcenter` creates a web map if none exists.

```
[centerLatitude, centerLongitude] = wmcenter()

centerLatitude =

     0


centerLongitude =

     0
```
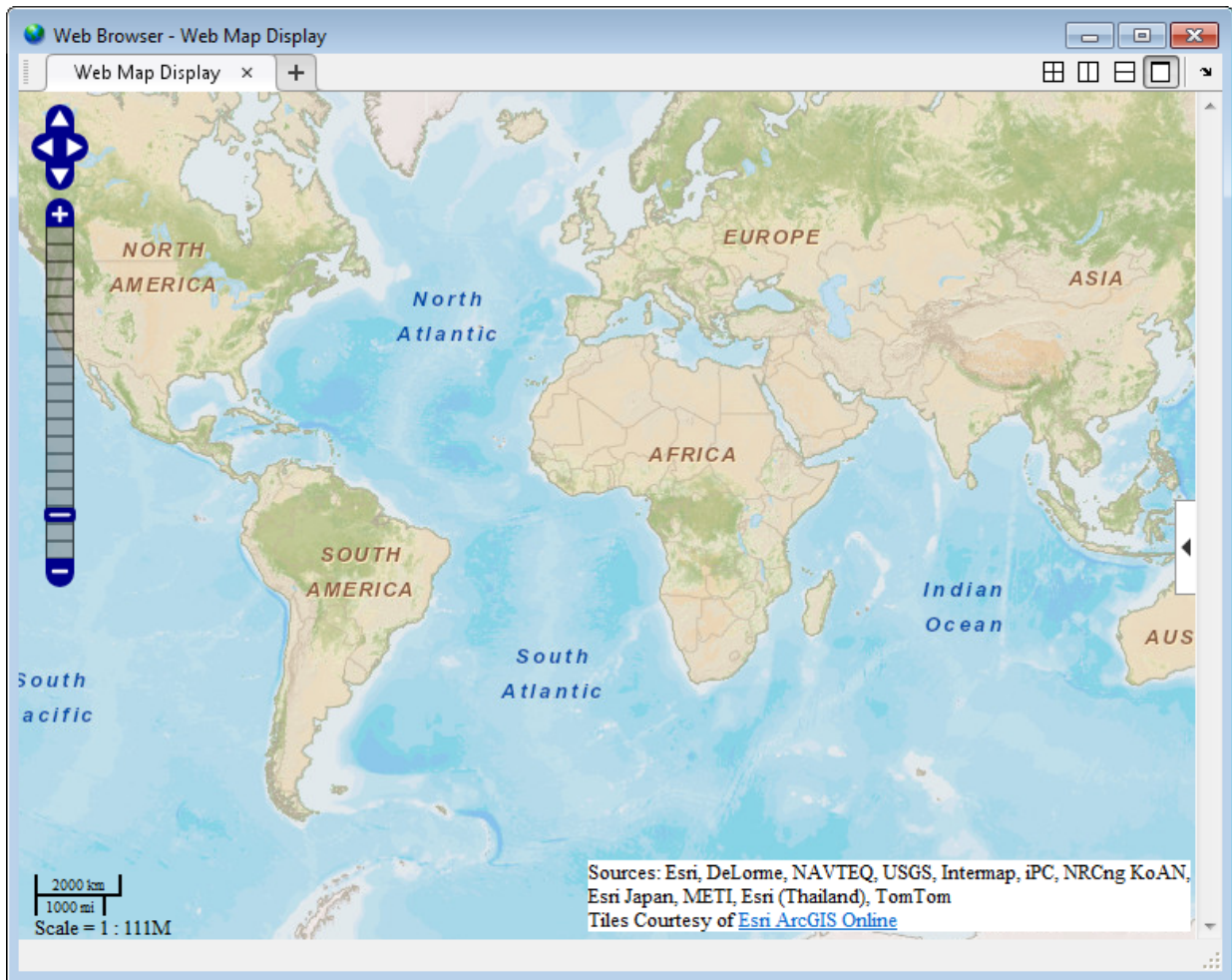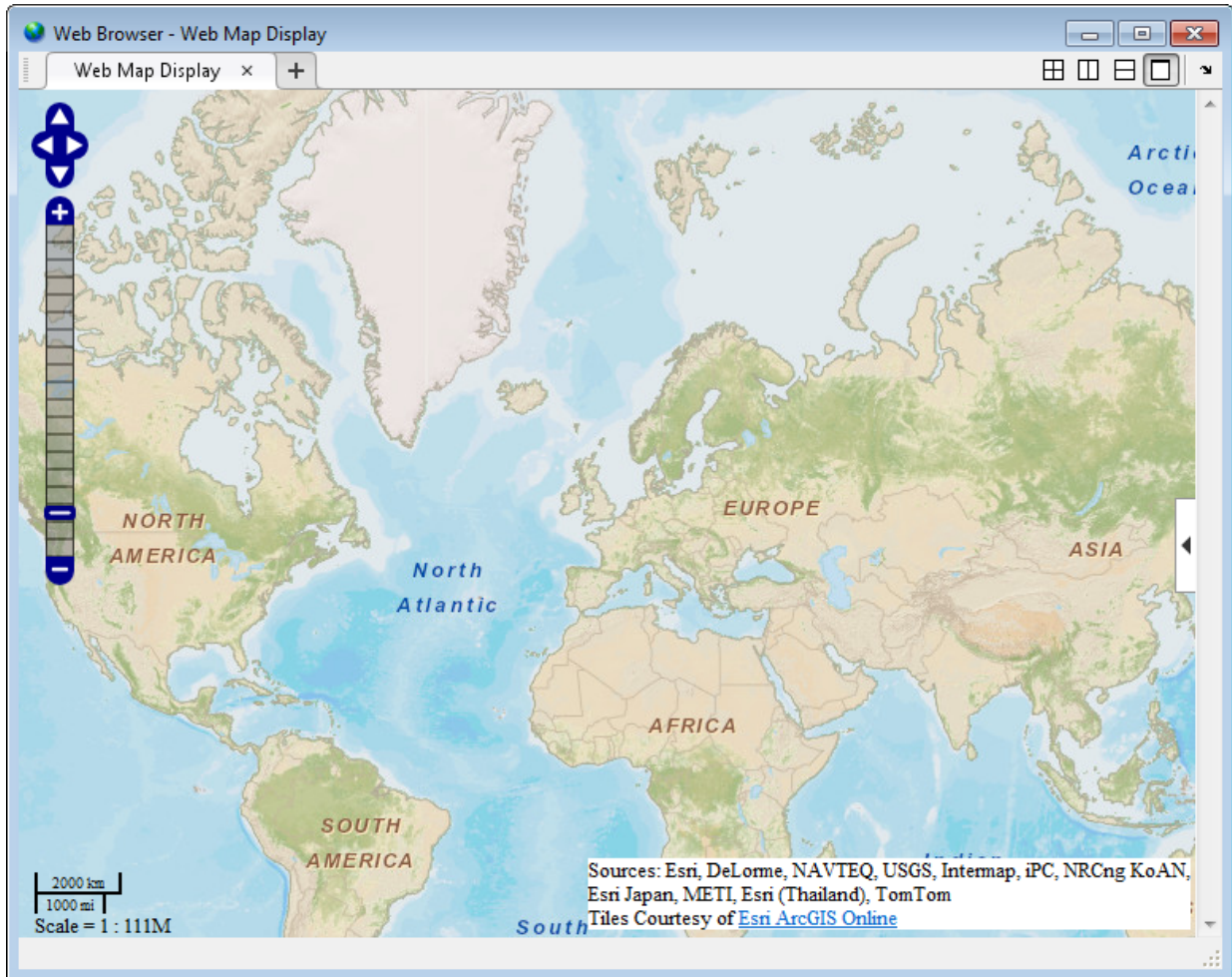
Center the map at the specified center point.

```
wmcenter(51.52, 0)
```



**Display multiple web maps centered and zoomed**

Create two web maps.

```
wm1 = webmap;
```

```
wm2 = webmap('mapquest open aerial map');
```



Center the two web maps, specifying the zoom level as well.

```
centerLatitude = 36.1;
centerLongitude = -113.2;
zoomLevel = 10;
wmcenter(wm1, centerLatitude, centerLongitude, zoomLevel)
wmcenter(wm2, centerLatitude, centerLongitude, zoomLevel)
```

## Input Arguments

**`centerLatitude` — Latitude of center point**
scalar in the range [`-90 90`] of type `single` or `double`

Latitude of center point, specified as a scalar in the range [`-90 90`] of type `single` or `double`.

Data Types: `single` | `double`

### centerLongitude — Longitude of center point
scalar in the range `[-180 180]` of type `single` or `double`

Longitude of center point, specified as a scalar in the range `[-180 180]` of type `single` or `double`.

Data Types: `single` | `double`

### wm — Web map
web map handle

Web map, specified as a web map handle.

### zoomLevel — Zoom level
scalar numeric integer in the range `[0 18]` of type `single` or `double`

Zoom level, specified as a scalar numeric integer in the range `[0 18]` of type `single` or `double`.

Data Types: `single` | `double`

## Output Arguments

### lat — Latitude of center point
scalar in the range `[-90 90]` of type `single` or `double`

Latitude of center point, returned as a scalar in the range `[-90 90]` of type `single` or `double`.

Data Types: `single` | `double`

### lon — Longitude of center point
scalar in the range `[-180 180]` of type `single` or `double`

Longitude of center point, returned as a scalar in the range `[-180 180]` of type `single` or `double`.

Data Types: `single` | `double`

# More About

**Tips**

- Particular maps may not support every available zoom level. If your map displays as completely white, try another zoom level. The map you are displaying may not support the zoom level you have currently selected. You can also select another base layer, which might support the specified zoom level.

## See Also
webmap | wmclose | wmlimits | wmline | wmmarker | wmprint | wmremove | wmzoom

# wmzoom

Set or obtain zoom level of web map

# Syntax

```
wmzoom(zoomLevel)
wmzoom(wm,zoomLevel)
zoomLevelOut = wmzoom()
zoomLevelOut = wmzoom(wm)
```

# Description

wmzoom(zoomLevel) specifies the zoom level of the current web map, zoomLevel. If there is no current web map, wmzoom creates a new web map.

wmzoom(wm,zoomLevel) specifies the zoom level of the web map, specified by the handle wm.

zoomLevelOut = wmzoom() returns the zoom level of the current web map.

zoomLevelOut = wmzoom(wm) returns the zoom level of the web map specified by the handle wm.

# Examples

### Get Zoom Level of Current Web Map and Specify New Zoom Level

Create a web map at default zoom level.

```
zoomLevel = wmzoom()

zoomLevel =

     0
```

Center the web map over London and zoom in.

```
wmcenter(51.52, 0)
wmzoom(10)
```

### Specify Zoom Level of Several Web Maps

Create two web maps.

```
wm1 = webmap;
wm2 = webmap('worldtopographic');
```

Center both maps at a specified latitude and longitude (Paris), and then zoom in.

```
lat = 48.821;
lon = 1.9391;
zoomLevel = 10;
wmcenter(wm1, lat, lon)
wmzoom(wm1, zoomLevel)
wmcenter(wm2, lat, lon)
wmzoom(wm2, zoomLevel)
```

# Input Arguments

### `zoomLevel` — Zoom level
scalar numeric integer in the range [0 18]

Zoom level, specified as a scalar numeric integer in the range [0 18].

Data Types: `single` | `double`

### `wm` — Web map
web map handle

Web map, specified as a web map handle.

# Output Arguments

### `zoomLevelOut` — Zoom level of the current web map
scalar numeric integer in the range [0 18]

Zoom level of the current web map, returned as a scalar numeric integer in the range [0 18].

Data Types: `single` | `double`

## More About

**Tips**

- Particular maps may not support every available zoom level. If your map displays as completely white, try another zoom level. The map you are displaying may not support the zoom level you have currently selected. You can also select another base layer, which might support the specified zoom level.

## See Also

`webmap` | `wmcenter` | `wmclose` | `wmlimits` | `wmline` | `wmmarker` | `wmprint` | `wmremove`

# wmlimits

Set or obtain web map limits

## Syntax

```
wmlimits(latitudeLimits,longitudeLimits)
wmlimits(wm,latitudeLimits,longitudeLimits)
[latlim,lonlim] = wmlimits()
[latlim,lonlim] = wmlimits(wm)
```

## Description

`wmlimits(latitudeLimits,longitudeLimits)` center the current web map within the specified latitude limits and the longitude limits. If there is no current web map, `wmlimits` creates one.

---

**Note:** The resulting limits often do not match the specified limits because the zoom level is quantized to discrete integer values and the longitude limits may be constrained if the map was constructed with the `WrapAround` property equal to `false`.

---

`wmlimits(wm,latitudeLimits,longitudeLimits)` centers the web map specified by the web map handle `wm` within the specified latitude limits and longitude limits.

`[latlim,lonlim] = wmlimits()` returns the latitude and longitude limits of the current web map.
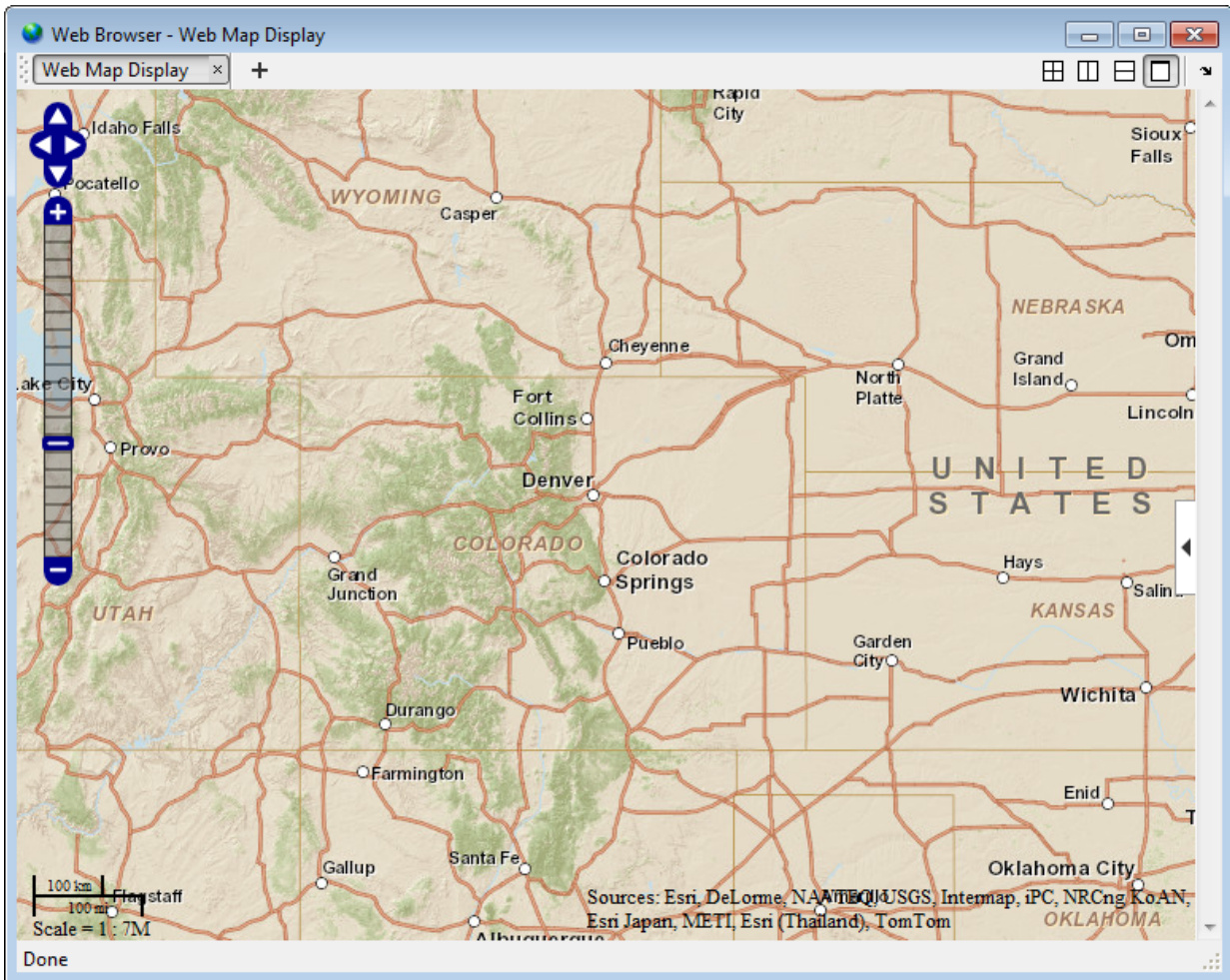
`[latlim,lonlim] = wmlimits(wm)` returns the latitude and longitude limits of the web map specified by `wm`.

## Examples

### Display Web Map Centered Within Specified Limits

Specify latitude and longitude limits. `wmlimits` creates the web map.

```
wmlimits([37, 42], [-108.9, -100.7])
```



Get the latitude and longitude limits of the current web map. (Note: The limits returned depend on the size of the web map frame and the zoom level. Your results may differ from the results shown in this example.)

```
[latitudeLimits, longitudeLimits] = wmlimits()
```

```
latitudeLimits =
```

```
   34.8664    43.9282


longitudeLimits =

 -112.9519   -96.6481
```

### Display Several Web Maps Centered Within Specified Limits

Create two web maps, specifying different base layers.

```
wm1 = webmap;
wm2 = webmap('worldtopographic');
```

Specify latitude and longitude limits and apply to both maps.

```
latitudeLimits = [37, 42];
longitudeLimits = [-108.9, -100.7];
wmlimits(wm1, latitudeLimits, longitudeLimits)
wmlimits(wm2, latitudeLimits, longitudeLimits)
```
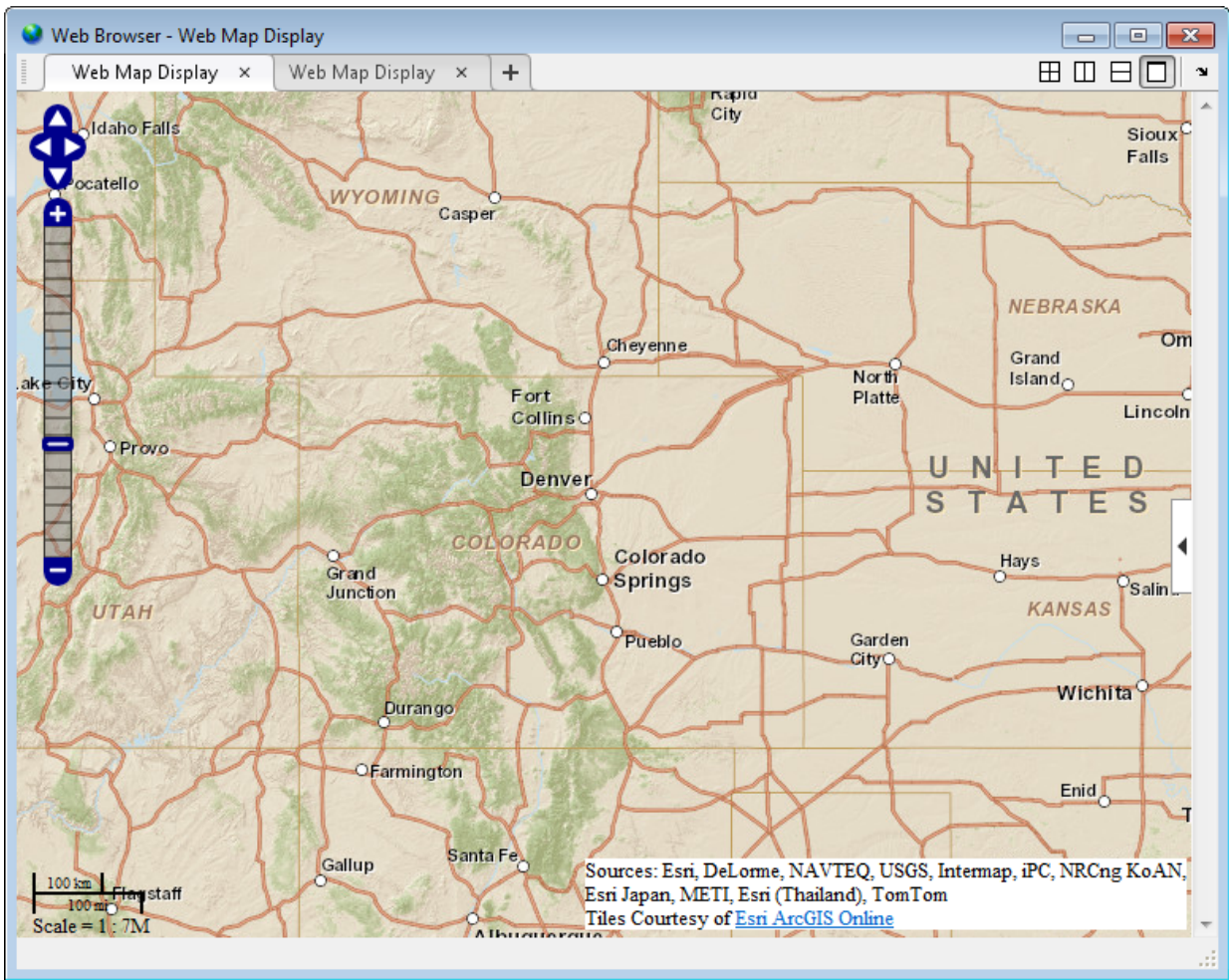
## Input Arguments

**`latitudeLimits` — Latitude limits in degrees**
1-by-2 row vector of type `double`

Latitude limits in degrees, specified as a 1-by-2 row vector of type `double` of the form
`[southern-limit northern-limit]`.

Example: [37, 42]

Data Types: double

### **longitudeLimits** — Longitude limits in degrees
1-by-2 row vector of type double

Longitude limits in degrees, specified as a 1-by-2 row vector of type double of the form [western-limit eastern-limit].

Example: [-108.9, -100.7]

Data Types: double

### **wm** — Web map
scalar web map handle

Web map, specified as a scalar web map handle.

## Output Arguments

### **latlim** — Latitude limits in degrees
1-by-2 row vector of type double

Latitude limits in degrees, returned as a 1-by-2 row vector of type double.

Example: [37, 42]

Data Types: double

### **lonlim** — Longitude limits in degrees
1-by-2 row vector of type double

Longitude limits in degrees, returned as a 1-by-2 row vector of type double.

Example: [-108.9, -100.7]

Data Types: double

## See Also
webmap | wmcenter | wmclose | wmline | wmmarker | wmprint | wmremove | wmzoom

# WMSCapabilities class

Web Map Service capabilities object

## Description

A `WMSCapabilities` object represents a Web Map Service (WMS) capabilities document obtained from a WMS server.

## Construction

`capabilities = WMSCapabilites(ServerURL, capabilitiesResponse)` constructs a `WMSCapabilities` object from the input string parameters. The `ServerURL` string, a WMS server URL, includes the protocol `'http://'` or `'https://'`. The `capabilitiesResponse` string contains XML elements that describe the capabilities of the `ServerURL` WMS server.

## Properties

**ServerTitle**

Title of server

**Data Type:** string

**ServerURL**

URL of server

**Data Type:** string

**ServiceName**

Name of Web map service

**Data Type:** string

**Version**

WMS version specification

**Data Type:** string

**Abstract**

Information about server

**Data Type:** string

**OnlineResource**

Online information about server

**Data Type:** string (URL)

**ContactInformation**

Contact information for an individual or an organization, including an email address, if provided

**Data Type:** structure

### ContactInformation Structure Array

| Field Name | Data Type | Field Content |
|---|---|---|
| Person | String | Name of individual |
| Organization | String | Name of organization |
| Email | String | Email address |

**AccessConstraints**

Constraints inherent in accessing the server, such as server load limits

**Data Type:** string

**Fees**

Types of fees associated with accessing server

**Data Type:** string

**KeywordList**

Descriptive keywords of the server

**Data Type:** cell array of strings

**ImageFormats**

Image formats supported by server

**Data Type:** cell array of strings

**LayerNames**

Layer names provided by server

**Data Type:** cell array of strings

**Layer**

Information about layers on WMS server. See the `WMSCapabilities.Layer` reference page for more information.

**Data Type:** `WMSLayer` array

**AccessDate**

Date of request to server

**Data Type:** string

## Methods

disp                                                    Display properties

## Examples

Construct a `WMSCapabilities` object from the contents of a downloaded capabilities file from the NASA SVS Image Server:

```
nasa = wmsfind('NASA SVS Image', 'SearchField', 'servertitle');
serverURL = nasa(1).ServerURL;
server = WebMapServer(serverURL);
capabilities = server.getCapabilities;
filename = 'capabilities.xml';
urlwrite(server.RequestURL, filename);

fid = fopen(filename, 'r');
capabilitiesResponse = fread(fid, 'uint8=>char');
fclose(fid);
capabilities = WMSCapabilities(serverURL, capabilitiesResponse);
```

## See Also
WMSLayer | WebMapServer | wmsinfo

# disp

**Class:** WMSCapabilities

Display properties

## Syntax

```
disp(capabilities)
```

## Description

`disp(capabilities)` displays the properties of the WMSCapabilities object, `capabilities`. The method removes hyperlinks and expands string and cell array properties.

# Layer property

**Class:** WMSCapabilities

Layer information

## Description

A `WMSLayer` array containing information about the layers available on a WMS server.

| Property Name | Data Type | Property Content |
| --- | --- | --- |
| ServerTitle | String | Descriptive title of server |
| ServerURL | String | URL of server |
| LayerTitle | String | Descriptive title of layer |
| LayerName | String | Name of layer |
| Latlim | Double array | Southern and northern latitude limits of layer |
| Lonlim | Double array | Western and eastern longitude limits of layer |
| Abstract | String | Information about layer |
| CoordRefSysCodes | Cell array | Codes of available coordinate reference systems |
| WMSLayer.Details | Structure | Detailed information about layer |

# wmsfind

Search local database for Web map servers and layers

## Syntax

```
layers = wmsfind(querystr)
layers = wmsfind(querystr, Name,Value, ...)
```

## Description

`layers = wmsfind(querystr)` searches the `LayerTitle` and `LayerName` fields of the installed Web Map Service (WMS) Database for partial matches with the string *querystr*. WMS servers, by definition, produce maps of spatially referenced raster data. You can search for specific types of data, known as layers, such as temperature or elevation. The string *querystr* can contain the wildcard character `'*'`. The array returned by `wmsfind`, `layers`, contains one element for each layer whose name or title partially matches *querystr*. Each element is a `WMSLayer` object.

The installed WMS Database contains a subset of the `WMSLayer` properties (`ServerTitle`, `ServerURL`, `LayerTitle`, `LayerName`, `Latlim`, and `Lonlim`). The information found in the database is static and is not automatically updated; it was validated at the time of the software release.

`layers = wmsfind(querystr, Name,Value, ...)` modifies the search of the WMS database based on the values of the parameters. You can abbreviate parameter names, and case does not matter.

## Input Arguments

**querystr**

Specifies the search string, such as `'temperature'`

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**`'IgnoreCase'`**

Logical that specifies whether to ignore case when performing string comparisons. Possible values are `true` or `false`.

**Default:** `true`

**`'Latlim'`**

Two-element vector of latitude specifying the latitudinal limits of the search in the form `[southern_limit northern_limit]` or scalar value representing the latitude of a single point. All angles are in units of degrees.

If provided and not empty, a given layer appears in the results only if its limits fully contain the specified `'Latlim'` limits. Partial overlap does not result in a match.

**`'Lonlim'`**

Two-element vector of longitude specifying the longitudinal limits of the search in the form `[western_limit eastern_limit]` or scalar value representing the longitude of a single point. All angles are in units of degrees.

If provided and not empty, a given layer appears in the results only if its limits contain the specified `'Lonlim'` limits. Partial overlap does not result in a match.

**`'MatchType'`**

String with value `'partial'` or `'exact'`. For a partial string match, specify `'partial'` for the `'MatchType'`. For an exact match, specify `'exact'`. If `'MatchType'` is `'exact'` and `querystr` is `'*'`, a match occurs when the search field matches the character `'*'`.

**Default:** `'partial'`

**'SearchFields'**

String or cell array of strings. Valid strings are `'layer'`, `'layertitle'`, `'layername'`, `'server'`, `'serverurl'`, `'servertitle'`, or `'any'`.

The function searches the entries in the `'SearchFields'` of the WMS database for a partial match with `querystr`. If you specify `'layer'`, then `wmsfind` searches both the `'layertitle'` and `'layername'` fields. If you specify `'server'`, then `wmsfind` searches both the `'serverurl'` and `'servertitle'` fields. The function returns layer information if any supplied `'SearchFields'` match.

**Default:** {`'layer'`}

# Output Arguments

**layers**

Array that contains one element for each layer whose name or title partially matches *querystr*. Each element is a `WMSLayer` object.

# Examples

Find all layers that contain temperature data and return a `WMSLayer` array:

```
layers = wmsfind('temperature');
```

Find all layers that contain global temperature data and return a `WMSLayer` array:

```
layers = wmsfind('global*temperature');
```

Find all layers that contain an exact match for `'Rivers'` in the `LayerTitle` field and return a `WMSLayer` array:

```
layers = wmsfind('Major Rivers', 'MatchType', 'exact', ...
   'IgnoreCase', false, 'SearchFields', 'layertitle');
```

Find all layers that contain a partial match for `'elevation'` in the `LayerName` field and return a `WMSLayer` array:

```
layers = wmsfind('elevation', 'SearchField', 'layername');
```

Find all unique servers that contain `'BlueMarbleNG'` as a layer name:

```
layers = wmsfind('BlueMarbleNG', ...
    'SearchField', 'layername', 'MatchType', 'exact');
servers = layers.servers;
```

Find layers that contain elevation data for Colorado and return a `WMSLayer` array:

```
latlim = [35,43];
lonlim = [-111,-101];
layers = wmsfind('elevation', ...
    'Latlim', latlim, 'Lonlim', lonlim);
```

Find all layers that contain temperature data for a point in Perth, Australia, and return a `WMSLayer` array:

```
lat = -31.9452;
lon = 115.8323;
layers = wmsfind('temperature', 'Latlim', lat, 'Lonlim', lon);
```

Find all the layers provided by servers located at the Jet Propulsion Laboratory (JPL). Display to the command window each server URL, layer title, and layer name:

```
layers = wmsfind('jpl.nasa.gov', 'SearchField', 'serverurl');
layers.disp('Properties', {'serverURL', 'layerTitle','layerName'});
```

Find all unique URLs of government servers:

```
layers = wmsfind('*.gov*', 'SearchField', 'serverurl');
servers = layers.servers;
```

Perform multiple searches. Find all layers that contain temperature in the layer name or title fields:

```
temperature = wmsfind('temperature', ...
    'SearchField',{'layertitle', 'layername'});
```

Find sea surface temperature layers:

```
sst = temperature.refine('sea surface');
```

Find and display to the command window a list of global sea surface temperature layers:

```
global_sst = sst.refine('global')
```

Perform multiple listings and searches of the entire WMS database. Please note that finding all the layers from the WMS database may take several seconds to execute and require a substantial amount of memory.

```
layers = wmsfind('*');
```

Sort and display to the command window the unique layer titles in the WMS database:

```
layerTitles = sort(unique({layers.LayerTitle}))'
```

Refine `layers` to include only layers with global coverage:

```
global_layers = layers.refineLimits('Latlim', [-90 90], ...
    'Lonlim', [-180 180]);
```

Refine `global_layers` to contain only topography layers that have global extent:

```
topography = global_layers.refine('topography');
```

Refine `layers` to contain only layers that have the terms "oil" and "gas" in the `LayerTitle`:

```
oil_gas = layers.refine('oil*gas', 'SearchField', 'layertitle');
```

## More About

### Tips

- The WMS Database does not store content for the properties `'Abstract'`, `'CoordRefSysCodes'`, and `'Details'`. Therefore, you cannot use `wmsfind` to search these properties. Populate these fields by using the `wmsupdate` function. This function updates these properties by downloading information from the server. The `WMSLayer.disp` method does not automatically display these unpopulated properties. Set the `WMSLayer.disp` `'Properties'` parameter to `'all'` to view. After you have viewed the information available from `wmsupdate`, if you still want to know more about the WMS server, use the function `wmsinfo` with the specific server URL.

## See Also

| wmsread | wmsupdate | wmsinfo

# wmsinfo

Information about WMS server from capabilities document

## Syntax

```
[capabilities, infoRequestURL] = wmsinfo(serverURL)
[capabilities, infoRequestURL] = wmsinfo(infoRequestURL)
[capabilities, infoRequestURL] = wmsinfo( ___ ,Name,Value)
```

## Description

`[capabilities, infoRequestURL] = wmsinfo(serverURL)` accesses the Internet to read a capabilities document from a Web Map Service (WMS) server. A capabilities document is an XML document that contains metadata describing the geographic content offered by the server. The `wmsinfo` function returns the contents of the capabilities document into `capabilities`, a `WMSCapabilities` object. The WMS server URL `serverURL` contains the protocol `'http://'` or `'https://'` and additional WMS or access keywords. You can insert the URL string `infoRequestURL`, composed of the `serverURL` with additional WMS parameters, into a browser or `urlread` to return the XML capabilities document. The `wmsinfo` function requires an Internet connection. Periodically, the WMS server is unavailable. Retrieving the map can take several minutes.

`[capabilities, infoRequestURL] = wmsinfo(infoRequestURL)` reads the capabilities document from a WMS `infoRequestURL` and returns the contents into `capabilities`.

`[capabilities, infoRequestURL] = wmsinfo( ___ ,Name,Value)` specifies a parameter-value pair that modifies the request to the server.

## Input Arguments

**serverURL**

WMS server URL that contains the protocol `'http://'` or `'https://'` and additional WMS or access keywords.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`'  '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

You can abbreviate the parameter name, which is case-insensitive.

**`'TimeoutInSeconds'`**

Integer-valued, scalar double that indicates the number of seconds to elapse before a server times out. A value of 0 causes the time-out mechanism to be ignored.

**Default:** 60 seconds

## Output Arguments

**`capabilities`**

`WMSCapabilities` object.

**Default:**

**`infoRequestURL`**

URL string composed of the `serverURL` with additional WMS parameters.

## Examples

Use `wmsinfo` to read a capabilities document and display the abstract of the first layer.

```
% Read the capabilities document from the NASA Goddard
% Space Flight Center WMS server.
serverURL = 'http://svs.gsfc.nasa.gov/cgi-bin/wms?';
capabilities = wmsinfo(serverURL);

% Display the layer information in the command window.
capabilities.Layer
```

Sample output follows:

```
        Index: 304
    ServerTitle: 'NASA SVS Image Server'
      ServerURL: 'http://svs.gsfc.nasa.gov/cgi-bin/wms?'
     LayerTitle: '(4096x2048 Animation)'
      LayerName: '3348_27724'
         Latlim: [-90.0000 90.0000]
         Lonlim: [-180.0000 180.0000]
       Abstract: 'NASA's Aqua satellite was launched ...
CoordRefSysCodes: {'CRS:84'}
        Details: [1x1 struct]


% Refine the list to include only layers with the term
% "glacier retreat" in the LayerTitle.
glaciers = capabilities.Layer.refine('glacier retreat', ...
   'SearchFields', 'LayerTitle');

% Display the abstract of the first layer.
glaciers(1).Abstract
```

Sample output follows:

```
Since measurements of Jakobshavn Isbrae were first taken....
```

# More About

### Tips

- To specify a proxy server to connect to the Internet, select **File>Preferences>Web** and enter your proxy information. Use this feature if you have a firewall.

- `wmsinfo` communicates with the server using a `WebMapServer` handle object representing an implementation of a WMS specification. The handle object acts as a proxy to a WMS server and resides physically on the client side. The handle object accesses the server's capabilities document. The handle object supports multiple WMS versions and negotiates with the server to use the highest known version that the server can support. The handle object automatically times-out after 60 seconds if a connection is not made to the server.

## See Also

`WMSLayer` | `wmsread` | `WebMapServer` | `WMSCapabilities` | `wmsfind`

# WMSLayer class

Web Map Service layer object

## Description

A `WMSLayer` object describes a Web Map Service (WMS) layer or layers. Obtain a `WMSLayer` object by using `wmsfind` or `wmsinfo`. The function `wmsfind` returns a `WMSLayer` array. The function `wmsinfo` returns a `WMSCapabilities` object, which contains a `WMSLayer` array in its `Layer` property.

## Construction

`layers = WMSLayer(param, val, ...)` constructs a `WMSLayer` object from the input parameter names and values. If a parameter name matches a property name of the `WMSLayer` class (`ServerTitle`, `ServerURL`, `LayerTitle`, `LayerName`, `Latlim`, `Lonlim`, `Abstract`, `CoordRefSysCodes`, or `Details`) then the values of the parameter are copied to the property. The size of the output `layers` is scalar unless all inputs are cell arrays, in which case, the size of `layers` matches the size of the cell arrays.

## Properties

You can only set the `'Latlim'` and `'Lonlim'` properties, which have public set access.

**ServerTitle**

Descriptive information about the server

**Data Type:** string

**Default:** `''`

**ServerURL**

The URL of the server

**Data Type:** string

**Default:** ' '

### LayerTitle

Descriptive information about the layer; clarifies the meaning of the raster values of the layer

**Data Type:** string

**Default:** ' '

### LayerName

The keyword the server uses to retrieve the layer

**Data Type:** string

**Default:** ' '

### Latlim

The southern and northern latitude limits of the layer in units of degrees and in the range [-90, 90].

**Data Type:** two-element vector

**Default:** [ ]

### Lonlim

The western and eastern longitude limits of the layer in units of degrees. The limits must be ascending and in the range [-180, 180] or [0 360].

**Data Type:** two-element vector

**Default:** [ ]

### Abstract

Information about the layer

**Data Type:** string

**Default:** ''

**CoordRefSysCodes**

String codes of available coordinate reference systems

**Data Type:** cell array

**Default:** {}

**Details**

Detailed information about the layer: MetadataURL, Attributes, Scale, Dimension, Style. See the `WMSLayer.Details` reference page for more information.

**Data Type:** structure

# Methods

| | |
|---|---|
| disp | Display properties |
| refine | Refine search |
| refineLimits | Refine search based on geographic limits |
| servers | Return URLs of unique servers |
| serverTitles | Return titles of unique servers |

# Examples

Construct a `WMSLayer` object from a WMS `GetMap` request URL:

```
serverURL = ['http://ims.cr.usgs.gov:80/wmsconnector/' ...
    'com.esri.wms.Esrimap/USGS_EDC_LandCover_NLCD2001?'];
requestURL = [serverURL 'SERVICE=WMS&FORMAT=image/jpeg&' ...
    'REQUEST=GetMap&' ...
    'STYLES=&SRS=EPSG:4326&VERSION=1.1.1&', ...
    'LAYERS=NLCD_2001_Land_Cover&', ...
    'WIDTH=1024&HEIGHT=470&' ...
    'BBOX=-128,23,-65,51&'];

% Construct the WMSLayer object by using the serverURL
```

```
% variable and the value of the WMS LAYERS parameter.
% Update the remaining information from the server.
layer = WMSLayer('ServerURL', serverURL, ...
    'LayerName', 'NLCD_2001_Land_Cover');
layer = wmsupdate(layer)

% Request the layer from the server using the WMS request
% parameters found in the requestURL string. Copy the WMS
% BBOX information to the latlim and lonlim parameters.
% Copy the WMS WIDTH and HEIGHT values to the ImageWidth and
% ImageHeight parameters.
lonlim = [-128, -65];
latlim = [23, 51];
[A, R] = wmsread(layer, 'Latlim', latlim, 'Lonlim', lonlim, ...
    'ImageHeight', 470 , 'ImageWidth', 1024);

% Display the rendered image.
figure
usamap(A, R)
geoshow(A, R)
title(layer.LayerTitle, 'Interpreter', 'none')

% The image can also be retrieved using the requestURL.
[A, R] = wmsread(requestURL);
```
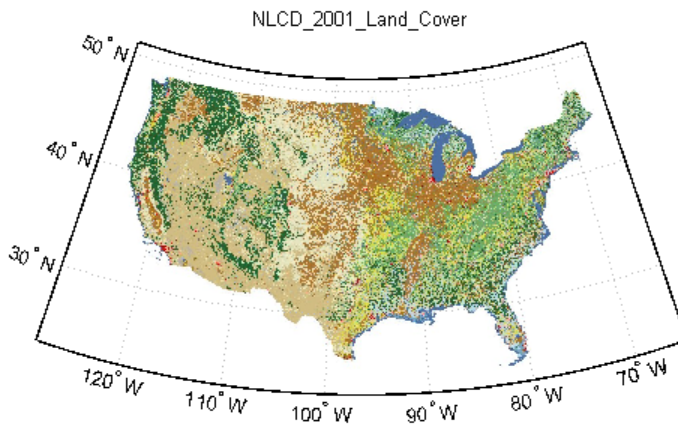


Courtesy U.S. Geological Survey

## See Also

WebMapServer | WMSCapabilities | wmsfind | wmsinfo | WMSMapRequest | wmsread | wmsupdate

# disp

**Class:** WMSLayer

Display properties

## Syntax

```
disp(layers,Name,Value,...)
```

## Description

disp(layers,Name,Value,...) displays the index number followed by the property names and property values of layers, a Web Map Service Layer object (WMSLayer). You can specify additional options using one or more Name,Value pair arguments.

## Input Arguments

### **layers** — Web Map Service layers
Web Map Service Layer (WMSLayer) object

Web Map Service layers, specified as a Web Map Service Layer (WMSLayer) object.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

You can abbreviate parameter names, and case does not matter.

### **'Properties'**

String or cell array of strings that determines which properties appear as output and the order in which they appear. Permissible values are: 'servertitle',

'servername', 'layertitle', 'layername', 'latlim', 'lonlim', 'abstract', 'coordrefsyscodes', 'details', or 'all'. To list all the properties, set 'Properties' to 'all'.

**Default:** 'all'

**'Label'**

A case-insensitive string with permissible values of 'on' or 'off'. If you set 'Label' to 'on', then the property name appears followed by its value. If you set 'Label' to 'off', then only the property value appears in the output.

**Default:** 'on'

**'Index'**

A case-insensitive string with permissible values of 'on' or 'off'. If you set 'Index' to 'on', then WMSLayer.disp lists the element's index in the output. If you set 'Index' to 'off', then WMSLayer.disp does not list the index value in the output.

**Default:** 'on'

## Examples

### Display Specific Properties of WMSLayer Object

Display LayerTitle and LayerName properties to the command window without an Index.

```
layers = wmsfind('srtm30');
disp(layers(1:5),'Index', 'off', ...
   'Properties',{'layertitle','layername'});

  5x1 WMSLayer

  Properties:
      LayerTitle: 'Topography, SRTM30+ Version 1.0, 30 arc second, Global - topo'
       LayerName: 'usgsCeSrtm30v1:topo'

      LayerTitle: 'Topography, SRTM30+ Version 6.0, 30 arc second, Global - topo'
       LayerName: 'usgsCeSrtm30v6:topo'
```

```
        LayerTitle: 'SRTM30 with Bathymetry (900m) merged with global ASTER (30m)'
         LayerName: 'EarthAsterElevations30m'

        LayerTitle: 'SRTM30 with Bathymetry (900m) merged with global ASTER (30m) and USG
         LayerName: 'mergedAsterElevations'

        LayerTitle: 'SRTM30 with Bathymetry (900m) merged SRTM3 V4.1 (90m) and USGS NED
         LayerName: 'mergedElevations'
```

**Sort and Display Property of WMSLayer Object**

Sort and display the `LayerName` property with an index.

Retrieve the layers.

```
layers = wmsfind('elevation');
```

Sort the layers.

```
[layerNames, index] = sort({layers.LayerName});
```

Display as a sample the first five results.

```
layers = layers(index);
disp(layers(1:5),'Label','off', 'Properties', 'layername');

  5x1 WMSLayer

  Properties:
          Index: 1
'0'

          Index: 2
'0'

          Index: 3
'10003_1'

          Index: 4
'10008_1'

          Index: 5
'10008_2'
```

1-1463

## See Also

`wmsfind`

# refine

**Class:** WMSLayer

Refine search

## Syntax

```
refine(layers,querystr,Name,Value,...)
```

## Description

`refine(layers,querystr,Name,Value,...)` searches for elements of `layers` in which values of the `Layer` or `LayerName` properties match the string *querystr*. Use the `'MatchType'` parameter to control whether the method uses partial or exact matching. The default is partial matching.

## Input Arguments

**layers**

Web Map Service layer object

**querystr**

Specifies the search string.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (`' '`). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'SearchFields'**

Case-insensitive string or cell array of strings. Valid strings are `'abstract'`, `'layer'`, `'layertitle'`, `'layername'`, `'server'`, `'serverurl'`, `'servertitle'`, or `'any'`.

**Default:** {'layer'}

**'MatchType'**

Case-insensitive string with value 'partial' or 'exact'. If you specify 'MatchType' as 'partial', then a match is determined if the query string is found in the property value. If you specify 'MatchType' as 'exact', then a match is determined only if the query string exactly matches the property value. If you specify 'MatchType' as 'exact' and querystr as '*', a match is found if the property value matches the character '*'.

**Default:** 'partial'

**'IgnoreCase'**

Logical. If you set 'IgnoreCase' to true, then WMSLayer.refine ignores case when performing string comparisons.

**Default:** true

## Tips

*   The WMSLayer.refine method searches the entries in the 'SearchFields' properties of layers for a partial match of the entry with querystr. The WMSLayer.refine method returns layer information if any supplied 'SearchFields' match. If you specify 'layer', then the method searches both the 'LayerTitle' and 'LayerName' properties. If you specify 'server', then the method searches both the 'ServerURL' and 'ServerTitle' fields. If you specify 'any', then the method searches the properties 'Abstract', 'LayerTitle', 'LayerName', 'ServerURL', and 'ServerTitle'.

## Examples

Refine a search of temperature layers to find two different sets of layers: (1) layers containing only annual sea surface temperatures, and (2) layers containing annual temperatures or sea surface temperatures.

```
temperature = wmsfind('temperature');
annual = refine(temperature,'annual');
```

```
sst = refine(temperature,'sea surface');
annual_and_sst = refine(sst,'annual');
annual_or_sst = [sst;annual];
```

## See Also
wmsfind | WMSLayer.refineLimits

# refineLimits

**Class:** WMSLayer

Refine search based on geographic limits

## Syntax

```
refineLimits(layers,Name,Value,...)
```

## Description

`refineLimits(layers,Name,Value,...)` searches for elements of `layers`, a Web Map Service Layers object, that match specific latitude or longitude limits. The results include a given layer only if the quadrangle specified by the optional `'Latlim'` and `'Lonlim'` parameters fully contains the boundary quadrangle, as defined by the `Latlim` and `Lonlim` properties. Partial overlap does not result in a match. All angles are in units of degrees.

## Input Arguments

**`layers` — Web map service layers**
Web Map Service Layers (WMSLayer) object

Web map service layers, specified as a Web Map Service Layers (WMSLayer) object

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**`'Latlim'`**

A two-element vector of latitude values specifying the latitudinal limits of the search in the form [`southern_limit northern_limit`] or a scalar value representing the latitude of a single point.

**'Lonlim'**

A two-element vector of longitude values specifying the longitudinal limits of the search in the form [western_limit eastern_limit] or a scalar value representing the longitude of a single point.

## Tips

- The default value of [] for either 'Latlim' or 'Lonlim' implies that all layers match the criteria. For example, if you specify the following, then the results include all the layers that cover the northern hemisphere.

```
refineLimits(layer,'Latlim', [0 90], 'Lonlim', [])
```

## Examples

Find layers containing global elevation data.

```
elevation = wmsfind('elevation');
latlim = [-90, 90];
lonlim = [-180, 180];
globalElevation = ...
    refineLimits(elevation,'Latlim', latlim, 'Lonlim', lonlim);
```

Print out the server titles from the unique servers.

```
globalElevation.serverTitles'

ans =

    'Ceoware2 WMS'
    'CubeSERV WMS'
    'CubeSERV Demo WMS'
    'deegree wms'
    'NASA Earth Observations (NEO) WMS'
    'JPL Planetary Map Service'
    'LMMP Tiled Web Map Service'
    'MicroImages TNTserver 7.3'
    'CubeSERV WMS'
    'ORNL DAAC WMS Server'
    'WMS GEOBASE / GEOBASE WMS'
```

```
'NASA WorldWind WMS'
'World Map'
'World Map'
'CubeSERV WMS'
```

## See Also

wmsfind

## servers

**Class:** WMSLayer

Return URLs of unique servers

## Syntax

```
serverURLs = servers(layers)
```

## Description

`serverURLs = servers(layers)` returns a cell array of URLs of unique servers.
`layers` is a Web Map Service Layers (WMSLayer) object.

## Examples

### Find all unique URLs of government servers

Find the URLs for government servers.

```
layers = wmsfind('*.gov*','SearchField', 'serverurl');
serverURLs = servers(layers);
sprintf('%s\n', serverURLs{:})

ans =

http://atlas.resources.ca.gov/ArcGIS/Services/Atmosphere_Climate/CaliforniaWeatherForec
http://atlas.resources.ca.gov/ArcGIS/Services/Atmosphere_Climate/RIDGE-345min/MapServer
http://atlas.resources.ca.gov/ArcGIS/Services/Atmosphere_Climate/RIDGE_Precip_Radar/Map
http://atlas.resources.ca.gov/ArcGIS/Services/Atmosphere_Climate/RIDGE_Radar_TimeSeries
http://atlas.resources.ca.gov/ArcGIS/Services/Atmosphere_Climate/USCityWeatherForecast/
http://atlas.resources.ca.gov/ArcGIS/Services/Base_Maps/JacksonStForest/MapServer/WMSSe
http://atlas.resources.ca.gov/ArcGIS/Services/Base_Maps/Topo_GrayScale/MapServer/WMSSer
http://atlas.resources.ca.gov/ArcGIS/Services/Base_Maps/Topo_Hillshade/MapServer/WMSSer
http://atlas.resources.ca.gov/ArcGIS/Services/Base_Maps/Topo_Regular/MapServer/WMSServe
.
.
```

1-1471

.

**List server URLs that contain a temperature layer along with the number of temperature layers**

Find server URLs that contain temperature layers and return them in a Web Map Service Layers (WMSLayer) object.

```
temperature = wmsfind('temperature');
serverURLs = servers(temperature);
for k=1:numel(serverURLs)
   querystr = serverURLs{k};
   layers = refine(temperature,querystr, ...
      'SearchFields', 'serverurl');
   fprintf('Server URL\n%s\n', layers(1).ServerURL);
   fprintf('Number of layers: %d\n\n', numel(layers));
end

Server URL
http://www.ifremer.fr/thredds/wms/METEOFRANCE-EUR-SST_L3MONOSENSOR_NRT-OBS_SEVIRI_1_1H_
Number of layers: 2

Server URL
http://www.ifremer.fr/thredds/wms/METEOFRANCE-EUR-SST_L3MONOSENSOR_NRT-OBS_SEVIRI_1_1H_
Number of layers: 2

Server URL
http://www.ifremer.fr/thredds/wms/METEOFRANCE-EUR-SST_L3MONOSENSOR_NRT-OBS_TMI-EUR-MYO
Number of layers: 2
.
.
.
```

## See Also
wmsfind | WMSLayer.refine | WMSLayer.serverTitles

# serverTitles

**Class:** WMSLayer

Return titles of unique servers

## Syntax

```
titles = serverTitles(layers)
```

## Description

`titles = serverTitles(layers)` returns a cell array of titles of unique servers, where `layers` identifies the Web Map Service layers.

## Examples

List titles of unique government servers:

```
layers = wmsfind('*.gov*', 'SearchField', 'serverurl');
titles = serverTitles(layers);
sprintf('%s\n', titles{:})

Atmosphere_Climate/CaliforniaWeatherForecasts
Atmosphere_Climate_RIDGE-345min
WMS
Atmosphere_Climate_RIDGE_Radar_TimeSeries
Atmosphere_Climate/USCityWeatherForecast
Base_Maps_JacksonStForest
Base_Maps_Topo_GrayScale
Base_Maps_Topo_Hillshade
Base_Maps_Topo_Regular
Base_Maps_Topo_Transparent
WMS
Base_Maps/california_hillshade_base_map
Boundaries/CountyBoundaries
Boundaries/ElectedOfficials
Boundaries_SRA_wgs84wm
```

```
Environment_Watersheds
Environment/bond_projects
GeoScience/CSMW_Geology
GeoScience/California_Fire_History
GeoScience/GeoMAC_ActiveFires
GeoScience_GeoMAC_InActiveFires
GeoScience_HazardMaps_09_01_2009_Fire
GeoScience/US_Fire_History
Health_ADP_Treatment_Providers
.
.
.
```

## See Also

wmsfind | WMSLayer.servers

# Details property

**Class:** WMSLayer

## Description

A structure containing detailed information about a layer

### Details Structure

| Field Name | Data Type | Field Content |
|---|---|---|
| MetadataURL | String | URL containing metadata information about layer. |
| Attributes | Structure | Attributes of layer. |
| BoundingBox | Structure array | Bounding box of layer. |
| Dimension | Structure array | Dimensional parameters of layer, such as time or elevation. |
| ImageFormats | Cell array | Image formats supported by server. |
| ScaleLimits | Structure | Scale limits of layer. |
| Style | Structure array | Style parameters that determine layer rendering. |
| Version | String | WMS version specification. |

### Attributes Structure

| Field Name | Data Type | Field Content |
|---|---|---|
| Queryable | Logical | True if you can query the layer for feature information. |
| Cascaded | Double | Number of times a Cascading Map server has retransmitted the layer. |
| Opaque | Logical | True if the map data are mostly or completely opaque. |

| Field Name | Data Type | Field Content |
|---|---|---|
| NoSubsets | Logical | True if the map must contain the full bounding box. false if the map can be a subset of the full bounding box. |
| FixedWidth | Logical | True if the map has a fixed width that the server cannot change. false if the server can resize the map to an arbitrary width. |
| FixedHeight | Logical | True if the map has a fixed height that the server cannot change. false if the server can resize the map to an arbitrary height. |

**BoundingBox Structure**

| Field Name | Data Type | Field Content |
|---|---|---|
| CoordRefSysCode | String | Code number for coordinate reference system. |
| XLim | Double array | X limit of layer in units of coordinate reference system. |
| YLim | Double array | Y limit of layer in units of coordinate reference system. |

**Dimension Structure**

| Field Name | Data Type | Field Content |
|---|---|---|
| Name | String | Name of the dimension; such as time, elevation, or temperature. |
| Units | String | Measurement unit. |
| UnitSymbol | String | Symbol for unit. |
| Default | String | Default dimension setting. For example, if default is 'time' and dimension is not specified, server returns time holding. |
| MultipleValues | Logical | True if multiple values of the dimension may be requested. false if only single values may be requested. |
| NearestValue | Logical | True if nearest value of dimension is returned in response to request for nearby value. false |

| Field Name | Data Type | Field Content |
|---|---|---|
| | | if request value must correspond exactly to declared extent values. |
| Current | Logical | True if temporal data are kept current (valid only for temporal extents). |
| Extent | String | Values for dimension. Expressed in any of the following ways:<br><br>• Single value (value)<br>• List of values (value1, value2, ...)<br>• Interval defined by bounds and resolution (min1/max1/res1)<br>• List of intervals (min1/max1/res1, min2/max2/res2, ...) |

**ScaleLimits Structure**

| Field Name | Data Type | Field Content |
|---|---|---|
| ScaleHint | Double array | Minimum and maximum scales for which it is appropriate to display layer. Expressed as scale of ground distance in meters represented by diagonal of central pixel in image. |
| MinScaleDenominator | Double | Minimum scale denominator of maps for which a layer is appropriate. |
| MaxScaleDenominator | Double | Maximum scale denominator of maps for which a layer is appropriate. |

**Style Structure Array**

| Field Name | Data Type | Field Content |
|---|---|---|
| Title | String | Descriptive title of style. |
| Name | String | Name of style. |
| Abstract | String | Information about style. |

| Field Name | Data Type | Field Content |
|---|---|---|
| LegendURL | Structure | Information about legend graphics. |

**LegendURL Structure**

| Field Name | Data Type | Field Content |
|---|---|---|
| OnlineResource | String | URL of legend graphics. |
| Format | String | Format of legend graphics. |
| Height | Double | Height of legend graphics. |
| Width | Double | Width of legend graphics. |

# WMSMapRequest class

Web Map Service map request object

## Description

A `WMSMapRequest` object contains a request to a WMS server to obtain a map, which represents geographic information. The WMS server renders the map as a color or grayscale image. The object contains properties that you can set to control the geographic extent, rendering, or size of the requested map.

## Construction

`mapRequest = WMSMapRequest(layer)` constructs a `WMSMapRequest` object. The `WMSLayer` array `layer` contains only one unique `ServerURL`. The `WMSMapRequest` class updates the properties of `layer`, if necessary.

`mapRequest = WMSMapRequest(layer, server)` constructs a `WMSMapRequest` object. `layer` is a `WMSLayer` object, and `server` is a scalar `WebMapServer` object. The `ServerURL` property of `layer` must match the `ServerURL` property of `server`. The `server` object updates `layer` properties.

## Properties

**Server**

Initialized to the `Server` input, if supplied to the constructor; otherwise constructed using the `ServerURL` of `Layer`.

**Data Type:** scalar `WebMapServer` object

**Layer**

Initialized to the `layer` input supplied to the constructor. The `Layer` property contains one unique `ServerURL`. The `Server` property updates the properties of `Layer` when the property is set. The `ServerURL` property of `Layer` must match the `ServerURL` property of `Server`.

**Data Type:** `WMSLayer` array

### CoordRefSysCode

Specifies the coordinate reference system code. Its default value is `'CRS:84'`, for WMS version 1.3.x, and `'EPSG:4326'` for all other versions. If `'EPSG:4326'` is not found in `Layer.CoordRefSysCodes`, then the `CoordRefSysCode` value is set from the first `CoordRefSysCode` found in the `Layer.Details.BoundingBox` structure array. When `CoordRefSysCode` is set to `'EPSG:4326'` or `'CRS:84'`, the `XLim` and `YLim` properties are set to `[]` and the `Latlim` and `Lonlim` properties are set to the geographic extent defined by the `Layer` array. When `CoordRefSysCode` is set to a value other than `'EPSG:4326'` or `'CRS:84'`, then the `XLim` and `YLim` properties are set from the values found in the `Layer.Details.BoundingBox` structure and the `Latlim` and `Lonlim` properties are set to `[]`. Automatic projections are not supported. (Automatic projections begin with the string `'AUTO'`.)

**Data Type:** string

**Default:** `'EPSG:4326'` or `'CRS:84'`

### RasterReference

References the raster map to an intrinsic coordinate system.

**Data Type:** map.rasterref.MapCellsReference or map.rasterref.GeographicCellsReference

### Latlim

Contains the southern and northern latitudinal limits of the request in units of degrees. The limits must be ascending.

**Data Type:** two-element vector

**Default:** Limits that span all latitudinal limits found in the `Layer.Latlim` property

### Lonlim

Contains the western and eastern longitudinal limits of the request in units of degrees. The limits must be ascending and in the range [-180, 180] or [0 360].

**Data Type:** two-element vector

**Default:** Limits that span all longitudinal limits in the `Layer.Lonlim` property

**XLim**

Contains the western and eastern limits of the request in units specified by the coordinate reference system. The limits must be ascending. You can set `XLim` only if you set `CoordRefSysCode` to a value other than `EPSG:4326`.

**Data Type:** two-element vector

**Default:** [ ]

**YLim**

Contains the southern and northern limits of the request in units specified by the coordinate reference system. The limits must be ascending. You can set `YLim` only if you set `CoordRefSysCode` to a value other than `EPSG:4326`.

**Data Type:** two-element vector

**Default:** [ ]

**ImageHeight**

Specifies the height in pixels for the requested raster map. The property `MaximumHeight` defines the maximum value for `ImageHeight`. The `WMSMapRequest` class initializes the `ImageHeight` property to either 512 or to an integer value that best preserves the aspect ratio of the coordinate limits, without changing the coordinate limits.

**Data Type:** scalar, positive integer

**ImageWidth**

Specifies the width in pixels for the requested raster map. The property `MaximumWidth` defines the maximum value for `ImageWidth`. The `WMSMapRequest` class initializes the `ImageWidth` property to either 512 or to an integer value that best preserves the aspect ratio of the coordinate limits, without changing the coordinate limits.

**Data Type:** scalar, positive integer

**Maximum Height**

Contains the maximum height in pixels for the requested map. Cannot be set. The value of `MaximumHeight` is 8192.

**Data Type:** `double`

**Maximum Width**

Contains the maximum width in pixels for the requested map. Cannot be set. The value of `MaximumWidth` is 8192.

**Data Type:** `double`

**Elevation**

Gives the elevation extent of the requested map. When you set the property, `'elevation'` must be the value of the `Layer.Details.Dimension.Name` field.

**Data Type:** string

**Default:** `''`

**Time**

Specifies the time extent of the requested map. See the `WMSMapRequest.Time` reference page for more information.

**Data Type:** string or double

**Default:** `''`

**SampleDimension**

Contains the name of a sample dimension (other than `'time'` or `'elevation'`) and its value. `SampleDimension{1}` must be the value of the `Layer.Details.Dimension.Name` field.

**Data Type:** two-element cell array of strings

**Transparent**

Specifies whether the map background is transparent. When you set `Transparent` to `true`, the server sets all pixels not representing features or data values in that layer to a transparent value, producing a composite map. When you set `Transparent` to `false`, the server sets all non-data pixels to the value of the background color.

**Data Type:** logical scalar

**Default:** `false`

**BackgroundColor**

Specifies the color of the background (non-data) pixels of the map. The values range from 0 to 255. The default value, `[255,255,255]`, specifies the background color as white. You can set `BackgroundColor` using `non-uint8` numeric values, but they are cast and stored as `uint8`.

**Data Type:** three-element vector of `uint8` values

**StyleName**

Specifies the style to use when rendering the image. The `StyleName` must be a valid entry in the `Layer.Details.Style.Name` field. (The cell array of strings contains the same number of elements as does `Layer`.)

**Data Type:** string or cell array of strings

**Default:** `{}`

**ImageFormat**

Specifies the desired image format used to render the map as an image. If set, the format must match an entry in the `Layer.Details.ImageFormats` cell array and an entry in the `ImageRenderFormats` property. If not set, the format defaults to a value in the `ImageRenderFormats` property.

**Data Type:** string

**ImageRenderFormats**

Contains the preferred image rendering formats when `Transparent` is set to `false`. The first entry is the most preferred image format. If the preferred format is not stored in the `Layer` property, then the next format from the list is selected, until a format is found. The `ImageRenderFormats` array is not used if the `ImageFormat` property is set. The `ImageRenderFormats` property cannot be set.

**Data Type:** cell array

**ImageTransparentFormats**

Contains the preferred image rendering formats when `Transparent` is set to `true`. When `Transparent` is set to `true`, the `ImageFormat` property is set to the first entry

in the `ImageTransparentFormats` list, if it is stored in the `Layer` property. Otherwise, the list is searched for the next element, until a match is found. If a transparent image format is not found in the list, or if the `ImageFormat` property is set to a non-default value, then `ImageFormat` is unchanged. The `ImageTransparentFormats` property cannot be set.

**Data Type:** cell array

**ServerURL**

Contains the server URL for the WMS `GetMap` request. In general, `ServerURL` matches the `ServerURL` of the `Layer`. However, some WMS servers, such as the Microsoft® TerraServer, require a different URL for `GetMap` requests than for WMS `GetCapabilities` requests.

**Data Type:** string

**Default:** `Layer(1).ServerURL`

**RequestURL**

Contains the URL for the WMS `GetMap` request. It is composed of the `ServerURL` with additional WMS parameter/value pairs. This property cannot be set.

**Data Type:** string

## Methods

| | |
|---|---|
| boundImageSize | Bound size of raster map |

## Examples

**Read Global Sea-Surface Temperature Map from NASA Server**

Read a global, half-degree resolution, sea-surface temperature map for the month of November 2009. The map, from the AMSR-E sensor on NASA's Aqua satellite, uses data provided by NASA's Earth Observations (NEO) WMS server.

```
sst = wmsfind('AMSRE_SSTAn_M');
server = WebMapServer(sst.ServerURL);
mapRequest = WMSMapRequest(sst, server);
timeRequest = '2009-11-01';
mapRequest.Time = timeRequest;
samplesPerInterval = .5;
mapRequest.ImageHeight = ...
   round(abs(diff(sst.Latlim))/samplesPerInterval);
mapRequest.ImageWidth = ...
   round(abs(diff(sst.Lonlim))/samplesPerInterval);
mapRequest.ImageFormat = 'image/png';
sstImage = server.getMap(mapRequest.RequestURL);
```

Read the legend for the layer using the `OnlineResource` URL field in the `LegendURL` structure. The legend shows that the temperature ranges from -2 to 35 degrees Celsius. The `WMSMapRequest` object updates the layer information from the server.

```
url = mapRequest.Layer.Details.Style(1).LegendURL.OnlineResource;
[legend,cmap] = imread(url);
if ~isempty(cmap)
   % Convert indexed image to RGB.
   legendImg = ind2rgb(legend,cmap);
else
   % Already have an RGB image.
   legendImg = legend;
end
```

Display the temperature map and legend.

```
fig = figure;
ax = worldmap('world');
setm(ax,'MlabelParallel',-90,'MlabelLocation',90)
    geoshow(sstImage,mapRequest.RasterReference);
    title({mapRequest.Layer.LayerTitle,timeRequest})

figurePosition = fig.Position;
centerWidth = figurePosition(3)/2;
left = centerWidth-size(legendImg,2)/2;
bottom = 30;
width  = size(legendImg,2);
height = size(legendImg,1);
axes('Units','pixels','Position',[left bottom width height])
image(legendImg)
axis off
```
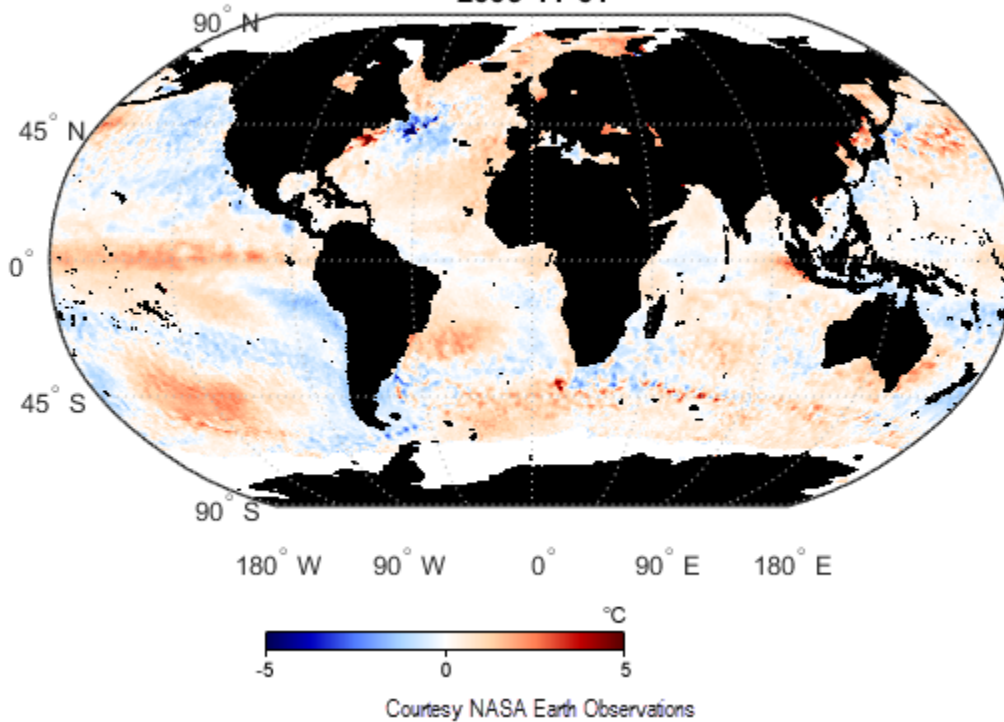
Read abstract information for this layer from the `MetadataURL` field.

```
options = weboptions('ContentType','xmldom','Timeout',10);
xml = webread(mapRequest.Layer.Details.MetadataURL,options);
abstract = xml.getElementsByTagName('abstract').item(0).getTextContent
```

### Read and Display Global Elevation and Bathymetry Layer

Read and display a global elevation and bathymetry layer for the Gulf of Maine at 30 arc-seconds sampling interval. The values are in units of meters.

```
layers = wmsfind('srtm30', 'SearchField', 'LayerName');
layer = refine(layers,'nasa.network','SearchField','serverurl')
server = WebMapServer(layer.ServerURL);
```

```
mapRequest = WMSMapRequest(layer,server);
mapRequest.Latlim = [40 46];
mapRequest.Lonlim = [-71 -65];
samplesPerInterval = 30/3600;
mapRequest.ImageHeight = ...
   round(abs(diff(mapRequest.Latlim))/samplesPerInterval);
mapRequest.ImageWidth = ...
   round(abs(diff(mapRequest.Lonlim))/samplesPerInterval);
mapRequest.ImageFormat = 'image/bil';
Z = getMap(server, mapRequest.RequestURL);
```

Display and contour the map at sea level (0 meters).

```
figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim)
geoshow(double(Z),mapRequest.RasterReference,'DisplayType','texturemap')
demcmap(double(Z))
contourm(double(Z),mapRequest.RasterReference,[O O],'Color','black')
colorbar
title ({'Gulf of Maine', mapRequest.Layer.LayerTitle}, ...
       'Interpreter', 'none', 'FontWeight', 'bold')
```
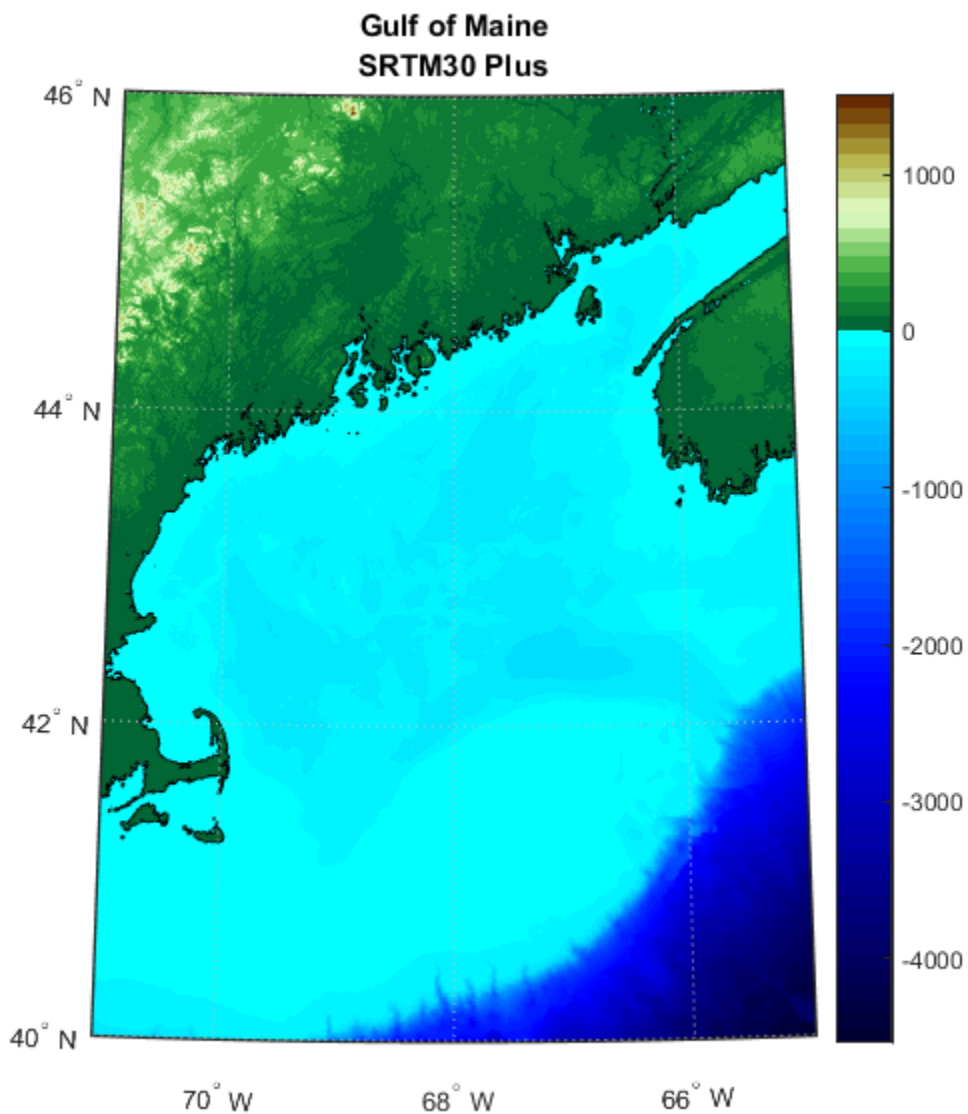
## See Also

WebMapServer | WMSLayer | wmsfind | wmsinfo | wmsread

# boundImageSize

**Class:** WMSMapRequest

Bound size of raster map

## Syntax

```
mapRequest = boundImageSize(mapRequest, imageLength)
```

## Description

`mapRequest = boundImageSize(mapRequest, imageLength)` bounds the size of the raster map based on the scalar `imageLength`. The scalar `mapRequest` is a `WMSMapRequest` object. The double `imageLength` indicates the length in pixels for the row (`ImageHeight`) or column (`ImageWidth`) dimension. The `boundImageSize` method calculates the row or column dimension length by using the aspect ratio of the `Latlim` and `Lonlim` properties or the aspect ratio of the `XLim` and `YLim` properties, if they are set.

The `boundImageSize` method measures image dimensions in geographic or map coordinates. The method sets the longest image dimension to `imageLength`, and the shortest to the nearest integer value that preserves the aspect ratio, without changing the coordinate limits. The maximum value of the `MaximumHeight` and `MaximumWidth` properties becomes the maximum value of `imageLength`.

## Examples

### Read and Display VMAP0 Basic Layer for Entire Globe

Read the VMAP0 basic layer for the entire globe.

```
vmap0 = wmsfind('vmap0.tiles','SearchField','serverurl');
vmap0 = wmsupdate(vmap0);
layer = refine(vmap0,'basic');
```

```
request = WMSMapRequest(layer);
request.Transparent = true;
imageLength = 720;
request = boundImageSize(request,imageLength);
globalImage = getMap(request.Server, request.RequestURL);
```

Display the map. The rendered map has a spatial resolution of .5 degrees per cell and an image size of 360-by-720 pixels.

```
figure; worldmap('world')
geoshow(globalImage, request.RasterReference);
title(['VMAP0 ' layer.LayerTitle ' Layer'])
```
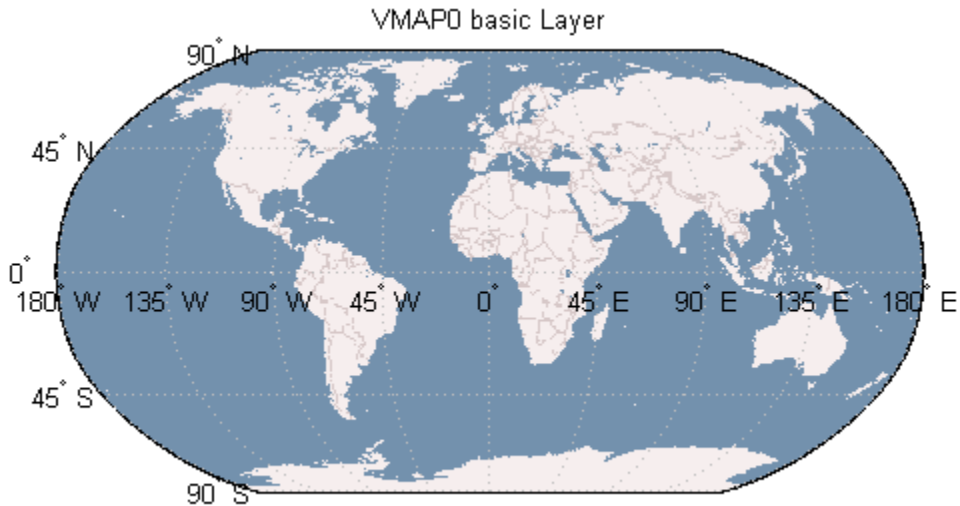


Courtesy Metacarta

**Read and Display Multiple Layers Centered Around London**

Read and display multiple layers centered around London. The rendered map has a spatial extent of .5 degrees and an image size of 1024-by-1024 pixels

```
vmap0 = wmsfind('vmap0.tiles','SearchField','serverurl');
vmap0 = wmsupdate(vmap0);
layers = [ refine(vmap0, 'rail'); refine(vmap0, 'river'); ...
        refine(vmap0, 'priroad'); refine(vmap0, 'secroad'); ...
```

```
        refine(vmap0, 'ctylabel'); refine(vmap0, 'basic')];
request = WMSMapRequest(layers);
cities = shaperead('worldcities', 'UseGeo', true);
london = cities(strcmpi('London', {cities.Name}));
extent = [-.25 .25];
request.Latlim = london.Lat + extent;
request.Lonlim = london.Lon + extent;

request.Transparent = true;
imageLength = 1024;
request = boundImageSize(request,imageLength);
londonImage = getMap(request.Server,request.RequestURL);
```

Display the map. The rendered map has a spatial extent of .5 degrees and an image size of 1024-by-1024 pixels

```
figure
worldmap(londonImage,request.RasterReference)
geoshow(londonImage,request.RasterReference)
title({'Region Surrounding London, England', ...
   ['with Primary and Secondary Roads, ', ...
    'Rivers, Rails, City Label, and Basic Layers']})
```

Region Surrounding London, England
with Primary and Secondary Roads, Rivers, Rails, City Label, and Basic Layers

Courtesy Metacarta
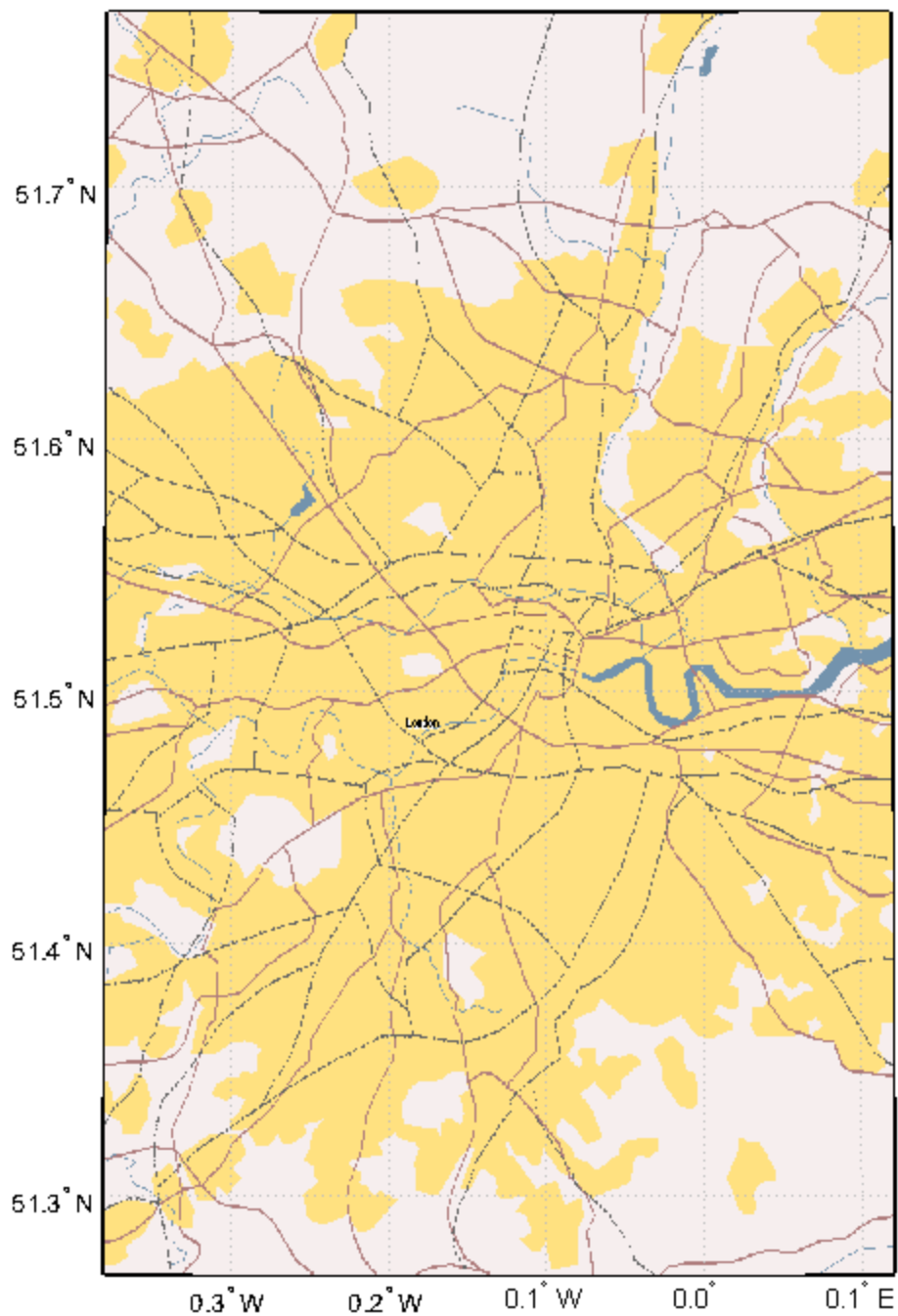
# Time property

**Class:** WMSMapRequest

Requested time extent

## Description

The `WMSMapRequest.Time` property stores time as a string or a double indicating the desired time extent of the requested map. When you set the property, `'time'` must be the value of the `Layer.Details.Dimension.Name` field. The default value is `''`.

Time is stored in the ISO® 8601:1988(E) extended format. In general, the `Time` property is stored in a `yyyy-mm-dd` format or a `yyyy-mm-ddThh:mm:ssZ` format, if the precision requires hours, minutes, or seconds. You can use several different string and numeric formats to set the `Time` property, according to the following table (where dateform number is the number used by the Standard MATLAB Date Format Definitions). Express all hours, minutes, and seconds in Coordinated Universal Time (UTC).

| Dateform (number) | Input (string) | Stored format |
|---|---|---|
| 0 | `dd-mm-yyyy HH:MM:SS` | yyyy-mm-ddTHH:MM:SSZ |
| 1 | `dd-mm-yyyy` | yyyy-mm-dd |
| 2 | `mm/dd/yy` | yyyy-mm-dd |
| 6 | `mm/dd` | yyyy-mm-dd (current year) |
| 10 | `yyyy` | yyyy |
| 13 | `HH:MM:SS` | yyyy-mm-ddTHH:MM:SSZ |
| 14 | `HH:MM:SS PM` | yyyy-mm-ddTHH:MM:SSZ |
| 15 | `HH:MM` | yyyy-mm-ddTHH:MM:00Z |
| 16 | `HH:MM PM` | yyyy-mm-ddTHH:MM:00Z |
| 21 | `mmm.dd,yyyy HH:MM:SS` | yyyy-mm-ddTHH:MM:SSZ |
| 22 | `mmm.dd,yyyy` | yyyy-mm-dd |
| 23 | `mm/dd/yyyy` | yyyy-mm-dd |

| Dateform (number) | Input (string) | Stored format |
|---|---|---|
| 26 | `yyyy/mm/dd` | yyyy-mm-dd |
| 29 | `yyyy-mm-dd` | yyyy-mm-dd |
| 30 | `yyyymmddTHHMMSS` | yyyy-mm-ddTHH:MM:SSZ |
| 31 | `yyyy-mm-dd HH:MM:SS` | yyyy-mm-ddTHH:MM:SSZ |

Inputs using the dateform numbers 13–16 return the date set to the current year, month, and day. Use of other dateform formats, especially 19, 20, 24, and 25, results in erroneous output.

In addition to these standard MATLAB dateform formats, the `WMSMapRequest.Time` property also accepts the following inputs.

| Input (string) | Description |
|---|---|
| `'current'` | The current time holdings of the server |
| `numeric datenum` | Numeric date value converted to yyyy-mm-dd string (dateform 29 format) |
| `Byyyy` | B.C.E. year |

Use the prefixes K, M, and G, followed by a string number (thousand, million, and billion years, respectively), for geologic data sets that refer to the distant past.

# wmsread

Retrieve WMS map from server

## Syntax

```
[A,R] = wmsread(layer)
[A,R] = wmsread(layer,Name,Value,...)
[A,R] = wmsread(mapRequestURL)
[A,R,mapRequestURL] = wmsread(...)
```

## Description

`[A,R] = wmsread(layer)` accesses the Internet to render and retrieve a raster map from a Web Map Service (WMS) server. The `ServerURL` property of the `WMSLayer` object, `layer`, specifies the server. If `layer` has more than one element, then the server overlays each subsequent layer on top of the base (first) layer, forming a single image. The server renders multiple layers only if all layers share the same `ServerURL` value.

The WMS server returns a raster map, either a color or grayscale image, in the output `A`. The second output, `R`, is a raster reference object that ties `A` to the EPSG:4326 geographic coordinate system. The rows of `A` are aligned with parallels, with even sampling in longitude. Likewise, the columns of `A` are aligned with meridians, with even sampling in latitude.

The geographic limits of `A` span the full latitude and longitude extent of `layer`. The `wmsread` function chooses the larger spatial size of `A` to match its larger geographic dimension. The larger spatial size is fixed at the value 512. In other words, assuming RGB output, `A` is `512-by-N-by-3` if the latitude extent exceeds longitude extent and `N-by-512-by-3` otherwise. In both cases N <= 512. The `wmsread` function sets N to the integer value that provides the closest possible approximation to equal cell sizes in latitude and longitude. The map spans the full extent supported for the `layer`.

`[A,R] = wmsread(layer,Name,Value,...)` specifies parameter-value pairs that modify the request to the server. You can abbreviate parameter names, which are case-insensitive.

`[A,R] = wmsread(mapRequestURL)` uses the input argument `mapRequestURL` to define the request to the server. The `mapRequestURL` string contains a WMS `serverURL` with additional WMS parameters. The URL string includes the WMS parameters `BBOX` and `GetMap` and the `EPSG:4326` or `CRS:84` keyword. Obtain a `mapRequestURL` from the output of `wmsread`, the `RequestURL` property of a `WMSMapRequest` object, or an Internet search.

`[A,R,mapRequestURL] = wmsread(...)` returns a WMS `GetMap` request URL in the string `mapRequestURL`. You can insert the `mapRequestURL` into a browser to make a request to a server, which then returns the raster map. The browser opens the returned map if its mime type is understood, or saves the raster map to disk.

## Examples

### Read and Display a Blue Marble Next Generation Layer from NASA

Read NASA layer and display it.

```
nasa = wmsfind('nasa', 'SearchField', 'serverurl');
layer = nasa.refine('bluemarbleng',  'SearchField', 'layername', ...
    'MatchType', 'exact');
[A, R] = wmsread(layer(1));
figure
axesm globe
axis off
geoshow(A, R)
title('Blue Marble')
```

Blue Marble
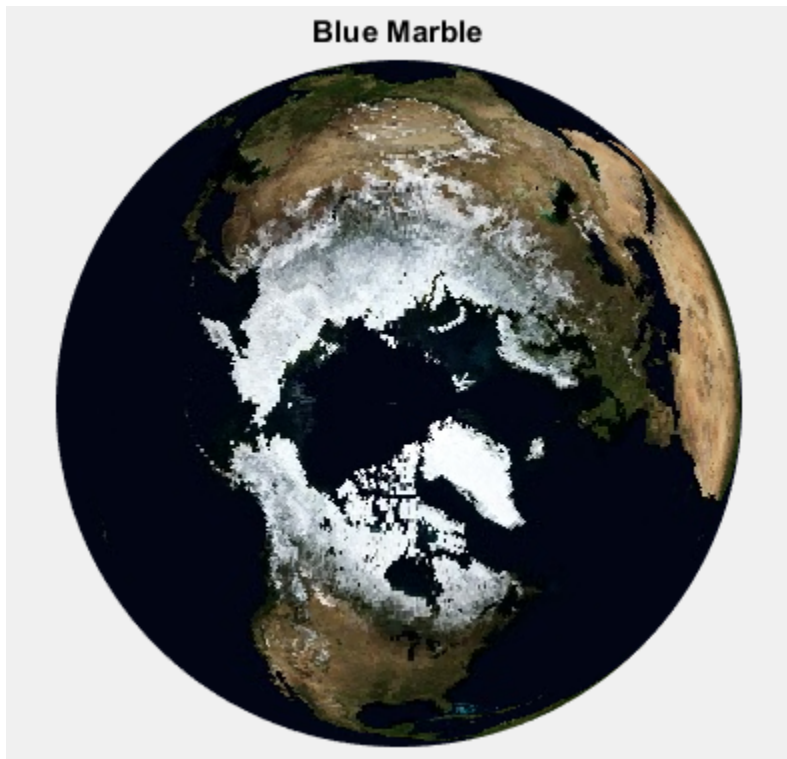
### Read and Display an Orthoimage

Read and display an orthoimage of the northern section of the Golden Gate Bridge in San Francisco, California, using the USGS National Map Seamless server.

Define region of interest.

```
latlim = [37.824928  37.829598];
lonlim = [-122.482373 -122.47768];
```

Find the USGS high-resolution ortho-imagery layer. The USGS National Map provides ortho-imagery from various regions of the United States. One method to obtain the high-resolution ortho-imagery layer is to obtain the capabilities document from the server. The ortho-imagery layer is the only layer from this server. Use multiple attempts to connect to the server in case it is busy.

```
numberOfAttempts = 5;
```

```
attempt = 0;
info = [];
serverURL = 'http://raster.nationalmap.gov/arcgis/services/Orthoimagery/USGS_EROS_Ortho
while(isempty(info))
    try
        info = wmsinfo(serverURL);
        orthoLayer = info.Layer(1);
    catch e

        attempt = attempt + 1;
        if attempt > numberOfAttempts
            throw(e);
        else
            fprintf('Attempting to connect to server:\n"%s"\n', serverURL)
        end
    end
end
```

Obtain the image and display it in a UTM projection.

```
imageLength = 1024;
[A, R] = wmsread(orthoLayer, 'Latlim', latlim, 'Lonlim', lonlim, ...
    'ImageHeight', imageLength, 'ImageWidth', imageLength);

figure
axesm('utm', 'Zone', utmzone(latlim, lonlim), ...
    'MapLatlimit', latlim, 'MapLonlimit', lonlim, ...
    'Geoid', wgs84Ellipsoid)
geoshow(A,R)
axis off
title({'San Francisco','Northern Section of Golden Gate Bridge'})
```

San Francisco
Northern Section of Golden Gate Bridge
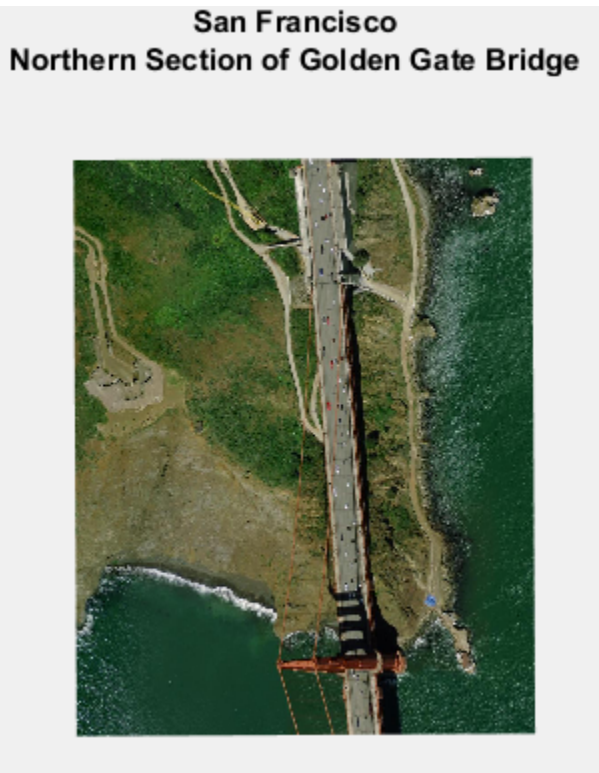
### Read and Display Global Monthly Composite of Sea Surface Temperature

Read and display a global monthly composite of sea surface temperature for April 16, 2010 based on data from the AMSR-E sensor on board the Aqua satellite. Include the coastline, landmask, and nation layers.

```
coastwatch = wmsfind('coastwatch', 'SearchField', 'serverurl');
layers = coastwatch.refine('erdAAsstamday', ...
    'Searchfield','serverurl');
time = '2010-04-16T00:00:00Z';
[A, R] = wmsread(layers(end:-1:1), 'Time', time);
figure
axesm('pcarree', 'Maplonlimit', [0, 360], ...
    'PLabelLocation', 45, 'MLabelLocation', 90, ...
    'MLabelParallel', -90, 'MeridianLabel', 'on', ...
    'ParallelLabel', 'on');
```

```
geoshow(A, R);
title({layers(end).LayerTitle, time})
```
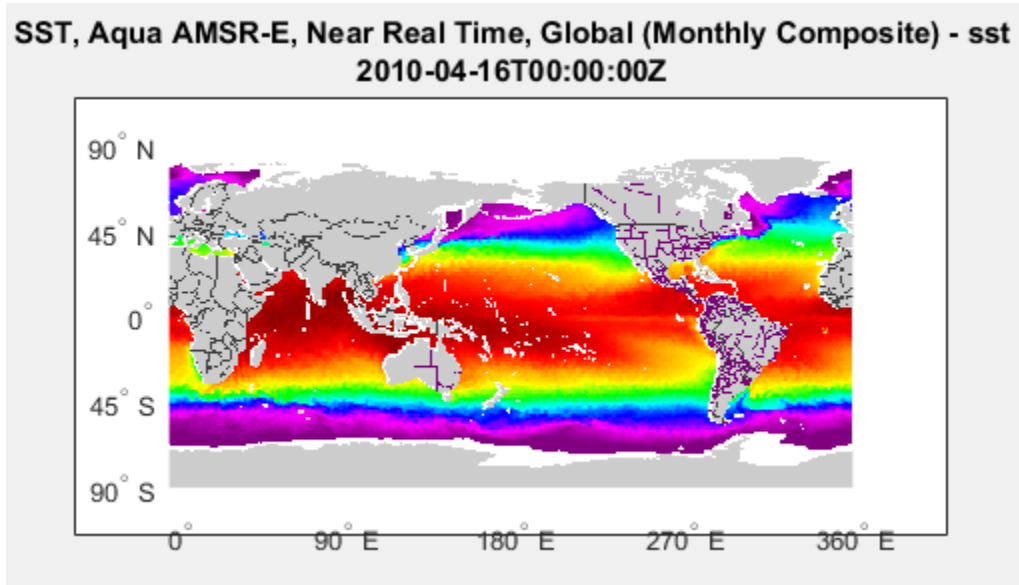


## Input Arguments

### **layer** — Information about the layer you are retrieving
WMSLayer object

Information about the layer you are retrieving, specified as a WMSLayer object.

Example: [A,R] = wmsread(layer(1));

### **mapRequestURL** — WMS **GetMap** request URL
character string

WMS GetMap request URL, specified as a character string.

Example: [A,R] = wmsread(mapURL);

Data Types: char

## Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[A,R] = wmsread(layer(1),'latlim',[40 50];`

### `'latlim'` — Latitude limits of the output image in degrees

`[ ]` (default) | two-element vector

Latitude limits of the output image in degrees, specifed as a two-element vector of the form `[southern_limit northern_limit]`. The limit values must be ascending. By default, `'Latlim'` is empty, and `wmsread` uses the full extent in latitude of `layer`. If `Layer.Details.Attributes.NoSubsets` is `true`, `'Latlim'` may not be modified.

Example: `[A,R] = wmsread(layer(1),'latlim',[40 50];`

Data Types: `double`

### `'lonlim'` — Longitude limits of the output image in degrees

`[ ]` (default) | two-element vector

Longitude limits of the output image in degrees, specified as a two-element vector in the form `[western_limit eastern_limit]`. The limit values must be ascending. By default, `'Lonlim'` is empty and the full extent in longitude of `layer` is used. If `Layer.Details.Attributes.NoSubsets` is `true`, you cannot modify `'Lonlim'`

Example: `[A,R] = wmsread(layer(1),'lonlim',[40 50];`

Data Types: `double`

### `'ImageHeight'` — Desired height of the raster map in pixels

scalar, positive, integer-valued number

Desired height of the raster map in pixels, specified as a scalar, positive, integer-valued number. `ImageHeight` cannot exceed `8192`. If `layer.Details.Attributes.FixedHeight` contains a positive number, you cannot modify `'ImageHeight'`.

Example: `[A,R] = wmsread(layer(1),'ImageHeight',[40 50];`

Data Types: `double`

### `'ImageWidth'` — Desired width of the raster map in pixels
scalar, positive, integer-valued number

Desired width of the raster map in pixels, specified as a scalar, positive, integer-valued number. ImageWidth cannot exceed 8192. If `Layer.Details.Attributes.FixedWidth` contains a positive number, you cannot modify `'ImageWidth'`.

Example: `[A,R] = wmsread(layer(1),'ImageWidth',100);`

Data Types: `double`

### `'CellSize'` — Target size of the output pixels (raster cells) in degrees
scalar or two-element vector

Target size of the output pixels (raster cells) in degrees, specified as a scalar or two-element vector. If you specify a scalar, the value applies to both height and width dimensions. If you specify a vector, use the form `[height width]`. The `wmsread` function issues an error if you specify both CellSize and ImageHeight or ImageWidth. The output raster map must not exceed a size of `[8192,8192]`.

Example: `[A,R] = wmsread(layer(1),'Cellsize',5);`

Data Types: `double`

### `'RelTolCellSize'` — Relative tolerance for `'CellSize'`
`.001` (default) | scalar or two-element vector

Relative tolerance for `'CellSize'`, specified as a scalar or two-element vector. If you specify a scalar, the value applies to both height and width dimensions. If you specify a vector, the tolerances appear in the order `[height width]`.

Example: `[A,R] = wmsread(layer(1),'RelTolCellsize',[4 5]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

### `'ImageFormat'` — Desired format to use in rendering the map as an image
first available format in the in the `Layer.Details.ImageFormats` cell array (default) | string

Desired format to use in rendering the map as an image, specified as one of the following strings. If specified, the format must match an entry in the

`Layer.Details.ImageFormats` cell array. If not specified, the format defaults to the first available format in the supported format list. .

| String | Description |
|---|---|
| `'image/jpeg'` | JPEG |
| `'image/gif'` | GIF |
| `'image/png'` | PNG |
| `'image/tiff'` | TIFF |
| `'image/geotiff'` | GeoTIFF |
| `'image/ geotiff8'` | GeoTIFF8 |
| `'image/tiff8'` | TIFF8 |
| `'image/png8'` | PNG8 |
| `'image/bil'` | Band Interleaved by Line (BIL) format. When you specify the `'image/bil'` format, wmsread returns A as a two-dimensional array with a class type of `int16` or `int32`. |

Example: `[A,R] = wmsread(layer(1),'ImageFormat','image/jpeg');`

Data Types: `char`

### `'StyleName'` — Style to use when rendering the image
`''` (default) | string | cell array of strings

Style to use when rendering the image, specified as a string or cell array of strings. The `StyleName` must be a valid entry in the `Layer.Details.Style.Name` field. If you request multiple layers, each with a different style, then `StyleName` must be a cell array of strings.

Example: `[A,R] = wmsread(layer(1),'StyleName','style');`

Data Types: `char` | `cell`

### `'Transparent'` — Pixel transparency
`false` (default) | `true`

Pixel transparency, specified as a logical value, `true` or `false`. When you set `Transparent` to `true`, pixel transparency is enabled, meaning all pixels not representing features or data values are set to a transparent value. When you set `Transparent` to `false`, non-data pixels are set to the value of the background color.

Example: `[A,R] = wmsread(layer(1),'Transparent',true);`

Data Types: `logical`

### `'BackgroundColor'` — Color used for background (nondata) pixels of the map
`[255,255,255]` (default) | three-element vector

Color used for background (nondata) pixels of the map, specified as a three-element vector.

Example: `[A,R] = wmsread(layer(1),'BackgroundColor',[0,0,255]);`

Data Types: `single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64`

### `'Elevation'` — Desired elevation extent of the requested map
string

Desired elevation extent of the requested map, specified as a string. The layer must contain elevation data, which is indicated by the `'Name'` field of the `Layer.Details.Dimension` structure. The `'Name'` field must contain the value `'elevation'`. The `'Extent'` field of the `Layer.Details.Dimension` structure determines the permissible range of values for the parameter.

Example: `[A,R] = wmsread(layer(1),'Elevation','test');`

Data Types: `char`

### `'Time'` — Desired time extent of the requested map
string | numeric date number

Desired time extent of the requested map, specified as a string or numeric date number. The layer must contain data with a time extent, which is indicated by the `'Name'` field of the `Layer.Details.Dimension` structure. The `'Name'` field must contain the value `'time'`. The `'Extent'` field of the `Layer.Details.Dimension` structure determines the permissible range of values for the parameter. For more information about setting this parameter, see the `WMSMapRequest.Time` property reference page.

Example: `[A,R] = wmsread(layer(1),'Time','June 15, 2015');`

Data Types: `double | char`

### `'SampleDimension'` — Name of sample dimension
two-element cell array of strings

Name of dimension, specified as a two-element cell array of strings, other than `'time'` or `'elevation'` and its string value. The layer must contain data with a sample dimension extent, which is indicated by the `'Name'` field of the `Layer.Details.Dimension` structure. The `'Name'` field must contain the value of the first element of `'SampleDimension'`. The `'Extent'` field of the `Layer.Details.Dimension` structure determines the permissible range of values for the second element of `'SampleDimension'`.

Example: `[A,R] = wmsread(layer(1),'SampleDimension', {'sample','test'});`

Data Types: `cell`

### `'TimeoutInSeconds'` — Number of seconds to elapse before issuing a server time-out
60 (default) | scalar integer

Number of seconds to elapse before issuing a server time-out, specified as a scalar integer. If you set the value to `0`, `wmsread` ignores the time-out mechanism.

Example: `[A,R] = wmsread(layer(1),'TimeoutInSeconds',80);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### `A` — Color or grayscale image
real, nonsparse, 2-D matrix

Color or grayscale image, returned as a real, nonsparse, 2-D matrix.

### `R` — Geographic raster reference object
`map.rasterref.GeographicCellsReference` object

Geographic raster reference object, returned as a `map.rasterref.GeographicCellsReference` object. A raster referencing object ties the image `A` to the EPSG:4326 geographic coordinate system.

### `mapRequestURL` — WMS `GetMap` request URL
character string

WMS `GetMap` request URL, returned as a character string.

## More About

### Tips

- Establish an Internet connection to use `wmsread`. Periodically, the WMS server is unavailable. Retrieving the map can take several minutes. `wmsread` communicates with the server using a `WebMapServer` handle object representing a WMS server. The handle object acts as a proxy to a WMS server and resides physically on the client side. The handle object retrieves the map from the server. The handle object automatically times-out after 60 seconds if a connection is not made to the server.

- To specify a proxy server to connect to the Internet, select **File** > **Preferences** > **Web** and enter your proxy information. Use this feature if you have a firewall.

- `wmsread` supports reading data in WMS versions 1.0.0, 1.1.1, and 1.3.0. For version 1.3.0 only, the WMS specification states, "EPSG:4326 refers to WGS 84 geographic latitude, then longitude. That is, in this CRS the *x*-axis corresponds to latitude, and the *y*-axis to longitude." Most servers provide data in this manner; however, some servers conform to version 1.1.1, where the *x*-axis corresponds to longitude and the *y*-axis to latitude.

  `wmsread` attempts to validate whether a server is confirming to the specification. It checks the EPSG:4326 bounding box, and if the `XLim` values exceeds the range of latitude, then the axes are swapped to conform to version 1.1.1 rather than 1.3.0. If `wmsread` does not detect that the `XLim` values exceed the range of latitude and you notice that the latitude and longitude limits are reversed, then you need to swap them. You can either modify the `bbox` parameters in the `mapRequestURL` or modify the `Latlim` and `Lonlim` parameter values, if permissible.

## See Also

WebMapServer | WMSCapabilities | WMSLayer | WMSMapRequest | `wmsfind` | `wmsinfo` | `wmsupdate`

**Introduced before R2006a**

# wmsupdate

Synchronize WMSLayer object with server

## Syntax

```
[updatedLayers,index] = wmsupdate(layers)
[...] = wmsupdate(layers,Name,Value, ...)
```

## Description

`[updatedLayers,index] = wmsupdate(layers)` returns a `WMSLayer` array with its properties synchronized with values from the server. The input `layers` contains only one unique `ServerURL`. Layers no longer available on the server are removed. The logical array `index` contains `true` for each available layer. Therefore, `updatedLayers` has the same size as `layers(index)`. Except for deletion, `updatedLayers` preserves the same order of layers as `layers`.

`[...] = wmsupdate(layers,Name,Value, ...)` specifies parameter-value pairs that modify the request. Parameter names can be abbreviated and are case-insensitive.

The function accesses the Internet to update the properties. Periodically, the WMS server is unavailable. Updating the layer can take several minutes. The function times-out after 60 seconds if a connection is not made to the server.

## Input Arguments

**layers**

Array of `WMSLayer` objects

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**'TimeoutInSeconds'**

Integer-valued, scalar double that indicates the number of seconds before a server times out. A value of `0` causes the time-out mechanism to be ignored.

**Default:** 60 seconds

**'AllowMultipleServers'**

Logical scalar that indicates whether the `layer` array may contain elements from multiple servers. Use caution when setting the value to `true`, since you are making a request to each unique server. Each request can take several minutes to finish.

**Default:** `false` (indicates the array must contain elements from the same server)

## Output Arguments

**updatedLayers**

`WMSLayer` array with its properties synchronized with values from the server.

**index**

Logical array that contains `true` for each available layer.

## Examples

Update the layers from the NASA Goddard Space Flight Center WMS SVS Image Server:

```
% Search the abstract field of the updated layers to find
% layers containing the term 'blue marble'. Read and display
% the first blue marble layer containing the term '512' and
% 'image' in its LayerTitle.
gsfc = wmsfind('svs.gsfc.nasa.gov', 'SearchField', 'serverurl');
gsfc = wmsupdate(gsfc);
blue_marble = gsfc.refine('blue marble', 'SearchField', ...
    'abstract');
```

```
queryStr = '*512*image';
layers = blue_marble.refine(queryStr);
layer = layers(1);

% Display the layer and abstract.
[A, R] = wmsread(layer);
figure
worldmap world
plabel off; mlabel off
geoshow(A, R);
title(layer.LayerTitle)
disp(layer.Abstract)
```
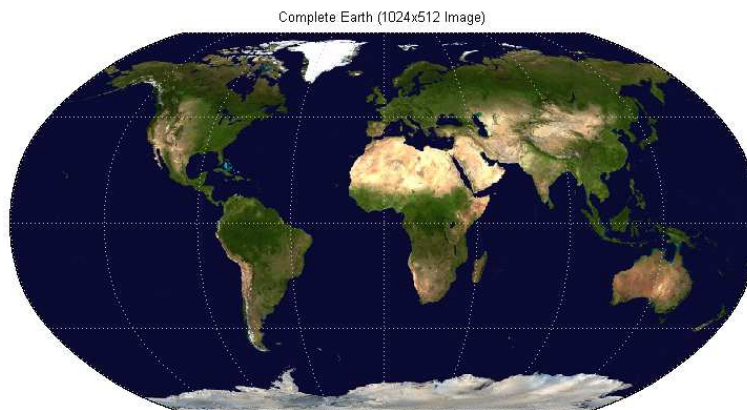


Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

Update the properties of all the layers from the NASA servers:

```
nasa = wmsfind('nasa', 'SearchField', 'serverurl');
nasa = wmsupdate(nasa, 'AllowMultipleServers', true);
```

## More About

### Tips

- To specify a proxy server to connect to the Internet, click **Preferences** and, in the Preferences dialog box, select **Web**. Enter your proxy information. Use this feature if you have a firewall.

## See Also

WMSLayer | wmsread | WebMapServer | wmsfind | wmsinfo

# worldFileMatrixToRefmat

Convert world file matrix to referencing matrix

## Syntax

```
refmat = worldFileMatrixToRefmat(W)
```

## Description

refmat = worldFileMatrixToRefmat(W) converts the 2-by-3 world file matrix W to a 3-by-2 referencing matrix refmat.

For a definition of a referencing matrix, see makerefmat.

For a definition of a world file matrix, see the worldFileMatrix method of the map raster reference and geographic raster reference classes.

### See Also
makerefmat | refmatToWorldFileMatrix

# worldfileread

Read world file and return referencing object or matrix

## Syntax

```
R = worldfileread(worldFileName, coordinateSystemType, rasterSize)
refmat = worldileread(worldFileName)
```

## Description

`R = worldfileread(worldFileName, coordinateSystemType, rasterSize)` reads the world file, `worldFileName`, and constructs a spatial referencing object, `R`. The type of referencing object is determined by the `coordinateSystemType` string, which can be either `'planar'` (including projected map coordinate systems) or `'geographic'` (for latitude-longitude systems). The `rasterSize` input should match the size of the image corresponding to the world file.

`refmat = worldileread(worldFileName)` reads the world file, `worldFileName`, and constructs a 3-by-2 referencing matrix, `refmat`.

## Examples

Read an ortho image referenced to a projected coordinate system (Massachusetts State Plane Mainland).

```
filename = 'concord_ortho_w.tif';
[X, cmap] = imread(filename);
worldFileName = getworldfilename(filename);
R = worldfileread(worldFileName, 'planar', size(X))
```

Read an image referenced to a geographic coordinate system.

```
filename = 'boston_ovr.jpg';
RGB = imread(filename);
worldFileName = getworldfilename(filename);
R = worldfileread(worldFileName, 'geographic', size(RGB))
```

## See Also

getworldfilename | map2pix | pix2map | worldfilewrite

# worldfilewrite

Write world file from referencing object or matrix

## Syntax

```
worldfilewrite(R, worldfilename)
```

## Description

worldfilewrite(R, worldfilename) calculates the world file entries corresponding to referencing object or matrix R and writes them into the file worldfilename. The argument R can be a map raster reference object, a geographic raster reference object, or a 3-by-2 referencing matrix.

## Examples

Write out the information from a referencing object for the image file concord_ortho_w.tif

```
info = imfinfo('concord_ortho_w.tif');
R = worldfileread('concord_ortho_w.tfw', ...
    'planar', [info.Height info.Width])
worldfilewrite(R, 'concord_ortho_w_test.tfw');
type concord_ortho_w_test.tfw
```

## See Also

getworldfilename | map2pix | pix2map | worldfileread

# worldmap

Construct map axes for given region of world

## Syntax

```
worldmap region
worldmap(region)
worldmap
worldmap(latlim, lonlim)
worldmap(Z, R)
h = worldmap(...)
```

## Description

`worldmap region` or `worldmap(region)` sets up an empty map axes with projection and limits suitable to the part of the world specified in `region`. `region` can be a string or a cell array of strings. Permissible strings include names of continents, countries, and islands as well as `'World'`, `'North Pole'`, `'South Pole'`, and `'Pacific'`.

`worldmap` with no arguments presents a menu from which you can select the name of a single continent, country, island, or region.

`worldmap(latlim, lonlim)` allows you to define a custom geographic region in terms of its latitude and longitude limits in degrees. `latlim` and `lonlim` are two-element vectors of the form `[southern_limit northern_limit]` and `[western_limit eastern_limit]`, respectively.

`worldmap(Z, R)` derives the map limits from the extent of a regular data grid georeferenced by `R`. `R` can be a geographic raster reference object, a referencing vector, or a referencing matrix.

If `R` is a geographic raster reference object, its `RasterSize` property must be consistent with `size(Z)`.

If `R` is a referencing vector, it must be a 1-by-3 with elements:

`[cells/degree northern_latitude_limit western_longitude_limit]`

If `R` is a referencing matrix, it must be 3-by-2 and transform raster row and column indices to/from geographic coordinates according to:

```
[lon lat] = [row col 1] * R
```

If `R` is a referencing matrix, it must define a (non-rotational, non-skewed) relationship in which each column of the data grid falls along a meridian and each row falls along a parallel.

`h = worldmap(...)` returns the handle of the map axes.

All axes created with `worldmap` are initialized with a spherical Earth model having a radius of 6,371,000 meters.

`worldmap` uses `tightmap` to adjust the axes limits around the map. If you change the projection, or just want more white space around the map frame, use `tightmap` again or `auto axis`

## Examples

### Example 1

Set up a world map and draw coarse coastlines:

```
worldmap('World')
load coast
plotm(lat, long)
```

### Example 2

Set up `worldmap` with land areas, major lakes and rivers, and cities and populated places:

```
ax = worldmap('World');
setm(ax, 'Origin', [0 180 0])
land = shaperead('landareas', 'UseGeoCoords', true);
geoshow(ax, land, 'FaceColor', [0.5 0.7 0.5])
lakes = shaperead('worldlakes', 'UseGeoCoords', true);
geoshow(lakes, 'FaceColor', 'blue')
rivers = shaperead('worldrivers', 'UseGeoCoords', true);
geoshow(rivers, 'Color', 'blue')
cities = shaperead('worldcities', 'UseGeoCoords', true);
```

```
geoshow(cities, 'Marker', '.', 'Color', 'red')
```
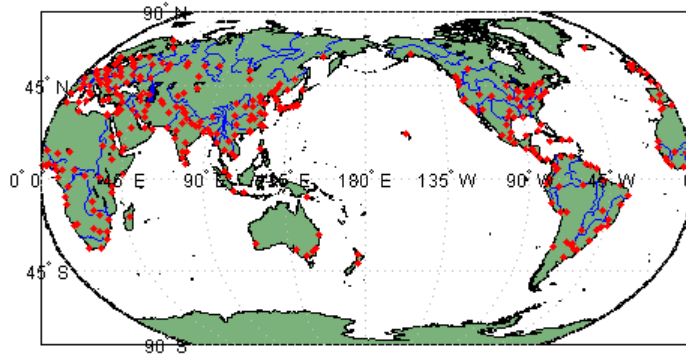


## Example 3

Draw a map of Antarctica:

```
worldmap('antarctica')
antarctica = shaperead('landareas', 'UseGeoCoords', true,...
  'Selector',{@(name) strcmp(name,'Antarctica'), 'Name'});
patchm(antarctica.Lat, antarctica.Lon, [0.5 1 0.5])
```

## Example 4

Draw a map of Africa and India with major cities and populated places:

```
worldmap({'Africa','India'})
land = shaperead('landareas.shp', 'UseGeoCoords', true);
geoshow(land, 'FaceColor', [0.15 0.5 0.15])
cities = shaperead('worldcities', 'UseGeoCoords', true);
geoshow(cities, 'Marker', '.', 'Color', 'red')
```

## Example 5

Make a map of the geoid over South America and the central Pacific:

```
worldmap([-50 50],[160 -30])
load geoid
geoshow(geoid, geoidrefvec, 'DisplayType', 'texturemap');
load coast
geoshow(lat, long)
```
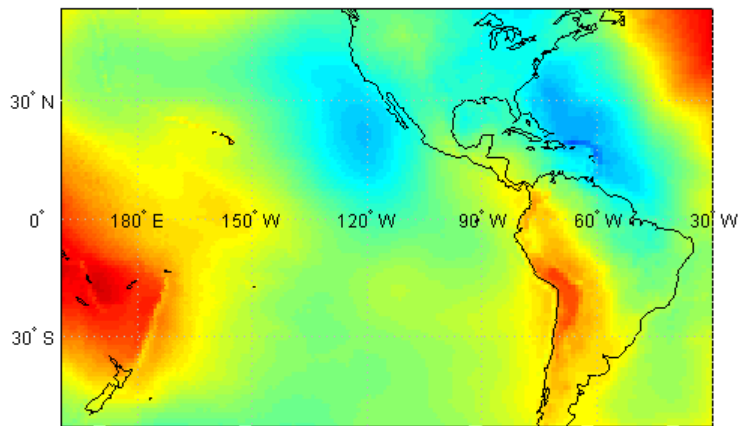


## Example 6

Draw a map of terrain elevations in Korea:

```
load korea
h = worldmap(map, refvec);
```

```
set(h, 'Visible', 'off')
geoshow(h, map, refvec, 'DisplayType', 'texturemap')
demcmap(map)
```

## Example 7

Make a map of the United States of America, coloring state polygons:

```
ax = worldmap('USA');
load coastlines
geoshow(ax, coastlat, coastlon,...
'DisplayType', 'polygon', 'FaceColor', [.45 .60 .30])
states = shaperead('usastatelo', 'UseGeoCoords', true);
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))}); % NOTE - colors are random
geoshow(ax, states, 'DisplayType', 'polygon', ...
  'SymbolSpec', faceColors)
```
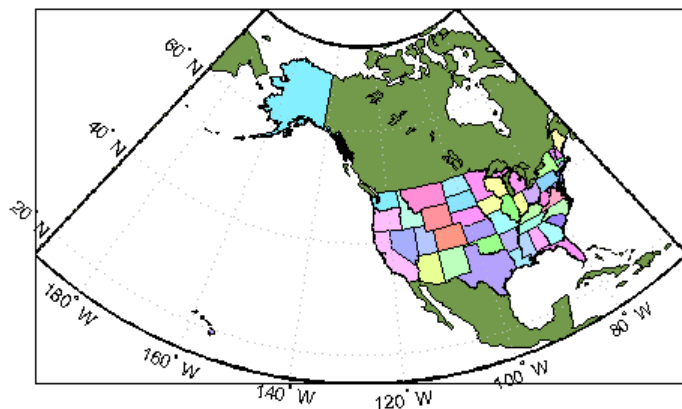


## See Also
axesm | framem | geoshow | gridm | mlabel | plabel | tightmap | usamap

# wrapTo180

Wrap angle in degrees to [-180 180]

## Syntax

```
lonWrapped = wrapTo180(lon)
```

## Description

`lonWrapped = wrapTo180(lon)` wraps angles in `lon`, in degrees, to the interval `[-180 180]` such that 180 maps to 180 and -180 maps to -180. (In general, odd, positive multiples of 180 map to 180 and odd, negative multiples of 180 map to -180.)

## See Also
`wrapTo360` | `wrapTo2Pi` | `wrapToPi`

# wrapTo360

Wrap angle in degrees to [0 360]

## Syntax

```
lonWrapped = wrapTo360(lon)
```

## Description

`lonWrapped = wrapTo360(lon)` wraps angles in `lon`, in degrees, to the interval `[0 360]` such that 0 maps to 0 and 360 maps to 360. (In general, positive multiples of 360 map to 360 and negative multiples of 360 map to zero.)

## See Also
`wrapTo180` | `wrapTo2Pi` | `wrapToPi`

# wrapTo2Pi

Wrap angle in radians to [0 2*pi]

## Syntax

```
lambdaWrapped = wrapTo2Pi(lambda)
```

## Description

`lambdaWrapped = wrapTo2Pi(lambda)` wraps angles in `lambda`, in radians, to the interval `[0 2*pi]`, such that 0 maps to 0 and `2*pi` maps to `2*pi`. (In general, positive multiples of `2*pi` map to `2*pi` and negative multiples of `2*pi` map to 0.)

## See Also

`wrapTo180` | `wrapTo360` | `wrapToPi`

# wrapToPi

Wrap angle in radians to [−pi pi]

## Syntax

```
lambdaWrapped = wrapToPi(lambda)
```

## Description

`lambdaWrapped = wrapToPi(lambda)` wraps angles in `lambda`, in radians, to the interval `[-pi pi]`. `pi` maps to `pi` and `-pi` maps to `-pi`. (In general, odd, positive multiples of `pi` map to `pi` and odd, negative multiples of `pi` map to `-pi`.)

## See Also

`wrapTo180` | `wrapTo360` | `wrapTo2Pi`

# zdatam

Adjust *z*-plane of displayed map objects

## Syntax

```
zdatam
zdatam(hndl)
zdatam('str')
zdatam(hndl,zdata)
zdatam('str',zdata)
```

## Description

`zdatam` displays a GUI for selecting an object from the current axes and modifying its `ZData` property.

`zdatam(hndl)` and `zdatam('str')` display a GUI to modify the `ZData` of the object(s) specified by the input. `str` is any string recognized by `handlem`.

`zdatam(hndl,zdata)` alters the *z*-plane position of displayed map objects designated by the MATLAB graphics handle `hndl`. The *z*-plane position may be the `Z` position in the case of text objects, or the `ZData` property in the case of other graphic objects. The function behaves as follows:

- If `hndl` is an `hggroup` handle, the `ZData` property of the children in the `hggroup` are altered.
- If the handle is scalar, then `ZData` can be either a scalar (*z*-plane definition), or a matrix of appropriate dimension for the displayed object.
- If `hndl` is a vector, then `ZData` can be a scalar or a vector of the same dimension as `hndl`.
- If `ZData` is a scalar, then all objects in `hndl` are drawn on the `ZData` *z*-plane.
- If `ZData` is a vector, then each object in `hndl` is drawn on the plane defined by the corresponding `ZData` element.
- If `ZData` is omitted, then a modal dialog box prompts for the `ZData` entry.

zdatam('*str*',zdata) identifies the objects by the input str, where str is any string recognized by handelm, and uses zdata as described above to update their ZData property.

This function adjusts the *z*-plane position of selected graphics objects. It accomplishes this by setting the objects' ZData properties to the appropriate values.

## See Also
handlem | setm

# zero22pi

Wrap longitudes to [0 360] degree interval

---

**Note:** The `zero22pi` function has been replaced by `wrapTo360` and `wrapTo2Pi`.

---

## Syntax

```
newlon = zero22pi(lon)
newlon = zero22pi(lon,angleunits)
```

## Description

`newlon = zero22pi(lon)` *wraps* the input angle `lon` in degrees to the 0 to 360 degree range.

`newlon = zero22pi(lon,angleunits)` works in the units defined by the string *angleunits*, which can be either `'degrees'` or `'radians'`. *angleunits* can be abbreviated and is case-insensitive.

## Examples

```
zero22pi(567.5)

ans =
    207.5

zero22pi(-567.5)

ans =
    152.5

zero22pi(-7.5,'radian')
```

```
ans =
    5.0664
```

## See Also
wrapTo2Pi | wrapTo360

## zerom

Construct regular data grid of 0s

## Compatibility

`zerom` is not recommended and may be removed in a future release. Instead, use `georefcells` to construct a geographic raster reference object and then use `zeros` to initialize a data grid of the appropriate size:

```
R = georefcells(latlim,lonlim,1/scale,1/scale);
Z = zeros(R.RasterSize);
```

## Syntax

```
[Z,refvec] = zerom(latlim,lonlim,scale)
```

## Description

`[Z,refvec] = zerom(latlim,lonlim,scale)` returns a full regular data grid consisting entirely of 0s and a three-element referencing vector for the returned Z. The two-element vectors `latlim` and `lonlim` define the latitude and longitude limits of the geographic region. They should be of the form `[south north]` and `[west east]`, respectively. The scalar `scale` specifies the number of rows and columns per degree of latitude and longitude.

## Examples

```
[Z,refvec] = zerom([46,51],[-79,-75],1)

Z =
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

```
refvec =
     1     51    -79
```

## See Also

limitm | nanm | onem | sizem | spzerom

# axesmui

Define map axes and modify map projection and display properties

## Activation

| Command Line | Maptool | Map Display |
|---|---|---|
| axesmui<br><br>c = axesmui(...) | **Display** > **Projection** | extend-click map display |

## Description

axesmui activates a Projection Control dialog box for the current map axes. The dialog box allows map projection definition and property modification.

c is an optional output argument that indicates whether the Projection Control dialog box was closed by the cancel button. c = 1 if the cancel button is pushed. Otherwise, c = 0.

Extend-clicking a map display brings up the Projection Control dialog box for that map axes.

# Controls



The **Map Projection** pull-down menu is used to select a map projection. The projections are listed by type, and each is preceded by a four-letter type indicator:

```
Cyln = Cylindrical
Pcyl = Pseudocylindrical
Coni = Conic
Poly = Polyconic
Pcon = Pseudoconic
Azim = Azimuthal
Mazi = Modified Azimuthal
Pazi = Pseudoazimuthal
```

The **Zone** button and edit box are used to specify the UTM or UPS zone. For non-UTM and UPS projections, the two are disabled.

The **Geoid** edit boxes and pull-down menu are used to specify the geoid. Units must be in meters for the UTM and UPS projections, since this is the standard unit for the two projections. For non-UTM and UPS projections, the geoid unit can be anything, bearing in mind that the resulting projected data will be in the same units as the geoid.

The **Angle Units** pull-down menu is used to specify the angle units used on the map projection. All angle entries corresponding to the current map projection must be entered in these units. Current angle entries are automatically updated when new angle units are selected.

The **Map Limits** edit boxes are used to specify the extent of the map data in geographic coordinates. The **Latitude** edit boxes contain the southern and northern limits of the map. The **Longitude** edit boxes contain the western and eastern limits of the map. The map limits establish the extent of the meridian and parallel grid lines, regardless of the display settings (see grid settings). Map limits are always in geographic coordinates, regardless of the map origin and orientation setting. In the normal aspect, the map display is trimmed to the minimum of the map and frame limits.

The **Frame Limits** edit boxes are used to specify the location of the map frame, measured from the center of the map projection in the base coordinate system. The **Latitude** edit boxes contain the southern and northern frame edge locations. The **Longitude** edit boxes contain the western and eastern frame edge locations. Displayed map data are trimmed at the frame limits. For azimuthal map projections, the latitude limits should be set to −inf and the desired trim distance from the map origin. In the normal aspect, the map display is trimmed to the minimum of the map and frame limits.

The **Map Origin** edit boxes are used to specify the origin and aspect angle of the map projection. The **Lat** and **Long** boxes specify the map origin in geographic coordinates. This is the point that is placed in the center of the projection. If either box is left blank, 0 degrees is used. The **Orientation** box specifies the azimuth angle of the North Pole relative to the map origin. Azimuth is measured clockwise from the top of the projection. If the **Orientation** box is disabled, then the selected map projection requires a fixed orientation. See the *Mapping Toolbox User's Guide* for a complete description of the map origin.

The **Cartesian Origin** edit boxes are used to specify the *x-y* offset, along with a desired scale factor of the map projection. The **False E and N** boxes specify the false easting and northing in Cartesian coordinates. These must be in the same units as the geoid. The **Scalefactor** box specifies the scale factor used in the map projection calculations.

The **Parallels** edit boxes specify the standard parallels of the selected map projection. A particular map projection may have one or two standard parallels. If the edit boxes are

disabled, then the selected projection has no standard parallels or the standard parallels are fixed.

The **Aspect** pull-down menu is used to select a normal or transverse display aspect. When the aspect is normal, *north* (on the base projection) is up, and the map is displayed in a *portrait* setting. In a transverse aspect, north (in the base projection) is to the right, and the map is displayed in a *landscape* setting. This property does not control the map projection aspect. The projection aspect is determined by the map `Origin` property).

The **Frame** button brings up the Map Frame Properties dialog box, which allows the map frame settings to be modified.

The **Grid** button brings up the Map Grid Properties dialog box, which allows the map grid settings to be modified.

The **Labels** button brings up the Map Label Properties dialog box, which allows the parallel and meridian label settings to be modified.

The **Fill in** button is used to compute projection and display settings based on any currently specified map parameters. Only settings that are left blank are affected when this button is pushed.

The **Reset** button is used to reset the default projection properties and display settings of the current map. Default display settings include frame, grid, and label properties set to `'off'`.

The **Apply** button is used to apply the projection and display settings to the current map, which results in the map being reprojected.

The **Help** button is used to bring up online help text for each control on the Projection Control dialog box.

The **Cancel** button disregards any modified projection or display settings and closes the Projection Control dialog box.

## Map Frame Properties Dialog Box

This dialog box allows modification of the map frame settings. It is accessed via the **Frame** button on the Projection Control dialog box.
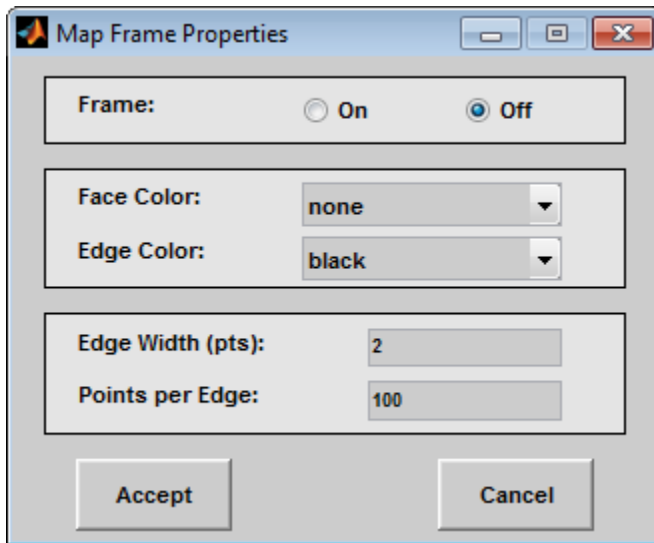
The **Frame** selection buttons determine whether the map frame is visible.

The **Face Color** pull-down menu is used to select the background color of the map frame. Selecting none results in a transparent frame background, i.e., the same as the axes color. Selecting custom allows a custom RGB triple to be defined for the background color.

The **Edge Color** pull-down menu is used to select the color of the frame edge. Selecting none hides the frame edge. Selecting custom allows a custom RGB triple to be defined for the edge color.

The **Edge Width** edit box is used to enter the line width of the frame edge, in points.

The **Points per Edge** edit box is used to enter the number of points used to display each edge of the map frame.

The **Accept** button accepts any modifications made to the map frame properties and returns to the Projection Control dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the map frame properties and returns to the Projection Control dialog box.

## Map Grid Properties Dialog Box

This dialog box allows modification of the map frame settings. It is accessed via the **Grid** button on the Projection Control dialog box.
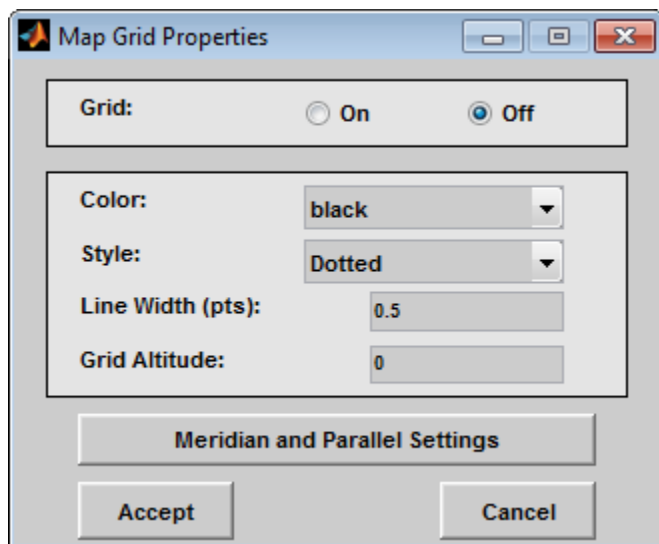


The Grid selection buttons determine whether the map grid is visible.

The **Color** pull-down menu is used to select the color of the map grid lines. Selecting custom allows a custom RGB triple to be defined for the grid line color.

The **Style** pull-down menu is used to select the line style of the map grid lines.

The **Line Width** edit box is used to enter the width of the map grid lines, in points.

The **Grid Altitude** edit box is used to enter *z*-axis location of the map grid. This property can be used to place some mapped objects above or below the map grid. The default map grid altitude is inf, which places the grid above all other mapped objects.

The **Meridian and Parallel Settings** button brings up the **Meridian and Parallel Properties** dialog box, which allows the properties of the meridian and parallel grid lines to be modified.

The **Accept** button accepts any modifications made to the map grid properties and returns to the Projection Control dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the map grid properties and returns to the Projection Control dialog box.

## Meridian and Parallel Properties Dialog Box

This dialog box is used to modify the settings for meridian and parallel grid lines. It is accessed via the **Meridian and Parallel Settings** button on the Map Grid Properties dialog box.
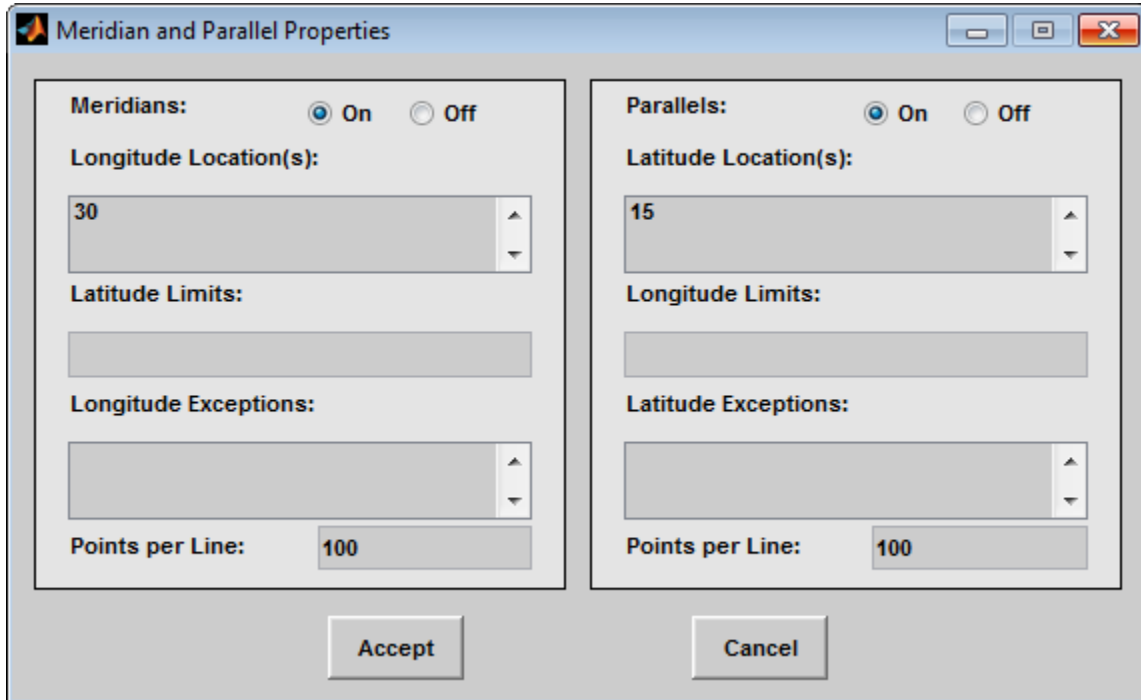


The **Meridians** selection buttons determine whether the meridian grid lines are visible when the map grid is turned on.

The **Longitude Location(s)** edit box is used to specify which meridians are to be displayed if the meridian lines are turned on. If a scalar interval value is entered,

meridian lines are displayed at that interval, starting from the Prime Meridian and proceeding in east and west directions. If a vector of values is entered, meridian lines are displayed at locations given by each element of the vector.

The **Latitude Limits** edit box is used to specify the latitude limits beyond which meridian lines do not extend. If this property is left empty, all meridian lines extend to the map latitude limits (specified by the Latitude Map Limits entry on the **Projection Control** dialog box). This entry must be a two-element vector enclosed in brackets.

The **Longitude Exceptions** edit box is used to enter specific meridians of the displayed grid that are to extend beyond the latitude limits, to the map limits. This entry is a vector of longitude values.

The **Parallels** selection buttons determine whether the parallel grid lines are visible when the map grid is turned on.

The **Latitude Location(s)** edit box is used to specify which parallels are to be displayed if the parallel lines are turned on. If a scalar interval value is entered, parallel lines are displayed at that interval, starting from the Equator and proceeding in north and south directions. If a vector of values is entered, parallel lines are displayed at locations given by each element of the vector.

The **Longitude Limits** edit box is used to specify the longitude limits beyond which parallel lines do not extend. If this property is left empty, all parallel lines extend to the map longitude limits (specified by the Longitude Map Limits entry on the **Projection Control** dialog box). This entry must be a two-element vector enclosed in brackets.

The **Latitude Exceptions** edit box is used to enter specific parallels of the displayed grid that are to extend beyond the longitude limits, to the map limits. This entry is a vector of latitude values.

The **Points per Line** edit boxes are used to enter the number of points used to plot each meridian and each parallel grid line. The default value is 100 points.

The **Accept** button accepts any modifications that have been made to the meridian and parallel grid line properties and return to the Map Grid Properties dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the meridian and parallel grid lines and returns to the Map Grid Properties dialog box.

## Map Label Properties Dialog Box

This dialog box is used to modify the settings of the meridian and parallel labels. It is accessed via the **Label** button on the Projection Control dialog box.
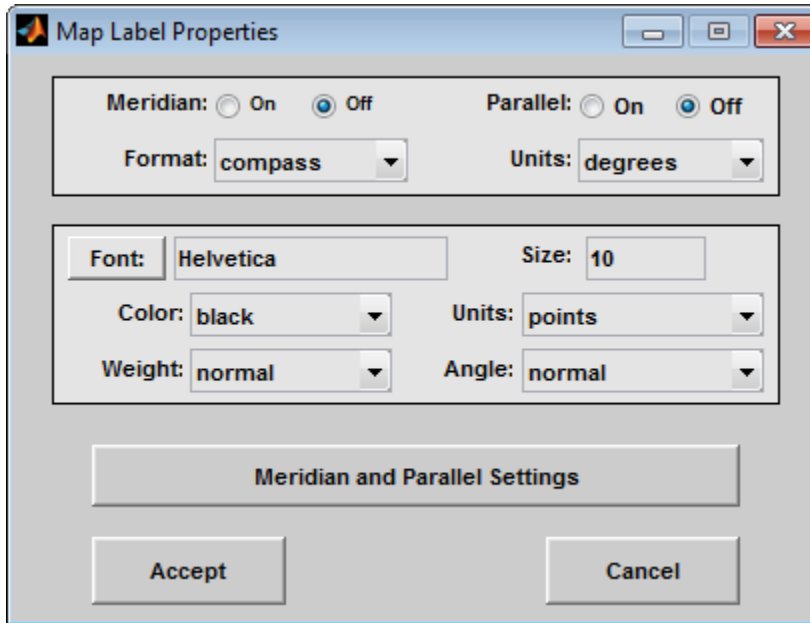


The **Meridian** and **Parallel** selection buttons determine whether the meridian and parallel labels are visible.

The **Format** pull-down menu is used to specify the format of the grid labels. If `compass` is selected, meridian labels are appended with `E` for east and `W` for west, and parallel labels are appended with `N` for north and `S` for south. If `signed` is chosen, meridian labels are prefixed with `+` for east and `-` for west, and parallel labels are prefixed with `+` for north and `-` for south. If `none` is selected, western meridian labels and southern parallel labels are prefixed by `-`, but no symbol precedes eastern meridian labels and northern parallel labels.

The label **Units** pull-down menu is used to specify the angle units used to display the parallel and meridian labels. These units, used for display purposes only, need not be the same as the angle units of the map projection.

The **Font** edit box is used to specify the character font used to display the parallel and meridian labels. If the font specified does not exist on the computer, the default of `Helvetica` is used. Pressing the **Font** button previews the selected font.

The font **Size** edit box is used to enter an integer value that specifies the font size of the parallel and meridian labels. This value must be in the units specified by the font **Units** pull-down menu.

The font **Color** pull-down menu is used to select the color of the parallel and meridian labels. Selecting `custom` allows a custom RGB triple to be defined for the labels.

The font **Weight** pull-down menu is used to specify the character weight of the parallel and meridian labels.

The font **Units** pull-down menu is used to specify the units used to interpret the font size entry. When set to `normalized`, the value entered in the **Size** edit box is interpreted as a fraction of the height of the axes. For example, a normalized font size of 0.1 sets the label text to a height of one tenth of the axes height.

The font **Angle** pull-down menu is used to select the character slant of the parallel and meridian labels. `normal` specifies nonitalic font. `italic` and `oblique` specify italic font.

The **Meridian and Parallel Settings** button brings up the Meridian and Parallel Label Properties dialog box, which allows modification of properties specific to the meridian and parallel grid labels.

The **Accept** button accepts any modifications that have been made to the map label properties and returns to the Projection Control dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the map labels and returns to the **Projection Control** dialog box.

## Meridian and Parallel Label Properties Dialog Box

This dialog box is used to modify properties specific to the meridian and parallel grid labels. It is accessed via the **Meridian and Parallel Settings** button on the Map Label Properties dialog box.
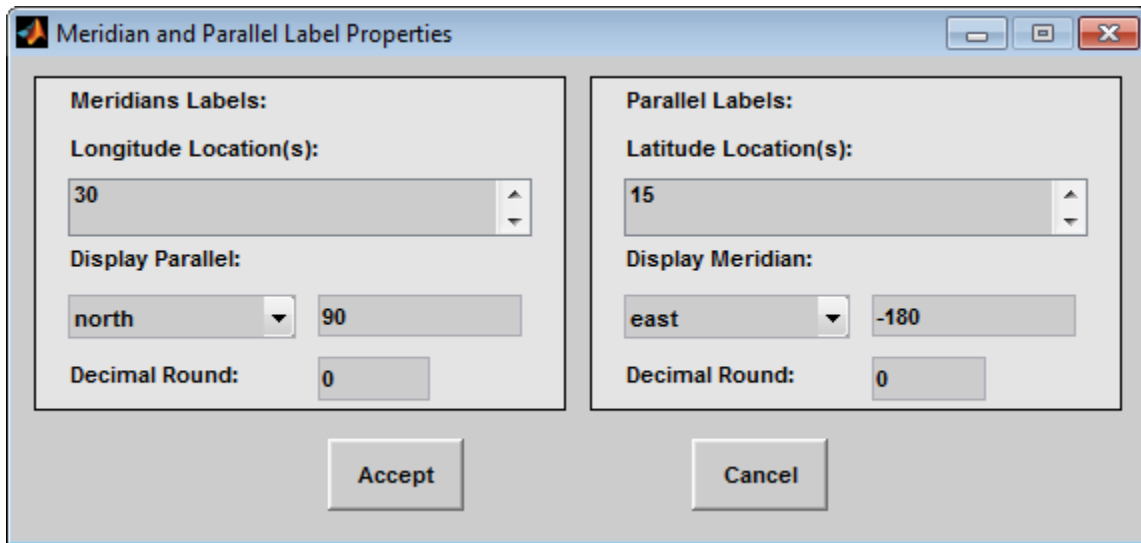
The **Longitude Location(s)** edit box is used to specify which meridians are to be labeled. Meridian labels need not coincide with displayed meridian grid lines. If a scalar interval value is entered, labels are displayed at that interval, starting from the Prime Meridian and proceeding in east and west directions. If a vector of values is entered, labels are displayed at longitude locations given by each element of the vector.

The **Display Parallel** pull-down menu and edit box are used to specify the latitude location of the meridian labels. If a scalar latitude value is provided in the edit box, the meridian labels are placed at that parallel. Alternatively, the pull-down menu can be used to select a latitude location. If north is chosen, meridian labels are placed at the maximum map latitude limit. If south is chosen, meridian labels are placed at the minimum map latitude limit.

The **Latitude Location(s)** edit box is used to specify which parallels are to be labeled. Parallel labels need not coincide with displayed parallel grid lines. If a scalar interval value is entered, labels are displayed at that interval, starting from the Equator and proceeding in north and south directions. If a vector of values is entered, labels are displayed at latitude locations given by each element of the vector.

The **Display Meridian** pull-down menu and edit box are used to specify the longitude location of the parallel labels. If a scalar longitude value is provided in the edit box, the parallel labels are placed at that meridian. Alternatively, the pull-down menu can be used to specify a longitude location. If east is chosen, parallel labels are placed at

the maximum map longitude limit. If `west` is chosen, parallel labels are placed at the minimum map longitude limit.

The **Decimal Round** edit boxes are used to specify the power of ten to which the meridian and parallel labels are rounded. For example, a value of -1 results in labels displayed to the tenths decimal place.

The **Accept** button accepts any modifications that have been made to the meridian and parallel label properties and return to the Map Label Properties dialog box. Changes are applied to the current map only when the **Apply** button on the **Projection Control** dialog box is pushed.

The **Cancel** button disregards any modifications to the meridian and parallel labels and returns to the Map Label Properties dialog box.

The **Map Geoid** edit box is used to specify the geoid (ellipsoid) definition for the current map axes. The geoid is defined by a two-element vector of the form `[semimajor-axis eccentricity]`. Eccentricity must be a value between 0 and 1, but not equal to 1. A nonzero eccentricity represents an ellipsoid. The default geoid is a sphere with radius 1, represented as `[1 0]`. If a scalar entry is provided, it is assumed to be the radius of a sphere.

The **Accept** button accepts any modifications that have been made to the map geoid and return to the Projection Control dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the map geoid and returns to the Projection Control dialog box.

## See Also
axesm

# clmo-ui

GUI to clear mapped objects

## Activation

| Command Line | Maptool |
|---|---|
| clmo | **Tools** > **Delete** > **Object** |

## Description

clmo brings up a Select Object dialog box for selecting mapped objects to delete.

## Controls

The scroll box is used to select the desired objects from the list of mapped objects.



Pushing the **Select all** button highlights all objects in the scroll box for selection.
Pushing the **OK** button deletes the selected objects from the map. Pushing the **Cancel**
button aborts the operation.

## See Also

clmo

# clrmenu

Add colormap menu to figure window

## Activation

| Command Line |
| --- |
| clrmenu |
| clrmenu(h) |

## Description

clrmenu adds a colormap menu to the current figure.

clrmenu(h) adds a colormap menu to the figure specified by the handle h.

## Controls

The following choices are included on the colormap menu:

**Parula**, **Gray**, **Hsv**, **Hot**, **Pink**, **Cool**, **Bone**, **Jet**, **Copper**, **Spring**, **Summer**, **Autumn**, **Winter**, **Flag**, and **Prism** generate colormaps.

**Rand** is a random colormap.

**Brighten** increases the brightness.

**Darken** decreases the brightness.

**Flipud** inverts the order of the colormap entries.

**Fliplr** interchanges the red and blue components.

**Permute** permutes the colormap: red > blue, blue > green, green > red.

**Define** allows a workspace variable to be specified for the colormap.

**Remember** stores the current colormap.

**Restore** reverts to the stored colormap (initially, the stored colormap is the colormap in use when `clrmenu` is invoked).

**Refresh** redraws the current figure window.

**Digital Elevation** activates the DEM Colormap Input dialog box. Use it to specify a colormap for a digital elevation map, and then apply the colormap to the current figure. The number of land and sea colors in the colormap is appropriate for the maximum elevations and depths of the data grid. The dialog box is shown and described below:
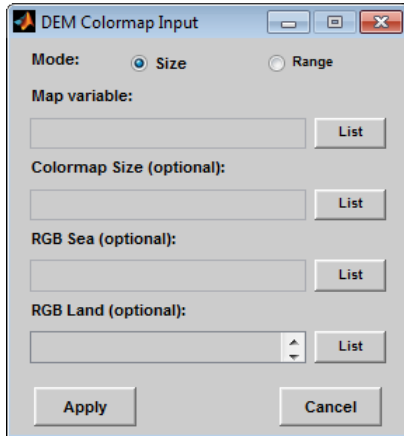


The **Mode** selection buttons are used to specify whether the length of the colormap is specified or whether the altitude range increment assigned to each color is specified.

The **Map variable** edit box is used to specify the data grid containing the elevation data.

The **Colormap Size** edit box is used in Size mode. This entry defines the length of the colormap. If omitted, a default length of 64 is used. This entry must be a scalar value.

The **Altitude Range** edit box is used in Range mode. This entry defines the altitude range increment assigned to each color. If omitted, a default increment of 100 is used. This entry must be a scalar value.

The **RGB Sea** edit box is used to define colors for data with negative values. The actual sea colors of the generated colormap are interpolated from this matrix. This entry can be a matrix of any length (n by 3). The colormap matrix of the current figure can be used

by entering the string `'window'` in this box. The `demcmap` function provides default sea colors, which are used if this entry is left blank.

The **RGB Land** edit box is used to define colors for data with positive values. The actual land colors of the generated colormap are interpolated from this matrix. This entry can be a matrix of any length (n by 3). The colormap matrix of the current figure can be used by entering the string `'window'` in this box. The `demcmap` function provides default sea colors, which are used if this entry is left blank.

Pressing the **Apply** button accepts the input data, creates the colormap, and assigns it to the current figure.

Pressing the **Cancel** button disregards any input data and closes the DEM Colormap Input dialog box.

## See Also
demcmap

# colorm

Create index map colormaps

## Compatibility

`colorm` will be removed in a future release.

## Syntax

```
colorm(datagrid,refvec)
```

## Description

`colorm(datagrid,refvec)` displays the data grid in a new figure window and allows a colormap to be edited and saved to a new variable. `datagrid` and `refvec` are the data grid and the referencing vector of the surface. `map` must have positive index values into the colormap.

## Controls
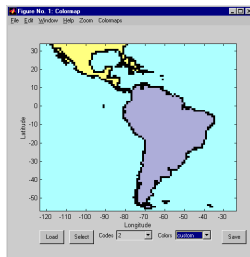


The `colorm` tool displays the surface map data in a new figure window with the current colormap. **Zoom** and **Colormaps** menus are activated for that figure.

The **Zoom On/Off** menu toggles interactive zoom on and off.

The **Colormaps** menu provided a variety of colormap options that can be applied to the map. See `clrmenu` in this guide for a description of the **Colormaps** menu options.

The **Load** button activates a dialog box, used to specify a colormap variable to be applied to the displayed surface map. This colormap can then be edited and saved.

The **Select** button activates the mouse cursor and allows a point on the map to be selected. The value of that point then appears in the **Codes** pull-down menu. The color of the selected point appears in the **Color** pull-down menu and can then be edited.

The **Codes** pull-down menu is used to select a particular value in the data grid. The color associated with that value then appears in the **Color** pull-down menu and can be edited.

The **Color** pull-down menu is used to select a particular color to assign to the value currently displayed in the Codes pull-down menu. A custom color can be defined by selecting the `custom` option. This brings up a custom color interface with which an RGB triple can be selected.

The **Save** button is used to save the modified colormap to the workspace. A dialog box appears in which the colormap variable name is entered.

## See Also

`encodem` | `getseeds` | `maptrim` | `seedm`

# demdataui

UI for selecting digital elevation data

## Activation

```
demdataui
```

## Description

`demdataui` is a graphical user interface to extract digital elevation map data from a number of external data files. You can extract data to MAT-files or the base workspace as regular data grids with referencing vectors.

The `demdataui` panel lets you read data from a variety of high-resolution digital elevation maps (DEMs). These DEMs range in resolution from about 10 kilometers to 100 meters or less. The data files are available over the Internet at no cost, or (in some cases) on CD-ROMs for varying fees. `demdataui` reads ETOPO5, TerrainBase, GTOPO30, GLOBE, satellite bathymetry, and DTED data. See the links under See Also for more information on these data sets. `demdataui` looks for these geospatial data files on the MATLAB path and, for some operating systems, on CD-ROM disks.

You use the list to select the source of data and the map to select the region of interest. When you click the **Get** button, data is extracted and displayed on the map. Use the **Save** button to save the data in a MAT-file or to the base workspace for later display. The **Close** button closes the window.

# Controls
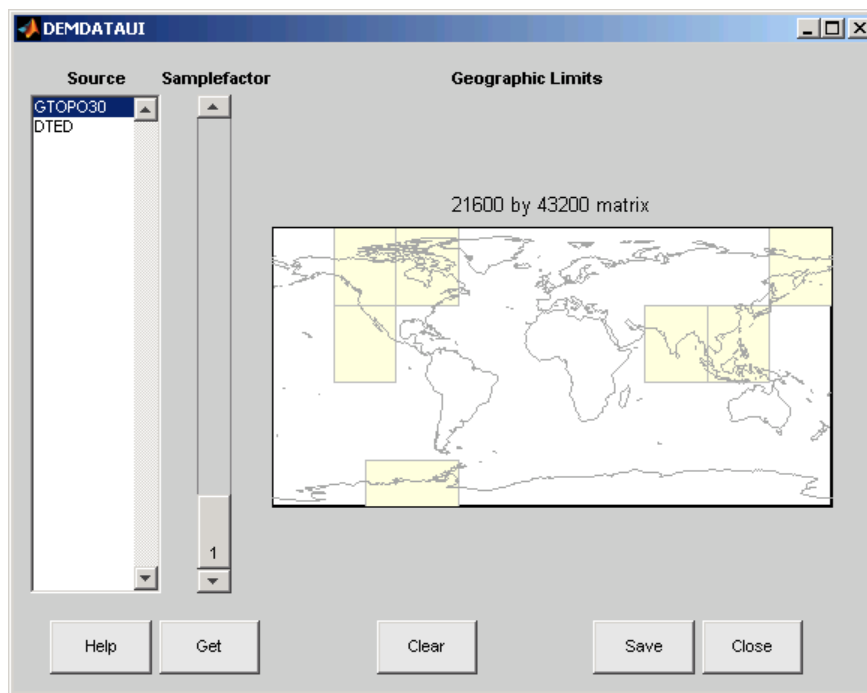


## The Map

The map controls the geographic extent of the data to be extracted. demdataui extracts data for areas currently visible on the map. Use the mouse to zoom in or out to the area of interest. See zoom for more on zooming.

Some data sources divide the world up into tiles. When extracting, data is concatenated across all visible tiles. The map shows the tiles in light yellow with light gray edges. When data resolution is high, extracting data for large area can take much time and memory. An approximate count of the number of points is shown above the map. Use the **Samplefactor** slider to reduce the amount of data.

## The List

The list controls the source of data to be extracted. Click a name to see the geographic coverage in light yellow. The sources list shows the data sources found when `demdataui` started.

`demdataui` searches for data files on the MATLAB path. On some computers, `demdataui` also checks for data files on the root level of letter drives. `demdataui` looks for the following data: `etopo5`: `new_etopo5.bil` or `etopo5.northern.bat` and `etopo5.southern.bat` files. `tbase`: `tbase.bin` file. `satbath`: `topo_6.2.img` file. `gtopo30`: a folder that contains subfolders with the data files. For example, `demdataui` would detect `gtopo30` data if a folder on the path contained the folders E060S10 and E100S10, each of which holds the uncompressed data files. `globedem`: a folder that contains data files and in the subfolder `/esri/hdr` and the `*.hdr` header files. `dted`: a folder that has a subfolder named DTED. The contents of the DTED folder are more subfolders organized by longitude and, below that, the DTED data files for each latitude tile. See the help for functions with the data source names for more on the data attributes and internet locations.

## The Samplefactor Slider

The **Sample Factor** slider allows you to reduce the density of the data. A sample factor of 2 returns every second point. The current sample factor is shown on the slider.

## The Get Button

The **Get** button reads the currently selected data and displays it on the map. Press the standard interrupt key combination for your platform to interrupt the process.

## The Clear Button

The **Clear** button removes any previously read data from the map.

## The Save Button

The **Save** button saves the currently displayed data to a MAT-file or the base workspace. If you choose to save to a file, you will be prompted for a file name and location. If you choose to save to the base workspace, you can choose the variable name under which the data will be stored.

Data are returned as Mapping Toolbox Version 1 display structures. For information about display structure format, see "Version 1 Display Structures" on page 1-200 in the reference page for `displaym`.

Use `load` and `displaym` to redisplay the data from a file on a map axes. To display the data in the base workspace, use `displaym`. To gain access to the data matrices, subscript into the structure (for example, `datagrid = demdata(1).map; refvec = demdata(1).maplegend`. Use `worldmap` to create easy displays of the elevation data (for example, `worldmap(datagrid,refvec)`. Use `meshm` to add regular data grids to existing displays, or `surfm` or a similar function for geolocated data grids (for example, `meshm(datagrid,refvec)` or `surfm(latgrat,longrat,z)`.

## The Close Button

The **Close** button closes the `demdataui` panel.

## See Also
`etopo` | `tbase` | `gtopo30` | `globedem` | `dted` | `satbath` | `vmap0ui`

# handlem-ui

GUI for handles of specified mapped objects

## Activation

| Command Line |
|---|
| `h = handlem` |
| `h = handlem('prompt')` |

## Description

`h = handlem` brings up a Select Object dialog box, which lists all currently displayed objects. Objects can be selected and their handles returned.

`h = handlem('prompt')` brings up a Specify Object dialog box, which allows greater control of object selection.

## Controls

Select Object Dialog Box



The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button returns the object handles in the variable `h`. Pushing the **Cancel** button aborts the operation.

Specify Object Dialog Box

The **Object** Controls are used to select an object type or tag. The **Name** pull-down menu is used to select from a list of predefined object strings. The **Other Tag** edit box is used to specify an object tag not listed in the **Name** pull-down menu. Pushing the **Select** button brings up the Select Object dialog box, which shows only the currently displayed objects for selection.

The **Match** Controls are used when a Handle Graphics object type (image, line, surface, patch, or text) is specified. The **Untagged Objects** selection button is used to return the handles of only those objects with empty tag properties. The **All Objects** selection button is used to return all object handles of the specified type, regardless of whether they are tagged.

Pushing the **Apply** button returns the handles of the specified objects. Pushing the **Cancel** button aborts the operation.

## See Also
handlem

# hidem-ui

Hide specified mapped objects

## Activation

| Command Line | Maptool |
|---|---|
| hidem | **Tools** > **Hide** > **Object** |

## Description

hidem brings up a Select Object dialog box for selecting mapped objects to hide (Visible property set to 'off').

## Controls



The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button changes the Visible property of the selected objects to 'off'. Pushing the **Cancel** button aborts the operation without changing any properties of the selected objects.

### See Also

hidem

# lightmui

Control position of lights on globe or 3-D map

---

**Note:** `lightmui` has been removed. Use `lightm` instead.

---

## Syntax

```
lightmui(hax)
```

## Description

`lightmui(hax)` creates a GUI to control the position of lights on a globe or 3-D map in map axes `hax`. You can control the position of lights by clicking and dragging the icon or by dialog boxes. Right-click the appropriate icon in the GUI to invoke the corresponding dialog box. You can change the light color by entering the RGB components manually or by clicking the pushbutton.

## See Also

`lightm`

# maptool

Add menu-activated tools to map figure

## Activation

| Command Line |
|---|
| maptool(*PropertyName*,PropertyValue) |
| maptool(*ProjectionFile*,...) |
| h = maptool(...) |

## Description

`maptool` adds several items to the menu in the current figure window with a map axes and opens the Projection Control dialog box for defining map projection and display properties. The figure window features a special menu bar that provides access to most of Mapping Toolbox GUIs.

`maptool(*PropertyName*,PropertyValue,...)` creates a figure window with a map axes defined by the supplied map properties. The `MapProjection` property must be the first input pair. `maptool` supports the same map properties as `axesm`.

`maptool(*ProjectionFile*,*PropertyName*, PropertyValue,...)` allows for the omission of the `MapProjection` property name. *ProjectionFile* must be the identifying string of an available map projection.

`h = maptool(...)` returns a two-element vector containing the handle of the `maptool` figure window and the handle of the map axes.

# Controls
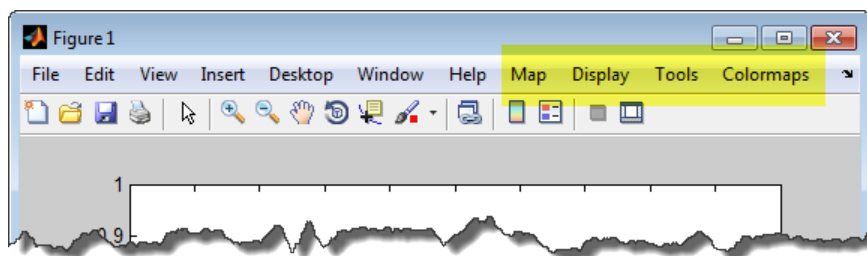


## Map Menu

The **Lines** option activates the Line Map Input dialog box for projecting two- and three-dimensional line objects onto the map axes.

The **Patches** option activates the Patch Map Input dialog box for projecting patch objects onto the map axes.

The **Regular Surfaces** option activates the Mesh Map Input dialog box for projecting a regular data grid onto a graticule projected onto the map axes.

The **General Surfaces** option activates the Surface Map Input dialog box for projecting a geolocated data grid onto the map axes.

The **Contours** option activates the Contour Map Input dialog box for projecting a two- or three-dimensional contour plot onto the map axes.

The **Quiver 2D** option activates the Quiver Map Input dialog box for projecting a two-dimensional quiver plot onto the map axes.

The **Quiver 3D** option activates the Quiver3 Map Input dialog box for projecting a three-dimensional quiver plot onto the map axes.

The **Stem** option activates the Stem Map Input dialog box for projecting a stem plot onto the map axes.

The **Scatter** option activates the Scatter Map Input dialog box for projecting a scatter plot onto the map axes.

The **Text** option activates the Text Map Input dialog box for projecting text objects onto the map axes.

## Display Menu

The **Projection** option activates the Projection Control dialog box for editing map projection properties and map display settings.

The **Graticule** option is used to view and edit the graticule size for surface maps.

The **Legend** option is used to display a contour map legend.

The **Frame** option is used to toggle the map frame on and off.

The **Grid** option is used to toggle the map grid on and off.

The **Meridian Labels** option is used to toggle the meridian grid labels on and off.

The **Parallel Labels** option is used to toggle the parallel grid labels on and off.

The **Tracks** option activates the Define Tracks input box for calculating and displaying Great Circle and Rhumb Line tracks on the map axes.

The **Small Circles** option activates the Define Small Circles input box for calculating and displaying small circles on the map axes.

The **Surface Distances** option activates the Surface Distance dialog box for distance, azimuth, and reckoning calculations.

## Tools Menu

The **Hide** option is used to hide the mouse tool buttons.

The **Off** option is used to turn off the current mouse tool.

The **Set Limits** option is used to define the zoom out limits to the current settings on the axes.

The **Full View** option is used to zoom out to the current axes limit settings.

The **Origin** option is used to toggle Origin (`originui`) mode on and off. Origin mode is used to interactively modify the map origin.

The **2D View** option is used to set the default two-dimensional view (azimuth=0, elevation=90).

The **Objects** option activates the Object Sets dialog box, which allows for property manipulation of objects displayed on the map axes.

The **Edit** option activates the MATLAB Property Editor to manipulate properties of a plotted object. Choose the **Current Object** option to edit the currently selected object or choose the **Select Object** option to open the Select Object dialog box and choose the object you want to edit.

The **Show** option is used to set the `Visible` property of mapped objects to `'on'`. The **All** option shows all currently mapped objects. The **Object** option activates the Select Object dialog box.

The **Hide** option is used to set the `Visible` property of mapped objects to `'off'`. Choose from the **All** or **Map** options, or choose the **Object** option to activate the Select Object dialog box.

The **Delete** option is used to clear the selected objects. The **All** option clears the current map, frame, and grid lines. The map definition is left in the axes definition. The **Map** option clears the current map, deleting objects plotted on the map but leaving the frame and grid lines displayed. The **Object** option activates the Select Object dialog box.

The **Axes** option is used to manipulate the MATLAB Cartesian axes. The **Show** option shows this axes, the **Hide** option hides this axes, and the **Color** option allows for custom color selection for this axes.

## Colormaps Menu

The **Colormaps** menu allows for manipulation of the colormap for the current figure. See the `clrmenu` reference page for details on the **Colormaps** menu options.

## See Also
axesm

# maptrim

Interactively trim and convert map data from vector to raster format

## Syntax

```
maptrim(lat,lon,linespec)
maptrim(datagrid,refvec)
maptrim(datagrid,refvec,PropertyName,PropertyValue)
```

## Description

`maptrim(lat,lon)` displays the supplied map data in a new figure window and allows a region of the map to be selected and saved in the workspace. `lat` and `lon` must be vector map data. The output can be line, patch, or regular surface (matrix) data. If patch map output is selected, the inputs `lat` and `lon` must originally be patch map data.

`maptrim(lat,lon,linespec)` displays the supplied map data using the `linespec` string.

`maptrim(datagrid,refvec)` displays data grid data in a new figure window and allows a subset of this map to be selected and saved. The output is regular surface data.

`maptrim(datagrid,refvec,PropertyName,PropertyValue)` displays the data grid using the surface properties provided. The object `Tag`, `EdgeColor`, and `UserData` properties cannot be set.

## Controls
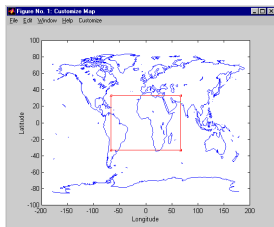
The `maptrim` tool displays the supplied map data in a new figure window and activates a **Customize** menu for that figure. The **Customize** menu has three menu options: **Zoom On/Off**, **Limits**, and **Save As**.

The **Zoom On/Off** menu toggles interactive zoom on and off.

The **Limits** menu option activates the Enter Map Limits dialog box, which is used to enter the latitude and longitude limits of the desired map subset. These entries are two-element vectors, enclosed in brackets. Pressing the **OK** button zooms in to the new limits. Pressing the **Cancel** button disregards the new limits and returns to the map display.
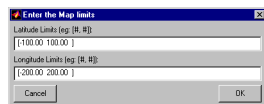


The **Save As** menu option is used to specify the variable names in which to save the map data subset. To save line and patch data, enter the new latitude and longitude variable names, along with the map resolution. For surface data, enter the new map and referencing vector variable names, along with the scale of the map. Latitude and longitude limits are optional.

## See Also
maptriml | maptrimp | maptrims

# mlayers

GUI to control plotting of display structure elements

## Compatibility

mlayers will be removed in a future release.

## Activation

| Command Line | Maptool |
|---|---|
| mlayers('filename')<br><br>mlayers('filename',h)<br><br>mlayers(cellarray)<br><br>mlayers(cellarray,h) | **Session > Layers** |

## Description

mlayers('filename') associates all display structures, which in this context are also called map layers, in the MAT-file filename with the current map axes. The display structure variables are accessible only through the mlayers tool, and not through the base workspace. filename must be a string.

mlayers('filename',h) assigns the layers found in filename to the map axes indicated by the handle h.

mlayers(cellarray) associates the layers specified by cellarray with the current map axes. cellarray must be of size n by 2. Each row of cellarray represents a map layer. The first column of cellarray contains the layer structure, and the second column contains the name of the layer structure. Such a cell array can be generated from data in the current workspace with the function rootlayr. In this case, the calling sequence would be rootlayr; mlayers(ans).

`mlayers(cellarray,h)` assigns the layers specified by `cellarray` to the map axes specified by the handle `h`.

# Controls
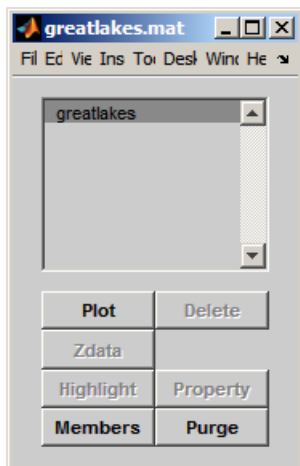


The scrollable list box displays all of the map layers currently associated with the map axes. An asterisk next to the layer name indicates that the layer is currently visible. An `h` next to the layer name indicates a layer that is plotted, but currently hidden.

The **Plot** button plots the selected map layer. Once the selected layer is plotted, the button toggles between **Hide** and **Show**, to turn the `Visible` property of the plotted objects to `'off'` and `'on'`, respectively.

The **Zdata** button activates the Specify Zdata dialog box, which is used to enter the workspace variable containing the `ZData` for the selected map layer. Pressing the **List** button produces a list of all current workspace variables, from which the `ZData` variable can be selected. This entry can also be a scalar.



The **Highlight** button is used to toggle the selected map layer between highlighted and normal display.

The **Members** button brings up a list of members of the selected map layer. Members of a layer are defined by their `Tag` property.

The **Delete** button deletes the selected map layer from the map.

The **Property** button activates the Define Layer Properties dialog box, which is used to specify or change properties of all objects in the selected map layer. String entries must be enclosed in single quotes.



The **Purge** button deletes the selected map layer from the `mlayers` tool. Selecting **Yes** from the Confirm Purge dialog box deletes the map layer from both the `mlayers` tool and the map display. Selecting **Data Only** from the Confirm Purge dialog box deletes the map layer from the `mlayers` tool, while retaining the plotted object on the map display.

## See Also
`mobjects` | `rootlayr`

# mobjects

Manipulate object sets displayed on map axes

## Compatibility

`mobjects` will be removed in a future release.

## Activation

| Command Line | Maptool |
|---|---|
| `mobjects`<br><br>`mobjects(h)` | **Tools** > **Objects** |

## Description

An object set is defined as all objects with identical tags. If no tags are supplied, object sets are defined by object type.

`mobjects` allows manipulation of the object sets on the current map axes.

`mobjects(h)` allows manipulation of the objects set on the map axes specified by the handle `h`.

# Controls



The scrollable list box displays all of the object sets associated with the map axes. An asterisk next to an object set name indicates that the object set is currently visible. An h next to an object set name indicates an object set that is plotted, but currently hidden. The order shown in the list indicates the stacking order of objects within the same plane.

The **Hide/Show** button toggles the Visible property of the selected object set to 'off' and 'on', respectively, depending on the current Visible status.

The **Zdata** button activates the Specify Zdata dialog box, which is used to enter the workspace variable containing the ZData. The ZData property is used to specify the plane in which the selected object set is drawn. Pressing the **List** button produces a list of all current workspace variables, from which the ZData variable can be selected. Alternatively, a scalar value can be entered instead of a variable.



The **Highlight** button highlights all objects belonging to the selected object set.

The **Tag** button brings up an Edit Tag dialog box, which allows the tag of all members of the selected object set to be modified.

The **Delete** button clears all objects belonging to the selected object set from the map. The cleared object set remains associated with the map axes.



The **Property** button activates the Define Object Properties dialog box, which is used to specify additional properties of all objects in the selected object set. String entries must be enclosed in single quotes.



The **Update** button updates the list box display with current objects sets.

The **Stacking Order** buttons are used to modify the drawing order of the selected object set in relation to other plotted object sets in the same plane. Objects drawn first appear at the bottom of the stack, and objects drawn last appear at the top of the stack. The **Top** button places the selected object set above all other object sets in its plane. The **Up** and **Dwn** buttons move the selected object set up and down one place in the stacking order, respectively. The **Btm** button places the selected object set below all other object sets in its plane. Note that the ZData property overrides stacking order, i.e., if an object is at the top of the stacking order for its plane, it can still be covered by an object drawn in a higher plane.

## See Also
mlayers

# originui

Interactively modify map origin

## Activation

| Command Line | Maptool |
|---|---|
| originui<br><br>originui on<br><br>originui off | **Tools** > **Origin (menu)** > **Origin**(button) |

## Description

originui provides a tool to modify the origin of a displayed map projection. A marker (dot) is displayed where the origin is currently located. This dot can be moved and the map reprojected with the identified point as the new origin.

originui automatically toggles the current axes into a mode where only actions recognized by originui are executed. Upon exit of this mode, all prior ButtonDown functions are restored to the current axes and its children.

originui on activates origin tool. originui off e-activates the tool. originui will toggle between these two states.

## Controls

### Keystrokes

originui recognizes the following keystrokes. **Enter** (or **Return**) will reproject the map with the identified origin and remain in the originui mode. **Delete** and **Escape** will exit the origin mode (same as originui off). **N**,**S**,**E**,**W** keys move the marker North, South, East or West by 10.0 degrees for each keystroke. **n**,**s**,**e**,**w** keys move the marker in the respective directions by 1 degree per keystroke.

## Mouse Actions

originui recognizes the following mouse actions when the cursor is on the origin marker.

- Single-click and hold moves the origin marker. Double-click the marker reprojects the map with the specified map origin and remains in the origin mode (same as originui **Return**).
- Extended-click moves the marker along the Cartesian X or Y direction only (depending on the direction of greatest movement).
- Alternate-click exits the origin tool (same as originui off).

Macintosh Key Mapping

- Extend-click: **Shift**+click mouse button
- Alternate-click: **Option**+click mouse button

Microsoft Windows Key Mapping

- Extend-click: **Shift**+click left button or both buttons
- Alternate-click: **Ctrl**+click left button or right button

X-Windows Key Mapping

- Extend-click: **Shift**+click left button or middle button
- Alternate-click: **Ctrl**+click left button or right button

## See Also

axesm | setm

# panzoom

Zoom settings on 2-D map

## Compatibility

`panzoom` will be removed in a future release. Use `zoom` instead.

## Syntax

```
panzoom
panzoom on
panzoom off
panzoom setlimits
panzoom out
panzoom fullview
```

## Description

`panzoom` toggles the pan and zoom tool on and off.

`panzoom on` is equivalent to `zoom on`.

`panzoom off` is equivalent to `zoom off`.

`panzoom setlimits` is equivalent to `zoom reset`.

`panzoom out` is equivalent to `zoom out`.

`panzoom fullview` sets the axes limit modes to `'auto'` and resets `zoom` to the resulting limits.

### See Also
`zoom`

# parallelui

Interactively modify map parallels

## Activation

| Command Line | Maptool |
|---|---|
| `parallelui`<br><br>`parallelui on`<br><br>`parallelui off` | **Tools** > **Parallels (menu)** |

## Description

`parallelui` toggles the parallel tool on and off.

`parallelui on` activates the parallel tool

`parallelui off` deactivates the parallel tool

The `parallelui` GUI provides a tool to modify the standard parallels of a displayed map projection. One or two red lines are displayed where the standard parallels are currently located. The parallel lines can be dragged to new locations, and the map reprojected with the locations of the parallel lines as the new standard parallels.

## Controls

Mouse Interaction

A single-click-and-drag moves the parallel lines. A double-click on one of the standard parallels reprojects the map using the new parallel locations.

## See Also

`axesm` | `setm`

# property editors

GUIs to edit properties of mapped objects

## Activation

| | |
|---|---|
| map display: | Alternate-click mapped object (for Click-and-Drag Property Editor) |
| | In plot edit mode, double-click mapped object (to obtain MATLAB Property Editor; click the **More Properties...** button to open the Property Inspector) |
| maptool: | **Tools** > **Edit Plot** menu item (for MATLAB Property Editor) |

## Description

Alternate (e.g., **Ctrl**+clicking a mapped object activates a property editor, which allows modification of some basic properties of the object through simple mouse clicks and drags. The objects supported by this editor are map axes, lines, text, patches, and surfaces, and the properties supported for each object type are shown below.

In plot edit mode, double-clicking a mapped object activates the MATLAB Property Editor for that object. From the Property Editor you can launch the Property Inspector, a GUI that lists the properties and values of the selected object and allows you to modify them.

## Controls

### Click-and-Drag Property Editor

The Click-and-Drag editor lists object properties and values. The object tag appears at the top of the editor. Property names and values that appear in blue are toggles. For example, clicking **Frame** in the axes editor toggles the value of the `Frame` property between `'on'` and `'off'`.

**axes**
Color
Map Projection    miller
**Grid**
**Frame**
**Parallel Labels**
**Meridian Labels**
**Hide   Delete**
**EXIT**

## Click-and-Drag Editor for a map axes

Property values that appear on the right side of the editor box are modified by clicking and dragging. For example, to change the MarkerColor property of a line object, click and hold the dot next to **MarkerColor**, and drag the cursor until the dot appears in the desired color.

**line**
Marker
MarkerColor
MarkerSize    ×
Color
LineWidth
LineStyle
**Hide   Delete**
**EXIT**

## Click-and-Drag Editor for a line object

The **Drag** control in the text editor is used to reposition the text string. In drag mode, use the mouse to move the text to a new location, and click to reposition the text. The **Edit** control in the text editor activates a **Text Edit** window, which is used to modify text.

**text**
Color         ab2
Alignment     left
FontName      ab2
FontSize      ab2
**Drag**
**Edit**
**Hide   Delete**
**EXIT**

## Click-and-Drag Editor for a text object

The **Marker** property name in the patch editor is used to toggle the marker on and off. The property value to the right of **Marker** can be modified by clicking and dragging until the desired marker symbol appears.

**patch**
MarkerColor
MarkerSize    ×
Marker
LineStyle
LineWidth
EdgeColor
FaceColor
**Hide   Delete**
**EXIT**

## Click-and-Drag Editor for a patch object

The **Graticule** control on the surface editor activates a Graticule Mesh dialog box, which is used to alter the size of the graticule.

To move the property editor around the figure window, hold down the **Shift** key while dragging the editor box. Alternate-clicking the background of the property editor closes the **Click-and-Drag** editing session.

Guide Property Editor

The MATLAB Property Inspector (the `inspect` function) allows you to view and modify property values for most properties of the selected object. Use it to expand and collapse the hierarchy of objects, showing an object's parents and children. A plus sign (+) before a property indicates that it can be expanded to show its components, for example the axes `AmbientLightColor` applied to the surface object displayed below. A minus sign (-) before an object indicates an object can be collapsed to hide its components. To activate the Object Browser, check the **Show Object Browser** check box. The **Property List** shows all the property names of the selected object and their current values. To activate the **Property List**, check the **Show Property List** check box. To change a property value, use the edit boxes above the Property List. Pressing the **Close** button closes the Guide Property Editor and applies the property modifications to the object.
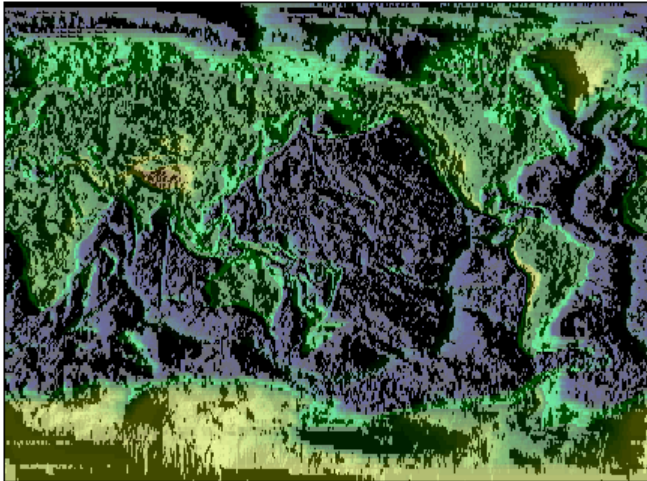


**A lit surface object in a map axes**
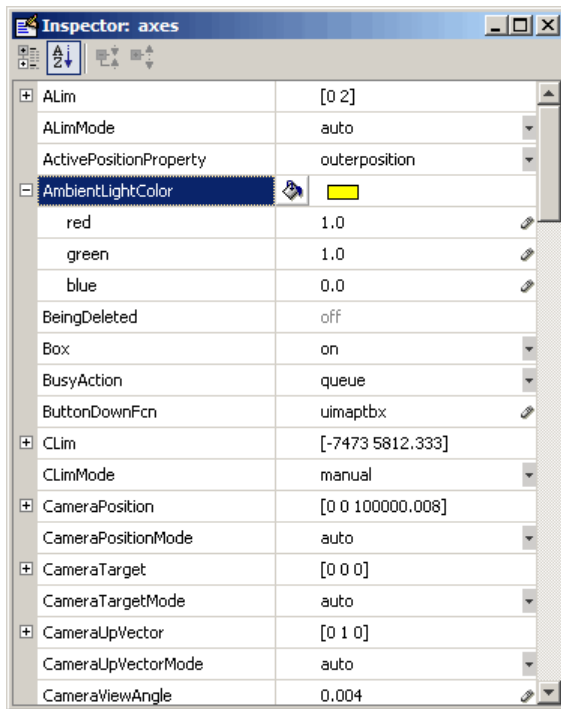
**Property Inspector view of axes object**

## See Also
propedit | uimaptbx | inspect

# qrydata

GUI to interactively perform data queries

## Compatibility

qrydata will be removed in a future release.

## Activation

| Command Line |
|---|
| qrydata(cellarray) |
| qrydata(*titlestr*,cellarray) |
| qrydata(h,cellarray) |
| qrydata(h,*titlestr*,cellarray) |
| qrydata(...,cellarray1,cellarray2,...) |

## Description

A data query is used to obtain the data corresponding to a particular (x,y) or (lat,lon) point on a standard or map axes.

qrydata(cellarray) activates a data query dialog box for interactive queries of the data set specified by cellarray (described below). qrydata can be used on a standard axes or a map axes. (x,y) or (lat,lon) coordinates are entered in the dialog box, and the data corresponding to these coordinates is then displayed.

qrydata(*titlestr*,cellarray) uses the string *titlestr* as the title of the query dialog box.

qrydata(h,cellarray) and qrydata(h,*titlestr*,cellarray) associate the data queries with the axes specified by the handle h, which in turn allows the input coordinates to be specified by clicking the axes.

The input `cellarray` is used to define the data set and the query. The first cell must contain the string used to label the data display line. The second cell must contain the type of query operation, either a predefined operation or a valid user-defined function name. This input must be a string. The predefined query operations are `'matrix'`, `'vector'`, `'mapmatrix'`, and `'mapvector'`.

The `'matrix'` query uses the MATLAB `interp2` function to find the value of the matrix Z at the input `(x,y)` point. The format of the `cellarray` input for this query is: `{'label','matrix',X,Y,Z,`*method*`}`. X and Y are matrices specifying the points at which the data Z is given. The rows and columns of X and Y must be monotonic. *method* is an optional argument that specifies the interpolation method. Possible *method* strings are `'nearest'`, `'linear'`, or `'cubic'`. The default is `'nearest'`.

The `'vector'` query uses the `interp2` function to find the value of the matrix Z at the input `(x,y)` point, then uses that value as an index to a data vector. The value of the data vector at that index is returned by the query. The format of `cellarray` for this type of query is: `{'label','vector',X,Y,Z, vector}`. X and Y are matrices specifying the points at which the data Z is given. The rows and columns of X and Y must be monotonic. `vector` is the data vector.

The `'mapmatrix'` query interpolates to find the value of the map at the input `(lat,lon)` point. The format of `cellarray` for this query is: `{'label','mapmatrix',datagrid,refvec,`*method*`}`. `datagrid` and `refvec` are the data grid and the corresponding referencing vector. *method* is an optional argument that specifies the interpolation method. Possible *method* strings are `'nearest'`, `'linear'`, or `'cubic'`. The default is `'nearest'`.

The `'mapvector'` query interpolates to find the value of the map at the input `(lat,lon)` point, then uses that value as an index to a data vector. The value of the vector at that index is returned by the query. The format of `cellarray` for this type of query is `{'label','mapvector',datagrid,refvec, vector}`. `datagrid` and `refvec` are the data grid and the corresponding referencing vector. `vector` is the data vector.

User-defined query operations allow for functional operations using the input `(x,y)` or `(lat,lon)` coordinates. The format of `cellarray` for this type of query is `{'label',`*function*`,other arguments...}` where the other arguments are the remaining elements of `cellarray` as in the four predefined operations above. *function* is a user-created function and must refer to a MATLAB function with the signature `z = fcn(x,y,other_arguments...)`.

qrydata(...,cellarray1,cellarray2,...) is used to input multiple cell arrays. This allows more than one data query to be performed on a given point.

## Controls
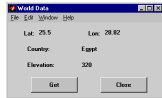


**Sample data query dialog box**

If an axes handle h is not provided, or if the axes specified by h is not a map axes, the currently selected point is labeled as **Xloc** and **Yloc** at the top of the query dialog box. If h is a map axes, the current point is labeled as **Lat** and **Lon**. Displayed below the current point are the results from the queries, each labeled as specified by the 'label' input arguments.

The **Get** button appears if an axes handle h is provided. Pressing this button activates a mouse cursor, which is used to select the desired point by clicking the axes. Once a point is selected, the queries are performed and the results are displayed.

The **Process** button appears if the handle h is not provided. In this case, the (x,y) coordinates of the desired point are entered into the edit boxes. Pressing the **Process** button performs the data queries and displays the results.

Pressing the **Close** button closes the query dialog box.

## Examples

This example illustrates use of a user-defined query to display city names for map points specified by a mouse click. The query is evaluated by a user-supplied file called qrytest.m, described below:

```
axesm miller
land = shaperead('landareas', 'UseGeoCoords', true);
geoshow(land, 'FaceColor', [0.5 0.7 0.5])
lakes = shaperead('worldlakes', 'UseGeoCoords', true);
geoshow(lakes, 'FaceColor', 'blue')
rivers = shaperead('worldrivers', 'UseGeoCoords', true);
geoshow(rivers, 'Color', 'blue')
```

```
cities = shaperead('worldcities', 'UseGeoCoords', true);
geoshow(cities, 'Marker', '.', 'Color', 'red')
tightmap
lat = [cities.Lat]';
lon = [cities.Lon]';
mat = char(cities.Name);
qrydata(gca,'City Data',{'City','qrytest',lat,lon,mat})
```

Create the file qrytest.m on your path, and in it put the following code:

```
function cityname = qrytest(lt, lg, lat, lon, mat)
% function QRYTEST returns city name for mouse click
% QRYTEST will find the closest city (min radius) from
% the mouse click, within an angle of 5 degrees.
%
latdiff = lt-lat;
londiff = lg-lon;
rad = sqrt(latdiff.^2+londiff.^2);
[minrad,index] = min(rad);
if minrad > 5
  index = [];
end
switch length(index)
  case 0, cityname = 'No city located near click';
  case 1, cityname = mat(index,:);
end
```
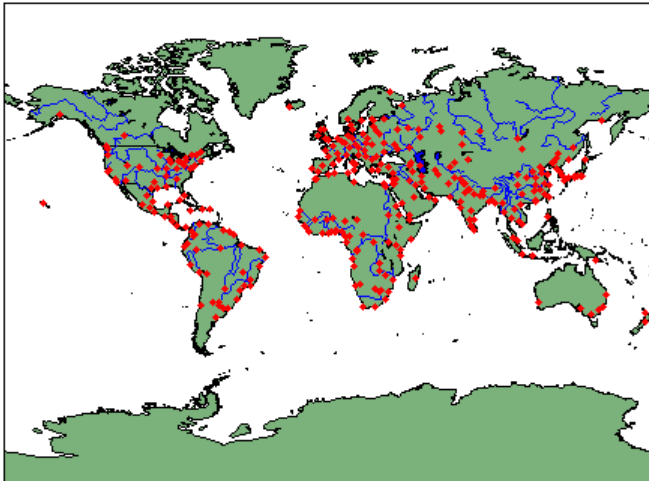
Clicking the mouse over a city marker displays the name of the selected city. Clicking the mouse in an area away from any city markers displays the string `'No city located near click'`.

## See Also
`interp2`

# scirclui

GUI to display small circles on map axes

---

**Note** scirclui is obsolete. Use scircleg instead.

---

## Activation

| Command Line | Maptool |
|---|---|
| scirclui | **Display mall Circles** |
| scirclui(h) | |

## Description

scirclui activates the Define Small Circles dialog box for adding small circles to the current map axes.

scirclui(h) activates the Define Small Circles dialog box for adding small circles to the map axes specified by the axes handle h.

## Controls
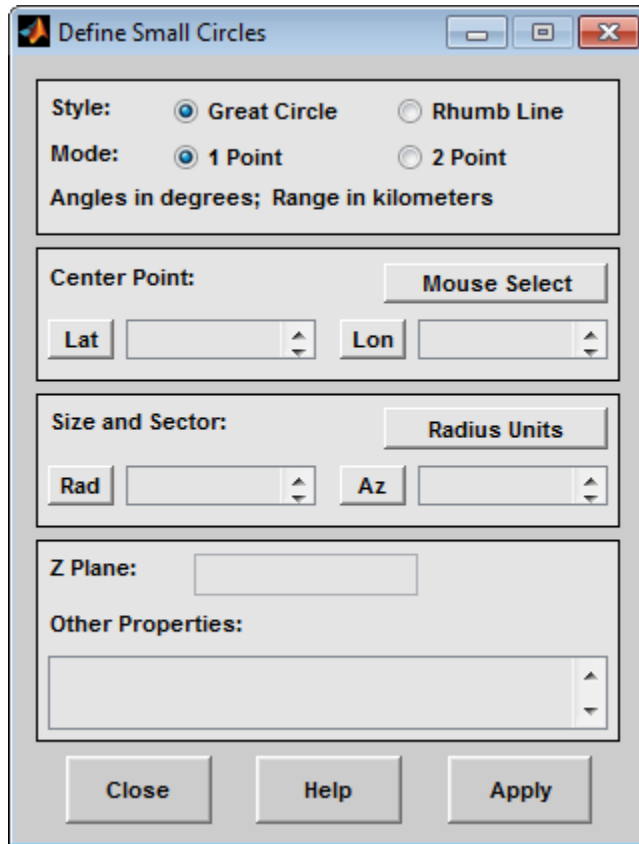


**Define Small Circles dialog box for one-point mode**

The **Style** selection buttons are used to specify whether the circle radius is a constant great circle distance or a constant rhumb line distance.

The **Mode** selection buttons are used to specify whether one point or two points are to be used in defining the small circle. If one-point mode is selected, a center point, radius, and azimuth are the required inputs. If two-point mode is selected, a center point, and perimeter point on the circle are the required inputs.

The **Center Point** controls are used in both one-point and two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the center point of the small circle to be displayed. These values must be in degrees. To display more than one small circle, a vector of values can be entered, enclosed in brackets in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for easier entry of long vectors. The **Mouse Select** button is used to select a center point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

The **Circle Point** controls are used only in two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of a point on the perimeter of the small circle to be displayed. These values must be in degrees. To display more than one small circle, a vector of values can be entered, enclosed in brackets in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for easier entry of long vectors. The **Mouse Select** button is used to select a perimeter point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

The **Size and Sector** controls are used only in one-point mode. The **Radius Units** button brings up a Define Range Units dialog box, which allows for modification of the small circle radius units and the normalizing geoid. The **Rad** edit box is used to enter the radius of the small circle in the proper units. The **Arc** edit box is used to specify the sector azimuth, measured in degrees, clockwise from due north. If the entry is omitted, a complete small circle is drawn. When entering radius and arc data for more than one small circle, vectors of values, enclosed in brackets, are entered in each edit box. Pushing the **Rad** or **Arc** button brings up an expanded edit box for that entry, which is useful for entering long vectors.

The **Z Plane** edit box is used to enter a scalar value that specifies the plane in which to display the small circles.

The **Other Properties** edit box is used to specify additional properties of the small circles to be projected, such as `'Color','b'`. String entries must be enclosed in quotes.

Pressing the **Apply** button accepts the input data and displays the small circles on the current map axes.

Pressing the **Cancel** button disregards any input data and closes the Define Small Circles dialog box.
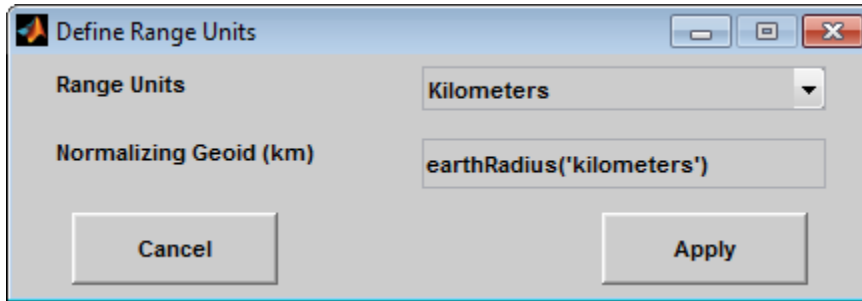
This dialog box, available only in one-point mode, allows for modification of the small circle radius units and the normalizing geoid.

The **Radius Units** pull-down menu is used to select the units of the small circle radius. The unit selected is displayed near the top of the Define Small Circles dialog box, and all latitude and longitude entries must be entered in these units. Users must also be sure to specify the normalizing geoid in the same units. If radians are selected, it is assumed the radius entry is a multiple of the radius used to display the current map, as defined by the map geoid property.

The **Normalizing Geoid** edit box is used modify the radius used to normalize the small circle radius to a radian value, which is necessary for proper calculations and map display. This entry must be in the same units as the small circle radius. If the small circle radius units are in radians, then the normalizing geoid must be the same as the geoid used for the current map axes.

Pressing the **Cancel** button disregards any modifications and closes the Define Radius Units dialog box.

Pressing the **Apply** button accepts any modifications and returns to the Define Small Circles dialog box.

## See Also
```
scircle1 | scircle2
```

# seedm

GUI to fill data grids with seeded values

# Compatibility

`seedm` will be removed in a future release.

# Activation

**Command Line**

```
seedm(datagrid,refvec)
```

# Description

Encoding is the process of filling in specific values in regions of a data grid up to specified boundaries, which are indicated by entries of 1 in the variable `map`. Encoding entire regions at one time allows indexed maps to be created quickly.

`seedm(datagrid,refvec)` displays the surface map in a new figure window and allows for seeds to be specified and the encoded map generated. The encoded map can then be saved to the workspace. `map` is the data grid and must consist of positive integer index values. `refvec` is the referencing vector of the surface.

# Controls
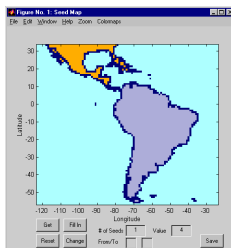
The **Zoom On/Off** menu toggles the zoom box on and off. The box can be moved by clicking the new location or by dragging the box to the new location. The box size can be increased or decreased by dragging a corner of the box. Pressing the **Return** key or double-clicking in the center of the box zooms in to the box limits.

The **Colormaps** menu provides a variety of colormap options that can be applied to the map. See clrmenu in this guide for a description of the **Colormaps** menu options.

The **Get** button allows mouse selection of points on the map to which seeds are assigned. The number of points to be selected is entered in the **# of Seeds** edit box. The value of the seed is entered in the **Value** edit box. This seed value is assigned to each point selected with the mouse. The **Get** button is pressed to begin mouse selection. After all the points have been selected, the **Fill In** button is pressed to perform the encoding operation. The region containing the seed point is filled in with the seed value. The **Reset** button is used to disregard all points selected with the mouse before the **Fill In** button is pressed.

Alternatively, specific map values can be globally replaced by using the **From/To** edit boxes. The value to be replaced is entered in the first edit box, and the new value is entered in the second edit box. Pressing the **Change** button replaces all instances of the **From** value to the **To** value in the map.

**Note** Values of 1 represent boundaries and should not be changed.

The **Save** button is used to save the encoded map to the workspace. A dialog box appears in which the map variable name is entered.

## See Also
colorm | encodem | getseeds | maptrim

# showm-ui

Show specified mapped objects

## Activation

| Command Line | Maptool |
|---|---|
| showm | **Tools** > **Show** > **Object** |

## Description

showm brings up a Select Object dialog box for selecting mapped objects to show (Visible property set to 'on').

## Controls



The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button changes the Visible property of the selected objects to 'on'. Pushing the **Cancel** button aborts the operation without changing any properties of the selected objects.

### See Also

showm

# surfdist

Interactive distance, azimuth, and reckoning calculations

## Activation

| Command Line | Maptool |
|---|---|
| surfdist<br><br>surfdist(h)<br><br>surfdist([]) | **Display** > **Surface** > **Distances** |

## Description

surfdist activates the Surface Distance dialog box for the current axes only if the axes has a proper map definition. Otherwise, the Surface Distance dialog box is activated, but is not associated with any axes.

surfdist(h) activates the Surface Distance dialog box for the axes specified by the handle h. The axes must be a map axes.

surfdist([]) activates the Surface Distance dialog box and does not associate it with any axes, regardless of whether the current axes has a valid map definition.

# Controls
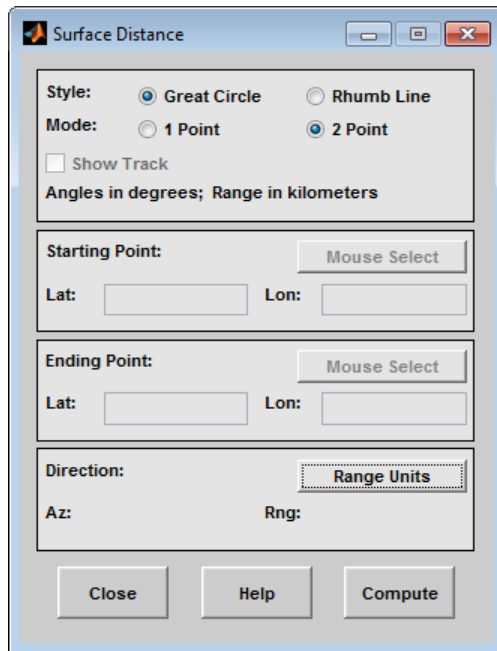


The **Style** selection buttons are used to specify whether a great circle or rhumb line is used to calculate the surface distance. When all other entries are provided, selecting a style updates the surface distance calculation.

The **Mode** selection buttons are used to specify whether one point or two points are to be used in defining the track distance. If one-point mode is selected, a starting point, azimuth, and range are the required inputs, and the ending point is computed. If two-point mode is selected, starting and ending points of the track are required, and the azimuth and distance along this track are then computed.

The **Show Track** check box is used to indicate whether the track is shown on the associated map display. The track is deleted when the Surface Distance dialog box is closed, or when the **Show Track** check box is unchecked and the surface distance calculations are recomputed.

The **Starting Point** controls are used for both one-point and two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the starting point of

the track. These values must be in degrees. Only one starting point can be entered. The **Mouse Select** button is used to select a starting point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

The **Ending Point** controls are enabled only for two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the ending point of the track. These values must be in degrees. Only one ending point can be entered. The **Mouse Select** button is used to select an ending point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection. During one-point mode, the Ending Point controls are disabled, but the ending point that results from the surface distance calculation is displayed.

The **Direction** controls are enabled only for one-point mode. The **Range Units** button brings up a Define Range Units dialog box which allows for modification of the range units and the normalizing geoid. The **Az** edit box is used to enter the azimuth, which sets the initial direction of the track from the starting point. Azimuth is measured in degrees clockwise from due north. The **Rng** edit box is used to specify the reckoning range of the track, in the proper units. The azimuth and reckoning range, along with the starting point, are used to compute the ending point of the track in one-point mode. During two-point mode, the **Direction** controls are disabled, but the azimuth and range values resulting from the surface distance calculation are displayed.

Pressing the **Close** button disregards any input data, deletes any surface distance tracks that have been plotted, and closes the Surface Distance dialog box.

Pressing the **Compute** button accepts the input data and computes the specified distances.

## Define Range Units Dialog Box

This dialog box, available only for one-point mode, allows for modification of the range units and the normalizing geoid.
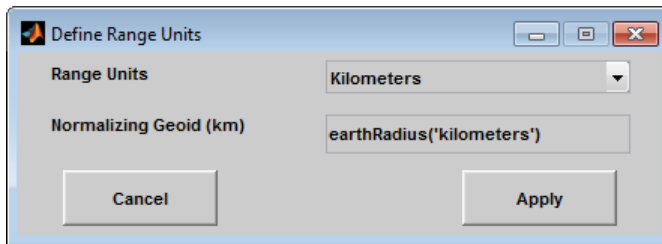
The **Range Units** pull-down menu is used to select the units of the reckoning range. The unit selected is displayed near the top of the Surface Distance dialog box, and all latitude and longitude entries must be entered in these units. Users must also be sure to specify the normalizing geoid in the same units. If radians are selected, it is assumed the range entry is a multiple of the radius of the normalizing geoid. In this case, the normalizing geoid must be the same as the geoid used to display the current map.

The **Normalizing Geoid** edit box is used modify the radius used to normalize range entries to radian values, which is necessary for proper calculations and map display. This entry must be in the same units as the range units. If the range units are in radians, then the normalizing geoid must be the same as the geoid used for the current map axes.

Pressing the **Cancel** button disregards any modifications and closes the Define Range Units dialog box.

Pressing the **Apply** button accepts any modifications and returns to the Surface Distance dialog box.

# tagm-ui

GUI to edit tag property of mapped object

## Activation

| Command Line |
| --- |
| `tagm` |
| `tagm(h)` |

## Description

`tagm` brings up a Select Object dialog box for selecting mapped objects and changing their `Tag` property. Upon selecting the objects, the Edit Tag dialog box is activated, in which the new tag is entered.

`tagm(h)` activates the Edit Tag dialog box for the objects specified by the handle `h`.

## Controls



**Select Object Dialog Box**

The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the Ok button activates the Edit Tag dialog box. Pushing the **Canel** button aborts the operation without changing any properties of the selected objects.

### Edit Tag Dialog Box

The new tag string is entered in the edit box. Pressing the **Apply** button changes the `Tag` property of all selected objected to the new tag string. Pressing the **Cancel** button closes the Edit Tag dialog box without changing the `Tag` property of the selected objects.

## See Also

tagm

# trackui

GUI to display great circles and rhumb lines on map axes

---

**Note** `trackui` is obsolete. Use `trackg` instead.

---

## Activation

| Command Line | Maptool |
|---|---|
| `trackui`<br><br>`trackui(h)` | **Display** > **Tracks** |

## Description

`trackui` activates the Define Tracks dialog box for adding great circle or rhumb line tracks to the current map axes.

`trackui(h)` activates the Define Tracks dialog box for adding great circle or rhumb line tracks to the map axes specified by the axes handle `h`.

# Controls
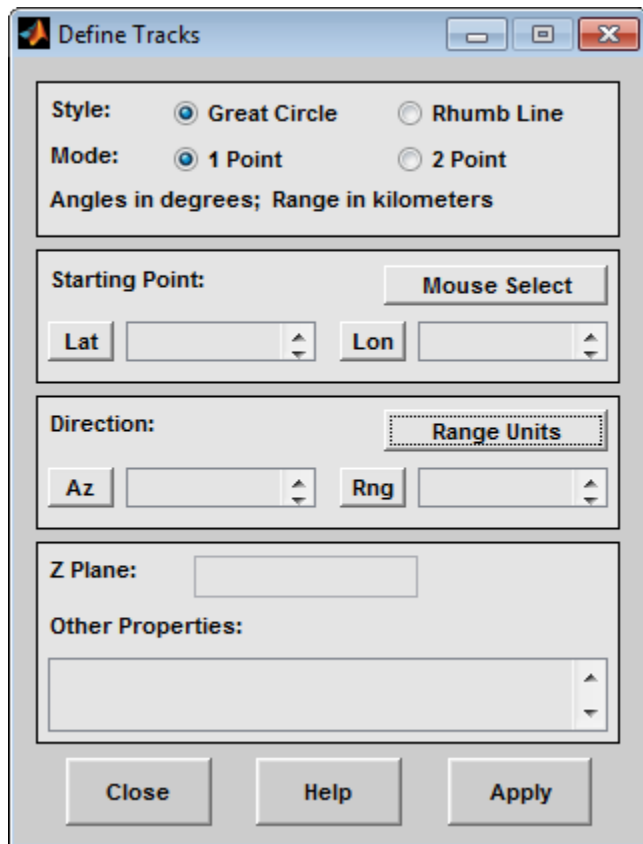


**Define Tracks dialog box for two-point mode**

The **Style** selection buttons are used to specify whether a great circle or rhumb line track is displayed.

The **Mode** selection buttons are used to specify whether one point or two points are to be used in defining the track. If one-point mode is selected, a starting point, azimuth, and range are the required inputs. If two-point mode is selected, starting and ending points are required.

The **Starting Point** controls are used for both one-point and two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the starting point of the track to be displayed. These values must be in degrees. To display more than one track, a vector of values can be entered, enclosed in brackets in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for easier entry of long vectors. The **Mouse Select** button is used to select a starting point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

The **Ending Point** controls are used only for two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the ending point of the track to be displayed. These values must be in degrees. To display more than one track, a vector of values can be entered, enclosed in brackets, in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for easier entry of long vectors. The **Mouse Select** button is used to select an ending point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.
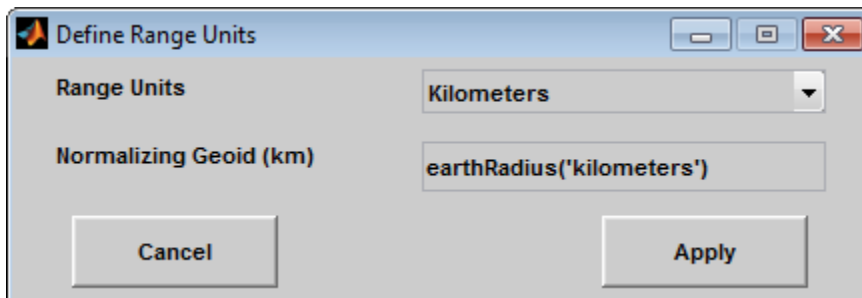
The **Direction** controls are used only for one-point mode. The **Range Units** button brings up a Define Range Units dialog box, which allows for modification of the range units and the normalizing geoid. The **Az** edit box is used to enter the azimuth, which sets the initial direction of the track from the starting point. Azimuth is measured in degrees clockwise from due north. The **Rng** edit box is used to specify the range of the track, in the proper units. If the range entry is omitted, a complete track is drawn. When inputting azimuth and range data for more than one track, vectors of values, enclosed in brackets, are entered in each edit box. Pushing the **Az** or **Rng** button brings up an expanded edit box for that entry, which is useful for entering long vectors.

The **Z Plane** edit box is used to enter a scalar value that specifies the plane in which to display the tracks.

The **Other Properties** edit box is used to specify additional properties of the tracks to be projected, such as `'Color','b'`. String entries must be enclosed in quotes.

Pressing the **Apply** button accepts the input data and displays the tracks on the current map axes.

Pressing the **Cancel** button disregards any input data and closes the Define Tracks dialog box.

**Define Range Units Dialog Box**

This dialog box, available only for one-point mode, allows for modification of the range units and the normalizing geoid.

The **Range Units** pull-down menu is used to select the units of the track range. The unit selected is displayed near the top of the Define Tracks dialog box, and all latitude and longitude entries must be entered in these units. Users must also be sure to specify the normalizing geoid in the same units. If radians are selected, it is assumed the range entry is a multiple of the radius used to display the current map.

The **Normalizing Geoid** edit box is used to modify the radius used to normalize range entries to radian values, which is necessary for proper calculations and map display. This entry must be in the same units as the range units. If the range units are in radians, then the normalizing geoid must be the same as the geoid used for the current map axes.

Pressing the **Cancel** button disregards any modifications and closes the Define Range Units dialog box.

Pressing the **Apply** button accepts any modifications and returns to the Define Tracks dialog box.

## See Also
track1 | track2

# uimaptbx

Handle buttondown callbacks for mapped objects

## Activation

set the `ButtonDownFcn` property to `'uimaptbx'`

## Description

`uimaptbx` processes mouse events for mapped objects. `uimaptbx` can be assigned to an object by setting the `ButtonDownFcn` to `'uimaptbx'`. This is the default setting for all objects created with Mapping Toolbox functions.

If `uimaptbx` is assigned to an object, the following mouse events are recognized: A single-click and hold on an object displays the object tag. If no tag is assigned, the object type is displayed. A double-click on an object activates the MATLAB Property Editor. An extend-click on an object activates the Projection Control dialog box, which allows the map projection and display properties to be edited. An alternate-click on an object allows basic properties to be edited using simple mouse clicks and drags.

Definitions of extend-click and alternate-click on various platforms are as follows:

| | |
|---|---|
| For `MS-Windows`: | Extend-click – **Shift**+click left button or both buttons |
| | Alternate-click – **Ctrl**+click left button or right button |
| For `X-Windows`: | Extend-click – **Shift**+click left button or middle button |
| | Alternate-click – **Ctrl**+ click left button or right button |

## See Also
axesm | axesmui | property editors

# utmzoneui

Choose or identify UTM zone by clicking map

## Activation

| Command Line |
|---|
| utmzoneui |
| utmzoneui(InitZone) |

## Description

zone = utmzoneui will create an interface for choosing a UTM zone on a world display map. It allows for clicking an area for its appropriate zone, or entering a valid zone to identify the zone on the map.

zone = utmzoneui(InitZone) will initialize the displayed zone to the zone string given in InitZone.

To interactively pick a UTM zone, activate the interface, and then click any rectangular zone on the world map to display its UTM zone. The selected zone is highlighted in red and its designation is displayed in the **Zone** edit field. Alternatively, type a valid UTM designation in the **Zone** edit field to select and see the location of a zone. Valid zone designations consist of an integer from 1 to 60 followed by a letter from C to X.

Typing only the numeric portion of a zone designation will highlight a column of cells. Clicking **Accept** returns a that UTM column designation. You cannot return a letter (row designation) in such a manner, however.

## Controls
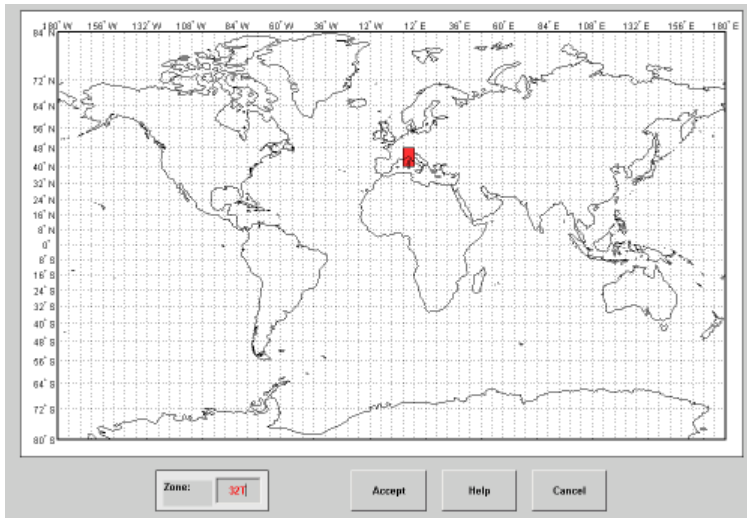


## More About

### Tips

The syntax of utmzoneui is similar to that of utmzone. If utmzone is called with no arguments, the utmzoneui interface is displayed for you to select a zone. Note that utmzone can return latitude-longitude coordinates of a specified zone, but that utmzoneui only returns zone names.

### See Also

utm | ups | utm | utmgeoid | utmzone

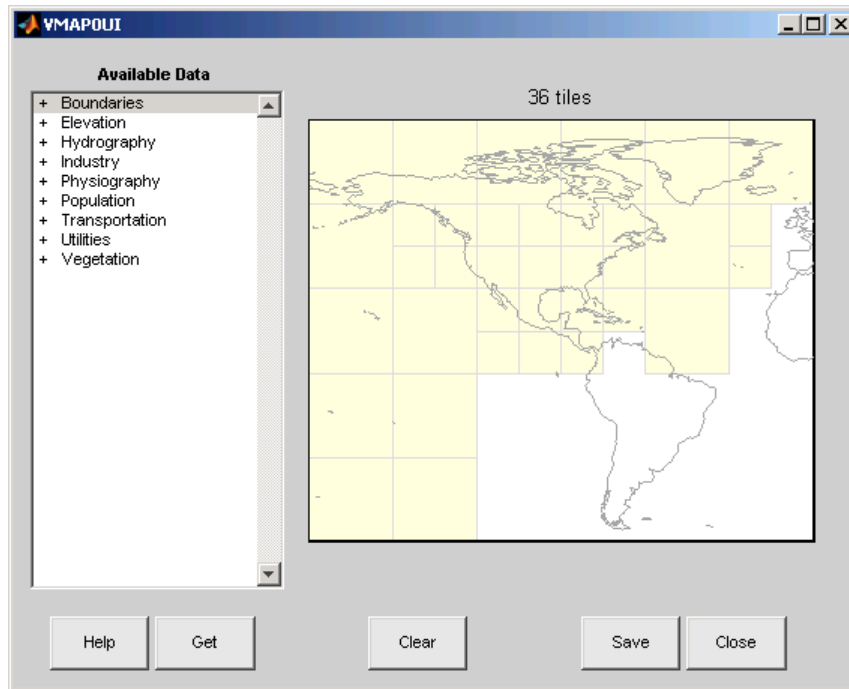# vmap0ui

UI for selecting data from Vector Map Level 0

## Description

vmap0ui(dirname) launches a graphical user interface for interactively selecting and importing data from a Vector Map Level 0 (VMAP0) data base. Use the string dirname to specify the folder containing the data base. For more on using vmap0ui, click the **Help** button after the interface appears.

vmap0ui(devicename) or vmap0ui devicename uses the logical device (volume) name specified in string devicename to locate CD-ROM drive containing the VMAP0 CD-ROM. Under the Windows operating system it could be 'F:', 'G:', or some other letter. Under Macintosh OS X it should be '/Volumes/VMAP'. Under other UNIX systems it could be '/cdrom/'.

vmap0ui can be used on Windows without any arguments. In this case it attempts to automatically detect a drive containing a VMAP0 CD-ROM. If vmap0ui fails to locate the CD-ROM device, then specify it explicitly.

# Controls



The vmapOui screen lets you read data from the Vector Map Level 0 (VMAP0). The VMAP0 is the most detailed world map database available to the public.

You use the list to select the type of data and the map to select the region of interest. When you click the **Get** button, data is extracted and displayed on the map. Use the **Save** button to save the data in a MAT-file or to the base workspace for later display. The **Close** button closes the window.

## The Map

The **Map** controls the geographic extent of the data to be extracted. vmapOui extracts data for areas currently visible on the map. Use the mouse to zoom in or out to the area of interest. Type help zoom for more on zooming.

The VMAP0 divides the world into tiles of about 5–by–5 degrees. When extracting, data is returned for all visible tiles, including those parts of the tile that are outside the

current view. The map shows the VMAP0 tiles in light yellow with light gray edges. The data density is high, so extracting data for a large number of tiles can take much time and memory. A count of the number of visible tiles is above the map.

## The List

The **List** controls the type of data to be extracted. The tree structure of the list reflects the structure of the VMAP0 database. Upon starting `vmap0ui`, the list shows the major categories of VMAP data, called themes. Themes are subdivided into features, which consist of data of common graphic types (patch, line, point, or text) or cultural types (airport, roads, railroads). Double-click a theme to see the associated features. Features can have properties and values, for example, a railroad tracks property, with values single or multiple. Double-click a feature to see the associated properties and values. Double-clicking an open theme or feature closes it. When a theme is selected, `vmap0ui` gets all the associated features. When a feature is selected, `vmap0ui` gets all of that feature's data. When properties and values are selected, `vmap0ui` gets the data for any of the properties and values that match (that is, the union operation).

## The Get Button

The **Get** button reads the currently selected VMAP0 data and displays it on the map. Use the **Cancel** button on the progress bar to interrupt the process. For a quicker response, press the standard interrupt key combination for your platform.

## The Clear Button

The **Clear** button removes any previously read data from the map.

## The Save Button

The **Save** button saves the currently displayed VMAP0 data to a MAT-file or the base workspace. If you choose to save to a file, you are prompted for a filename and location. If you choose to save to the base workspace, you are notified of the variable names that will be overwritten.

Data are returned as Mapping Toolbox display structures with variable names based on theme and feature names. You can update vector display structures to geographic data structures. For information about display structure format, see "Version 1 Display

Structures" on page 1-200 in the reference page for `displaym`. The `updategeostruct` function performs such conversions.

Use `load` and `displaym` to redisplay the data from a file on a map axes. You can also use the `mlayers` GUI to read and display the data from a file. To display the data in the base workspace, use `displaym`. To display all the display structures, use `rootlayr; displaym(ans)`. To display all of the display structures using the `mlayers` GUI, type `rootlayr; mlayers(ans)`.

### The Close Button

The **Close** button closes the `vmap0ui` panel.

## Examples

**1** Launch `vmap0ui` and automatically detect a CD-ROM on Microsoft Windows:

```
vmap0ui
```

**2** Launch `vmap0ui` on Macintosh OS X (need to specify volume name):

```
vmap0ui('Volumes/VMAP')
```

## See also

`displaym`, `extractm`, `mlayers`, `vmap0data`

# zdatam-ui

GUI to adjust *z*-plane of mapped objects

## Activation

| Command Line |
|---|
| zdatam |
| zdatam(h) |
| zdatam(*str*) |

## Description

zdatam brings up a Select Object dialog box for selecting mapped objects and adjusting their ZData property. Upon selecting the objects, the Specify Zdata dialog box is activated, in which the new ZData variable is entered. Note that not all mapped objects have the ZData property (for example text objects).

zdatam(h) activates the Specify Zdata dialog box for the objects specified by the handle h.

zdatam(*str*) activates the Specify Zdata dialog box for the objects identified by *str*, where *str* is any string recognized by handlem.

## Controls



**Select Object Dialog Box**

The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button activates another Specify Zdata dialog box. Pushing the **Cancel** button aborts the operation without changing any properties of the selected objects.



### Specify ZData Dialog Box

The **Zdata Variable** edit box is used to specify the name of the ZData variable. Pressing the **List** button produces a list of all current workspace variables, from which the ZData variable can be selected. A scalar value or a valid MATLAB expression can also be entered. Pressing the **Apply** button changes the ZData property of all selected objected to the new values. Pressing the **Cancel** button closes the Specify ZData dialog box without changing the ZData property of the selected objects.

## See Also
zdatam