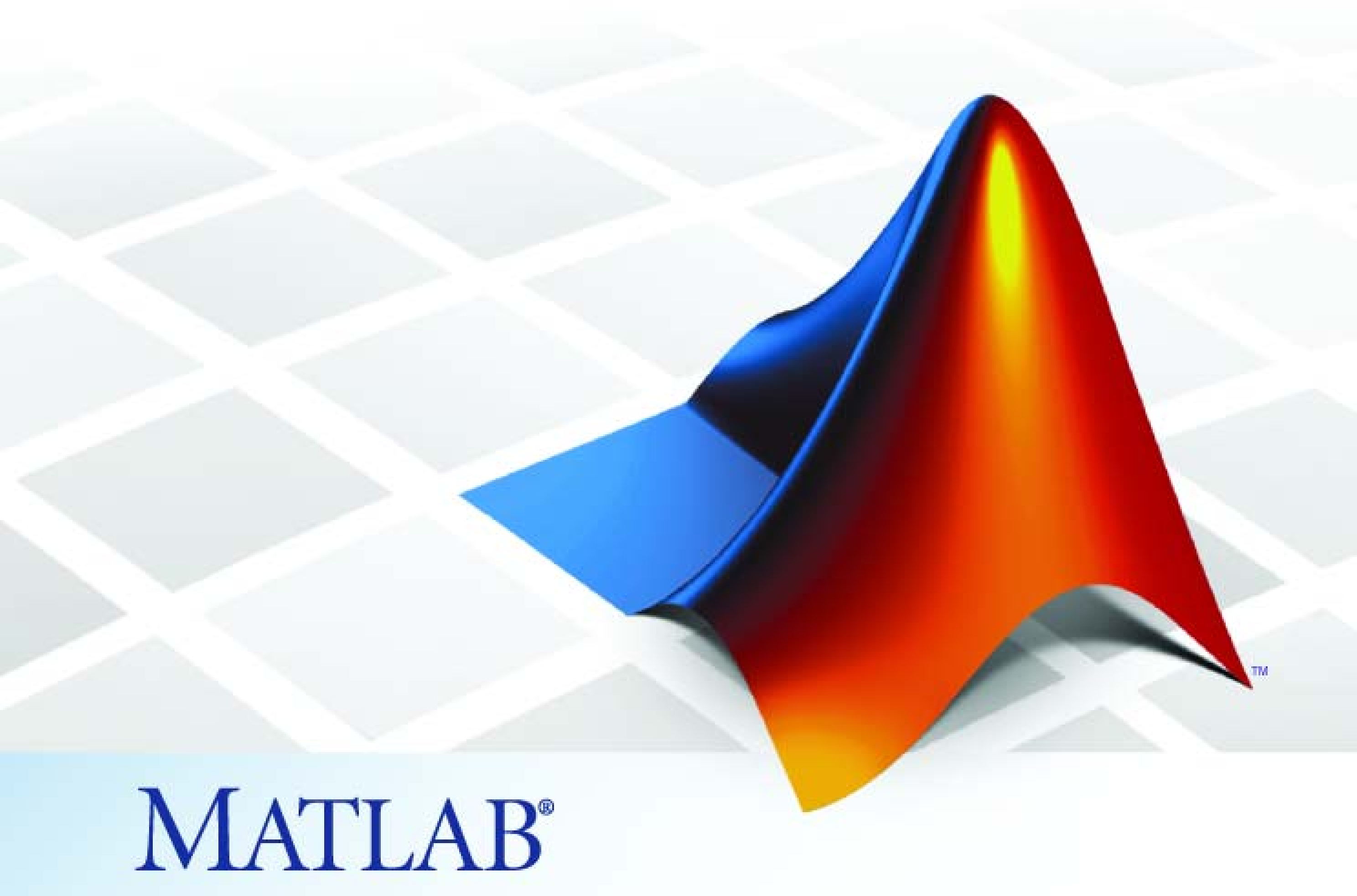
Mapping Toolbox™ 3
User's Guide



The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Mapping Toolbox™ User's Guide

© COPYRIGHT 1997–2009 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

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)

Contents

3 Understanding Geospatial Geometry

4 Creating and Viewing Maps

6 Customizing and Printing Maps

11 Function Reference

Getting Started

Note Some cross-references in this document refer to reference material included only in the electronic version of this user's guide. The complete user's guide is available in the MATLAB® Help browser, and in HTML and PDF formats on the MathWorks Web site, at `http://www.mathworks.com/access/helpdesk/help/toolbox/map/map.html`.

Product Overview

The Mapping Toolbox™ product comprises an extensive set of functions and graphical user interfaces (GUIs) for creating map displays and analyzing and manipulating geospatial data in the MATLAB environment. You can create maps that combine different types of data from multiple sources and display them in their correct spatial relationships. The toolbox supports spatial analysis methods such as line-of-sight calculations on terrain data and geographic computations that account for the curvature of the Earth's surface. Its library of map projections and georeferencing utilities give you precise control over projected and unprojected coordinate systems.

Most Mapping Toolbox functions are written in the open MATLAB language. This means that you can inspect the algorithms, adapt them to create your own custom functions, and automate frequently performed tasks. The toolbox also includes sample data sets, examples, and demos that illustrate key concepts, which provide starting points for geospatial data analysis projects of your own.

Briefly summarized, the toolbox provides functionality in the following areas:

- Import and export of file-based geospatial data
- Vector map data and geographic data structures
- Georeferenced images and data grids
- Web Map Service layer selection and map retrieval
- Map projections and coordinates
- Map display and interaction
- Geographic calculations for vector and raster data
- A map viewer and other graphical user interfaces

The sections that follow get you started using Mapping Toolbox capabilities, and describe what its documentation and demos contain and where to look for categories of information. For a complete classified list of Mapping Toolbox functions and features, see Chapter 11, "Function Reference".

Dedication and Acknowledgment

In memory of John P. Snyder (1926–97), whose meticulous studies and systematic descriptions of map projections inspired and enabled the creation of Mapping Toolbox software.

This software was originally developed and maintained through Version 1.3 by Systems Planning and Analysis, Inc. (SPA), of Alexandria, Virginia.

Except where noted, the information contained in demo and sample data files (found in `toolbox/map/mapdemos`) is derived from publicly available digital data sets. These data files are provided as a convenience to Mapping Toolboxusers. The MathWorks, Inc. makes no claims that any of this data is free of defects or errors, or that the representations of geographic features or names are up to date or authoritative.

Your First Maps

In this section...

This section helps you exercise high-level functions and GUIs to explore mapping and visualizing geodata. It explores `worldmap` and other functions, and then describes how to use the Map Viewer (`mapview`). Run the demos described in "Mapping Toolbox Demos and Data" on page 1-28 and search the index of examples to further acquaint yourself with Mapping Toolbox capabilities.

See the World

Spatial data is a general term that refers to data describing the location, shape, and spatial relationships of anything, from engineering drawings to maps of galaxies. *Geospatial data* is spatial data that is in some way *georeferenced*, or tied to specific locations on, under, or above the surface of a planet.

Geospatial data can be voluminous, complex, and difficult to work with. Mapping Toolbox functions handle many of the details of loading and displaying data for you, and has built-in data structures for representing geospatial data. Nevertheless, the more you understand about your data and the capabilities of the toolbox, the more interesting applications you will be able to pursue, and the more useful their results will be to you and others.

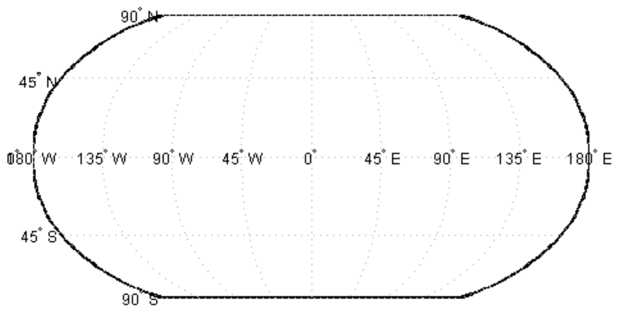
Getting started making world maps with the toolbox is easy.

1 In the MATLAB Command Window, type

   ```
   worldmap world
   ```

   This creates an empty map axes, ready to hold the data of your choice. Function `worldmap` automatically selected a reasonable choice for your map projection and coordinate limits. In this case, it chooses a Robinson projection centered on the prime meridian and the equator (0° latitude, 0° longitude).

Note that if you type `worldmap` without an argument a list box appears from which you can select a country, continent, or region. The `worldmap` function then generates a map axes with appropriate projection and map limits.



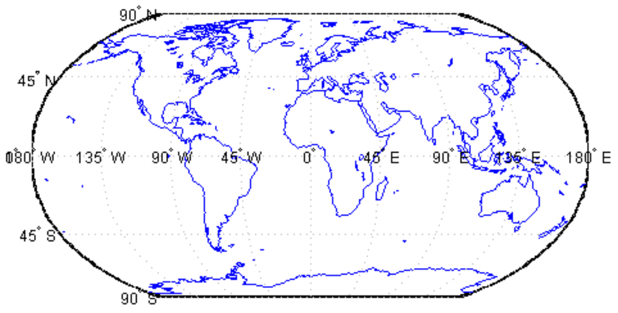
2 Import low-resolution world coastlines stored as simple MATLAB coordinate vectors in a MAT-file:

```
whos -file coast.mat
  Name          Size            Bytes  Class     Attributes

  lat       9865x1              78920  double
  long      9865x1              78920  double
```

3 Load and plot the coastlines on the world map:

```
load coast
plotm(lat, long)
```

The `plotm` function is a geographic equivalent to the MATLAB `plot` function. It accepts coordinates in latitude and longitude, which it transforms to *x* and *y* via a specified map projection (in this case specified by `worldmap`) before displaying them in a figure axes. Certain Mapping Toolbox functions that end with `m`, such as `plotm` and `textm`, are modeled after familiar MATLAB functions that handle nongeographic coordinate data.



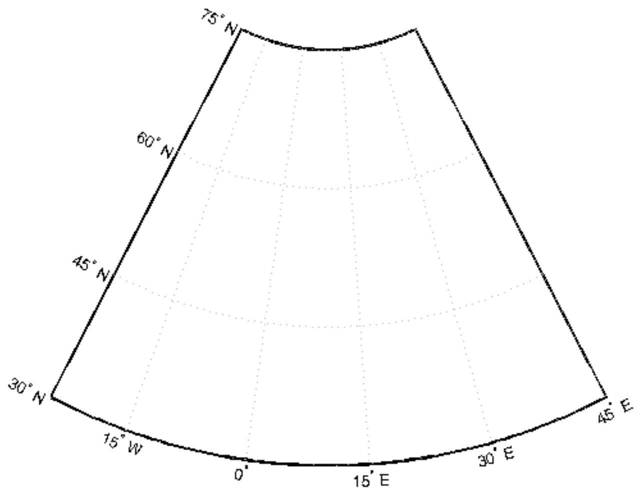
Notice how the world coastlines form distinct polygons, even though only a single vector of latitudes and a corresponding vector of longitudes are provided. The reason is because of NaN separators, which implicitly divide each vector into multiple parts.

```
[latcells, loncells] = polysplit(lat, long);
numel(latcells)

ans =
   241
```

`lat` and `long` include NaN terminators as well as separators, showing that the `coast` data set is organized into precisely 241 polygons.

4 Now create a new map axes for plotting data over Europe, and this time specify a return argument:

```
h = worldmap('Europe');
```



For the map of the world, `worldmap` chose a pseudocylindrical Robinson projection. For Europe, it chose an Equidistant Conic projection. How can you tell which projection `worldmap` is using?

When you specify a return argument for `worldmap` and certain other mapping functions, a handle (e.g., `h`) to the figure's axes is returned. The axes object on which map data is displayed is called a map axes. In addition to the graphics properties common to any MATLAB axes object, a map axes object contains additional properties covering map projection type, projection parameters, map limits, etc. The `getm` and `setm` functions and others allow you to define, access, and modify these properties.

5 To inspect the map axes properties for the map of Europe, first dereference the handle with the `getm` command (which is similar to the MATLAB `get` command, but returns map-specific data):

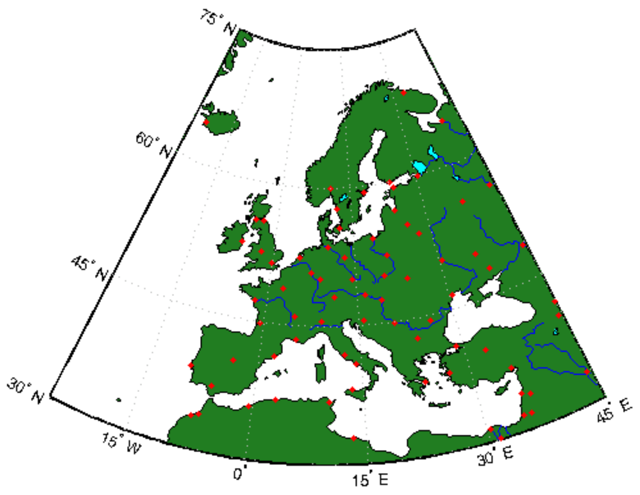
```
mstruct = getm(h);
```

6 Now you can inspect the 1-by-1 structure `mstruct` by listing it, using the property editor, or by accessing any field directly. For instance, to see the map projection selected for the map of Europe, type

```
mstruct.mapprojection
ans =
eqdconic
```

7 Add data to the map of Europe using the `geoshow` function and importing from several shapefiles in the `toolbox/map/mapdemos` directory:

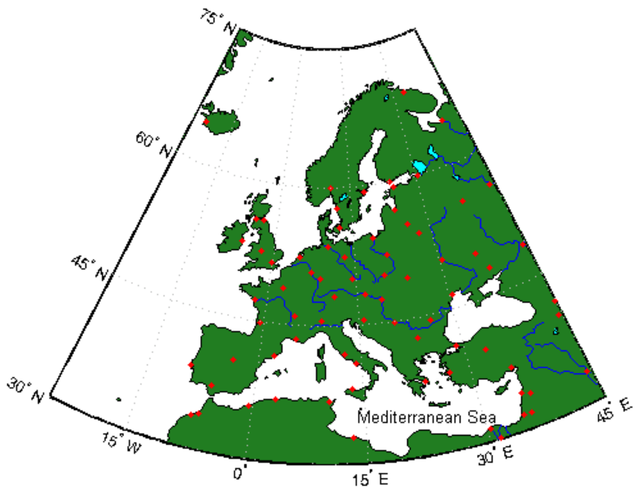
```
geoshow('landareas.shp', 'FaceColor', [0.15 0.5 0.15])
geoshow('worldlakes.shp', 'FaceColor', 'cyan')
geoshow('worldrivers.shp', 'Color', 'blue')
geoshow('worldcities.shp', 'Marker', '.',...
                           'MarkerEdgeColor', 'red')
```



Note how `geoshow` can plot data directly from files onto a map axes without first loading it into the MATLAB workspace.

8 Finally, place a label on the map to identify the Mediterranean Sea.

```
labelLat = 35;
labelLon = 14;
textm(labelLat, labelLon, 'Mediterranean Sea')
```



Look at the reference documentation for `worldmap` and experiment with its options. To learn more about display properties for map axes and how to control them, see "Accessing and Manipulating Map Axes Properties" on page 4-14. See the reference page for `geoshow` to find out more about its capabilities.

Tour Boston with the Map Viewer

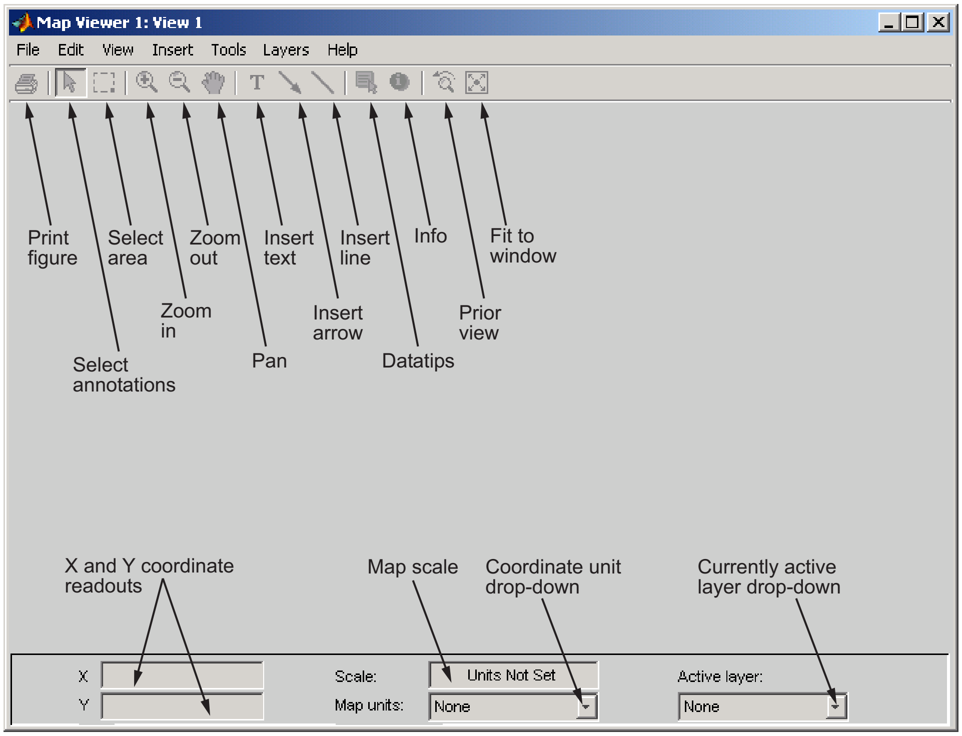
The Map Viewer is an interactive tool for browsing map data. With it you can assemble layers of vector and raster geodata and render them in 2-D. You can import, reorder, symbolize, hide, and delete data layers, identify coordinate locations, list data attributes, and display selected ones as *datatips* (signposts that identify attribute values, such as place names or route numbers). The following exercise shows how the Map Viewer works and what it can do.

A Map Viewer Session

1 You start a Map Viewer session by typing

```
mapview
```

at the MATLAB prompt. The Map Viewer opens with a blank canvas (no data is present). The viewer and its tools are shown below.



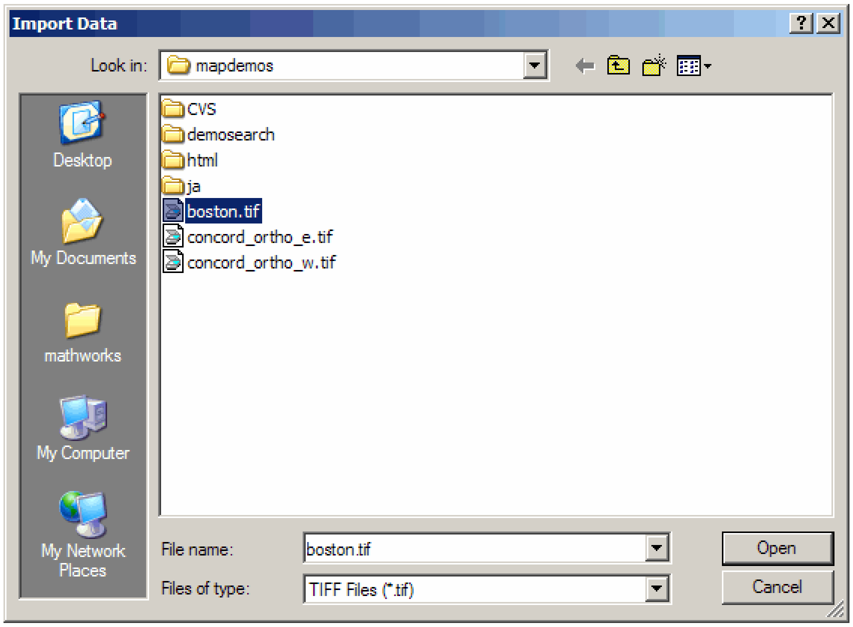
Most of the tool buttons can also be activated from the **Tools** menu.

2 For ease in importing Mapping Toolbox demo data, set your working directory as follows:

```
cd(fullfile(matlabroot,'toolbox','map','mapdemos'))
```

However, you can also navigate to this directory with the Map Viewer Import Data dialog if you prefer.

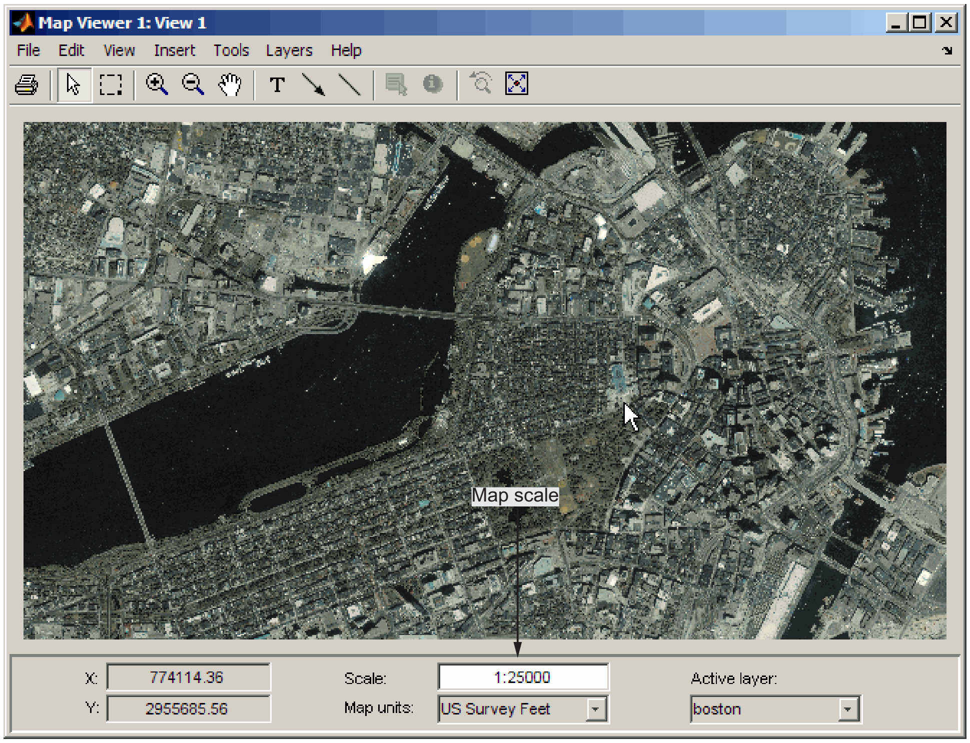
3 Select **Import From File** from the **File** menu and open the GeoTIFF file `boston.tif` in the Map Viewer, as shown below.



The file opens in the Map Viewer. The image is a visible red, green, and blue composite from a georeferenced IKONOS-2 panchromatic/multispectral product created by GeoEye™. Copyright © GeoEye, all rights reserved. For further information about the image, refer to the text files `boston.txt` and `boston_metdata.txt`.

4 To see the map scale, set the map distance units. Use the drop-down **Map units** menu at the bottom center to select `US Survey Feet`.

5 Now set the scale to 1:25,000 by typing `1:25000` in the **Scale** box, which is above the **Map units** drop-down. The viewer now looks like this.



X: 774114.36
Y: 2955685.56
Scale: 1:25000
Map units: US Survey Feet
Active layer: boston

Note that the cursor is pointing at the front of the Massachusetts State House (capitol building). The map coordinates for this location are shown in the readout at the lower left as 774,114.36 feet easting (**X**), 2,955,685.56 feet northing (**Y**), in Massachusetts State Plane coordinates.

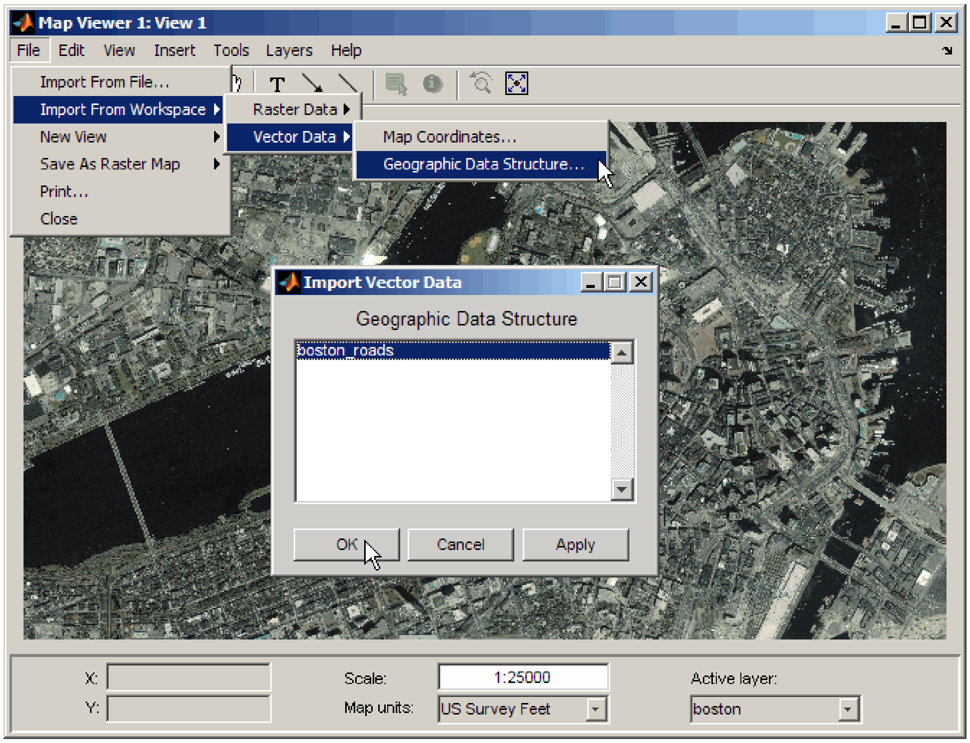
6 Next, import a vector data layer, the streets and highways in the central Boston area from the line shapefile `boston_roads.shp`. However, as is frequently the case when overlaying geodata, the coordinate system used by `boston_roads.shp` (which has units of meters) does not completely agree with the one for the satellite image, `boston.tif` (which uses units of feet). If you were to ignore this, the two data sets would be out of registration by quite a large distance.

Convert the units of `boston_roads.shp` to meters. First, read the file into the workspace as a *geographic data structure* using `shaperead`, and then convert its X and Y coordinate fields from U.S. survey feet to meters using the following code:

```
boston_roads = shaperead('boston_roads.shp');
surveyFeetPerMeter = unitsratio('survey feet','meter');
for k = 1:numel(boston_roads)
    boston_roads(k).X = surveyFeetPerMeter * boston_roads(k).X;
    boston_roads(k).Y = surveyFeetPerMeter * boston_roads(k).Y;
end
```

The `unitsratio` function computes conversion factors between a variety of units of length.

7 Because you want to map data that is already in the workspace, this time use **Import From Workspace > Vector Data > Geographic Data Structure** from the **File** menu; specify `boston_roads` as the data to import from the workspace, and click **OK**.



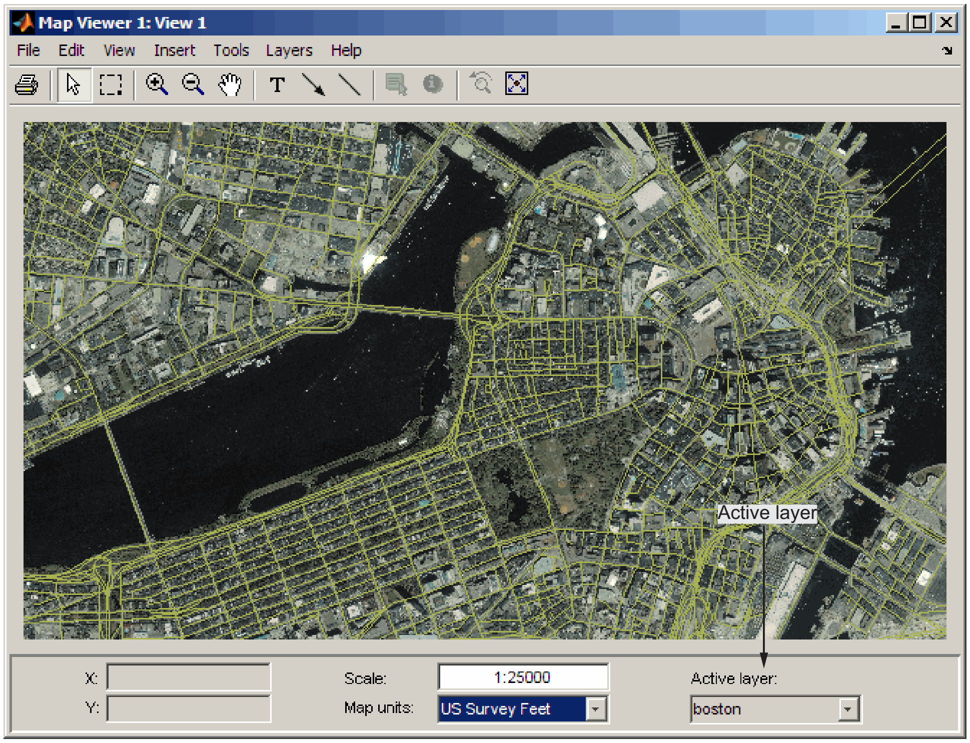
Geographic Data Structure

boston_roads

X:
Y:
Scale: 1:25000
Map units: US Survey Feet
Active layer: boston

You could clear the workspace now if you wanted, because all the data that `mapview` needs is now loaded into it.

8 After the Map Viewer finishes importing the roads layer, it selects a random color and renders all the shapes with that color as solid lines. The view looks like this.

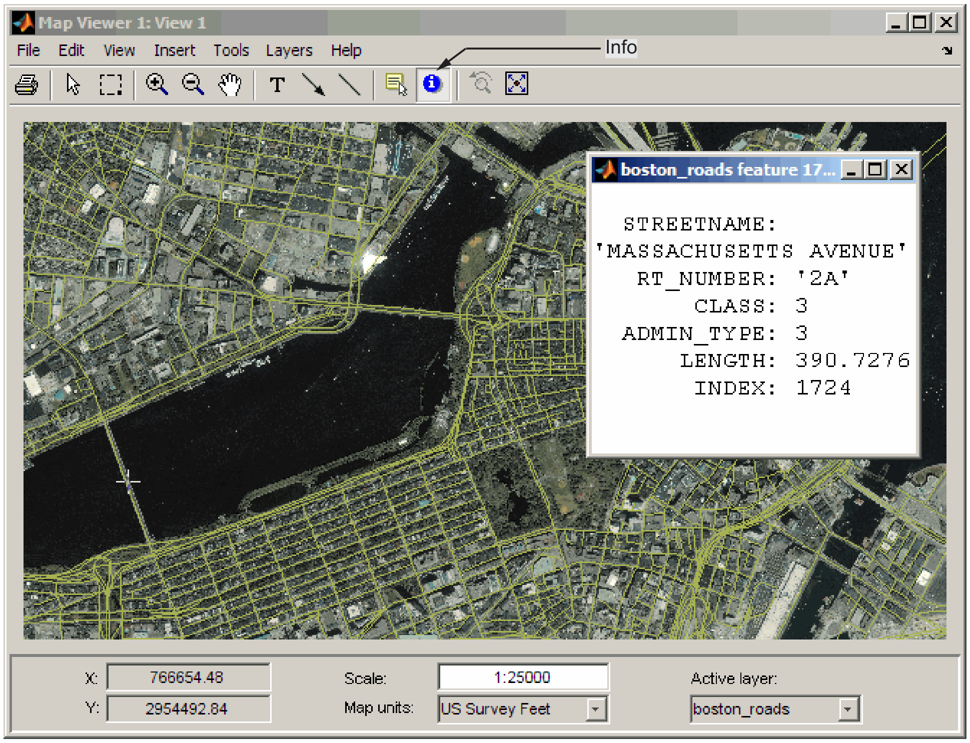


Being random, the color you see for the road layer can differ. How you can specify road colors is discussed below.

9 You can designate any layer to be the *active layer* (the one that you can query); it does not need to be the topmost layer. By default, no layer is active. Use the **Active layer** drop-down menu at the bottom left to select `boston_roads`.

Changing the active layer has no visual effect. Doing so allows you to query attributes of the layer you select.

10 One way to see the attributes for a vector layer is to use the **Info** tool, a button near the right end of the toolbar. Select the **Info** tool and click somewhere along the bridge across the Charles River near the lower left of the map. This opens a text window displaying the attribute/values for the selected object.


Map Viewer 1: View 1
File Edit View Insert Tools Layers Help
Info
boston_roads feature 17...

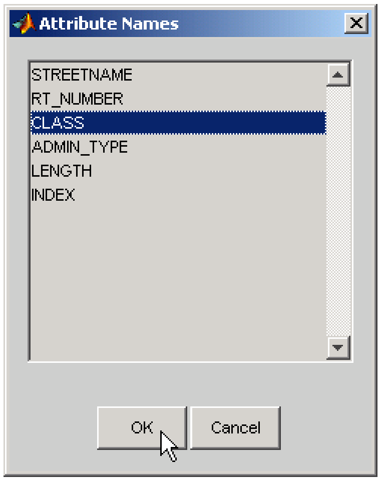
```
  STREETNAME:
'MASSACHUSETTS AVENUE'
   RT_NUMBER: '2A'
       CLASS: 3
  ADMIN_TYPE: 3
      LENGTH: 390.7276
       INDEX: 1724
```

X: 766654.48
Y: 2954492.84
Scale: 1:25000
Map units: US Survey Feet
Active layer: boston_roads

The selected road is Massachusetts Avenue (Route 2A). As the above figure shows, the `boston_roads` vectors have six attributes.

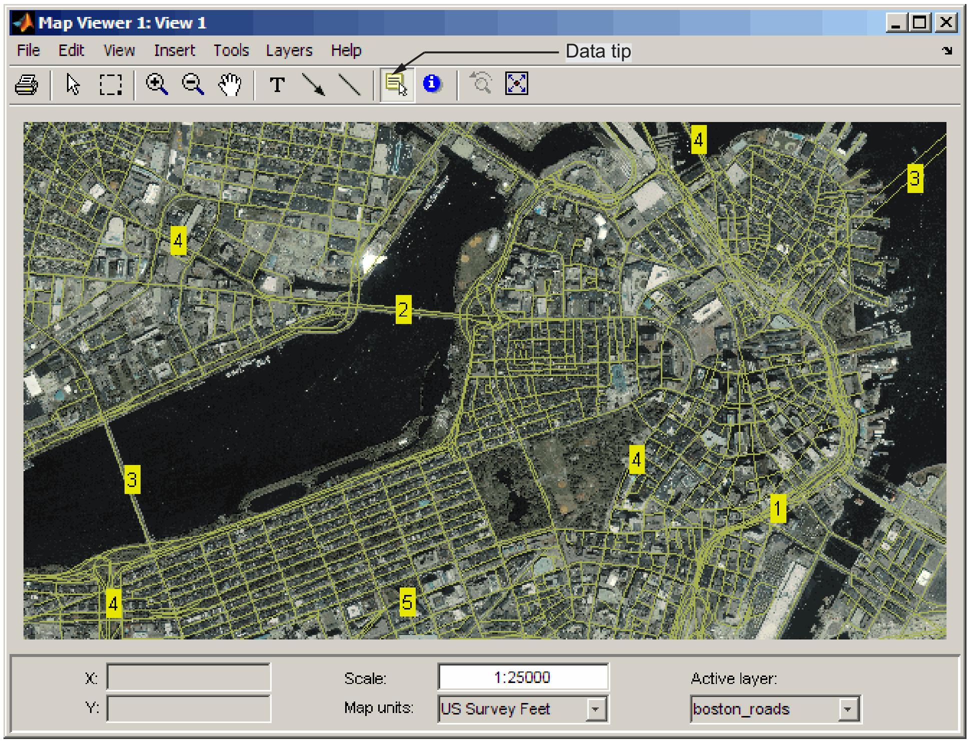
11 Get information about some other roads. Dismiss open Info windows by clicking their close boxes.

12 Choose an attribute for the **Datatip** tool to inspect. From the **Layers** menu, select **boston_roads > Set Label Attribute**. From the list in the list box of the Attribute Names dialog, select `CLASS` and click **OK** to dismiss it. The dialog looks like this.



13 Select the **Datatip** tool. The cursor assumes a crosshairs (+) shape.

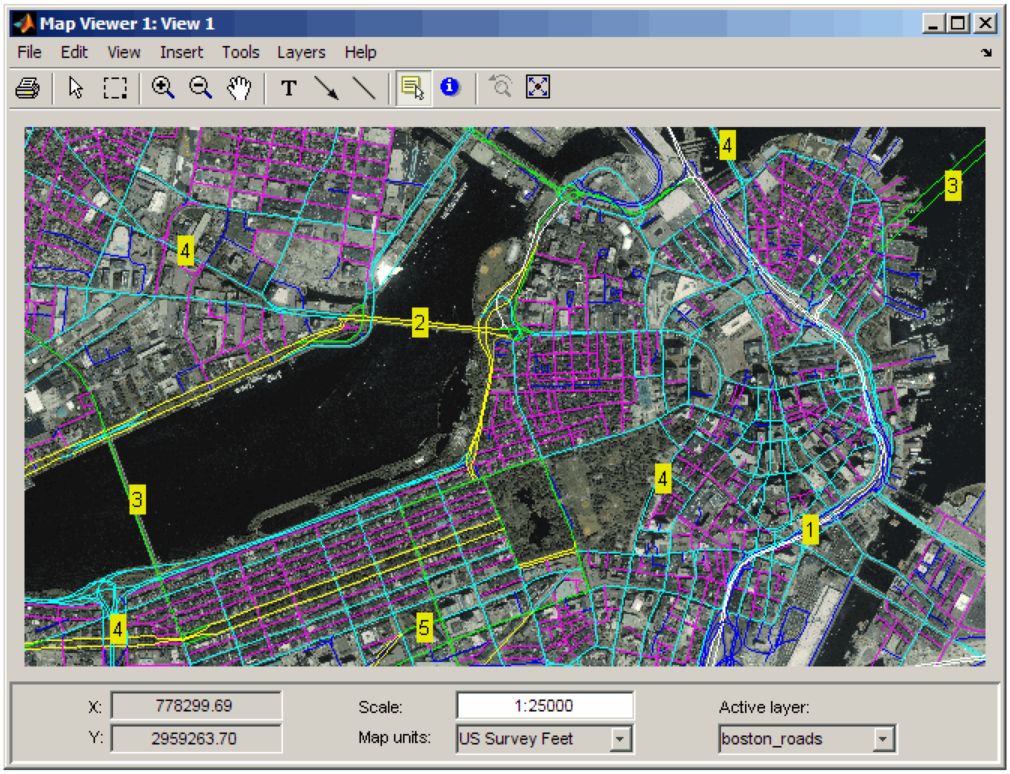
14 Use the **Datatip** tool to identify the administrative class of any road displayed. When you click on a road segment, a data tip is left in that place to indicate the `CLASS` attribute of the active layer, as illustrated below.



15 You can change how the roads are rendered by identifying an attribute to which to key line symbology. Color roads according to their `CLASS` attribute, which takes on the values 1:6. Do this by creating a *symbolspec* in the workspace. A symbolspec is a cell array that associates attribute names and values to graphic properties for a specified geometric class (`'Point'`, `'MultiPoint'`, `'Line'`, `'Polygon'`, or `'Patch'`). To create a symbolspec for line objects (in this case roads) that have a `CLASS` attribute, type

```
roadcolors = makesymbolspec('Line', ...
{'CLASS',1,'Color',[1 1 1]}, {'CLASS',2,'Color',[1 1 0]}, ...
{'CLASS',3,'Color',[0 1 0]}, {'CLASS',4,'Color',[0 1 1]}, ...
{'CLASS',5,'Color',[1 0 1]}, {'CLASS',6,'Color',[0 0 1]})

roadcolors =
     ShapeType: 'Line'
         Color: {6x3 cell}
```

16 The Map Viewer recognizes and imports symbolspecs from the workspace. To apply the one you just created, select **boston_roads > Set Symbol Spec** from the **Layers** menu. From the Set Symbol Spec dialog, select the `roadcolors` symbolspec you just created and click **OK**. After `mapview` has read and applied the symbolspec, the map looks like this.



X: 778299.69
Y: 2959263.70
Scale: 1:25000
Map units: US Survey Feet
Active layer: boston_roads

17 Remove the datatips before going on. To dismiss data tips, right-click each of them and select **Delete datatip** or **Delete all datatips** from the pop-up context menu that appears.

18 Add another layer, a set of points that identify 13 Boston landmarks. As you did with the `boston_roads` layer, you import it from a shapefile. The locations for these landmarks are also given in meters, so you must convert their coordinates to units of survey feet before importing them into Map Viewer, as before.

Read the shapefile and convert from meters to survey feet using this code:

```
boston_placenames = shaperead('boston_placenames.shp');
surveyFeetPerMeter = unitsratio('survey feet','meter');
for k = 1:numel(boston_placenames)
    boston_placenames(k).X = ...
        surveyFeetPerMeter * boston_placenames(k).X;
    boston_placenames(k).Y = ...
        surveyFeetPerMeter * boston_placenames(k).Y;
end
```

From the **File** menu choose **Import From Workspace > Vector Data > Geographic Data Structure**; choose `boston_placenames` as the data to import from the workspace by selecting it, and click **OK**:

The points of interest are symbolized as small `x` markers.

19 As the `boston_placenames` markers are difficult to see over the orthophoto, hide the other map layers temporarily. To do this, go to the **Layers** menu, select **boston_roads**, and then slide right and deselect **Visible**. Do the same to hide the **boston** image layer.

You can now see the 13 markers showing points of interest.

20 To make the markers more visually prominent, create a symbolspec for them to represent them as red filled circles. At the MATLAB command line, type

```
places = makesymbolspec('Point',{'Default','Marker','o', ...
'MarkerEdgeColor','r','MarkerFaceColor','r'})
```

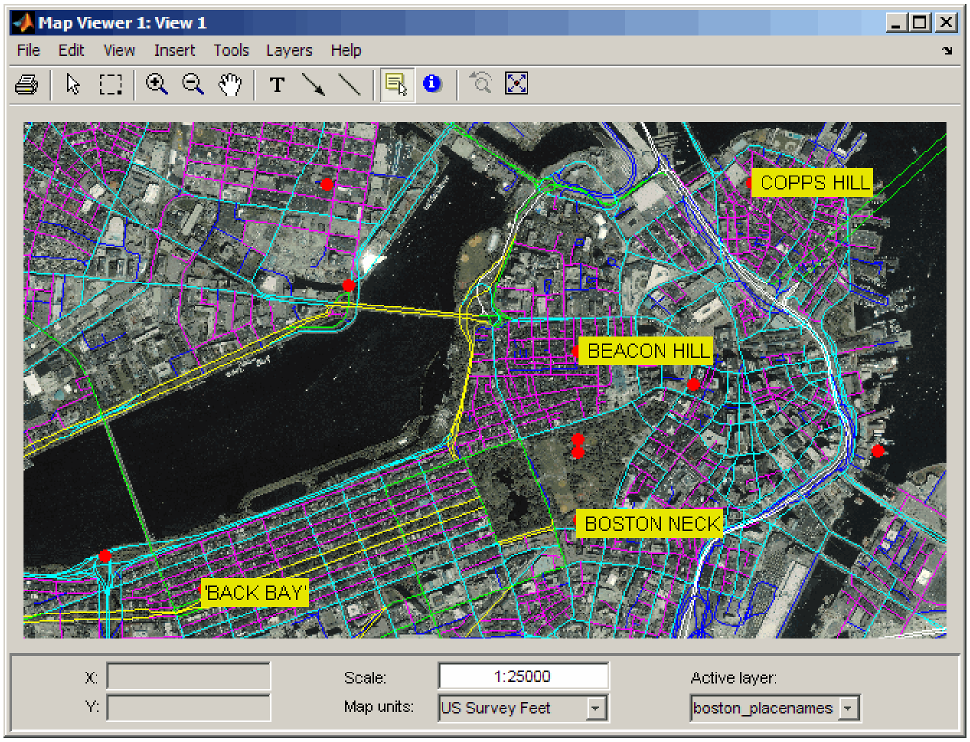
The `Default` keyword causes the specified symbol to be applied to all point objects in a given layer unless specifically overridden by an attribute-coded symbol in the same or a different symbolspec.

21 To activate this symbolspec, pull down the **Layers** menu, select **boston_placenames**, slide right, and select **Set Symbol Spec**. In the Layer Symbols dialog that appears, highlight `places` and click **OK**.

The Map Viewer reads the workspace variable `places`; the cross marks turn into red circles. Note that a layer need not be active in order for you to apply a symbolspec to it.

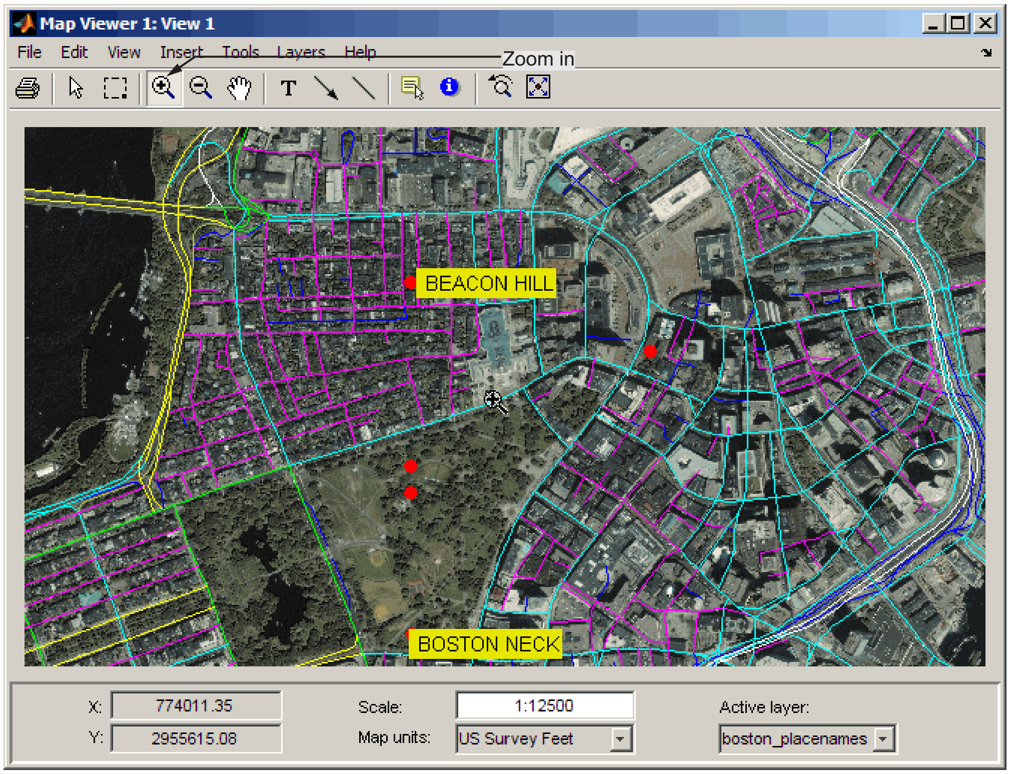
22 Now restore the other layers' visibility. In the **Layers** menu, select **boston_roads**, and then slide right and select **Visible**. Do the same to show the **boston** image layer. The `boston_placenames` marker layer, because it was read in most recently, is on top.

23 Use the **Active layer** drop-down menu to make `boston_placenames` the currently active layer, and then select the **Datatip** tool. Click any red circle to see the name of the feature it marks. The map looks like this (depending on which data tips you show).



Scale: 1:25000
Map units: US Survey Feet
Active layer: boston_placenames

24 Zoom in on Beacon Hill for a closer view of the Massachusetts State House and Boston Common. Select the **Zoom in** tool, move the (magnifier) cursor until the **X** readout is approximately 236,000 M and the **Y** readout is roughly 900,900 M, then click once to enlarge the view. The scale changes to about 1:10,000 and the map appears as below.

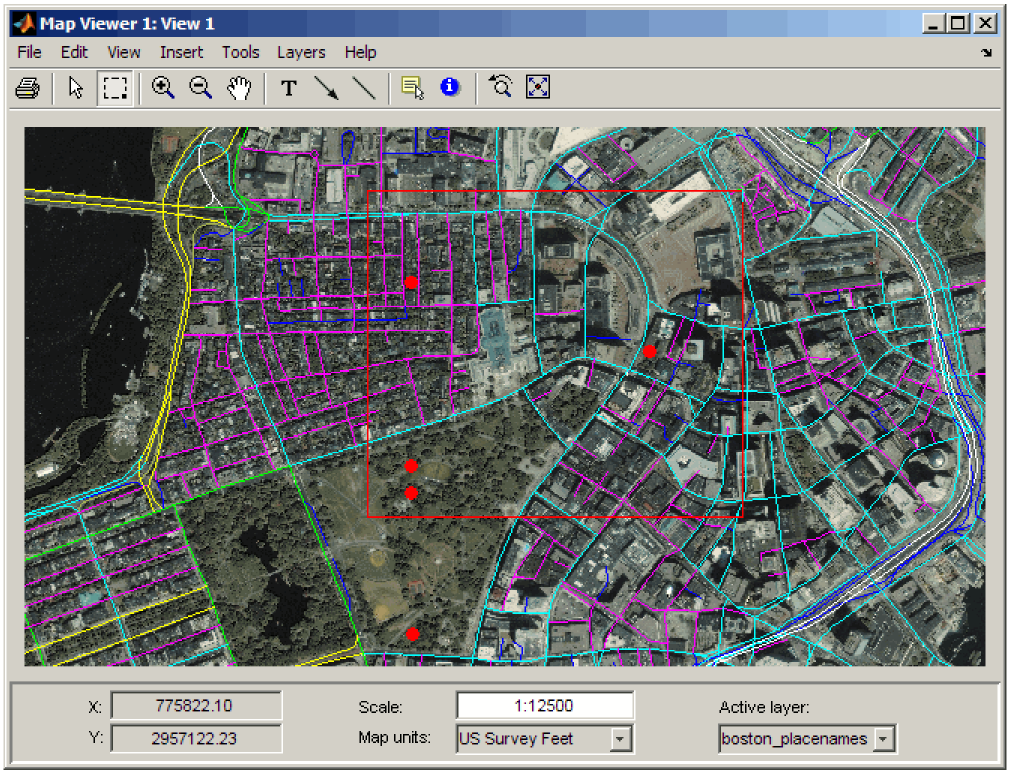

Map Viewer 1: View 1
File Edit View Insert Tools Layers Help
Zoom in
BEACON HILL
BOSTON NECK
X: 774011.35
Y: 2955615.08
Scale: 1:12500
Map units: US Survey Feet
Active layer: boston_placenames

25 Right-click any of the data tips and select **Delete all datatips** from the pop-up context menu. This clears the place names you added to the maps.

26 Select an area of interest to save as an image file. Click the **Select area** tool, and then hold the mouse button down as you draw a selection rectangle. If you do not like the selection, repeat the operation until you are satisfied. If you know what ground coordinates you want, you can use the coordinate readouts to make a precise selection. The selected area appears as a red rectangle.

27 In order to be able to save a file in the next step, change your working directory to a writable directory, such as `/work`.

28 Save your selection as an image file. From the **File** menu, select **Save As Raster Map > Selected Area** to open a Save As dialog, as shown below.



X: 775822.10
Y: 2957122.23
Scale: 1:12500
Map units: US Survey Feet
Active layer: boston_placenames

In the Export to File dialog, navigate to a directory where you want to save the map image, and save the selected area's image as a `.tif` file, calling it `central_boston.tif` (PNG and JPG formats are also available). A worldfile, `central_boston.tfw`, is created there along with the TIF.

Whenever you save a raster map in this manner, two files are created:

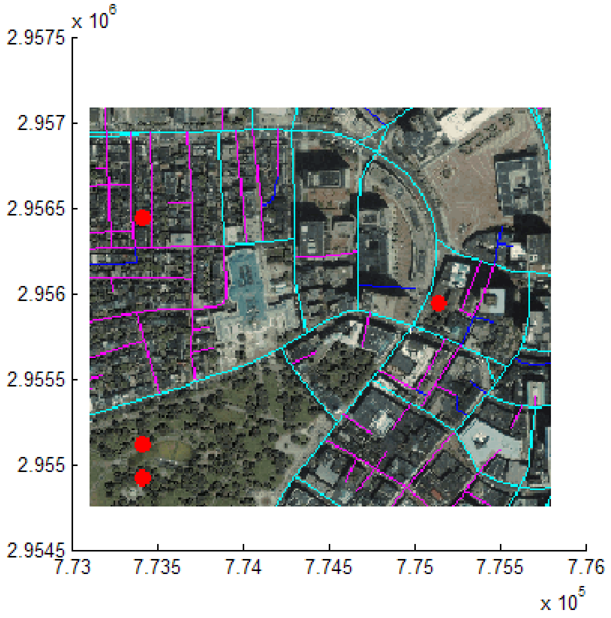
- An image file (`*file*.tif`, `*file*.png`, or `*file*.jpg`)

- An accompanying *worldfile* that georeferences the image (*`file`*`.tfw`, *`file`*`.pgw`, or *`file`*`.jgw`)

The following steps shows you how to read such files and display a georeferenced image outside of `mapview`.

29 Read in the saved image and its colormap with the MATLAB function `imread`, create a referencing matrix for it by reading in `central_boston.tfw` with `worldfileread`, and display with `mapshow`:

```
[image cmap] = imread('central_boston.tif');
R = worldfileread('central_boston.tfw');
figure
mapshow(image, cmap, R);
```



See the documentation for `mapshow` for another example of displaying a georeferenced image.

30 Experiment with other tools and menu items. For example, you can annotate the map with lines, arrows, and text, fit the map to the window, draw a bounding box for any layer, and print the current view. You can also spawn a new Map Viewer using **New View** from the **File** menu. A new view can duplicate the current view, cover the active layer's extent, cover all layer extents, or include only the selected area, if any. When you are through with a viewing session, close the Map Viewer using the window's close box or select **Close** from the **File** menu.

Getting More Help

In this section...

Ways to Get Mapping Toolbox Help

The Mapping Toolbox documentation is available in electronic form as PDF and HTML files through the `helpdesk` command. You might want to print the reference chapters to browse through them. This is best done from the PDF version, available at the MathWorks Web site, `http://www.mathworks.com/access/helpdesk/help/pdf_doc/map/map_ug.pdf`.

You can find a classified list of functions in the "Geospatial Data Import and Access" on page 11-2 (online only). Help is available for individual commands and classes of Mapping Toolbox commands:

- `help map` for computational functions
- `mapdemos` for a list of Mapping Toolbox demos
- `maps` lists all Mapping Toolbox map projections by class, name, and ID string.
- `maplist` returns a structure describing all Mapping Toolbox map projections.
- `projlist` to list map projections supported by `projfwd` and `projinv`
- `help` *`functionname`* for help on a specific function, often including examples
- `helpwin` *`functioname`* to see the output of help displayed in the Help browser window instead of the Command Window
- `doc` *`functionname`* to read a function's reference page in the Help browser, including examples and illustrations

Consulting Release Notes

To learn how one version of Mapping Toolbox software differs from the next and what important changes have been introduced in it, read the Mapping

Toolbox Release Notes, which include information on enhancements, syntax and GUI changes, known software and documentation problems, and compatibility issues.

Mapping Toolbox Demos and Data

In this section...

Available Demos

You can run demonstrations of Mapping Toolbox functions to further acquaint you with their use. Most of the demos highlight and explain features added in the current version. To see the full list of demos, click the **Demos** icon in the **Contents** pane in the **Help Navigator**. Another way to obtain this list is to type

```
mapdemos
```

at the MATLAB prompt. This will bring the Help browser to the fore.

When you view any of the following demos, you can then execute and view its code by clicking links in the banner for that page that say "**Run in the Command Window**" and "**Open <demo>.m in the Editor**":

- mapexkmlexport — Exporting Vector Point Data to KML
- mapexfindcity — Interactive Global City Finder
- mapexgeo — Creating Maps Using `geoshow` (for latitude, longitude data)
- mapexmap — Creating Maps Using `mapshow` (for x, y data)
- mapexrefmat — Creating a Half-Resolution Georeferenced Image
- mapexreg — Georeferencing an Image to an Orthotile Base Layer
- mapex3ddome — Plotting a 3-D Dome as a Mesh Over a Globe
- mapexunprojectdem — Un-Projecting a Digital Elevation Model (DEM)
- mapexgshhs — Converting Coastline Data (GSHHS) to Shapefile Format
- mapexwmsanimate — Compositing and Animating Web Map Service (WMS) Meteorological Layers

You can type

```
help mapdemos
```

to see this list of links as well as descriptions of the sample data provided in Mapping Toolbox.

Locating Geospatial Data

Sample data sets are provided in the Mapping Toolbox `mapdemos` directory. You can find them along with the demos described below in `toolbox\map\mapdemos`. Most of the sample data sets have `.txt` files that provide information or metadata about the their source and content.

For information on locating digital map data you can download over the Internet, see the following documentation at the MathWorks Web site. `http://www.mathworks.com/support/tech-notes/2100/2101.html`

Understanding Map Data

Maps and Map Data

In this section...

What Is a Map?

Mapping Toolbox software manipulates electronic representations of geographic data. It lets you import, create, use, and present geographic data in a variety of forms and to a variety of ends. In the digital network era, it is easy to think of geospatial data as maps and maps as data, but you should take care to note the differences between these concepts.

The simplest (although perhaps not the most general) definition of a *map* is *a representation of geographic data*. Most people today generally think of maps as two-dimensional; to the ancient Egyptians, however, maps first took the form of lists of place names in the order they would be encountered when following a given road. Today such a list would be considered as *map data* rather than as a map. When most people hear the word "map" they tend to visualize two-dimensional renditions such as printed road, political, and topographic maps, but even classroom globes and computer graphic flight simulation scenes are maps under this definition.

In this toolbox, map data is any variable or set of variables representing a set of geographic locations, properties of a region, or features on a planet's surface, regardless of how large or complex the data is, or how it is formatted. Such data can be rendered as maps in a variety of ways using the functions and user interfaces provided.

What Is Geospatial Data?

Geospatial data comes in many forms and formats, and its structure is more complicated than tabular or even nongeographic geometric data. It is, in fact, a subset of spatial data, which is simply data that indicates where things are within a given *coordinate system*. Mileposts on a highway, an engineering drawing of an automobile part, and a rendering of a building elevation all have coordinate systems, and can be represented as spatial data when properly quantified (digitized). Such coordinate systems, however, are local and not explicitly tied or oriented to the Earth's surface; thus, most digital representations of mileposts, machine parts, and buildings do not qualify as geospatial data (also called *geodata*).

What sets geospatial data apart from other spatial data is that it is absolutely or relatively positioned on a planet, or *georeferenced*. That is, it has a *terrestrial coordinate system* that can be shared by other geospatial data. There are many ways to define a terrestrial coordinate system and also to transform it to any number of local coordinate systems, for example, to create a map projection. However, most are based on a framework that represents a planet as a sphere or spheroid that spins on a north-south axis, and which is girded by an *equator* (an imaginary plane midway between the poles and perpendicular to the rotational axis).

Geodata is coded for computer storage and applications in two principal ways: *vector* and *raster* representations. It has been said that "raster is faster but vector is corrector." There is truth to this, but the situation is more complex. The following discussions explore these two representations: how they differ, what data structures support them, why you would choose one over the other, and how they can work together in the toolbox. The conclude by summarizing the functions available for importing and exporting geospatial data formats.

Types of Map Data Handled by the Toolbox

In this section...

Vector Geodata

Vector data (in the computer graphics sense rather than the physics sense) can represent a map. Such vectors take the form of sequences of latitude-longitude or projected coordinate pairs representing a point set, a linear map feature, or an areal map feature. For example, points delineating the boundary of the United States, the interstate highway system, the centers of major U.S. cities, or even all three sets taken together, can be used to make a map. In such representations, the geographic data is in *vector* format and displays of it are referred to as *vector maps*. Such data consists of lists of specific coordinate locations (which, if describing linear or areal features, are normally points of inflection where line direction changes), along with some indication of whether each is connected to the points adjacent to it in the list.

In the Mapping Toolbox environment, vector data consists of sequentially ordered pairs of geographic (latitude, longitude) or projected (*x,y*) coordinate pairs (also called *tuples*). Successive pairs are assumed to be connected in sequence; breaks in connectivity must be delineated by the creation of separate vector variables or by inserting separators (usually NaNs) into the sets at each breakpoint. For vector map data, the connectivity (topological structure) of the data is often only a concern during display, but it also affects the computation of statistics such as length and area.

A Look at Vector Data

1 To inspect an example of vector map data, enter the following commands:

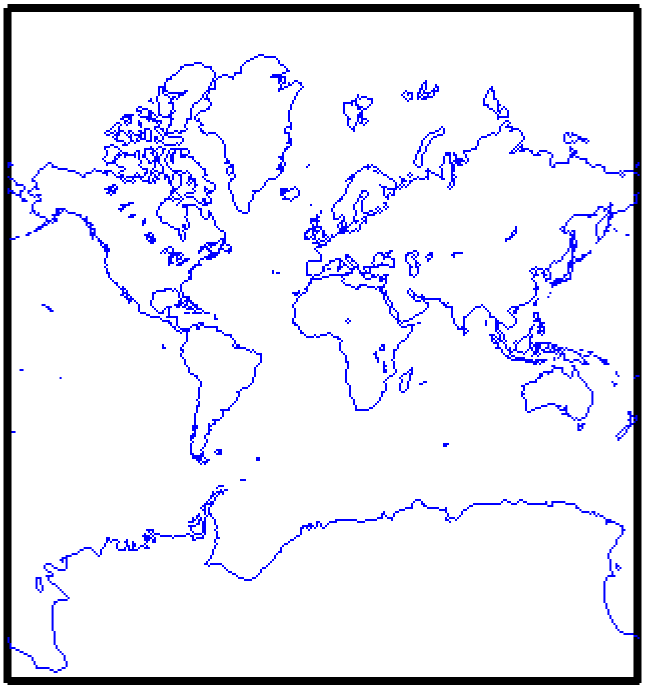
```
load coast
whos
  Name         Size              Bytes  Class     Attributes
```

```
ans            1x45                   90  char
lat         9589x1                 76712  double
long        9589x1                 76712  double
```

The variables `lat` and `long` are vectors in the `coast` MAT-file, which together form a vector map of the coastlines of the world.

2 To view a map of this data, enter these commands:

```
axesm mercator
framem
plotm(lat,long)
```



Inspect the first 20 coordinates of the coastline vector data:

```
[lat(1:20) long(1:20)]
```

```
ans =
-83.83 -180
-84.33 -178
-84.5 -174
-84.67 -170
-84.92 -166
-85.42 -163
-85.42 -158
-85.58 -152
-85.33 -146
-84.83 -147
-84.5 -151
-84 -153.5
-83.5 -153
-83 -154
-82.5 -154
-82 -154
-81.5 -154.5
-81.17 -153
-81 -150
-80.92 -146.5
```

Does this give you any clue as to which continent's coastline these locations represent?

3 To see the coastline these vector points represent, type this command to display them in red:

```
plotm(lat(1:20), long(1:20),'r')
```

As you may have deduced by looking at the first column of the data, there is only one continent that lies below -80º latitude, Antarctica.

The above example presents the map in a Mercator projection. A map projection displays the surface of a sphere (or a spheroid) in a two-dimensional plane. As the word "plane" indicates, points on the sphere are geometrically projected to a plane surface. There are many possible ways to project a map, all of which introduce various types of distortions.

For further information on how Mapping Toolbox software manages map projections, see Chapter 8, "Using Map Projections and Coordinate Systems".

For details on data structures that the toolbox uses to represent vector geodata, see “Mapping Toolbox Geographic Data Structures” on page 2-16.

Raster Geodata

You can also map data represented as a *matrix* (a 2-D MATLAB array) in which each row-and-column element corresponds to a rectangular patch of a specific geographic area, with implied topological connectivity to adjacent patches. This is commonly referred to as *raster data*. *Raster* is actually a hardware term meaning a systematic scan of an image that encodes it into a regular grid of pixel values arrayed in rows and columns.

When data in raster format represents the surface of a planet, it is called a *data grid*, and the data is stored as an array or matrix. The toolbox leverages the power of MATLAB matrix manipulation in handling this type of map data. This documentation uses the terms *raster data* and *data grid* interchangeably to talk about geodata stored in two-dimensional array form.

A raster can encode either an average value across a cell or a value sampled (posted) at the center of that cell. While geolocated data grids explicitly indicate which type of values are present (see “Geolocated Data Grids” on page 2-44), external metadata/user knowledge is required to be able to specify whether a regular data grid encodes averages or samples of values.

Digital Elevation Data

When raster geodata consists of surface elevations, the map can also be referred to as a *digital elevation model/matrix* (DEM), and its display is a *topographical map*. The DEM is one of the most common forms of *digital terrain model* (DTM), which can also be represented as contour lines, triangulated elevation points, quadtrees, octtrees, or otherwise.

The `topo` global terrain data is an example of a DEM. In this 180-by-360 matrix, each row represents one degree of latitude, and each column represents one degree of longitude. Each element of this matrix is the average elevation, in meters, for the one-degree-by-one-degree region of the Earth to which its row and column correspond.

Remotely Sensed Image Data

Raster geodata also encompasses georeferenced imagery. Like data grids, images are organized into rows and columns. There are subtle distinctions, however, which are important in certain contexts. One distinction is that an image may contain RGB or multispectral channels in a single array, so that it has a third (color or spectral) dimension. In this case a 3-D array is used rather than a 2-D (matrix) array. Another distinction is that while data grids are stored as class double in the toolbox, images may use a range of MATLAB storage classes, with the most common being `uint8`, `uint16`, `double`, and `logical`. Finally, for grayscale and RGB images of class double, the values of individual array elements are constrained to the interval `[0 1]`.

In terms of georeferencing—converting between column/row subscripts and 2-D map or geographic coordinates—images and data grids behave the same way (which is why both are considered to be a form of raster geodata). However, when performing operations that process the values raster elements themselves, including most display functions, it is important to be aware of whether you are working with an image or a data grid, and for images, how spectral data is encoded.

For further details concerning the structure of raster map data, see "Understanding Raster Geodata" on page 2-33.

A Look at Raster Data

1 Load the `topo` data grid.

```
load topo topo
```

2 `topo` contains raster elevation data. Create a referencing matrix to georeference `topo`.

```
topoR = makerefmat('RasterSize', size(topo), ...
   'Latlim', [-90 90], 'Lonlim', [-180 180]);
```

3 Create an equal-area map projection to view the topographic data:

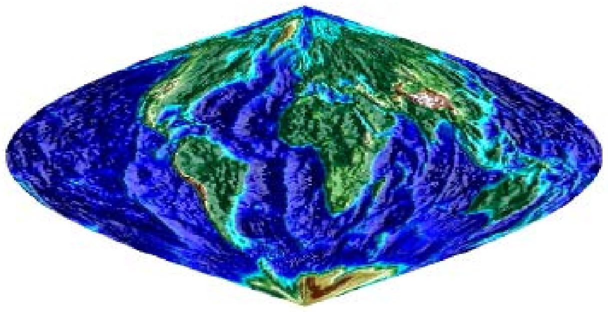
```
axesm sinusoid
```

A figure window is created with map axes set to display a sinusoidal projection.

4 Generate a shaded relief map. You can do this in several ways. First use `geoshow` and apply a topographic colormap using `demcmap`:

```
geoshow(topo,topoR,'DisplayType','texturemap')
demcmap(topo)
```

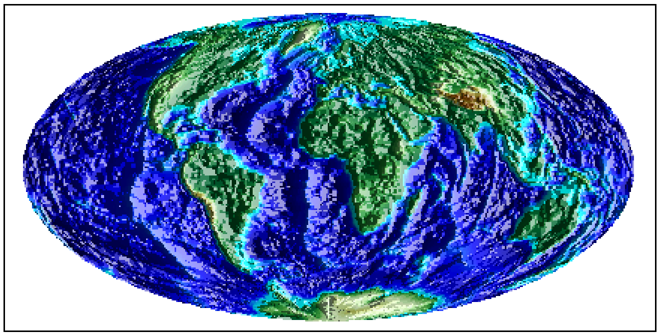
The `geoshow` function displays geodata in geographic (unprojected) coordinates. The `geoshow` output is shown below:



5 Now create a new figure using a Hammer projection (which, like the sinusoidal, is also equal-area), and display `topo` using `meshlsrm`, which enables control of lighting effects:

```
figure; axesm hammer
meshlsrm(topo,topoR)
```

A colored relief map of the `topo` data set, illuminated from the east, is rendered in the second figure window.



For additional details on controlling the illumination of maps, see "Shading and Lighting Terrain Maps" on page 5-22.

Note that the content, symbolization, and the projection of the map are completely independent. The structure and content of the `topo` variable are the same no matter how you display it, although how it is projected and symbolized can affect its interpretation. The following example illustrates this.

Combining Vector and Raster Geodata

Vector map variables and data grid variables are often used or displayed together. For example, continental coastlines in vector form might be displayed with a grid of temperature data to make the latter more useful. When several map variables are used together, regardless of type, they can be referred to as a single map. To do this, of course, the different data sets must use the same coordinate system (i.e., geographic coordinates on the same ellipsoid or an identical map projection). See Chapter 3, "Understanding Geospatial Geometry" for an introduction to these concepts.

Viewing Raster and Vector Data on the Same Map

Using the `coast` and `topo` data from the previous examples, you can combine them in a single map and see how well the two types of data work together:

1 Clear the current map:

```
clma
```

2 Reload the coastline data:

```
load coast
```

3 If the topo data is not already in the workspace, load it as well:

```
load topo
```

4 Set up a Robinson projection:

```
axesm robinson
```

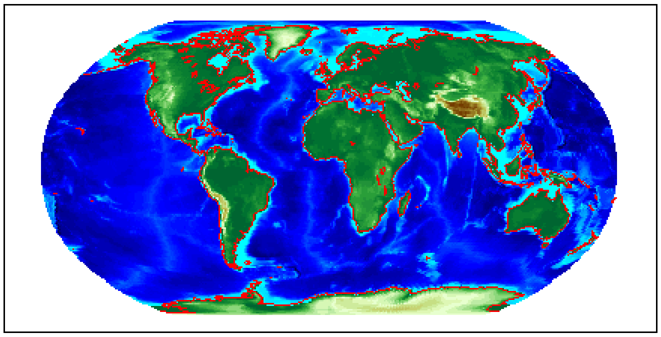
5 Plot the raster topographic data with an appropriate colormap:

```
geoshow(topo,topolegend,'DisplayType','texturemap')
demcmap(topo)
```

6 Plot the coastline data in white on top of the terrain map:

```
geoshow(lat,long,'Color','r')
```

Note that you can use `geoshow` to display both raster and vector data. Here is the resulting map.



For additional details on how Mapping Toolbox functions handles raster geodata, see "Understanding Raster Geodata" on page 2-33.

The remainder of this chapter focuses on the fundamental principles of geographic measurement and data manipulation that are a prerequisite for creating map displays. "Reading and Writing Geospatial Data" on page 2-52 summarizes input functions for importing many formats of geospatial data into the toolbox. Chapter 3, “Understanding Geospatial Geometry” introduces geodetic concepts that underlie all geospatial data and its handling.

Understanding Vector Geodata

In this section...

Points, Lines, Polygons

Vector geospatial data is used to represent linear features such as rivers, coastlines, boundaries, and highways. Vector data can also represent areal features such as water bodies, political units, and enumeration districts. This section familiarizes you with how vector data structures digitally encode geographic entities and how to use this form of data.

In the context of geodata, *vector data* means "geometric descriptions of geographic objects" rather than its more general mathematical definition, "a quantity specified by a magnitude and a direction." In fact, some vector geodata is specified as points having neither magnitude nor direction. Other geodata—such as postcodes, highway mileposts, or census statistics—only implies an underlying geometry, which vector 2-D coordinate data is required to map or spatially analyze.

In the MATLAB workspace, vector data is expressed as pairs of variables that represent the geographic or plane coordinates for a set of points of interest. For example, the following two variables can be mapped as a vector:

```
lat = [45.6 -23.47 78];
long = [13 -97.45 165];
```

Note that either row or column vectors can be used, but both variables should have the same shape. For example, `lat` and `long` could be defined as columns:

```
lat = [45.6 -23.47 78]';
long = [13 -97.45 165]';
```

These values could mean anything. They could represent three locations over which geosynchronous satellites are stationed, and can be communicated by plotting a symbol for each point on a map of the Earth. Alternatively, they might represent a starting point, a midcourse marker, and a finish point of a sailboat race, in which case they can be rendered by plotting two line segments. Or perhaps the values represent the vertices of a triangle bounding a region of interest, and thus constitute a simple polygon.

Note When polygons become graphic objects, they are called patches. In this documentation, the words *patch* and *polygon* are often used interchangeably.

Mapping Toolbox functions provide for each of these interpretations. For many purposes, the distinction is irrelevant; for others, the choice of a function implies one interpretation over the others. For example, the function `plotm` displays the data as a line, while `fillm` displays it as a filled polygon. While you can draw an unfilled polygon with `fillm` that looks like the output from `plotm`, the resulting object has a different graphic data type (*patch* versus *line*), hence different properties you can set.

A line must contain at least two coordinate elements for each coordinate dimension, and a polygon at least three (note that it is not necessary to duplicate the first point as the last point to define or render a polygon). The toolbox places no limit (beyond available memory) on how large or how complex the shape of a line and polygon can be, other than the restriction that it should not cross itself.

Objects in the real world that vector geodata represents can have many parts, for example, the islands that make up the state of Hawaii. When encoding as vector variables the shapes of such compound entities, you must separate successive entities. To indicate that such a discontinuity exists, the toolbox uses the convention of placing NaNs in identical positions in both vector variables. For example, if a second segment is to be added to the preceding map, the two objects can reside in the same pair of variables:

```
lat = [45.6 -23.47 78 NaN 43.9 -67.14 90 -89];
lon = [13 -97.45 165 NaN 0 -114.2 -18 0];
```

Notice that the NaNs must appear in the same locations in both variables. Here is a segment of three points separated from a segment of four points. The NaNs perform two functions: they provide a means of identifying breakpoints in the data, and they serve as *pen-up* commands when plotting vector maps. The NaNs are used to separate both distinct (but possibly connecting) lines and disconnected patch faces.

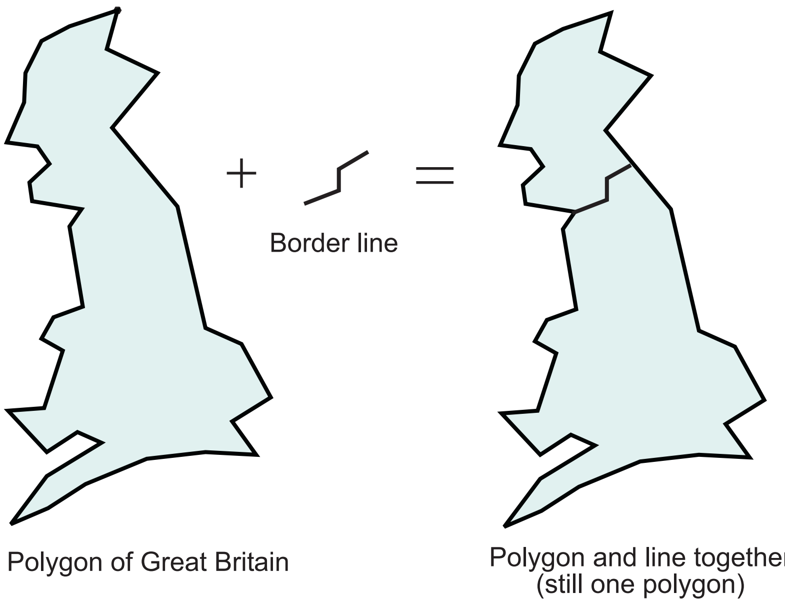
Note This convention departs from regular MATLAB graphics, in which NaN-separated polygons cannot be interpreted or displayed as separate patches.

Segments Versus Polygons

Geographic objects represented by vector data might or might not be formatted as polygons. Imagine two variables, `latcoast` and `loncoast`, that correspond to a sequence of points that caricature the coast of the island of Great Britain. If this data returns to its starting point, then a polygon describing Great Britain exists. This data might be plotted as a patch or as a line, and it might be logically employed in calculations as either.

Now suppose you want to represent the Anglo-Scottish border, proceeding from the west coast at Solway Firth to the east coast at Berwick-upon-Tweed. This data can only be properly defined as a line, defined by two or more points, which you can represent with two more variables, `latborder` and `lonborder`. When plotted together, the two pairs of variables can form a map. The patch of Great Britain plus the line showing the Scottish border might look like two patches or regions, but there is no object that represents England and no object that represents Scotland, either in the workspace or on the map axes.

In order to represent both regions properly, the Great Britain polygon needs to be split at the two points where the border meets it, and a copy of `latborder` and `lonborder` concatenated to both lines (placing one in reverse order). The resulting two polygons can be represented separately (e.g., in four variables named `latengland`, `lonengland`, `latscotland`, and `lonscotland`) or in two variables that define two polygons each, delineated by NaNs (e.g., `latuk`, `lonuk`).



The distinction between line and polygon data might not appear to be important, but it can make a difference when you are performing geographic analysis and thematic mapping. For example, polygon data can be treated as line data and displayed with functions such as `linem`, but line data cannot be handled as polygons unless it is restructured to make all objects close on themselves, as described in "Matching Line Segments" on page 7-4.

Mapping Toolbox Geographic Data Structures

In examples provided in prior chapters, geodata was in the form of individual variables. Mapping Toolbox software also provides an easy means of displaying, extracting, and manipulating collections vector map features that have been organized in a family of specially organized *geographic data structures*.

Note The data structures discussed in this section are different from the *map projection structure* (also called an *mstruct*), which defines a map projection and related display properties for map axes. See the `defaultm` function reference page for information about mstructs.

The following subsections describe what the geographic data structures contain and how to create and display them. Version 1 of the toolbox used a different kind of geographic data structure (called a *display structure*), which had a more rigid definition.

Note Version 1 display structures are being phased out of the toolbox and are currently generated only by a few functions. You can use the `updategeostruct` to convert a display structure containing vector features to a geographic data structure. You must do this in order to export data imported from VMAP level 0 or DCW data sets with `shapewrite`, for example. For more information, see the `displaym` reference page, which describes the format and contents of display structures.

Representing Geographic Features

Certain Mapping Toolbox functions read, create, or manipulate vector geodata organized within a geographic data structure. This is a MATLAB structure array that has one element per geographic feature. Each feature is represented by coordinates and any kind and number of attributes. When it holds geographic coordinates (latitude and longitude), it is called a *geostruct*; when it contains plane (projected *x* and *y*) map coordinates, it is called a *mapstruct*.

Geostructs and mapstructs most frequently originate when vector geodata is imported from a shapefile. They hold only vector features and cannot be used to hold raster data, whether images, regular data grids, or geolocated data grids. The `shaperead` function returns either a geostruct or a mapstruct that encapsulates some or all of the data stored in a shapefile data set (which stores attributes and coordinates in separate files). The `gshhs` function also returns a geostruct.

By default, `shaperead` returns a mapstruct containing `X` and `Y` fields. This is appropriate if the data set coordinates are already projected (are in map space). Otherwise, when the coordinate data is unprojected (are in geographic space), use the parameter-value pair `'UseGeoCoords',true` to make `shaperead` return a geostruct having `Lon` and `Lat` fields instead. If you do not know whether coordinates in the shapefile are projected or unprojected, ask your data provider.

To determine what kinds of data a shapefile data set contains, query the data set using the `shapeinfo` function. `shapeinfo` returns a 1-by-1 structure containing high-level descriptions; it cannot be used as a geostruct or mapstruct.

Geographic data structures conform to a few simple rules, making them easy to construct programmatically:

- Each array element represents an entity of the same geometric class (*Point*, *MultiPoint*, *Line*, or *Polygon*).
- Geostructs: For geographic coordinates, the fields `Geometry`, `Lat`, and `Lon` must be present.
- Mapstructs: For plane coordinates, the fields `Geometry`, `Y`, and `X` must be present.
- Except for Point geometry, a field called `BoundingBox` that contains `[minX minY; maxX maxY]` is provided.
- If present, additional fields (for attributes) must contain either strings or scalar numbers.

Coordinate data is stored in fields called `X` or `Lon` and `Y` or `Lat`, depending on what type of coordinates were read in or placed there. The mandatory coordinate field names tell map display and data export functions that accept geographic data structures whether a map projection has already been applied to coordinates, which can prevent them from generating erroneous data and graphics. Be aware that a mapstruct does not itself identify the map projection or projection parameters to which the `X` and `Y` coordinates are referenced. This is also generally true for geodata stored in shapefiles; see the `shapeinfo` reference page for more information.

Any number of attribute fields can be included in a geostruct or mapstruct, and can contain either a double or a string data type. This type must be consistent for a given attribute across all elements of the structure array. The fields in a geostruct representing imported data can vary according to the type of geometry and the names of attributes that have been read. If a shapefile attribute name cannot be directly used as a field name, `shaperead` assigns the field an appropriately mangled name, usually by substituting underscores for spaces. `shaperead` can filter out unwanted attributes; see "Selecting Data to Read with the shaperead Function" on page 2-27 for information on how to set up such filters.

Here is an example of an unfiltered mapstruct returned by `shaperead`:

```
S = shaperead('concord_roads.shp')

S =
609x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

This indicates that the shapefile contains 609 features. In addition to the `Geometry`, `BoundingBox`, and coordinate fields (`X` and `Y`), there are five attribute fields: `STREETNAME`, `RT_NUMBER`, `CLASS`, `ADMIN_TYPE`, and `LENGTH`. Looking at the 10th element,

```
S(10)

ans =
       Geometry: 'Line'
    BoundingBox: [2x2 double]
              X: [1x9 double]
              Y: [1x9 double]
     STREETNAME: 'WRIGHT FARM'
      RT_NUMBER: ''
```

```
          CLASS: 5
     ADMIN_TYPE: 0
         LENGTH: 79.0347
```

you can see that this mapstruct contains line features, and that the tenth line has nine vertices. The first two attributes are string-valued (the second happens to be empty), and the final three attributes are numeric. Across the elements of `S`, `X` and `Y` can have various lengths, but `STREETNAME` and `RT_NUMBER` must always contain strings, and `CLASS`, `ADMIN_TYPE`, and `LENGTH` must always contain scalar doubles.

Note In mapstructs and geostructs having `Line` or `Polygon` geometries, individual features can have multiple parts—separated, disconnected line segments and polygon rings. The parts can include inner rings that run counterclockwise and outline voids. Each disconnected part is separated from the next by a NaN within the `X` and `Y` (or `Lat` and `Lon`) vectors. You can use the `isShapeMultipart` function to determine if a feature has NaN-separated parts.

Each NaN-separated multipart line, polygon, or multipoint entity constitutes a single feature and thus has one string or scalar double value per attribute field. It is not possible to assign distinct attributes to the different parts of such a feature; any string or numeric attribute imported with (or subsequently added to) the geostruct or mapstruct applies to all the feature's parts.

How to Construct Geostructs and Mapstructs

Functions such as `shaperead` or `gshhs` return geostructs when importing vector geodata. However, you might want to create geostructs or mapstructs yourself in some circumstances. For example, you might *import* vector geodata that is not stored in a shapefile (for example, from a MAT-file, from an Microsoft® Excel® spreadsheet, or by reading in a delimited text file). You also might *compute* vector geodata and attributes by calling various MATLAB or Mapping Toolbox functions. In both cases, the coordinates and other data are typically vectors or matrices in the workspace. Packaging variables into a geostruct or mapstruct can make mapping and exporting them easier, because geographic data structures provide several advantages over coordinate arrays:

- All associated geodata variables are packaged in one container, a structure array.
- The structure is self-documenting through its field names.
- You can vary map symbology for points, lines, and polygons according to their attribute values by constructing a *symbolspec* for displaying the geostruct or mapstruct.
- A one-to-one correspondence exists between structure elements and geographic features, which extends to the children of hggroups constructed by `mapshow` and `geoshow`.

Achieving these benefits is not difficult. Use the following example as a guide to packaging vector geodata you import or create into geographic data structures.

Example — Making Point and Line Geostructs. The following example first creates a point geostruct containing three cities on different continents and plots it with `geoshow`. Then it creates a line geostruct containing data for great circle navigational tracks connecting these cities. Finally, it plots these lines using a symbolspec.

1 Begin with a small set of point data, approximate latitudes and longitudes for three cities on three continents:

```
latparis =  48.87084; lonparis =   2.41306;   % Paris coords
latsant  = -33.36907; lonsant  = -70.82851;   % Santiago
latnyc   =  40.69746; lonnyc   = -73.93008;   % New York City
```

2 Build a point geostruct; it needs to have the following required fields:

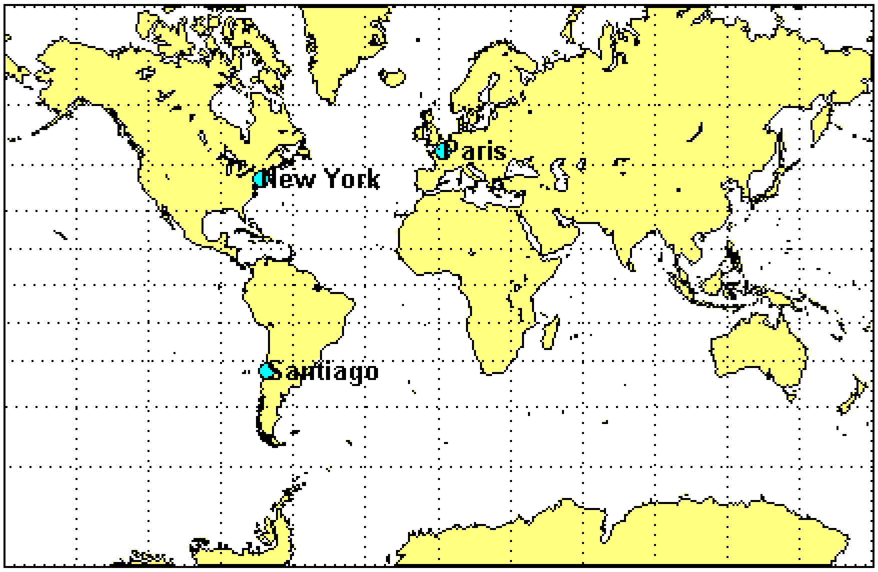
- `Geometry` (in this case `'Point'`)
- `Lat` (for points, this is a scalar double)
- `Lon` (for points, this is a scalar double)

```
% The first field by convention is Geometry (dimensionality).
% As Geometry is the same for all elements, assign it with deal:
[Cities(1:3).Geometry] = deal('Point');

% Add the latitudes and longitudes to the geostruct:
Cities(1).Lat = latparis; Cities(1).Lon = lonparis;
   Cities(2).Lat = latsant;  Cities(2).Lon = lonsant;
   Cities(3).Lat = latnyc;   Cities(3).Lon = lonnyc;

   % Add city names as City fields. You can name optional fields
   % anything you like other than Geometry, Lat, Lon, X, or Y.
   Cities(1).Name = 'Paris';
   Cities(2).Name = 'Santiago';
   Cities(3).Name = 'New York';
   % Inspect your completed geostruct and its first member
   Cities

   Cities =
   1x3 struct array with fields:
       Geometry
       Lat
       Lon
       Name

   Cities(1)

   ans =
       Geometry: 'Point'
            Lat: 48.8708
            Lon: 2.4131
           Name: 'Paris'
```

3 Display the geostruct on a Mercator projection of the Earth's land masses stored in the `landareas.shp` demo shapefile , setting map limits to exclude polar regions:

```
axesm('mercator','grid','on','MapLatLimit',[-75 75]); tightmap;
% Map the geostruct with the continent outlines
geoshow('landareas.shp')

% Map the City locations with filled circular markers
geoshow(Cities,'Marker','o',...
    'MarkerFaceColor','c','MarkerEdgeColor','k');

% Display the city names using data in the geostruct field Name.
% Note that you must treat the Name field as a cell array.
```

```
textm([Cities(:).Lat],[Cities(:).Lon],...
    {Cities(:).Name},'FontWeight','bold');
```



4 Next, build a Line geostruct to package great circle navigational tracks between the three cities:

```
% Call the new geostruct Tracks and give it a line geometry:
[Tracks(1:3).Geometry] = deal('Line');

% Create a text field identifying kind of track each entry is.
% Here they all will be great circles, identified as 'gc'
% (string signifying great circle arc to certain functions)
trackType = 'gc';
[Tracks.Type] = deal(trackType);

% Give each track an identifying name
Tracks(1).Name = 'Paris-Santiago';
[Tracks(1).Lat Tracks(1).Lon] = ...
        track2(trackType,latparis,lonparis,latsant,lonsant);

Tracks(2).Name = 'Santiago-New York';
```

```
[Tracks(2).Lat Tracks(2).Lon] = ...
        track2(trackType,latsant,lonsant,latnyc,lonnyc);

Tracks(3).Name = 'New York-Paris';
[Tracks(3).Lat Tracks(3).Lon] = ...
        track2(trackType,latnyc,lonnyc,latparis,lonparis);
```

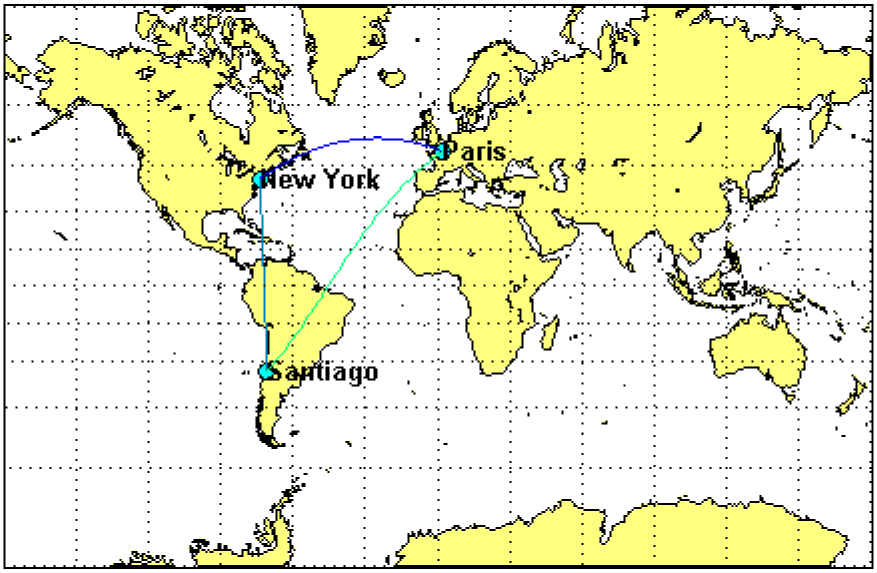
5 Compute lengths of the great circle tracks:

```
% The distance function computes distance and azimuth between
% given points, in degrees. Store both in the geostruct.
for j = 1:numel(Tracks)
    [dist az] = ...
        distance(trackType,Tracks(j).Lat(1),...
                           Tracks(j).Lon(1),...
                           Tracks(j).Lat(end),...
                           Tracks(j).Lon(end));
    [Tracks(j).Length] = dist;
    [Tracks(j).Azimuth] = az;
end
% Inspect the first member of the completed geostruct
Tracks(1)

ans =
    Geometry: 'Line'
        Type: 'gc'
        Name: 'Paris-Santiago'
         Lat: [100x1 double]
         Lon: [100x1 double]
      Length: 104.8274
     Azimuth: 235.8143
```

6 Map the three tracks in the line geostruct:

```
% On cylindrical projections like Mercator, great circle tracks
% are curved except those that follow the Equator or a meridian.

% Graphically differentiate the tracks by creating a symbolspec;
% key line color to track length, using the 'summer' colormap.
% Symbolspecs make it easy to vary color and linetype by
% attribute values. You can also specify default symbologies.
```

```
colorRange = makesymbolspec('Line',...
            {'Length',[min([Tracks.Length]) ...
              max([Tracks.Length])],...
             'Color',winter(3)});
geoshow(Tracks,'SymbolSpec',colorRange);
```



You can save the geostructs you just created as shapefiles by calling `shapewrite` with a filename of your choice, for example:

```
shapewrite(Cities,'citylocs');
shapewrite(Tracks,'citytracks');
```

Making Polygon Geostructs. Creating a geostruct or mapstruct for polygon data is similar to building one for point or line data. However, if your polygons include multiple, NaN-separated parts, recall that they can have only one value per attribute, not one value per part. Each attribute you place in a structure element for such a polygon pertains to all its parts. This means that if you define a group of islands, for example with a single NaN-separated list for each coordinate, all attributes for that element describe the islands as a group, not particular islands. If you want to associate attributes with a particular island, you must provide a distinct structure element for that island.

Be aware that the ordering of polygon vertices matters. When you map polygon data, the direction in which polygons are traversed has significance for how they are rendered by functions such as `geoshow`, `mapshow`, and `mapview`. Proper directionality is particularly important if polygons contain holes. The Mapping Toolbox convention encodes the coordinates of outer rings (e.g., continent and island outlines) in clockwise order; counterclockwise ordering is used for inner rings (e.g., lakes and inland seas). Within the coordinate array, each ring is separated from the one preceding it by a NaN.

When plotted by `mapshow` or `geoshow`, clockwise rings are filled. Counterclockwise rings are unfilled; any underlying symbology shows through such holes. To ensure that outer and inner rings are correctly coded according to the above convention, you can invoke the following functions:

- `ispolycw` — True if vertices of polygonal contour are clockwise ordered
- `poly2cw` — Convert polygonal contour to clockwise ordering
- `poly2ccw` — Convert polygonal contour to counterclockwise ordering
- `poly2fv` — Convert polygonal region to face-vertex form for use with `patch` in order to properly render polygons containing holes

Three of these functions check or change the ordering of vertices that define a polygon, and the fourth one converts polygons with holes to a completely different representation.

For more information about working with polygon geostructs, see the Mapping Toolbox “Converting Coastline Data (GSHHS) to Shapefile Format” demo mapexgshhs.

Mapping Toolbox Version 1 Display Structures

Prior to Version 2, when geostructs and mapstructs were introduced, a different data structure was employed when importing geodata from certain external formats to encapsulate it for map display functions. These *display structures* accommodated both raster and vector map data and other kinds of objects, but lacked the generality of current geostructs and mapstructs for representing vector features and are being phased out of the toolbox. However, you can convert display structures that contain vector geodata to geostruct form using `updategeostruct`. For more information about Version 1 display structures and their usage, see "Version 1 Display Structures" on page 12-142 in the reference page for `displaym`. Additional information is located in reference pages for `updategeostruct`, `extractm`, and `mlayers`.

Selecting Data to Read with the shaperead Function

The `shaperead` function provides you with a powerful method, called a *selector*, to select only the data fields and items you want to import from shapefiles.

A selector is a cell array with two or more elements. The first element is a handle to a predicate function (a function with a single output argument of type `logical`). Each remaining element is a string indicating the name of an attribute.

For a given feature, `shaperead` supplies the values of the attributes listed to the predicate function to help determine whether to include the feature in its output. The feature is excluded if the predicate returns `false`. The converse is not necessarily true: a feature for which the predicate returns `true` may be excluded for other reasons when the selector is used in combination with the bounding box or record number options.

The following examples are arranged in order of increasing sophistication. Although they use MATLAB function handles, anonymous functions, and nested functions, you need not be familiar with these features in order to master the use of selectors for `shaperead`.

Example 1: Predicate Function in Separate File

1 Define the predicate function in a separate file. (Prior to Release 14, this was the only option available.) Create a file named `roadfilter.m`, with the following contents:

```
function result = roadfilter(roadclass,roadlength)
mininumClass = 4;
minimumLength = 200;
result = (roadclass  >= mininumClass) && ...
         (roadlength >= minimumLength);
end
```

2 You can then call `shaperead` like this:

```
roadselector = {@roadfilter, 'CLASS', 'LENGTH'}

roadselector =
    @roadfilter    'CLASS'    'LENGTH'

s = shaperead('concord_roads', 'Selector', roadselector)

s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

or, in a slightly more compact fashion, like this:

```
s = shaperead('concord_roads',...
              'Selector', {@roadfilter, 'CLASS', 'LENGTH'})

s =
115x1 struct array with fields:
        Geometry
        BoundingBox
        X
        Y
        STREETNAME
        RT_NUMBER
        CLASS
        ADMIN_TYPE
        LENGTH
```

Prior to Version 7 of the Mapping Toolbox software, putting the selector in a file or subfunction of its own was the only way to work with a selector.

Note that if the call to `shaperead` took place within a function, then `roadfilter` could be defined in a subfunction thereof rather than in an M-file of its own.

Example 2: Predicate as Function Handle

As a simple variation on the previous example, you could assign a function handle, `roadfilterfcn`, and use it in the selector:

```
roadfilterfcn = @roadfilter
s = shaperead('concord_roads',...
              'Selector', {roadfilterfcn, 'CLASS', 'LENGTH'})
roadfilterfcn =
@roadfilter
s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Example 3: Predicate as Anonymous Function

Having to define predicate functions in M-files of their own, or even as subfunctions, may sometimes be awkward. Anonymous functions allow the predicate function to be defined right where it is needed. For example:

```
roadfilterfcn = ...
    @(roadclass, roadlength) (roadclass >= 4) && ...
    (roadlength >= 200)

roadfilterfcn =
    @(roadclass, roadlength) (roadclass >= 4) ...
               && (roadlength >= 200)

s = shaperead('concord_roads','Selector', ...
              {roadfilterfcn, 'CLASS', 'LENGTH'})

s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Example 4: Predicate (Anonymous Function) Defined Within Cell Array

There is actually no need to introduce a function handle variable when defining the predicate as an anonymous function. Instead, you can place the whole expression within the selector cell array itself, resulting in somewhat more compact code. This pattern is used in many examples throughout Mapping Toolbox documentation and M-file help.

```
s = shaperead('concord_roads', 'Selector', ...
    {@(roadclass, roadlength)...
    (roadclass >= 4) && (roadlength >= 200),...
    'CLASS', 'LENGTH'})

s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Example 5: Parameterizing the Selector; Predicate as Nested Function

In the previous patterns, the predicate involves two hard-coded parameters (called `minimumClass` and `minimumLength` in `roadfilter.m`), as well as the `roadclass` and `roadlength` input variables. If you use any of these patterns in a program, you need to decide on minimum cut-off values for `roadclass` and `roadlength` at the time you write the program. But suppose that you wanted to wait and decide on parameters like `minimumClass` and `minimumLength` at run time?

Fortunately, nested functions provide the additional power that you need to do this; they allow you utilize workspace variables in as parameters, rather than requiring that the parameters be hard-coded as constants within the predicate function. In the following example, the workspace variables `minimumClass` and `minimumLength` could have been assigned through a variety of computations whose results were unknown until run-time, yet their values can be made available within the predicate as long as it is defined as a nested function. In this example the nested function is wrapped in an M-file called `constructroadselector.m`, which returns a complete selector: a handle to the predicate (named `nestedroadfilter`) and the two attibute names:

```
function roadselector = ...
    constructroadselector(minimumClass, minimumLength)
roadselector = {@nestedroadfilter, 'CLASS', 'LENGTH'};
```

```
        function result = nestedroadfilter(roadclass, roadlength)
            result = (roadclass  >= minimumClass) && ...
                     (roadlength >= minimumLength);
        end
    end
```

The following four lines show how to use `constructroadselector`:

```
minimumClass = 4;     % Could be run-time dependent
minimumLength = 200;  % Could be run-time dependent

roadselector = constructroadselector(...
    minimumClass, minimumLength);

s = shaperead('concord_roads', 'Selector', roadselector)

s =
115x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
    LENGTH
```

Understanding Raster Geodata

In this section...

Georeferencing Raster Data

Raster geodata consists of georeferenced data grids and images that in the MATLAB workspace are stored as matrices. While raster geodata looks like any other matrix of real numbers, what sets it apart is that it is georeferenced, either to the globe or to a specified map projection, so that each pixel of data occupies a known patch of territory on the planet.

Whether a raster geodata set covers the entire planet or not, its placement and resolution must be specified. Raster geodata is georeferenced in the toolbox through a companion data structure called a *referencing matrix*. This 3-by-2 matrix of doubles describes the scaling, orientation, and placement of the data grid on the globe. For a given referencing matrix, R, one of the following relations holds between rows and columns and coordinates (depending on whether the grid is based on map coordinates or geographic coordinates, respectively):

```
[x y] = [row col 1] * R, or
[long lat] = [row col 1] * R
```

For additional details about and examples of using referencing matrices, see the reference page for `makerefmat`.

Referencing Vectors

In many instances (when the data grid or image is based on latitude and longitude and is aligned with the geographic graticule), a referencing matrix has more degrees of freedom than the data requires. In such cases, you can use a more compact representation, a three-element *referencing vector*. A referencing vector defines the pixel size and northwest origin for a regular, rectangular data grid:

```
refvec = [cells-per-degree north-lat west-lon]
```

In MAT-files, this variable is often called `refvec` or `maplegend`. The first element, cells-per-degree, describes the angular extent of each grid cell (e.g., if each cell covers five degrees of latitude and longitude, cells-per-degree would be specified as `0.2`). Note that if the latitude extent of cells differs from their longitude extent you cannot use a referencing vector, and instead must specify a referencing matrix. The second element, north-lat, specifies the northern limit of the data grid (as a latitude), and the third element, west-lon, specifies the western extent of the data grid (as a longitude). In other words, north-lat, west-lon is the northwest corner of the data grid. Note, however, that cell (1,1) is always in the southwest corner of the grid. This need not be the case for grids or images described by referencing matrices, as opposed to referencing vectors.

Note Versions of Mapping Toolbox software prior to 2.0 did not use referencing matrices, and called referencing vectors *map legend vectors* or sometimes just *map legend*s. The current version of the toolbox uses the term *legend* only to refer to keys to symbolism.

An example of such a grid is the `geoid` data set (a MAT-file), which represents the shape of the geoid. In the `geoid` matrix, each cell represents one degree, the entire northern edge occupies the north pole, the southern edge occupies the south pole, and the western edge runs down the prime meridian. Thus, the referencing vector for `geoid` is

```
geoidrefvec = [1 90 0]
```

This structure is stored in the `geoid` MAT-file (note that it is duplicated by the `geoidlegend` referencing vector for backward compatibility). Interpret this referencing vector as follows:

- Each data grid entry represents one degree of latitude and one degree of longitude.
- The northern edge of the map is at 90ºN (the North Pole).
- The western edge of the map is at 0º (the prime meridian).

All regular data grids require a referencing matrix or vector, even if they cover the entire planet. Geolocated data grids do not, as they explicitly identify the geographic coordinates of all rows and columns. For details on geolocated grids, see "Geolocated Data Grids" on page 2-44. For additional information on referencing matrices and vectors, see the reference pages for `makerefmat`, `limitm`, and `sizem`.

Regular Data Grids

Regular data grids are rectangular, non-sparse, matrices of class `double`.

Constructing a Global Data Grid

Imagine an extremely coarse map of the world in which each cell represents 60º. Such a map matrix would be 3-by-6.

1 First create data for this, starting with the data grid:

```
miniZ = [1 2 3 4 5 6; 7 8 9 10 11 12; 13 14 15 16 17 18];
```

2 Now make a referencing matrix:

```
miniR = makerefmat('RasterSize', size(miniZ), ...
   'Latlim', [-90 90], 'Lonlim', [-180 180])

miniR =

     0    60
    60     0
  -210  -120
```

3 Set up an equidistant cylindrical map projection:

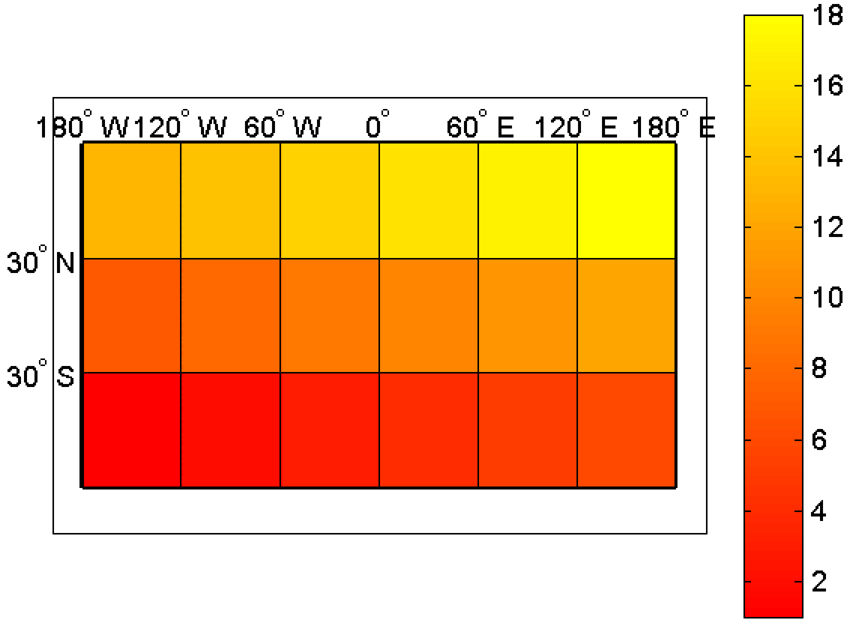
```
axesm('MapProjection', 'eqdcylin')
setm(gca,'GLineStyle','-', 'Grid','on','Frame','on')
```

4 Draw a graticule with parallel and meridian labels at 60º intervals:

```
setm(gca, 'MlabelLocation', 60, 'PlabelLocation',[-30 30],...
    'MLabelParallel','north', 'MeridianLabel','on',...
    'ParallelLabel','on','MlineLocation',60,...
    'PlineLocation',[-30 30])
```

5 Map the data using `meshm` and display with a color ramp and legend:

```
meshm(miniZ, miniR); colormap('autumn'); colorbar
```



Note that the first row of the matrix is displayed as the bottom of the map, while the last row is displayed as the top.

Computing Map Limits from Referencing Vectors

Given a regular data grid and its referencing vector, the full extent of the grid can be computed using the `limitm` function. To understand how this works for a data grid that does not encompass the entire world, do the following exercise:

1 Load the Korea 5-arc-minute elevation grid and inspect the referencing vector, `refvec`:

```
load korea
refvec
refvec =
            12           45           115
```

The `refvec` referencing vector indicates that there are 12 cells per angular degree. This horizontal resolution is 5 times finer than that of the `topo` data grid, which is one cell per degree.

2 Use `limitm` to determine that the `korea` region extends from 30ºN to 45ºN and from 115ºW to 135ºW:

```
[latlimits,longlimits] = limitm(map,refvec)

latlimits =
    30    45
longlimits =
   115   135
```

3 Verify this computation manually by getting the dimensions of the elevation array and computing the eastern and southern map limits from the reference vector:

```
[rows cols] = size(map)

rows =
   180
cols =
   240

southlat = refvec(2) - rows/refvec(1)

southlat =
    30

eastlon = refvec(3) + cols/refvec(1)

eastlon =
   135
```

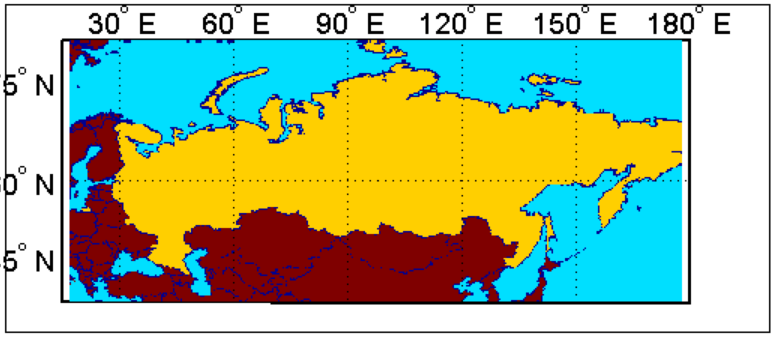
The results match `latlimits(1)` and `longlimits(2)`. The two formulas use different signs because latitudes decrease southwards and longitudes increase eastward.

Geographic Interpretation of Matrix Elements

You can access and manipulate gridded geodata and its associated referencing vector by either geographic or matrix coordinates. Use the `russia` data set to explore this. As was demonstrated above, the north, south, east, and west limits of the mapped area can be determined as follows:

```
clear; load russia
[latlim,longlim] = limitm(map,refvec)

latlim =
 35    80
longlim =
 15   190
```

The data grid in the `russia` MAT-file extends over the international date line (180º longitude). You could use the function `wrapTo180` to rename the eastern limit to be -170, or 170ºW.



The function `setltln` retrieves the geographic coordinates of a particular matrix element. The returned coordinates actually show the center of the geographic area represented by the matrix entry:

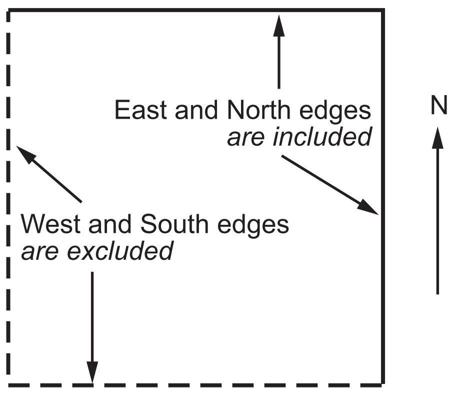
```
row = 23; col = 79;
[lat,long] = setltln(map,refvec,row,col)

lat =
 39.5
long =
  30.7
```

`setpostn` does the reverse of this, determining the row and column of the data grid element containing a given geographic point location:

```
[r,c] = setpostn(map,maplegend,lat,long)

r =
 23
c =
 79
```

The Geography of Gridded Geodata

Each matrix element (analogous to a pixel) can be thought of as a spheroidal *quadrangle*, which includes its northern and eastern edges, but not its western edge or southern edge.



An Element in a Data Grid

The exceptions to this are that the southernmost row (row 1) also contains its southern edge, and the westernmost column (column 1) contains its western edge, except when the map encompasses the entire 360º of longitude. In that case, the westernmost edge of the first column is not included, because it is identical to the easternmost edge of the last column. These exceptions ensure that all points on the globe can be represented exactly once in a regular data grid.

Although each data grid element represents an area, not a point, it is often useful to assign singular coordinates to provide a point of reference. The `setltln` function does this. It geolocates an element by the point in the center of the area represented by the element. The following code references the center cell coordinate for the row 3, column 17 of the Russia map:

```
clear; load russia
row = 3; col = 17;
[lat,long] = setltln(map,refvec,row,col)

lat =
 35.5
long =
 18.3
```

Because the cells in the `russia` matrix represent 0.2º squares (5 cells per degree), the cell in question extends from north of 35.4ºS to exactly 35.6ºS, and from east of 18.2ºE to exactly 18.4ºE.

Accessing Data Grid Elements

The actual values contained within the map matrix entries are important as well. Several Mapping Toolbox functions can access and alter the values of data grid elements.

If the actual row and column of a desired entry are known, then a simple matrix index can return the appropriate value:

1 Use the row and column from the previous example (row 3, column 17) to determine the value of that cell simply by querying the matrix:

   ```
   value = map(row,col)

   value =
    2
   ```

2 More often, the geographic coordinates are known, and the value can be retrieved with `ltln2val`:

   ```
   value = ltln2val(map,maplegend,lat,long)
   ```

```
value =
 2
```

3 The latitude-longitude coordinates associated with particular values in a data grid can be found with `findm`, analogous to the MATLAB function `find`. Here the coordinates of elements in the `topo` matrix have values greater than 5,500 meters:

```
load topo
[lats,longs] = findm(topo>5500,topolegend);
[lats longs]

ans =
 34.5000   79.5000
 34.5000   80.5000
 30.5000   84.5000
 28.5000   86.5000
```

4 To get the row and column indices instead, simply use the Mapping Toolbox function `find`:

```
[i,j]=find(topo>5500)

i =
   125
   125
   121
   119
j =
    80
    81
    85
    87
```

5 To recode a specific matrix value to some other value, use `changem`. Load or reload the `russia` MAT-file, and then change all instances of a given value in a data grid to a new value in one step:

```
oldcode = ltln2val(map,maplegend,37,79)

oldcode =
    4

   newmap = changem(map,5,oldcode);
   newcode = ltln2val(newmap,maplegend,37,79)

   newcode =
    5
```

All entries in `newmap` corresponding to 4s in `map` now have the value 5.

Using a Mask to Recode a Data Grid

You can also define a logical mask to identify the map entries to change. A mask is a matrix the same size as the map matrix, with 1s everywhere that values are to change. A mask is often generated by a logical operation on a map variable, a topic that is described in greater detail below:

1 The `russia` data grid contains `3` for each cell covering Russia. To set every non-Russia matrix entry to zero, use the following commands:

```
clear; load russia
nonrussia = map;
nonrussia(map~=3) = 0;
```

2 Verify the data that results from these operations:

```
whos
  Name              Size                Bytes  Class
  clrmap            4x3                    96  double
  description       5x69                  690  char
  map             225x875             1575000  double
  maplegend         1x3                    24  double
  nonrussia       225x875             1575000  double
  refvec            1x3                    24  double
  source            1x68                  136  char

newcode = ltln2val(nonrussia,refvec,37,79)

newcode =
 0
```

Precomputing the Size of a Data Grid

Finally, if you know the latitude and longitude limits of a region, you can calculate the required matrix size and an appropriate referencing vector for any desired map resolution and scale. However, before making a large, memory-taxing data grid, you should first determine what its size will be. For a map of the continental U.S. at a scale of 10 cells per degree, do the following:

1 Compute the matrix dimensions using `sizem`, specifying latitude limits of 25ºN to 50ºN and longitudes from 60ºW to 130ºW:

```
cellsperdeg = 10;
[r,c,maplegend] = sizem([25 50],[-130 -60],cellsperdeg)

r =
   250
c =
   700
maplegend =
    10    50  -130

msize = r * c * 8

msize =
     1400000
```

This data grid would be 250-by-700, and consume 1,400,000 bytes.

2 Now determine what the storage requirements would be if the scale were reduced to 5 rows/columns per degree:

```
cellsperdeg2 = 5;
[r,c,maplegend] = sizem([25 50],[-130 -60],cellsperdeg2)

r =
   125
c =
   350
maplegend =
     5    50  -130

msize = r * c * 8
msize =
      350000
```

A 125-by-300 matrix that used 350,000 bytes might be more manageable, if it had sufficient resolution at its intended publication scale.

Geolocated Data Grids

In addition to regular data grids, the toolbox provides another format for geodata: *geolocated data grids*. These multivariate data sets can be displayed, and their values and coordinates can be queried, but unfortunately much of the functionality supporting regular data grids is not available for geolocated data grids.

The examples thus far have shown maps that covered simple, regular quadrangles, that is, geographically rectangular and aligned with parallels and meridians. Geolocated data grids, in addition to these rectangular orientations, can have other shapes as well.

Geolocated Grid Format

To define a geolocated data grid, you must define three variables:

- A matrix of indices or values associated with the mapped region
- A matrix giving cell-by-cell latitude coordinates
- A matrix giving cell-by-cell longitude coordinates

The following exercise demonstrates this data representation:

1 Load the MAT-file example of an irregularly shaped geolocated data grid called `mapmtx`:

```
load mapmtx
whos
  Name               Size             Bytes  Class     Attributes

  description        1x54               108  char
  lg1               50x50             20000  double
  lg2               50x50             20000  double
```

```
    lt1                50x50              20000  double
    lt2                50x50              20000  double
    map1               50x50              20000  double
    map2               50x50              20000  double
    source              1x43                 86  char
```

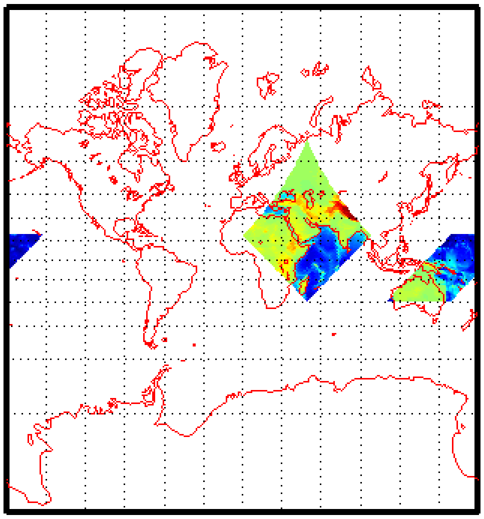
Two geolocated data grids are in this data set, each requiring three variables. The values contained in `map1` correspond to the latitude and longitude coordinates, respectively, in `lt1` and `lg1`. Notice that all three matrices are the same size. Similarly, `map2`, `lt2`, and `lg2` together form a second geolocated data grid. These data sets were extracted from the `topo` data grid shown in previous examples. Neither of these maps is regular, because their columns do not run north to south.

2 To see their geography, display the grids one after another:

```
close all
axesm mercator
gridm on
framem on
h1=surfm(lt1,lg1,map1);
h2=surfm(lt2,lg2,map2);
```

3 Showing coastlines will help to orient you to these skewed grids:

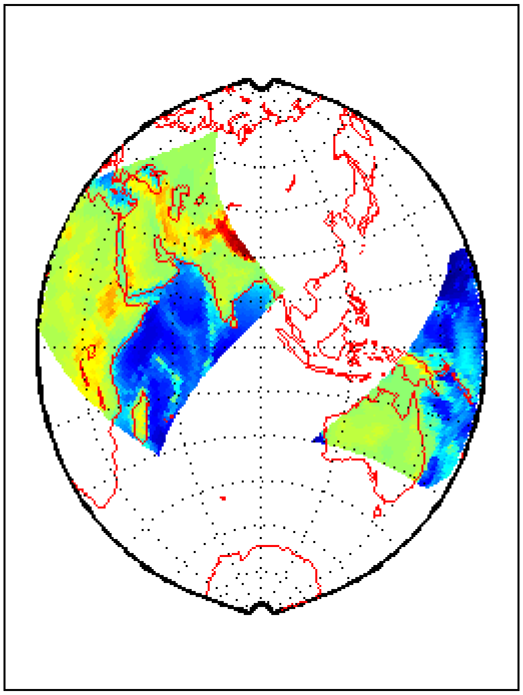
```
load coast
plotm(lat,long,'r')
```



Notice that neither `topo` matrix is a regular rectangle. One looks like a diamond geographically, the other like a trapezoid. The trapezoid is displayed in two pieces because it crosses the edge of the map. These shapes can be thought of as the geographic organization of the data, just as rectangles are for regular data grids. But, just as for regular data grids, this organizational logic does not mean that displays of these maps are necessarily a specific shape.

4 Now change the view to a polyconic projection with an origin at 0ºN, 90ºE:

```
setm(gca,'MapProjection','polycon', 'Origin',[0 90 0])
```



As the polyconic projection is limited to a 150° range in longitude, those portions of the maps outside this region are automatically trimmed.

Geographic Interpretations of Geolocated Grids

Mapping Toolbox software supports three different interpretations of geolocated data grids:

- First, a map matrix having the same number of rows and columns as the latitude and longitude coordinate matrices represents the values of the map data at the corresponding geographic points (centers of data cells).
- Next, a map matrix having one fewer row and one fewer column than the geographic coordinate matrices represents the values of the map data within the area formed by the four adjacent latitudes and longitudes.

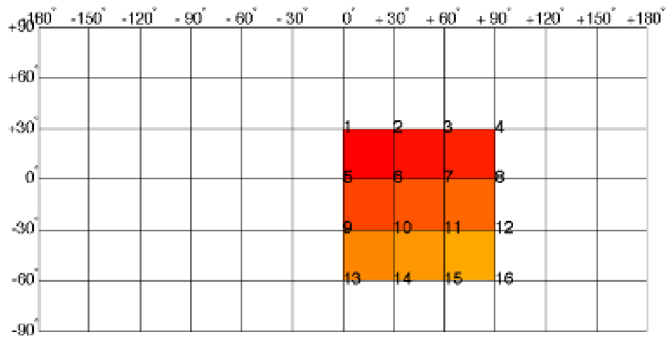
- Finally, if the latitude and longitude matrices have smaller dimensions than the map matrix, you can interpret them as describing a coarser *graticule*, or mesh of latitude and longitude cells, into which the blocks of map data are warped.

This section discusses the first two interpretations of geolocated data grids. For more information on the use of graticules, see “The Map Grid” on page 4-53.

Type 1: Values associated with upper left grid coordinate. As an example of the first interpretation, consider a 4-by-4 map matrix whose cell size is 30-by-30 degrees, along with its corresponding 4-by-4 latitude and longitude matrices:

```
map = [ 1  2  3  4;...
 5  6  7  8;...
 9 10 11 12;...
 3 14 15 16];

lat = [ 30  30  30  30;...
 0   0   0   0;...
 -30 -30 -30 -30;...
 -60 -60 -60 -60];

long = [0 30 60 90;...
 0 30 60 90;...
 0 30 60 90;...
 0 30 60 90];
```

This geolocated data grid is displayed with the values of `map` shown at the associated latitudes and longitudes.

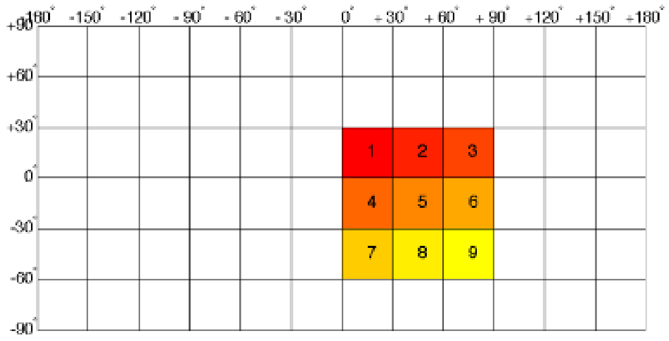


Notice that only 9 of the 16 total cells are displayed. The value displayed for each cell is the value at the upper left corner of that cell, whose coordinates are given by the corresponding `lat` and `long` elements. By convention, the last row and column of the map matrix are not displayed, although they exist in the `CData` property of the surface object.

Type 2: Values centered within four adjacent coordinates. For the second interpretation, consider a 3-by-3 map matrix with the same `lat` and `long` variables:

```
map = [1 2 3;...
   4 5 6;...
   7 8 9];
```

Here is a surface plot of the map matrix, with the values of `map` shown at the center of the associated cells:



All the map data is displayed for this geolocated data grid. The value of each cell is the value at the center of the cell, and the latitudes and longitudes in the coordinate matrices are the boundaries for the cells.

Ordering of Cells. You may have noticed that the first row of the matrix is displayed as the top of the map, whereas for a regular data grid, the opposite was true: the first row corresponded to the bottom of the map. This difference is entirely due to how the `lat` and `long` matrices are ordered. In a geolocated data grid, the order of values in the two coordinate matrices determines the arrangement of the displayed values.

Transforming Regular to Geolocated Grids. When required, a regular data grid can be transformed into a geolocated data grid. This simply requires that a pair of coordinates matrices be computed at the desired spatial resolution from the regular grid. Do this with the `meshgrat` function, as follows:

```
load topo
[lat,lon] = meshgrat(topo,topolegend);
  Name                 Size                 Bytes  Class     Attributes

  lat                180x360               518400  double
  lon                180x360               518400  double
  topo               180x360               518400  double
  topolegend           1x3                     24  double
  topomap1            64x3                   1536  double
  topomap2           128x3                   3072  double
```

Transforming Geolocated to Regular Grids. Conversely, a regular data grid can also be constructed from a geolocated data grid. The coordinates and values can be embedded in a new regular data grid. The function that performs this conversion is `geoloc2grid`; it takes a geolocated data grid and a cell size as inputs.

Reading and Writing Geospatial Data

In this section...

Functions that Read and Write Geospatial Data

Many vector and raster data formats have been developed for storing geospatial data in computer files. Some formats are widely used, others are obscure; some are simple, while others are elaborate. Some formats are government or international standards, others are simply popular. A format can be general-purpose, specific to a narrow class of data, or may be used just to publish a certain data set.

Using Mapping Toolbox functions, you can read geodata files in generic exchange formats (e.g., SDTS, shapefiles, and GeoTIFF files) that a variety of mapping and image processing applications can also read and write. You can also read files that are in special formats designed to exchange specific sets of geodata (e.g., AVHRR, GSHHS, DCW, DEM, and DTED files). You can order, and in some cases download, such data over the Internet from public agencies and private distributors.

In addition, the toolbox provides generalized sample data in the form of data files for the entire Earth and its major regions, as well as some more detailed demo geodata files covering small areas. These data sets, which are located in *`matlabroot`*`/toolbox/map/mapdemos`, are used in most of the code examples provided in this documentation. Many of the sample data sets are described in text files also located in that directory.

If you need to locate geospatial data in particular formats, or for specific themes or regions, you can consult the following MathWorks Tech Note 2101, which is regularly updated.
`http://www.mathworks.com/support/tech-notes/2100/2101.html`

The following table lists Mapping Toolbox functions that read geospatial data products and file formats and write geospatial data files. Note that the `geoshow` and `mapshow` functions and the `mapview` GUI can read and display both vector and raster geodata files in several formats. Click function names to see their details in the Mapping Toolbox reference documentation. The **Type of Coordinates** column describes whether the function returns or writes data in geographic ("geo") or projected ("map") coordinates, or as geolocated data grids (which, for the functions listed, all contain geographic coordinates). Some functions can return either geographic or map coordinates, depending on what the file being read contains; these functions do not signify what type of coordinates they return (in the case of `shaperead`, however, you can specify whether the structure it returns should have `X` and `Y` or `Lon` and `Lat` fields).

Function	Description	Type of Data	Type of Coordinates
`arcgridread`	Read a gridded data set in Arc ASCII Grid Format	raster	map
`avhrrgoode`	Read data products derived from the Advanced Very High Resolution Radiometer (AVHRR) and stored in the Goode Homosoline projection: Global Land Cover Classification (GLCC) or Normalized Difference Vegetation Index (NDVI)	raster	geolocated
`avhrrlambert`	Read AVHRR GLCC and NDVI data products stored in the Lambert Conformal Conic projection	raster	geolocated
`dcwdata`	Read selected data from the Digital Chart of the World (DCW)	vector	geo
`dcwgaz`	Search for entries in the DCW gazette	vector	geo
`dcwread`	Read a DCW file	vector	geo
`dcwrhead`	Read a DCW file header	properties	geo
`demdataui`	GUI for interactively selecting data from various Digital Elevation Models (DEMs)	raster	geo

Function	Description	Type of Data	Type of Coordinates
`dted`	Read U. S. Dept. of Defense Digital Terrain Elevation Data (DTED)	raster	geo
`dteds`	List DTED data grid filenames for a specified latitude-longitude quadrangle	filenames	geo
`egm96geoid`	Read 15-minute gridded geoid heights from the EGM96 geoid model	raster	geo
`etopo`	Read data from ETOPO2 or ETOPO5 (2-minute or 5-minute) gridded global terrain relief data sets	raster	geo
`fipsname`	Read Federal Image Processing Standards (FIPS) names for Topographically Integrated Geographic Encoding and Referencing (TIGER) thinned boundary files	FIPS names and identifiers	geo
`geotiffinfo`	Information about a GeoTIFF file	properties	map geo
`geotiffread`	Read a georeferenced image from GeoTIFF file	image	map
`getworldfilename`	Derive a worldfile name from an image filename	filename	geo map
`globedem`	Read Global Land One-km Base Elevation (GLOBE) 30-arc-second (1 km) Digital Elevation Model	raster	geo
`globedems`	List GLOBE data filenames for a specified latitude-longitude quadrangle	filenames	geo
`gshhs`	Read Global Self-Consistent Hierarchical High-Resolution Shoreline (GSHHS) data	vector	geo

Function	Description	Type of Data	Type of Coordinates
gtopo30	Read GTOPO30 30-arc-second (1 km) global elevation data	raster	geo
gtopo30s	List GTOPO30 data filenames for a specified latitude-longitude quadrangle	filenames	geo
kmlwrite	Write vector coordinates and attributes to a file in KML format	vector points and attributes	geo
readfk5	Read data from the Fifth Fundamental Catalog of Stars	vector	astro
satbath	Read 2-minute (4 km) global topography sea floor derived by Smith and Sandwell from ship soundings and satellite bathymetry	raster	geolocated
sdtsdemread	Read U.S. Geological Survey (USGS) digital elevation model (DEM) stored in SDTS (Spatial Data Transfer Standard) format (Raster Profile)	raster	geo map
sdtsinfo	Information about SDTS data set	properties	geo
shapeinfo	Information about the geometry and attributes of geographic features stored in a shapefile (a set of ".shp", ".shx" and ".dbf" files)	properties	geo map
shaperead	Read geographic feature coordinates and associated attributes from a shapefile	vector	geo map
shapewrite	Write geospatial data and associated attributes in shapefile format	vector	geo map

Function	Description	Type of Data	Type of Coordinates
`tbase`	Read data from the 5-minute TerrainBase global digital terrain model	raster	geo
`usgs24kdem`	Read USGS 1:24,000 (30 m or 10 m) digital elevation models	raster	geolocated
`usgsdem`	Read USGS 1:250,000 (100 m) digital elevation models	raster	map
`usgsdems`	List USGS digital elevation model (DEM) filenames covering a specified latitude-longitude quadrangle	filenames	map
`vmap0data`	Extract selected data from the Vector Map Level 0 (VMAP0) CD-ROMs	vector	geo
`vmap0read`	Read a VMAP0 file	vector	geo
`vmap0rhead`	Read VMAP0 file headers	properties	geo
`vmap0ui`	Activate GUI for interactively selecting VMAP0 data	vector	geo
`worldfileread`	Read a worldfile and return a referencing matrix	georeferencing information	geo
`worldfilewrite`	Export a referencing matrix into an equivalent worldfile	georeferencing information	geo

The MATLAB environment provides many general file reading and writing functions (for example, `imread`, `imwrite`, `urlread`, and `urlwrite`) which you can use to access geospatial data you want to use with Mapping Toolbox software. For example, you can read a TIFF image with `imread` and its accompanying worldfile with `worldfileread` to import the image and construct a referencing matrix to georeference it. See the Mapping Toolbox demos “Creating a Half-Resolution Georeferenced Image” and “Georeferencing an Image to an Orthotile Base Layer” for examples of how you can do this.

Exporting Vector Geodata

When you want to share geodata you are working with, Mapping Toolbox functions can export it two principal formats, shapefiles and KML files. Shapefiles are binary files that can contain point, line, vector, and polygon data plus attributes. Shapefiles are widely used to exchange data between different geographic information systems. KML files are text files that can contain the same type of data, and are used mainly to upload geodata the Web. The toolbox functions `shapewrite` and `kmlwrite` export to these formats.

To format attributes, `shapewrite` uses an auxiliary structure called a *DBF spec*, which you can generate with the `makedbfspec` function. Similarly, you can provide attributes to `kmlwrite` to format as a table by providing an *attribute spec*, a structure you can generate using the `makeattribspec` function or create manually.

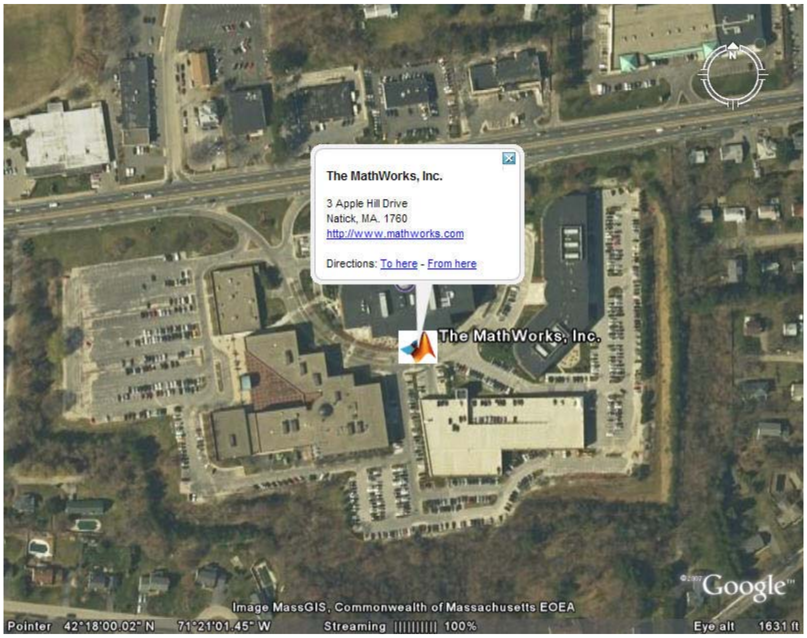
For examples of and additional information about reading and writing shapefiles and DBF specs, see the documentation for `shapeinfo`, `shaperead`, `shapewrite`, and `makedbfspec`. The example provided in "How to Construct Geostructs and Mapstructs" on page 2-20 also demonstrates exporting vector data using `shapewrite`. For information about creating KML files, see the following section.

Exporting KML Files for Viewing in Earth Browsers

Keyhole Markup Language (KML) is an XML dialect for formatting 2-D and 3-D geodata for display in "Earth browsers," such as Google™ Earth mapping service, Google Maps mapping service, Google Mobile wireless service, and NASA WorldWind. Other Web browser applications, such as Yahoo!® Pipes, also support KML either by rendering or generating files. A KML file specifies a set of features (placemarks, images, polygons, 3-D models, textual descriptions, etc.) and how they are to be displayed in browsers and applications.

Each place must at least have an address or a longitude and a latitude. Places can also have textual descriptions, including hyperlinks. KML files can also specify display styles for markers, lines and polygons, and "camera view" parameters such as tilt, heading, and altitude. You can generate placemarks in KML files for individual points and sets of points that include attributes in table form. You can include HTML markups in these tables, with or without hyperlinks, but you cannot currently control the camera view of a placemark. (However, the users of an Earth browser can generally control their views of it).

Generating A Single Placemark. Here is a placemark produced by `kmlwrite` that locates the headquarters of The MathWorks, as displayed in the Google Earth application.


The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA. 1760
http://www.mathworks.com
Directions: To here - From here
The MathWorks, Inc.
Google
Image MassGIS, Commonwealth of Massachusetts EOEA
Pointer 42°18'00.02" N 71°21'01.45" W
Streaming |||||||||| 100%
Eye alt 1631 ft

The location, text, and icon for the placemark were specified to `kmlwrite` as follows:

```
lat =  42.299827;
lon = -71.350273;
description = sprintf('%s<br>%s</b><br>%s</b>', ...
    '3 Apple Hill Drive', 'Natick, MA. 01760', ...
    'http://www.mathworks.com');
name = 'The MathWorks, Inc.';
iconFilename = ...
    'http://www.mathworks.com/products/product_listing/images/ml_icon.gif';
iconScale = 1.0;
```

```
filename = 'The_MathWorks.kml';
kmlwrite(filename, lat, lon, ...
    'Description', description, 'Name', name, ...
    'Icon', iconFilename, 'IconScale', iconScale);
```

The file produced by `kmlwrite` looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://earth.google.com/kml/2.1">
   <Document>
      <name>The_MathWorks</name>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description>3 Apple Hill Drive&amp;lt;br&gt;Natick, MA. 01760&amp;lt;/b&gt;
                     &amp;lt;br&gt;http://www.mathworks.com&amp;lt;/b&gt;
         </description>
         <name>The MathWorks, Inc.</name>
         <Style>
            <IconStyle>
               <Icon>
                 <href>
                   http://www.mathworks.com/products/product_listing/images/ml_icon.gif
                 </href>
               </Icon>
               <scale>1</scale>
            </IconStyle>
         </Style>
         <Point>
            <coordinates>-71.350273,42.299827,0.0</coordinates>
         </Point>
      </Placemark>
   </Document>
</kml>
```

If you view this in an Earth Browser, notice that the text inside the placemark, “http://www.mathworks.com,” was automatically rendered as a hyperlink. The Google Earth service also adds a link called “Directions”. `kmlwrite` does not include location coordinates in placemarks. This is because it is easy for users to read out where a placemark is by mousing over it or by viewing its Properties dialog box.

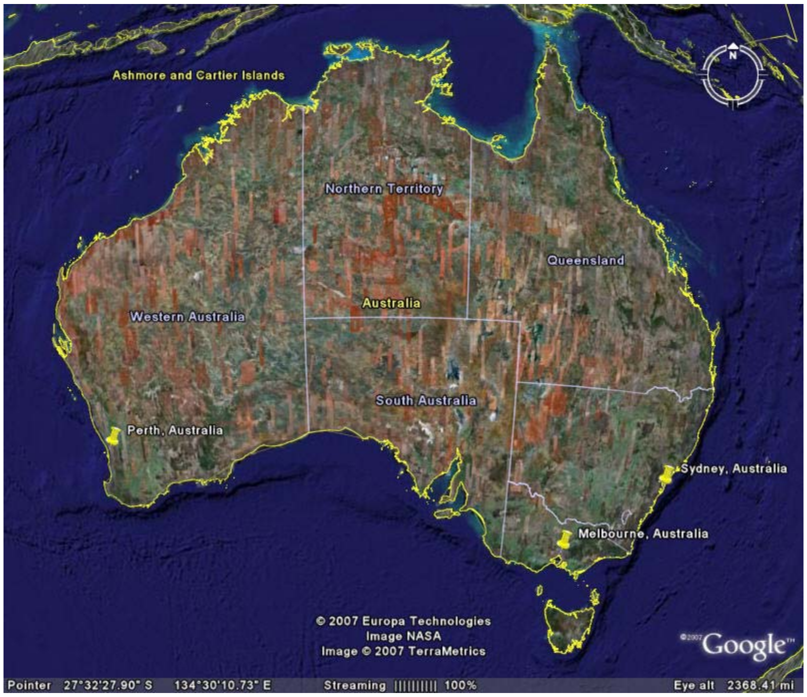
Placemarks from Addresses. You do not need coordinates in order to geolocate placemarks; instead, you can specify street addresses or more general addresses such as postal codes, city, state, or country names in a KML file. (Note that the Google Maps service does not support address-based placemarks.) If the viewing application is capable of looking up addresses, such placemarks can be displayed in appropriate, although possibly imprecise, locations. When you use addresses, `kmlwrite` creates an `<address>` element for each placemark rather than `<point>` elements containing `<coordinates>` elements. For example, here is code for `kmlwrite` that generates address-based placemarks for three cities in Australia from a cell array:

```
address = {'Perth, Australia', ...
           'Melbourne, Australia', ...
           'Sydney, Australia'};
filename = 'Australian_Cities.kml';
kmlwrite(filename, address, 'Name', address);
```

The generated KML file has the following structure and content:

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://earth.google.com/kml/2.1">
   <Document>
      <name>Australian_Cities</name>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Perth, Australia</name>
         <address>Perth, Australia</address>
      </Placemark>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Melbourne, Australia</name>
         <address>Melbourne, Australia</address>
      </Placemark>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Sydney, Australia</name>
         <address>Sydney, Australia</address>
      </Placemark>
    </Document>
  </kml>
```

The placemarks display in a Google Earth map like this, with default placemark icons.



Exporting Point Geostructs to Placemarks. This example shows how to selectively read data from shapefiles and generate a KML file that identifies all or selected attributes, which you can then view in an earth browser such as Google Earth. It also shows how to customize placemark icons and vary them according to attribute values.

The Mapping Toolbox tsunamis demo shapefiles contain a database of 162 tsunami (tidal wave) events reported between 1950 and 2006, described as point locations with 21 variables (including 18 attributes). You can type out the metadata file `tsunamis.txt` to see the definitions of all the data fields.

The steps below select some of these from the shapefiles and display them as tables in exported KML placemarks.

1 Read the tsunami shapefiles, selecting certain attributes.

   There are several ways to select attributes from shapefiles. One is to pass `shaperead` a cell array of attribute names in the Attributes parameter. For example, you might just want to map the maximum wave height, the suspected cause, and also show the year, location and country for each event. Set up a cell array with the corresponding attribute field names as follows, remembering that field names are case-sensitive.

```
attrs = {'Max_Height','Cause','Year','Location','Country'};
```

   Since the data file uses latitude and longitude coordinates, you need to specify `'UseGeoCoords',true` to ensure that `shaperead` returns a geostruct (having `Lat` and `Lon` fields).

```
tsunamis = shaperead('tsunamis.shp','useGeoCoords',true,...
                     'Attributes',attrs);
```

   Look at the first record in the `tsunamis` geostruct returned by `shaperead`.

```
tsunamis(1)

          Geometry: 'Point'
          Lon: 128.3000
          Lat: -3.8000
    Max_Height: 2.8000
         Cause: 'Earthquake'
          Year: 1950
      Location: 'JAVA TRENCH, INDONESIA'
       Country: 'INDONESIA'
```

2 Output the tsunami data to a KML file with `kmlwrite`

   By default, `kmlwrite` outputs all attribute data in a geostruct to a KML formatted file as an HTML table containing unstyled text. When you view it, the Google Earth program supplies a default marker.

```
kmlfilename = 'tsunami1.kml';
```

```
kmlwrite(kmlfilename,tsunamis);
```

3 View the placemarks in an earth browser

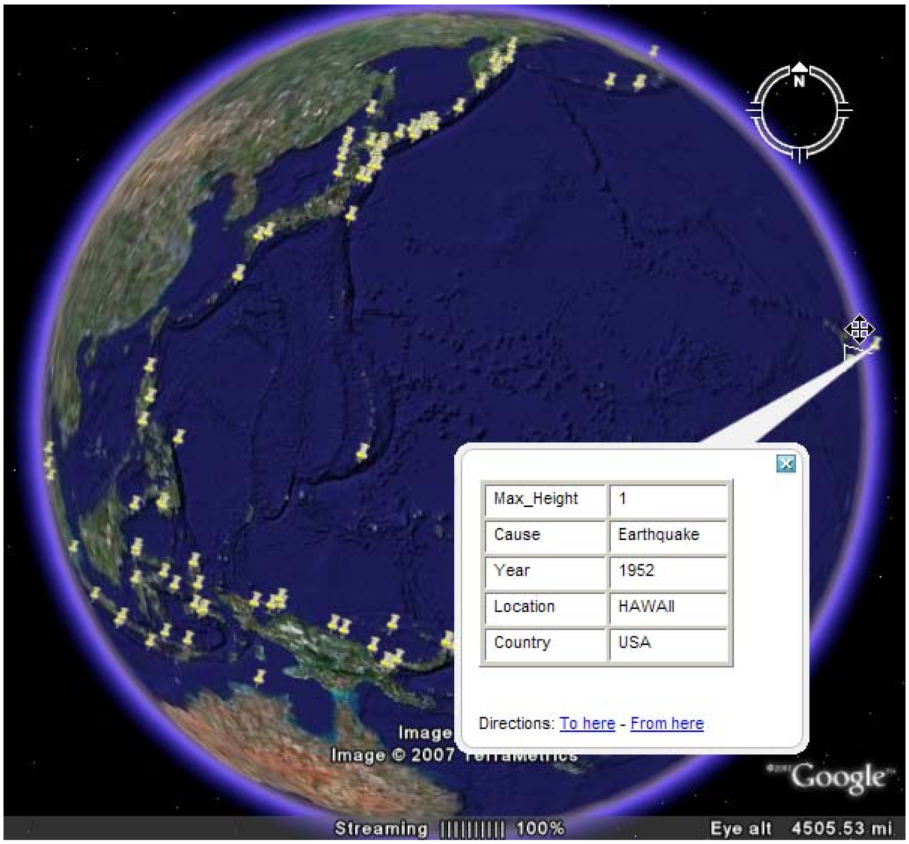
On Windows®, use `winopen` to open Google Earth (which must be installed) to view the KML file.

```
winopen(kmlfilename)
```

On Macintosh® or Linux® platforms, use the `system` command to launch Google Earth.

```
cmd = 'googleearth ';
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

Rotate to the Western Pacific ocean and zoom to inspect the placemarks. Click on the pushpin icons to see the attribute table for any event. `kmlwrite` formats tables by default to display all the attributes in the geostruct passed to it.



Max_Height	1
Cause	Earthquake
Year	1952
Location	HAWAII
Country	USA

Directions: To here - From here

4 Customize the placemark contents

To customize the HTML table in the placemark, use the `makeattribspec` function. Create an attribute spec for the `tsunamis` geostruct and inspect it.

```
attribspec = makeattribspec(tsunamis)

attribspec =
    Max_Height: [1x1 struct]
         Cause: [1x1 struct]
          Year: [1x1 struct]
      Location: [1x1 struct]
       Country: [1x1 struct]
```

Format the label for Max_Height as bold text, give units information about Max_Height, and also set the other attribute labels in bold.

```
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>';
```

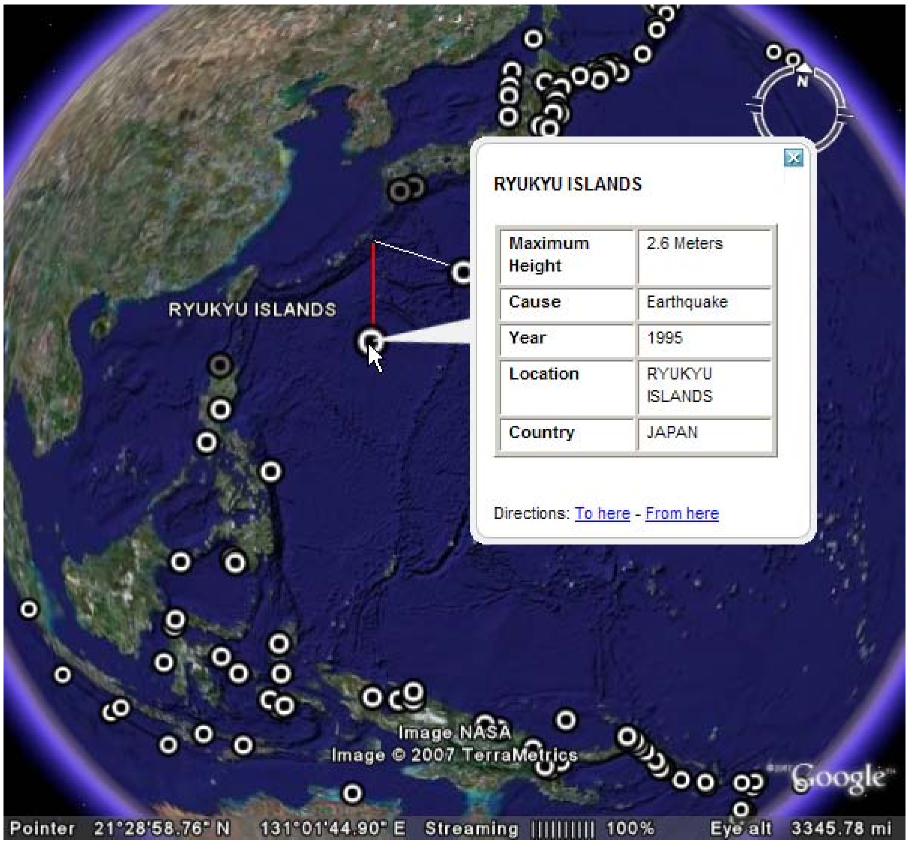
When you use the attribute spec, all the attributes it lists are included in the placemarks generated by kmlwrite unless you remove them from the spec manually (e.g., with rmfield).

5 Customize the placemark icon

You can specify your own icon using kmlwrite to use instead of the default pushpin symbol. The black-and-white bullseye icon used here is specified as URL for an icon in the Google KML library.

```
iconname = ...
  'http://maps.google.com/mapfiles/kml/shapes/placemark_circle.png';
kmlwrite(kmlfilename,tsunamis,'Description',attribspec,...
  'Name',{tsunamis.Location},'Icon',iconname,'IconScale',2);
```

Refresh the earth browser to display the new version of the KML file.


RYUKYU ISLANDS
Image NASA
Image © 2007 TerraMetrics
Google
Pointer 21°28'58.76" N 131°01'44.90" E Streaming |||||||||| 100% Eye alt 3345.78 mi

RYUKYU ISLANDS

Maximum Height	2.6 Meters
Cause	Earthquake
Year	1995
Location	RYUKYU ISLANDS
Country	JAPAN

Directions: To here - From here

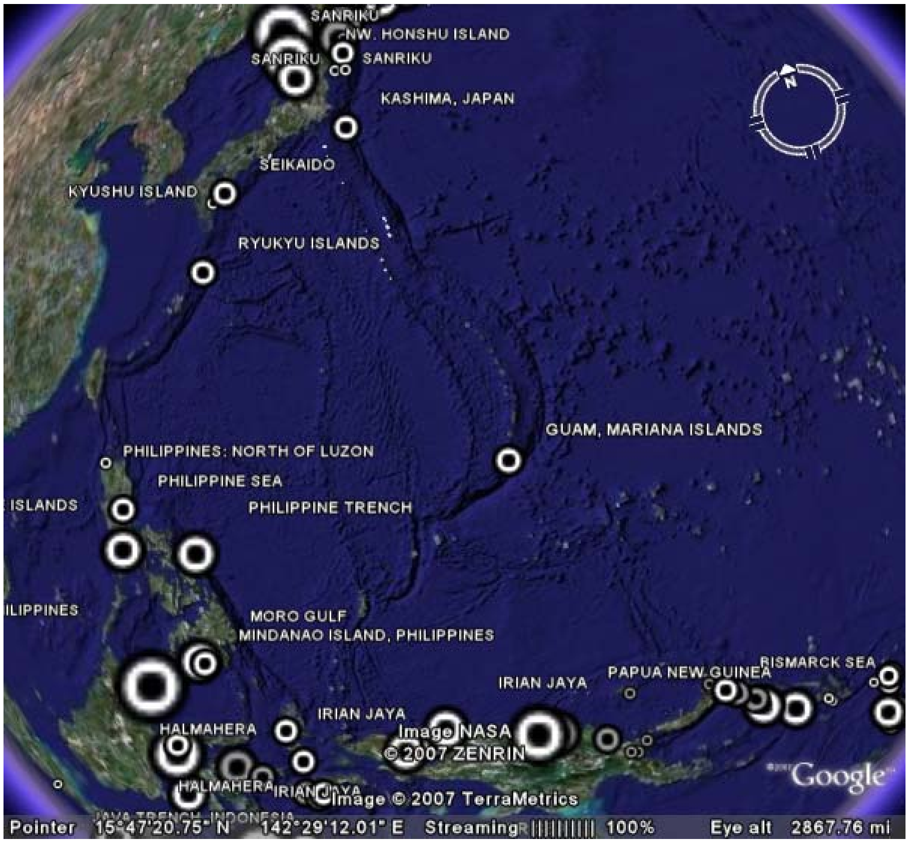
6 Vary placemark size by tsunami height

To vary the size of placemark icons, specify an icon file and a scaling factor for every observation as vectors of names (all the same) and scale factors (all computed individually) when writing a KML file. Scale the width and height of the markers to the log of `Max_Height`. Scaling factors for point icons are data-dependent and can take some experimenting with to get right.

```
% Create vector with log2 exponents of |Max_Height| values
[loghgtx loghgte] = log2([tsunamis.Max_Height]);
% Create a vector replicating the icon URL
iconnames = cellstr(repmat(iconname,numel(tsunamis),1));
```

```
kmlwrite(kmlfilename,tsunamis,'Description',attribspec,...
    'Name',{tsunamis.Location},'Icon',iconname,...
    'IconScale',loghgte);
```

Refresh the earth browser to display the new version of the KML file. Position the viewpoint to compare with the previous view of the Pacific region. Diameters of placemarks now correspond to `log(Max_Height)`.



Functions That Read and Write Files in Compressed Formats

Geospatial data, like other files, are frequently stored and transmitted in compressed or archive formats, such a tar, zip, or GNU zip. Several MATLAB functions read or write such files. All create files in a directory for which you must have write permission. Input files can exist on your host computer, reside on a local area network, or be located on the Internet (in which case they are identified using URLs).

The following table identifies MATLAB functions that you can use to read, uncompress, compress, and write archived data files, geospatial or otherwise. Click any link to read the function's documentation.

Function	**Purpose**
`gunzip`	Uncompress files in the GNU zip format
`untar`	Extract the contents of a tar file
`unzip`	Extract the contents of a zip file
`gzip`	Compress files into the GNU zip format
`tar`	Compress files into a tar file
`zip`	Compress files into a zip file

Use the functions `gunzip`, `untar`, and `unzip` to read data files specified with a URL or with path syntax. Use the functions `gzip`, `tar`, and `zip` to create your own compressed files and archives. This capability is useful, for example, for packaging a set of shapefiles, or a worldfile along with the data grid or image it describes, for distribution.

Understanding Geospatial Geometry

See Chapter 2, "Understanding Map Data" for information on how geographic phenomena are encoded and represented numerically, and how geodata is structured.

Understanding Spherical Coordinates

In this section...

Spheres, Spheroids, and Geoids

Working with geospatial data involves geographic concepts (e.g., geographic and plane coordinates, spherical geometry) and geodetic concepts (such as ellipsoids and datums). This group of sections explain, at a high level, some of the concepts that underlie geometric computations on spherical surfaces.

Although the Earth is very round, it is an oblate *spheroid* rather than a perfect sphere. This difference is so small (only one part in 300) that modeling the Earth as spherical is sufficient for making small-scale (world or continental) maps. However, making accurate maps at larger scale demands that a spheroidal model be used. Such models are essential, for example, when you are mapping high-resolution satellite or aerial imagery, or when you are working with coordinates from the Global Positioning System (GPS). This section addresses how Mapping Toolbox software accurately models the shape, or figure, of the Earth and other planets.

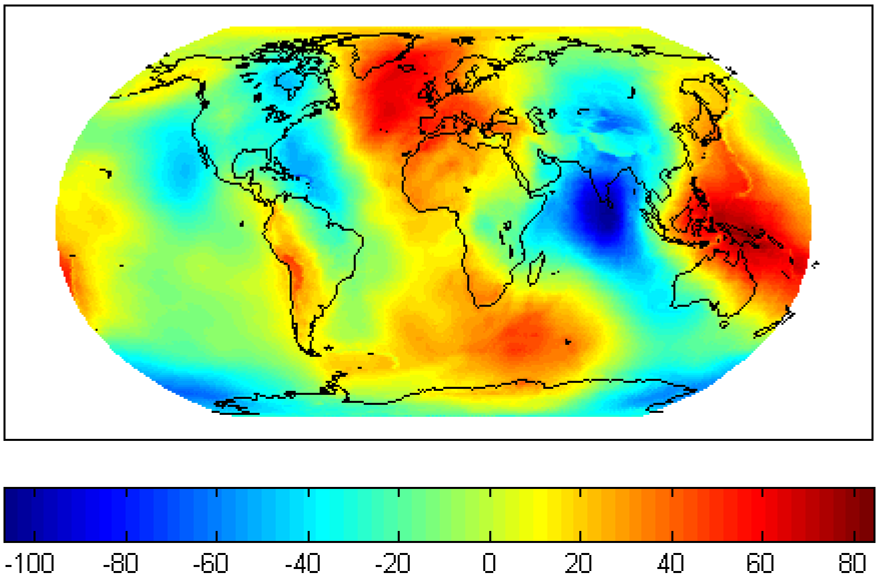
Geoid and Ellipsoid

Literally, *geoid* means *Earth-shaped*. The geoid is an empirical approximation of the figure of the Earth (minus topographic relief), its "lumpiness.." Specifically, it is an equipotential surface with respect to gravity, more or less corresponding to mean sea level. It is approximately an oblate ellipsoid, but not exactly so because local variations in gravity create minor hills and dales (which range from -100 m to +60 m across the Earth). This variation in height is on the order of one percent of the differences between the semimajor and semiminor ellipsoid axes used to approximate the Earth's shape, as described in "The Ellipsoid Vector" on page 3-4.

Mapping the Geoid

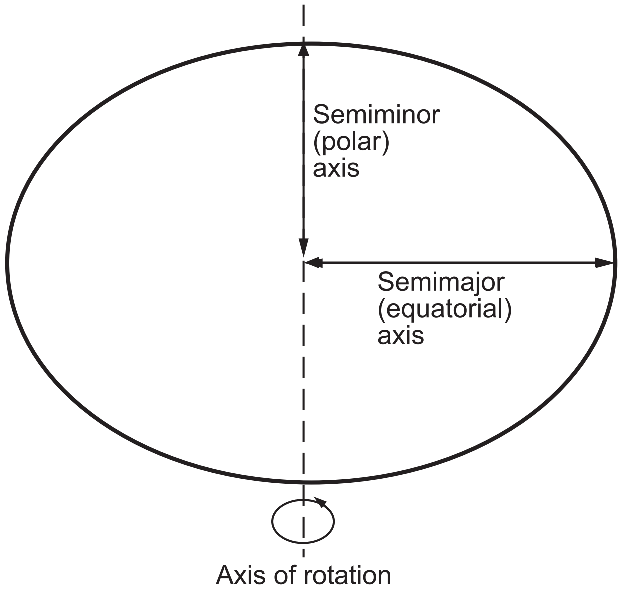
The following figure, made using the `geoid` data set, maps the figure of the Earth. To execute these commands, select them all by dragging over the list in the Help browser, then click the right mouse button and choose `Evaluate Selection`:

```
load geoid; load coast
figure; axesm robinson
geoshow(geoid,geoidlegend,'DisplayType','texturemap')
colorbar('horiz')
geoshow(lat,long,'color','k')
```



The shape of the geoid is important for some purposes, such as calculating satellite orbits, but need not be taken into account for every mapping application. However, knowledge of the geoid is sometimes necessary, for example, when you compare elevations given as height above mean sea level to elevations derived from GPS measurements. Geoid representations are also inherent in datum definitions.

You can define ellipsoids in several ways. They are usually specified by a *semimajor* and a *semiminor axis*, but are often expressed in terms of a semimajor axis and either *inverse flattening* (which for the Earth, as mentioned above, is one part in 300) or *eccentricity*. Whichever parameters are used, as long as an axis length is included, the ellipsoid is fully constrained and the other parameters are derivable. The components of an ellipsoid are shown in the following diagram.



The toolbox includes ellipsoid models that represent the figures of the Sun, Moon, and planets, as well as a set of the most common ellipsoid models of the Earth.

The Ellipsoid Vector

- "Mapping Toolbox Ellipsoid Management" on page 3-7
- "Functions that Define Ellipsoid Vectors" on page 3-9
- "What Is the "Correct" Ellipsoid Vector?" on page 3-9

Mapping Toolbox ellipsoid representations are two-element vectors, called *ellipsoid vectors*. The ellipsoid vector has the form `[semimajor_axis eccentricity]`. The semimajor axis can be in any unit of distance; the choice of units typically drives the units used for distance outputs in the toolbox functions. Meters, kilometers, or Earth radii (i.e., a unit sphere) are most frequently used. See "Functions that Define Ellipsoid Vectors" on page 3-9 for details.

Eccentricity can range from 0 to 1. Most toolbox functions accept a scalar in place of an ellipsoid vector. In this case, its value is interpreted as the radius of a reference sphere, which is equivalent to an ellipsoid with an eccentricity of zero.

Standard values for the ellipsoid vector, along with several other kinds of planetary data for each of the planets and the Earth's moon, are provided by the Mapping Toolbox `almanac` function (see "Planetary Almanac Data" on page 3-46). In the `almanac` function, the default ellipsoid for the Earth is the 1980 Geodetic Reference System ellipsoid:

```
format long g
almanac('earth','ellipsoid','kilometers')

ans =
   6378.137        0.0818191910428158
```

Compare this to a spherical ellipsoid definition:

```
almanac('earth','sphere','kilometers')

ans =
        6371           0
```

You should set `format` to `long g`, as above, if you want to display eccentricity values at full precision.

For example, examine the parameters of the `wgs72` (the 1972 World Geodetic System) ellipsoid, using the `almanac` function:

```
wgs72 = almanac('earth','wgs72','kilometers')

wgs72 =
  6378.135        0.0818188106627487
```

Compare this with Bessel's 1841 ellipsoid:

```
format long g
bessel = almanac('earth','bessel','kilometers')

bessel =
               6377.397155        0.0816968312225275
```

The ellipsoid vector's values are the semimajor axis, in kilometers, and eccentricity. Both eccentricity and flattening are dimensionless ratios. The toolbox has functions to convert elliptical definitions from these forms to ellipsoid vector form. For example, the function `axes2ecc` returns an eccentricity when given semimajor and semiminor axes as arguments.

The ellipse in the previous diagram is highly exaggerated. For the Earth, the semimajor axis is about 21 kilometers longer than the semiminor axis. Use the `almanac` function to verify this:

```
grs80 = almanac('earth','ellipsoid','kilometers')

grs80 =
                  6378.137        0.0818191910428158

semiminor = minaxis(grs80)

semiminor =
   6356.75231414036

semidiff = grs80(1) - semiminor

semidiff =
          21.3846858596444
```

When compared to the semimajor axis, which is almost 6400 kilometers, this difference seems insignificant and can be neglected for world and other small-scale maps. For example, the scale at which 21.38 km would be smaller than a 0.5 mm line on a map (which is a typical line weight in cartography) is

```
kmtomm = unitsratio('mm','km')
kmtomm =
     1000000

scalelim = semidiff * kmtomm / 0.5

scalelim =
  4.2769e+007
```

The `unitsratio` function was used to convert the distance `semidiff` from kilometers into millimeters. This indicates that the Earth's eccentricity is not geometrically meaningful at scales of less than 1:43,000,000, which is roughly the scale of a world map shown on a page of this document. Consequently, most Mapping Toolbox functions default to a spherical model of the Earth. Another reason for defaulting to a sphere is that angular distances are not meaningful on ellipsoids, and some Mapping Toolbox functions compute or use angular distances. See "Working with Distances on the Sphere" on page 3-23 for more information. Regardless, you are free to specify any ellipsoid when you define map axes or otherwise operate on geodata.

Mapping Toolbox Ellipsoid Management

Most maps you make with the toolbox are displayed in a map axes, which is a MATLAB axes that contains a key data structure called a "map projection structure," or *mstruct*. A reference ellipsoid is fundamental to defining a map axes, and is stored in the `geoid` field of the mstruct. (The geographic term "geoid" actually refers to a model of the shape of the earth that is much more detailed. See "Geoid and Ellipsoid" on page 3-2 for more information.) Other mstruct fields specify parameters that define the map axes' current projection and for controlling the appearance of the map frame, grid, and grid labels. You define an mstruct with the `axesm` or `defaultm` functions. See "Map Axes Object Properties" on page 12-36 for definitions of the fields found in mstructs.

You can pass an mstruct to certain functions you call. Other functions obtain the mstruct from the current map axes. (If it is not a map axes, such functions error.) When `axesm` or `defaultm` create a map axes containing an mstruct, their default behavior is to use a unit sphere for the ellipsoid vector. Unless you override this default, you must work in units of earth radii (or radii of whatever planet you are mapping). The following short example shows this clearly (`getm` obtains mstruct parameters from a map axes):

```
worldmap australia
ellipsoid = getm(gca,'geoid')

ans =
     1     0
```

The `worldmap` function chooses map projections and parameters appropriate to the region specified to it and sets up default values for the rest of the mstruct. The geoid parameter is the ellipsoid vector that `worldmap` generated. The first element of the output vector indicates that the semimajor axis has a length of 1; the second element indicates that there is no eccentricity. Therefore, you are working with a sphere—a unit sphere, to be specific.

If, instead of using default ellipsoid vectors, you prefer to be explicit about your reference ellipsoid, then you can work in the length units of your choice, on either a sphere or an ellipsoid. In following example (on the sphere),

```
axesm('mapprojection','mercator',...
      'geoid',almanac('earth','radius','meters'))
 [x, y] = mfwdtran(0,90)

x =
   1.0008e+07
y =
     0
```

the projected map coordinates for a point at 0 degrees latitude, 90 degrees longitude falls just over 10^7 meters east of the origin. If you then revert to a unit sphere (the default ellipsoid), the distance units are quite different:

```
axesm mercator
[x, y] = mfwdtran(0,90)

x =
    1.5708
y =
     0
```

This value for `x` turns out to equal `π/2`, which might tempt you to think that the Mercator projection has simply converted degrees to radians. But what has actually changed is that the point at (0, 90) now maps to a point 1 earth radius east of the origin. Because Mercator is a cylindrical projection having no length distortion along the equator, and because a radian is defined in terms of a sphere's radius, the numbers just happen to work out this way.

Functions that Define Ellipsoid Vectors

Some functions define a radius or an ellipsoid and can make different choices when doing so. In addition to `axesm` and `defaultm`, which create mstructs with ellipsoid vectors that default to a unit sphere, the following functions have default ellipsoid vectors or radii:

The elevation Function. The `elevation` function uses the GRS 80 ellipsoid in meters as its default; unless you specify a reference ellipsoid vector yourself, `elevation` will assume that input altitudes and the output slant range are both in units of meters.

The distance and reckon Functions. These functions assume by default a reference sphere with a radius of 1 (a unit sphere), but scale their range inputs and outputs to equal the size (in degrees) of the angle subtended by rays joining the center of the Earth (or planet) to the start and end points. To obtain results on an ellipsoid you must specify an ellipsoid vector such as `almanac` provides.

Angle-Distance Conversion Functions. The default behavior of the 12 angle-distance conversion utilities (itemized in "Working with Distances on the Sphere" on page 3-23) is different than the above; as discussed below, these functions assume a sphere with a radius of 6371 kilometers (or, equivalently, 3440.065 nautical miles or 3958.748 statute miles), which is a reasonable average radius for Earth.

See the documentation for individual functions if you are not clear whether or how they may generate default reference ellipsoids.

What Is the "Correct" Ellipsoid Vector?

Many different reference ellipsoids have been proposed through the years. They differ because of the surveying information upon which they are based, or because they are intended to approximate the Earth only within a specific geographic region. In many cases you will want to use either the Geodetic Referencing System of 1980 (GRS80) ellipsoid or the World Geodetic System

1984 (WGS84); their semimajor axis lengths are equal and their semiminor axes (i.e., center to pole) differ in length by just over 1/10 mm, as the following code demonstrates:

```
grs80 = almanac('earth','grs80','meters');
wgs84 = almanac('earth','wgs84','meters');
minaxis(wgs84) - minaxis(grs80)

ans =
    1.0482e-004
```

The toolbox supports several other ellipsoid vectors, for models ranging from Everest's 1830 ellipsoid (used for India) to the International Astronomical Union ellipsoid of 1965 (used for Australia). These can be referenced by the following statements:

```
ellipsoid1 = almanac('earth','ellipsoid','kilometers','everest');
ellipsoid2 = almanac('earth','ellipsoid','kilometers','iau65');
```

See the reference page for the `almanac` function for more information on the ellipsoids that are built into the toolbox. If you cannot find the ellipsoid vector you need, you can create it in the following form:

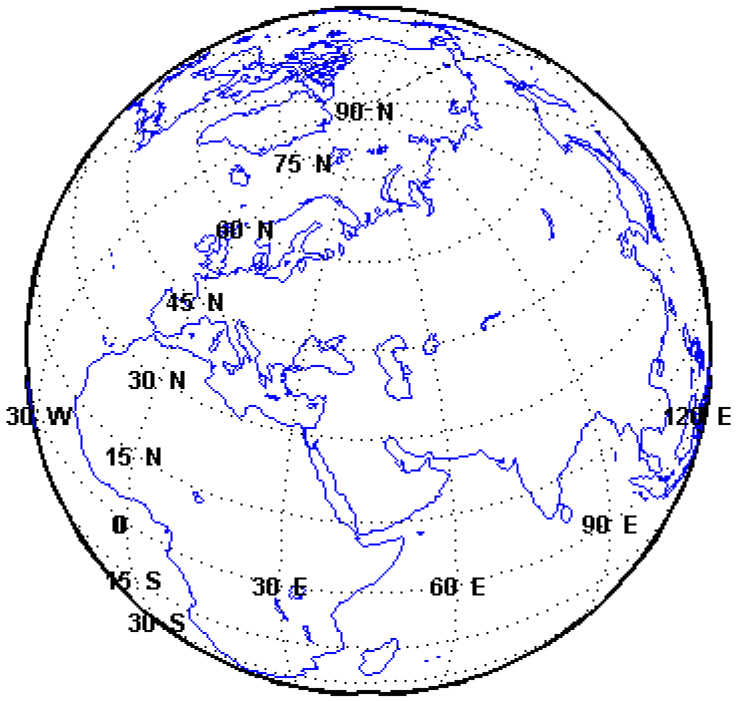
```
ellipsoidvec = [semimajor_axis eccentricity]
```

Understanding Latitude and Longitude

Two angles, *latitude* and *longitude*, specify the position of a point on the surface of a planet. These angles can be in degrees or radians; however, degrees are far more common in geographic notation.

Latitude is the angle between the plane of the equator and a line connecting the point in question to the planet's rotational axis. There are different ways to construct such lines, corresponding to different types of and resulting values for latitudes. Latitude is positive in the northern hemisphere, reaching a limit of +90º at the north pole, and negative in the southern hemisphere, reaching a limit of -90º at the south pole. Lines of constant latitude are called `parallels`. This system is depicted in the following figure, commands for which are

```
load coast
axesm('ortho','origin',[45 45]); axis off;
gridm on; framem on;
mlabel('equator')
plabel(0); plabel('fontweight','bold')
plotm(lat, long)
```



Longitude is the angle at the center of the planet between two planes that align with and intersect along the axis of rotation, perpendicular to the plane of the equator. One plane passes through the surface point in question, and the other plane is the *prime meridian* (0° longitude), which is defined by the location of the Royal Observatory in Greenwich, England. Lines of constant longitude are called *meridians*. All meridians converge at the north and south poles (90ºN and -90ºS), and consequently longitude is under-specified in those two places.

Longitudes typically range from -180° to +180°, but other ranges can be used, such as 0° to +360°. Longitudes can also be specified as east of Greenwich (positive) and west of Greenwich (negative). Adding or subtracting 360° from its longitude does not alter the position of a point. The toolbox includes a set of functions (`wrapTo180`, `wrapTo360`, `wrapToPi`, and `wrapTo2Pi`) that convert longitudes from one range to another. It also provides `unwrapMultipart`, which “unwraps” vectors of longitudes in radians by removing the artifical discontinuities that result from forcing all values to lie within some 360°-wide interval.

Understanding Angles, Directions, and Distances

In this section...

Positions, Azimuths, Headings, Distances, Length, and Ranges

When using spherical coordinates, distances are expressed as angles, not lengths. As there is an infinity of arcs that can connect two points on a sphere or spheroid, by convention the shortest one (the *great circle* distance) is used to measure how close two points are. As is explained in "Working with Distances on the Sphere" on page 3-23, you can convert angular distance on a sphere to linear distance. This is different from working on an ellipsoid, where one can only speak of linear distances between points, and to compute them one must specify which reference ellipsoid to use.

In spherical or geodetic coordinates, a *position* is a latitude taken together with a longitude, e.g., `(lat,lon)`, which defines the horizontal coordinates of a point on the surface of a planet. When we consider two points, e.g.,`(lat1,lon1)` and `(lat2,lon2)`, there are several ways in which their 2–D spatial relationships are typically quantified:

- The azimuth (also called heading) to take to get from `(lat1,lon1)` to `(lat2,lon2)`
- The back azimuth (also called heading) from `(lat2,lon2)` to `(lat1,lon1)`
- The spherical distance separating `(lat1,lon1)` from `(lat2,lon2)`
- The linear distance (range) separating `(lat1,lon1)` from `(lat2,lon2)`

The first three are angular quantities, while the last is a length. Mapping Toolbox functions exist for computing these quantities. For more information, see “Directions and Areas on the Sphere and Spheroid” on page 3-38 and also “Navigation” on page 10-11 for additional examples.

There is no single default unit of distance measurement in the toolbox. Navigation functions use nautical miles as a default, the `almanac` function uses kilometers, and the `distance` function uses degrees of arc length. For many functions, the default unit for distances and positions is degrees, but you need to verify the default assumptions before using any of these functions.

Note When distances are given in terms of angular units (degrees or radians), be careful to remember that these are specified in terms of arc length. While a degree of latitude always subtends one degree of arc length, this is only true for degrees of longitude along the equator.

Working with Length and Distance Units

Linear measurements of lengths and distances on spheres and spheroids can use the same units they do on the plane, such as feet, meters, miles, and kilometers. They can be used for

- Absolute positions, such as map coordinates or terrain elevations
- Dimensions, such as a planet’s radius or its semimajor and semiminor axes
- Distances between points or along routes, in 2-D or 3-D space or across terrain

Length units are needed to describe

- The dimensions of a reference sphere or ellipsoid
- The line-of-sight distance between points

- Distances along great circle or rhumb line curves on an ellipsoid or sphere
- *X*-*Y* locations in a projected coordinate system or map grid
- Offsets from a map origin (false eastings and northings)
- *X*-*Y*-*Z* locations in Earth-centered Earth-fixed (ECEF) or local vertical systems
- Heights of various types (terrain elevations above a geoid, an ellipsoid, or other reference surface

Choosing Units of Length

Using the toolbox effectively depends on being consistent about units of length. Depending on the specific function and the way you are calling it, when you specify lengths, you could be

- Explicitly specifying a radius or reference ellipsoid vector
- Relying on the function itself to specify a default radius or ellipsoid
- Relying on the reference ellipsoid associated with a map projection structure (mstruct)

Whenever you are doing a computation that involves a reference sphere or ellipsoid, make sure that the units of length you are using are the same units used to define the radius of the sphere or semimajor axis of the ellipsoid. These considerations are discussed below.

Converting Units of Length

The following Mapping Toolbox functions convert between different units of length:

- `unitsratio` computes multiplicative factors for converting between 12 different units of length as well as between degrees and radians. You can use `unistratio` to perform conversions when neither the input units of length nor the output units of length are known until run time. See "Converting Angle Units that Vary at Run Time" on page 3-22 for more information.

- `km2nm`, `km2sm`, `nm2km`, `nm2sm`, `sm2km`, and `sm2nm` perform simple and convenient conversions between kilometers, nautical miles, and statute miles.

These utility functions accept scalars, vectors, and matrices, or any shape. For an overview of these functions and angle conversion functions, see "Summary: Available Distance and Angle Conversion Functions" on page 3-26.

Computing Conversion Factors

The `unitsratio` function can compute the ratio between any of the following units of length:

- Microns
- Millimeters
- Centimeters
- Meters
- Kilometers
- Inches
- International feet
- U.S. survey feet
- Yards
- International miles
- U.S. survey (statute) miles

The syntax for `unitsratio` is

```
ratio = unitsratio(to-unit,from-unit)
```

You can use the output from `unitsratio` as a multiplicative conversion factor.

1. For example, the following shows that 4 inches span just over 10 centimeters:

   ```
   cmPerInch = unitsratio('cm','inch')
   cm = cmPerInch * 4
   ```

```
cmPerInch =
    2.5400

cm =
    10.16
```

2 To convert this number of centimeters back to inches, type

```
inch = unitsratio('in','centimeter') * cmPerInch

inch =
     1
```

Note that `unitsratio` supports various abbreviations for units of length.

The `unitsratio` function also lets you convert angles between degrees and radians.

Working with Angles: Units and Representations

- "Radians and Degrees" on page 3-19
- "Default and Variable Angle Units" on page 3-20
- "Degrees, Minutes, and Seconds" on page 3-20
- "Converting Angle Units that Vary at Run Time" on page 3-22

Angular measurements have many distinct roles in geospatial data handling. For example, they are used to specify

- Absolute positions — latitudes and longitudes
- Relative positions — azimuths, bearings, and elevation angles
- Spherical distances between point locations

Absolute positions are expressed in *geodetic coordinates*, which are actually angles between lines or planes on a reference sphere or ellipsoid. Relative positions use units of angle to express the direction between one place on the reference body from another one. Spherical distances quantify how far two places are from one another in terms of the angle subtended along a great-circle arc. On nonspherical reference bodies, distances are usually given in linear units such as kilometers (because on them, arc lengths are no longer proportional to subtended angle).

Radians and Degrees

The basic unit for angles in MATLAB is the radian. For example, if the variable theta represents an angle and you want to take its sine, you can use `sin(theta)` if and only if the value of theta is expressed in radians. If a variable represents the value of an angle in degrees, then you must convert the value to radians before taking the sine. For example,

```
thetaInDegrees = 30;
thetaInRadians = thetaInDegrees * (pi/180)
sinTheta = sin(thetaInRadians)
```

As shown above, you can scale degrees to radians by multiplying by `pi/180`. However, you should consider using the Mapping Toolbox function `degtorad` for this purpose:

```
thetaInRadians = degtorad(thetaInDegrees)
```

Likewise, you can perform the opposite conversion by applying the inverse factor,

```
thetaInDegrees = thetaInRadians * (180/pi)
```

or by using `radtodeg`,

```
thetaInDegrees = radtodeg(thetaInRadians)
```

The practice of using these functions has two significant advantages:

- It reduces the likelihood of human error (e.g., you might type “pi/108” by mistake)
- It signals clearly your intent—important to do should others ever read, modify, or debug your code

The functions `radtodeg` and `degtorad` are very simple and efficient, and operate on vector and higher-dimensioned input as well as scalars.

Default and Variable Angle Units

Unlike MATLAB trigonometric functions, Mapping Toolbox functions do not always assume that angular arguments are in units of radians.

The low-level utility functions intended as building blocks of more complex features or applications work only in units of radians. Examples include the functions `unwrapMultipart` and `meridianarc`.

Many high-level functions, including `distance`, can work in either degrees or radians. Their interpretation of angles is controlled by a string-valued `'angleunits'` input argument. (`angleunits` can be either `'degrees'` or `'radians'`, and can generally be abbreviated.) This flexibility balances convenience and efficiency, although it means that you must take care to check what assumptions each function is making about its inputs.

Degrees, Minutes, and Seconds

In all Mapping Toolbox computations that involve angles in degrees, floating-point numbers (generally MATLAB class double) are used, which allows for integer and fractional values and rational approximations to irrational numbers. However, several traditional notations, which are still in wide use, represent angles as pairs or triplets of numbers, using minutes of arc (1/60 of degree) and seconds of arc (1/60 of a minute):

- Degrees-minutes notation (DM), e.g., 35° 15', equal to 35.25°
- Degrees-minutes-seconds notation (DMS) , e.g., 35° 15' 45", equal to 35.2625°

In degrees-minutes representation, an angle is split into three separate parts:

1. A sign
2. A nonnegative, integer-valued degrees component
3. A nonnegative minutes component, real-valued and in the half-open interval [0 60)

For example, -1 radians is represented by a minus sign (-) and the numbers [57, 17.7468...]. (The fraction in the minutes part approximates an irrational number and is rounded here for display purposes. This subtle point is revisited in the following section.)

The toolbox includes the function `degrees2dm` to perform conversions of this sort. You can use this function to export data in DM form, either for display purposes or for use by another application. For example,

```
degrees2dm(radtodeg(-1))

ans =
  -57.0000   17.7468
```

More generally, `degrees2dm` converts a single-columned input to a pair of columns. Rather than storing the sign in a separate element, `degrees2dm` applies to the first nonzero element in each row. Function `dm2degrees` converts in the opposite direction, producing a real-valued column vector of degrees from a two-column array having an integer degrees and real-valued minutes column. Thus,

```
dm2degrees(degrees2dm(pi)) == pi

ans =
     1
```

Similarly, in degrees-minutes-seconds representation, an angle is split into four separate parts:

1. A sign
2. A nonnegative integer-valued degrees component
3. A minutes component which can be any integer from 0 through 59
4. A nonnegative minutes component, real-valued and in the half-open interval [0 60)

For example, -1 radians is represented by a minus sign (-) and the numbers [57, 17, 44.8062...], which can be seen using Mapping Toolbox function `degrees2dms`,

```
degrees2dms(radtodeg(-1))

ans =
  -57.0000   17.0000   44.8062
```

degrees2dms works like degrees2dm; it converts single-columned input to three-column form, applying the sign to the first nonzero element in each row.

A fourth function, dms2degrees, is similar to dm2degrees and supports data import by producing a real-valued column vector of degrees from an array with an integer-valued degrees column, an integer-value minutes column, and a real-valued seconds column. As noted, the four functions, degrees2dm, degrees2dms, dm2degrees, and dms2degrees, are particular about the shape of their inputs; in this regard they are distinct from the other angle-conversion functions in the toolbox.

The toolbox makes no internal use of DM or DMS representation. The conversion functions dm2degrees and dms2degrees are provided only as tools for data import. Likewise, degrees2dm and degrees2dms are only useful for displaying geographic coordinates on maps, publishing coordinate values, and for formatting data to be exported to other applications. Methods for accomplishing this are discussed below, in "Formatting Latitudes and Longitudes as Strings" on page 3-28.

Converting Angle Units that Vary at Run Time

Functions degtorad and radtodeg are simple to use and efficient, but how do you write code to convert angles if you do not know ahead of time what units the data will use? The toolbox provides a set of utility functions that help you deal with such situations at run time.

In almost all cases—even at the time you are coding—you know either the input or destination angle units. When you do, you can use one of these functions:

- fromDegrees
- toDegrees
- fromRadians
- toRadians

For example, you might wish to implement a very simple sinusoidal projection on the unit sphere, but allow the input latitudes and longitudes to be in either degrees or radians. You can accomplish this as follows:

```
function [x, y] = sinusoidal(lat, lon, angleunits)
  [lat, lon] = toRadians(angleunits, lat, lon);
  x = lon .* cos(lat);
  y = lat;
```

Whenever *`angleunits`* turns out to be `'radians'` at run time, the `toRadians` function has no real work to do; all the functions in this group handle such "no-op" situations efficiently.

In the very rare instances when you must code an application or MATLAB function in which the units of both input angles and output angles remain unknown until run time, you can still accomplish the conversion by using the `unitsratio` function. For example,

```
fromUnits = 'radians';
toUnits = 'degrees';
piInDegrees = unitsratio(toUnits, fromUnits) * pi

piInDegrees =
   180
```

Working with Distances on the Sphere

- "Examples of Spherical-Linear Distance Conversions" on page 3-25
- "Range as an Angle in the distance and reckon Functions" on page 3-26
- "Summary: Available Distance and Angle Conversion Functions" on page 3-26

Many geospatial domains (seismology, for example) describe distances between points on the surface of the earth as angles. This is simply the result of dividing the length of the shortest great-circle arc connecting a pair points by the radius of the Earth (or whatever planet one is measuring). This gives the angle (in radians) subtended by rays from each point that join at the center of the Earth (or other planet). This is sometimes called a "spherical distance." You can thus call the resulting number a "distance in radians." You could also call the same number a “distance in earth radii.” When you work with transformations of geodata, keep this in mind.

You can easily convert that angle from radians to degrees. For example, you can call `distance` to compute the distance in meters from London to Kuala Lumpur:

```
latL =  51.5188;
lonL =  -0.1300;
latK =   2.9519;
lonK = 101.8200;
earthRadiusInMeters = 6371000;
distInMeters = distance(latL, lonL,...
               latK, lonK, earthRadiusInMeters)

distInMeters =
  1.0571e+007
```

Then convert the result to an angle in radians:

```
distInRadians = distInMeters / earthRadiusInMeters

distInRadians =
    1.6593
```

Finally, convert to an angle in degrees:

```
distInDegrees = radtodeg(distInRadians)

distInDegrees =
   95.0692
```

This really only makes sense and produces accurate results when we approximate the Earth (or planet) as a sphere. On an ellipsoid, one can only describe the distance along a geodesic curve using a unit of length.

Mapping Toolbox software includes a set of six functions to conveniently convert distances along the surface of the Earth (or another planet) from units of kilometers (km), nautical miles (nm), or statue miles (sm) to spherical distances in degrees (deg) or radians (rad):

- km2deg, nm2deg, and sm2deg go from length to angle in degrees
- km2rad, nm2rad, and sm2rad go from length to angle in radians

You could replace the final two steps in the preceding example with

```
distInKilometers = distInMeters/1000;
earthRadiusInKm = 6371;
km2deg(distInKilometers, earthRadiusInKm)

ans =
   95.0692
```

Because these conversion can be reversed, the toolbox includes another six convenience functions that convert an angle subtended at the center of a sphere, in degrees or radians, to a great-circle distance along the surface of that sphere:

- deg2km, deg2nm, and deg2sm go from angle in degrees to length
- rad2km, rad2nm, and rad2sm go from angle in radians to length

When given a single input argument, all 12 functions assume a radius of 6,371,000 meters (6371 km, 3440.065 nm, or 3958.748 sm), which is widely-used as an estimate of the average radius of the Earth. An optional second parameter can be used to specify a planetary radius (in output length units) or the name of an object in the Solar System.

Examples of Spherical-Linear Distance Conversions

On the Earth, a degree of arc length at the equator is about 60 nautical miles:

```
nauticalmiles = deg2nm(1)

nauticalmiles =
 60.0405
```

The Earth is the default assumption for these conversion functions. You can use other radii, however:

```
nauticalmiles = deg2nm(1,almanac('moon','radius'))
nauticalmiles =
 30.3338
```

The function `deg2sm` returns distances in statute, rather than nautical, miles:

```
deg2sm(1)

ans =
   69.0932
```

Range as an Angle in the distance and reckon Functions

Certain syntaxes of the `distance` and `reckon` functions use angles to denote distances in the way described above. In the following statements, the range argument, `rng`, is in degrees (along with all the other inputs and outputs):

```
[rng, az] = distance(lat1, lon1, lat2, lon2)
[latout, lonout] = reckon(lat, lon, rng, az)
```

By adding the optional `units` argument, you can use radians instead:

```
[rng, az] = distance(lat1, lon1, lat2, lon2, 'radians')
[latout, lonout] = reckon(lat, lon, rng, az, 'radians')
```

If an `ellipsoid` argument is provided, however, then `rng` has units of length, and they match the units of the semimajor axis length of the reference ellipsoid. If you specify `ellipsoid = [1 0]` (the unit sphere) `rng` can be considered to either an angle in radians or a length defined in units of earth radii. It has the same value either way. Thus, in the following computation, `lat1`, `lon1`, `lat2`, lon2, and `az` are in degrees, but `rng` will appear to be in radians:

```
[rng, az] = distance(lat1, lon1, lat2, lon2, [1 0])
```

Summary: Available Distance and Angle Conversion Functions

The following table shows the Mapping Toolbox unit-to-unit distance and arc conversion functions. They all accept scalar, vector, and higher-dimension inputs. The first two columns and rows involve angle units, the last three involve distance units:

Functions that Directly Convert Angles, Lengths, and Spherical Distances

Convert	To Degrees	To Radians	To Kilometers	To Nautical Miles	To Statute Miles
Degrees	`toDegrees` `fromDegrees`	`degtorad` `toRadians` `fromDegrees`	`deg2km`	`deg2nm`	`deg2sm`
Radians	`radtodeg` `toDegrees` `fromRadians`	`toRadians` `fromRadians`	`rad2km`	`rad2nm`	`rad2sm`
Kilometers	`km2deg`	`km2rad`		`km2nm`	`km2sm`
Nautical Miles	`nm2deg`	`nm2rad`	`nm2km`		`nm2sm`
Statute Miles	`sm2deg`	`sm2rad`	`sm2km`	`sm2nm`	

The angle conversion functions along the major diagonal, `toDegrees`, `toRadians`, `fromDegrees`, and `fromRadians`, can have no-op results. They are intended for use in applications that have no prior knowledge of what angle units might be input or desired as output.

Angles as Binary and Formatted Numbers

The terms *decimal degrees* and *decimal minutes* are often used in geospatial data handling and navigation. The preceding section avoided using them because its focus was on the representation of angles within MATLAB, where they can be arbitrary binary floating-point numbers.

However, once an angle in degrees is converted to a string, it is often helpful to describe that string as representing the angle in decimal degrees. Thus,

```
num2str(radtodeg(1))

ans =
57.2958
```

gives a value in decimal degrees. In casual communication it is common to refer to a quantity such as `radtodeg(1)` as being in decimal degrees, but strictly speaking, that is not true until it is somehow converted to a string in base 10. That is, a binary floating-point number is not a decimal number, whether it represents an angle in degrees or not. If it does represent an angle and that number is then formatted and displayed as having a fractional part, only then is it appropriate to speak of "decimal degrees." Likewise, the term "decimal minutes" applies when you convert a degrees-minutes representation to a string, as in

```
num2str(degrees2dm(radtodeg(1)))

ans =
57      17.7468
```

Formatting Latitudes and Longitudes as Strings

When a DM or DMS representation of an angle is expressed as a string, it is traditional to tag the different components with the special characters `d`, `m`, and `s`, or °, ', and ".

When the angle is a latitude or longitude, a letter often designates the sign of the angle:

- `N` for positive latitudes
- `S` for negative latitudes
- `E` for positive longitudes
- `W` for negative longitudes

For example, 123 degrees, 30 minutes, 12.7 seconds west of Greenwich can be written as 123d30m12.7sW, 123° 30° 12.7" W, or -123° 30° 12.7".

Use the function `str2angle` to import latitude and longitude data formatted as such strings. Conversely, you can format numeric degree data for display or export with `angl2str`, or combine `degrees2dms` or `degrees2dm` with `sprintf` to customize formatting.

See "Degrees, Minutes, and Seconds" on page 3-20 for more details about DM and DMS representation.

Understanding Map Projections

In this section...

What Is a Map Projection?

While all geospatial data needs to be georeferenced (pinned to locations on the Earth's surface) in some way, a given data set might or might not explicitly describe locations with geographic coordinates (latitudes and longitudes). When it does, many applications—particularly map display—cannot make direct use of geographic coordinates, and must transform them in some way to plane coordinates. This transformation process, called *map projection*, is both algorithmic and the core of the cartographer's art.

A map projection is a procedure that unwraps a sphere or ellipsoid to flatten it onto a plane. Usually this is done through an intermediate surface such as a cylinder or a cone, which is then unwrapped to lie flat. Consequently, map projections are classified as cylindrical, conical, and azimuthal (a direct transformation of the surface of part of a spheroid to a circle). See "The Three Main Families of Map Projections" on page 8-5 for discussions and illustrations of how these transformations work.

Mapping Toolbox map projection libraries feature dozens of map projections, which you principally control with `axesm`. Some are ancient and well-known (such as Mercator), others are ancient and obscure (such as Bonne), while some are modern inventions (such as Robinson). Some are suitable for showing the entire world, others for half of it, and some are only useful over small areas. When geospatial data has geographic coordinates, any projection can be applied, although some are not good choices. The toolbox can project both vector data and raster data.

See Chapter 8, "Using Map Projections and Coordinate Systems" for more details on the properties of different classes of projections. For a list of Mapping Toolbox map projections, with links to their reference pages, see Chapter 14, "Map Projections Reference". "Summary and Guide to

Projections" on page 8-63 lists all the available map projections and their intrinsic properties.

Forward and Inverse Projection

When geospatial data has plane coordinates (i.e., it comes preprojected, as do many satellite images and municipal map data sets), it is usually possible to recover geographic coordinates if the projection parameters and datum are known. Using this information, you can perform an *inverse projection*, running the projection backward to solve for latitude and longitude. The toolbox can perform accurate inverse projections for any of its projection functions as long as the original projection parameters and reference ellipsoid (or spherical radius) are provided to it.

Note Converting a position given in latitude-longitude to its equivalent in a projected map coordinate system involves converting from units of angle to units of length. Likewise, unprojecting a point position changes its units from those of length to those of angle). Unit conversion functions such as `deg2km` and `km2deg` also convert coordinates between angles and lengths, but do not transform the space they inhabit. You cannot use them to project or unproject coordinate data.

Projection Distortions

All map projections introduce distortions compared to maps on globes. Distortions are inherent in flattening the sphere, and can take several forms:

- Areas — Relative size of objects (such as continents)
- Distances — Relative separations of points (such as a set of cities)
- Directions — Azimuths (angles between points and the poles)
- Shapes — Relative lengths and angles of intersection

Some classes of map projections maintain areas, and others preserve local shapes, distances, and/or directions. No projection, however, can preserve all these characteristics. Choosing a projection thus always requires compromising accuracy in some way, and that is one reason why so many different map projections have been developed. For any given projection, however, the smaller the area being mapped, the less distortion it introduces if properly centered. Mapping Toolbox tools help you to quantify and visualize projection distortions.

Great Circles, Rhumb Lines, and Small Circles

In this section...

Great Circles

In plane geometry, lines have two important characteristics. A line represents the shortest path between two points, and the slope of such a line is constant. When describing lines on the surface of a spheroid, however, only one of these characteristics can be guaranteed at a time.

A *great circle* is the shortest path between two points along the surface of a sphere. The precise definition of a great circle is the intersection of the surface with a plane passing through the center of the planet. Thus, great circles always bisect the sphere. The equator and all meridians are great circles. All great circles other than these do not have a constant azimuth, the spherical analog of slope; they cross successive meridians at different angles. That great circles are the shortest path between points is not always apparent from maps, because very few map projections (the Gnomonic is one of them) represent arbitrary great circles as straight lines.

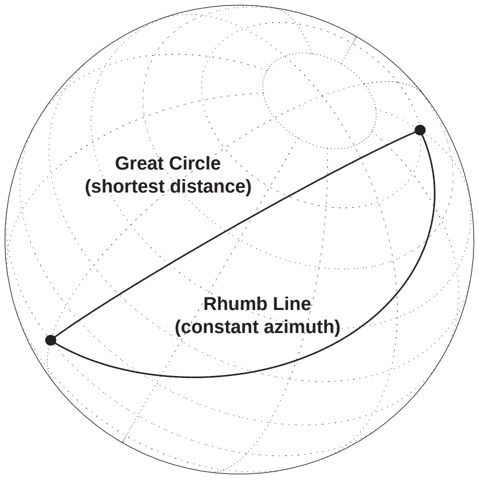
Because they define paths that minimize distance between two (or three) points, great circles are examples of *geodesics*. In general, a geodesic is the straightest possible path constrained to lie on a curved surface, independent of the choice of a coordinate system. The term comes from the Greek *geo-*, earth, plus *daiesthai*, to divide, which is also the root word of *geodesy*, the science of describing the size and shape of the Earth mathematically.

Rhumb Lines

A *rhumb line* is a curve that crosses each meridian at the same angle. This curve is also referred to as a *loxodrome* (from the Greek *loxos*, slanted, and *drome*, path). Although a great circle is a shortest path, it is difficult to navigate because your bearing (or *azimuth*) continuously changes as you proceed. Following a rhumb line covers more distance than following a geodesic, but it is easier to navigate.

All parallels, including the equator, are rhumb lines, since they cross all meridians at 90º. Additionally, all meridians are rhumb lines, in addition to being great circles. A rhumb line always spirals toward one of the poles, unless its azimuth is true east, west, north, or south, in which case the rhumb line closes on itself to form a parallel of latitude (small circle) or a pair of antipodal meridians.

The following figure depicts a great circle and one possible rhumb line connecting two distant locations. Descriptions and examples of how to calculate points along great circles and rhumb lines appear below.



Small Circles

In addition to rhumb lines and great circles, one other smooth curve is significant in geography, the *small circle*. Parallels of latitude are all small circles (which also happen to be rhumb lines). The general definition of a small circle is the intersection of a plane with the surface of a sphere. On ellipsoids, this only yields true small circles when the defining plane is parallel to the equator. Mapping Toolbox software extends this definition to include planes passing through the center of the planet, so the set of all small circles includes all great circles as limiting cases. This usage is not universal.

Small circles are most easily defined by distance from a point. *All points 45 nm (nautical miles) distant from (45ºN,60ºE)* would be the description of one small circle. If degrees of arc length are used as a distance measurement, then (on a sphere) a great circle is the set of all points 90º distant from a particular *center* point.

For true small circles, the distance must be defined in a great circle sense, the shortest distance between two points on the surface of a sphere. However, Mapping Toolbox functions also can calculate *loxodromic small circles*, for which distances are measured in a rhumb line sense (along lines of constant azimuth). Do not confuse such figures with true small circles.

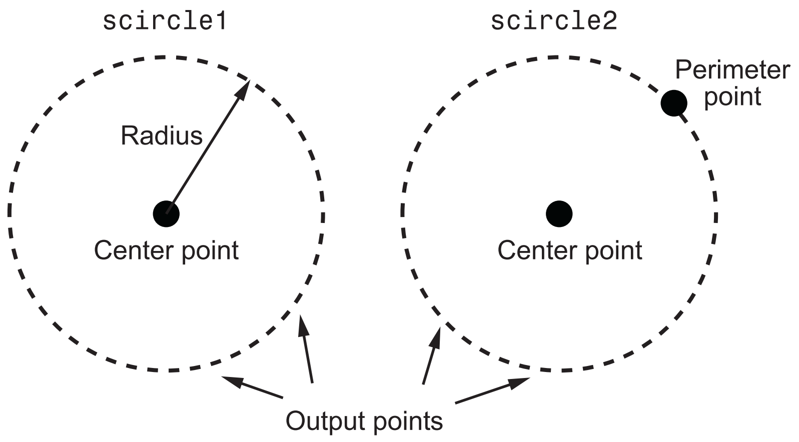
Computing Small Circles

You can calculate vector data for points along a small circle in two ways. If you have a center point and a known radius, use `scircle1`; if you have a center point and a single point along the circumference of the small circle, use `scircle2`. For example, to get data points describing the small circle at 10º distance from (67ºN, 135ºW), use the following:

```
[latc,lonc] = scircle1(67,-135,10);
```

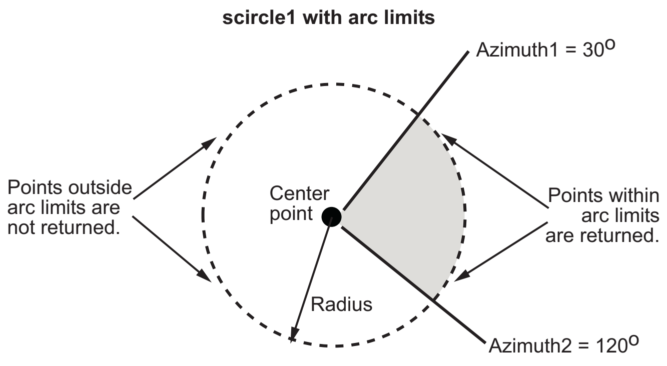
To get the small circle centered at the same point that passes through the point (55ºN,135ºW), use `scircle2`:

```
[latc,lonc] = scircle2(67,-135,55,-135);
```



The `scircle1` function also allows you to calculate points along a specific arc of the small circle. For example, if you want to know the points 10º in distance and between 30º and 120º in azimuth from (67ºN,135ºW), simply provide arc limits:

```
[latc,lonc] = scircle1(67,-154,10,[30 120]);
```

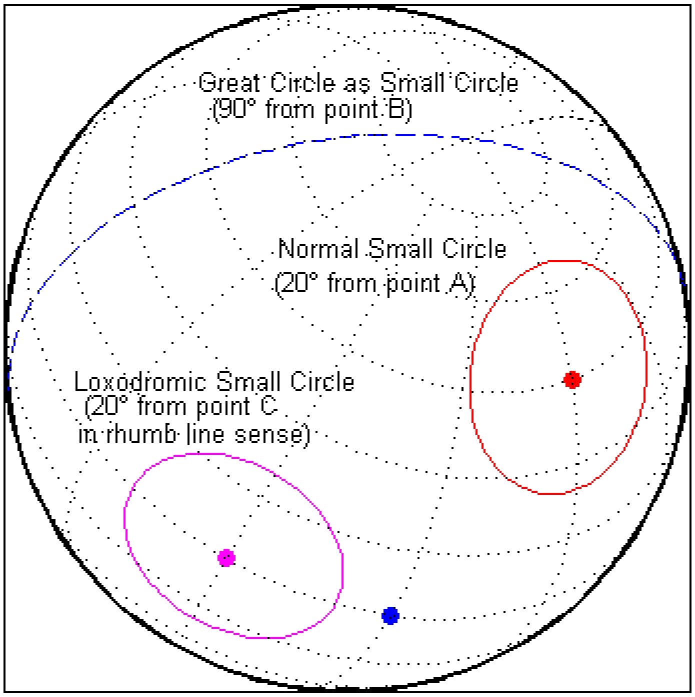

scircle1 with arc limits
Azimuth1 = 30°
Points outside arc limits are not returned.
Center point
Points within arc limits are returned.
Radius
Azimuth2 = 120°

When an entire small circle is calculated, the data is in polygon format. For all calculated small circles, 100 points are returned unless otherwise specified. You can calculate several small circles at once by providing vector inputs. For more information, see the `scircle1` and `scircle2` function reference pages.

An Annotated Map Illustrating Small Circles. The following Mapping Toolbox commands illustrate generating small circles of the types described above, including the limiting case of a large circle. To execute these commands, select them all by dragging over the list in the Help browser, then click the right mouse button and choose `Evaluate Selection`:

```
figure;
axesm ortho; gridm on; framem on
setm(gca,'Origin', [45 30 30], 'MLineLimit', [75 -75],...
'MLineException',[0 90 180 270])
A = [45 90];
B = [0 60];
C = [0 30];
sca = scircle1(A(1), A(2), 20);
scb = scircle2(B(1), B(2), 0, 150);
scc = scircle1('rh',C(1), C(2), 20);
plotm(A(1), A(2),'ro','MarkerFaceColor','r')
plotm(B(1), B(2),'bo','MarkerFaceColor','b')
plotm(C(1), C(2),'mo','MarkerFaceColor','m')
plotm(sca(:,1), sca(:,2),'r')
plotm(scb(:,1), scb(:,2),'b--')
plotm(scc(:,1), scc(:,2),'m')
textm(50,0,'Normal Small Circle')
textm(46,6,'(20\circ from point A)')
textm(4.5,-10,'Loxodromic Small Circle')
textm(4,-6,'(20\circ from point C')
textm(-2,-4,'in rhumb line sense)')
textm(40,-60,'Great Circle as Small Circle')
textm(45,-50,'(90\circ from point B)')
```

The result is the following display.


Great Circle as Small Circle
(90° from point B)
Normal Small Circle
(20° from point A)
Loxodromic Small Circle
(20° from point C
in rhumb line sense)

Directions and Areas on the Sphere and Spheroid

In this section...

About Azimuths

Azimuth is the angle a line makes with a meridian, measured clockwise from north. Thus the azimuth of due north is 0º, due east is 90º, due south is 180º, and due west is 270º. You can instruct several Mapping Toolbox functions to compute azimuths for any pair of point locations, either along rhumb lines or along great circles. These will have different results except along cardinal directions. For great circles, the result is the azimuth at the initial point of the pair defining a great circle path. This is because great circle azimuths other than 0º, 90º, 180º, and 270º do not remain constant. Azimuths for rhumb lines are constant along their entire path (by definition).

For rhumb lines, computing an azimuth backward (from the second point to the first) yields the complement of the forward azimuth ((Az + 180º) mod 360º). For great circles, the back azimuth is generally not the complement, and the difference depends on the distance between the two points.

In addition to forward and back azimuths, Mapping Toolbox functions can compute locations of points a given distance and azimuth from a reference point, and can calculate tracks to connect waypoints, along either great circles or rhumb lines on a sphere or ellipsoid.

Reckoning — The Forward Problem

A common problem in geographic applications is the determination of a destination given a starting point, an initial azimuth, and a distance. In the toolbox, this process is called *reckoning*. A new position can be reckoned in a great circle or a rhumb line sense (great circle or rhumb line track).

As an example, an airplane takes off from La Guardia Airport in New York (40.75ºN, 73.9ºW) and follows a northwestern rhumb line flight path at 200 knots (nautical miles per hour). Where would it be after 1 hour?

```
[rhlat,rhlong] = reckon('rh',40.75,-73.9,nm2deg(200),315)

rhlat =
    43.1054
rhlong =
   -77.0665
```

Notice that the distance, 200 nautical miles, must be converted to degrees of arc length with the nm2deg conversion function to match the latitude and longitude inputs. If the airplane had a flight computer that allowed it to follow an exact great circle path, what would the aircraft's new location be?

```
[gclat,gclong] = reckon('gc',40.75,-73.9,nm2deg(200),315)

gclat =
    43.0615
gclong =
   -77.1238
```

Notice also that for short distances at these latitudes, the result hardly differs between great circle and rhumb line. The two destination points are less than 4 nautical miles apart. Incidentally, after 1 hour, the airplane would be just north of New York's Finger Lakes.

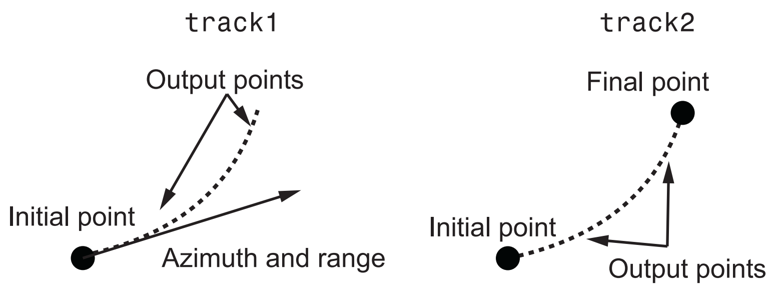
Calculating Tracks — Great Circles and Rhumb Lines

You can generate vector data corresponding to points along great circle or rhumb line tracks using `track1` and `track2`. If you have a point on the track and an azimuth at that point, use `track1`. If you have two points on the track, use `track2`. For example, to get the great circle path starting at (31ºS, 90ºE) with an azimuth of 45º with a length of 12º, use `track1`:

```
[latgc,longc] = track1('gc',-31,90,45,12);
```

For the great circle from (31ºS, 90ºE) to (23ºS, 110ºE), use `track2`:

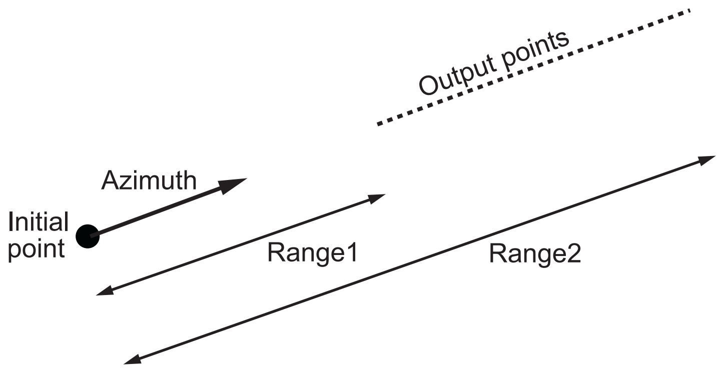
```
[latgc,longc] = track2('gc',-31,90,-23,110);
```



The `track1` function also allows you to specify range endpoints. For example, if you want points along a rhumb line starting 5º away from the initial point and ending 13º away, at an azimuth of 55º, simply specify the range limits:

```
[latrh,lonrh] = track1('rh',-31,90,55,[5 13]);
```

track1 with range limits



When no range is provided for `track1`, the returned points represent a *complete track*. For great circles, a complete track is 360º, encircling the planet and returning to the initial point. For rhumb lines, the complete track terminates at the poles, unless the azimuth is 90º or 270º, in which case the complete track is a parallel that returns to the initial point.

For calculated tracks, 100 points are returned unless otherwise specified. You can calculate several tracks at one time by providing vector inputs. For more information, see the `track1` and `track2` reference pages. More vector path calculations are described later in “Navigation” on page 10-11.

Distance, Azimuth, and Back-Azimuth (the Inverse Problem)

When Mapping Toolbox functions calculate the distance between two points in geographic space, the result depends upon whether you specify great circle or rhumb line distance. The `distance` function returns the appropriate distance between two points as an angular arc length, employing the same angular units as the input latitudes and longitudes. The default path type is the shorter great circle, and the default angular units are degrees. The previous figure shows two points at (15ºS, 0º) and (60ºN, 150ºE). The great circle distance between them, in degrees of arc, is as follows:

```
distgc = distance(-15,0,60,150)

distgc =
   129.9712
```

The rhumb line distance is greater:

```
distrh = distance('rh',-15,0,60,150)

distrh =
   145.0288
```

To determine how much longer the rhumb line path is in, say, kilometers, you can use a distance conversion function on the difference:

```
kmdifference = deg2km(distrh-distgc)

kmdifference =
   1.6744e+03
```

Several distance conversion functions are available in the toolbox, supporting degrees, radians, kilometers, meters, statute miles, nautical miles, and feet. Converting distances between angular arc length units and surface length units requires the radius of a planet or spheroid. By default, the radius of the Earth is used.

Calculating Azimuth and Elevation

Azimuth is the angle a line makes with a meridian, taken clockwise from north. When the azimuth is calculated from one point to another using the toolbox, the result depends upon whether you want a great circle or a rhumb line azimuth. For great circles, the result is the azimuth at the starting point of the connecting great circle path. In general, the azimuth along a great circle is not constant. For rhumb lines, the resulting azimuth is constant along the entire path.

Azimuths, or bearings, are returned in the same angular units as the input latitudes and longitudes. The default path type is the shorter great circle, and the default angular units are degrees. In the example, the great circle azimuth from the first point to the second is

```
azgc = azimuth(-15,0,60,150)

azgc =
   19.0391
```

For the rhumb line, the constant azimuth is

```
azrh = azimuth('rh',-15,0,60,150)

azrh =
   58.8595
```

One feature of rhumb lines is that the inverse azimuth, from the second point to the first, is the complement of the forward azimuth and can be calculated by simply adding 180º to the forward value:

```
inverserh = azimuth('rh',60,150,-15,0)

inverserh =
   238.8595

difference = inverserh-azrh

difference =
   180
```

This is not true, in general, of great circles:

```
inversegc = azimuth('gc',60,150,-15,0)

inversegc =
   320.9353

difference = inversegc-azgc

difference =
   301.8962
```

The azimuths associated with cardinal and intercardinal compass directions are the following:

North	0º or 360º
Northeast	45º
East	90º
Southeast	135º
South	180º
Southwest	225º
West	270º
Northwest	315º

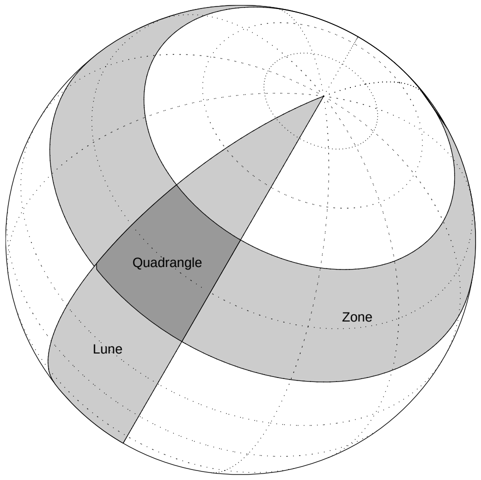
Elevation is the angle above the local horizontal of one point relative to the other. To compute the elevation angle of a second point as viewed from the first, provide the position and altitude of the points. The default units are degrees for latitudes and longitudes and meters for altitudes, but you can specify other units for each. What are the elevation, slant range, and azimuth of a point 10 kilometers east and 10 kilometers above a surface point?

```
[elevang,slantrange,azim] = elevation(0,0,0,0,km2deg(10),10000)

elevang =
       44.901
slantrange =
        14156
azim =
        90
```

On an ellipsoid, azimuths returned from `elevation` generally will differ from those returned by `azimuth` and `distance`.

Measuring Area of Spherical Quadrangles

In solid geometry, the area of a spherical quadrangle can be exactly calculated. A spherical quadrangle is the intersection of a *lune* and a *zone*. In geographic terms, a *quadrangle* is defined as a region bounded by parallels north and south, and meridians east and west.



In the pictured example, a quadrangle is formed by the intersection of a zone, which is the region bounded by 15ºN and 45ºN latitudes, and a *lune*, which is the region bounded by 0º and 30ºE longitude. Under the spherical planet assumption, the fraction of the entire spherical surface area inscribed in the quadrangle can be calculated:

```
area = areaquad(15,0,45,30)

area =
   0.0187
```

That is, less than 2% of the planet's surface area is in this quadrangle. To get an absolute figure in, for example, square miles, you must provide the appropriate spherical radius. The radius of the Earth is about 3958.9 miles:

```
area = areaquad(15,0,45,30,3958.9)

area =
     3.6788e+06
```

The surface area within this quadrangle is over 3.6 million square miles for a spherical Earth.

Planetary Almanac Data

Mapping Toolbox functions include one that provides almanac data (size and shape statistics) for the major bodies of our solar system. Basic geometric parameters, such as ellipsoid vectors, radii, surface areas, and volumes, can be accessed for the Sun, the Earth's moon, and all of the planets, in any of the supported units of distance measurement.

Many planets have ellipsoid vectors available. Some planets return spherical ellipsoid vectors only:

```
almanac('earth','ellipsoid','nauticalmiles')

ans =
   3443.92          0.08

almanac('mars','ellipsoid','kilometers')

ans =
   3396.90          0.11

almanac('moon','ellipsoid','statutemiles')

ans =
   1079.97             0
```

When you specify `'radius'`, a scalar is returned representing the radius of the best spherical model of the planet. Notice that for a spherical model, the radius in radians is 1:

```
almanac('mercury','radius','kilometers')

ans =
   2439

almanac('neptune','radius','radians')

ans =
   1
```

Surface areas and volumes are calculated based on a spherical model by default. In most cases, you can use the ellipsoid model instead, and for the Earth you can specify any of the supported ellipsoid models. You can also request the actual tabulated values of the Earth:

```
almanac('mars','surfarea','kilometers','ellipsoid')

ans =
   1.4441e+08

almanac('earth','volume','kilometers','international')

ans =
   1.0833e+12

almanac('earth','volume','kilometers','actual')

ans =
   1.0832e+12
```

For a complete description of available data, see the `almanac` reference page.

Creating and Viewing Maps

Introduction to Mapping Graphics

Even though geospatial data often is manipulated and analyzed without being displayed, high-quality interactive cartographic displays can play valuable roles in exploratory data analysis, application development, and presentation of results.

Using Mapping Toolbox capabilties, you can display geographic information almost as easily as you can display tabular or time-series data in MATLAB plots. Most mapping functions are similar to MATLAB plotting functions, except they accept data with geographic/geodetic coordinates (latitudes and longitudes) instead of Cartesian and polar coordinates. Mapping functions typically have the same names as their MATLAB counterparts, with the addition of an `'m'` suffix (for maps). For example, the Mapping Toolbox analog to the MATLAB `plot` function is `plotm`.

Mapping Toolbox software manages most of the details in displaying a map. It projects your data, cuts and trims it to specified limits, and displays the resulting map at various scales. With the toolbox you can also add customary cartographic elements, such as a frame, grid lines, coordinate labels, and text labels, to your displayed map. If you change your projection properties, or even the projection itself, some Mapping Toolbox map displays are automatically redrawn with the new settings, undoing any cuts or trims if necessary. See “Accessing, Computing, and Inverting Map Projection Data” on page 8-37 for information on how to project data without displaying it.

The toolbox also makes it easy to modify and manipulate maps. You can modify the map display and mapped objects either from the command line or through and property editing tools you can invoke by clicking on the display.

Note In its current implementation, the toolbox maintains the map projection and display properties by storing special data in the `UserData` property of the map axes. The toolbox also takes over the `UserData` property of mapped objects. Therefore, never attempt to set the `UserData` property of a map axes or a projected map object. Do not apply the MATLAB `get` function to axes `UserData`, depend on the contents of `UserData` in any way, or apply functions that set or get `UserData` to the handles of map axes or mapped objects. Only use the Mapping Toolbox functions `getm` and `setm` to obtain and modify map axes properties.

Using worldmap and usamap

In this section...

Continent, Country, Region, and State Maps Made Easy

Mapping Toolbox functions `axesm` and `setm` enable you to control the full range of properties when constructing a projected map axes. Functions `worldmap` and `usamap`, on the other hand, trade control for simplicity and convenience. These two functions each create a map axes object that is suitable for a country or region of the world or the United States, automatically selecting the map projection, limits, and other properties based on the name of the area you want to map. Once you have jump-started your map with `worldmap` or `usamap`, you are ready to add your data, using `geoshow` or any of the lower level geographic data display functions. Optionally, you can use the map axes object created by `worldmap` or `usamap` as a starting point, and then customize it by adjusting selected properties with `setm`.

Setting Background Colors for Map Displays

The default color for MATLAB figures is gray. If you prefer that your maps have white backgrounds instead, you can create figures with the command

```
figure('Color','white')
```

If you want a custom background color, specify a color triplet in place of `white`. For example, to make a beige background, type

```
figure('Color',[.95 .9 .8])
```

To give a white background to an existing figure, type

```
set(gca,'color','white')
```

If you want all figures in a session to have white backgrounds, set this as a default with the command

```
set(0, 'DefaultFigureColor', 'white');
```

To avoid having to do this every time you start MATLAB, place this command in your `startup.m` file.

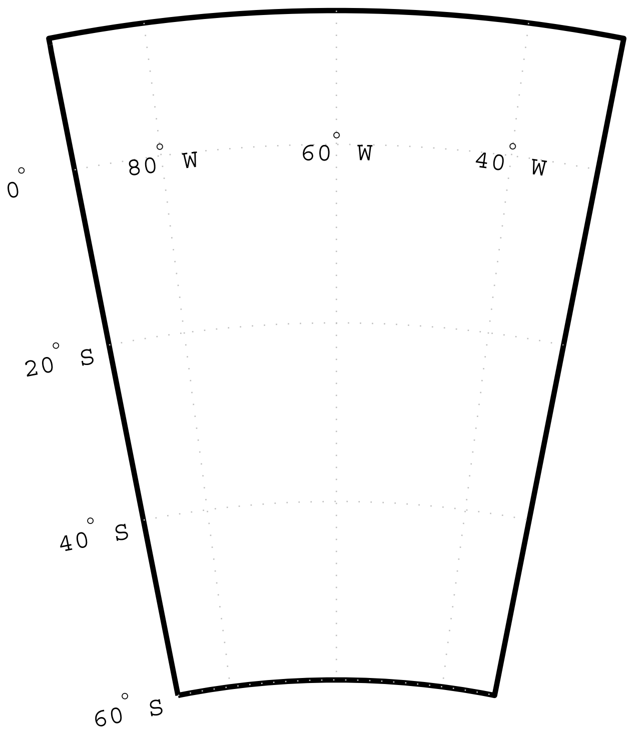
You can also use the Property Editor, part of the MATLAB plotting tools, to modify background colors for figures and axes. See "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation for more information.

Using worldmap

Here are two examples that create simple maps using sample data sets from *`matlabroot`*`/toolbox/map/mapdemos`. The first one creates a map of South America with land areas, major lakes and rivers, and populated places.

1. First, set up the map frame, allowing `worldmap` to pick a projection:

   ```
   figure
   worldmap 'south america'
   axis off
   ```



2 You can find out what map projection `worldmap` selected this way:

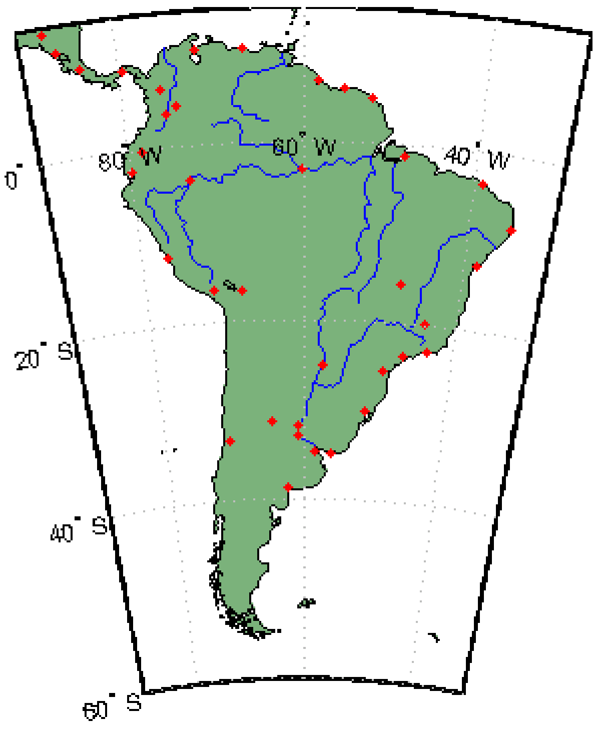
```
getm(gca,'MapProjection')

ans =
eqdconic
```

This denotes the Equidistant Conic Projection, which is appropriate for regions in middle latitudes that are elongated along the polar axis.

3 Next, use `geoshow` to import data for land areas, major rivers, and major cities from shapefiles and display it using colors you specify:

```
    geoshow('landareas.shp', 'FaceColor',  [0.5 0.7 0.5])
    geoshow('worldrivers.shp', 'Color', 'blue')
    geoshow('worldcities.shp', 'Marker', '.', 'Color', 'red')
```

The map now looks like this.



Using usamap

The `usamap` function allows you to make maps of the United States as a whole, just the conterminous portion (the "lower 48" states), groups of states or a single state. The easiest way to use it is to type

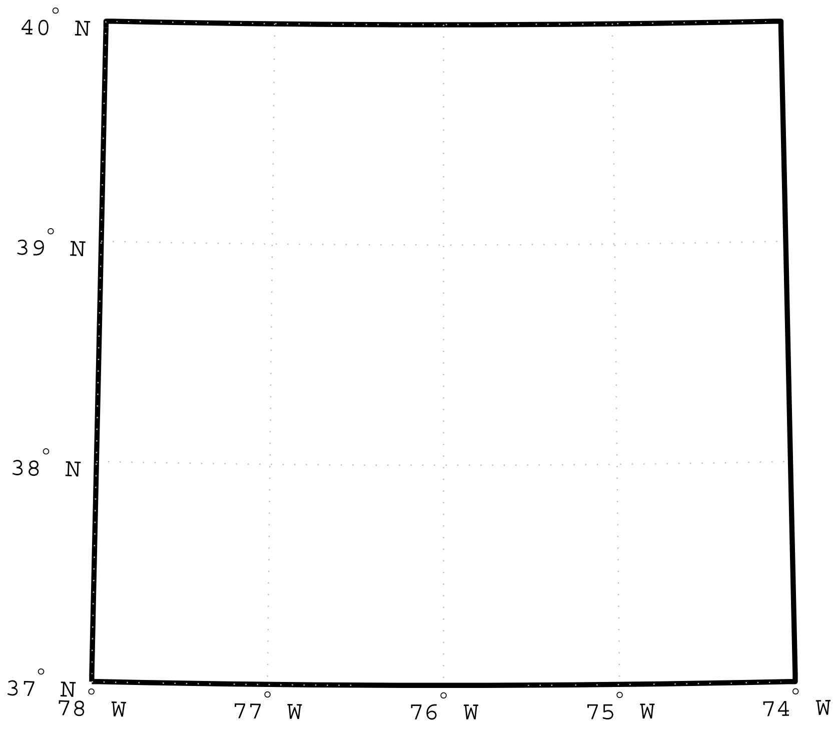
```
usamap
```

at the MATLAB prompt. This opens a GUI with a list box from which you can select the entire U.S., the conterminous states, or an individual state to map. The map axes you create with `usamap` has a labelled grid fitted around the area you specify, but contains no data, allowing you to generate the kind of map you want using display functions such as `geoshow`.

This example creates a map of the Chesapeake Bay region by specifying geographic limits.

1 First, specify limits and set up a map axes object:

```
latlim = [ 37  40];
lonlim = [-78 -74];
figure
ax = usamap(latlim,lonlim);
axis off
```



The `Lambert Conformal Conic Projection` is often used for maps of the conterminous United States.

2 Here is the map projection `usamap` selected:

```
getm(gca,'MapProjection')

ans =
lambert
```

3 Next, use `shaperead` to read U.S. state polygon boundaries from the `usastatehi` demo shapefile into a geostruct named `states`:

```
states = shaperead('usastatehi',...
    'UseGeoCoords', true, 'BoundingBox', [lonlim', latlim']);
```

4 Make a symbolspec to create a political map using the `polcmap` function:

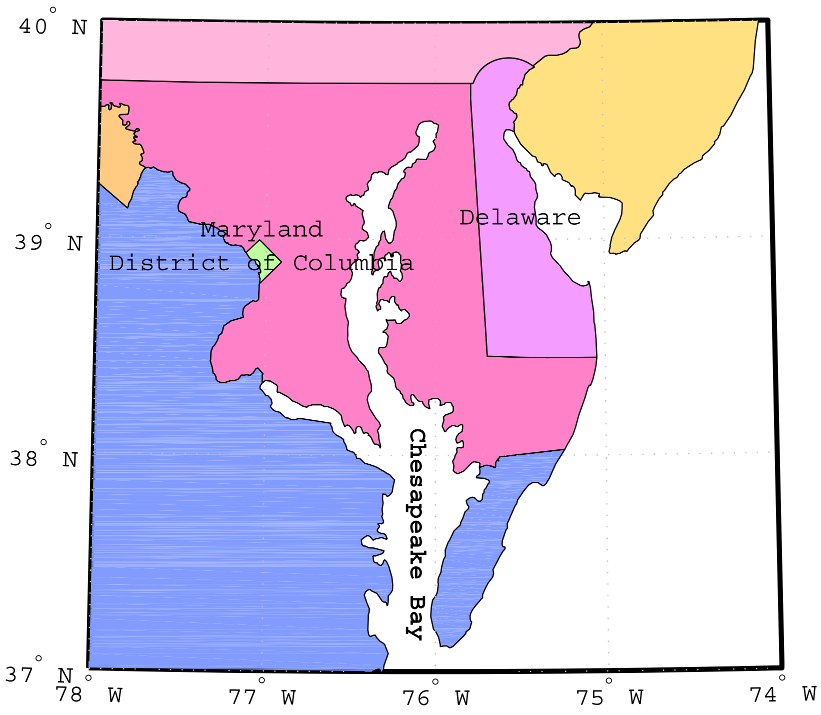
```
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], ...
    'FaceColor', polcmap(numel(states))});
```

5 Display the filled polygons with `geoshow`:

```
geoshow(ax, states, 'SymbolSpec', faceColors)
```

6 Extract the names for states within the window from the geostruct and use `textm` to plot them at the label points provided by the geostruct:

```
for k = 1:numel(states)
  labelPointIsWithinLimits =...
    latlim(1) < states(k).LabelLat &&...
    latlim(2) > states(k).LabelLat &&...
    lonlim(1) < states(k).LabelLon &&...
    lonlim(2) > states(k).LabelLon;
  if labelPointIsWithinLimits
    textm(states(k).LabelLat,...
    states(k).LabelLon, states(k).Name, ...
        'HorizontalAlignment', 'center')
  end
end
textm(38.2,-76.1,' Chesapeake Bay ',...
    'fontweight','bold','Rotation', 270)
```



Note that as `polcmap` assigns random pastel colors to patches, your map might display different colors than this example. For further information on options for these functions, see the reference pages for `geoshow`, `shaperead`, `worldmap`, and `usamap`.

Axes for Drawing Maps

In this section...

What Is a Map Axes?

When you create a map, you can use one of the Mapping Toolbox built-in user interfaces (UIs), or you can build the graphic with MATLAB and Mapping Toolbox functions. Many MATLAB graphics are built using the `axes` function:

```
axes
axes('PropertyName',PropertyValue,...)
axes(h)
h = axes(...)
```

Mapping Toolbox functions include an extended version of `axes`, called `axesm`, that includes information about the current coordinate system (map projection), as well as data to define the map grid and its labeling, the map frame and its limits, and other properties. Its syntax is similar to that of `axes`:

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

The `axesm` function without arguments brings up a UI that lists all supported projections and assists in defining their parameters. You can also summon this UI with the `axesmui` function once you have created a map axes.

You can also list all the names, classes, and ID strings of Mapping Toolbox map projections with the `maps` function.

Axes created with `axesm` share all properties associated with regular axes, and have additional properties related to projections, scale, and positioning in geographic coordinates. See the `axes` and `axesm` reference pages for lists of properties.

You can place many types of objects in a map axes, such as lines, patches, markers, scale rulers, north arrows, grids, and text. You can use the `handlem` function and its associated UI to list these objects and obtain handles to them. See the `handlem` reference page for a list of the objects that can occupy a map axes and how to query for them.

Map axes objects created by `axesm` contain projection information in a structure. For an example of what these properties are, type

```
h = axesm('MapProjection','mercator')
```

and then use the `getm` function to retrieve all the map axes properties:

```
p = getm(h)
```

For complete descriptions of all map axes properties, see the `axesm` reference page. For more information on the use of `axesmui`, refer to the `axesm, axesmui` reference page.

Using `axesm`

The figure window created using `axesm` contains the same set of tools and menus as any MATLAB figure, and is by default blank, even if there is map data in your workspace. You can toggle certain properties, such as grids, frames, and axis labels, by right-clicking in the figure window to obtain a pop-up menu.

You can define multiple independent figures containing map axes, but only one can be active at any one time. Return handles for them when you create them to allow them to be referenced when they are no longer current. Use `axes(handle)` to activate an existing map axes object.

Accessing and Manipulating Map Axes Properties

Just as the properties of the underlying standard axes can be accessed and manipulated using the MATLAB functions `set` and `get`, map axes properties can also be accessed and manipulated using the functions `setm` and `getm`.

Note Use the `axesm` function only to *create* a map axes object. Use the `setm` function to *modify* existing map axes.

---

1. As an example, create a map axes object containing no map data:

   ```
   axesm('MapProjection','miller','Frame','on')
   ```

   Note that you specify `MapProjection` string values in lowercase. At this point you can begin to customize the map. For example, you might decide to make the frame lines bordering the map thicker. First, you need to identify the current line width of the frame, which you do by querying the current axes, identified as `gca`.

2. Access the current `FLineWidth` property value by typing

   ```
   getm(gca,'FLineWidth')
   ans =
    2
   ```

3. Now reset the line width to four points. The default `fontunits` value for axes is `points`. You can set `fontunits` to be `points`, `normalized`, `inches`, `centimeters`, or `pixels`.

   ```
   setm(gca,'FLineWidth',4)
   ```

4. You can set any number of properties simultaneously with `setm`. Continue by reducing the line width, changing the projection to equidistant cylindrical, and verify the changes:

   ```
   setm(gca,'FLineWidth',3,'MapProjection','eqdcylin')

   getm(gca,'FLineWidth')
   ans =
    3
   ```

```
getm(gca,'MapProjection')
ans =
eqdcylin
```

5 To inspect the entire set of map axes properties at their current settings, use the following command:

```
getm(gca)
ans =
     mapprojection: 'eqdcylin'
              zone: []
        angleunits: 'degrees'
            aspect: 'normal'
      falseeasting: []
     falsenorthing: []
       fixedorient: []
             geoid: [1 0]
       maplatlimit: [-90 90]
       maplonlimit: [-180 180]
      mapparallels: 30
        nparallels: 1
            origin: [0 0 0]
       scalefactor: []
           trimlat: [-90 90]
           trimlon: [-180 180]
             frame: 'on'
             ffill: 100
        fedgecolor: [0 0 0]
        ffacecolor: 'none'
         flatlimit: [-90 90]
        flinewidth: 3
         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: 9
         fontunits: 'points'
        fontweight: 'normal'
       labelformat: 'compass'
        labelunits: 'degrees'
     meridianlabel: 'off'
    mlabellocation: 30
    mlabelparallel: 90
       mlabelround: 0
     parallellabel: 'off'
    plabellocation: 15
    plabelmeridian: -180
       plabelround: 0
```

Note that the list of properties includes both those particular to map axes and general ones that apply to all MATLAB axes.

6 Similarly, use the `setm` function alone to display the set of properties, their enumerated values, and defaults:

```
setm(gca)
AngleUnits                    [ {degrees} | radians ]
Aspect                        [ {normal} | transverse ]
FalseEasting
FalseNorthing
FixedOrient                   FixedOrient is a read-only property
Geoid
MapLatLimit
MapLonLimit
MapParallels
```

```
MapProjection
NParallels                    NParallels is a read-only property
Origin
ScaleFactor
TrimLat                       TrimLat is a read-only property
TrimLon                       TrimLon is a read-only property
Zone
Frame                         [ on | {off} ]
FEdgeColor
FFaceColor
FFill
FLatLimit
FLineWidth
FLonLimit
Grid                          [ on | {off} ]
GAltitude
GColor
GLineStyle                    [ - | -- | -. | {:} ]
GLineWidth
MLineException
MLineFill
MLineLimit
MLineLocation
MLineVisible                  [ {on} | off ]
PLineException
PLineFill
PLineLimit
PLineLocation
PLineVisible                  [ {on} | off ]
FontAngle                     [ {normal} | italic | oblique ]
FontColor
FontName
FontSize
FontUnits                    [ inches | centimeters | normalized |
{points} | pixels ]
FontWeight                    [ {normal} | bold ]
LabelFormat                   [ {compass} | signed | none ]
LabelRotation                 [ on | {off} ]
LabelUnits                    [ {degrees} | radians ]
MeridianLabel                 [ on | {off} ]
```

```
MLabelLocation
MLabelParallel
MLabelRound
ParallelLabel                [ on | {off} ]
PLabelLocation
PLabelMeridian
PLabelRound
```

Many, but not all, property choices and defaults can also be displayed individually:

```
setm(gca,'AngleUnits')
AngleUnits                   [ {degrees} | radians ]
setm(gca,'MapProjection')
An axes's "MapProjection" property does not have a fixed set
 of property values.
setm(gca,'Frame')
Frame                        [ on | {off} ]
setm(gca,'FixedOrient')
FixedOrient                  FixedOrient is a read-only property
```

7 In the same way, `getm` displays the current value of any axes property:

```
getm(gca,'AngleUnits')
ans =
degrees

getm(gca,'MapProjection')
ans =
eqdconic

getm(gca,'Frame')
ans =
on

getm(gca,'FixedOrient')
ans =
     []
```

For a complete listing and descriptions of map axes properties, see the reference page for `axesm`. To identify what properties apply to a given map projection, see the reference page for that projection.

Using the Map Limit Properties

In many common situations, the map limit properties, `MapLatLimit` and `MapLonLimit`, provide a convenient way of specifying your map projection origin or frame limits. Note that these properties are intentionally redundant; you can always avoid them if you wish and instead use the `Origin`, `FLatLimit`, and `FLonLimit` properties to set up your map. When they're applicable, however, you'll probably find that it's easier and more intuitive to set `MapLatLimit` and `MapLonLimit`, especially when creating a new map axes with `axesm`.

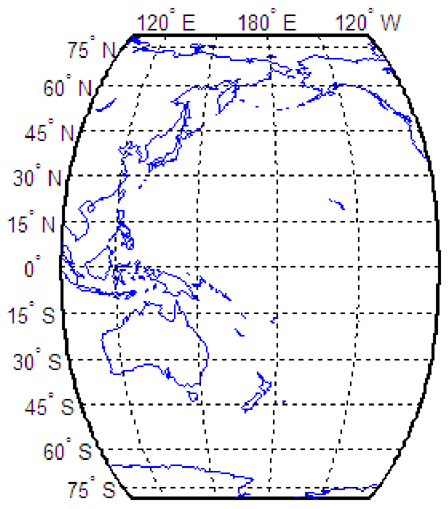
Example 1: Robinson Projection

Often, you'll want to create a map using a cylindrical projection (such as Mercator, Miller, or Plate Carée) or a pseudo-cylindrical projection (such as Mollweide or Robinson) showing all or most of the Earth, with the Equator running as a straight horizontal line across the center of the map. Your map will be bounded by a geographic quadrangle, and the projection origin will be located on the Equator and centered between the longitude limits. In this case, you can easily control the north-south extent of the quadrangle with the `MapLatLimit` property and the east-west extent with the `MapLonLimit` property. `axesm` will automatically set the `Origin` and assign consistent values for the frame limits (`FLatLimit` and `FLonLimit`).

For example, here's a way to create a map with a Robinson projection showing the western Pacific Ocean and surrounding areas:

```
latlim = [-80 80];
lonlim = [100 -120];
figure('Color','white')
axesm('robinson','MapLatLimit',latlim,'MapLonLimit',lonlim, ...
   'Frame','on','Grid','on','MeridianLabel','on', ...
   'ParallelLabel','on')
axis off
setm(gca,'MLabelLocation',60)
coast = load('coast.mat');
```

```
plotm(coast.lat,coast.long)
```



Note The western limit (100 degrees E, in this case) must always precede the eastern limit (-120 degrees E, or 120 degrees W), even if the second number in the longitude-limit vector is smaller than the first.

Note that the map spans 140 degrees from west to east:

```
wrapTo360(diff(lonlim))

ans =
    140
```

`axesm` automatically sets the `Origin` and frame limits based on the values you selected for `MapLatLim` and `MapLonLim`. You can check the `Origin` and frame limits by using `getm`.

```
origin = getm(gca,'Origin');
flatlim = getm(gca,'FLatLimit');
flonlim = getm(gca,'FLonLimit');
```

The origin longitude should be located halfway between the longitude limits of 100 E and 120 W. Adding half of 140 to the western limit gives 100 + 70 = 170 degrees E. This should, and does, equal the second element of the origin vector:

```
origin(2)

ans =
   170
```

The frame is centered on this longitude with a half-width of 70 degrees:

```
flonlim

flonlim =
   -70    70
```

The story with latitudes is somewhat simpler; the origin latitude is on the Equator:

```
origin(1)

ans =
     0
```

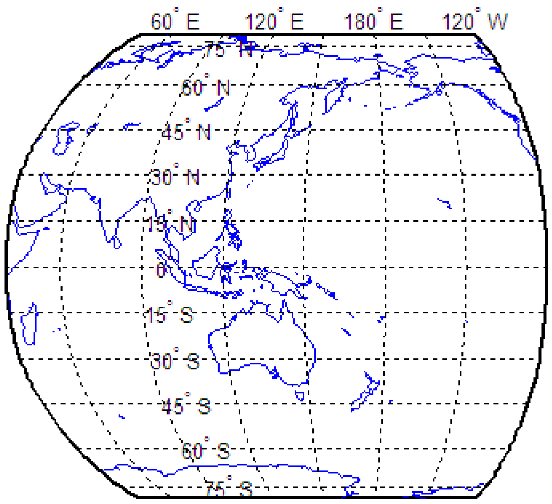
and therefore the latitude limits of the frame equal the value supplied for `MapLatLimit`:

```
flatlim

flatlim =
   -80    80
```

Of course, after you've called `axesm`, you may look at your map and decide that you're not completely satisfied with your initial choice of map limits. Suppose that you decide it would be better to shift the western longitude limit to 40 degrees E in order to include a little more of Asia. You can do this by calling `setm` with a new `MapLonLimit` value:

```
setm(gca,'MapLonLimit',[40 -120])
```

but the asymmetric appearance of the resulting map may surprise you.



You might have expected to see a symmetric map just like the one you would get if you replaced `lonlim` in the earlier call to `axesm` with [40 -120], but that's not what happened. This apparent inconsistency turns out to be an important consequence of the fact that `MapLatLimit` and `MapLonLimit` are redundant properties.

Before you call `axesm`, none of the map axes properties have been set yet because the map axes doesn't exist. Therefore, there's no value yet for the `Origin` property, and there's no problem in setting the longitude origin halfway between the longitudes specified in the `MapLonLimit` vector. But once `axesm` has been called, your map axes does have a projection origin. Since the projection origin is such a fundamental property, it takes precedence over the `MapLonLimit` property.

Therefore, if you try to reset your longitude limits without also resetting the origin, `setm` will maintain your current origin. So, the center of the map limits moved west, but the origin stayed fixed. This combination caused the asymmetry.

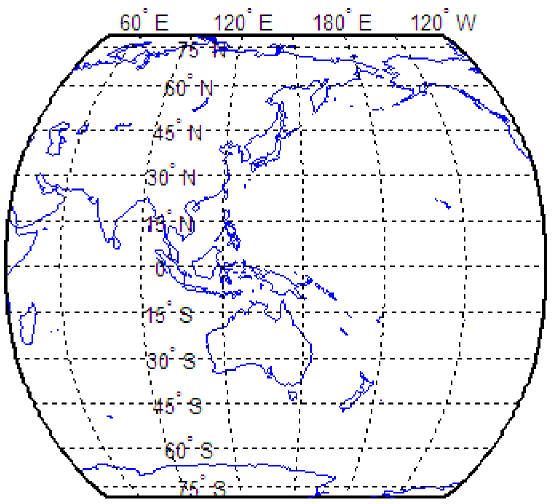
To avoid this asymmetry, you can repeat the operations shown above to figure out that the new central longitude must be at 140 degrees E and add this in the call to `setm` like this:

```
setm(gca,'MapLonLimit',[40 -120],'Origin',[0 140])
```

but you don't actually need to go through such trouble.

Instead, you can just tell `setm` that you'd like to calculate a new origin by providing an empty array instead of a new value for the `Origin` property.

```
setm(gca,'MapLonLimit',[40 -120],'Origin',[])
```



Notice the symmetry of the resulting map frame. Usually this is the easiest thing to do.

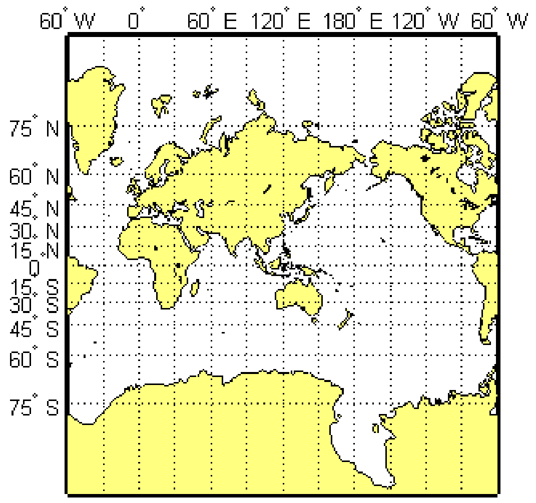
Example 2: Cylindrical Projection

Load the "coast" MAT-file.

```
coast = load('coast');
```

Construct a Mercator projection covering the full range of permissible latitudes with longitudes covering a full 360 degrees starting at 60 West.

```
figure('Color','w')
axesm('mercator','MapLatLimit',[-90 90],'MapLonLimit',[-60 300])
axis off; framem on; gridm on; mlabel on; plabel on;
setm(gca,'MLabelLocation',60)
geoshow(coast.lat,coast.long,'DisplayType','polygon')
```



The call to `axesm` above is equivalent to:

```
axesm('mercator','Origin',[0 120 0], ...
     'FLatLimit',[-90 90],'FlonLimit',[-180 180])
```

You can verify this by checking these properties:

```
getm(gca,'Origin')
getm(gca,'FLatLimit')
getm(gca,'FLonLimit')

ans =
             0        120.00              0
ans =
          -86.00         86.00
ans =
         -180.00        180.00
```

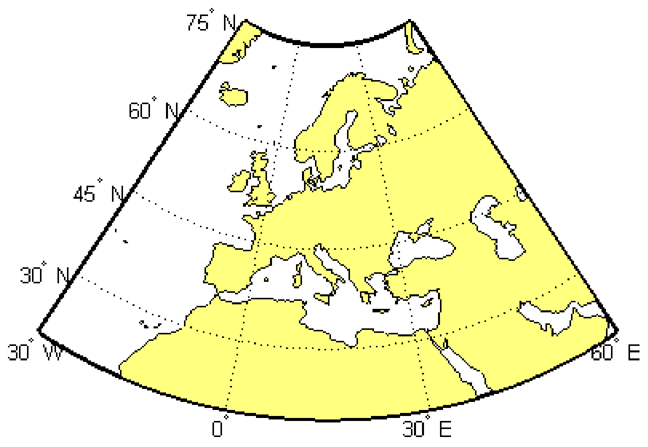
Note that the map and frame limits are clamped to the range of `[-86 86]` imposed by the read-only `TrimLat` property.

```
getm(gca,'MapLatLimit')
getm(gca,'FLatLimit')
getm(gca,'TrimLat')

ans =
          -86.00         86.00
ans =
          -86.00         86.00
ans =
          -86.00         86.00
```

Example 3: Conic Projection

Create a map of the standard version of the Lambert Conformal Conic projection covering latitudes 20 North to 75 North and longitudes covering 90 degrees starting at 30 degrees West.

```
coast = load('coast');
figure('Color','w')
axesm('lambertstd','MapLatLimit',[20 75],'MapLonLimit',[-30 60])
axis off; framem on; gridm on; mlabel on; plabel on;
geoshow(coast.lat, coast.long, 'DisplayType', 'polygon')
```



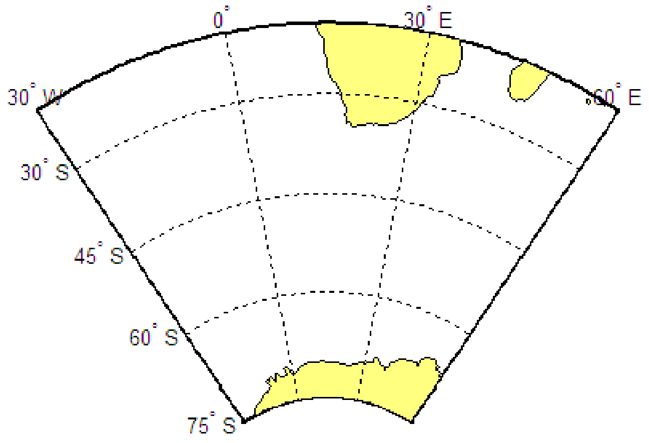
The call to `axesm` above is equivalent to:

```
axesm('lambertstd','Origin',[0 15 0],'FLatLimit',[20 75], ...
  'FlonLimit',[-45 45])
```

Example 4: Southern Hemisphere Conic Projection

"Reflect" the preceding map into the Southern Hemisphere. Override the default standard parallels as well as change `MapLatLimit`.

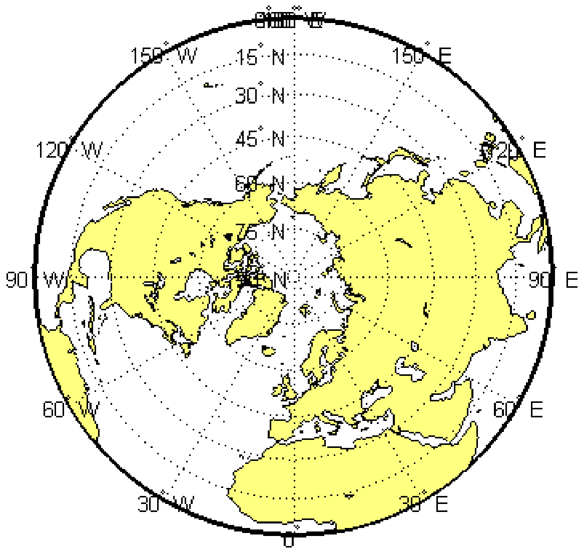
```
coast = load('coast');
figure('Color','w')
axesm('lambertstd','MapParallels',[-75 -15], ...
  'MapLatLimit',[-75 -20],'MapLonLimit',[-30 60])
axis off; framem on; gridm on; mlabel on; plabel on;
geoshow(coast.lat,coast.long,'DisplayType','polygon')
```



Example 5: North-Polar Azimuthal Projection

Construct a North-polar Equal-Area Azimuthal projection map extending from the Equator to the pole and centered by default on longitude 0.

```
coast = load('coast');
figure('Color','w')
axesm('eqaazim','MapLatLimit',[0 90])
axis off; framem on; gridm on; mlabel on; plabel on;
setm(gca,'MLabelParallel',0)
geoshow(coast.lat,coast.long,'DisplayType','polygon')
```



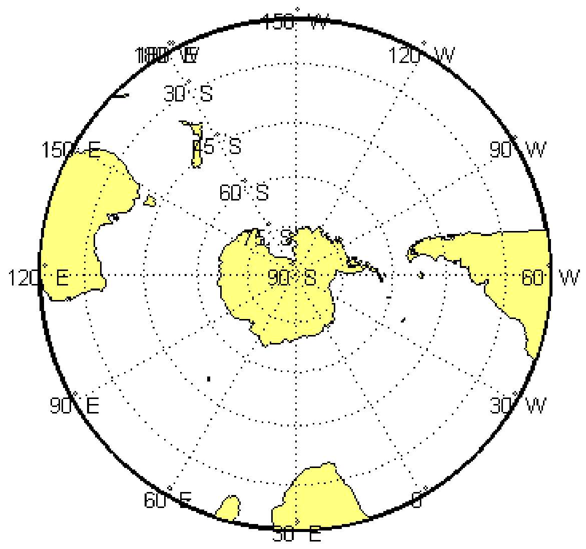
The call to axesm above is equivalent to:

```
axesm('eqaazim','MLabelParallel',0,'Origin',[90 0 0], ...
  'FLatLimit',[-Inf 90])
```

Example 6: South-Polar Azimuthal Projection

Create a South-polar Stereographic Azimuthal projection map extending from the South Pole to 20 degrees S, centered on longitude 150 degrees West. Include a value for the `Origin` property in order to control the central meridian.

```
coast = load('coast');
figure('Color','w')
axesm('stereo','Origin',[-90 -150],'MapLatLimit',[-90 -20])
axis off; framem on; gridm on; mlabel on; plabel on;
setm(gca,'MLabelParallel',-20)
geoshow(coast.lat,coast.long,'DisplayType','polygon')
```



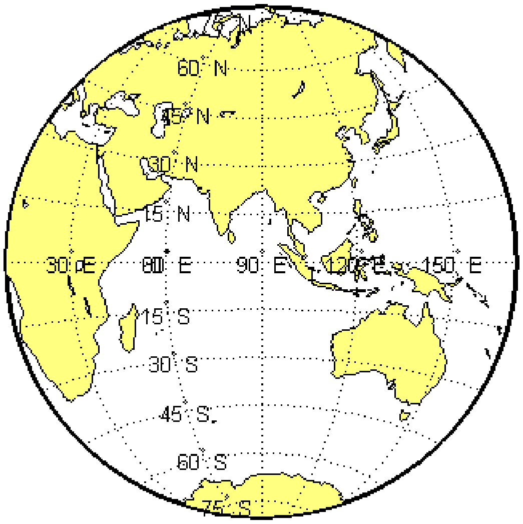
The call to `axesm` above is equivalent to:

```
axesm('stereo','Origin',[-90 -150 0],'FLatLimit',[-Inf 70])
```

Example 7: Equatorial Azimuthal Projection

Create a map of an Equidistant Azimuthal projection with the origin on the Equator, covering from 10 E to 170 E. The origin longitude falls at the center of this range (90 E), and the map reaches north and south to within 10 degrees of each pole.

```
coast = load('coast');
figure('Color','w')
axesm('eqdazim','MapLonLimit',[10 170])
axis off; framem on; gridm on; mlabel on; plabel on;
setm(gca,'MLabelParallel',0,'PLabelMeridian',60)
geoshow(coast.lat,coast.long,'DisplayType','polygon')
```



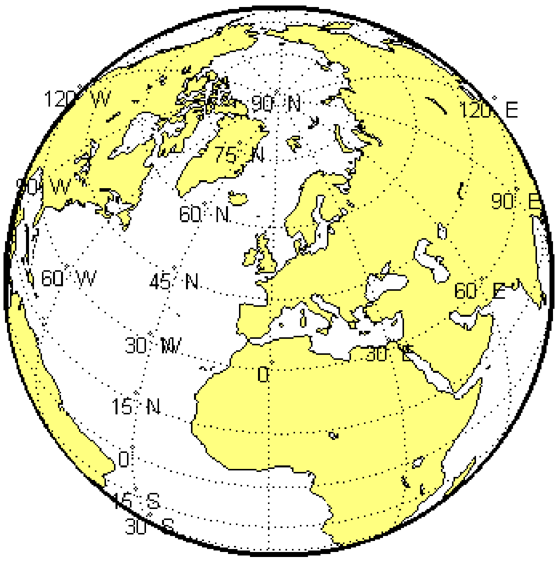
The call to axesm above is equivalent to:

```
axesm('eqaazim','Origin',[0 90 0],'FLatLimit',[-Inf 80])
```

Example 8: General Azimuthal Projection

Construct an Orthographic projection map with the origin centered near Paris. You can't use MapLatLimit or MapLonLimit in this case.

```
coast = load('coast');
originLat = dm2degrees([48 48]);
originLon = dm2degrees([ 2 20]);

figure('Color','w')
axesm('ortho','Origin',[originLat originLon])
axis off; framem on; gridm on; mlabel on; plabel on;
setm(gca,'MLabelParallel',30,'PLabelMeridian',-30)
geoshow(coast.lat,coast.long,'DisplayType','polygon')
```

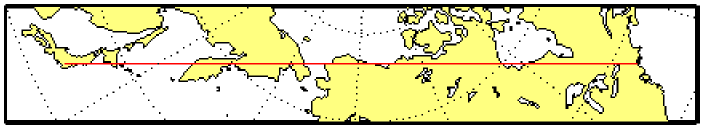


Example 9: Oblique Mercator Projection

Create a map with a long, narrow, oblique Mercator projection showing the area 10 degrees to either side of the great-circle flight path from Tokyo to New York. You can't use `MapLatLimit` or `MapLonLimit` in this case, either.

```
coast = load('coast');
latTokyo = dm2degrees([ 35 40]);
lonTokyo = dm2degrees([139 45]);

latNewYork = dm2degrees([ 40 47]);
lonNewYork = dm2degrees([-73 58]);

[dist,az] = distance(latTokyo,lonTokyo,latNewYork,lonNewYork);
[midLat,midLon] = reckon(latTokyo,lonTokyo,dist/2,az);
midAz = azimuth(midLat,midLon,latNewYork,lonNewYork);

buf = [-10 10];
```

```
figure('Color','w')
axesm('mercator','Origin',[midLat midLon 90-midAz], ...
    'FLatLimit',buf,'FLonLimit',[-dist/2 dist/2] + buf)
axis off; framem on; gridm on; tightmap
geoshow(coast.lat,coast.long,'DisplayType','polygon')
plotm([latTokyo latNewYork],[lonTokyo lonNewYork],'r-')
```



General Applicability of Map Limit Properties

As the preceding examples illustrate, most typically you use the `MapLatLimit` and `MapLonLimit` properties to set up a map axes with a non-oblique, non-azimuthal projection, with its origin on the Equator. (Most of the projections included in the Mapping Toolbox fall into this category; e.g., cylindrical, pseudo-cylindrical, conic, or modified azimuthal.) In addition, even with a non-zero origin latitude (origin off the Equator), you can use the `MapLatLimit` and `MapLonLimit` properties with projections that are implemented directly rather than via rotations of the sphere (e.g., `tranmerc`, `utm`, `lambertstd`, `cassinistd`, `eqaconicstd`, `eqdconicstd`, and `polyconicstd`). This list includes the projections used most frequently for large-scale maps, such as U.S. Geological Survey topographic quadrangle maps. Finally, when the origin is located at a pole or on the Equator, you can use the map limit properties with any azimuthal projection (e.g., `stereo`, `ortho`, `breusing`, `eqaazim`, `eqdazim`, `gnomonic`, or `vperspec`).

On the other hand, you should avoid the map limit properties, working instead with the `Origin`, `FLatLimit`, and `FLonLimit` properties, when:

- You want your map frame to be positioned asymmetrically with respect to the origin longitude.
- You want to use an oblique aspect (that is, assign a non-zero rotation angle to the third element of the "orientation vector" supplied as the `Origin` property value).

- You want to change your projection's default aspect (normal vs. transverse).
- You want to to use a nonzero origin latitude, except in one of the special cases noted above.
- You are using one of the following projections:
  - `globe` — No need for map limits; always covers entire planet
  - `cassini` — Always in a transverse aspect
  - `wetch` — Always in a transverse aspect
  - `bries` — Always in an oblique aspect

There's no need to supply a value for the `MapLatLimit` property if you've already supplied one for the `Origin` and `FLatLimit` properties. In fact, if you supply all three when calling either `axesm` or `setm`, the `FLatLimit` value will be ignored. Likewise, if you supply values for `Origin`, `FLonLimit`, and `MapLonLimit`, the `FLonLimit` value will be ignored.

If you do supply a value for either `MapLatLimit` or `MapLonLimit` in one of the situations listed above, `axesm` or `setm` will ignore it and issue a warning. For example,

```
axesm('lambert','Origin',[40 0],'MapLatLimit',[20 70])
```

generates the warning message:

```
Ignoring value of MapLatLimit due to use of nonzero origin
 latitude with the lambert projection.
```

Using the Map Limit Properties with `setm`

As shown in the earlier example in which the longitude limits of a map in the Robinson projection are changed via `setm`, it's important to understand that `MapLatLimit` and `MapLonLimit` are extra, redundant properties that are coupled to the `Origin`, `FLatLimit`, and `FLonLimit` properties. On the other hand, it's not too difficult to know how to update your map axes if you keep in mind the following:

- The Origin property takes precedence. It is set (implicitly, if not explicitly) every time you call axesm and you cannot change it just by changing the map limits. (Note that when creating a new map axes from scratch, the map limits are used to help set the origin if it is not explicitly specified.)
- MapLatLimit takes precedence over FLatLimit if both are provided in the same call to axesm or setm, but changing either one alone affects the other.
- MapLonLimit and FLonLimit have a similar relationship.

As shown in the example, the precedence of Origin means that if you want to reset your map limits with setm and have setm also determine a new origin, you must set Origin to [] in the same call. For example,

```
setm(gca,'Origin',[],'MapLatLimit',newMapLatlim,'MapLonLimit',newMapLonl
```

On the other hand, a call like this will automatically update the values of FLatLimit and FLonLimit. Similarly, a call like:

```
setm(gca,'FLatLimit',newFrameLatlim,'FLonLimit',newFrameLonlim)
```

will update the values of MapLatLimit and MapLonLimit.

Finally, you probably don't want to try the following:

```
setm(gca,'Origin',[],'FLonLimit',newFrameLonlim)
```

because the value of FLonLimit (unlike MapLonLimit) will not affect Origin, which will merely change to a projection-dependent default value (typically [0 0 0]).

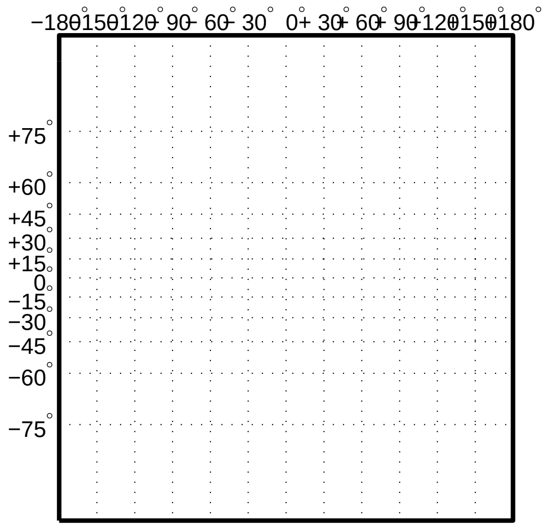
Switching Between Projections

Once a map axes object has been created with axesm, whether map data is displayed or not, it is possible to change the current projection as well as many of its parameters. You can use setm or the maptool UI to reset the projection. The rest of this section describes the considerations and parameters involved in switching projections in a map axes. Additional details are given for doing this with the geoshow function in "Changing Map Projections when Using geoshow" on page 4-40.

When you switch from one projection to another, `setm` clears out settings that were specific to the earlier projection, updates the map frame and graticule, and generally keeps the map covering the same part of the world—even when switching between azimuthal and non-azimuthal projections. But in some cases, you might need to further adjust the map axes properties to achieve proper appearance. Settings that are suitable for one projection might not be appropriate for another. Most often, you'll need to update the positioning of your meridian and parallel labels.

1 Create a Mercator projection with meridian and parallel labels.

```
axesm mercator
framem on; gridm on; mlabel on; plabel on
setm(gca,'LabelFormat','signed')
axis off
```



2 Get the default map and frame latitude limits for the Mercator projection.

```
[getm(gca,'MapLatLimit'); getm(gca,'FLatLimit')]
ans =
-86    86
-86    86
```

Both the frame and map latitude limits are set to 86º north and south for the Mercator projection to maintain a safe distance from the singularity at the poles.

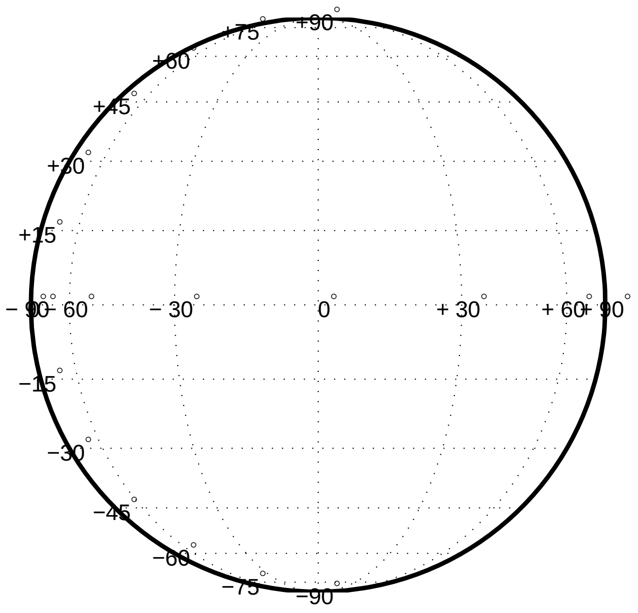
3 Now switch the projection to an orthographic azimuthal.

```
setm(gca,'MapProjection','ortho')
```

4 Manually specify new locations for the meridian and parallel labels. (See “Labeling Grids” on page 4-56.)

```
setm(gca,'MLabelParallel',0,'PLabelMeridian',-90,'PLabelMeridian',-30)
```

Now the map is displayed correctly.



Projected and Unprojected Graphic Objects

Many Mapping Toolbox cartographic functions project features on a map axes based on their designated latitude-longitude positions. The latitudes and longitudes are mathematically transformed to *x* and *y* positions using the formulas for the current map projection. If the map projection or its parameters change, objects on a map axes can be automatically reprojected to update the map display accordingly, but only under the circumstances detailed in the following sections.

Auto-Reprojection of Mapped Objects and Its Limitations

Using the `setm` function, you can change the current map projection on the fly if the map display was created in a way that permits reprojection. Note that map displays can contain objects that cannot be reprojected, and may need to be explicitly deleted and redrawn. Automatic reprojection will take place when you use `setm` to modify the `MapProjection` property, or any other map axes property from the following list:

- `AngleUnits`
- `Aspect`
- `FalseEasting`
- `FalseNorthing`
- `FLatLimit`
- `FLonLimit`
- `Geoid`
- `MapLatLimit`
- `MapLonLimit`
- `MapParallels`
- `Origin`
- `ScaleFactor`
- `TrimLat`
- `TrimLon`
- `Zone`

Auto-reprojection takes place for objects created with any of the following Mapping Toolbox functions:

- `contourm`
- `contour3m`
- `fillm`
- `fill3m`
- `gridm`
- `linem`
- `meshm`

- `patchm`
- `plotm`
- `plot3m`
- `surfm`
- `surfacem`
- `textm`

In general, objects created with `geoshow` or with a combination of calls to `mfwdtran` followed by ordinary MATLAB graphics functions, such as `line`, `patch`, or `surface`, are *not* automatically reprojected. You should delete such objects whenever you change one or more of the map axes properties listed above, and then redisplay them.

The above Mapping Toolbox functions are analogous to standard MATLAB graphics functions having the same name, less the trailing `m`. You can use both types of functions to plot data on a map axes, as long as you are aware that the standard MATLAB graphics functions do not apply map projection transformations, and therefore require you to specify positions in map *x-y* space.

If you have preprojected vector or raster map data or read such data from files, you can display it with `mapshow`, `mapview`, or standard MATLAB graphics functions, such as `plot` or `mesh`. If its projection is known and is included in the Mapping Toolbox projection libraries, you can use its parameters to project geodata in geographic coordinates to display it in the same axes. For additional information, see "Using Cartesian MATLAB Display Functions" on page 6-28.

There are four common use cases for changing a map projection in a map axes with `setm` or for reprojecting map data plotted on a regular MATLAB axes:

Mapping Use Case	Type of Axes	Reprojection Behavior
Plot geographic (*latitude-longitude*) vector coordinate data or data grid using a Mapping Toolbox function from releases prior to Version 2 (e.g., `plotm`)	Map axes	Automatic reprojection
Plot geographic vector data with `geoshow`	Map axes	No automatic reprojection; delete graphics objects prior to changing the projection and redraw them afterwards.
Plot data grids, images, and contours with geographic coordinates with `geoshow`	Map axes	Automatic reprojection; this behavior could change in a future release
Plot projected (*x*-*y*) vector or raster map data with `mapshow` or with a MATLAB graphics function (e.g., `line`, `contour`, or `surf`)	Regular axes	Manual reprojection (reproject coordinates with `minvtran` /`mfwdtran` or `projinv`/`projfwd`); delete graphics objects prior to changing the projection and redraw them afterwards.

You can use `handlem` to help identify which objects to delete when manual deletion is necessary. See "Determining and Manipulating Object Names" on page 4-82 for an example of its use. The following section describes reprojection behavior in more detail and illustrates some of these cases.

Changing Map Projections when Using geoshow

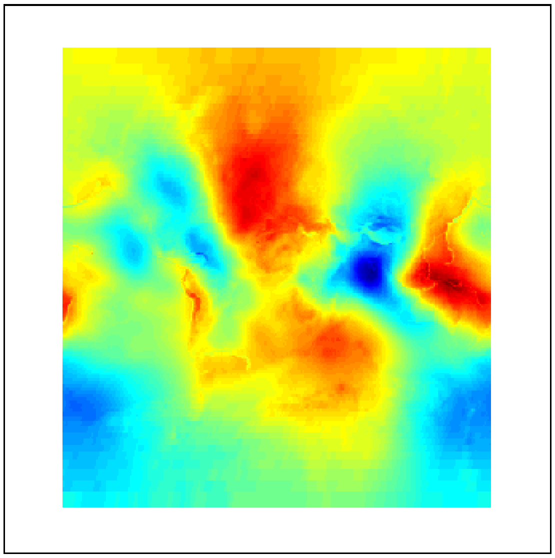
You can display latitude-longitude vector and raster geodata using the `geoshow` function (use `mapshow` to display preprojected coordinates and grids). When you use `geoshow` to display maps on a map axes, the data are projected according to the map projection assigned when `axesm`, `worldmap`, or `usamap` created the map axes (e.g., `axesm('mapprojection','mercator')`).

You can also use `geoshow` to display latitude-longitude data on a regular axes (created by the `axes` function, for example). When you do this, the latitude-longitude data are displayed using a `Plate CarrØe Projection`, which linearly maps longitude to *x* and latitude to *y*.

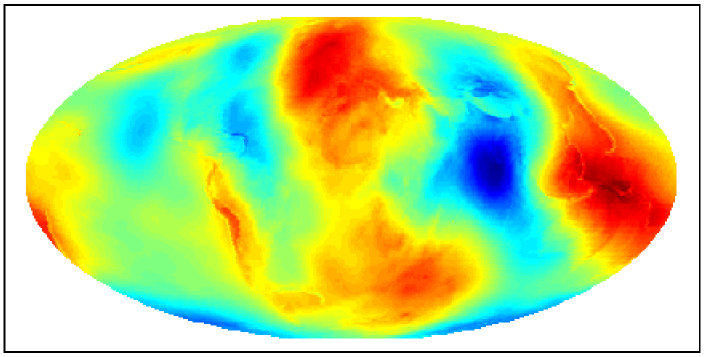
If you are using `geoshow` with a map axes and want to change the map projection after you have displayed data in geographic coordinates, do the following, depending on whether the data are raster or vector:

Raster Data. Change the projection using `setm`. For example,

```
load geoid
figure; axesm mercator
geoshow(geoid,geoidrefvec,'DisplayType','texturemap')
```

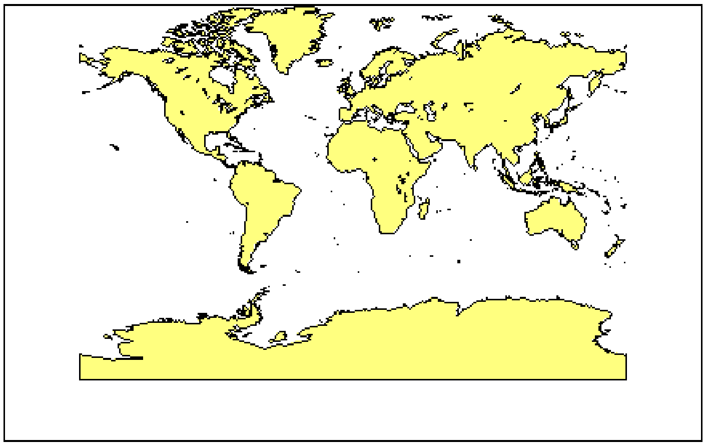


```
setm(gca,'mapprojection','mollweid')
```

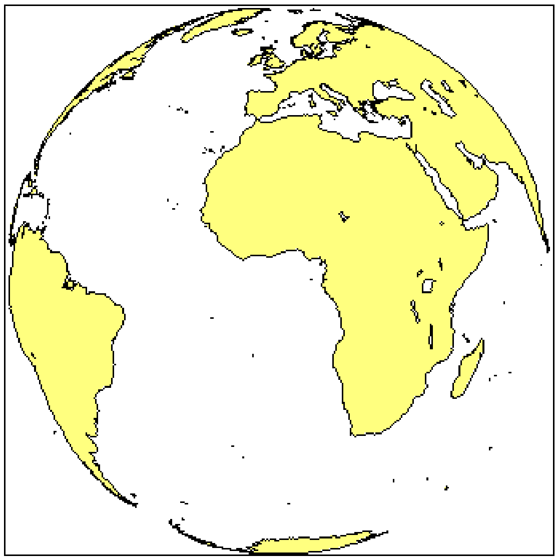


Vector Data. Obtain handles to the line or patch graphic objects, delete the objects from the axes, change the projection using `setm`, and replot the vector data using `geoshow`:

```
figure; axesm miller
h = geoshow('landareas.shp')
```



```
delete(h)
setm(gca,'mapprojection','ortho')
geoshow('landareas.shp')
```



In the above example, h is a handle to an hggroup object, which geoshow constructs when plotting point, line, and polygon data.

If you need to change projections when displaying both raster and vector geodata, you can combine these techniques; removing the vector graphic objects does not affect raster data already displayed.

Placing Geographic and Nongeographic Objects in a Map Axes

Here is an example of how the two types of functions can interact when you place text objects:

1 Make a Miller map axes with a latitude-longitude grid:

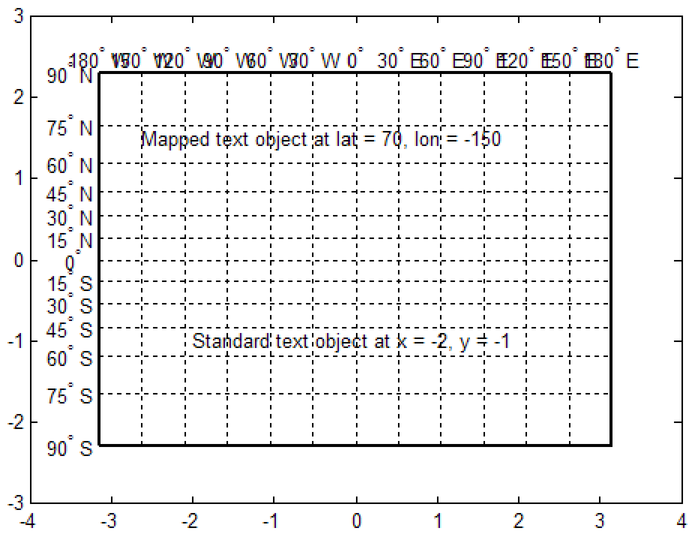
```
axesm miller; framem on; gridm on; mlabel on; plabel on;
showaxes; grid off;
```

These function calls create a map axes object, a map frame enclosing the region of interest, and geographic grid lines. The *x-y* axes, which are normally hidden, are displayed, and the axes `x-y` grid is turned off. The Mapping Toolbox function `gridm` constructs lines to illustrate the latitude-longitude grid, unlike the MATLAB function `grid`, which draws an *x-y* grid for the underlying projected map coordinates. Depending on the type of projection, a latitiude-longitude grid (or *graticule*) can contain curves while a MATLAB grid never does. For more information about graticules, see "The Map Grid" on page 4-53.

2 Now place a standard MATLAB text object and a mapped text object, using the two separate coordinate systems:

```
text(-2,-1,'Standard text object at x = -2, y = -1')
textm(70,-150,'Mapped text object at lat = 70, lon = -150')
```

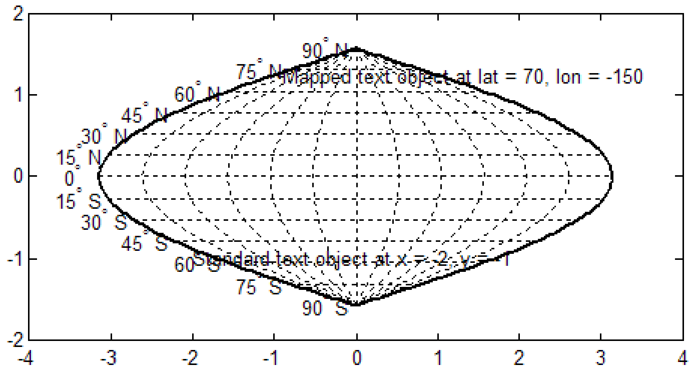
In the figure, shown below, a standard text object is placed at `x=-2` and `y=-1`, while a mapped text object is placed at (70ºN,150ºW) in the Miller projection.



3 Now change the projection to sinusoidal. The standard text object remains at the same Cartesian position, which alters its latitude-longitude position.

The mapped text object remains at the same geographic location, so its *x-y* position is altered. Also, the frame and grid lines reflect the new map projection:

```
setm(gca,'MapProjection','sinusoid')
showaxes; grid off; mlabel off
```



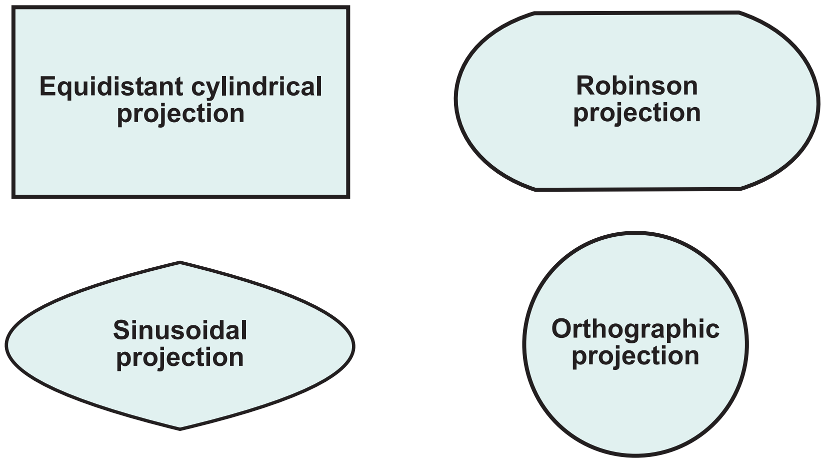
Similarly, vector and matrix data can be displayed using either mapping or standard functions (e.g., `plot`/`plotm`, `surf`/`surfm`). See "Displaying Vector Data with Mapping Toolbox Functions" on page 4-58 for information on plotting vector geodata, and "Displaying Data Grids" on page 4-68 for information on plotting raster geodata.

Controlling Map Frames and Grids

In this section...

The Map Frame

The Mapping Toolbox *map frame* is the outline of the limits of a map, often in the form of a *box*, the "edge of the world," so to speak. The frame is displayed if the map axes property `Frame` is set to `'on'`. This can be accomplished upon map axes creation with `axesm`, or later with `setm`, or with the direct command `framem on`. The frame is geographically defined as a latitude-longitude quadrangle that is projected appropriately. For example, on a map of the world, the frame might extend from pole to pole and a full 360º range of longitude. In appearance, the frame would take on the characteristic shape of the projection. The examples below are full-world frames shown in four very different projections.

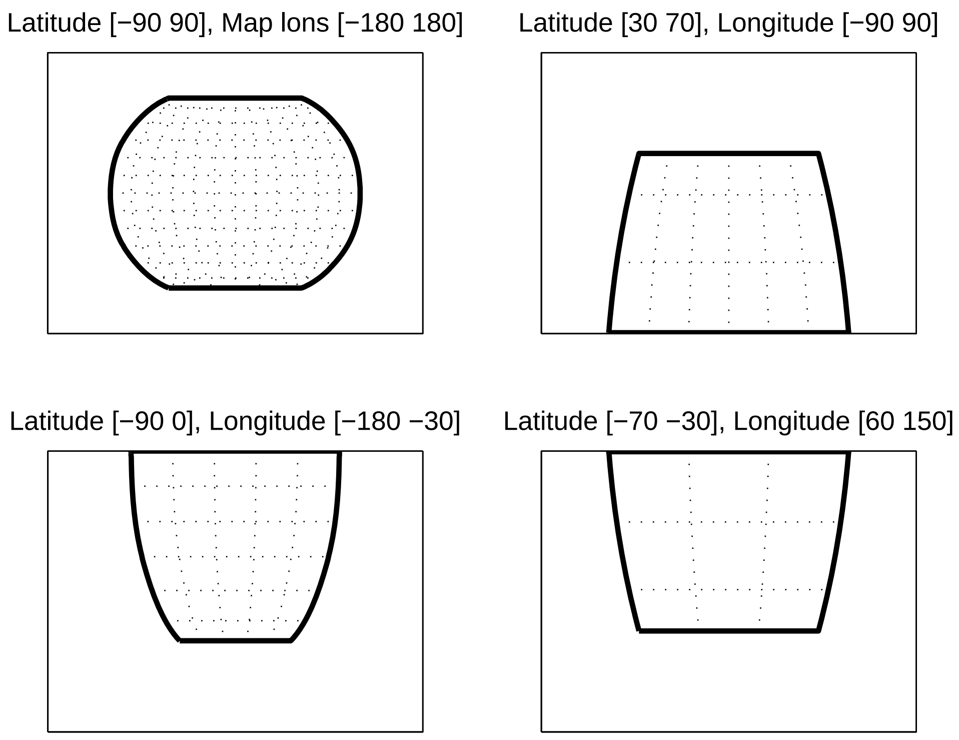


Full-World Map Frames

As a map object, each of the previously displayed frames is identical; however, the selection of a display projection has varied their appearance. Because each of the examples shows the entire world, `FLatLimit` is `[-90 90]`, and

`FLonLimit` is `[-180 180]` for each case. The frame quadrangle can encompass smaller regions, as well, in which case the shape is a section of a full-world outline or simply a quadrilateral with straight or curving sides. Execute this code to produce the figure that follows:

```
% Plot four regions of Robinson frame and grid using map limits
%
figure('color','white')
% Default map frame
subplot(2,2,1);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
title('Latitude [-90 90], Map lons [-180 180]','FontSize',10)
%
subplot(2,2,2);
axesm('MapProjection','robinson',...
    'MapLatLimit',[30 70],'MapLonLimit',[-90 90],...
    'Frame','on','Grid','on')
title('Latitude [30 70], Longitude [-90 90]','FontSize',10)
%
subplot(2,2,3);
axesm('MapProjection','robinson',...
    'MapLatLimit',[-90 0],'MapLonLimit',[-180 -30],....
    'Frame','on','Grid','on')
title('Latitude [-90 0], Longitude [-180 -30]','FontSize',10)
%
subplot(2,2,4);
axesm('MapProjection','robinson',...
    'MapLatLimit',[-70 -30],'MapLonLimit',[60 150],...
    'Frame','on','Grid','on')
title('Latitude [-70 -30], Longitude [60 150]','FontSize',10)
```


Latitude [−90 90], Map lons [−180 180]
Latitude [30 70], Longitude [−90 90]
Latitude [−90 0], Longitude [−180 −30]
Latitude [−70 −30], Longitude [60 150]

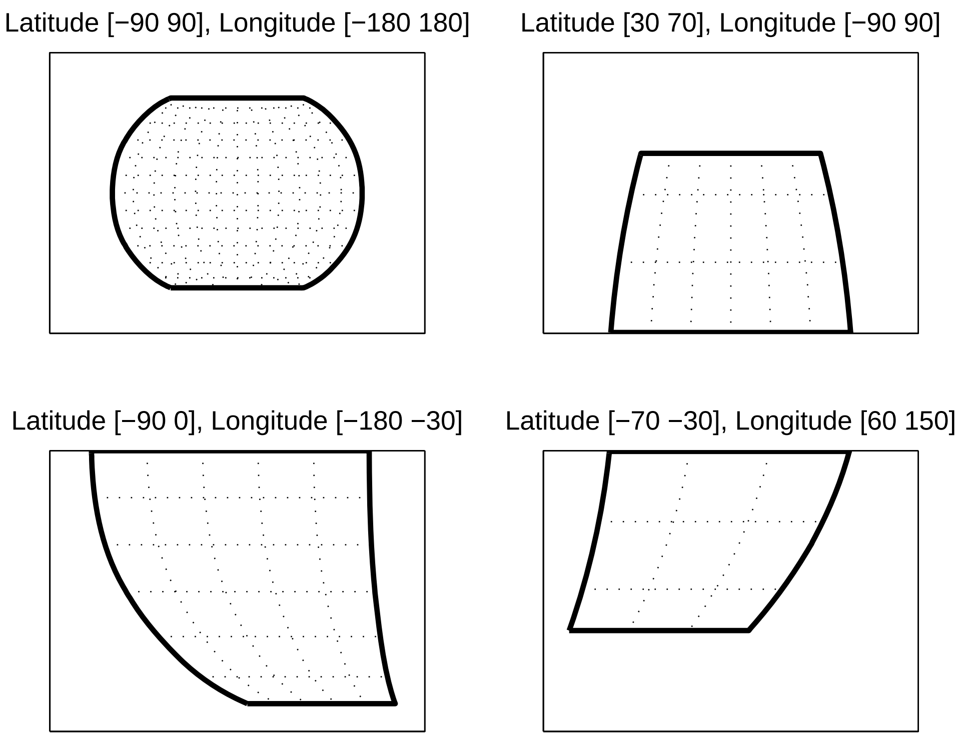
Frame Quadrangles in the Robinson Projection (Symmetric About Prime Meridian)

For the frames shown above, the projection is centered on the prime meridian, or 0 longitude. Such a frame would be the result of creating a map axes with the defaults for the Robinson projection and then resetting the frame limits to cover just part of the world.

When you want your frame to be symmetric about the region of interest, let `axesm` determine the proper settings for you. If you specify the map limits without specifying the map origin and frame limits, `axesm` will automatically set the appropriate values for a proper symmetric frame.

In the following example, the axes limits are set using `setm` after the Robinson map axes is created. Note that map axes properties that concern frames begin with "F":

```
% Same regions as above, but with frame limits
%    altered after projecting
%
figure('color','white')
% Default frame limits
h11 = subplot(2,2,1);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
title('Latitude [-90 90], Longitude [-180 180]')
%
h12 = subplot(2,2,2);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
setm(h12,'FLatLimit',[30 70],'FLonLimit',[-90 90])
title('Latitude [30 70], Longitude [-90 90]')
%
h21 = subplot(2,2,3);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
setm(h21,'FLatLimit',[-90 0],'FLonLimit',[-180 -30])
title('Latitude [-90 0], Longitude [-180 -30]')
%
h22 = subplot(2,2,4);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
setm(h22,'FLatLimit',[-70 -30],'FLonLimit',[60 150])
title('Latitude [-70 -30], Longitude [60 150]')
```


Latitude [−90 90], Longitude [−180 180]
Latitude [30 70], Longitude [−90 90]
Latitude [−90 0], Longitude [−180 −30]
Latitude [−70 −30], Longitude [60 150]

Frame Quadrangles in the Robinson Projection (Symmetric About Map Limits)

The differences between the two examples are obvious when projections are not centered on the prime meridian. If you wanted to create a symmetric frame in the lower right subplot of the above figure, reset the map limits instead of the frame limits, but be sure to reset the `'Origin'` property in the same call:

```
setm(h22,'MapLonLimit',[60 150],'Origin',[])
```

You can manipulate properties beyond the latitude and longitude limits of the frame. Frame properties are established upon map axes object creation; you can modify them subsequently with the `setm` and the `framem` functions. The command `framem` alone is a toggle for the `Frame` property, which controls the visibility of the frame. You can also call `framem` with property names and values to alter the appearance of the frame:

```
framem('FlineWidth',4,'FEdgeColor','red')
```

The frame is actually a patch with a default face color set to `'none'` and a default edge color of black. You can alter these map axes properties by manipulating the `FFaceColor` and `FEdgeColor` properties. For example, the command

```
setm(gca,'FFaceColor','cyan')
```

makes the background region of your display resemble water. Since the frame patch is always the lowest layer of a map display, other patches, perhaps representing land, will appear above the "water." If an object is subsequently plotted "below" the frame patch, the frame altitude can be recalculated to lie below this object with the command `framem reset`. The frame is replaced and not reprojected.

Set the line width of the edge, which is 2 points by default, using the `FLineWidth` property.

The primary advantage of displaying the map frame is that it can provide positional context for other displayed map objects. For example, when vector data of the coasts is displayed, the frame provides the "edge" of the world.

See the `framem` reference page for more details.

Map and Frame Limits

The Mapping Toolbox map and frame limits are two related map axes properties that limit the map display to a defined region. The map latitude and longitude limits define the extents of geodata to be displayed, while the frame limits control how the frame fits around the displayed data. Any object that extends outside the frame limits is automatically trimmed.

The frame limits are also specified differently from the map limits. The map limits are in absolute geographic coordinates referenced to an origin at the intersection of the prime meridian and the equator, while the frame limits are referenced to the rotated coordinate system defined by the map axes origin.

For all nonazimuthal projections, frame limits are specified as quadrangles ([*latmin latmax*] and [*longmin longmax*]) in the frame coordinate system. In the case of azimuthal projections, the frames are circular and are described by a polar coordinate system. One of the frame latitude limits must be a negative infinity (`-Inf`) to indicate an azimuthal frame (think of this as the center of the circle), while the other limit determines the radius of the circular frame (*rlatmax*). The longitude limits of azimuthal frames are inconsequential, since a full circle is always displayed.

If you are uncertain about the correct format for a particular projection frame limit, you can reset the formats to the default values using empty matrices.

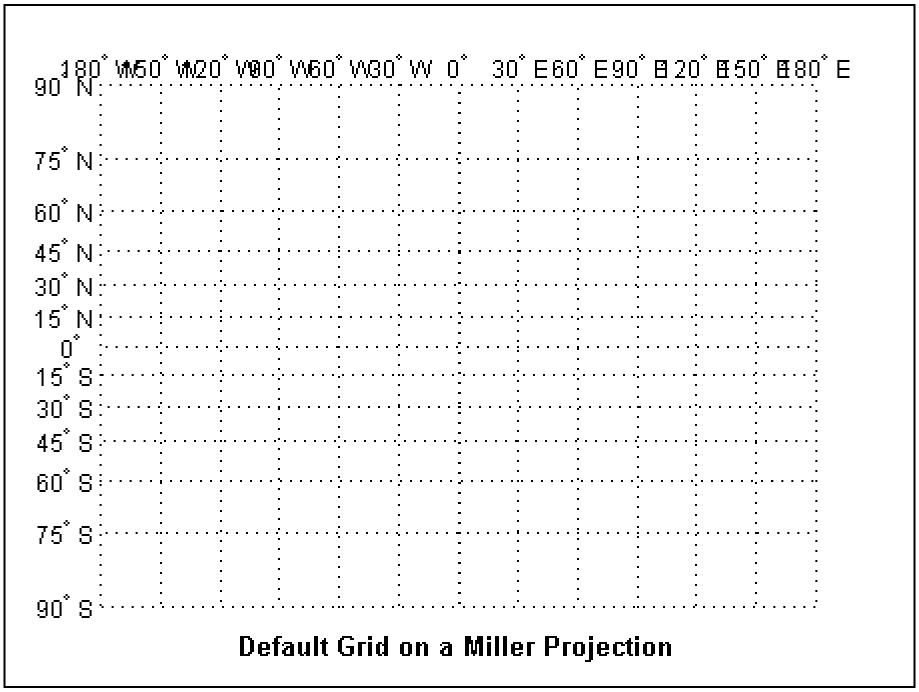
Note For nonazimuthal projections in the normal aspect, the map extent is limited by the minimum of the map limits and the frame limits; hence, the two limits will coincide after evaluation. Therefore, if you manually change one set of limits, you might want to clear the other set to get consistent limits.

The Map Grid

The *map grid* is the set of displayed meridians and parallels, also known as a *graticule*. Display the grid by setting the map axes property `Grid` to `'on'`. You can do this when you create map axes with `axesm`, with `setm`, or with the direct command `gridm on`.

Grid Spacing

To control display of meridians and parallels, set a scalar meridian spacing or a vector of desired meridians in the `MLineLocation` property. The property `PLineLocation` serves a corresponding purpose for parallels. The default values place grid lines every 30º for meridians and every 15º for parallels.



Grid Layering

By default, the grid is placed as the top layer of any display. You can alter this by changing the `GAltitude` property, so that other map objects can be placed “above” the grid. The new grid is drawn at its new altitude. The units used for `GAltitude` are specified with the `daspectm` function.

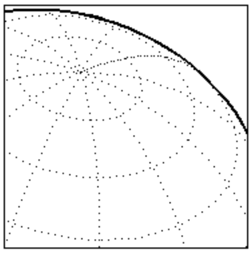
To reposition the grid back to the top of the display, use the command `gridm reset`. You can also control the appearance of grid lines with the `GLineStyle` and `GLineWidth` properties, which are `':'` and `0.5`, respectively, by default.

Limiting Grid Lines

The Miller projection is an example in which all the meridians can extend to the poles without appearing to be cluttered. In other projections, such as the orthographic (below), the map grid can obscure the surface where they converge. Two map axes properties, `MLineLimit` and `MLineException`, enable you to control such clutter:

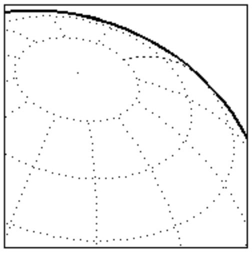
- Use the `MLineLimit` property to specify a pair of latitudes at which to terminate the meridians. For example, setting `MLineLimit` to `[-75 75]` completely clears the region above and below this latitude range of meridian lines.
- If you want some lines to reach the poles but not others, you can specify them with the `MLineException` property. For example, if `MLineException` is set to `[-90 0 90 180]`, then the meridians corresponding to the four cardinal longitudes will continue past the limit on to the pole.

The use of these properties is illustrated in the figure below. Note that there are two corresponding map axes properties, `PLineLimit` and `PLineException`, for controlling the extent of displayed parallels.



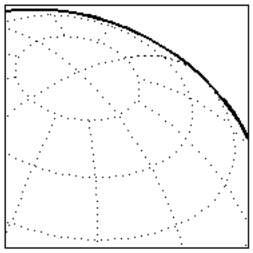
Default grid allows all displayed meridians to extend to the poles:

```
axesm('MapProjection','ortho',...
  'Origin',[40,40,14],...
  'Grid','on','Frame','on');
```



The property `MLineLimit` truncates meridians at given latitudes:

```
axesm('MapProjection','ortho',...
  'Origin',[40,40,14],...
  'Grid','on','Frame','on',...
  'MLineLimit', [-75 75]);
```



The property `MLineLineException` permits certain meridians to extend to the poles, regardless of `MLineLimit`:

```
axesm('MapProjection','ortho',...
  'Origin',[40,40,14],...
  'Grid','on','Frame','on',...
  'MLineLimit', [-75 75],...
  'MLineException',[-90 0 90 180]);
```

Labeling Grids

You can label displayed parallels and meridians. `MeridianLabel` and `ParallelLabel` are on-off properties for displaying labels on the meridians and parallels, respectively. They are both `'off'` by default. Initially, the label locations coincide with the default displayed grid lines, but you can alter this by using the `PlabelLocation` and `MlabelLocation` properties. These grid lines are labeled across the north edge of the map for meridians and along the west edge of the map for parallels. However, the property `MlabelParallel` allows you to specify `'north'`, `'south'`, `'equator'`, or a specific latitude at which to display the meridian labels, and `PlabelMeridian` allows the choice of `'west'`, `'east'`, `'prime'`, or a specific longitude for the parallel labels. By default, parallel labels are displayed in the range of 0° to 90° north and south of the equator while meridian labels are displayed in the range of 0° to

180° east and west of the prime meridian. You can use the `mlabelzero22pi` function to redisplay the meridian labels in the range of 0° to 360° east of the prime meridian.

Properties affecting grid labeling are listed below.

Property	**Effect**
`MeridianLabel`	Toggle display of meridian labels
`ParallelLabel`	Toggle display of parallel labels
`MlabelLocation`	Alternate interval for labeling meridians
`PlabelLocation`	Alternate interval for labeling parallels
`MlabelParallel`	Keyword or latitude for placing meridian labels
`PlabelMeridian`	Keyword or longitude for placing parallel labels
`mlabelzero22pi`(function)	Relabel meridians with positive angle from 0° to 360°

For complete descriptions of all map axes properties, refer to the `axesm` reference page.

Displaying Vector Data with Mapping Toolbox Functions

In this section...

Programming and Scripting Map Construction

Although `mapview`, `maptool`, and other Mapping Toolbox GUIs are convenient and quick tools for making maps, most mapping applications require additional effort. By using and combining Mapping Toolbox and MATLAB functions, you can create and customize more elaborate maps interactively by entering commands in the Command Window or by writing M-code in functions and scripts. This section describes how to use the principal mapping functions for displaying vector geospatial data. The following section describes displaying raster map data.

Displaying Vector Data as Points and Lines

Mapping Toolbox vector map display of line objects works much like MATLAB line display functions. Mapping Toolbox line graphics functions have MATLAB analogs, the names of which can usually be determined by appending an `m` to the MATLAB function name. For instance, the Mapping Toolbox version of `plot` is `plotm`. The main difference between the two classes of functions comes from the need for Mapping Toolbox functions to work with geographic coordinates and map projections.

The following table lists the available Mapping Toolbox line display functions.

Function	Used For
`contourm`	Contour plot of map data
`contour3m`	Contour plot of map data in 3-D space
`geoshow`	High-level function to plot points, lines, patches, grids, and georeferenced images in geocoordinates

Function	Used For
`linem`	Draws line objects projected on map axes
`mapshow`	High-level function to plot points, lines, patches, grids, and georeferenced images in plane coordinates
`plotm`	Clears figure and draws line objects projected on map axes
`plot3m`	Projects lines on map axes in 3-D space

The following exercise shows how some of these functions work:

1 Set up a map axes and frame:

```
load coast
axesm mollweid
framem('FEdgeColor','blue','FLineWidth',0.5)
```

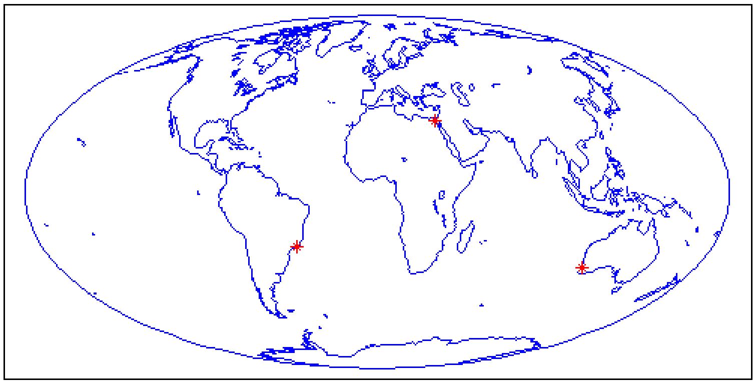
2 Plot the `coast` vector data using `plotm`. Just as with `plot`, you can specify line property names and values in the command.

```
plotm(lat,long,'LineWidth',1,'Color','blue')
```

Sometimes vector data represents specific points. Suppose you have variables representing the locations of Cairo (30ºN,32ºE), Rio de Janeiro (23ºS,43ºW), and Perth (32ºS,116ºE), and you want to plot them as markers only, without connecting line segments.

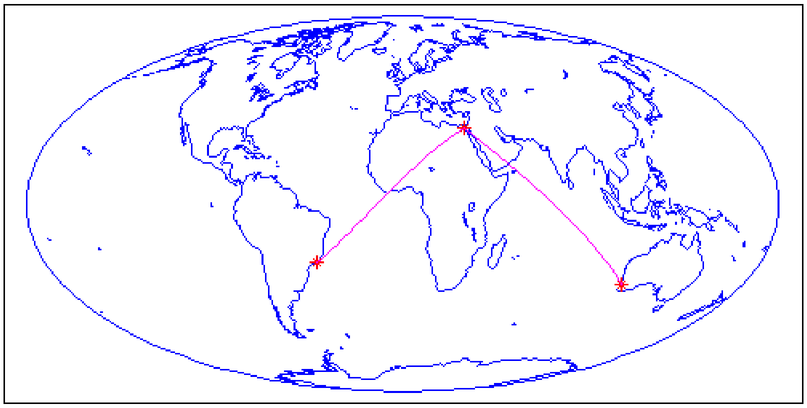
3 Define the three city geographic locations and plot symbols at them:

```
citylats = [30 -23 -32]; citylongs = [32 -43 116];
plotm(citylats,citylongs,'r*')
```



4 In addition to these sorts of “permanent” geographic data, you can also display calculated vector data. Calculate and plot a great circle track from Cairo to Rio de Janeiro, and a rhumb line track from Cairo to Perth:

```
[gclat,gclong] = track2('gc',citylats(1),citylongs(1),...
                        citylats(2),citylongs(2));
[rhlat,rhlong] = track2('rh',citylats(1),citylongs(1),...
                        citylats(3),citylongs(3));
plotm(gclat,gclong,'m-'); plotm(rhlat,rhlong,'m-')
```



Note You can also use `geoshow` (for data in geographic coordinates) or `mapshow` (for data in projected coordinates) to create such maps, either in a map axes or in a regular axes. Both functions accept either vectors of coordinates or geographic data structures as input data.

For more information, see "Mapping Toolbox Geographic Data Structures" on page 2-16, which includes examples of creating geostructs and displaying their contents in "How to Construct Geostructs and Mapstructs" on page 2-20.

Displaying Vector Maps as Lines or Patches

Vector map data that is properly formatted (i.e., as closed polygons) can be displayed as patches, or filled-in polygons. In addition, it and other vector data can be displayed as lines.

Note The Mapping Toolbox patch display functions differ from their MATLAB equivalents by allowing you to display patch vector data that uses NaNs to separate closed regions.

Vector map data for lines or polygons can be represented by simple coordinate arrays, geostructs, or mapstructs. This example illustrates the use of coordinate arrays for both line and polygon features as well as a geostruct containing line features.

1 The `conus` (conterminous U.S.) MAT-file nicely illustrates how polygon data is structured, manipulated, and displayed. Use `who` to see what it contains before loading it.

```
who -file conus.mat

Your variables are:
description  gtlakelon    statelat     uslat
gtlakelat    source       statelon     uslon

load conus
```

The variables `uslat` and `uslon` together describe three polygons (separated by `NaN`s), the largest of which represents the outline of the conterminous United States. The two smaller polygons represent Long Island, NY, and Martha's Vineyard, an island off Massachusetts. The variables `gtlakelat` and `gtlakelon` describe three polygons (separated by `NaN`s) for the Great Lakes. The variables `statelat` and `statelon` contain line-segment data (separated by `NaN`s) for the borders between states, which is not formatted for patch display.

2 Verify that line and polygon data contains `NaN`s (hence multiple objects) by typing a command similar to `find(isnan(vector))`:

```
find(isnan(gtlakelon)) %or gtlakelat
ans =

         883
        1058
        1229
```

The `find` command returns three values indicating that the `gtlakelon` (or `gtlakelat`) geographic coordinate arrays contain three polygons representing one or a group of Great Lakes.

3 Read the `worldrivers` shapefile for the region that covers the conterminous United States. This data, stored as a geographic data structure, is useful for illustrating lines.

```
uslatlim = [min(uslat) max(uslat)]
uslatlim =

   25.1200   49.3800

uslonlim = [min(uslon) max(uslon)]
uslonlim =

 -124.7200  -66.9700

rivers = shaperead('worldrivers', 'UseGeoCoords', true, ...
    'BoundingBox', [uslonlim', uslatlim'])
rivers =
23x1 struct array with fields:
    Geometry
    BoundingBox
    Lon
    Lat
    Name
```

4 The struct `rivers` is a geographic data structure having five fields. Note that the `Geometry` field specifies whether the data is stored as a `'Point'`, `'MultiPoint'`, `'Line'`, or a `'Polygon`':

```
rivers(1).Geometry

ans =
    Line
```

For further details on Mapping Toolbox geographic data structures, see "Understanding Vector Geodata" on page 2-13 and "Understanding Raster Geodata" on page 2-33.

5 Now you can set up a map axes to display the state coordinates. As conic projections are appropriate for mapping the entire United States, create a map axes object using an Albers equal-area conic projection (`'eqaconic'`). Specifying map limits that contain the region of interest automatically centers the projection on an appropriate longitude; the frame encloses just the mapping area, not the entire globe. As a general rule, you should specify map limits that extend slightly outside your area of interest (`worldmap` and `usamap` do this for you).

Note Conic projections need two standard parallels (latitudes at which scale distortion is zero). A good rule is to set the standard parallels at one-sixth of the way from both latitude extremes. Or, to use default latitudes for the standard parallels, simply provide an empty matrix in the call to `axesm`.

The three options that follow demonstrate how you can set map latitude and longitude limits to `axesm`:

a Obtain default latitudes by providing an empty matrix as the standard parallels:

```
figure
axesm('MapProjection','eqaconic', 'MapParallels',[],...
      'MapLatLimit',[23 52], 'MapLonLimit',[-130 -62])
```

b If you do not know what latitude and longitude limits are appropriate for your map, as a starting point you could use the exact ones that the geostruct contains. Using them eliminates white space around the map:

```
axesm('MapProjection','eqaconic', 'MapParallels',[],...
      'MapLatLimit',uslatlim, 'MapLonLimit',uslonlim)
```

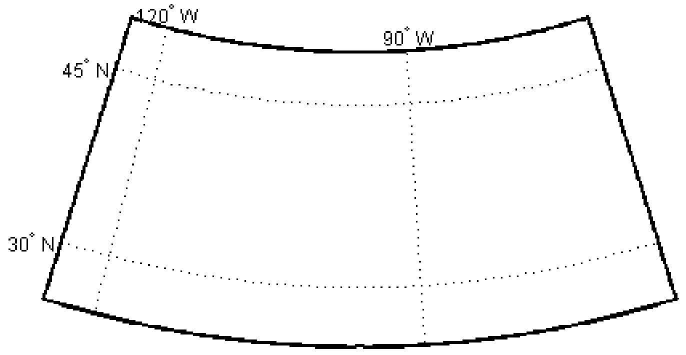
c If you want to add white space around the map, you can do so as follows (here, 2 degrees are added):

```
axesm('MapProjection', 'eqaconic', 'MapParallels', [], ...
      'MapLatLimit', uslatlim + [-2 2], ...
      'MapLonLimit', uslonlim + [-2 2])
```

6 Turn on the map frame, the map grid, and the meridian and parallel labels:

```
axis off; framem; gridm; mlabel; plabel
```

The empty map looks like this.



7 When geographic data is displayed, some layers can hide others. You can control the visibility of your map layers by varying the order in which you

display them. For example, some U.S. state boundaries follow major rivers, so display the rivers last to avoid obscuring the rivers with the boundaries.

The coordinate array pairs (`uslat`, `uslon`), (`gtlakelat`, `gtlakelon`), and (`statelat`, `statelon`) simply contain sequences of `NaN`-separated map segments, and their geometric interpretation is ambiguous. In order to display them appropriately as either patches or lines with `geoshow`, you need to use the `DisplayType` parameter. In contrast, `DisplayType` is not needed when you map data from a geostruct like `rivers`.

a Plot a patch to display the area occupied by the conterminous United States; use the `geoshow` function with a `'polygon'` `DisplayType`:

```
geoshow(uslat,uslon, 'DisplayType','polygon','FaceColor',...
    [1 .5 .3], 'EdgeColor','none')
```

b Plot the Great Lakes on top of the land area, using `geoshow` again:

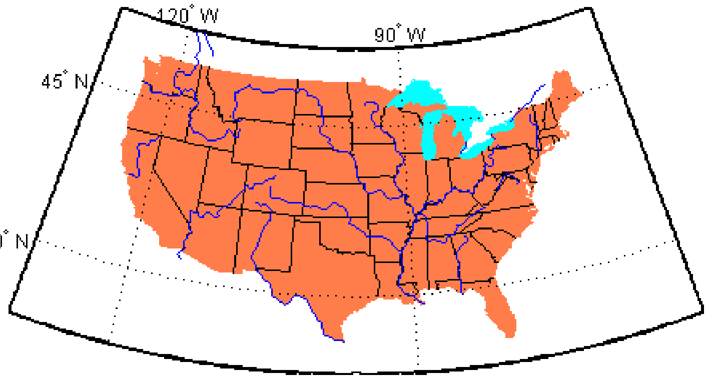
```
geoshow(gtlakelat,gtlakelon, 'DisplayType','polygon',...
    'FaceColor','cyan', 'EdgeColor','none')
```

c Plot the line segment data showing state boundaries, using `geoshow` with a `'line'` `DisplayType`:

```
geoshow(statelat,statelon,'DisplayType','line','Color','k')
```

d Finally, use `geoshow` to plot the river network. Note that you can omit `DisplayType`:

```
geoshow(rivers, 'Color', 'blue')
```



Summary of Polygon Mapping Functions

The following table lists the available Mapping Toolbox patch polygon display functions.

Function	Used For
`fillm`	Filled 2-D map polygons
`fill3m`	Filled 3-D map polygons in 3-D space
`geoshow`	Display map latitude and longitude data in 2-D
`mapshow`	Display map data without projection in 2-D
`patchm`	Patch objects projected on map axes
`patchesm`	Patches projected as individual objects on map axes

The `fillm` function makes use of the low-level function `patchm`. The toolbox provides another patch drawing function called `patchesm`. The optimal use of either depends on the application and user preferences. The `patchm` function creates one displayed object and returns one handle for a patch, which can contain multiple faces that do not necessarily connect. Mapping Toolbox data arrays contain NaNs to separate unconnected patch faces, unlike MATLAB patch display functions, which cannot handle NaN-delimited data for patches. The `patchesm` function, on the other hand, treats each face as a separate object and returns an array containing a handle for each patch. In general, `patchm` requires more memory but is faster than `patchesm`. The `patchesm` function is useful if you need to manipulate the appearance of individual patches (as thematic maps often require).

The `geoshow` and `mapshow` functions provide a superset of functionality for displaying unprojected and projected geodata, respectively, in two dimensions. These functions accept geographic data structures (geostructs and mapstructs) and coordinate vector arrays, but can also directly read shapefiles and geolocated raster files. With them, you can map polygon data, controlling rendering by constructing *symbolspecs*, data structures that you can construct with the `makesymbolspec` function. You can easily construct symbolspecs for point and line data as well as polygon data to control its display in `geoshow`, `mapshow`, and `mapview`.

Reprojectability of Maps with Vector Data. If you want to be able to change the projection of a map on the fly, you should not use `geoshow`. Some display functions, such as `patchm` , `fillm`, `displaym`, and `linem`, enable you to reproject vector map data, but `geoshow` does not. That is, when you change a map axes projection, with `setm` for example, vector map symbology that was created with `geoshow` will not be transformed. Gridded data rendered with `geoshow` (when `DisplayType` is `surface`, `texturemap`, or `contour`), however, can be reprojected.

Displaying Data Grids

In this section...

Types of Data Grids and Raster Display Functions

Mapping Toolbox functions and GUIs display both regular and geolocated data grids originating in a variety of formats. Recall that regular data grids require a *referencing vector or matrix* that describes the sampling and location of the data points, while geolocated data grids require matrices of latitude and longitude coordinates.

The data grid display functions are geographic analogies to the MATLAB surface drawing functions, but operate specifically on map axes objects. Like the line-plotting functions discussed in the previous chapter, some Mapping Toolbox grid function names correspond to their MATLAB counterparts with an `m` appended.

Note Mapping Toolbox functions beginning with `mesh` are used for regular data grids, while those beginning with `surf` are reserved for geolocated data grids. This usage differs from the MATLAB definition; `mesh` plots are used for colored wire-frame views of the surface, while `surf` displays colored faceted surfaces.

Surface map objects can be displayed in a variety of different ways. You can assign colors from the figure colormap to surfaces according to the values of their data. You can also display images where the matrix data consists of indices into a colormap or display the matrix as a three-dimensional surface, with the *z*-coordinates given by the map matrix. You can use monochrome surfaces that reflect a pseudo-light source, thereby producing a three-dimensional, shaded relief model of the surface. Finally, you can use a combination of color and light shading to create a lighted shaded relief map.

The following table lists the available Mapping Toolbox surface map display functions.

Function	Used For
`geoshow`	Display map data gridded in latitude and longitude in 2-D
`mapshow`	Display gridded map data without projection in 2-D
`meshm`	Regular data grid warped to projected graticule mesh
`surfm`	Geolocated data grid projected on map axes
`pcolorm`	Projected data grid in z = 0 plane
`surfacem`	Data grid warped to projected graticule mesh
`surflm`	3-D shaded surface with lighting projected on map axes
`meshlsrm`	3-D lighted shaded relief of regular data grid
`surflsrm`	3-D lighted shaded relief of geolocated data grid

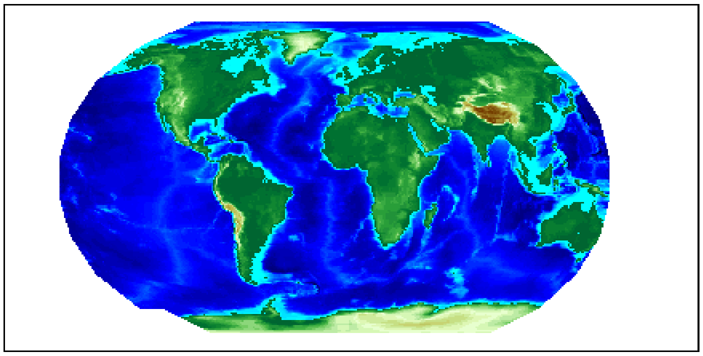
Fitting Gridded Data to the Graticule

The toolbox projects surface objects in a manner similar to the traditional methods of mapmaking. A cartographer first lays out a grid of meridians and parallels called the *graticule*. Each graticule cell is a geographic quadrangle. The cartographer calculates or interpolates the appropriate *x-y* locations for every vertex in the graticule grid and draws the projected graticule by connecting the dots. Finally, the cartographer draws the map data freehand, attempting to account for the shape of the graticule cells, which usually change shape across the map. Similarly, the toolbox calculates the *x-y* locations of the four vertices of each graticule cell and warps or samples the matrix data to fit the resulting quadrilateral.

In mapping data grids using the toolbox, as in traditional cartography, the finer the mesh (analogous to using a graticule with more meridians and parallels), the greater precision the projected map display will have, at the cost of greater effort and time. The graticule in a printed map is analogous to the spacing of grid elements in a regular data grid, the Mapping Toolbox representation of which is two-element vectors of the form `[`*`number-of-parallels`*`,`*`number-of-meridians`*`]`. The graticule for geolocated data grids is similar; it is the size of the latitude and longitude coordinate matrices: `number-of-parallels` = `mrows-1` and `number-of-meridians` = `ncols-1`. However, because geolocated data grids have arbitrary cell corner locations, spacing can vary and thus their graticule is not a regular mesh.

The `topo` regular data grid can be displayed quickly using a coarse graticule, at a cost of precision in terms of positioning the grid on the map. Observe the map that results from the following commands:

```
% Get data grid
load topo

% Create referencing matrix
topoR = makerefmat('RasterSize', size(topo), ...
    'Latlim', [-90 90], 'Lonlim', [0 360]);

% Set up Robinson proj
figure; axesm robinson

% Specify a 10x20 cell graticule
spacing = [10 20];

% Display data mapped to the graticule
h = meshm(topo,topoR,spacing);

% Set DEM color map
demcmap(topo)
```

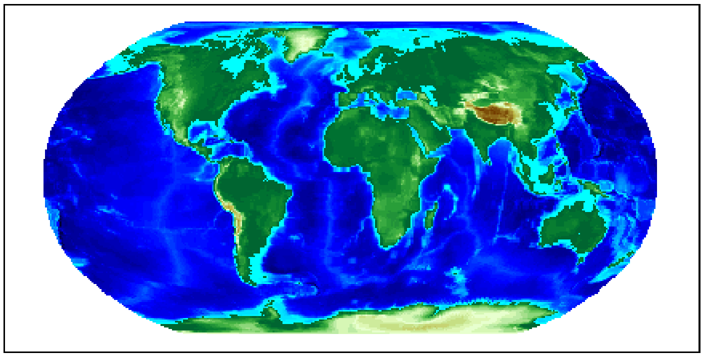


Notice that for this coarse graticule, the edges of the map do not appear as smooth curves. Previous displays used the default [50 100] graticule, for which this effect is negligible.

Regardless of the graticule resolution, the grid data is unchanged. In this case, the data grid is the 180-by-360 `topo` matrix, and regardless of where it is positioned, the data values are unchanged.

Map objects displayed as surfaces have all the properties of any MATLAB surface, which can be set at object creation or by using the MATLAB `set` function. The toolbox `setm` function allows the `MeshGrat` graticule property to be changed after a regular data grid has been displayed. Since you saved the handle of the last displayed map, reset its graticule to a very fine grid. Because making the mesh more precise is a trade-off of resolution versus time and memory, doing this takes longer, and requires more memory, to display the map:

```
setm(h,'MeshGrat',[200 400])
```



Another way you can reset a graticule is with the `meshgrat` function:

```
[latgrat,longrat] = meshgrat(topo,topoR,[200 400]);
setm(h,'Graticule',latgrat,longrat);
```

The vectors `latgrat` and `longrat` produced by `meshgrat` are vectors containing parallel and meridian values in each mesh direction.

Notice that the result does not appear to be any better than the original display with the default [50 100] graticule, but it took much longer to produce. There is no point to specifying a mesh finer than the data resolution (in this case, 180-by-360 grid cells). In practice, it makes sense to use coarse graticules for development tasks and fine graticules for final graphics production.

Using Raster Data to Create 3-D Displays

The simplest way to display raster data is to assign colors to matrix elements according to their data values and view them in two dimensions. Raster data maps also can be displayed as 3-D surfaces using the matrix values as the *z* data. Here you explore some basic concepts and operations for setting up surface views, which requires explicit horizontal coordinates.

Note The difference between regular raster data and a geolocated data grid is that each grid intersection for a geolocated grid is explicitly defined with (*x,y*) or (*latitude, longitude*) matrices or is interpolated from a graticule, while a regular matrix only implies these locations (which is why it needs a georeferencing vector or matrix).

You will use the raster elevation data in the `korea` MAT-file, which also includes bathymetry data for the region around the Korean peninsula, along with a referencing vector variable, which indicates that the data set is a regular data grid and locates it on the Earth.

1. Load the MAT-file and transform this representation to a fully geolocated data grid by calculating a mesh via the `meshgrat` function:

   ```
   load korea
   [lat,lon] = meshgrat(map,refvec);
   ```

2. Next use the `km2deg` function to convert the units of elevation from meters to degrees, so they are commensurate with the latitude and longitude coordinate matrices:

   ```
   map = km2deg(map/1000);
   ```

3. Observe the results by typing the `whos` command:

```
whos

  Name                  Size              Bytes  Class     Attributes

  description           2x64                256  char
  lat                 180x240            345600  double
  lon                 180x240            345600  double
  map                 180x240            345600  double
  maplegend             1x3                  24  double
  refvec                1x3                  24  double
  source                2x76                304  char
```

The `lat` and `lon` coordinate matrices form a mesh the same size as the `map` matrix. This is a requirement for constructing 3-D surfaces, unlike the example given above using the `topo` raster data set, which was displayed in 2-D using the `meshm` function. If you inspect `lat` and `lon` in the MATLAB Variable Editor, you find that in `lon`, all columns contain the same number for a given row, and in `lat`, all rows contain the same number for a given column. This is because the mesh produced by `meshgrat` in this case is regular, but such data grids need not have equal spacing.

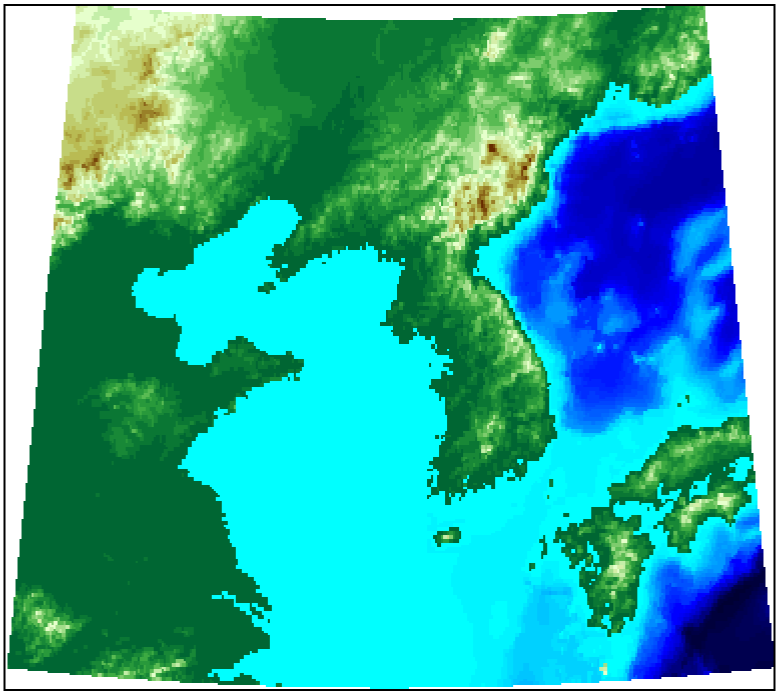
4. Now set up a map axes object with the equal area conic projection:

```
axesm('MapProjection','eqaconic','MapParallels',[],...
      'MapLatLimit',[30 45],'MapLonLimit',[115 135])
```

5. Instead of using the `meshm` function to make this map, display the `korea` geolocated data grid using the `surfm` function, and set an appropriate colormap:

```
surfm(lat,lon,map,map); demcmap(map)
tightmap
```

Here is the result, which is the same as what `meshm` would produce.

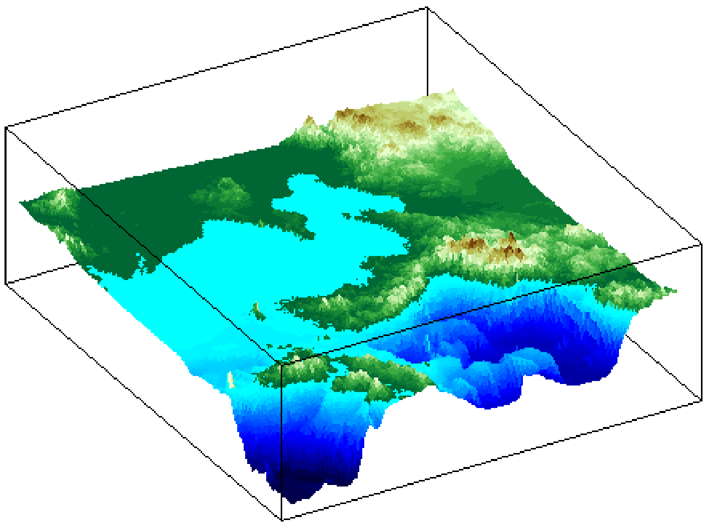


Be aware, however, that this map is really a 3-D view seen from directly overhead (the default perspective). To appreciate that, all you need to do is to change your viewpoint.

6 Use the `view` function to specify a viewing azimuth of 60 degrees (from the east southeast) and a viewing elevation of 30 degrees above the horizon:

```
view(60,30)
```

The figure immediately rotates to the specified perspective:



For information on Mapping Toolbox controls over perspective map representations or for additional help on constructing 3-D map displays, see Chapter 5, “Making Three-Dimensional Maps” .

Interacting with Displayed Maps

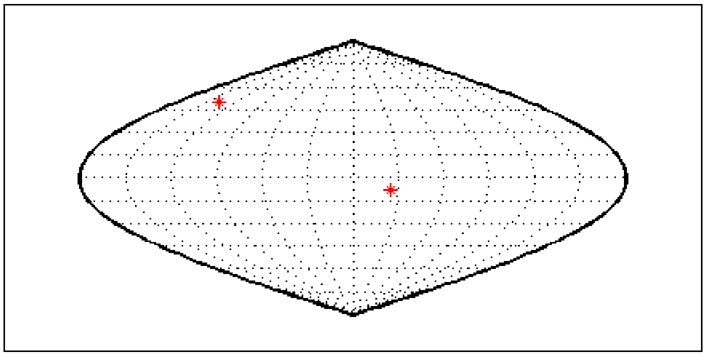
In this section...

Picking Locations Interactively

You can use Mapping Toolbox functions and GUIs to interact with maps, both in `mapview` and in figures created with `axesm`. This section describes two useful graphic input functions, `inputm` and `gcpmap`. The `inputm` function (analogous to the MATLAB `ginput` function) allows you to get the latitude-longitude position of a mouse click. The `gcpmap` function (analogous to the MATLAB function `get(gca,'CurrentPoint')`) returns the current mouse position, also in latitude and longitude.

Explore `inputm` with the following commands, which display a map axes with its grid and then request three mouse clicks, the locations of which are stored as geographic coordinates in the variable `points`. Then the `plotm` function plots the points you clicked as red markers. The display you see depends on the points you select:

```
axesm sinusoid
framem on; gridm on
points=inputm(3)
points =
  -41.7177 -145.0293
    7.9211   -0.5332
   38.5492  149.2237
plotm(points,'r*')
```



Note If you click outside the map frame, `inputm` returns a valid but incorrect latitude and longitude, even though the point you indicated is off the map.

One reason you might want to manually identify points on a map is to interactively explore how much distortion a map projection has at given locations. For example, you can feed the data acquired with `inputm` to the `distortcalc` function, which computes area and angular distortions at any location on a displayed map axes. If you do so using the `points` variable, the results of the previous three mouse clicks are as follows:

```
[areascale,angledef] = distortcalc(points(1,1),points(1,2))
areascale =
    1.0000
angledef =
    85.9284
>> [areascale,angledef] = distortcalc(points(2,1),points(2,2))
areascale =
    1.0000
angledef =
    3.1143
[areascale,angledef] = distortcalc(points(3,1),points(3,2))
areascale =
    1.0000
angledef =
    76.0623
```

This indicates that the current projection (sinusoidal) has the equal-area property, but exhibits variable angular distortion across the map, less near the equator and more near the poles.

To see a working application that uses the `inputm` function, view and run the `mapexfindcity` Interactive Global City Finder demo.

Defining Small Circles and Tracks Interactively

Geographic line annotations such as navigational tracks and small circles can be generated interactively. Great circle tracks are the shortest distance between points, and when closed partition the Earth into equal halves; a small circle is the locus of points at a constant distance from a reference point. Use `trackg` and `scircleg` to create them by clicking on the map. Double-click the tracks or circles to modify the lines. **Shift**+click to type specific parameters into a control panel. The control panels also allow you to retrieve or set properties of tracks and circles (for instance, great circle distances and small circle radii).

The following example illustrates how to interactively create a great circle track from Los Angeles, California, to Tokyo, Japan, and a 1000 km radius small circle centered on the Hawaiian Islands. The track is made via the `trackg` function, which prompts you to select endpoints for a track with the mouse. The `scircleg` function prompts for two points also, a center and any point on the circumference of the small circle. The specifics of the track and the circle are then adjusted more precisely with dialog controls:

1. Set up an orthographic view centered over the Pacific Ocean. Use the `coast` MAT-file:

   ```
   axesm('ortho','origin',[30 180])
   framem;gridm
   load coast
   plotm(lat,long,'k')
   ```

2. Create a track with the `trackg` function, which prompts for two endpoints. The default track type is a great circle:

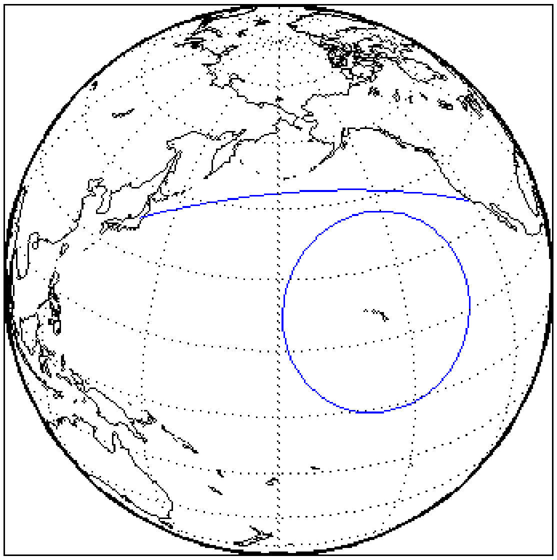
   ```
   trackg
   Track1:  Click on starting and ending points
   ```

Click near Los Angeles and Tokyo, and the track is drawn.

3 Now create a small circle around Hawaii with the `scircleg` function, which prompts for a center point and a point on the perimeter. Make the circle's radius about 2000 km, but don't worry about getting the size exact:

```
scircleg
Circle 1:  Click on center and perimeter
```

The map should look approximately like this.



4 Adjust the size of the small circle to be 2000 km by **Shift**+clicking anywhere on its perimeter. The Small Circles dialog box appears.

5 Type `2000` into the **Radius** field.

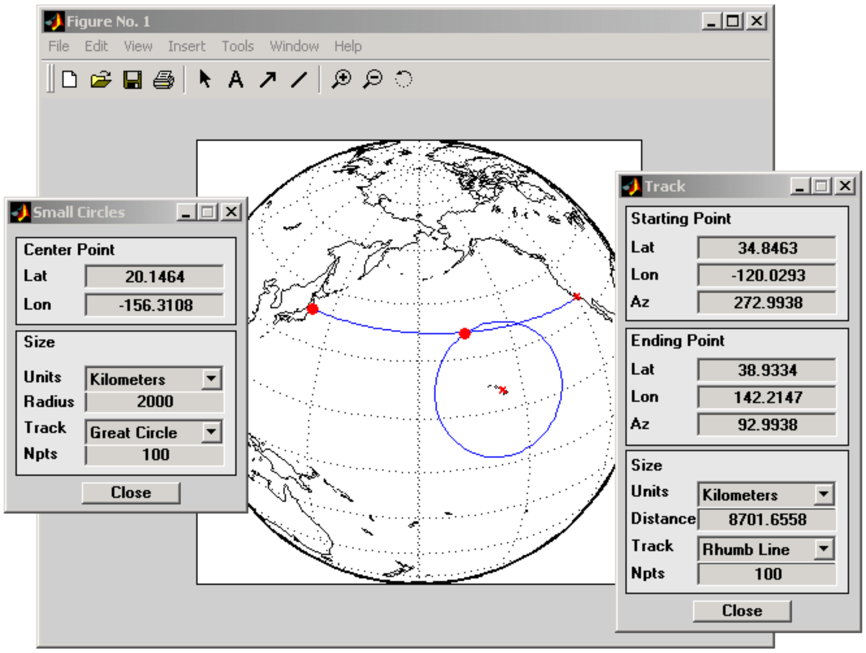
6 Click **Close**. The small circle readjusts to be 2000 km around Hawaii.

7 To adjust the track between Los Angeles and Tokyo, **Shift**+click on it. This brings up the Track dialog, with which you specify a position and initial azimuth for either endpoint, as well as the length and type of the track.

8 Change the track type from `Great Circle` to `Rhumb Line` with the Track pop-up menu. The track immediately changes shape.

9 Experiment with the other Track dialog controls. Also note that you can move the endpoints of the track with the mouse by dragging the red circles, and obtain the arc's length in various units of distance.

The following figure shows the Small Circles and Track dialog boxes.



Small Circles

Center Point

Lat	20.1464
Lon	-156.3108

Size

Units	Kilometers
Radius	2000
Track	Great Circle
Npts	100

Track

Starting Point

Lat	34.8463
Lon	-120.0293
Az	272.9938

Ending Point

Lat	38.9334
Lon	142.2147
Az	92.9938

Size

Units	Kilometers
Distance	8701.6558
Track	Rhumb Line
Npts	100

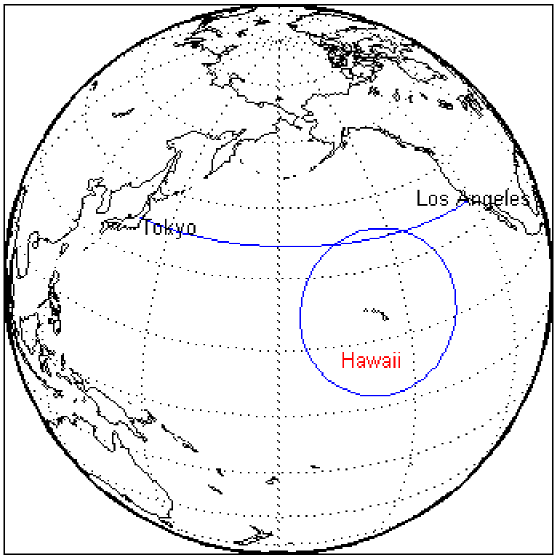
Interactive Text Annotation

You can also interactively place text annotations by clicking on a map display. The `textm` function, which requires numerical arguments for locating a specified text string, was illustrated in "Placing Geographic and Nongeographic Objects in a Map Axes" on page 4-43. The `gtextm` function, which takes a text string and optional properties as arguments, interactively defines the location for the specified text object based on where you click on the map.

Try these `gtextm` commands to label the locations you have just annotated:

```
gtextm('Hawaii','color','r')
gtextm('Tokyo')
gtextm('Los Angeles')
```

The following figure displays the results of these `gtextm` commands. After you place text, you can move it interactively using the selection tool in the map figure window.



Working with Objects by Name

You can manipulate displayed map objects by name. Many Mapping Toolbox functions assign descriptive names to the `Tag` property of the objects they create. The `namem` and related functions allow you to control the display of groups of similarly named objects, determine the names and change them if desired, and use the name in the Handle Graphics® `set` and `get` functions. There is also a Mapping Toolbox graphical user interface, `mobjects`, to help you manage the display and control of objects.

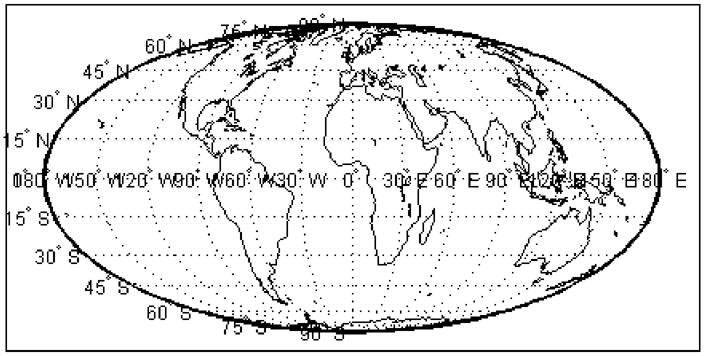
Some mapping display functions like `framem`, `gridm`, and `contourm` assign object tags by default. You can also set the name upon display by assigning a string to the `Tag` property in mapping display functions that use property name/property value pairs. If the `Tag` does not contain a string, the name defaults to an object's `Type` property, such as `'line'` or `'text'`.

Determining and Manipulating Object Names

1 Display a vector map of the world:

```
f = axesm('fournier')
framem on; gridm on;
plabel on; mlabel('MLabelParallel',0)
load coast
plotm(lat,long,'k','Tag','Coastline')
```

Below is the resulting map.



75° N
90° N
60° N
45° N
30° N
15° N
180° W150° W120° W90° W60° W30° W 0° 30° E 60° E 90° E120° E150° E180° E
15° S
30° S
45° S
60° S
75° S
90° S

2 List the names of the objects in the current axes using `namem`:

```
namem
ans =
Coastline
PLabel
MLabel
Meridian
Parallel
```

```
Frame
```

3 The `handlem` function allows you to dereference graphic objects and to get or set their properties. Change the line width of the coastline with `set`:

```
set(handlem('Coastline'),'LineWidth',2)
```

4 Change the colors of the meridian and parallel labels separately:

```
set(handlem('Mlabel'),'Color',[.5 .2 0])
set(handlem('Plabel'),'Color',[.2 .5 0])
```

You can also change these labels to be the same color using `setm`:

```
setm(f,'fontcolor', [.4 .5 .6])
```

5 The `handlem` command with no arguments summons a UI control with a list of map axes objects. This is useful for selecting objects interactively. Try

```
handlem
```

or

```
h = handlem
```

6 Combined with `set`, this makes it simple to change properties such as color. Remember, however, to use the right property name. Patches, for example, have a `FaceColor` and `EdgeColor`, while most other objects simply have `Color`, as is the case with the `Coastline` object. Now use `handlem` to call a color picker to set the coastline to any color you like:

```
set(handlem,'Color',uisetcolor)
```

The reference page for `handlem` lists the object names that it recognizes. Note that most of these names can be prefixed with "`all`", which returns an array of all handles for that class of object.

7 Now try `handlem` using the `all` modifier:

```
t = handlem('alltext')
l = handlem('allline')
```

Note that you can also use `all` with the `hidem` and `showm` functions:

```
hidem('alltext')
showm('alltext')
```

For more information on the use of functions and tools for manipulating objects, consult the `setm`, `getm`, `handlem`, `hidem`, `showm`, `clmo`, `namem`, `tagm`, and `mobjects` reference pages.

5

Making Three-Dimensional Maps

Sources of Terrain Data

In this section...

Digital Terrain Elevation Data from NGA

Nearly all published terrain elevation data is in the form of data grids. "Displaying Data Grids" on page 4-68 described basic approaches to rendering surface data grids with Mapping Toolbox functions, including viewing surfaces in 3-D axes. The following sections describe some common data formats for terrain data, and how to access and prepare data sets for particular areas of interest.

The Digital Terrain Elevation Data (DTED) Model is a series of gridded elevation models with global coverage at resolutions of 1 kilometer or finer. DTEDs are products 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). The data is provided as 1-by-1 degree tiles of elevations on geographic grids with product-dependent grid spacing. In addition to NGA's own DTEDs, terrain data from Shuttle Radar Topography Mission (SRTM), a cooperative project between NASA and NGA, are also available in DTED format, levels 1 and 2 (see below).

The lowest resolution data is the DTED Level 0, with a grid spacing of 30 arc-seconds, or about 1 kilometer. The DTED files are binary. The files have filenames with the extension `dtN`, where `N` is the level of the DTED product. You can find published specifications for DTED at the NGA web site.

NGA also provides higher resolution terrain data files. DTED Level 1 has a resolution of 3 arc-seconds, or about 100 meters, increasing to 18 arc-seconds near the poles. It was the primary source for the USGS 1:250,000 (1 degree) DEMs. Level 2 DTED files have a minimum resolution of 1 arc-second near the equator, increasing to 6 arc-seconds near the poles. DTED files are available on from several sources on CD-ROM, DVD, and on the Internet.

Note For information on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

Digital Elevation Model Files from USGS

The United States Geological Survey (USGS) has prepared terrain data grids for the U.S. suitable for use at scales between 1:24,000 and 1:250,000 and beyond. Some of this data originated from Defense Mapping Agency DTEDs. Specifications and data quality information are available for these digital elevation models (DEMs) and other U.S. National Mapping Program geodata from the USGS. USGS no longer directly distributes 1:24,000 DEMs and other large-scale geodata. U.S. DEM files in SDTS format are available from private vendors, either for a fee or at no charge, depending on the data sets involved.

The largest scale USGS DEMs are partitioned to match the USGS 1:24,000 scale map series. The grid spacing for these elevations models is 30 meters on a Universal Transverse Mercator grid. Each file covers a 7.5-minute quadrangle. (Note, however, that only a subset of paper quadrangle maps are projected with UTM, and that USGS vector geodata products might not use this coordinate system.) The map and data series is available for much of the conterminous United States, Hawaii, and Puerto Rico.

Determining What Elevation Data Exists for a Region

Several Mapping Toolbox functions and a GUI help you identify file names for and manage digital elevation model data for areas of interest. These tools do not retrieve data from the Internet; however, they do locate files that lie on the Mapping Toolbox path and indicate the names of data sets that you can download or order on magnetic media or CD-ROM.

Certain Mapping Toolbox utility functions can describe and import elevation data. The following table describes functions that read in data, determine what file names might exist for a given area, or return metadata for elevation grid files. These files are data products packaged by government agencies; with minor exceptions, the format used for each is unique to that data product, which is why special functions are required to read them and why their filenames and/or footprints can be known *a priori*.

File Type	Description	Function to Read Files	Function to Identify or Summarize Files
DTED	U.S. Department of Defense Digital Terrain Elevation Data	`dted`	`dteds`
DEM	USGS 1-degree (3-arc-second resolution) digital elevation models	`usgsdem`	`usgsdems`
DEM24K	USGS 1:24K (30-meter resolution) digital elevation models	`usgs24kdem`	N/A
ETOPO5 ETOPO2	Earth Topography – 5-minute (ETOPO5) and 2-minute (ETOPO2)	`etopo`	N/A
GTOPO30	Tiles of 30-arc-second global elevation models	`gtopo30`	`gtopo30s`
SATBATH	Global 2-minute (4 km) satellite topography and bathymetry data	`satbath`	N/A
SDTS DEM	Digital elevation models in U.S. SDTS format	`sdtsdemread`	`sdtsinfo` (reads metadata from catalog file)
TBASE	TerrainBase topography and bathymetry binary and ASCII grid files	`tbase`	N/A

Note that the names of functions that identify file names are those of their respective file-reading functions appended with `s`. These functions determine file names for areas of interest, and have calling arguments of the form (`latlim`, `lonlim`), with which you indicate the latitude and longitude limits for an area of interest, and all return a list of filenames that provide the elevations required. The southernmost latitude and the western-most longitude must be the first numbers in `latlim` and `lonlim`, respectively.

Using dteds, usgsdems, and gtopo30s to Identify DEM Files

Suppose you want to obtain elevation data for the area around Cape Cod, Massachusetts. You define your area of interest to extend from 41.1ºN to 43.9ºN latitude and from 71.9ºW to 69.1ºW longitude.

1 To determine which DTED files you need, use the `dteds` function, which returns a cell array of strings:

```
dteds([41.1 43.9],[-71.9 -69.1])
ans =
    '\DTED\W072\N41.dt0'
    '\DTED\W071\N41.dt0'
    '\DTED\W070\N41.dt0'
    '\DTED\W072\N42.dt0'
    '\DTED\W071\N42.dt0'
    '\DTED\W070\N42.dt0'
    '\DTED\W072\N43.dt0'
    '\DTED\W071\N43.dt0'
    '\DTED\W070\N43.dt0'
```

Note three important considerations about using DTED files:

a DTED filenames reflect latitudes only and thus do not uniquely specify a data set; they must be organized within directories that specify longitudes. When you download level 0 DTEDs, the `DTED` directory and its subdirectories are transferred as a compressed archive that you must decompress before using.

b Some files that the `dteds` function identifies do not exist, either because they completely cover water bodies or have never been created or released by NGA. The `dted` function that reads the DTEDs handles missing cells appropriately.

c NGA might or might not continue to make DTED data sets available to the general public online. For information on availability of terrain data from NGA and other sources, see `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

2 To determine the USGS DEM files you need, use the `usgsdems` function:

```
usgsdems([41.1 43.9],[-71.9 -69.1])
ans =
    'portland-w'
    'portland-e'
    'bath-w'
    'boston-w'
    'boston-e'
    'providence-w'
    'providence-e'
    'chatham-w'
```

Note that, in contrast to the `dteds` command you executed above, there are eight rather than nine files listed to cover the 3-by-3-degree region of interest. The cell that consists entirely of ocean has no name and is thus omitted from the output cell array.

3 To determine the GTOPO30 files you need, use the `gtopo30s` function:

```
gtopo30s([41.1 43.9],[-71.9 -69.1])
ans =
    'w100n90'
```

Note The DTED, GTOPO30, and small-scale (low-resolution) USGS DEM grids are in latitude and longitude. Large-scale (24K) USGS DEMs grids are in UTM coordinates. The `usgs24kdem` function automatically unprojects the UTM grids to latitude and longitude; the `stdsdemread` function does not.

For additional information, see the reference pages for `dteds`, `usgsdems`, `usgs24kdem`, and `gtopo30s`.

Mapping a Single DTED File with the DTED Function

In this exercise, you render DTED level 0 data for a portion of Cape Cod. The 1°-by-1° file can be downloaded from NGA or purchased on CD-ROM. You read and display the elevation data at full resolution as a lighted surface to show both large- and small-scale variations in the data.

1 Define the area of interest and determine the file to be obtained:

```
latlim = [ 41.20  41.95];
lonlim = [-70.95 -70.10];
```

2 To determine which DTED files you need, use the `dteds` function, which returns a cell array of strings:

```
dteds(latlim, lonlim)
ans =
    'dted\w071\n41.dt0'
```

In this example, only one DTED file is needed, so the answer is a single string. For more information on the `dteds` function, see "Using dteds, usgsdems, and gtopo30s to Identify DEM Files" on page 5-5).

3 Unless you have a CD-ROM containing this file, download it from the source indicated in the following tech note:

```
http://www.mathworks.com/support/tech-notes/2100/2101.html
```

The original data comes as a compressed tar or zip archive that you must expand before using.

4 Use the `dted` function to create a terrain grid and a referencing vector in the workspace at full resolution. If more than one DTED file named n41.dt0 exists on the path, your working directory must be `/dted/w071` in order to be sure that `dted` finds the correct file. If the file is not on the path, you are prompted to navigate to the `n41.dt0` file by the `dted` function:

```
samplefactor = 1;
[capeterrain, caperef] = dted('n41.dt0', ...
  samplefactor, latlim, lonlim);
```

5 Because DTED files contain no bathymetric depths, decrease elevations of zero slightly to render them with blue when the colormap is reset:

```
capeterrain(capeterrain == 0) = -1;
```

6 Use `usamap` to construct an empty map of axes for the region defined by the latitude and longitude limits:

```
figure;
ax = usamap(latlim,lonlim);
```

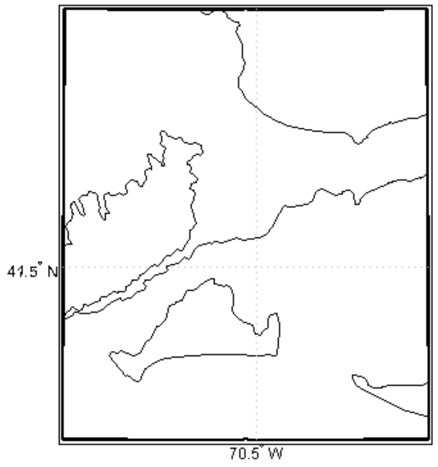
7 Read data for the region defined by the latitude and longitude limits from the `usastatehi` shapefile:

```
capecoast = shaperead('usastatehi',...
    'UseGeoCoords', true,...
    'BoundingBox', [lonlim' latlim']);
```

8 Display coastlines on the map axes that was created with `usamap`:

```
geoshow(ax, capecoast, 'FaceColor', 'none');
```

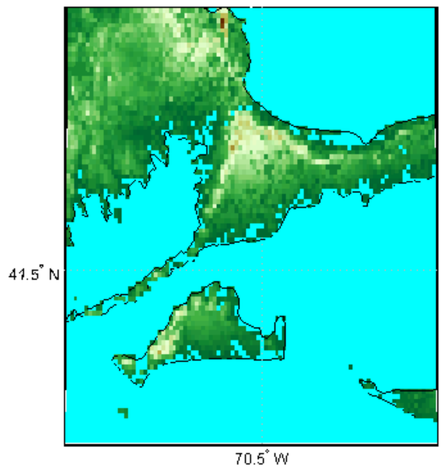
At this point the map looks like this:



9 Render the elevations, and set the colormap accordingly:

```
meshm(capeterrain, caperef, size(capeterrain), capeterrain);
demcmap(capeterrain)
```

The resulting map, shown below, is a window on Cape Cod, and illustrates the relative coarseness of DTED level 0 data.



Mapping Multiple DTED Files with the DTED Function

When your region of interest extends across more than one DTED tile, the `dted` function concatenates the tiles into a single matrix, which can be at full resolution or a sample of every *n*th row and column. You can specify a single DTED file, a directory containing several files (for different latitudes along a constant longitude), or a higher level directory containing subdirectories with files for several longitude bands.

1 To follow this exercise, you need to acquire the necessary DTED files from the Internet as described in the following tech note

   ```
   http://www.mathworks.com/support/tech-notes/2100/2101.html
   ```

   or from a CD-ROM. This yields a set of directories that contain the following files:

   ```
   /dted
       /w070
           n41.avg
           n41.dt0
           n41.max
           n41.min
           n43.avg
   ```

```
            n43.dt0
            n43.max
            n43.min
        /w071
            n41.avg
            n41.dt0
            n41.max
            n41.min
            n42.avg
            n42.dt0
            n42.max
            n42.min
            n43.avg
            n43.dt0
            n43.max
            n43.min
        /w072
            n41.avg
            n41.dt0
            n41.max
            n41.min
            n42.avg
            n42.dt0
            n42.max
            n42.min
            n43.avg
            n43.dt0
            n43.max
            n43.min
```

2 Change your working directory to the directory that includes the top-level DTED directory (which is always named `dted`).

3 Use the `dted` function, specifying that directory as the first argument:

```
latlim = [ 41.1  43.9];
lonlim = [-71.9 -69.1];
samplefactor = 5;
[capetopo,caperef] = dted(pwd, samplefactor, latlim, lonlim);
```

The sample factor value of 5 specifies that only every fifth data cell, in both latitude and longitude, will be read from the original DTED file. You can choose a larger value to save memory and speed processing and display, at the expense of resolution and accuracy. The size of your elevation array (`capetopo`) will be inversely proportional to the square of the sample factor.

Note You can specify a DTED filename rather than a directory name if you are accessing only one DTED file. If the file cannot be found, a file dialog is presented for you to navigate to the file you want. See the example "Mapping a Single DTED File with the DTED Function" on page 5-6.

---

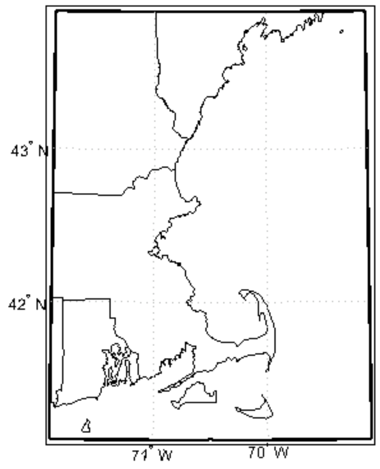
4. As DTEDs contain no bathymetric depths, recode all zero elevations to -1, to enable water areas to be rendered properly:

   ```
   capetopo(capetopo==0)=-1;
   ```

5. Obtain the elevation grid's latitude and longitude limits; use them to draw an outline map of the area to orient the viewer:

   ```
   [latlim,lonlim] = limitm(capetopo,caperef);

   figure;
   ax = usamap(latlim,lonlim);
   capecoast = shaperead('usastatehi',...
       'UseGeoCoords', true,...
       'BoundingBox', [lonlim' latlim']);
   geoshow(ax,capecoast,'FaceColor','None');
   ```

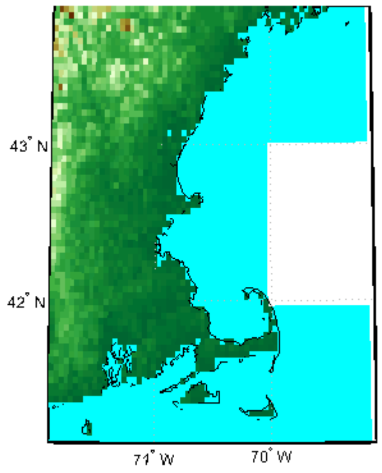
   The map now looks like this.



6 Render the elevation grid with `meshm`, and then recolor the map with `demcmap` to display hypsometric colors (elevation tints):

```
meshm(capetopo, caperef, size(capetopo), capetopo);
demcmap(capetopo)
```

Here is the map; note the missing tile to the right where no DTED data exists.



Reading Elevation Data Interactively

Extracting DEM Data with demdataui

You can browse many formats of digital elevation map data using the `demdataui` graphical user interface. The `demdataui` GUI determines and graphically depicts coverage of ETOPO5, TerrainBase, the satellite bathymetry model (SATBATH), GTOPO30, GLOBE, and DTED data sets on local and network file systems, and can import these files into the workspace.

Note When it opens, `demdataui` scans your Mapping Toolbox path for candidate data files. On PCs, it also checks the root directories of CD-ROMs and other drives, including mapped network drives. This can cause a delay before the GUI appears.

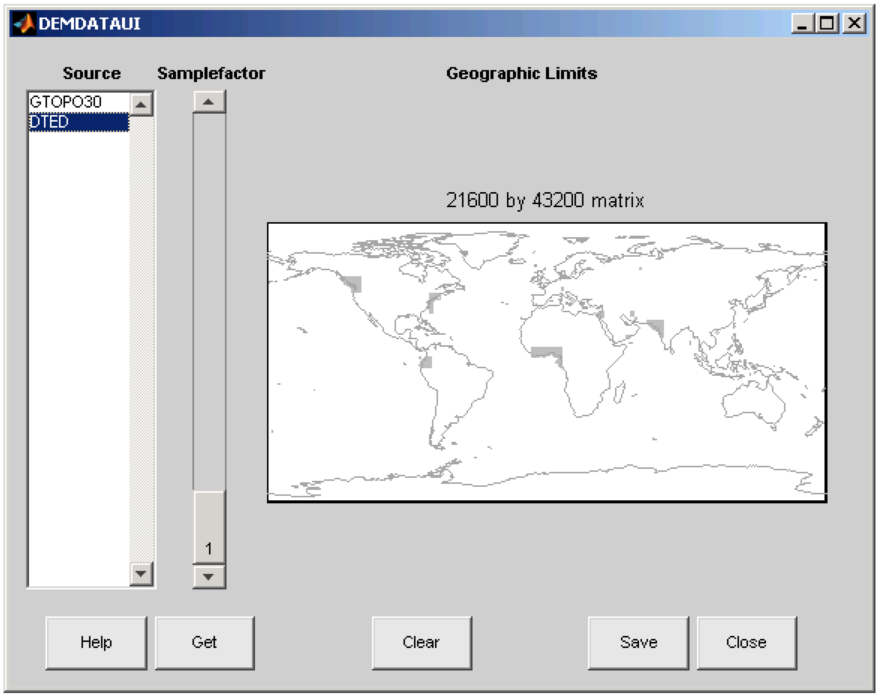
You can choose to read from any of the data sets `demdataui` has located. If `demdataui` does not recognize data you think it should find, check your path and click **Help** to read about how files are identified.

This exercise illustrates how to use the `demdataui` interface. You will not necessarily have all the DEM data sets shown in this example. Even if you have only one (the DTED used in the previous exercise, for example), you can still follow the steps to obtain your own results:

1 Open the `demdataui` UI. It scans the path for data before it is displayed:

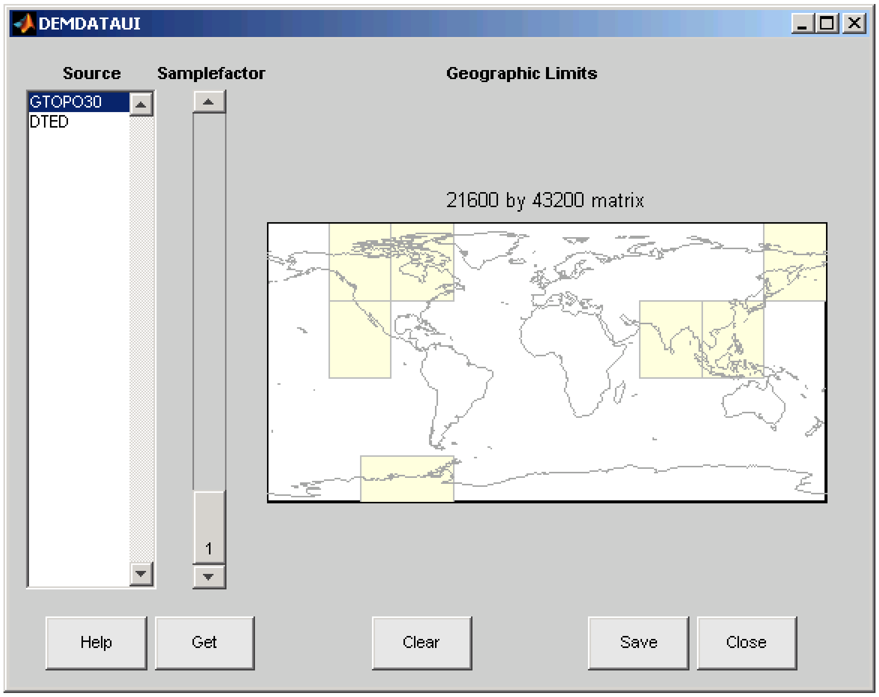
```
demdataui
```

The **Source** list in the left pane shows the data sets that were found. The coverage of each data set is indicated by a yellow tint on the map with gray borders around each tile of data. Here, the source is selected to present all DTED files available to a user.



21600 by 43200 matrix

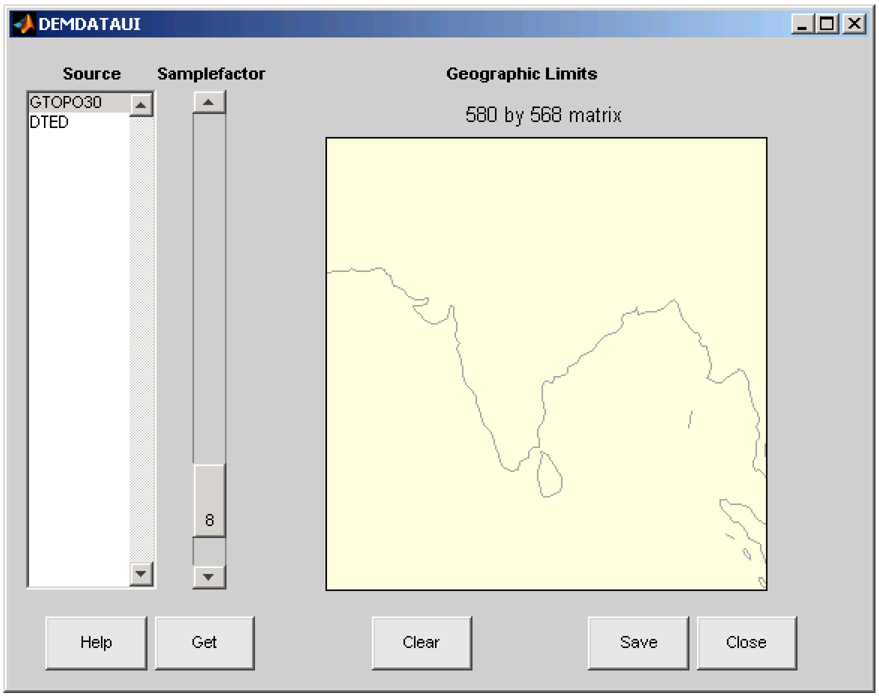
2 Clicking a different source in the left column updates the coverage display. Here is the coverage area for available GTOPO30 tiles.



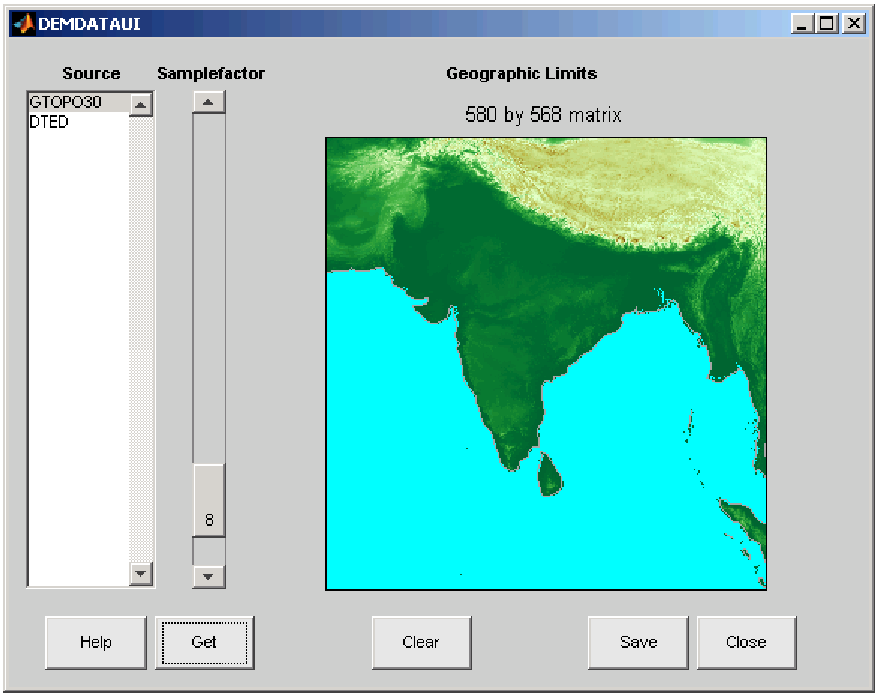
21600 by 43200 matrix

3 Use the map in the UI to specify the location and density of data to extract. To interactively set a region of interest, click in the map to zoom by a factor of two centered on the cursor, or click and drag across the map to define a rectangular region. The size of the matrix of the area currently displayed is printed above the map. To reduce the amount of data, you can continue to zoom in, or or you can raise the **Samplefactor** slider. A sample factor of 1 reads every point, 2 reads every other point, 3 reads every third point, etc. The matrix size is updated when you move the **Samplefactor** slider.

Here is the UI panel after selecting ETOPO30 data and zooming in on the Indian subcontinent.



4 To see the terrain you have windowed at the sample factor you specified, click the **Get** button. This causes the GUI map pane to repaint to display the terrain grid with the `demcmap` colormap. In this example, the data grid contains 580-by-568 data values, as shown below.



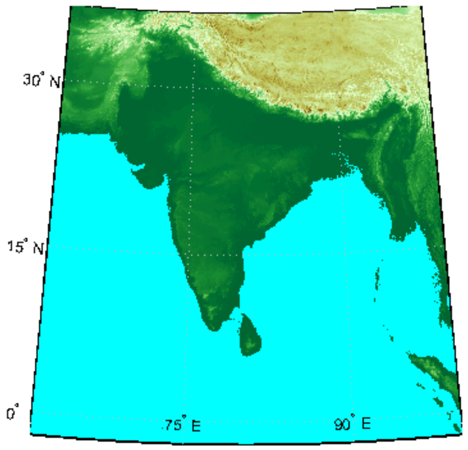
5 If you are not satisfied with the result, click the **Clear** button to remove all data previously read in via **Get** and make new selections. You might need to close and reopen `demdatui` in order to select a new region of interest.

6 When you are ready to import DEM data to the workspace or save it as a MAT-file, click the **Save** button. Select a destination and name the output variable or file. You can save to a MAT-file or to a workspace variable. The `demdataui` function returns one or more matrices as an array of display structures, having one element for each separate *`get`* you requested (assuming you did not subsequently **Clear**). You then use `geoshow` or `mlayers` to add the data grids to a map axes.

The data returned by `demdataui` contains display structures. You cannot update these to geographic data structures (geostructs) using the `updategeostruct` function, because they are of type `surface`, which the updating function does not recognize. However, you can still display them with `geoshow`, as shown in the next step.

7 To access the contents of the display structure, use its field names. Here `map` and `maplegend` are copied from the structure and used to create a lighted three-dimensional elevation map display using `worldmap`. (`demdata` is the default name for the structure, which you can override when you save it.)

```
Z = demdata.map;
refvec = demdata.maplegend;
figure
ax = worldmap(Z, refvec);
geoshow(ax, Z, refvec, 'DisplayType', 'texturemap');
axis off
demcmap(Z);
```



Determining and Visualizing Visibility Across Terrain

Computing Line of Sight with los2

You can use regular data grids of elevation data to answer questions about the mutual visibility of locations on a surface (intervisibility). For example,

- Is the line of sight from one point to another obscured by terrain?
- What area can be seen from a location?
- What area can see a given location?

The first question can be answered with the `los2` function. In its simplest form, `los2` determines the visibility between two points on the surface of a digital elevation map. You can also specify the altitudes of the observer and target points, as well as the datum with respect to which the altitudes are measured. For specialized applications, you can even control the actual and effective radius of the Earth. This allows you to assume, for example, that the Earth has a radius 1/3 larger than its actual value, which is a model frequently used in predicting radio wave propagation.

The following example shows a line-of-sight calculation between two points on a regular data grid generated by the `peaks` function. The calculation is performed by the `los2` function, which returns a logical result: `1` (points are intervisible), or `0` (points are not intervisible).

1. Create an elevation grid using `peaks` with a maximum elevation of 500, and set its origin at (0°N, 0°W), with a spacing of 1000 cells per degree):

   ```
   map = 500*peaks(100);
   maplegend = [ 1000 0 0];
   ```

2. Define two locations on this grid to test intervisibility:

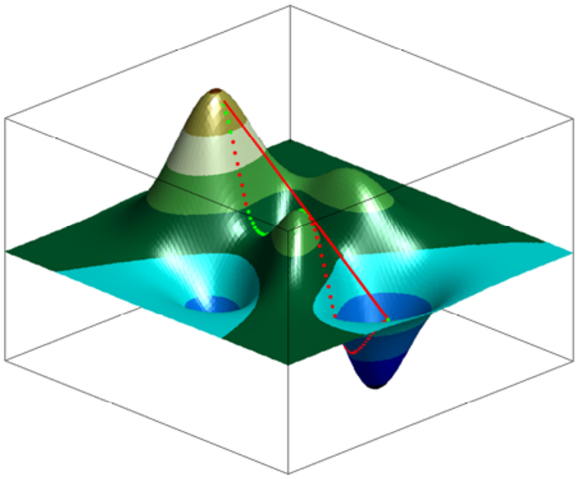
   ```
   lat1 = -0.027; lon1 = 0.05; lat2 = -0.093; lon2 = 0.042;
   ```

3. Calculate intervisibility. The final argument specifies the altitude (in meters) above the surface of the first location (`lat1`, `lon1`) where the observer is located (the viewpoint):

   ```
   los2(map,maplegend,lat1,lon1,lat2,lon2,100)
   ```

```
ans =
1
```

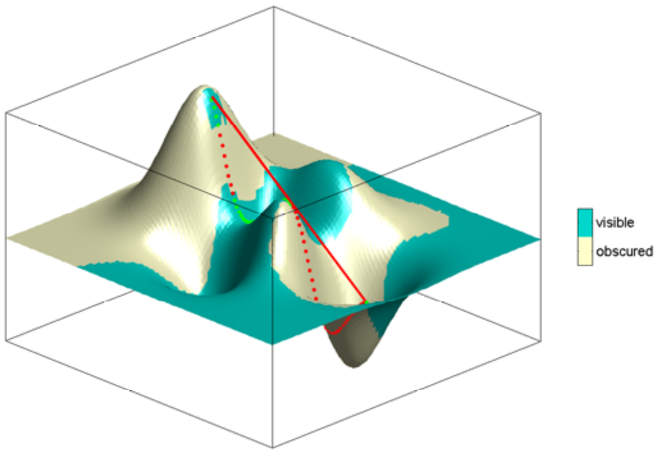
The `los2` function also produces a profile diagram in a figure window showing visibility at each grid cell along the line of sight that can be used to interpret the Boolean result. In this example, the diagram shows that the line between the two locations just barely clears an intervening peak.



You can also compute the *viewshed*, a name derived from watershed, which is all of the areas that are visible from a particular location. The `viewshed` function can be thought of as performing the `los2` line-of-sight calculation from one point on a digital elevation map to every other entry in the matrix. The `viewshed` function supports the same options as `los2`.

The following shows which parts of the peaks elevation map in the previous example are visible from the first point:

```
[vismap,vismapleg] = viewshed(map,maplegend,lat1,lon1,100);
```


visible
obscured

Shading and Lighting Terrain Maps

In this section...

Lighting a Terrain Map Constructed from a DTED File

The `lightm` function creates light objects in the current map. To modify the positions and colors of lights created on world maps or large regions you can use the interactive `lightmui` GUI. For finer control over light position (for example, in small areas lit by several lights), you have to specify light positions using projected coordinates. This is because lights are children of axes and share their coordinate space. See "Lighting a Global Terrain Map with lightm and lightmui" on page 5-25 for an example of using `lightmui`.

In this exercise, you manually specify the position of a single light in the northwest corner of a DTED DEM for Cape Cod.

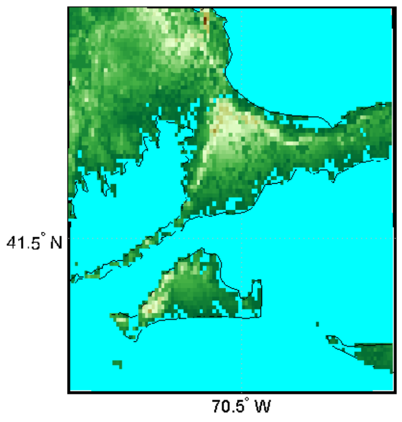
1 To illustrate lighting terrain maps, begin by following the exercise in "Mapping a Single DTED File with the DTED Function" on page 5-6, or execute the steps below:

```
latlim = [ 41.20  41.95];
lonlim = [-70.95 -70.10];
cd dted\w071 %Note: Your absolute path may vary
samplefactor = 1;
[capeterrain, caperef] = dted('n41.dt0', samplefactor,...
  latlim, lonlim);
capeterrain(capeterrain == 0) = -1;
capecoast = shaperead('usastatehi',...
    'UseGeoCoords', true,...
    'BoundingBox', [lonlim' latlim']);
```

2 Construct a map of the region within the specified latitude and longitude limits:

```
figure
ax = usamap(latlim,lonlim);
geoshow(ax, capecoast, 'FaceColor', 'none');
geoshow(ax, capeterrain, caperef, 'DisplayType', 'mesh');
demcmap(capeterrain)
```

The map looks like this.



3 Set the vertical exaggeration. Use `daspectm` to specify that elevations are in meters and should be multiplied by 20:

```
daspectm('m',20)
```

4 Make sure that the line data is visible. To ensure that it is not obscured by terrain, use `zdatam` to set it to the highest elevation of the `cape1` terrain data:

```
zdatam('allline',max(capeterrain(:)))
```

5 Specify a location for a light source with `lightm`:

```
h = lightm(42,-71);
```

If you omit arguments, a GUI for setting positional properties for the new light opens.

6 To see the properties of light objects, inspect the handle returned by `lightm`:

```
get(h)
 Position = [-0.00616097 0.796039 1]
 Color = [1 1 1]
 Style = infinite

 BeingDeleted = off
 ButtonDownFcn =
 Children = []
 Clipping = on
 CreateFcn =
 DeleteFcn =
 BusyAction = queue
 HandleVisibility = on
 HitTest = on
 Interruptible = on
 Parent = [138.001]
 Selected = off
 SelectionHighlight = on
 Tag =
 Type = light
 UIContextMenu = []
 UserData = [ (1 by 1) struct array]
 Visible = on
```

Had you used the MATLAB `light` function in place of `lightm`, you would have needed to specify the position in Cartesian 3-space.

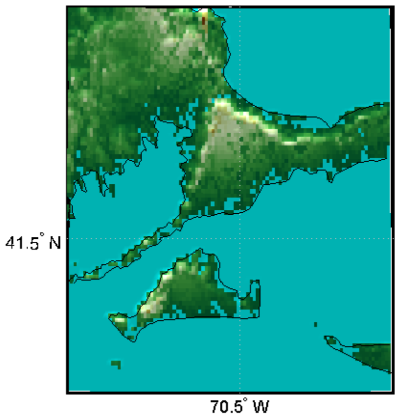
7 The lighting computations caused the map to become quite dark with specular highlights. Now restore its luminance by specifying three surface reflectivity properties in the range of 0 to 1:

```
ambient = 0.7; diffuse = 1; specular = 0.6;
material([ambient diffuse specular])
```

The surface looks blotchy because there is no interpolation of the lighting component (flat facets are being modeled). Correct this by specifying Phong shading:

```
lighting phong
```

The map now looks like this.



8 If you want to compare the lit map with the unlit version, you can toggle the lighting off:

```
lighting none
```

For additional information, see the reference pages for `daspectm`, `lightm`, `light`, `lighting`, and `material`.

Lighting a Global Terrain Map with lightm and lightmui

In this example, you create a global topographic map and add a local light at a distance of 250 km above New York City, (40.75 °N, 73.9 °W). You then change the material and lighting properties, add a second light source, and activate the `lightmui` tool to change light position, altitude, and colors.

The `lightmui` display plots lights as circular markers whose `facecolor` indicates the light color. To change the position of a light, click and drag the circular marker. Alternatively, right-clicking the circular marker summons a dialog box for changing the position or color of the light object. Clicking the color bar in that dialog box invokes the `uisetcolor` dialog box that can be used to specify or pick a color for the light.

1 Load the `topo` DTM files, and set up an orthographic projection:

   ```
   load topo
   axesm ('mapprojection','ortho', 'origin',[10 -20 0])
   ```

2 Plot the topography and assign a topographic colormap:

   ```
   meshm(topo,topolegend);
   demcmap(topo)
   ```

3 Set up a yellow light source over New York City:

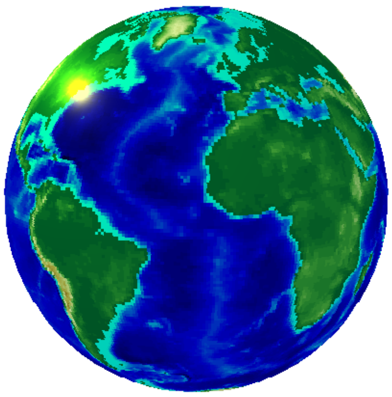
   ```
   hl = lightm(40.75,-73.9, 500/almanac('earth','radius'),...
   'color','yellow', 'style', 'local');
   ```

   The first two arguments to `lightm` are the latitude and longitude of the light source. The third argument is its altitude, in units of Earth radii (in this case they are in kilometers, the default units of `almanac`).

4 The surface is quite dark, so give it more reflectivity by specifying

   ```
   material([0.7270  1.5353  1.9860  4.0000  0.9925]);
   lighting phong; hidem(gca)
   ```

   The lighted orthographic map looks like this.



5 If you want, add more lights, as follows:

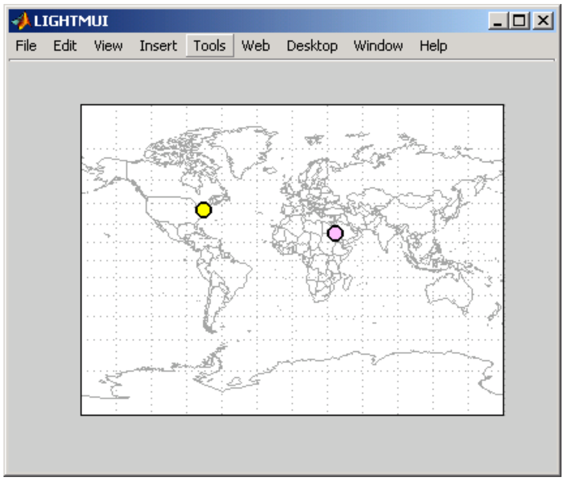
```
h2 = lightm(20,40, .1,'color','magenta', 'style', 'local')
```

The second light is magenta, and positioned over the Gulf of Arabia.

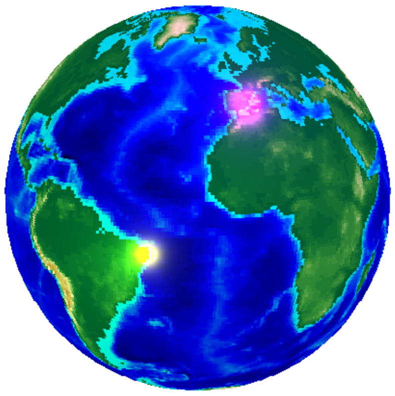
6 To modify the lights, use the `lightmui` GUI, which lets you drag lights across a world map and specify their color and altitudes:

```
lightmui(gca)
```

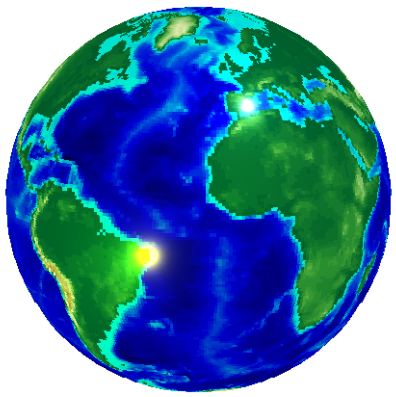
The lights are shown as appropriately colored circles, which you can drag to new positions. You can also **Ctrl**+click a circle to bring up a dialog box for directly specifying that light's position, altitude, and color. The GUI and the map look like this at this point.



7 In the `lightmui` window, drag the yellow light to the eastern tip of Brazil, and drag the magenta light to the Straits of Gibraltar.



8 **Ctrl**+click or **Shift**+click the magenta circle in the `lightmui` window. A second UI, for setting light position and color, opens. Set the **altitude** to `0.04` (Earth radii). Set the light **color** components to `1.0` (red), `0.75` (green), and `1.0` (blue). Press **Return** after each action. The colorbar on the UI changes to indicate the color you set. If you prefer to pick a color, click on the colorbar to bring up a color-choosing UI. The map now looks like this.



For additional information, see the reference pages for `lightm` and `lightmui`.

Surface Relief Shading

You can make dimensional monochrome shaded-relief maps with the function `surflm`, which is analogous to the MATLAB `surfl` function. The effect of `surflm` is similar to using lights, but the function models illumination itself (with one "light source" that you specify when you invoke it, but cannot reposition) by weighting surface normals rather than using light objects.

Shaded relief maps of this type are usually portrayed two-dimensionally rather than as perspective displays. The `surflm` function works with any projection except `globe`.

The `surflm` function accepts geolocated data grids only. Recall, however, that regular data grids are a subset of geolocated data grids, to which they can be converted using `meshgrat` (see "Fitting Gridded Data to the Graticule" on page 4-69). The following example illustrates this procedure.

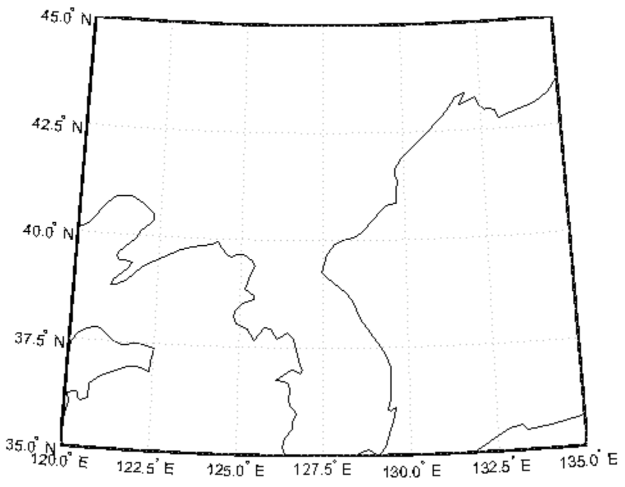
Creating Monochrome Shaded Relief Maps Using surflm

As stated above, `surflm` simulates a single light source instead of inserting light objects in a figure. Conduct the following exercise with the `korea` data set to see how `surflm` behaves. It uses `worldmap` to set up an appropriate map axes and reference outlines.

1 Set up a projection and display a vector map of the Korean peninsula with `worldmap`:

```
figure;
ax = worldmap('korea');

latlim = getm(ax,'MapLatLimit');
lonlim = getm(ax,'MapLonLimit');

coastline = shaperead('landareas',...
    'UseGeoCoords', true,...
    'BoundingBox', [lonlim' latlim']);

geoshow(ax, coastline, 'FaceColor', 'none');
```

`worldmap` chooses a projection and map bounds to make this map.



2 Load the `korea` terrain model:

```
load korea
```

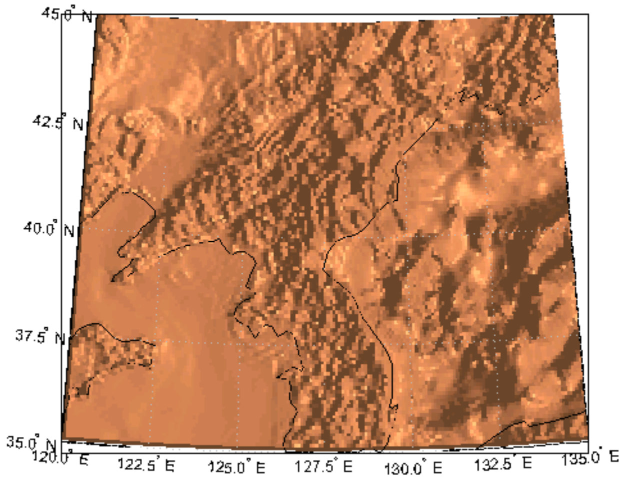
3 Generate the grid of latitudes and longitudes to transform the regular data grid to a geolocated one:

```
[klat,klon] = meshgrat(map,refvec);
```

4 Use `surflm` to generate a default shaded relief map, and change the colormap to a monochromatic scale, such as `gray`, `bone`, or `copper`.

```
ht = surflm(klat,klon,map);
colormap('copper')
```

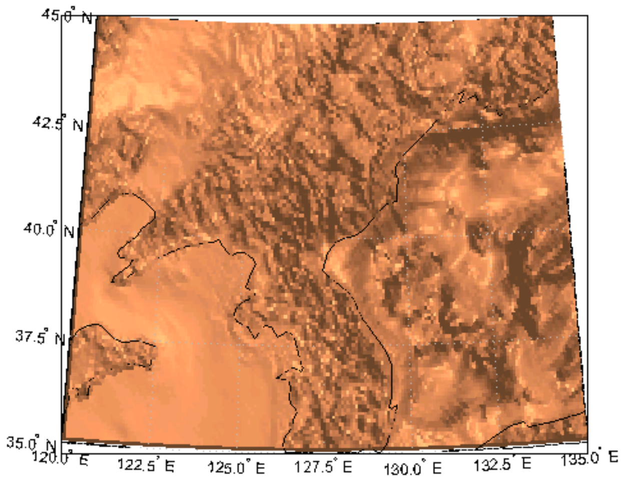
In this default case, the lighting direction is set at 45° counterclockwise from the viewing direction; thus the "sun" is in the southeast. This map is shown below.



5 To make the light come from some other direction, specify the light source's azimuth and elevation as the fourth argument to `surflm`. Clear the terrain map and redraw it, specifying an azimuth of 135° (northeast) and an elevation of 60° above the horizon:

```
clmo(ht); ht=surflm(klat,klon,map,[135,60]);
```

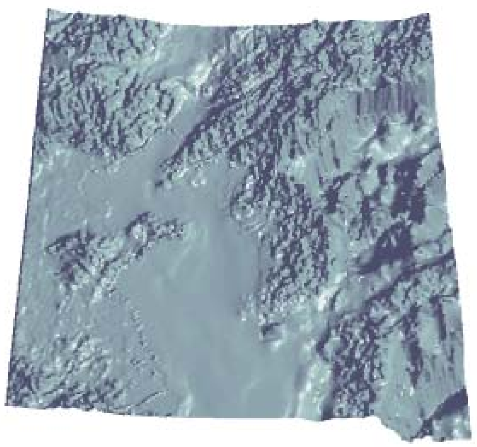
The surface lightens and has a new character because it is lit closer to overhead and from a different direction.



6 Now shift the light to the northwest (-135º azimuth), and lower it to 40º above the horizon. Because a lower "sun" decreases the overall reflectance when viewed from straight above, also specify a more reflective surface as a fifth argument to `surflm`. This is a 1-by-4 vector describing relative contributions of ambient light, diffuse reflection, specular reflection, and a specular shine coefficient. It defaults to `[.55 .6 .4 10]`.

```
clmo(ht); ht=surflm(klat,klon,map,[-135, 30],[.65 .4 .3 10]);
```

This is a good choice for lighting this terrain, because of the predominance of mountain ridges that run from northeast to southwest, more or less perpendicular to the direction of illumination. Here is the final map.



For further information, see the reference pages for `surflm` and `surfl`.

Shaded relief representations can highlight the fine structure of the land and sea floor, but because of the monochromatic coloration, it is difficult to distinguish land from sea. The next section describes how to color such maps to set off land from water.

Colored Surface Shaded Relief

The functions `meshlsrm` and `surflsrm` display maps as shaded relief with surface coloring as well as light source shading. You can think of them as extensions to `surflm` that combine surface coloring and surface light shading. Use `meshlsrm` to display regular data grids and `surflsrm` to render geolocated data grids.

These two functions construct a new colormap and associated `CData` matrix that uses grayscales to lighten or darken a matrix component based on its calculated surface normal to a light source. While there are no analogous MATLAB display functions that work like this, you can obtain similar results using MATLAB light objects, as "Relief Mapping with Light Objects" on page 5-36 explains.

Coloring Shaded Relief Maps and Viewing Them in 3-D

In this exercise, you use `surflsrm` in a way similar to how you used `surflm` in the preceding exercise, “Creating Monochrome Shaded Relief Maps Using surflm” on page 5-29. In addition, you set a vertical scale and view the map from various perspectives.

1 Start with a new map axes and the `korea` data, and then georeference the regular data grid:

```
load korea
[klat,klon] = meshgrat(map,refvec);
axesm miller
```

2 Create a colormap for DEM data; it is transformed by `surflsm` to shade relief according to how you specify the sun’s altitude and azimuth:

```
[cmap,clim] = demcmap(map);
```

3 Plot the colored shaded relief map, specifying an azimuth of -135º and an altitude of 50º for the light source:

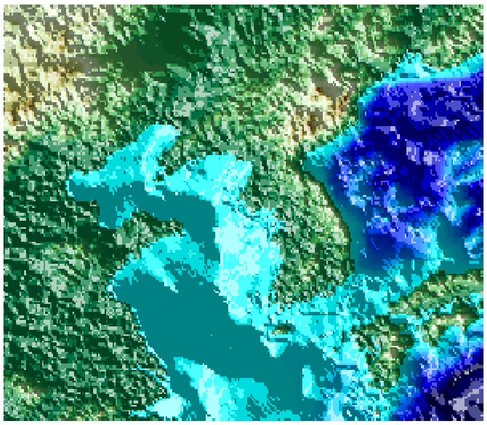
```
surflsrm(klat,klon,map,[-130 50],cmap,clim)
```

You can also achieve the same effect with `meshlsrm`, which operates on regular data grids (it first calls `meshgrat`, just as you just did), e.g., `meshlsrm(map,maplegend)`.

4 The surface has more contrast than if it were not shaded, and it might help to lighten it uniformly by 25% or so:

```
brighten(.25)
```

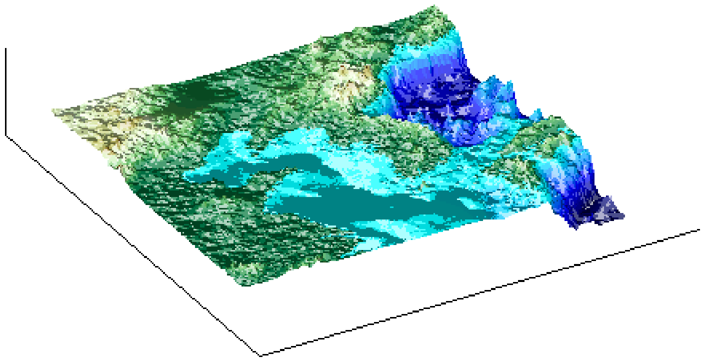
The map, which has an overhead view, looks like this.



5 Plot an oblique view of the surface by hiding its bounding box, exaggerating terrain relief by a factor of 50, and setting the view azimuth to -30° (south-southwest) and view altitude to 30° above the horizon:

```
set(gca,'Box','off')
daspectm('meters',50)
view(-30,30)
```

The map now looks like this.



6 You can continue to rotate the perspective with the `view` function (or interactively with the **Rotate 3D** tool in the figure window), and to change the vertical exaggeration with the `daspectm` function. You cannot change the built-in lighting direction without generating a new view using surflsrm.

For further information, see the reference pages for surflsrm, meshlsrm, daspectm, and view.

Relief Mapping with Light Objects

In the exercise “Lighting a Global Terrain Map with lightm and lightmui” on page 5-25, you created light objects to illuminate a Globe display. In the following one, you create a light object to mimic the map produced in the previous exercise (“Coloring Shaded Relief Maps and Viewing Them in 3-D” on page 5-34), which uses shaded relief computations rather than light objects.

The meshlsrm and surflsrm functions simulate lighting by modifying the colormap with bands of light and dark. The map matrix is then converted to indices for the new “shaded” colormap based on calculated surface normals. Using light objects allows for a wide range of lighting effects. The toolbox manages light objects with the lightm function, which depends upon the MATLAB light function. Lights are separate MATLAB graphic objects, each with its own object handle.

Colored 3-D Relief Maps Illuminated with Light Objects

As a comparison to the lighted shaded relief example shown earlier, add a light source to the surface colored data grid of the Korean peninsula region:

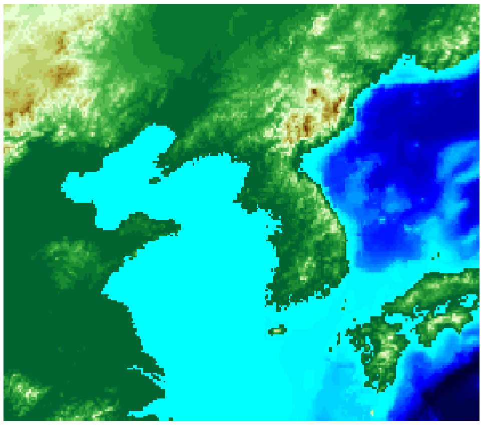
1 If you need to, load the korea DEM, and create a map axes using the Miller projection:

```
load korea
figure; axesm('MapProjection','miller',...
     'MapLatLimit',[30 45],'MapLonLimit',[115 135])
```

2 Display the DEM with meshm, and color it with terrain hues:

```
meshm(map,refvec,size(map),map);
demcmap(map)
```

The map, without lighting effects, looks like this.



3 Create a light object with `lightm` (similar to the MATLAB `light` function, but specifies position with latitude and longitude rather than *x, y, z*). The light is placed at the northwest corner of the grid, one degree high:

```
h=lightm(45,115,1)
```

The figure becomes darker.

4 To see any relief in perspective, it is necessary to exaggerate the vertical dimension. Use a factor of 50 for this:

```
daspectm('meters',50)
```

The figure becomes darker still, with highlights at peaks.

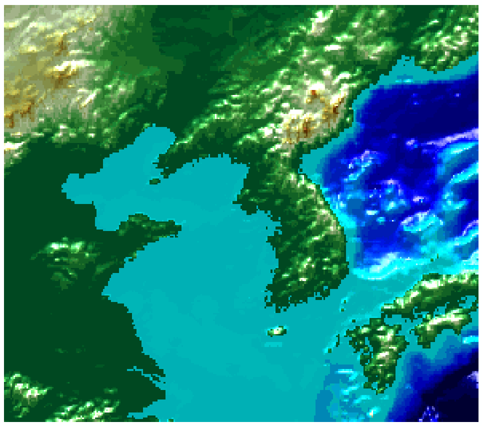
5 Set the ambient (direct), diffuse (skylight), and specular (highlight) surface reflectivity characteristics, respectively:

```
material ([.7, .9, .8])
```

6 By default, the lighting is flat (plane facets). Change this to Phong shading (interpolated normal vectors at facet corners):

```
lighting phong
```

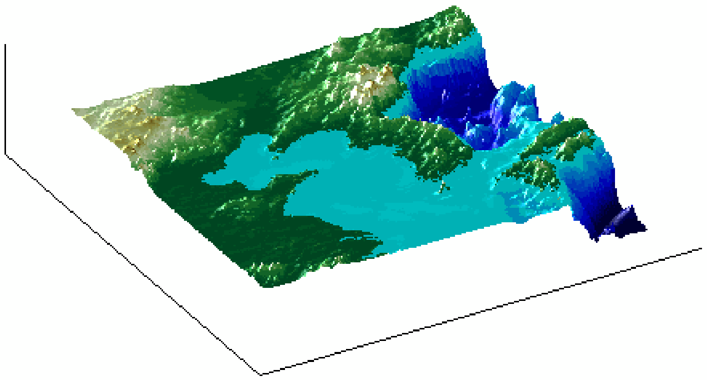
The map now looks like this.



7 Finally, remove the edges of the bounding box and set a viewpoint of -30° azimuth, 30° altitude:

```
set(gca,'Box','off')
view(-30,30)
```

The view from (-30,30) with one light at (45,115,1) and Phong shading is shown below. Compare it to the final map in the previous exercise, "Coloring Shaded Relief Maps and Viewing Them in 3-D" on page 5-34.



To remove a light (when there is only one) from the current figure, type

```
clmo(handlem('light'))
```

For more information, consult the reference pages for `lightm`, `daspectm`, `material`, `lighting`, and `view`, along with the section on “Lighting as a Visualization Tool” in the 3–D Visualization documentation.

Draping Data on Elevation Maps

In this section...

Draping Geoid Heights over Topography

Lighting effects can provide important visual cues when elevation maps are combined with other kinds of data. The shading resulting from lighting a surface makes it possible to "drape" satellite data over a grid of elevations. It is common to use this kind of display to overlay georeferenced land cover images from Earth satellites such as LANDSAT and SPOT on topography from digital elevation models. Mapping Toolbox displays use variations of techniques described in the previous section.

When the elevation and image data grids correspond pixel-for-pixel to the same geographic locations, you can build up such displays using the optional altitude arguments in the surface display functions. If they do not, you can interpolate one or both source grids to a common mesh.

The following example shows the figure of the Earth (the `geoid` data set) draped on topographic relief (the `topo` data set). The geoid data is shown as an attribute (using a color scale) rather than being depicted as a 3-D surface itself. The two data sets are both 1-by-1-degree meshes sharing a common origin.

Note The geoid can be described as the surface of the ocean in the absence of waves, tides, or land obstructions. It is influenced by the gravitational attraction of denser or lighter materials in the Earth's crust and interior and by the shape of the crust. A model of the geoid is required for converting ellipsoidal heights (such as might be obtained from GPS measurements) to orthometric heights. Geoid heights vary from a minimum of about 105 meters below sea level to a maximum of about 85 meters above sea level.

---

1 Begin by loading the `topo` and `geoid` regular data grids:

```
load topo
load geoid
```

2 Create a map axes using a Gall stereographic cylindrical projection (a perspective projection):

```
axesm gstereo
```

3 Use `meshm` to plot a colored display of the geoid's variations, but specify `topo` as the final argument, to give each `geoid` grid cell the height (z-value) of the corresponding `topo` grid cell:

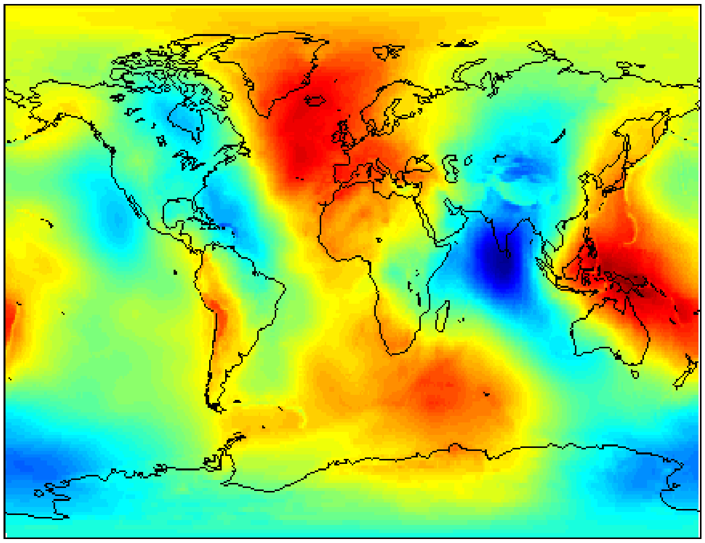
```
meshm(geoid,geoidrefvec,size(geoid),topo)
```

Low geoid heights are shown as blue, high ones as red.

4 For reference, plot the world coastlines in black, raise their elevation to 1000 meters (high enough to clear the surface in their vicinity), and expand the map to fill the frame:

```
load coast
plotm(lat,long,'k')
zdatam(handlem('allline'),1000)
tightmap
```

At this point the map looks like this.



5 Due to the vertical view and lack of lighting, the topographic relief is not visible, but it is part of the figure's surface data. Bring it out by exaggerating relief greatly, and then setting a view from the south-southeast:

```
daspectm('m',200); tightmap
view(20,35)
```

6 Remove the bounding box, shine a light on the surface (using the default position, offset to the right of the viewpoint), and render again with Phong shading:

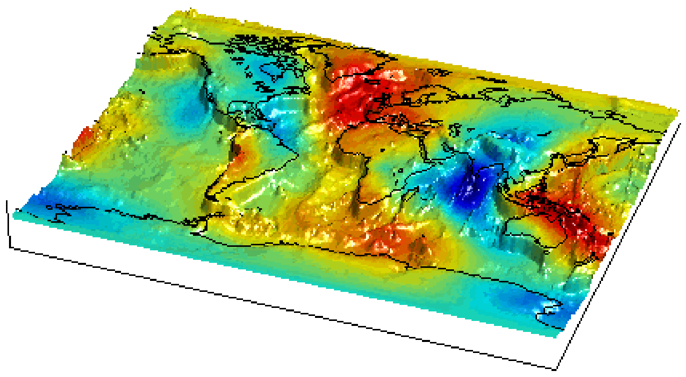
```
set(gca,'Box','off')
camlight;
lighting phong
```

7 Finally, set the perspective to converge slightly (the default perspective is orthographic):

```
set(gca,'projection','perspective')
```

The final map is shown below. From it, you can see that the geoid mirrors the topography of the major mountain chains such as the Andes, the

Himalayas, and the Mid-Atlantic Ridge. You can also see that large areas of high or low geoid heights are not simply a result of topography.



Draping Data over Terrain with Different Gridding

If you want to combine elevation and attribute (color) data grids that cover the same region but are gridded differently, you must resample one matrix to be consistent with the other. It helps if at least one of the grids is a geolocated data grid, because their explicit horizontal coordinates allow them to be resampled using the `ltln2val` function. To combine dissimilar grids, you can do one of the following:

- Construct a geolocated grid version of the regular data grid values.
- Construct a regular grid version of the geolocated data grid values.

The following two examples illustrate these closely related approaches.

Draping via Converting a Regular Grid to a Geolocated Data Grid

This example drapes slope data from a regular data grid on top of elevation data from a geolocated data grid. Although the two data sets actually have the same origin (the geolocated grid derives from the `topo` data set), this approach works with any dissimilar grids. The example uses the geolocated data grid as the source for surface elevations and transforms the regular data grid into slope values, which are then sampled to conform to the geolocated data grid (creating a set of slope values for the diamond-shaped grid) and color-coded for surface display.

Note When you use `ltln2val` to resample a regular data grid over an irregular area, make sure that the regular data grid completely covers the area of the geolocated data grid.

1. Begin by loading the geolocated data grids from `mapmtx`, which contains two regions. You will only use the diamond-shaped portion of `mapmtx` (`lt1`, `lg1`, and `map1`) centered on the Middle East, not the `lt2`, `lg2`, and `map1` data:

   ```
   load mapmtx lt1
   load mapmtx lg1
   load mapmtx map1
   ```

   Load the `topo` global regular data grid:

   ```
   load topo
   ```

2. Compute surface aspect, slope, and gradients for `topo`. You use only the slopes in subsequent steps:

   ```
   [aspect,slope,gradN,gradE] = gradientm(topo,topolegend);
   ```

3. Use `ltln2val` to interpolate slope values to the geolocated grid specified by `lt1`, `lg1`:

   ```
   slope1 = ltln2val(slope,topolegend,lt1,lg1);
   ```

   The output is a 50-by-50 grid of elevations matching the coverage of the `map1` variable.

4. Set up a figure with a Miller projection and use `surfm` to display the slope data. Specify the *z*-values for the surface explicitly as the `map1` data, which is terrain elevation:

   ```
   figure; axesm miller
   surfm(lt1,lg1,slope1,map1)
   ```

The map mainly depicts steep cliffs, which represent mountains (the Himalayas in the northeast), and continental shelves and trenches.

5 The coloration depicts steepness of slope. Change the colormap to make the steepest slopes magenta, the gentler slopes dark blue, and the flat areas light blue:

```
colormap cool;
```

6 Use `view` to get a southeast perspective of the surface from a low viewpoint:

```
view(20,30); daspectm('m',200)
```

In 3-D, you immediately see the topography as well as the slope.

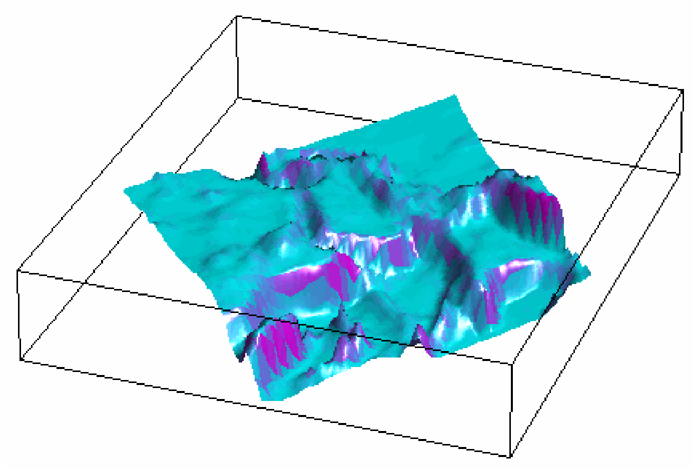
7 The default rendering uses faceted shading (no smooth interpolation). Render the surface again, this time making it shiny with Phong shading and lighting from the east (the default of `camlight` lights surfaces from over the viewer's right shoulder):

```
material shiny;camlight;lighting phong
```

8 Finally, remove white space and re-render the figure in perspective mode:

```
axis tight; set(gca,'Projection','Perspective')
```

Here is the mapped result.



Draping a Geolocated Grid on Regular Data Grid via Texture Mapping

The second way to combine a regular and a geolocated data grid is to construct a regular data grid of your geolocated data grid's *z*-data. This approach has the advantage that more computational functions are available for regular data grids than for geolocated ones. Another aspect is that the color and elevation grids do not have to be the same size. If the resolutions of the two are different, you can create the surface as a three-dimensional elevation map and later apply the colors as a texture map. You do this by setting the surface `Cdata` property to contain the color matrix, and setting the surface face color to `'TextureMap'`.

In the following steps, you create a new regular data grid that covers the region of the geolocated data grid, then embed the color data values into the new matrix. The new matrix might need to have somewhat lower resolution than the original, to ensure that every cell in the new map receives a value.

1 Load the `topo` and terrain data from `mapmtx`:

```
load topo;
load mapmtx lt1
load mapmtx lg1
load mapmtx map1
```

2 Identify the geographic limits of one of the `mapmtx` geolocated grids:

```
latlim = [min(lt1(:)) max(lt1(:))];
lonlim = [min(lg1(:)) max(lg1(:))];
```

3 Trim the `topo` data to the rectangular region enclosing the smaller grid:

```
[topo1,topo1ref] = maptrims(topo,topolegend,latlim,lonlim);
```

4 Create a regular grid filled with `NaN`s to receive texture data:

```
[curve1,curve1ref] = nanm(latlim,lonlim,.5);
```

The last parameter establishes the grid at 1/.5 cells per degree.

5 Use `imbedm` to embed values from `map1` into the `curve1` grid; the values are the discrete Laplacian transform (the difference between each element of the `map1` grid and the average of its four orthogonal neighbors):

```
curve1 = imbedm(lt1,lg1,del2(map1),curve1,curve1ref);
```

6 Set up a map axes with the Miller projection and use `meshm` to draw the `topo1` extract of the `topo` DEM:

```
figure; axesm miller
h = meshm(topo1,topo1ref,size(topo1),topo1);
```

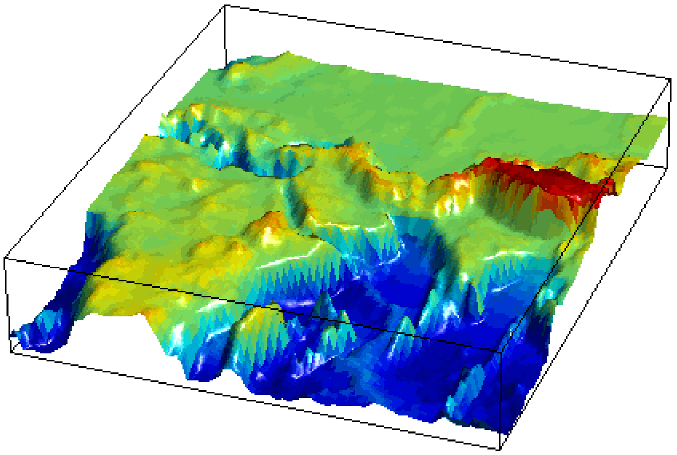
7 Render the figure as a 3-D view from a 20º azimuth and 30º altitude, and exaggerate the vertical dimension by a factor of 200:

```
view(20,30); daspectm('m',200)
```

8 Light the view and render with Phong shading in perspective:

```
material shiny; camlight; lighting phong
axis tight; set(gca,'Projection','Perspective')
```

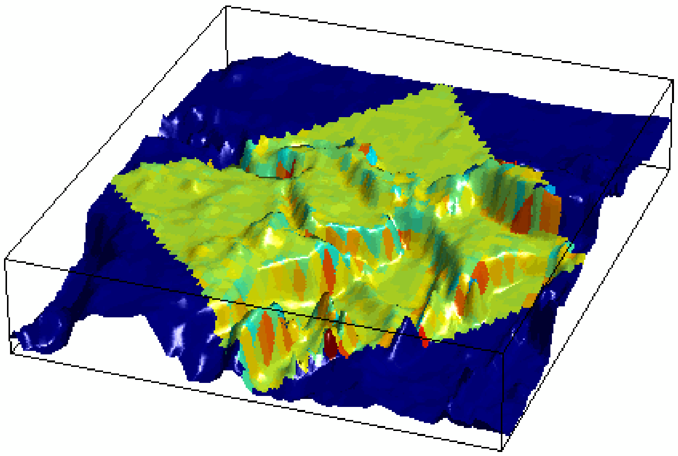
So far, both the surface relief and coloring represent topographic elevation.



9 Apply the `curve1` matrix as a texture map directly to the figure using the `set` function:

```
set(h,'Cdata',curve1,'FaceColor','TextureMap')
```

The area originally covered by the `[lt1, lg1, map1]` geolocated data grid, and recoded via the Laplacian transform as `curve1`, now controls color symbolism, with the `NaN`-coded outside cells rendered in black.



Working with the Globe Display

In this section...

What Is the Globe Display?

The *Globe display* is a three-dimensional view of geospatial data capable of mapping terrain relief or other data for an entire planet viewed from space. Its underlying transformation maps latitude, longitude, and elevation to a three-dimensional Cartesian frame. All Mapping Toolbox projections transform latitudes and longitudes to map *x*- and *y*-coordinates. The `globe` function is special because it can render relative relief of elevations above, below, or on a sphere. In Earth-centered Cartesian (*x*,*y*,*z*) coordinates, *z* is not an optional elevation; rather, it is an axis in Cartesian three-space. `globe` is useful for geospatial applications that require three-dimensional relationships between objects to be maintained, such as when one simulates flybys, and/or views planets as they rotate.

The Globe display is based on a *coordinate transformation*, and is not a map projection. While it has none of the distortions inherent in planar projections, it is a three-dimensional model of a planet that cannot be displayed without distortion or in its entirety. That is, in order to render the globe in a figure window, either a perspective or orthographic transformation must be applied, both of which necessarily involve setting a viewpoint, hiding the back side and distortions of shape, scale, and angles.

The Globe Display Compared with the Orthographic Projection

The following example illustrates the differences between the two-dimensional orthographic projection, which looks spherical but is really flat, and the three-dimensional Globe display. Use the **Rotate 3D** tool to manipulate the display.

1 Load the `topo` data set and render it with an orthographic map projection:

```
load topo
axesm ortho; framem
meshm(topo,topolegend);demcmap(topo)
```

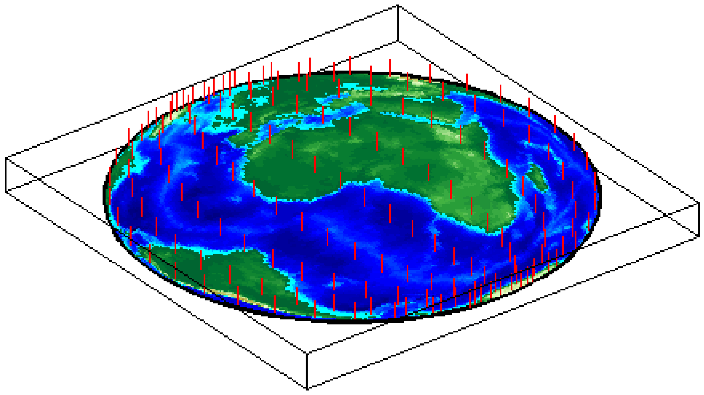
2 View the map obliquely:

```
view(3); daspectm('m',1)
```

3 You can view it in 3-D from any perspective, even from underneath. To visualize this, define a geolocated data grid with `meshgrat`, populate it with a constant *z*-value, and render it as a stem plot with `stem3m`:

```
[latgrat,longrat] = meshgrat(topo,topolegend,[20 20]);
stem3m(latgrat,longrat,500000*ones(size(latgrat)),'r')
```

Use the **Rotate 3D** tool on the figure window toolbar to change your viewpoint. No matter how you position the view, you are looking at a disc with stems protruding perpendicularly.



4 Create another figure using the Globe transform rather than orthographic projection:

```
figure
axesm('globe','Geoid',almanac('earth','radius','m'))
```

5 Display the `topo` surface in this figure and view it in 3-D:

```
meshm(topo,topolegend); demcmap(topo)
view(3)
```

6 Include the stem plot to visualize the difference in surface normals on a sphere:

```
stem3m(latgrat,longrat,500000*ones(size(latgrat)),'r')
```

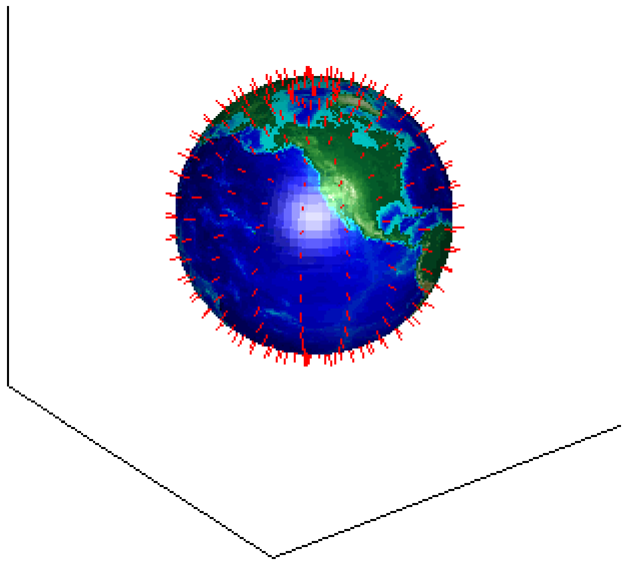
7 You can apply lighting to the display, but its location is fixed, and does not move as the camera position is shifted:

```
camlight('headlight','infinite')
```

8 If you prefer a more unobstructed view, hide the 3-D axes:

```
set(gca,'Box','off')
```

Here is a representative view using the Globe display with the headlight.



You can use the `LabelRotation` property when you use the Orthographic or any other Mapping Toolbox projection to align meridian and parallel labels with the graticule. Because the Globe display is not a true map projection and is handled differently internally, `LabelRotation` does not work with it.

For additional information on functions used in this example, see the reference pages for `view`, `camlight`, `meshgrat`, and `stem3m`.

Using Opacity and Transparency in Globe Displays

Because Globe displays depict 3-D objects, you can see into and through them as long as no opaque surfaces (e.g., patches or surfaces) obscure your view. This can be particularly disorienting for point and line data, because features on the back side of the world are reversed and can overlay features on the front side.

Here is one way to create an opaque surface over which you can display line and point data:

1. Create a figure and set up a Globe display:

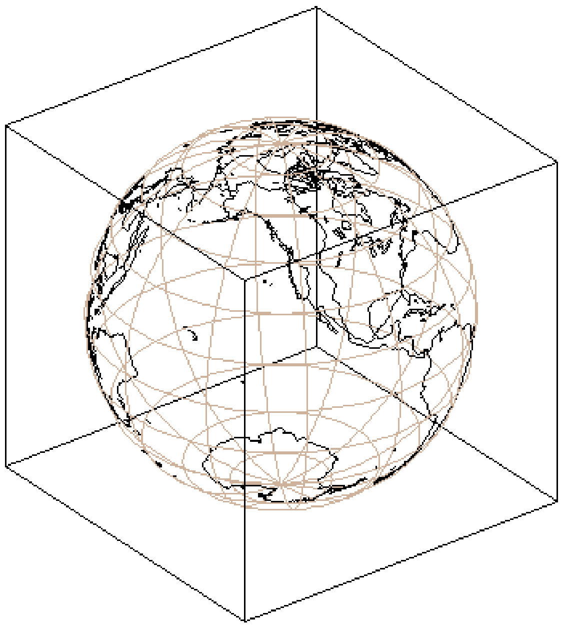
```
figure; axesm('globe')
```

2 Draw a graticule in a light color, slightly raised from the surface:

```
gridm('GLineStyle','-','Gcolor',[.8 .7 .6],'Galtitude', .02)
```

3 Load and plot the coast data in black, and set up a 3-D perspective:

```
load coast
plot3m(lat,long,.01,'k')
view(3)
```



4 Use the **Rotate 3D** tool on the figure's toolbar to rotate the view. Note how confusing the display is because of its transparency.

5 Make a uniform 1-by-1-degree grid and a referencing matrix for it:

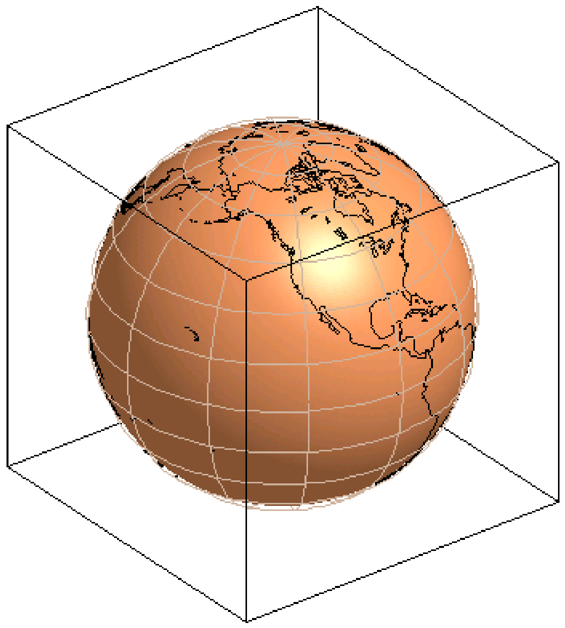
```
base = zeros(180,360);
baseR = makerefmat('RasterSize', size(base), ...
   'ColumnsStartFrom', 'north', 'RowsStartFrom', 'west');
```

6 Render the grid onto the globe, color it copper, light it from camera right, and make the surface reflect more light:

```
hs = meshm(base,baseR,size(base));
colormap copper
camlight right
material([.8 .9 .4])
```

Note Another way to make the surface of the globe one color is to change the `FaceColor` property of a displayed surface mesh (e.g., `topo`).

If you haven't rotated it, the display looks like this.



When you manually rotate this map, its movement can be jerky due to the number of vectors that must be redisplayed. In any position, however, the copper surface effectively hides all lines on the back side of the globe.

Note The technique of using a uniform surface to hide rear-facing lines has limitations for the display of patch symbolism (filled polygons). As patch polygons are represented as planar, in three-space the interiors of large patches can intersect the spherical surface mesh, allowing its symbolism to show through.

Over-the-Horizon 3-D Views Using Camera Positioning Functions

You can create dramatic 3-D views using the Globe display. The `camtargm` and `camposm` functions (Mapping Toolbox functions corresponding to `camtarget` and `campos`) enable you to position focal point and a viewpoint, respectively, in geographic coordinates, so you do not need to deal with 3-D Cartesian figure coordinates.

In this exercise, you display coastlines from the `landareas` shapefile over topographic relief, and then view the globe from above Washington, D.C., looking toward Moscow, Russia.

1. Set up a Globe display and obtain topographic data for the map:

   ```
   figure
   axesm globe
   load topo
   ```

2. Display `topo` without the vertical component (by omitting the fourth argument to `meshm`):

   ```
   meshm(topo, topolegend, size(topo)); demcmap(topo);
   ```

   The default view is from above the North Pole with the central meridian running parallel to the *x*-axis.

3. Add world coastlines from the global `landareas` shapefile and plot them in light gray:

   ```
   coastlines = shaperead('landareas',...
       'UseGeoCoords', true, 'Attributes', {});
   plotm([coastlines.Lat], [coastlines.Lon], 'Color', [.7 .7 .7])
   ```

4 Read the coordinate locations for Moscow and Washington from the `worldcities` shapefile:

```
moscow = shaperead('worldcities',...
    'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'Moscow'), 'Name'});
washington = shaperead('worldcities',...
    'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'Washington D.C.'),...
    'Name'});
```

5 Create a great circle track to connect Washington with Moscow and plot it in red:

```
[latc,lonc] = track2('gc',...
    moscow.Lat, moscow.Lon, washington.Lat, washington.Lon);
plotm(latc,lonc,'r')
```

6 Point the camera at Moscow. Wherever the camera is subsequently moved, it always looks toward `[moscow.Lat moscow.Lon]`:

```
camtargm(moscow.Lat, moscow.Lon, 0)
```

7 Station the camera above Washington. The third argument is an altitude in Earth radii:

```
camposm(washington.Lat, washington.Lon, 3)
```

8 Establish the camera up vector with the camera target's coordinates. The great circle joining Washington and Moscow now runs vertically:

```
camupm(moscow.Lat, moscow.Lon)
```

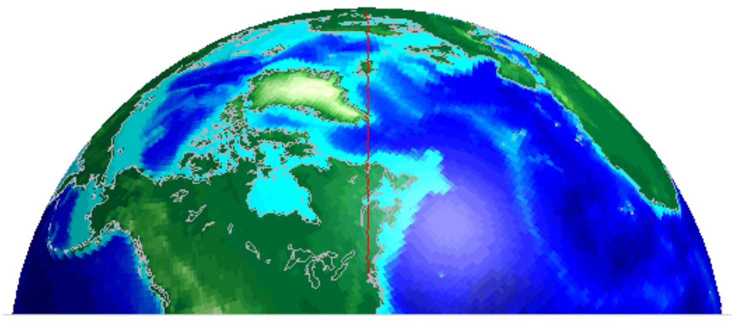
9 Set the field of view for the camera to 20° for the final view:

```
camva(20)
```

10 Add a light, specify a relatively nonreflective surface material, and hide the map background:

```
camlight; material(0.6*[ 1 1 1])
hidem(gca)
```

Here is the final view.



For additional information, see the reference pages for `extractm`, `camtargm`, `camposm`, `camupm`, `globe`, and `camlight`.

Displaying a Rotating Globe

Because the Globe display can be viewed from any angle without the need to recompute a projection, you can easily animate it to produce a rotating globe. If the displayed data is simple enough, such animations can be redrawn at relatively fast rates. In this exercise, you progressively add or replace features on a Globe display and rotate it under the control of an M-file that resets the view to rotate the globe from west to east in one-degree increments.

1 In the Mapping Toolbox editor, create an M-file containing the following code:

```
% spin.m: Rotates a view around the equator one revolution
% in 5-degree steps. Negative step makes it rotate normally
% (west-to-east).
for i=360:-5:0
    view(i,23.5);     %  Earth's axis tilts by 23.5 degrees
    drawnow
end
```

Save this as `spin.m` in your current directory or on the Mapping Toolbox path. Note that the azimuth parameter for the figure does not have the same origin as geographic azimuth: it is 90 degrees to the west.

2 Set up a Globe display with a graticule, as follows:

```
axesm('globe','Grid','on','Gcolor',[.7 .8 .9],'GlineStyle','-')
```

The view is from above the North Pole.

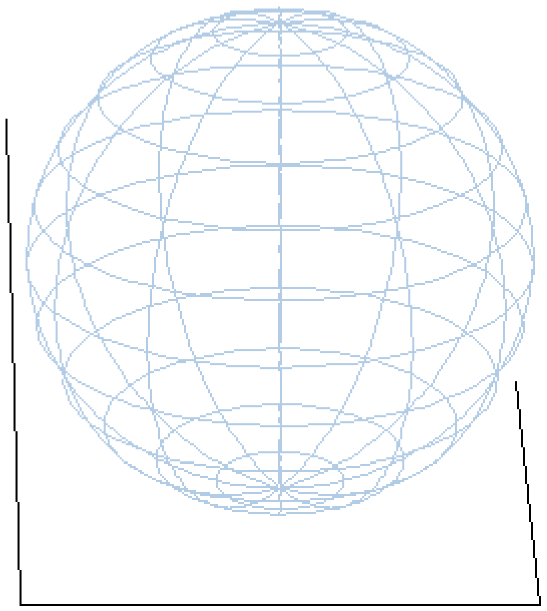
3 Show the axes, but hide the edges of the figure's box, and view it in perspective rather than orthographically (the default perspective):

```
set(gca, 'Box','off', 'Projection','perspective')
```

4 Spin the globe one revolution with your M-file:

```
spin
```

The globe spins rapidly. The last position looks like this.



5 To make the globe opaque, create a sea-level data grid as you did for the previous exercise, "Using Opacity and Transparency in Globe Displays" on page 5-52:

```
base = zeros(180,360); baseref = [1 90 0];
hs = meshm(base,baseref,size(base));
colormap copper
```

The globe now is a uniform dark copper color with the grid overlaid.

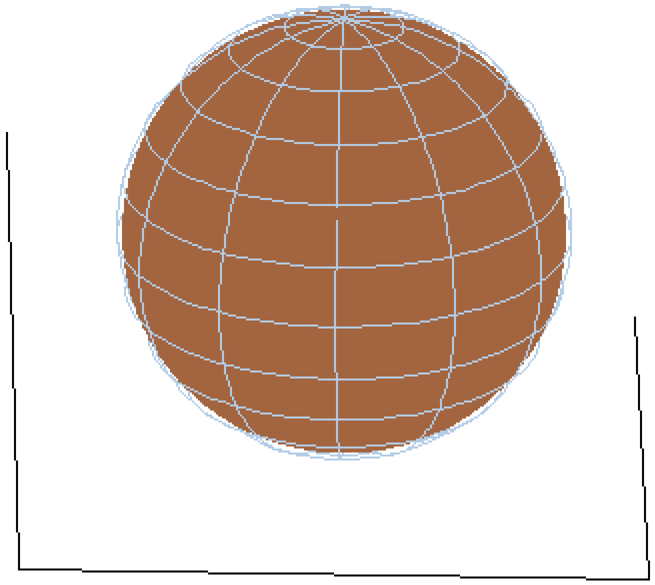
6 Pop up the grid so it appears to float 2.5% above the surface. Prevent the display from stretching to fit the window with the `axis vis3d` command:

```
setm(gca, 'Galtitude',0.025);
axis vis3d
```

7 Spin the globe again:

```
spin
```

The motion is slower, due to the need to rerender the 180-by-360 mesh: The last frame looks like this.



8 Get ready to replace the uniform sphere with topographic relief by deleting the copper mesh:

```
clmo(hs)
load topo
```

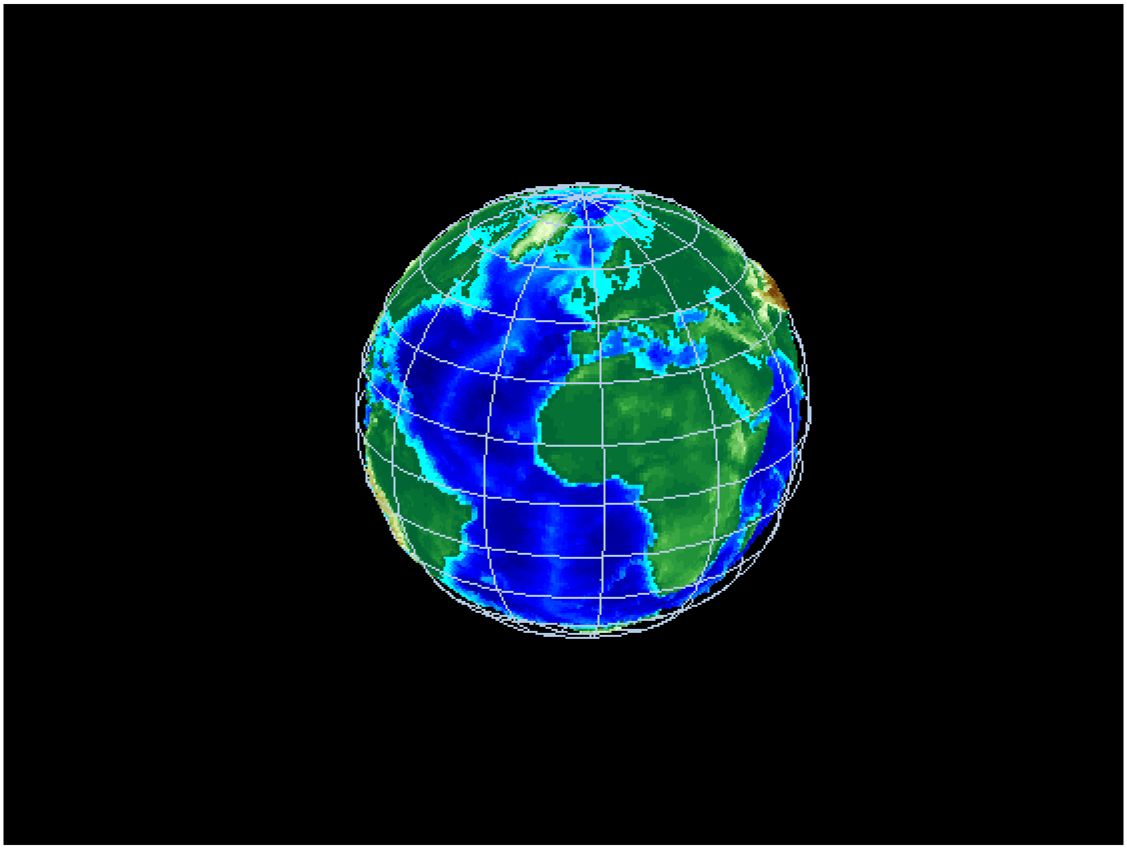
9 Scale the elevations to have an exaggeration of 50 (in units of Earth radii) and plot the surface:

```
topo = topo / (almanac('earth','radius')* 20);
hs = meshm(topo,topolegend,size(topo),topo);
demcmap(topo)
```

10 Show the Earth in space; blacken the figure background, turn off the three axes, and spin again:

```
set(gcf,'color','black');
axis off;
spin
```

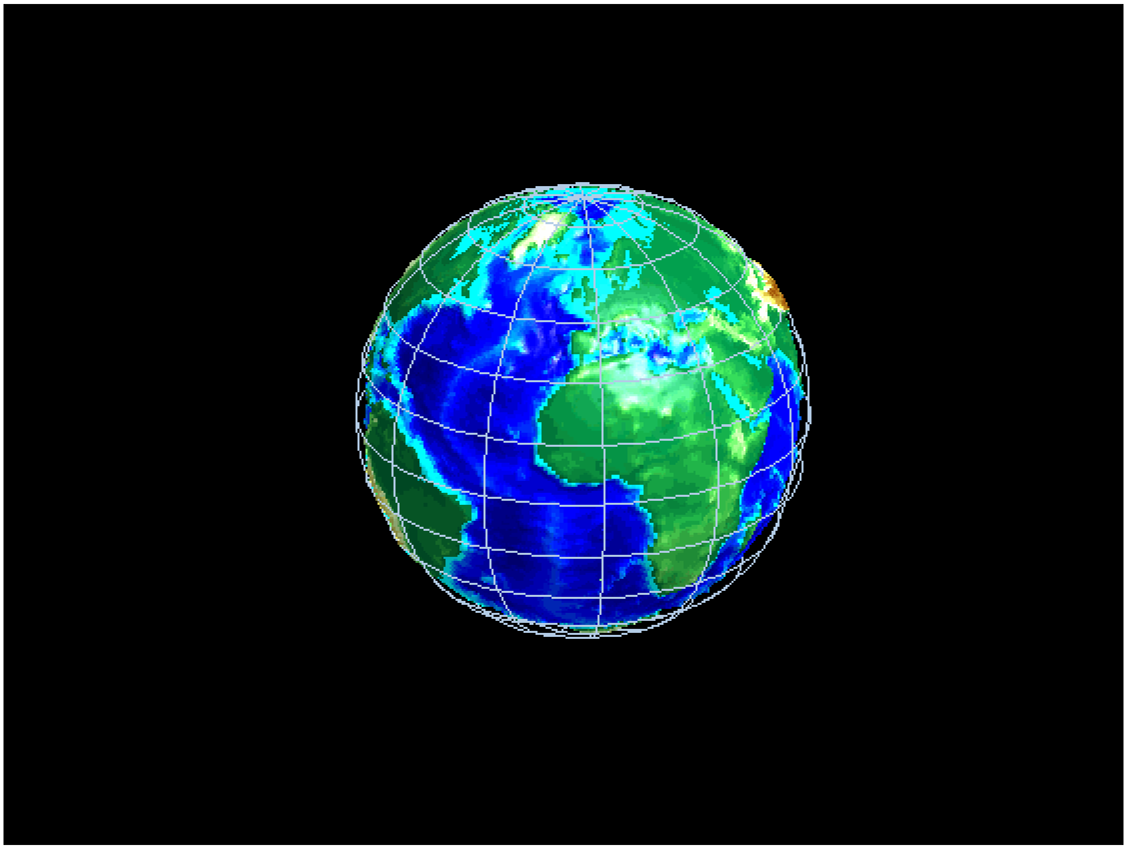
Here is a representative view, showing the Himalayas rising on the Eastern limb of the planet and the Andes on the Western limb.



11 You can apply lighting as well, which shifts as the planet rotates. Try the following settings, or experiment with others:

```
camlight right
lighting phong;
material ([.7, .9, .8])
```

Here is the illuminated version of the preceding view:



For additional information, see the `globe`, `camlight`, and `view` reference pages.

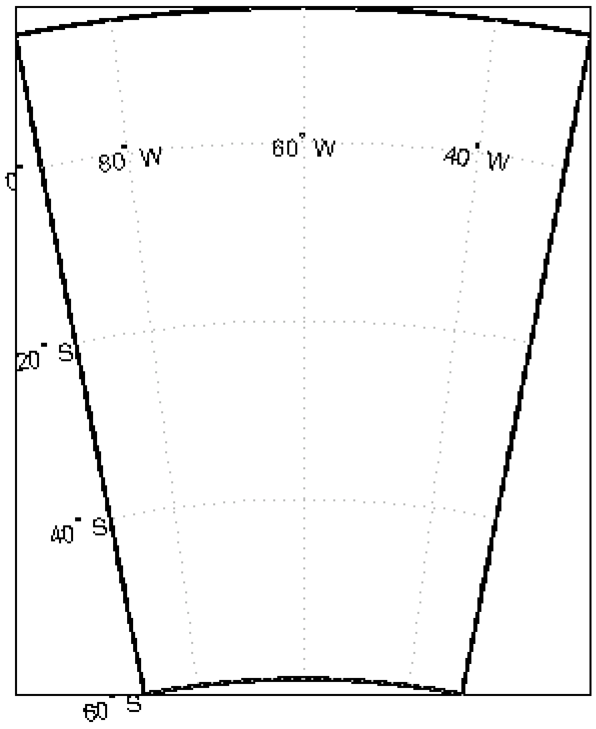
Customizing and Printing Maps

Inset Maps

Inset maps are often used to display widely separated areas, generally at the same scale, or to place a map in context by including overviews at smaller scales. You can create inset maps by nesting multiple axes in a figure and defining appropriate map projections for each. To ensure that the scale of each of the maps is the same, use `axesscale` to resize them. As an example, create an inset map of California at the same scale as the map of South America, to relate the size of that continent to a more familiar region:

1 Begin by defining a map frame for South America using `worldmap`:

```
figure
h1 = worldmap('south america');
```

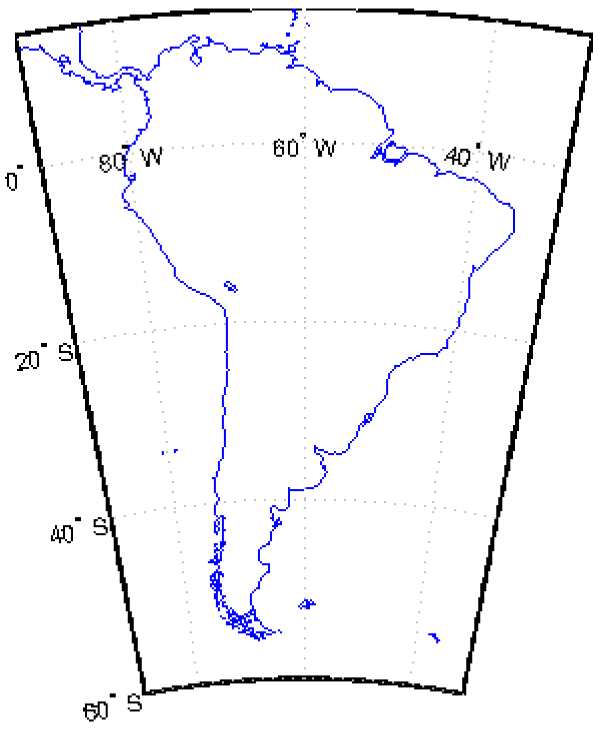


2 Use `shaperead` to read the demo world land areas polygon shapefile:

```
land = shaperead('landareas.shp', 'UseGeoCoords', true);
```

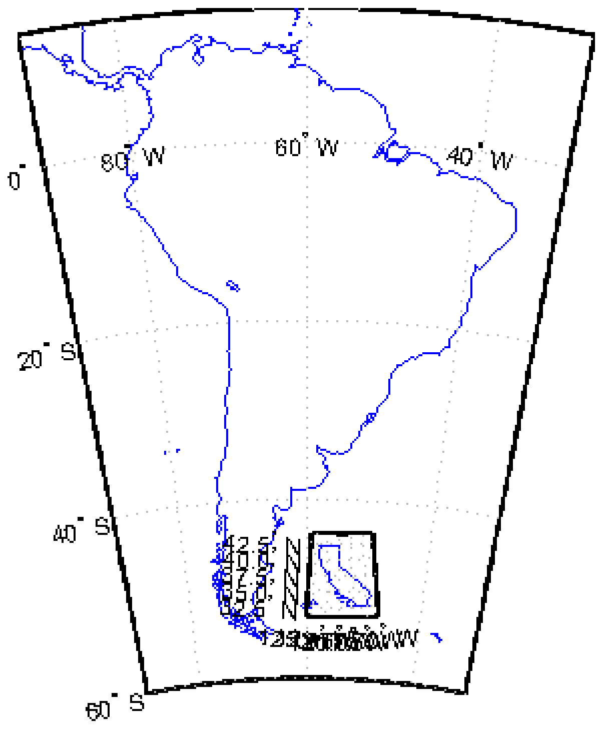
3 Display the data in the map axes:

```
geoshow([land.Lat],[land.Lon])
setm(h1,'FFaceColor','w') % set the frame fill to white
```



4 Place axes for an inset in the lower middle of the map frame, and project a line map of California:

```
h2 = axes('pos',[.5 .2 .1 .1]);
CA = shaperead('usastatehi', 'UseGeoCoords', true, ...
   'Selector', {@(name) isequal(name,'California'), 'Name'});
usamap('california')
geoshow([CA.Lat],[CA.Lon])
```

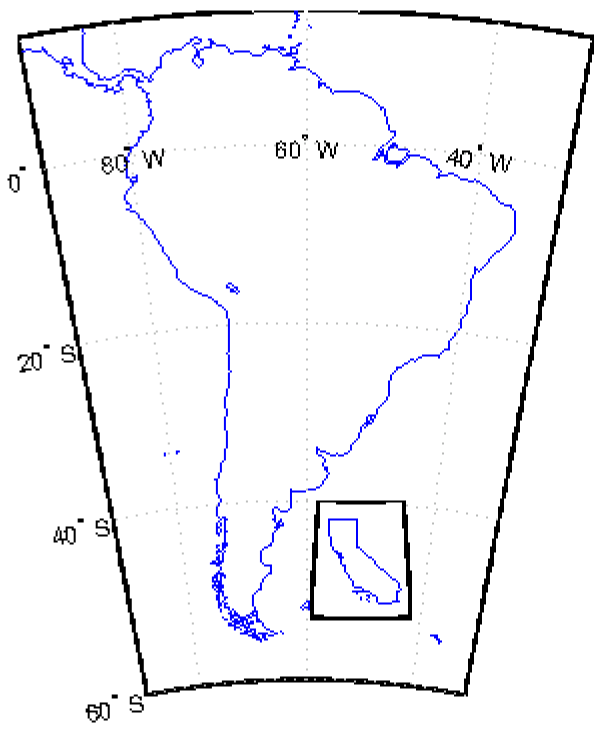


5 Set the frame fill color and set the labels:

```
setm(h2,'FFaceColor','w')
mlabel; plabel; gridm % toggle off
```

6 Make the scale of the inset axes, `h2` (California), match the scale of the original axes, `h1` (South America). Hide the map border:

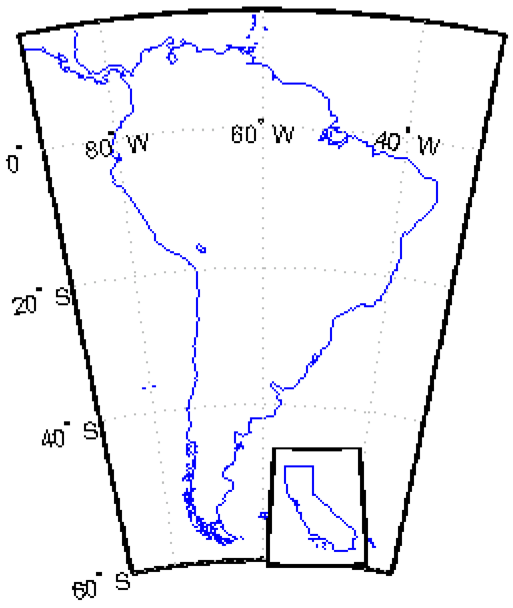
```
axesscale(h1)
set([h1 h2], 'Visible', 'off')
```



Note that the toolbox software chose a different projection and appropriate parameters for each region based on its location and shape. You can override these choices to make the two projections the same.

7 Find out what map projections are used, and then make South America's projection the same as California's:

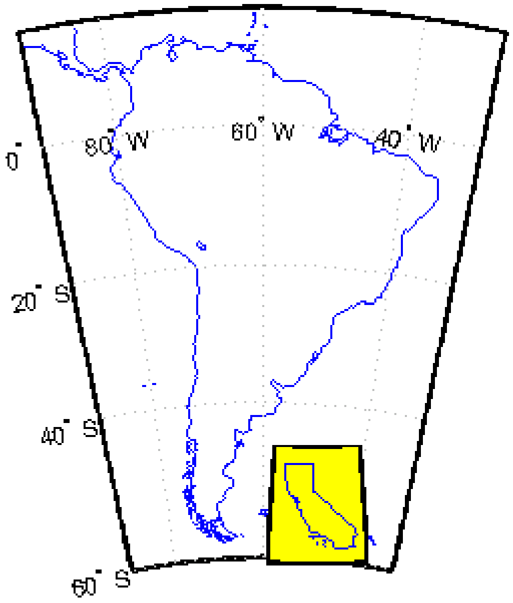
```
getm(h1, 'mapprojection')
ans =
      eqdconic

getm(h2, 'mapprojection')
ans =
      lambert

setm(h1, 'mapprojection', getm(h2, 'mapprojection'))
```



Note that the parameters for South America defaulted properly (those appropriate for California were not used).

8 Finally, experiment with changing properties of the inset, such as its color:

```
setm(h2, 'ffacecolor', 'y')
```


80° W
60° W
40° W
0°
20° S
40° S
60° S

Graphic Scales

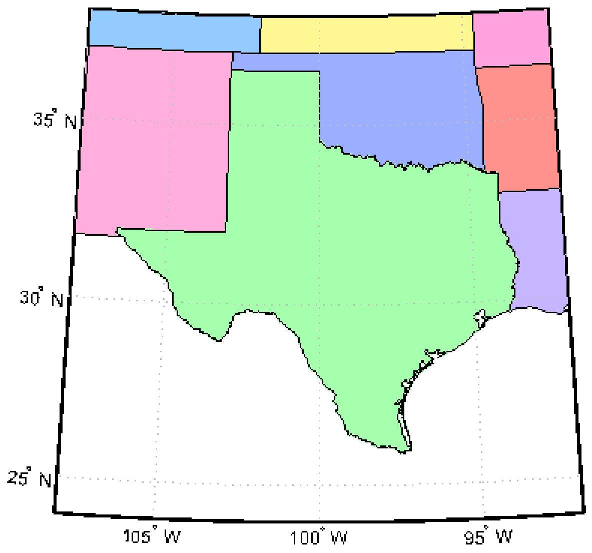
Graphic scale elements are used to provide indications of size even more frequently than insets are. These are ruler-like objects that show distances on the ground at the nominal scale of the projection. You can use the `scaleruler` function to add a graphic scale to the current map. You can check and modify the `scaleruler` settings using `getm` and `setm`. You can also move the graphic scale to a new position by dragging its baseline.

Try this by creating a map, adding a graphic scale with the default settings, and shifting its location. Then add a second scale in nautical miles, and change the tick mark style and direction:

1 Use `usamap` to plot a map of Texas and surrounding states as filled polygons:

```
states = shaperead('usastatehi.shp', 'UseGeoCoords', true);
usamap('Texas')
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], ...
    'FaceColor', polcmap(numel(states))});
geoshow(states,'DisplayType', 'polygon',...
    'SymbolSpec', faceColors)
```

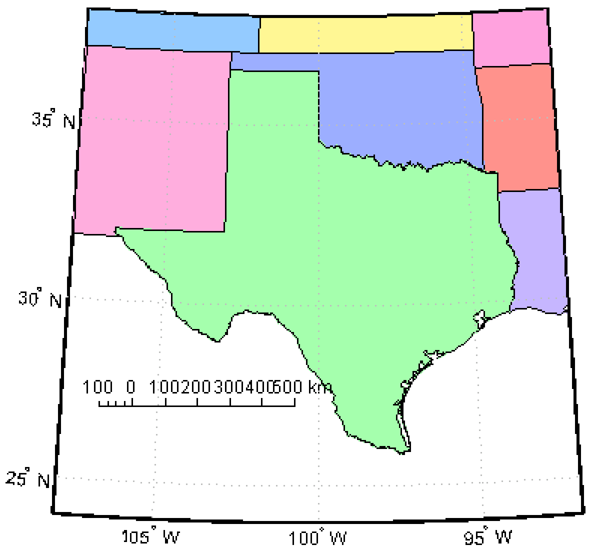
Because `polcmap` randomizes patch colors, your display can look different.



2 Add a default graphic scale and then move it to a new location:

```
scaleruler on
setm(handlem('scaleruler1'),'YLoc',.5)
```

The units of `scaleruler` default to kilometers. Note that `handlem` accepts the keyword `'scaleruler'` or `'scaleruler1'` for the first scaleruler, `'scaleruler2'` for the second one, etc. If there is more than one scaleruler on the current axes, specifying the keyword `'scaleruler'` returns a vector of handles.



3 Obtain a handle to the scaleruler's hggroup using `handlem` and inspect its properties using `getm`:

```
s = handlem('scaleruler');
getm(s)
ans =
            Azimuth: 0
           Children: 'scaleruler1'
              Color: [0 0 0]
          FontAngle: 'normal'
           FontName: 'Helvetica'
           FontSize: 9
          FontUnits: 'points'
         FontWeight: 'normal'
              Label: ''
                Lat: 19.07296767149959
               Long: 24.00830075180499
          LineWidth: 0.50000000000000
          MajorTick: [0 100 200 300 400 500]
     MajorTickLabel: {6x1 cell}
    MajorTickLength: 20
```

```
          MinorTick: [0 25 50 75 100]
     MinorTickLabel: '100'
    MinorTickLength: 12.50000000000000
             Radius: 'earth'
         RulerStyle: 'ruler'
            TickDir: 'up'
           TickMode: 'auto'
              Units: 'km'
               XLoc: 0.15000000000000
               YLoc: 0.50000000000000
               ZLoc: []
```

4 Change the scaleruler's font size to 8 points:

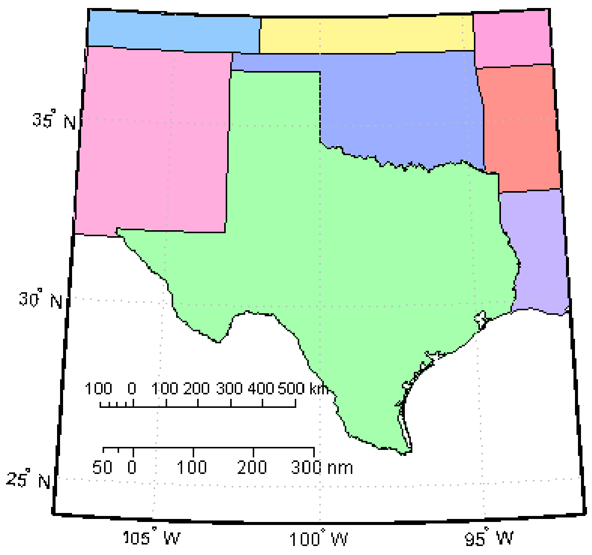
```
setm(s,'fontsize',8)
```

5 Place a second graphic scale, this one in units of nautical miles:

```
scaleruler('units','nm')
```

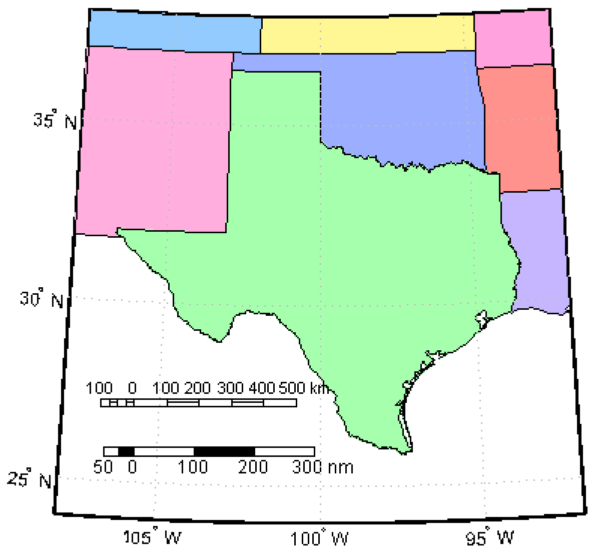
6 Modify its tick properties:

```
setm(handlem('scaleruler2'), 'YLoc', .48,...
    'MajorTick', 0:100:300,...
    'MinorTick', 0:25:50, 'TickDir', 'down',...
    'MajorTickLength', km2nm(25),...
    'MinorTickLength', km2nm(12.5))
```



7 Experiment with the two other ruler styles available:

```
setm(handlem('scaleruler1'), 'RulerStyle', 'lines')
setm(handlem('scaleruler2'), 'RulerStyle', 'patches')
```

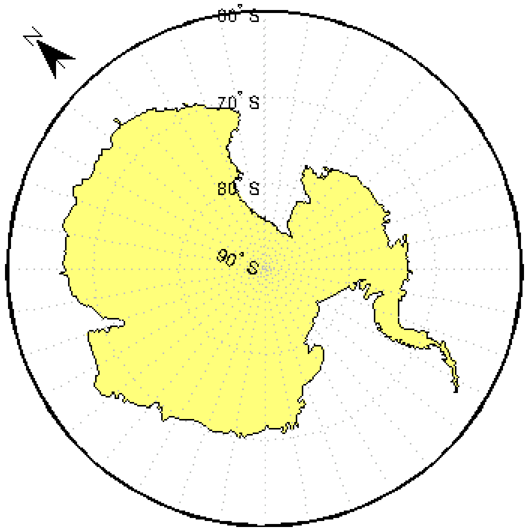

35° N
30° N
25° N
100 0 100 200 300 400 500 km
50 0 100 200 300 nm
105° W
100° W
95° W

North Arrows

The north arrow element provides the orientation of a map by pointing to the geographic North Pole. You can use the `northarrow` function to display a symbol indicating the direction due north on the current map. The north arrow symbol can be repositioned by clicking and dragging its icon. The orientation of the north arrow is computed, and does not need manual adjustment no matter where you move the symbol. **Ctrl**+clicking the icon creates an input dialog box with which you can change the location of the north arrow:

1 To illustrate the use of north arrows, create a map centered at the South Pole and add a north arrow symbol at a specified geographic position:

```
Antarctica = shaperead('landareas', 'UseGeoCoords', true, ...
    'Selector',{@(name) strcmpi(name,{'Antarctica'}), 'Name'});
figure;
worldmap('south pole')
geoshow(Antarctica)
northarrow('latitude', -57, 'longitude', 135);
```



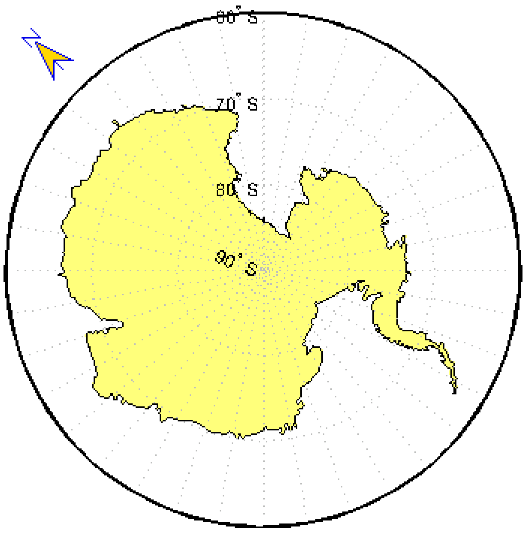
2 Click and drag the north arrow symbol to another corner of the map. Note that it always points to the North Pole.

3 Drag the north arrow back to the top left corner.

4 Right-click or **Ctrl**+click the north arrow. The Inputs for North Arrow dialog opens, which lets you specify the line weight, edge and fill colors, and relative size of the arrow. Set some properties and click **OK**.

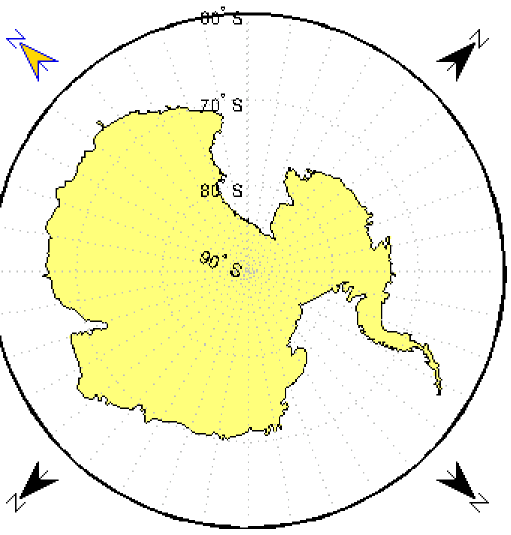
5 Also set some north arrow properties manually, just to get a feel for them:

```
h = handlem('NorthArrow');
set(h, 'FaceColor', [1.000 0.8431 0.0000],...
   'EdgeColor', [0.0100 0.0100 0.9000])
```



6 Make three more north arrows, to show that from the South Pole, every direction is north:

```
northarrow('latitude',-57,'longitude', 45);
northarrow('latitude',-57,'longitude',225);
northarrow('latitude',-57,'longitude',315);
```



Note North arrows are created as objects in the MATLAB axes (and thus have Cartesian coordinates), not as mapping objects. As a result, if you create more than one north arrow, any Mapping Toolbox function that manipulates a north arrow will affect only the last one drawn.

Thematic Maps

In this section...

What Is a Thematic Map?

Most published and online maps fall into four categories:

- Navigation maps, including topographic maps and nautical and aeronautical charts
- Geophysical maps, that show the structure and dynamics of earth, oceans and atmosphere
- Location maps, that depict the locations and names of physical features
- Thematic maps, that portray attribute data about locations and features

Although online maps often combine these categories in new and unexpected ways, published maps and atlases tend to respect them.

Thematic maps tend to be more highly stylized than other types of maps and frequently omit locational information such as place names, physical features, coordinate grids, and map scales. This is because rather than showing physical features on the ground, such as shorelines, roads, settlements, topography, and vegetation, a thematic map displays quantified facts (a "theme"), such as statistics for a region or sets of regions. Examples include the locations of traffic accidents in a city, or election results by state. Thematic maps have a wide vocabulary of cartographic symbols, such as point symbols, dot distributions, "quiver" vectors, isolines, colored zones, raised prisms, and continuous 3-D surfaces. Mapping Toolbox functions can generate most of these types of map symbology.

Choropleth Maps

The most familiar form of thematic map is probably the choropleth map (from the Greek *choros*, for place, and *plethos*, for magnitude). Often used to present data in newspapers, magazines, and reports, choropleth maps fill geographic zones (such as countries or states, but also matrices) with colors and/or patterns to represent nominal, ordinal, or cardinal data values. As there are usually more possible data values than unique symbols or colors capable of differentiating them, choropleth maps usually classify their data into value ranges.

Mapping Toolbox choropleth maps are constructed with patch objects . It assigns a color to each patch face to represent a specified variable, one value per patch. When the variable is scalar (as opposed to nominal) it generally represents a density (such as population per unit area), intensity (such as income per family), or incidence rate (such as fatalities per thousand persons). It can also convey extensive measurements or counts (such as electoral votes per state) if used carefully.

To make a choropleth map you need to input or compute a vector of values, one for each patch in a vector data set. Symbolizing such data values with the toolbox is straightforward. It involves assigning the data values to the `CData` property of a set of patches, and then setting up a colormap with an appropriate color scheme and range. Colormaps usually map N or fewer values (for N patches) to M colors. M can be any number between 2 and N, but typically ranges between 5 and 10.

In the following example, patches representing the 50 states of the U.S. (and the District of Columbia) are displayed and colored according to the surface areas calculated by the `areaint` function. An equal-area projection is appropriate for this and other choropleth maps. This is because data is often computed or normalized over the patches being displayed, and thus area distortion should be minimized, even at the expense of shape distortion.

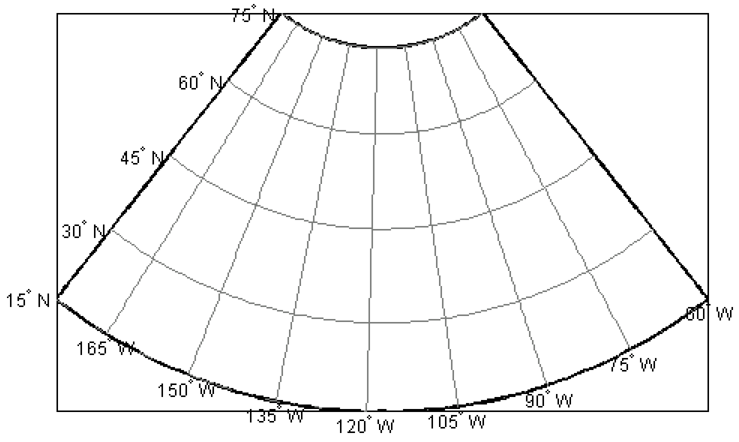
1 Import low-resolution U.S. state boundary polygons:

```
states = shaperead('usastatelo', 'UseGeoCoords', true);
```

This data set includes patch data for individual states, the United States, and its Great Lakes.

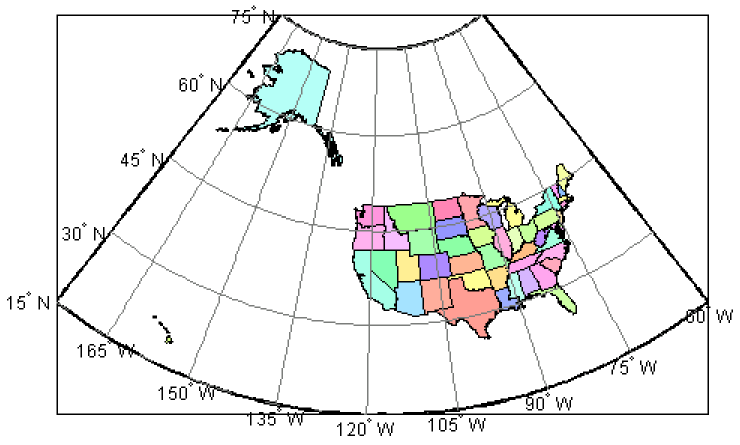
2 Set up map axes with a projection suitable to display all 50 states with equal areas, a graticule, and grid labels:

```
axesm('MapProjection', 'eqaconic', 'MapParallels', [],...
  'MapLatLimit', [15 75], 'MapLonLimit', [-175 -60],...
  'MLineLocation', 15, 'MLabelParallel', 'south',...
  'MeridianLabel', 'on', 'ParallelLabel', 'on',...
  'GLineStyle', '-', 'GColor' , 0.5*[1 1 1],...
  'Grid', 'on', 'Frame', 'on')
```



3 Draw the polygon map in the state structure using face colors randomly selected by `polcmap`:

```
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))});
geoshow(states, 'DisplayType', 'polygon', ...
'SymbolSpec', faceColors)
```



4 Choose an ellipsoid for computing spherical area:

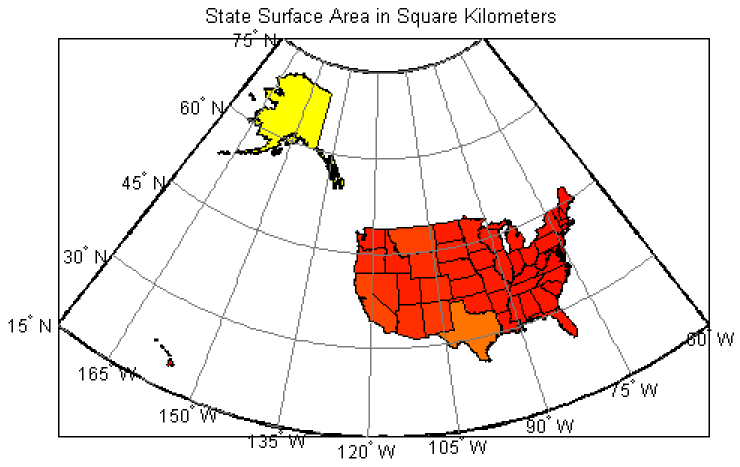
```
wgs84 = almanac('earth', 'geoid', 'kilometers', 'grs80');
```

5 Add a `'SurfaceArea'` field to the states geostruct, and assign surface areas in square kilometers for each U.S. state plus D.C. with a `for` loop:

```
for k = 1:numel(states)
    states(k).SurfaceArea = sum(areaint(states(k).Lat, ...
    states(k).Lon, wgs84));
end
maxarea = max([states.SurfaceArea]);
```

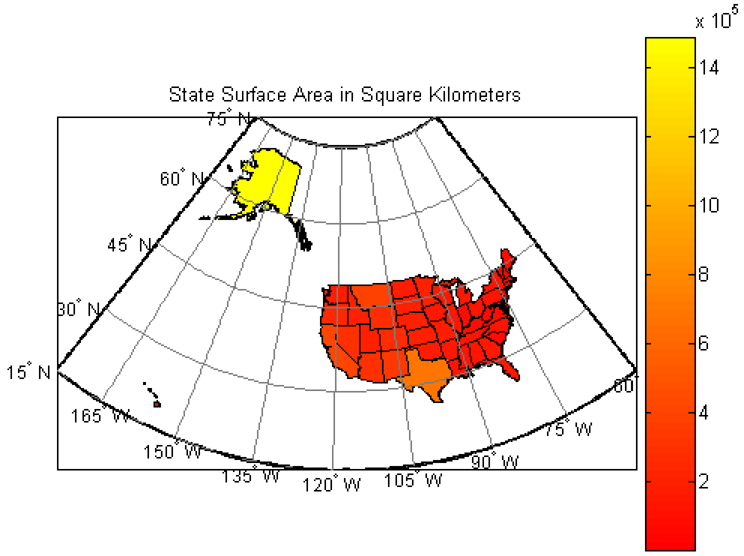
6 Redisplay the states based on the surface area. Use a monotonic colormap from red to yellow.

```
surfaceColors = makesymbolspec('Polygon',...
    {'SurfaceArea', [0 maxarea], ...
    'FaceColor', autumn(numel(states))});
geoshow(states, 'DisplayType', 'polygon', ...
'SymbolSpec', surfaceColors)
title('State Surface Area in Square Kilometers')
```



7 Show a colorbar as a key to the symbology, in its default location. This legend relates patch color to area in square km:

```
caxis([0 maxarea])
colormap('autumn')
colorbar
```



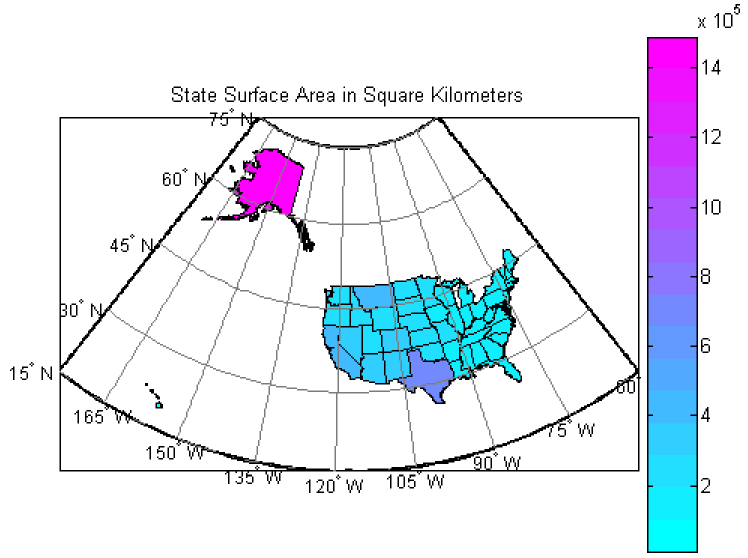
8 The map is mostly red, as the above figure shows. Experiment with other colormaps. Some names of predefined colormaps are `autumn`, `cool`, `copper`, `gray`, `pink`, and `jet`.

Note that while the color scale varies continuously, many states appear to be the same color. This is because of the skewed distribution of state areas. One way to differentiate the symbology is to clamp the lower end (because the smallest patches, such as District of Columbia and Rhode Island, are much smaller than average) and the upper end (because Alaska's area is so much larger than that of any other state).

9 Change the colormap to one that has more hues and a smaller number of steps, and redraw the colorbar to display the new value range:

```
minarea = 10000;
surfaceColors = makesymbolspec('Polygon',...
    {'Default','FaceColor','red'}, ...
    {'SurfaceArea', [ minarea maxarea], 'FaceColor', cool(16)});
geoshow(states,'DisplayType', 'polygon', ...
    'SymbolSpec', surfaceColors)
```

```
caxis([minarea maxarea])
colormap(cool(16))
colorbar
```



Note how you can specify the size of a colormap with the colormap syntax used above. Be aware that, because you clamped the value range, the numeric limits of the colorbar overstate the minimum area and understate the maximum area. However, the map gives much more information overall because more states have distinct symbology, as the resulting map depicts.

Special Thematic Mapping Functions

In addition to choropleth maps, other Mapping Toolbox display and symbology functions include

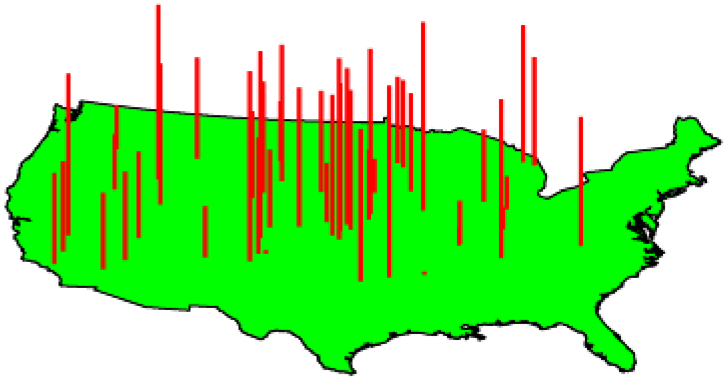
Function	Used For
`cometm`	Traces (animates) vectors slowly from a comet head
`comet3m`	Traces (animates) vectors in 3-D slowly from a comet head

Function	Used For
quiverm	Plots directed vectors in 2-D from specified latitudes and longitudes with lengths also specified as latitudes and longitudes
quiver3m	Plots directed vectors in 3-D from specified latitudes, longitudes, and altitudes with lengths also specified as latitudes and longitudes and altitudes
scatterm	Draws fixed or proportional symbol maps for each point in a vector with specified marker symbol. Similar maps can be generated using geoshow and mapshow using appropriate symbol specifications ("symbolspecs").
stem3m	Projects a 3-D stem plot map on the current map axes

The cometm and quiverm functions operate like their MATLAB counterparts comet and quiver. The stem3m function allows you to display geographic bar graphs. Like the MATLAB scatter function, the scatterm function allows you to display a thematic map with proportionally sized symbols. The tissot function calculates and displays Tissot Indicatrices, which graphically portray the shape distortions of any map projection. For more information on these capabilities, consult the descriptions of these functions in the reference pages.

Stem Maps

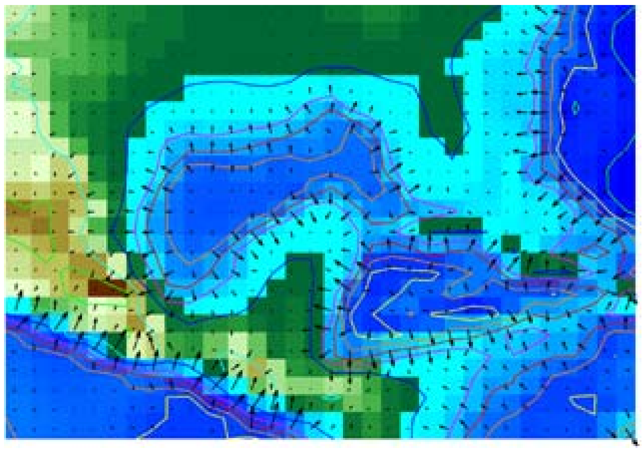
Stem plots are 3-D geographic bar graphs portraying numeric attributes at point locations, usually on vector base maps. Below is an example of a stem plot over a map of the continental United States. The bars could represent anything from selected city populations to the number of units of a product purchased at each location:



Contour Maps

Contour and quiver plots can be useful in analyzing matrix data. In the following example, contour elevation lines have been drawn over a topographical map. The region displayed is the Gulf of Mexico, obtained from the `topo` matrix. Quiver plots have been added to visualize the gradient of the topographical matrix.

Here is the displayed map:

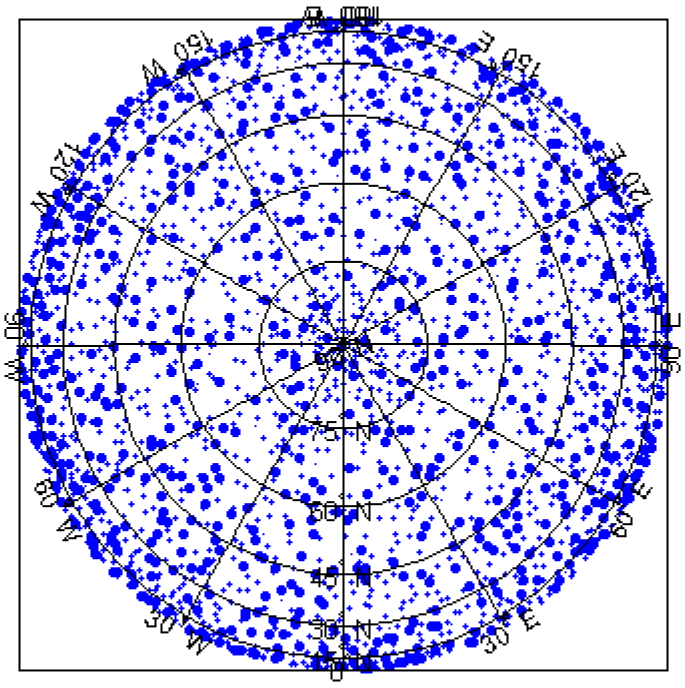


Scatter Maps

The `scatterm` function plots symbols at specified point locations, like the MATLAB `scatter` function. If the symbols are small and inconspicuous and do not vary in size, the result is a *dot-distribution map*. If the symbols vary in size and/or shape according to a vector of attribute values, the result is a *proportional symbol map*.

Below is an example of using `scatterm` to create a star chart of the northern sky. The stars are represented by filled circles whose size is negatively proportional to visual magnitude, because the brighter a star is, the smaller its magnitude. (The very brightest stars actually have negative magnitude values.) To execute the following commands, select them all by dragging over the list in the Help browser, then right-click and choose `Evaluate Selection`.

```
% View the sky orthographically
axesm('MapProjection','ortho','MapLatLimit',[0 90])
gridm on
setm(gca,'LabelFormat','compass','LabelRotation','on')
setm(gca,'MLabelParallel',0,'PLabelMeridian',0)
setm(gca,'MeridianLabel','on','ParallelLabel','on')
setm(gca,'GlineStyle','-')

% For each star, use a symbol with area negatively
% proportional to the star's visual magnitude.
load stars
symbolArea = (20 - 2*vmag);

% Plot each star as a blue-filled circle.
h = scatterm(lat,long,symbolArea,'b','filled');
```


180
150 W
150 E
120 W
120 E
90 W
90 E
60 W
60 E
30 W
30 E
0
75 N
60 N
45 N
30 N
15 N

Using Cartesian MATLAB Display Functions

In this section...

Adding Graphic Objects to Map Axes

If you cannot find a Mapping Toolbox display function that makes the kind of display that you need, you might be able to use MATLAB functions. When placing graphic objects on a map axes, you can use the MATLAB function to add the graphic objects to the display, using latitude and longitude as *x* and *y*, and then project the data afterwards.

Note Before applying MATLAB functions to geodata, you should take into consideration that performing Cartesian geometric operations on geographic coordinates can yield inaccurate results when the data covers large regions of a planet or lies near one of its poles.

Example 1: Triangulating Data Points

There is no Mapping Toolbox function that displays a triangulated surface from random data points, a structure generally known as a *triangulated irregular network* (TIN). However, MATLAB does have a function to create *Delaunay triangles*, a method that is often used to form TINs from projected point coordinate data. Explore triangulating some point data and bringing the result into a Mapping Toolbox map axes:

1 Use the seamount data provided as MATLAB demo data:

```
load seamount
```

2 Determine the bounds of the coordinates and add a degree of white space:

```
latlim = [min(y)-.5 max(y)+.5];
lonlim = [min(x)-.5 max(x)+.5];
```

3 Create map axes to contain the seamount region (`worldmap` selects a projection for you):

```
worldmap(latlim,lonlim)
```

4 Create a Delaunay triangulation of *x* and *y* (longitude and latitude):

```
tri = delaunay(y,x);
```

5 Generate a 3-D surface that combines the triangulation and *z*-values:

```
h = trisurf(tri,y,x,z);
```

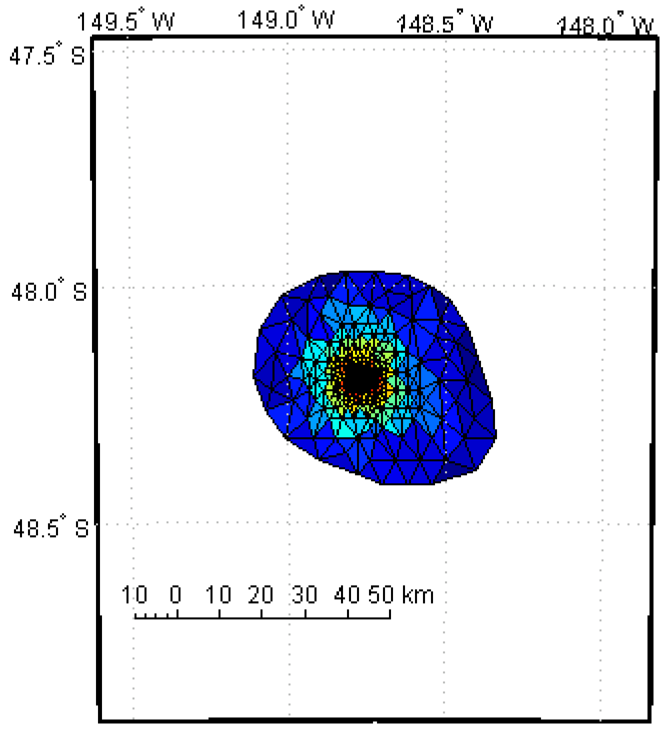
6 Map the surface onto the axes by projecting to the *x*-*y* plane (`project` is a Mapping Toolbox function especially for this purpose):

```
project(h,'yx')
```

Note that even though the triangulated surface appears on the map, it does not have a geostruct (see "Mapping Toolbox Geographic Data Structures" on page 2-16).

7 Add a default graphic scale to the display:

```
scaleruler on
```



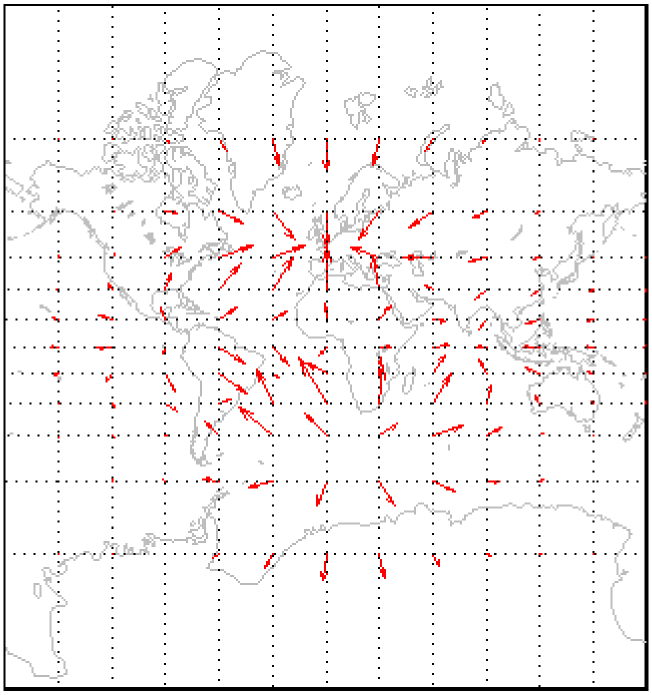
If, as in this example, the displayed objects are already in the right place and do not need to be projected, you can trim them to the map frame and convert them to mapped objects using `trimcart` and `makemapped`. They can then be manipulated as if they had been created with map display functions.

Example 2: Constructing Quiver Maps

As was briefly described for text objects in “Projected and Unprojected Graphic Objects” on page 4-37, you can also combine Mapping Toolbox and MATLAB functions to mix spherical and Cartesian coordinates. An example would be a quiver plot (sometimes known as a *vector field*) in which the locations of the vectors are geographic, but the lengths, being specified by attributes, are not. In that case, you can use Mapping Toolbox map projection calculations and MATLAB graphics functions. Cylindrical projections are the simplest to use because north is up, south is down, and east and west are on an orthogonal axis.

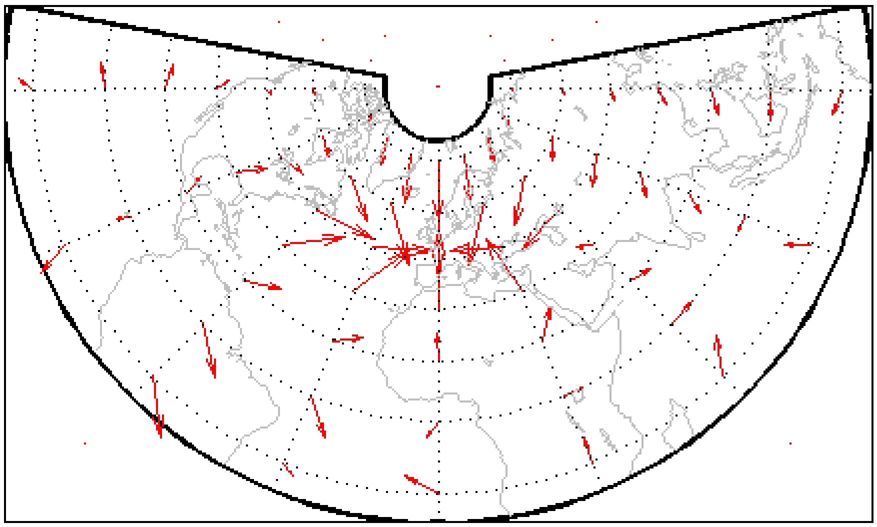
In this example, you will impose a quiver map of the slope of a surface on a world map. The surface is a Gaussian field generated by the MATLAB `peaks` function.

```
figure; axesm mercator; framem; gridm
load coast
plotm(lat,long,'color',[.75 .75 .75])
[u,v] = gradient(peaks(13)/10);
[mlat,mlon] = meshgrat(-90:15:90,-180:30:180);
[x,y] = mfwdtran(mlat,mlon);
h = quiver(x,y,u,v,.2,'r');
trimcart(h)
tightmap
```



An extra step might be required for noncylindrical projections. In these projections, compass directions vary with location. To make the directions agree with the map grid, vectors should be rotated to bring them into alignment. This can be done with the vector transformation function `vfwdtran`. Consider the same data displayed on a conic projection.

```
load coast; figure
axesm('lambert','MapLatLimit',[-20 80])
framem; gridm
plotm(lat,long,'color',[.75 .75 .75])
[u,v] = gradient(peaks(13)/10);
[mlat,mlon] = meshgrat(-90:15:90,-180:30:180);
[x,y] = mfwdtran(mlat,mlon);
thproj = degtorad(vfwdtran(mlat,mlon,90*ones(size(mlat))));
[th,r] = cart2pol(u,v);
[uproj,vproj] = pol2cart(th+thproj,r);
h = quiver(x,y,uproj,vproj,0,'r') ;
trimcart(h)
tightmap
```



Conformal projections, such as this Lambert conformal conic, are often the best choice for quiver displays. They preserve angles, ensuring that the difference between north and east will always be 90 degrees in projected coordinates.

Using Colormaps and Colorbars

In this section...

Colormap for Terrain Data

Colors and colorscales (ordered progressions of colors) are invaluable for representing geographic variables on maps, particularly when you create terrain and thematic maps. The following sections describe techniques and provide examples for applying colormaps and colorbars to maps.

In previous examples, the function `demcmap` was used to color several digital elevation model (DEM) topographic displays. This function creates colormaps appropriate to rendering DEMs, although it is certainly not limited to DEMs.

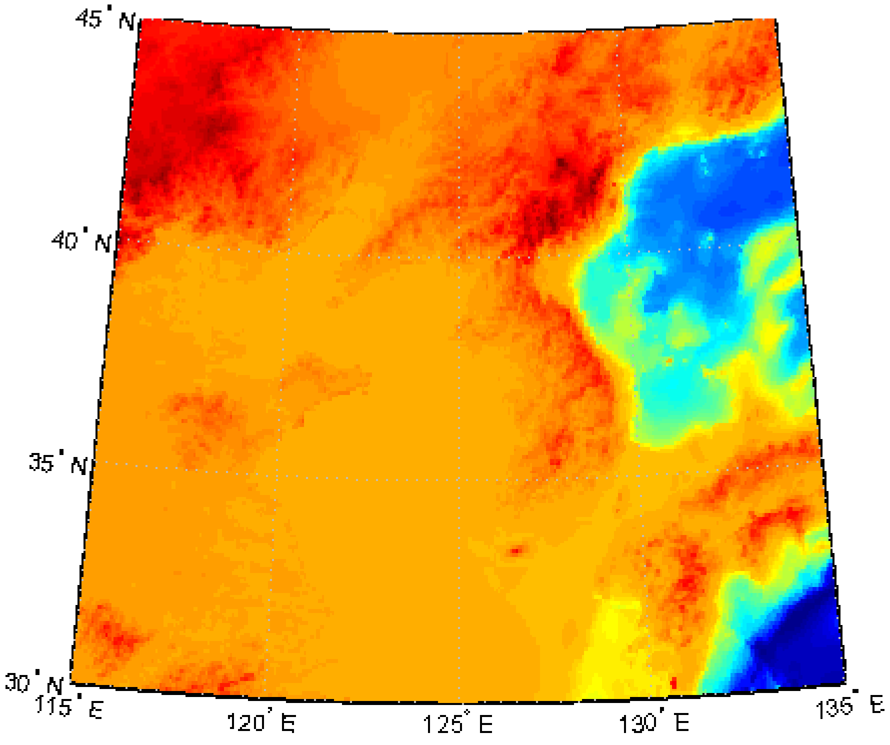
These colormaps, by default, have atlas-like colors varying with elevation or depth that properly preserve the land-sea interface. In cartography, such color schemes are called *hypsometric tints*.

1. Here you explore `demcmap` using the topographic data for the Korean peninsula provided in the `korea` data set. To set up an appropriate map projection, pass the `korea` data grid and referencing vector to `worldmap`:

   ```
   load korea
   figure
   worldmap(map,refvec)
   ```

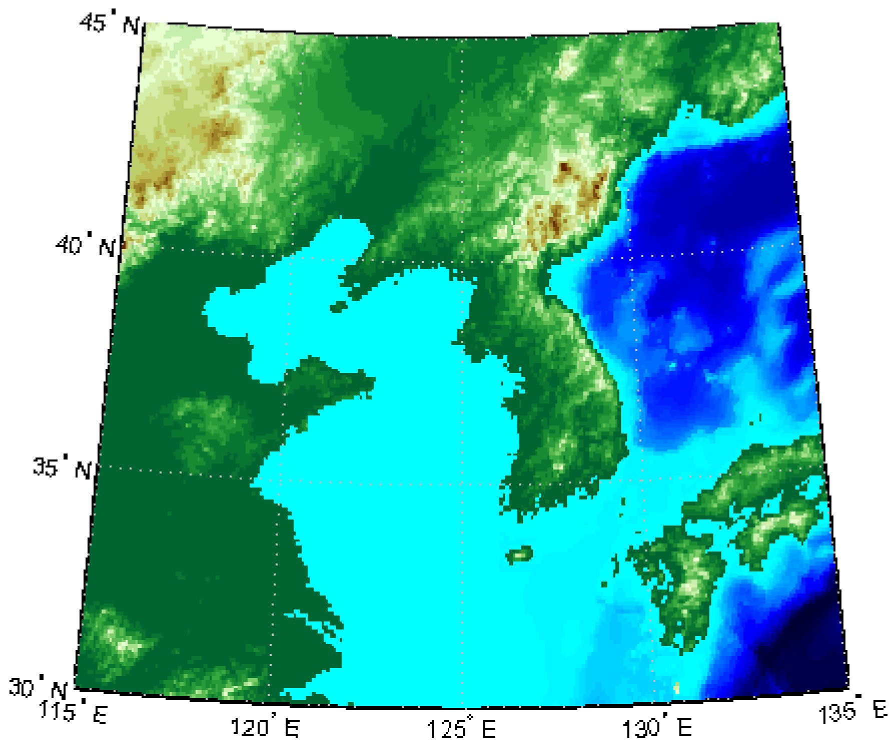
2. Display the data grid with `geoshow`:

   ```
   geoshow(map, refvec, 'DisplayType', 'mesh')
   ```



3 The Korea DEM is displayed using the default colormap, which is inappropriate and causes the surface to be unrecognizable. Now apply the default DEM colormap:

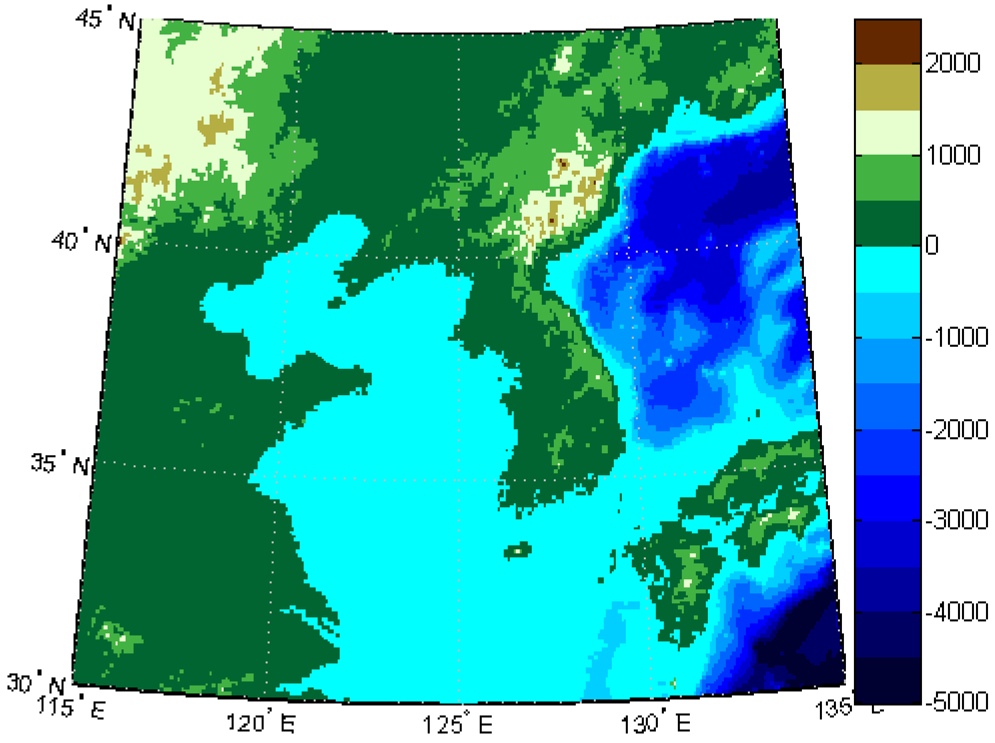
```
demcmap(map)
```



4 You can also make `demcmap` assign all altitudes within a particular range to the same color. This results in a quasi-contour map with breaks at a constant interval. Now color this map using the same color scheme coarsened to display 500 meter bands:

```
demcmap('inc',map,500)
colorbar
```

Note that the first argument to `demcmap`, `'inc'`, indicates that the third argument should be interpreted as a value range. If you prefer, you can specify the desired number of colors with the third argument by setting the first argument to `'size'`.



Contour Colormaps

You can create colormaps that make surfaces look like contour maps for other types of data besides terrain. The `contourcmap` function creates a colormap that has color changes at a fixed value increment. Its required arguments are the increment value and the name of a colormap function. Optionally, you can also use `contourcmap` to add and label a colorbar similarly to the MATLAB `colorbar` function:

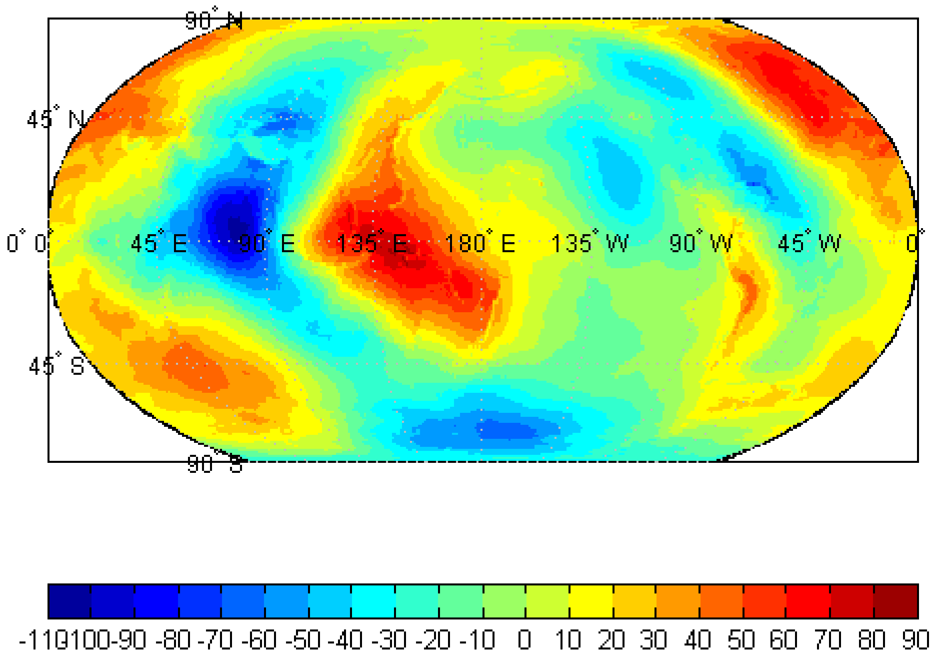
1 Explore `contourcmap` by loading the world `geoid` data set and rendering it with a default colormap:

```
load geoid
figure;
worldmap(geoid,geoidrefvec)
```

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

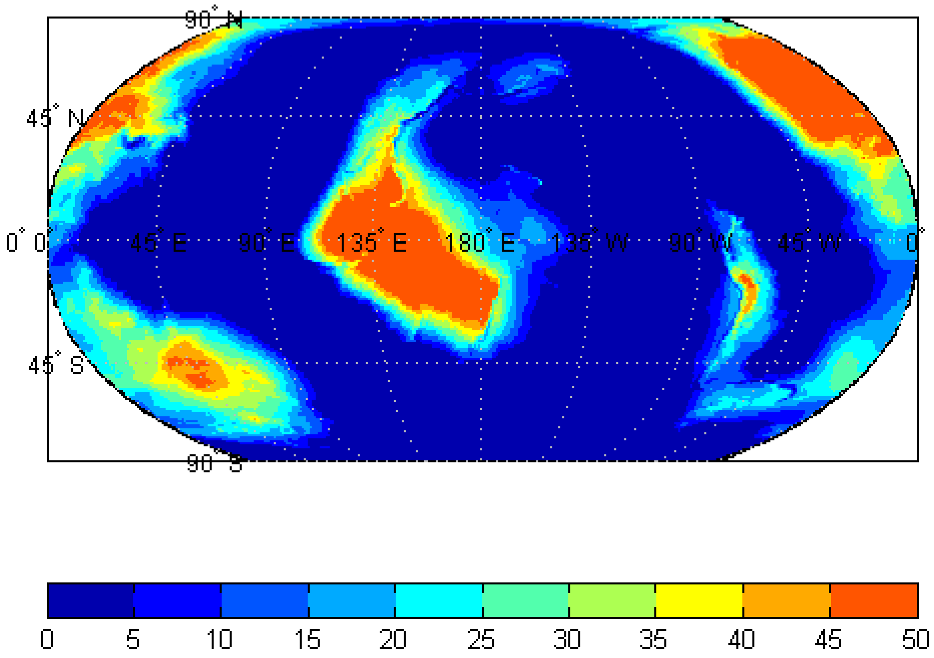
2 Use `contourcmap` to specify a contour interval of 10 (meters), and to place a colorbar beneath the map:

```
contourcmap(10,'jet','colorbar','on','location','horizontal')
```



3 If you want to render a restricted value range, you can enter a vector of evenly spaced values for the first argument. Here you specify a 5-meter interval and truncate symbology at 0 meters on the low end and 50 meters at the high end:

```
contourcmap([0:5:50],...
'jet','colorbar','on','location','horizontal')
```



Should you need to write a custom colormap function, for example, one that has irregular contour intervals, you can easily do so, but it should have the N-by-3 structure of MATLAB colormaps.

Colormaps for Political Maps

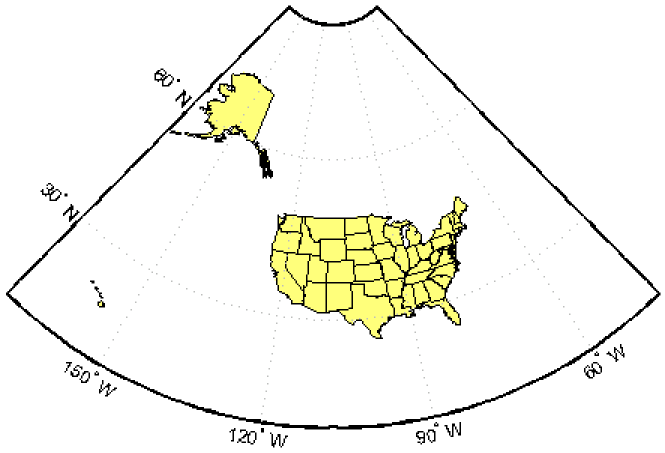
Political maps typically use muted, contrasting colors that make it easy to distinguish one country from its neighbors. You can create colormaps of this kind using the `polcmap` function. The `polcmap` function creates a colormap with randomly selected colors of all hues. Since the colors are random, if you don't like the result, execute `polcmap` again to generate a different colormap:

1 To explore political colormaps, display the `usastatelo` data set as patches, setting up the map with `worldmap` and plotting it with `geoshow`:

```
figure
worldmap na
```

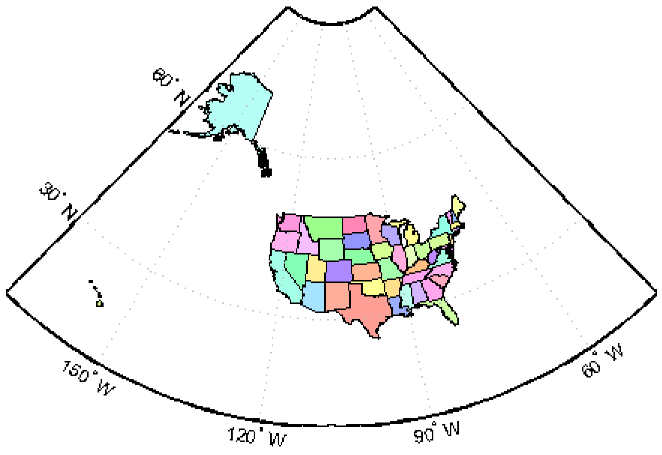
```
states = shaperead('usastatelo', 'UseGeoCoords', true);
geoshow(states)
```

Note that the default face color is black, which is not very interesting.



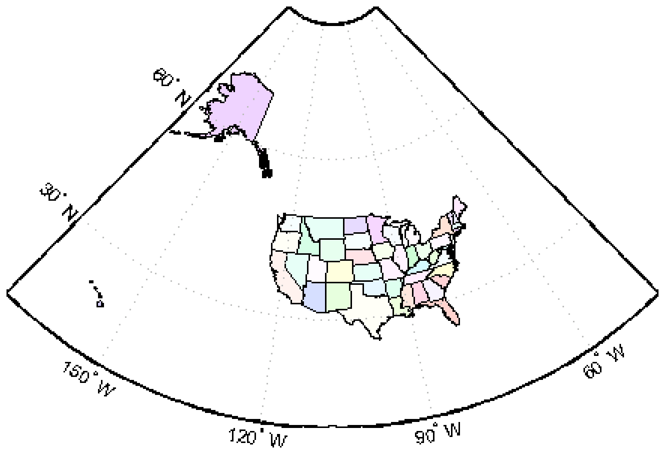
2 Use `polcmap` to populate color definitions to a symbolspec to randomly recolor the patches and expand the map to fill the frame:

```
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor',...
    polcmap(numel(states))});
geoshow(states,'SymbolSpec',faceColors)
```



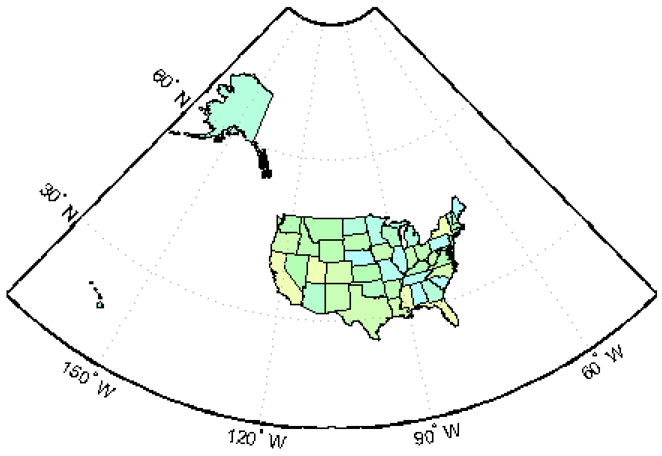
3 The `polcmap` function can also control the number and saturation of colors. Reissue the command specifying 256 colors and a maximum saturation of 0.2. To ensure that the colormap is always the same, reset the seed on the MATLAB random number function using the `'state'` argument with a fixed value of your choice:

```
figure
worldmap na
rand('state',0)
faceColors = makesymbolspec('Polygon',...
   {'INDEX', [1 numel(states)], 'FaceColor', polcmap(256,.2)});
geoshow(states, 'SymbolSpec', faceColors)
```



4 For maximum control over the colors, specify the ranges of hues, saturations, and values. Use the same set of random color indices as before.

```
figure
worldmap na
rand('state',0)
faceColors = makesymbolspec('Polygon', ...
   {'INDEX', [1 numel(states)], ...
    'FaceColor', polcmap(256,[.2 .5],[.3 .3],[1 1]) });
geoshow(states, 'SymbolSpec', faceColors)
```

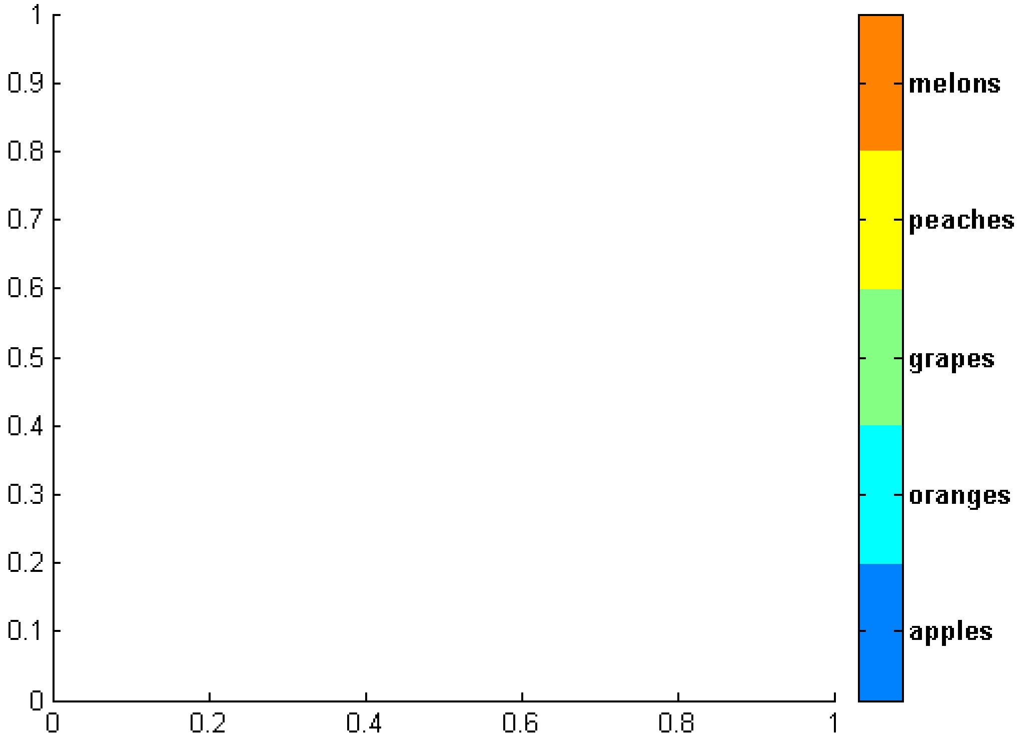


Note The famous Four Color theorem states that any political map can be colored to completely differentiate neighboring patches using only four colors. Experiment to find how many colors it takes to color neighbors differently with `polcmap`.

Labeling Colorbars

Political maps are an example of nominal data display. Many nominal data sets have names associated with a set of integer values, or consist of codes that identify values that are ordinal in nature (such as low, medium, and high). The function `lcolorbar` creates a colorbar having a text label aligned with each color. Nominal colorbars are customarily used only with small colormaps (where 10 categories or fewer are being displayed). `lcolorbar` has options for orienting the colorbar and aligning text in addition to the graphic properties it shares with axes objects.

```
figure; colormap(jet(5))
labels = {'apples','oranges','grapes','peaches','melons'};
lcolorbar(labels,'fontweight','bold');
```



Editing Colorbars

Maps of nominal data often require colormaps with special colors for each index value. To avoid building such colormaps by hand, use the MATLAB GUI for colormaps, `colormapeditor`, described in the MATLAB Function Reference pages. Also see the MATLAB `colormap` function documentation.

Printing Maps to Scale

Maps are often printed at a size that makes objects on paper a particular fraction of their real size. The linear ratio of the mapped to real object sizes is called *map scale*, and it is usually notated with a colon as "1:1,000,000" or "1:24,000." Another way of specifying scale is to call out the printed and real lengths, for example "1 inch = 1 mile."

You can specify the printed scale using the `paperscale` function. It modifies the size of the printed area on the page to match the scale. If the resulting dimensions are larger than your paper, you can reduce the amount of empty space around the map using `tightmap`, `zoom`, or `panzoom`, and by changing the axes position to fill the figure. This also reduces the amount of memory needed to print with the `zbuffer` (raster image) renderer. Be sure to set the paper scale last. For example,

```
set(gca,'Units','Normalized','Position',[0 0 1 1])
tightmap
paperscale(1,'in', 5,'miles')
```

The `paperscale` function also can take a scale denominator as its first and only argument. If you want the map to be printed at 1:20,000,000, type

```
paperscale(2e7)
```

To check the size and extent of text and the relative position of axes, use `previewmap`, which resizes the figure to the printed size.

```
previewmap
```

For more information on printing, see the "Printing and Exporting" section of the MATLAB Graphics documentation.

Manipulating Geospatial Data

For some purposes, geospatial data is fine to use as is. Sooner or later, though, you need to extract, combine, massage, and transform geodata. This chapter discusses some Mapping Toolbox tools and techniques provided for such purposes.

Manipulating Vector Geodata

In this section...

Repackaging Vector Objects

It can be difficult to identify line or patch segments once they have been combined into large `NaN`-clipped vectors. You can separate these polygon or line vectors into their component segments using `polysplit`, which takes column vectors as inputs.

Extracting and Joining Polygons or Line Segments

1 Enter two `NaN`-delimited arrays in the form of column vectors:

```
lat = [45.6 -23.47 78 NaN 43.9 -67.14 90 -89]';
long = [13 -97.45 165 NaN 0 -114.2 -18 0]';
```

2 Use `polysplit` to create two cell arrays, `latc` and `lonc`:

```
[latc,lonc] = polysplit(lat,long)

latc =
    [3x1 double]    [4x1 double]
lonc =
    [3x1 double]    [4x1 double]
```

3 Inspect the contents of the cell arrays:

```
[latc{1} lonc{1}]
[latc{2} lonc{2}]

ans =
                      45.6                      13
                    -23.47                  -97.45
                        78                     165

ans =
                      43.9                       0
                    -67.14                  -114.2
                        90                     -18
                       -89                       0
```

Note that each cell array element contains a segment of the original line.

4 To reverse the process, use `polyjoin`:

```
[lat2,lon2] = polyjoin(latc,lonc);
```

5 The joined segments are identical with the initial `lat` and `lon` arrays:

```
[lat long] == [lat2 lon2]

ans =
     1     1
     1     1
     1     1
     0     0
     1     1
     1     1
     1     1
     1     1
```

The logical comparison is false for the `NaN` delimiters, by definition.

6 You can test for global equality, including NaNs, as follows:

```
isequalwithequalnans(lat,lat2) & isequalwithequalnans(long,lon2)

ans =
      1
```

See the polysplit and polyjoin reference pages for further information.

Matching Line Segments

A common operation on sets of line segments is the concatenation of segments that have matching endpoints. The polymerge command compares endpoints of segments within latitude and longitude vectors to identify endpoints that match exactly or lie within a specified distance. The matching segments are then concatenated, and the process continues until no more coincidental endpoints can be found. The two required arguments are a latitude (*x*) vector and a longitude (*y*) vector. The following exercise shows this process at work.

Linking Line Segments into Polygons

1 Construct column vectors representing coordinate values:

```
lat = [3 2 NaN 1 2 NaN 5 6 NaN 3 4]';
lon = [13 12 NaN 11 12 NaN 15 16 NaN 13 14]';
```

2 Concatenate the segments that match exactly:

```
[latm,lonm] = polymerge(lat,lon);
[latm lonm]

ans =

      1     11
      2     12
      3     13
      4     14
    NaN    NaN
      5     15
      6     16
```

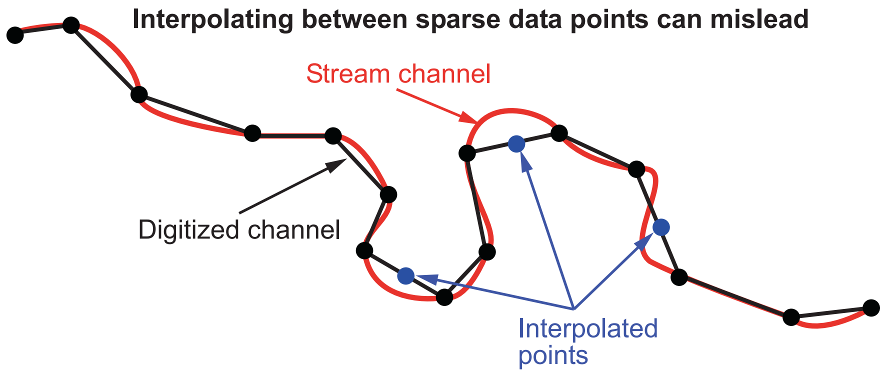
```
     NaN    NaN
```

The original four segments are merged into two segments.

The `polymerge` function takes an optional third argument, a (circular) distance tolerance that permits inexact matching. A fourth argument enables you to specify whether the function outputs vectors or cell arrays. See the `polymerge` reference page for further information.

Geographic Interpolation of Vectors

When using vector data, remember that, like raster data, coordinates are sampled measurements. This involves unavoidable assumptions concerning what the geographic reality is between specified data points. The normal assumption when plotting vector data requires that points be connected with straight line segments, which essentially indicates a lack of knowledge about conditions between the measured points. For lines that are by nature continuous, such as most rivers and coastlines, such piecewise linear interpolation can be false and misleading, as the following figure depicts.



Interpolating Sparse Vector Data

Despite the possibility of misinterpretation, circumstances do exist in which geographic data interpolation is useful or even necessary. To do this, use the `interpm` function to interpolate between known data points. One value of linearly interpolating points is to fill in lines of constant latitude or longitude (e.g., administrative boundaries) that can curve when projected.

Interpolating Vectors to Achieve a Minimum Point Density

This example interpolates values in a set of latitude and longitude points to have no more than one degree of separation in either direction.

1 Define two fictitious latitude and longitude data vectors:

```
 lats = [1 2 4 5]; longs = [1 3 4 5];
```

2 Specify a densification parameter of 1 (the default unit is degrees):

```
maxdiff = 1;
```

3 Call `interpm` to fill in any gaps greater than 1° in either direction:

```
[newlats,newlongs] = interpm(lats,longs,maxdiff)

newlats =
 1.0000
 1.5000
 2.0000
 3.0000
 4.0000
 5.0000
newlongs =
 1.0000
 2.0000
 3.0000
 3.5000
 4.0000
 5.0000
```

In `lats`, a gap of 2° exists between the values 2 and 4. A linearly interpolated point, `(3,3.5)` was therefore inserted in `newlats` and `newlongs`. Similarly, in `longs`, a gap of 2° exists between the 1 and the 3. The point `(1.5, 2)` was therefore interpolated and placed into `newlats` and `newlongs`. Now, the separation of adjacent points is no greater than `maxdiff` in either `newlats` or `newlongs`.

See the `interpm` reference page for further information.

Interpolating Coordinates at Specific Locations

`interpm` returns both the original data and new linearly interpolated points. Sometimes, however, you might want only the interpolated values. The functions `intrplat` and `intrplon` work similarly to the MATLAB `interp1` function, and give you control over the method used for interpolation. Note that they only interpolate and return one value at a time.

Use `intrplat` to interpolate a latitude for a given longitude. Given a monotonic set of longitudes and their matching latitude points, you can interpolate a new latitude for a longitude you specify, interpolating along linear, spline, cubic, rhumb line, or great circle paths. The longitudes must increase or decrease monotonically. If this is not the case, you might be able to use the `intrplon` companion function if the latitude values are monotonic.

Interpolate a latitude corresponding to a longitude of 7.3º in the following data in a linear, great circle, and rhumb line sense:

1 Define some fictitious latitudes and longitudes:

```
longs = [1 3 4 9 13]; lats = [57 68 60 65 56];
```

2 Specify the longitude for which to compute a latitude:

```
newlong = 7.3;
```

3 Generate a new latitude with linear interpolation:

```
newlat = intrplat(longs,lats,newlong,'linear')

newlat =
 63.3000
```

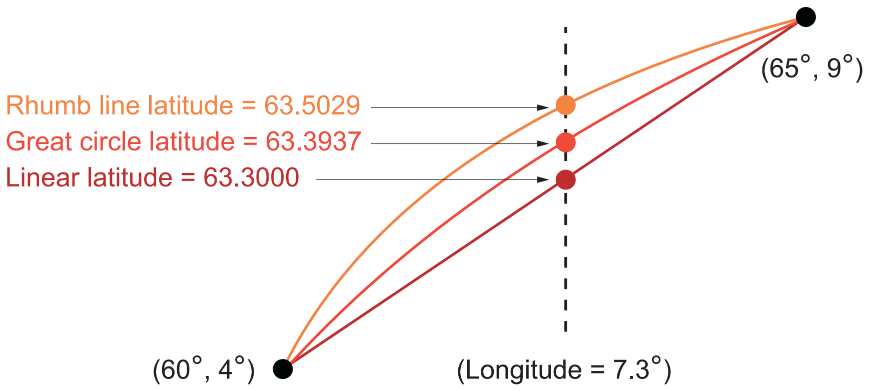
4 Generate the latitude using great circle interpolation:

```
newlat = intrplat(longs,lats,newlong,'gc')

newlat =
  63.5029
```

5 Generate it again, specifying interpolation along a rhumb line:

```
newlat = intrplat(longs,lats,newlong,'rh')

newlat =
 63.3937
```

The following diagram illustrates these three types of interpolation. The `intrplat` function also can perform spline and cubic spline interpolations.


Rhumb line latitude = 63.5029
Great circle latitude = 63.3937
Linear latitude = 63.3000
(65°, 9°)
(60°, 4°)
(Longitude = 7.3°)

As mentioned above, the `intrplon` function provides the capability to interpolate new longitudes from a given set of longitudes and monotonic latitudes.

See the `intrplat` and `intrplon` reference pages for further information.

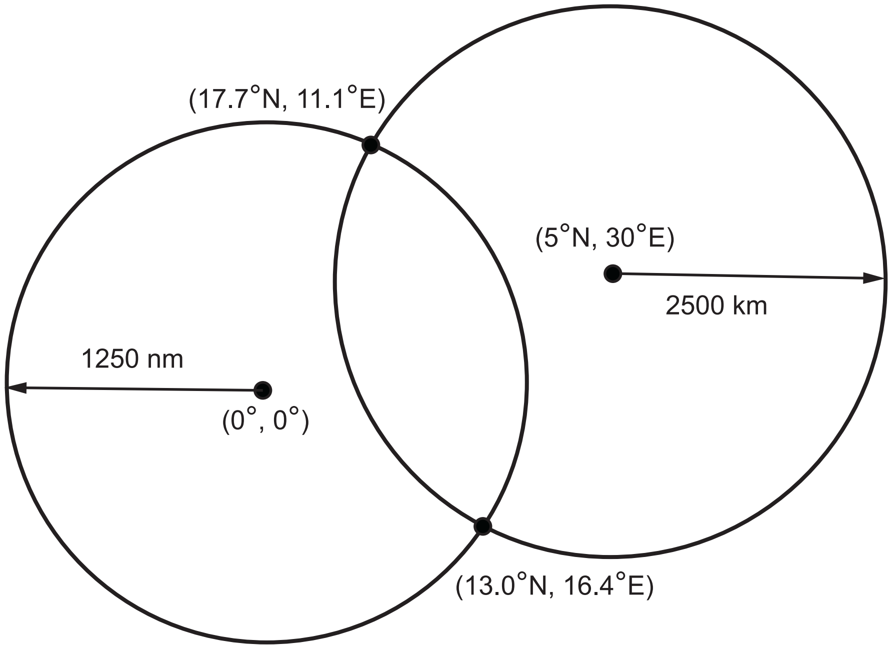
Vector Intersections

A set of Mapping Toolbox functions perform intersection calculations on vector data computed by the toolbox, which include great and small circles as well as rhumb line tracks. The functions also determine intersections of arbitrary vector data.

Compute the intersection of a small circle centered at (0º,0º) with a radius of 1250 nautical miles and a small circle centered at (5ºN,30ºE) with a radius of 2500 kilometers:

```
[lat,long] = scxsc(0,0,nm2deg(1250),5,30,km2deg(2500))
```

```
lat =
 17.7487 -12.9839
long =
 11.0624 16.4170
```



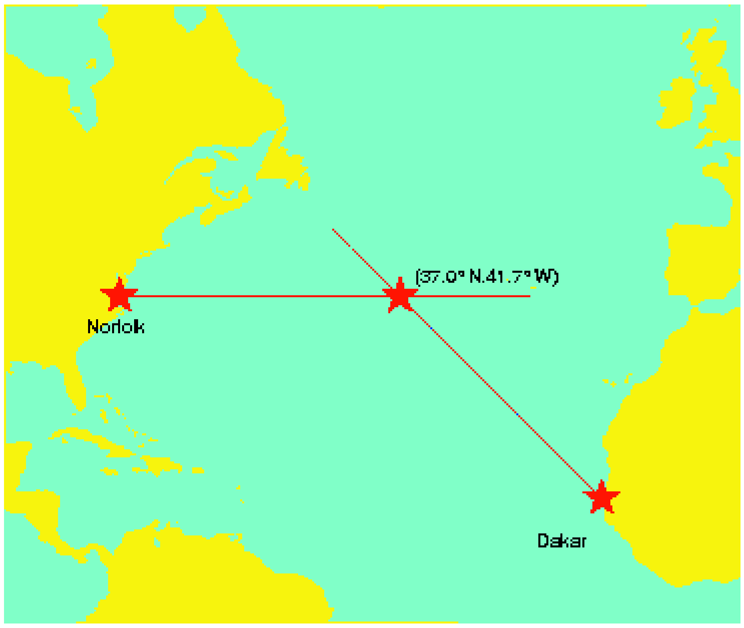
In general, small circles intersect twice or never. For the case of exact tangency, `scxsc` returns two identical intersection points. Other similar commands include `rhxrh` for intersecting rhumb lines, `gcxgc` for intersecting great circles, and `gcxsc` for intersecting a great circle with a small circle.

Imagine a ship setting sail from Norfolk, Virginia (37ºN,76ºW), maintaining a steady due-east course (90º), and another ship setting sail from Dakar, Senegal (15ºN,17ºW), with a steady northwest course (315º). Where would the tracks of the two vessels cross?

```
[lat,long] = rhxrh(37,-76,90,15,-17,315)

lat =
 37
```

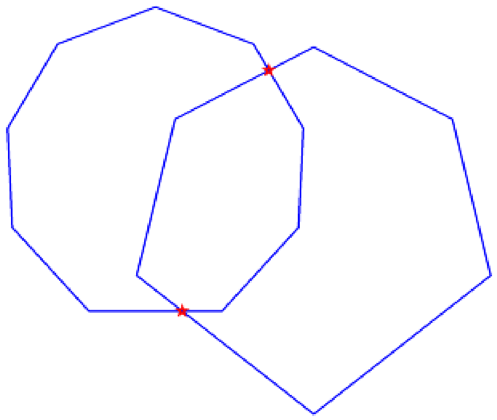
```
long =
 -41.7028
```

The intersection of the tracks is at (37ºN,41.7ºW), which is roughly 600 nautical miles west of the Azores in the Atlantic Ocean.



You can also compute the intersection points of arbitrary vectors of latitude and longitude. The `polyxpoly` command finds the segments that intersect and interpolates to find the intersection points. The interpolation is done linearly, as if the points were in a Cartesian *x*-*y* coordinate system. The `polyxpoly` command can also identify the line segment numbers associated with the intersections:

```
[xint,yint] = polyxpoly(x1,y1,x2,y2);
```



If the spacing between points is large, there can be some difference between the intersection points computed by `polyxpoly` and the intersections shown on a map display. This is a result of the difference between straight lines in the unprojected and projected coordinates. Similarly, there can be differences between the `polyxpoly` result and intersections assuming great circles or rhumb lines between points.

Polygon Area

Use the function `areaint` to calculate geographic areas for vector data in polygon format. The function performs a numerical integration using Green's Theorem for the area on a surface enclosed by a polygon. Because this is a discrete integration on discrete data, the results are not exact. Nevertheless, the method provides the best means of calculating the areas of arbitrarily shaped regions. Better measures result from better data.

The Mapping Toolbox function `areaint` (for area by integration), like the other area functions, `areaquad` and `areamat`, returns areas as a fraction of the entire planet's surface, unless a radius is provided. Here you calculate the area of the continental United States using the `conus` MAT-file. Three areas are returned, because the data contains three polygons: Long Island, Martha's Vineyard, and the rest of the continental U.S.:

```
load conus
earthradius = almanac('earth','radius');
```

```
area = areaint(uslat,uslon,earthradius)

area =
1.0e+06 *
  7.9256
  0.0035
  0.0004
```

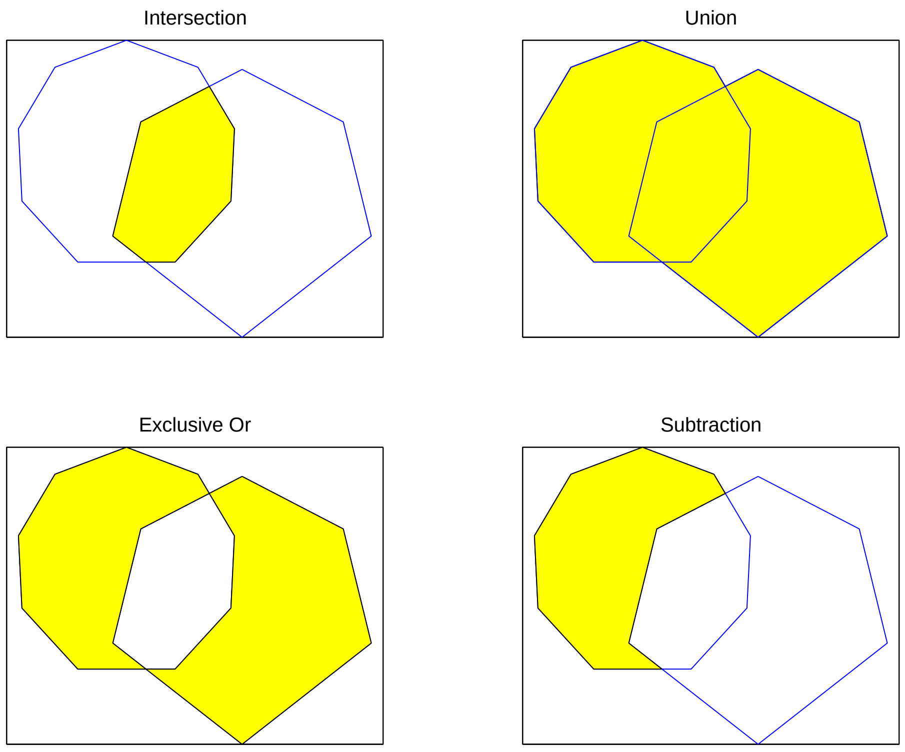
Because the default Earth radius is in kilometers, the area is in square kilometers. From the same variables, the areas of the Great Lakes can be calculated, this time in square miles:

```
load conus
earthradius = almanac('earth','radius','miles');
area = areaint(gtlakelat,gtlakelon,earthradius)

area =
  1.0e+004 *
    8.0119
    1.0381
    0.7634
```

Again, three areas are returned, the largest for the polygon representing Superior, Michigan, and Huron together, the other two for Erie and Ontario.

Overlaying Polygons with Set Logic

Polygon set operations are used to answer a variety of questions about logical relationships of vector data polygon objects. Standard set operations include intersection, union, subtraction, and an exclusive OR operation. The `polybool` function performs these operations on two sets of vectors, which can represent *x*-*y* or latitude-longitude coordinate pairs. In computing points where boundaries intersect, interpolations are carried out on the coordinates as if they were planar. Here is an example that shows all the available operations.



The result is returned as `NaN`-clipped vectors by default. In cases where it is important to distinguish outer contours of polygons from interior holes, `polybool` can also accept inputs and return outputs as cell arrays. In the cell array format, a cell array entry starts with the list of points making up the outer contour. Subsequent `NaN`-clipped faces within the cell entry are interpreted as interior holes.

Overlaying Polygons with the polybool Function

The following exercise demonstrates how you can use `polybool`:

1 Construct a twelve-sided polygon:

```
theta = -(0:pi/6:2*pi)';
lat1 = sin(theta);
lon1 = cos(theta);
```

2 Construct a triangle that overlaps it:

```
lat2 = [0 1 -1 0]';
lon2 = [0 2  2 0]';
```

3 Plot the two shapes together with blue and red lines:

```
axesm miller
plotm(lat1,lon1,'b')
plotm(lat2,lon2,'r')
```

4 Compute the intersection polygon and plot it as a green patch:

```
[loni,lati] = polybool('intersection',lon1,lat1,lon2,lat2);
[lati loni]
geoshow(lati,loni,'DisplayType','polygon','FaceColor','g')

ans =
         0    1.0000
   -0.4409    0.8819
         0         0
    0.4409    0.8819
         0    1.0000
```

5 Compute the union polygon and plot it as a magenta patch:

```
[lonu,latu] = polybool('union',lon1,lat1,lon2,lat2);
[latu lonu]
geoshow(latu,lonu,'DisplayType','polygon','FaceColor','m')

ans =
   -1.0000    2.0000
   -0.4409    0.8819
   -0.5000    0.8660
   -0.8660    0.5000
   -1.0000    0.0000
   -0.8660   -0.5000
   -0.5000   -0.8660
         0   -1.0000
    0.5000   -0.8660
    0.8660   -0.5000
    1.0000   -0.0000
    0.8660    0.5000
    0.5000    0.8660
    0.4409    0.8819
    1.0000    2.0000
   -1.0000    2.0000
```

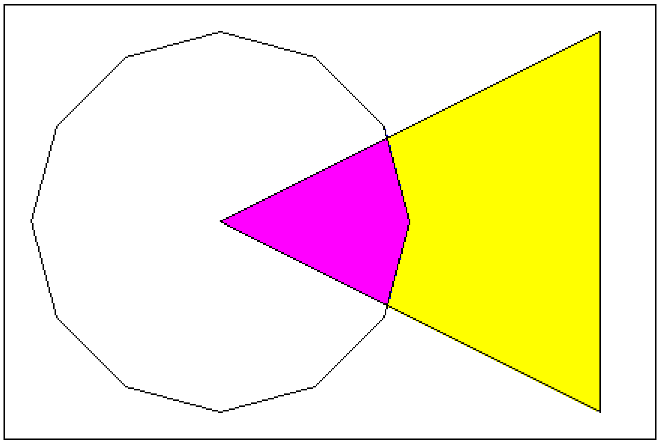
6 Compute the exclusive OR polygon and plot it as a yellow patch:

```
[lonx,latx] = polybool('xor',lon1,lat1,lon2,lat2);
[latx lonx]
geoshow(latx,lonx,'DisplayType','polygon','FaceColor','y')

ans =
   -1.0000    2.0000
   -0.4409    0.8819
   -0.5000    0.8660
   -0.8660    0.5000
   -1.0000    0.0000
   -0.8660   -0.5000
   -0.5000   -0.8660
         0   -1.0000
    0.5000   -0.8660
    0.8660   -0.5000
    1.0000   -0.0000
    0.8660    0.5000
    0.5000    0.8660
    0.4409    0.8819
    1.0000    2.0000
   -1.0000    2.0000
       NaN       NaN
    0.4409    0.8819
         0         0
   -0.4409    0.8819
         0    1.0000
    0.4409    0.8819
```

7 Subtract the triangle from the 12-sided polygon and plot the resulting concave polygon as a white patch:

```
[lonm,latm] = polybool('minus',lon1,lat1,lon2,lat2);
[latm lonm]
geoshow(latm,lonm,'DisplayType','polygon','FaceColor','w')

ans =
    0.8660    0.5000
    0.5000    0.8660
    0.4409    0.8819
         0         0
   -0.4409    0.8819
   -0.5000    0.8660
   -0.8660    0.5000
   -1.0000    0.0000
   -0.8660   -0.5000
   -0.5000   -0.8660
         0   -1.0000
    0.5000   -0.8660
    0.8660   -0.5000
    1.0000   -0.0000
    0.8660    0.5000
```

The final set of colored shapes is shown below.



See the `polybool` reference page for further information.

Cutting Polygons at the Date Line

Polygon set operations treat input vectors as plane coordinates. The `polyxpoly` function can be confused by geographic data that has discontinuities in longitude coordinates at date-line crossings. This can happen when points with longitudes near 180° connect to points with longitudes near -180°, as might be the case for eastern Siberia and Antarctica, and also for small circles and other patch objects generated by toolbox functions.

You can prepare such geographic data for use with `polybool` or for patch rendering by cutting the polygons at the date line with the `flatearthpoly` function. The result of `flatearthpoly` is a polygon with points inserted to follow the date line up to the pole, traverse the longitudes at the pole, and return to the date line crossing along the other edge of the date line.

Removing Discontinuities from a Small Circle

1 Create an orthographic view of the Earth and plot `coast` on it:

```
axesm ortho
setm(gca,'Origin', [60 170]); framem on; gridm on
```

```
load coast
plotm(lat, long)
```

2 Generate a small circle that encompasses the North Pole and color it yellow:

```
[latc,lonc] = scircle1(75,45,30);
patchm(latc,lonc,'y')
```

3 Flatten the small circle with `flatearthpoly`:

```
[latf,lonf] = flatearthpoly(latc,lonc);
```

4 Plot the cut circle that you just generated as a magenta line:

```
plotm(latf,lonf,'m')
```

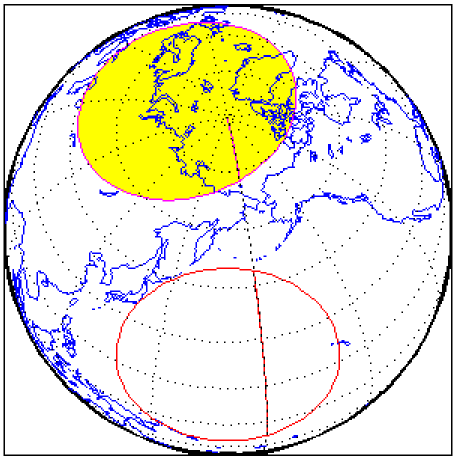
5 Generate a second small circle that does not include a pole:

```
[latc1 lonc1] = scircle1(20, 170, 30);
```

6 Flatten it and plot it as a red line:

```
[latf1,lonf1] = flatearthpoly(latc1,lonc1);
plotm(latf1,lonf1,'r')
```

The following figure shows the result of these operations. Note that the second small circle, which does not cover a pole, has been clipped into two pieces along the date line. On the right, the polygon for the first small circle is plotted in plane coordinates to illustrate its flattened shape.



The `flatearthpoly` function assumes that the interior of the polygon being flattened is in the hemisphere that contains most of its edge points. Thus a polygon produced by `flatearthpoly` does not cover more than a hemisphere.

Note As this figure illustrates, you do not need to use `flatearthpoly` to prepare data for a map display. The Mapping Toolbox display functions automatically cut and trim geographic data if required by the map projection. Use this function only when conducting set operations on polygons.

See the `flatearthpoly` reference page for further information.

Building Buffer Zones

A *buffer zone* is the area within a specified distance of a map feature. For vector geodata, buffer zones are constructed as polygons. For raster geodata, buffer zones are collections of contiguous, identically coded grid cells. When the feature is a polygon, a buffer zone can be defined as the locus of points within a certain distance of its boundary, either inside or outside the polygon. A buffer zone for a linear object is the locus of points a certain distance away from it. Buffer zones form equidistant contour lines around objects.

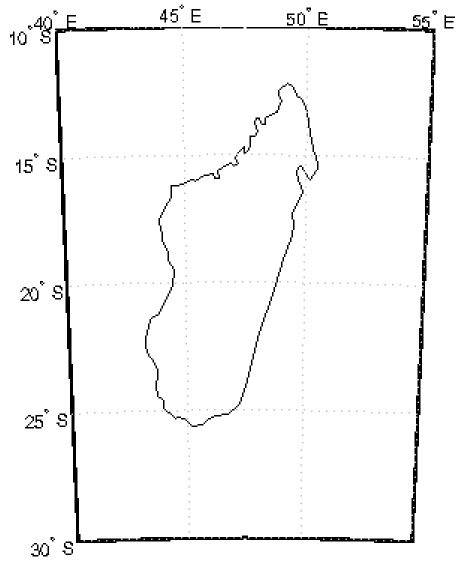
The bufferm function computes and returns vectors that represent a set of points that define a buffer zone. It forms the buffer by placing small circles at the vertices of the polygon and rectangles along each of its line segments, and applying the union set operation to these objects.

Generating a Buffer Around a Compound Polygon

Demonstrate bufferm using a compound polygon representing the Island of Madagascar that you extract from the landareas data set. The boundary of Madagascar is passed to bufferm as NaN-clipped latitude and longitude vectors. Using this data, compute a buffer zone at a distance of 0.75 degrees out from the boundaries of Madagascar:

1 Create a base map of the area surrounding Madagascar, and hide the border:

```
ax = worldmap('madagascar');
madagascar = shaperead('landareas',...
    'UseGeoCoords', true,...
    'Selector', {@(name)strcmpi(name,'Madagascar'), 'Name'});
geoshow(ax, madagascar, 'FaceColor', 'none');
madaLat = madagascar.Lat;
madaLon = madagascar.Lon;
```



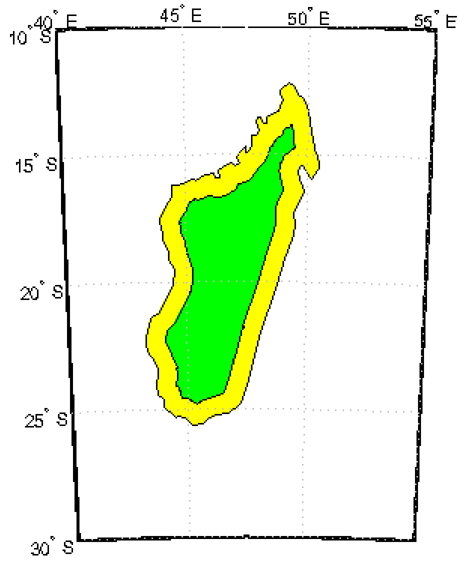
2 Use `bufferm` to process the polygon and output a buffer zone .75 degrees inland:

```
[latb,lonb] = bufferm(madaLat, madaLon, .75, 'in');
```

This can take several minutes, because of the great number of geometric computations that bufferm is performing.

3 Show the buffer zone in yellow, and the rest of the region in green. This is achieved by drawing Madagascar in yellow and the buffer zone in green:

```
patchesm(madaLat, madaLon, 'y')
patchesm(latb, lonb, 'g')
```



Trimming Vector Data to a Rectangular Region

It is not unusual for vector data to extend beyond the geographic region currently of interest. For example, you might have coastline data for the entire world (such as the `coast` data set), but are interested in mapping Australia only. In this and other situations, you might want to eliminate unnecessary data from the workspace and from calculations in order to save memory or to speed up processing and display.

Line data and patch data need to be trimmed differently. You can trim line data by simply removing points outside the region of interest by clipping lines at the map frame or to some other defined region. Patch data requires a more complicated method to ensure that the patch objects are correctly formed.

For the vector data, two functions are available to achieve this. If the vectors are to be handled as line data, the `maptriml` function returns variables containing only those points that lie within the defined region. If, instead, you want to maintain polygon format, use the `maptrimp` function. Be aware, however, that patch-trimmed data is usually larger and more expensive to compute.

Note When drawing maps, Mapping Toolbox display functions automatically trim vector geodata to the region specified by the frame limits (`FLatLimit` and `FLonLimit` map axes properties) for azimuthal projections, or to frame or map limits (`MapLatLimit` and `MapLonLimit` map axes properties) for nonazimuthal projections. The trimming is done internally in the display routine, keeping the original data intact. For further information on trimming vector geodata, see "Axes for Drawing Maps" on page 4-12, along with the reference pages for the trimming functions.

Trimming Vectors to Form Lines and Polygons

1 Load the `coast` MAT-file for the entire world:

```
load coast
```

2 Define a region of interest centered on Australia:

```
latlim = [-50 0]; longlim = [105 160];
```

3 Use `maptriml` to delete all data outside these limits, producing line vectors:

```
[linelat,linelong] = maptriml(lat,long,latlim,longlim);
```

4 Do this again, but use `maptrimp` to produce polygon vectors:

```
[polylat,polylong] = maptrimp(lat,long,latlim,longlim);
```

5 See how much data has been reduced:

```
whos
```

```
  Name              Size            Bytes  Class

  lat            9589x1             76712  double
  latlim            1x2                16  double
  linelat         870x1              6960  double
  linelong        870x1              6960  double
  long           9589x1             76712  double
```

```
longlim         1x2                16  double
polylat       878x1              7024  double
polylong      878x1              7024  double
```

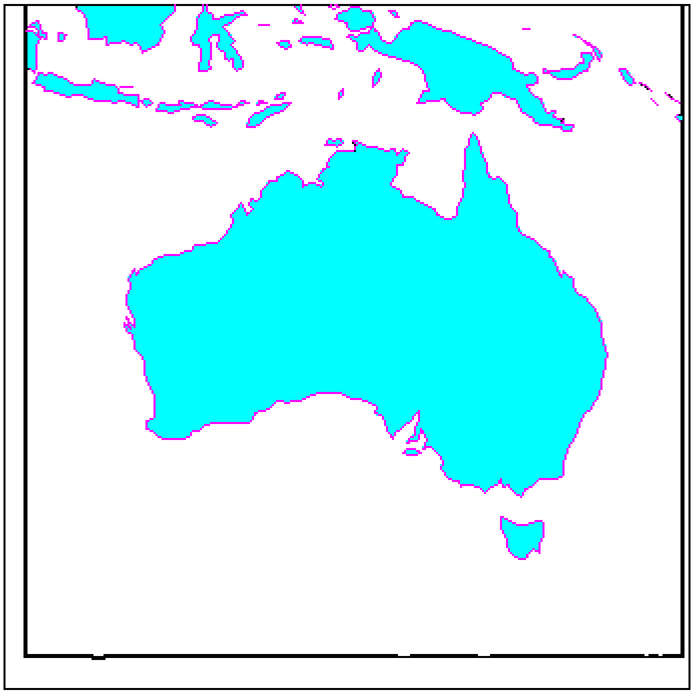
Note that the clipped data is only 10% as large as the original data set.

6 Plot the trimmed patch vectors on a Miller projection:

```
axesm('MapProjection', 'miller', 'Frame', 'on',...
'FlatLimit', latlim, 'FlonLimit', longlim)
patchesm(polylat, polylong, 'c')
```

7 Plot the trimmed line vectors to see that they conform to the patches:

```
plotm(linelat, linelong, 'm')
```



See the `maptriml` and `maptrimp` reference pages for further information.

Trimming Vector Data to an Arbitrary Region

Often a set of data contains unwanted data mixed in with the desired values. For example, your data might include vectors covering the entire United States, but you only want to work with those falling in Alabama. Sometimes a data set contains noise—perhaps three or four points out of several thousand are obvious errors (for example, one of your city points is in the middle of the ocean). In such cases, locating outliers and errors in the data arrays can be quite tedious.

The `filterm` command uses a data grid to filter a vector data set. Its calling sequence is as follows:

```
[flats,flons] = filterm(lats,lons,grid,refvector,allowed)
```

Each location defined by `lats` and `lons` is mapped to a cell in `grid`, and the value of that grid cell is obtained. If that value is found in `allowed`, that point is output to `flats` and `flons`. Otherwise, the point is filtered out.

The grid might encode political units, and the allowed values might be the code or codes indexing certain states or countries (e.g., Alabama). The grid might also be real-valued (e.g., terrain elevations), although it could be awkward to specify all the values allowed. More often, logical or relational operators give better results for such grids, enabling the allowed value to be `1` (for `true`). For example, you could use this transformation of the `topo` grid:

```
[flats,flons] = filterm(lats,lons,double(topo>0),topolegend,1)
```

The output would be those points in `lats` and `lons` that occupy dry land (mostly because some water bodies are above sea level).

For further information, see the `filterm`reference page. Also see "Data Grids as Logical Variables" on page 7-40.

Simplifying Vector Coordinate Data

Avoiding visual clutter in composing maps is an essential part of cartographic presentation. In cartography, this is described as map generalization, which involves coordinating many techniques, both manual and automated. Limiting the number of points in vector geodata is an important part of generalizing maps, and is especially useful for conditioning cartographic data, plotting maps at small scales, and creating versions of geodata for use at small scales.

An easy, but naive, approach to point reduction is to discard every *n*th element in each coordinate vector (simple decimation). However, this can result in poor representations of the original shapes. The toolbox provides a function to eliminate insignificant geometric detail in linear and polygonal objects, while still maintaining accurate representations of their shapes. The `reducem` function implements a powerful line simplification algorithm (known as Douglas-Peucker) that intelligently selects and deletes visually redundant points.

The `reducem` function takes latitude and longitude vectors, plus an optional linear tolerance parameter as arguments, and outputs reduced (simplified) versions of the vectors, in which deviations perpendicular to local "trend lines" in the vectors are all greater than the tolerance criterion. Endpoints of vectors are preserved. Optional outputs are an error measure and the tolerance value used (it is computed when you do not supply a value).

Note Simplified line data might not always be appropriate for display. If all or most intermediate points in a feature are deleted, then lines that appear straight in one projection can be incorrectly displayed as straight lines in others, and separate lines can be caused to intersect. In addition, when you are reducing data over large world regions, the effective degree of reduction near the poles are less than that achieved near the equator, due to the fact that the algorithm treats geographic coordinates as if they were planar.

Using reducem to Simplify Lines

The `reducem` function works on both patch and line data. Getting results that look right at an intended scale might require some experimentation with the tolerance parameter. The best way to proceed might be to allow the tolerance to default, and have `reducem` return the tolerance it computed as the fourth return argument. If the output still has too much detail, then double the tolerance and try again. Similarly, if the output lines do not have enough detail, halve the tolerance and try again. You can also use the third return parameter, which indicates the percentage of line length that was eliminated by reduction, as a guide to achieve consistent simplification results, although this parameter is sensitive to line geometry and thus can vary by feature type.

To demonstrate the use of `reducem`, this example extracts the outline of the state of Massachusetts from the `usastatehi` high-resolution shapefile:

1 Read Massachusetts data from the shapefile. Use the `Selector` parameter to read only the vectors representing the Massachusetts state boundaries:

```
ma = shaperead('usastatehi.shp',...
    'UseGeoCoords', true,...
    'Selector', {@(name)strcmpi(name,'Massachusetts'), 'Name'});
```

2 Extract the coordinate data for simplification. There are 957 points to begin with:

```
maLat = ma.Lat;
maLon = ma.Lon;
numel(maLat)


ans =
   957
```

3 Use `reducem` to simplify the boundary vectors, and inspect the results:

```
[maLat1, maLon1, cerr, tol] = reducem(maLat', maLon');
numel(maLat1)


ans =
   252
```

4 The number of points used to represent the boundary has dropped from 958 to 253. Compute the degree of reduction:

```
numel(maLat1)/numel(maLat)


ans =
      0.2633
```

The vectors have been reduced to about a quarter of their original size using the default tolerance.

5 Examine the error and tolerance values returned by `reducem`:

```
[cerr tol]


ans =
    0.0331    0.0060
```

The `cerr` value says that only 3.3% of total boundary length was eliminated (despite removing 74% of the points). The tolerance that achieved this was computed by `reducem` as 0.006 decimal degrees, or about 0.66 km.

6 Use `geoshow` to plot the reduced outline in red over the original outline in blue:

```
figure
axesm('MapProjection', 'eqdcyl', 'FlatLim', [41.1 43.0],...
'FlonLim', [-69.8, -73.6], 'Frame', 'off', 'Grid', 'off');
geoshow(maLat, maLon, 'DisplayType', 'line', 'color', 'blue')
geoshow(maLat1, maLon1, 'DisplayType', 'line', 'color', 'red')
```

Differences in details are not apparent unless you zoom in two or three times; click the Zoom tool to explore the map.

7 Double the tolerance, and reduce the original boundary into new variables:

```
[maLat2,maLon2,cerr2,tol2] = reducem(maLat', maLon', 0.012);
```

8 Repeat step 3 with new data and plot it in dark green:

```
numel(maLat2)/numel(maLat)


ans =
      0.1641

[cerr2 tol2]
```

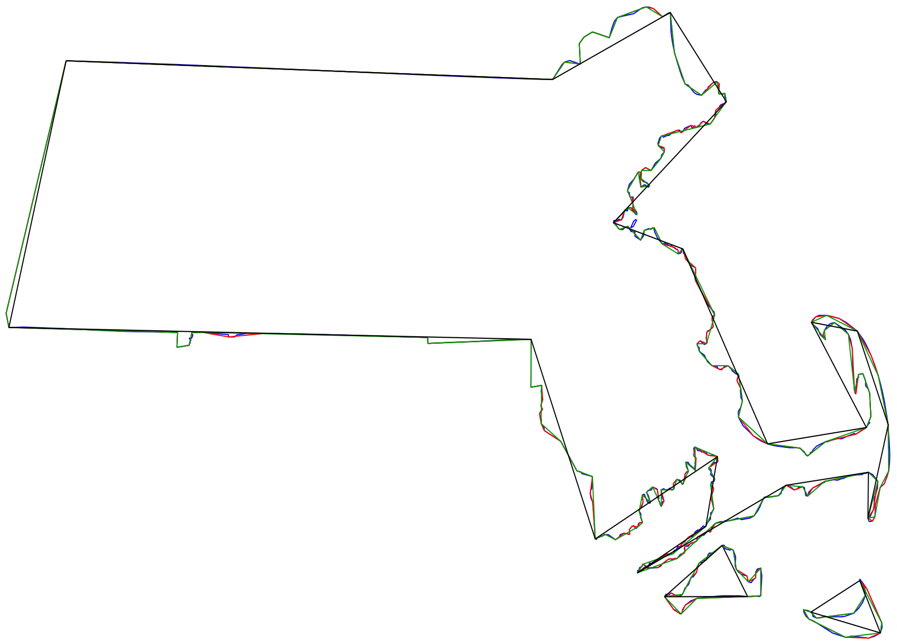
```
ans =
     0.0517 0.0120

geoshow(maLat2, maLon2, 'DisplayType', 'line', 'color', [0 .6 0])
```

Now you have removed 84% of the points, and 5.2% of total length.

9 Repeat one more time, raising the tolerance to 0.1 degrees, and plot the result in black:

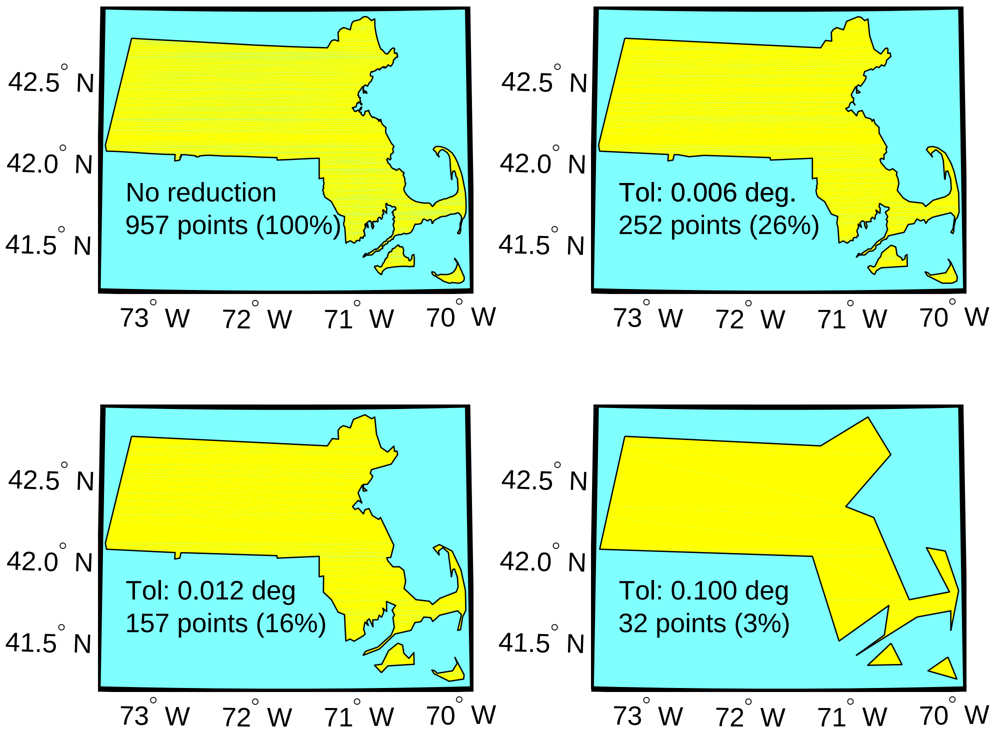
```
[maLat3, maLon3, cerr3, tol3] = reducem(maLat', maLon', 0.1);
geoshow(maLat3, maLon3, 'DisplayType', 'line', 'color', 'black')
```

As overlaid with the original data, the reduced state boundaries look like this.



As this example and the composite map below demonstrate, the visual effects of point reduction are subtle, up to a point. Most of the vertices can be eliminated before the effects of line simplification are very noticeable. Experimentation is usually required, because the visual effects a given value for a tolerance yield depend on the degrees and types of line complexity, and they are often nonlinear with respect to tolerance values. Also, the extent of line detail reduction should be informed by the purpose of the map and the scale at which it is to be displayed.

Note This exercise generalized a set of disconnected patches. When patches are contiguous (such as the U.S. state outlines), using `reducem` can result in inconsistencies in boundary representation and gaps at points where states meet. For best results, `reducem` should be applied to line data.



See the `reducem` reference page for further information.

Manipulating Raster Geodata

In this section...

Vector-to-Raster Data Conversion

You can convert latitude-longitude vector data to a grid at any resolution you choose to make a raster base map or grid layer. Certain Mapping Toolbox GUI tools help you do some of this, but you can also perform vector-to-raster conversions from the command line. The principal function for gridding vector data is `vec2mtx`, which allocates lines to a grid of any size you indicate, marking the lines with 1s and the unoccupied grid cells with 0s. The grid contains doubles, but if you want a logical grid (see "Data Grids as Logical Variables" on page 7-40, below) cast the result to be a logical array.

If the vector data consists of polygons (patches), the gridded outlines are all hollow. You can differentiate them using the `encodem` function, calling it with an array of rows, columns, and seed values to produce a new grid containing polygonal areas filled with the seed values to replace the binary values generated by `vec2mtx`.

Creating Data Grids from Vector Data

To demonstrate vector-to-raster data conversion, use patch data for Indiana from the `usastatehi` shapefile:

1 Use `shaperead` to get the patch data for the boundary:

```
indiana = shaperead('usastatehi.shp',...
    'UseGeoCoords', true,...
    'Selector', {@(name)strcmpi('Indiana',name), 'Name'});
inLat = indiana.Lat;
inLon = indiana.Lon;
```

2 Set the grid density to be 40 cells per degree, and use `vec2mtx` to rasterize the boundary and generate a referencing vector for it:

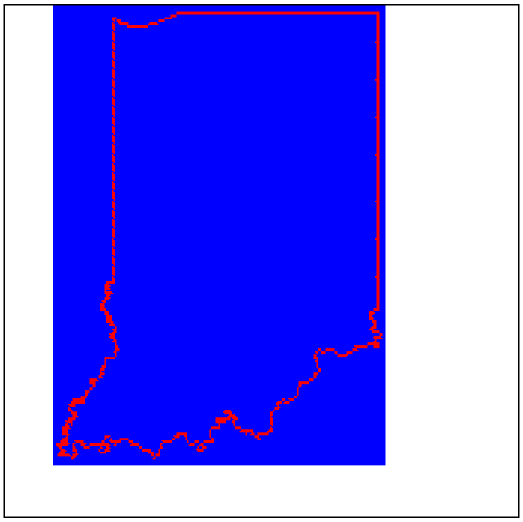
```
gridDensity = 40;
[inGrid, inRefVec] = vec2mtx(inLat, inLon, gridDensity);
whos
```

```
 Name                  Size                 Bytes  Class

  gridDensity          1x1                      8  double
  inGrid             164x137               179744  double
  inLat                1x626                 5008  double
  inLon                1x626                 5008  double
  inRefVec             1x3                     24  double
  indiana              1x1                  10960  struct
```

The resulting grid contains doubles, and has 80 rows and 186 columns.

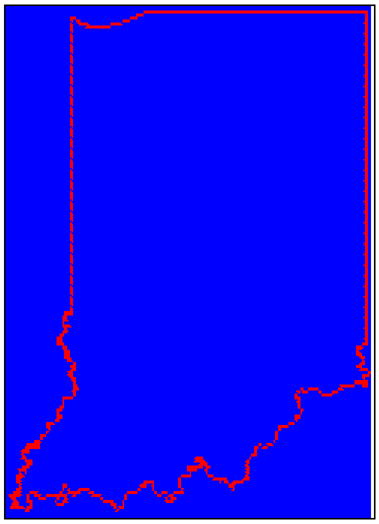
3 Make a map of the data grid in contrasting colors:

```
figure
axesm eqdcyl
meshm(inGrid, inRefVec)
colormap jet(4)
```



4 Set up the map limits:

```
[latlim, lonlim] = limitm(inGrid, inRefVec);
setm(gca, 'Flatlimit', latlim, 'FlonLimit', lonlim)
tightmap
```



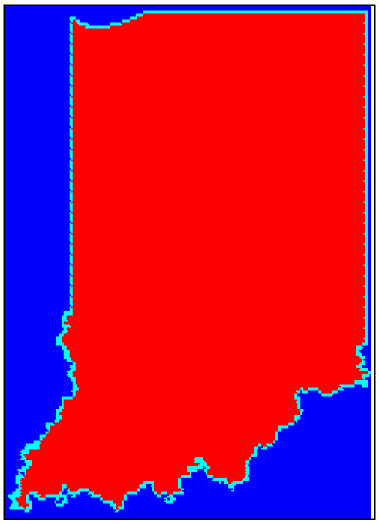
5 To fill (recode) the interior of Indiana, you need a seed point (which must be identified by row and column) and a seed value (to be allocated to all cells within the polygon). Select the middle row and column of the grid and choose an index value of 3 to identify the territory when calling encodem to generate a new grid:

```
inPt = round([size(inGrid)/2, 3]);
inGrid3 = encodem(inGrid, inPt,1);
```

The last argument (1) identifies the code for boundary cells, where filling should halt.

6 Clear and redraw the map using the filled grid:

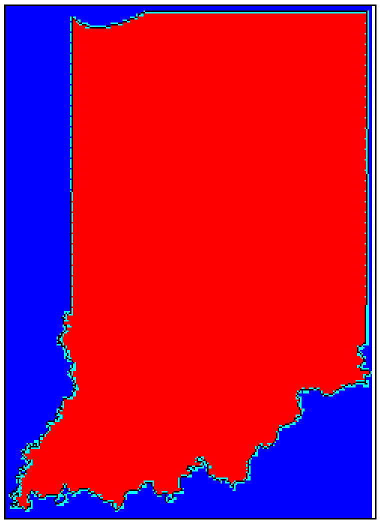
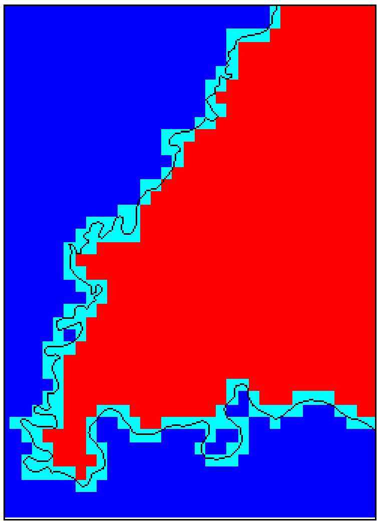
```
meshm(inGrid3, inRefVec)
```



7 Plot the original vectors on the grid to see how well data was rasterized:

```
plotm(inLat, inLon,'k')
```

The resulting map is shown on the left below. Use the **Zoom** tool on the figure window to examine the gridding results more closely, as the right-hand figure shows.

See the `vec2mtx` and `encodem` reference pages for further information. `imbedm` is a related function for gridding point values.

Using a GUI to Rasterize Polygons

In the previous example, had you wanted to include the states that border Indiana, you could also have extracted Illinois, Kentucky, Ohio, and Michigan, and then deleted unwanted areas of these polygons using `maptrimp` (see "Trimming Vector Data to a Rectangular Region" on page 7-22 for specific details on its use). Use the `seedm` function with seed points found using the `getseeds` GUI to fill multiple polygons after they are gridded:

1 Extract the data for Indiana and its neighbors by passing their names in a cell array to `shaperead`:

```
pcs = {'Indiana', 'Michigan', 'Ohio', 'Kentucky', 'Illinois'};

centralUS = shaperead('usastatelo.shp',...
    'UseGeoCoords', true,...
    'Selector',{@(name)any(strcmpi(name,pcs),2), 'Name'});
```

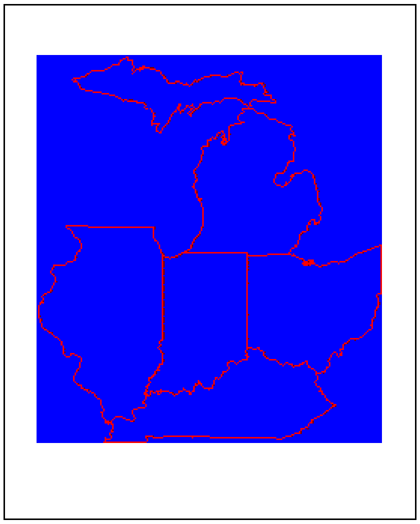
```
meLat = [centralUS.Lat];
meLon = [centralUS.Lon];
```

2 Rasterize the trimmed polygons at a 1-arc-minute resolution (60 cells per degree), also producing a referencing vector:

```
[meGrid, meRefVec] = vec2mtx(meLat, meLon, 60);
```

3 Set up a map figure and display the binary grid just created:

```
figure
axesm eqdcyl
geoshow(meLat, meLon, 'Color', 'r');
meshm(meGrid, meRefVec)
colormap jet(8)
```



4 Use `getseeds` to interactively pick seed points for Indiana, Michigan, Ohio, Kentucky, and Illinois, in that order:

```
[row,col,val] = getseeds(meGrid, meRefVec, 5, [3 4 5 6 7]);
[row col val]
```

```
ans =
   207   140     3
   219   326     4
   212   506     5
    56   459     6
   393   433     7
```

The MATLAB prompt returns after you pick five locations in the figure window. As you chose them yourself, your `row` and `col` numbers will differ.

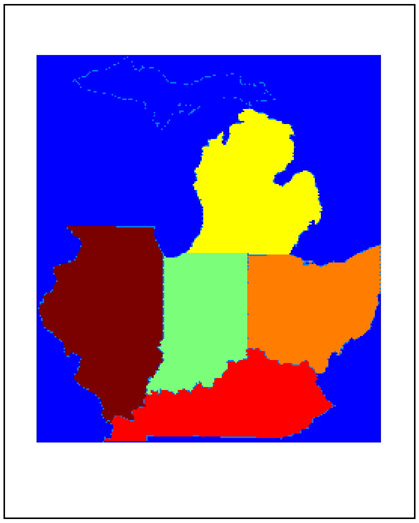
5 Use `encodem` to fill each country with a unique value, producing a new grid:

```
meGrid5 = encodem(meGrid, [row col val], 1);
```

6 Clear the display and display `meGrid5` to see the result:

```
clma
meshm(meGrid5, meRefVec)
```

The rasterized map of Indiana and its neighbors is shown below.



See the `getseeds` reference page for more information. The`maptrim` and `seedm` GUI tools are also useful in this context.

Data Grids as Logical Variables

You can apply logical criteria to numeric data grids to create *logical grids*. Logical grids are data grids consisting entirely of 1s and 0s. You can create them by performing logical tests on data grid variables. The resulting binary grid is the same size as the original grid(s) and can use the same referencing vector, as the following hypothetical data operation illustrates:

```
logicalgrid = (realgrid > 0);
```

This transforms all values greater than 0 into 1s and all other values to 0s. You can apply multiple conditions to a grid in one operation:

```
logicalgrid = (realgrid >- 100)&(realgrid < 100);
```

If several grids are the same size and share the same referencing vector (i.e., the grids are co-registered), you can create a logical grid by testing joint conditions, treating the individual data grids as map layers:

```
logicalgrid = (population > 10000)&(elevation < 400)&...
              (country == nigeria);
```

Several Mapping Toolbox functions enable the creation of logical grids using logical and relational operators. Grids resulting from such operations contain logical rather than numeric values (which reduce storage by a factor of 8), but might need to be cast to `double` in order to be used in certain functions. Use the `onem` and `zerom` functions to create grids of all 1s and all 0s.

Obtaining the Area Occupied by a Logical Grid Variable

You can analyze the results of logical grid manipulations to determine the area satisfying one or more conditions (either coded as 1s or an expression that yields a logical value of 1). The `areamat` function can provide the fractional surface area on the globe associated with 1s in a logical grid. Each grid element is a quadrangle, and the sum of the areas meeting the logical condition provides the total area:

1 You can use the topo grid and the greater-than relational operator to determine what fraction of the Earth lies above sea level:

```
load topo
topoR = makerefmat('RasterSize', size(topo), ...
    'Latlim', [-90 90], 'Lonlim', [0 360]);
a = areamat((topo > 0),topoR)


a =
 0.2890
```

The answer is about 30%. (Note that land areas below sea level are excluded.)

2 You can include a planetary radius in specified units if you want the result to have those units. Here is the same query specifying units of square kilometers:

```
a = areamat((topo > 0),topoR,almanac('earth','radius'))


a =
 1.4739e+08
```

3 Use the `usamtx` data grid codes to find the area of a specific state within the U.S.A. As an example, determine the area of the state of Texas, which is coded as 46 in the `usamtx` grid:

```
load usamtx
a = areamat((map == 46), refvec, almanac('earth', 'radius'))


a =
    6.2528e+005
```

The grid codes 625,277 square kilometers of land area as belonging to the U.S.

4 You can construct more complex queries. For instance, using the last example, compute what portion of the land area of the conterminous U.S.

that Texas occupies (water and bordering countries are coded with 2 and 3, respectively):

```
usaland = areamat((map > 3 | map == 1), maplegend);
texasland = areamat((map == 46), maplegend);
texasratio = texasland/usaland


texasratio =
    0.0735
```

This indicates that Texas occupies roughly 7.35% of the land area of the U.S.

For further information, see the `areamat` reference page.

Data Grid Values Along a Path

A common application for gridded geodata is to calculate data values along a path, for example, the computation of terrain height along a transect, a road, or a flight path. The `mapprofile` function does this, based on numerical data defining a set of waypoints, or by defining them interactively via graphic input from a map display. Values computed for the resulting profile can be displayed in a new plot or returned as output arguments for further analysis or display.

Using the mapprofile Function

The following example computes the elevation profile along a straight line:

1. Load the Korean elevation data:

   ```
   figure;
   load korea
   ```

2. Get its latitude and longitude limits using `limitm` and use them to set up a map frame via `worldmap`:

   ```
   [latlim, lonlim] = limitm(map, maplegend);
   worldmap(latlim, lonlim)
   ```

   `worldmap` plots only the map frame.

3 Render the map and apply a digital elevation model (DEM) colormap to it:

```
meshm(map,maplegend,size(map),map)
demcmap(map)
```

4 Define endpoints for a straight-line transect through the region:

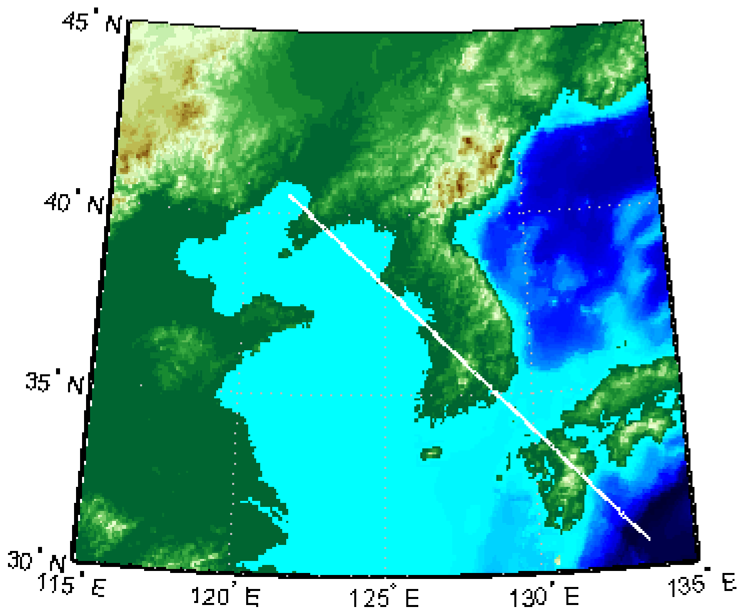
```
plat = [40.5 30.7];
plon = [121.5 133.5];
```

5 Compute the elevation profile, defaulting the track type to `great circle` and the interpolation type to `bilinear`:

```
[z,rng,lat,lon] = mapprofile(map,maplegend,plat,plon);
```

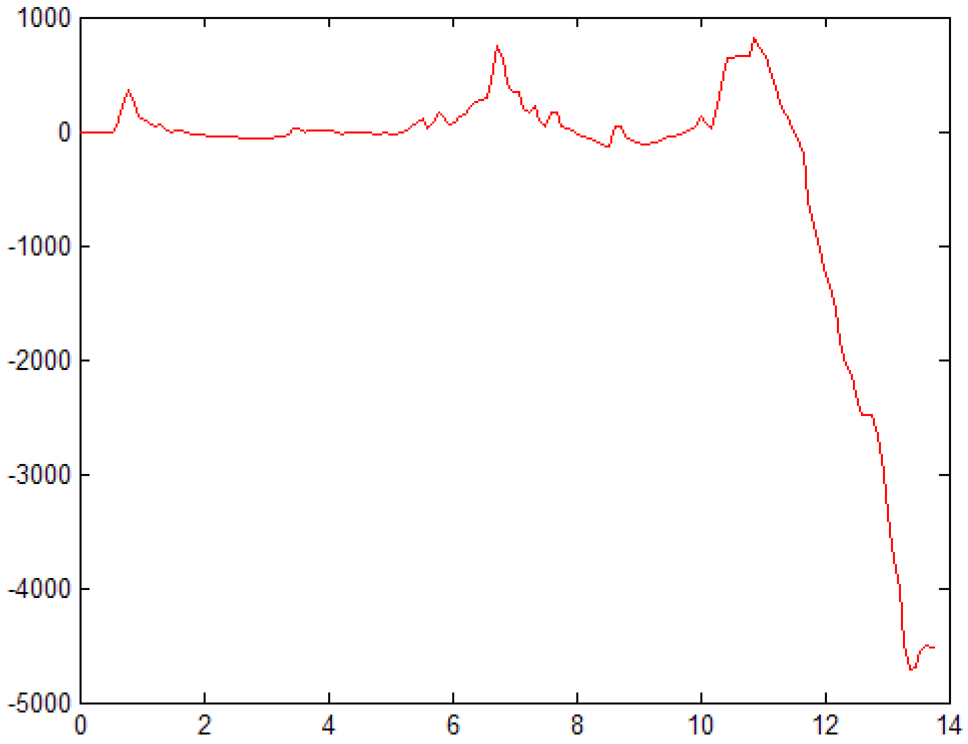
6 Draw the transect in 3-D so it follows the terrain:

```
plot3m(lat,lon,z,'w','LineWidth',2)
```



7 Construct a plot of transect elevation and range:

```
figure; plot(rng,z,'r')
```



The `mapprofile` function has other useful options, including the ability to interactively define tracks and specify units of distance for them. For further information, see the `mapprofile` reference page.

Data Grid Gradient, Slope, and Aspect

A map profile is often used to determine slopes along a path. A related application is the calculation of slope at all points on a matrix. The `gradientm` function uses a finite-difference approach to compute gradients for either a regular or a georeferenced data grid. The function returns the components of the gradient in the north and east directions (i.e., north-to-south, east-to-west), as well as slope and aspect. The *gradient* components are the change in the grid variable per meter of distance in the north and east directions. If the grid contains elevations in meters, the *aspect* and *slope* are the angles of the surface normal clockwise from north and up from the horizontal. Slope is defined as the change in elevation per unit distance along the path of steepest ascent or descent from a grid cell to one of its eight immediate neighbors, expressed as the arctangent. The angles are in units of degrees by default.

Computing Gradient Data from a Regular Data Grid

The following example illustrates computation of gradient, slope, and aspect data grids for a regular data grid based on the MATLAB `peaks` function:

1 Construct a 100-by-100 grid using the `peaks` function and construct a referencing matrix for it:

```
datagrid = 500*peaks(100);
R = makerefmat('RasterSize',size(datagrid));
```

2 Use `gradientm` to generate grids containing aspect, slope, gradients to north, and gradients to east:

```
[aspect,slope,gradN,gradE] = gradientm(datagrid,R);
whos


  Name            Size             Bytes  Class

  aspect       100x100             80000  double
  datagrid     100x100             80000  double
  gradE        100x100             80000  double
  gradN        100x100             80000  double
  gridrv         1x3                  24  double
  slope        100x100             80000  double
```

3 Map the surface data in a cylindrical equal area projection. Start with the original elevations:

```
figure; axesm eqacyl
meshm(datagrid,R)
colormap (jet(64))
colorbar('vert')
```

```
    title('Peaks: elevation')
    axis square
```

4 Clear the frame and display the `slope` grid:

```
    figure; axesm eqacyl
    meshm(slope,R)
    colormap (jet(64))
    colorbar('vert');
    title('Peaks: slope')
```

5 Map the `aspect` grid:

```
    figure; axesm eqacyl
    meshm(aspect,R)
    colormap (jet(64))
    colorbar('vert');
    title('Peaks: aspect')
```

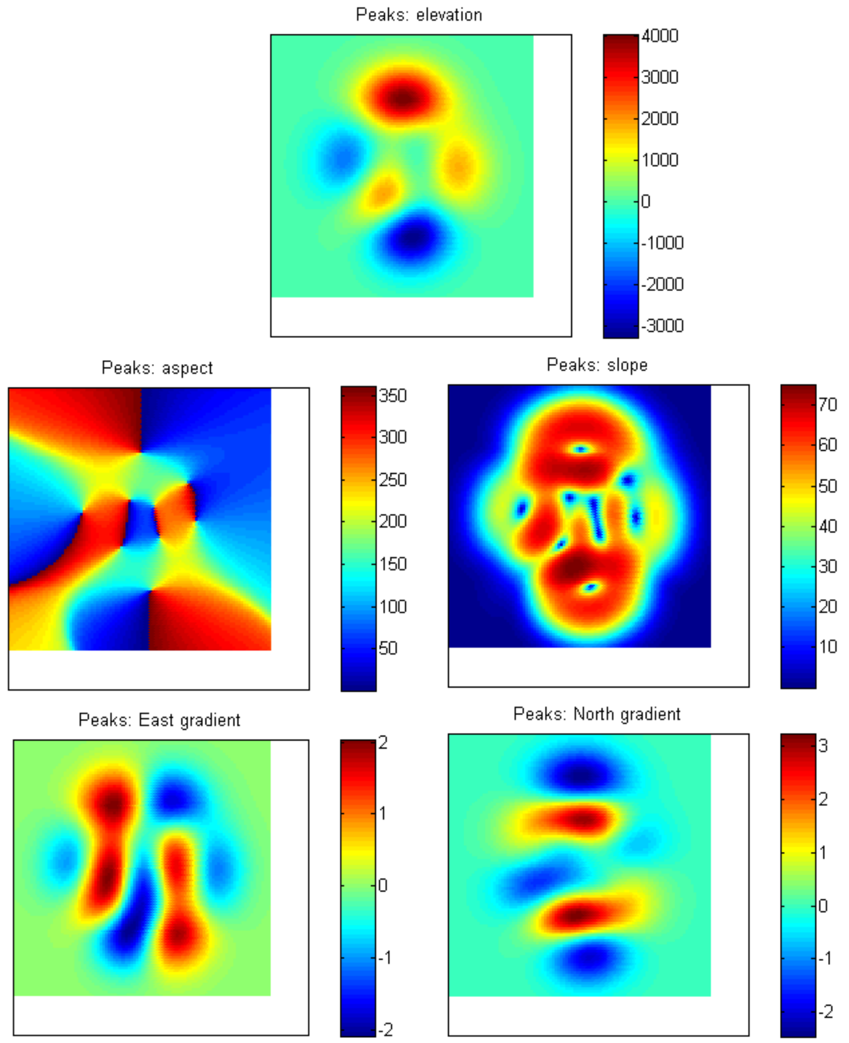
6 Map the gradients to the north:

```
    figure; axesm eqacyl
    meshm(gradN,R)
    colormap (jet(64))
    colorbar('vert');
    title('Peaks: North gradient')
```

7 Finally, map the gradients to the east:

```
    figure; axesm eqacyl
    meshm(gradE,R)
    colormap (jet(64))
    colorbar('vert');
    title('Peaks: East gradient')
```

The maps of the `peaks` surface elevation and gradient data are shown below. See the `gradientm` reference page for additional information.


Peaks: elevation
4000
3000
2000
1000
0
-1000
-2000
-3000
Peaks: aspect
350
300
250
200
150
100
50
Peaks: slope
70
60
50
40
30
20
10
Peaks: East gradient
2
1
0
-1
-2
Peaks: North gradient
3
2
1
0
-1
-2

Using Map Projections and Coordinate Systems

All geospatial data must be flattened onto a display surface in order to visually portray what exists where. The mathematics and craft of map projection are central to this process. Although there is no limit to the ways geodata can be projected, conventions, constraints, standards, and applications generally prescribe its usage. This chapter describes what map projections are, how they are constructed and controlled, their essential properties, and some possibilities and limitations.

If you are not acquainted with the types, properties, and uses of map projections, read the first four sections. When constructing maps—especially in an environment in which a variety of projections are readily available—it is important to understand how to evaluate projections to select one appropriate to the contents and purpose of a given map.

What Is a Map Projection?

Human beings have known that the shape of the Earth resembles a sphere and not a flat surface since classical times, and possibly much earlier than that. If the world were indeed flat, cartography would be much simpler because map projections would be unnecessary.

To represent a curved surface such as the Earth in two dimensions, you must geometrically transform (literally, and in the mathematical sense, "map") that surface to a plane. Such a transformation is called a *map projection*. The term projection derives from the geometric methods that were traditionally used to construct maps, in the fashion of optical projections made with a device called *camera obscura* that Renaissance artists relied on to render three-dimensional perspective views on paper and canvas.

While many map projections no longer rely on physical projections, it is useful to think of map projections in geometric terms. This is because map projection consists of constructing points on geometric objects such as cylinders, cones, and circles that correspond to homologous points on the surface of the planet being mapped according to certain rules and formulas.

The following sections describe the basic properties of map projections, the surfaces onto which projections are developed, the types of parameters associated with different classes of projections, how projected data can be mapped back to the sphere or spheroid it represents, and details about one very widely used projection system, called Universal Transverse Mercator.

Note Most map projections in the toolbox are implemented as M-functions; however, these are only used by certain calling functions (such as `geoshow` and `axesm`), and thus have no documented public API.

For more detailed information on specific projections, browse the Chapter 14, "Map Projections Reference" (available online and in the PDF version of this document). For further reading, Appendix A, "Bibliography" provides references to books and papers on map projection.

Quantitative Properties of Map Projections

A sphere, unlike a polyhedron, cone, or cylinder, cannot be reformed into a plane. In order to portray the surface of a round body on a two-dimensional flat plane, you must first define a *developable surface* (i.e., one that can be *cut* and *flattened* onto a plane without stretching or creasing) and devise rules for systematically representing all or part of the spherical surface on the plane. Any such process inevitably leads to distortions of one kind or another. Five essential characteristic properties of map projections are subject to distortion: *shape*, *distance*, *direction*, *scale*, and *area*. No projection can retain more than one of these properties over a large portion of the Earth. This is not because a sufficiently clever projection has yet to be devised; the task is physically impossible. The technical meanings of these terms are described below.

- Shape (also called *conformality*)

  Shape is preserved locally (within "small" areas) when the scale of a map at any point on the map is the same in any direction. Projections with this property are called conformal. In them, meridians (lines of longitude) and parallels (lines of latitude) intersect at right angles. An older term for conformal is *orthomorphic* (from the Greek *orthos*, straight, and *morphe*, shape).

- Distance (also called *equidistance*)

  A map projection can preserve distances from the center of the projection to all other places on the map (but from the center only). Such a map projection is called *equidistant*. Maps are also described as equidistant when the separation between parallels is uniform (e.g., distances along meridians are maintained). No map projection maintains distance proportionality in all directions from any arbitrary point.

- Direction

  A map projection preserves direction when azimuths (angles from the central point or from a point on a line to another point) are portrayed correctly in all directions. Many azimuthal projections have this property.

- Scale

  Scale is the ratio between a distance portrayed on a map and the same extent on the Earth. No projection faithfully maintains constant scale over large areas, but some are able to limit scale variation to one or two percent.

- Area (also called *equivalence*)

  A map can portray areas across it in proportional relationship to the areas on the Earth that they represent. Such a map projection is called equal-area or equivalent. Two older terms for equal-area are *homolographic* or *homalographic* (from the Greek *homalos* or *homos*, same, and *graphos*, write), and *authalic* (from the Greek *autos*, same, and *ailos*, area), and *equireal*. Note that no map can be both equal-area and conformal.

For a complete description of the properties that specific map projections maintain, see "Summary and Guide to Projections" on page 8-63.

The Three Main Families of Map Projections

In this section...

Unwrapping the Sphere to a Plane

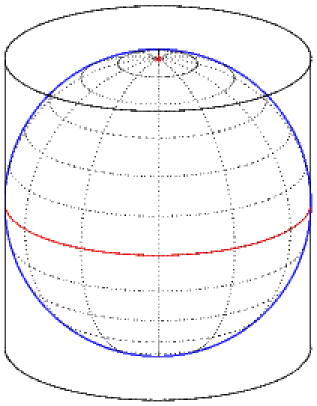
Mapmakers have developed hundreds of map projections, over several thousand years. Three large families of map projection, plus several smaller ones, are generally acknowledged. These are based on the types of geometric shapes that are used to transfer features from a sphere or spheroid to a plane. As described above, map projections are based on *developable surfaces*, and the three traditional families consist of cylinders, cones, and planes. They are used to classify the majority of projections, including some that are not analytically (geometrically) constructed. In addition, a number of map projections are based on polyhedra. While polyhedral projections have interesting and useful properties, they are not described in this guide.

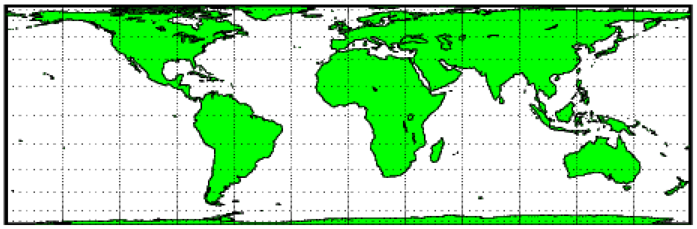
Which developable surface to use for a projection depends on what region is to be mapped, its geographical extent, and the geometric properties that areas, boundaries, and routes need to have, given the purpose of the map. The following sections describe and illustrate how the cylindrical, conic, and azimuthal families of map projections are constructed and provides some examples of projections that are based on them.

Cylindrical Projections

A *cylindrical* projection is produced by wrapping a cylinder around a globe representing the Earth. The map projection is the image of the globe projected onto the cylindrical surface, which is then unwrapped into a flat surface. When the cylinder aligns with the polar axis, parallels appear as horizontal lines and meridians as vertical lines. Cylindrical projections can be either equal-area, conformal, or equidistant. The following figure shows a regular cylindrical or *normal aspect* orientation in which the cylinder is tangent to the

Earth along the Equator and the projection radiates horizontally from the axis of rotation. The projection method is diagrammed on the left, and an example is given on the right (equal-area cylindrical projection, normal/equatorial aspect).





For a description of projection aspect, see "Projection Aspect" on page 8-10.

Some widely used cylindrical map projections are

- Equal-area cylindrical projection
- Equidistant cylindrical projection
- Mercator projection
- Miller projection
- Plate Carrée projection
- Universal transverse Mercator projection

Pseudocylindrical Map Projections

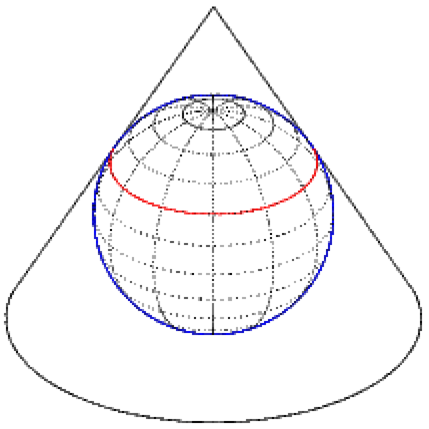
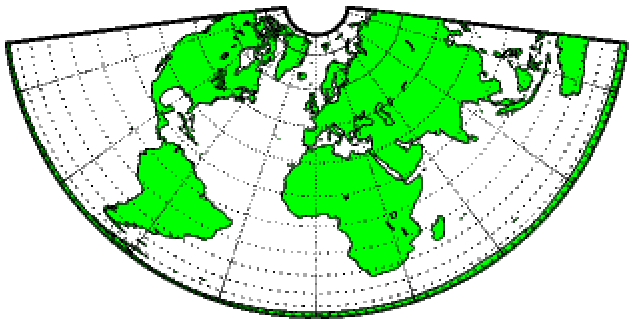
All cylindrical projections fill a rectangular plane. *Pseudocylindrical* projection outlines tend to be barrel-shaped rather than rectangular. However, they do resemble cylindrical projections, with straight and parallel latitude lines, and can have equally spaced meridians, but meridians are curves, not straight lines. Pseudocylindrical projections can be equal-area, but are not conformal or equidistant.

Some widely-used pseudocylindrical map projections are

- Eckert projections (I-VI)
- Goode homolosine projection
- Mollweide projection
- Quartic authalic projection
- Robinson projection
- Sinusoidal projection

Conic Projections

A *conic* projection is derived from the projection of the globe onto a cone placed over it. For the *normal aspect*, the apex of the cone lies on the polar axis of the Earth. If the cone touches the Earth at just one particular parallel of latitude, it is called *tangent*. If made smaller, the cone will intersect the Earth twice, in which case it is called *secant*. Conic projections often achieve less distortion at mid- and high latitudes than cylindrical projections. A further elaboration is the *polyconic* projection, which deploys a family of tangent or secant cones to bracket a succession of bands of parallels to yield even less scale distortion. The following figure illustrates conic projection, diagramming its construction on the left, with an example on the right (Albers equal-area projection, polar aspect).

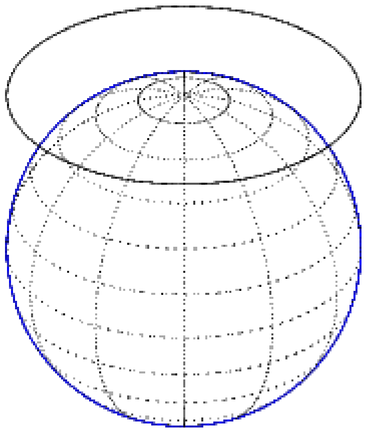



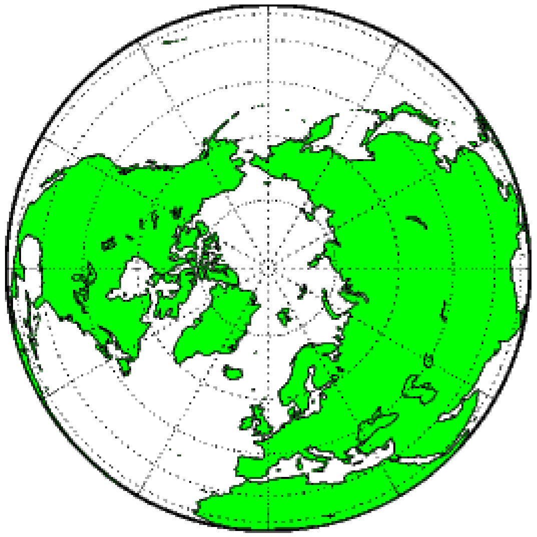
Some widely-used conic projections are

- Albers Equal-area projection
- Equidistant projection
- Lambert conformal projection
- Polyconic projection

Azimuthal Projections

An *azimuthal* projection is a projection of the globe onto a plane. In polar aspect, an azimuthal projection maps to a plane tangent to the Earth at one of the poles, with meridians projected as straight lines radiating from the pole, and parallels shown as complete circles centered at the pole. Azimuthal projections (especially the orthographic) can have equatorial or oblique aspects. The projection is centered on a point, that is either on the surface, at the center of the Earth, at the antipode, some distance beyond the Earth, or at infinity. Most azimuthal projections are not suitable for displaying the entire Earth in one view, but give a sense of the globe. The following figure illustrates azimuthal projection, diagramming it on the left, with an example on the right (orthographic projection, polar aspect).





Some widely used azimuthal projections are

- Equidistant azimuthal projection
- Gnomonic projection
- Lambert equal-area azimuthal projection
- Orthographic projection
- Stereographic projection
- Universal polar stereographic projection

For additional information on families of map projections and specific map projections, see Chapter 14, “Map Projections Reference” (available online and in the PDF version of this document).

Projection Aspect

A map projection's *aspect* is its orientation on the page or display screen. If north or south is straight up, the aspect is said to be *equatorial*; for most projections this is the *normal* aspect. When the central axis of the developable surface is oriented east-west, the projection's aspect is *transverse*. Projections centered on the North Pole or the South Pole have a *polar* aspect, regardless of what meridian is up. All other orientations have an *oblique* aspect. So far, the examples and discussions of map displays have focused on the normal aspect, by far the most commonly used. This section discusses the use of *transverse*, *oblique*, and *skew-oblique* aspects.

Projection aspect is primarily of interest in the display of maps. However, this section also discusses how the idea of projection aspect as a coordinate system transformation can be applied to map variables for analytical purposes.

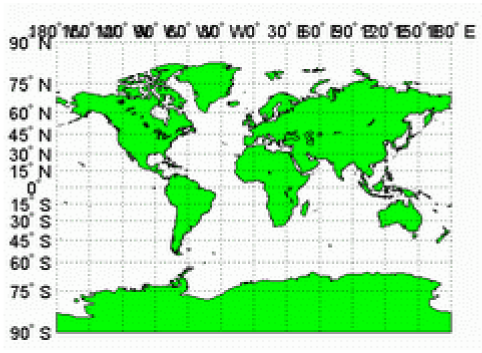
The Orientation Vector

A map axes `Origin` property is a vector describing the geometry of the displayed projection. This Mapping Toolbox property is called an *orientation vector* (prior versions called it the *origin vector*). The vector takes this form:

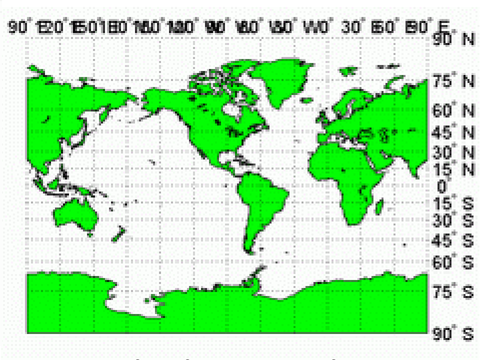
```
orientvec = [latitude longitude orientation]
```

The latitude and longitude represent the geographic coordinates of the center point of the display from which the projection is calculated. The orientation refers to the clockwise angle from *straight up* at which the North Pole points from this center point. The default orientation vector is [0 0 0]; that is, the projection is centered on the geographic point (0º,0º) and the North Pole is *straight up* from this point. Such a display is in a *normal* aspect. Changes to only the longitude value of the orientation vector do not change the aspect; thus, a normal aspect is one centered on the Equator in latitude with an orientation of 0º.

Both of these Miller projections have normal aspects, despite having different orientation vectors:



Origin at (0°,0°) with a 0° orientation
(orientation vector = [0 0 0])

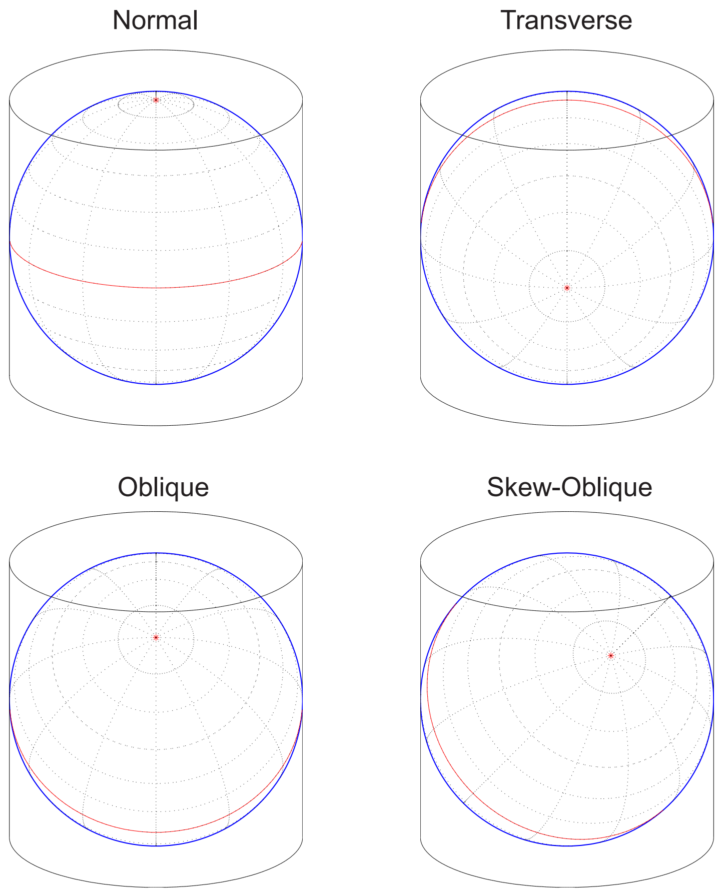


Origin at (0°,90°W) with a 0° orientation
(orientation vector = [0 -90 0])

This makes sense if you think about a simple, true cylindrical projection. This is the projection of the globe onto a cylinder wrapped around it. For normal aspects, this cylinder is tangent to the globe at the Equator, and changing the origin longitude simply corresponds to rotating the sphere about the longitudinal axis of the cylinder. If you continue with the wrapped-cylinder model, you can understand the other aspects as well.

Following this description, a *transverse* projection can be thought of as a cylinder wrapped around the globe tangent at the poles and along a meridian and its antipodal meridian. Finally, when such a cylinder is tangent along any great circle other than a meridian, the result is an *oblique* projection.

Here are diagrams of the four cylindrical map orientations, or aspects:



Of course, few projections are true cylindrical projections, but the concept of the wrapped cylinder is nonetheless a convenient way to describe aspect.

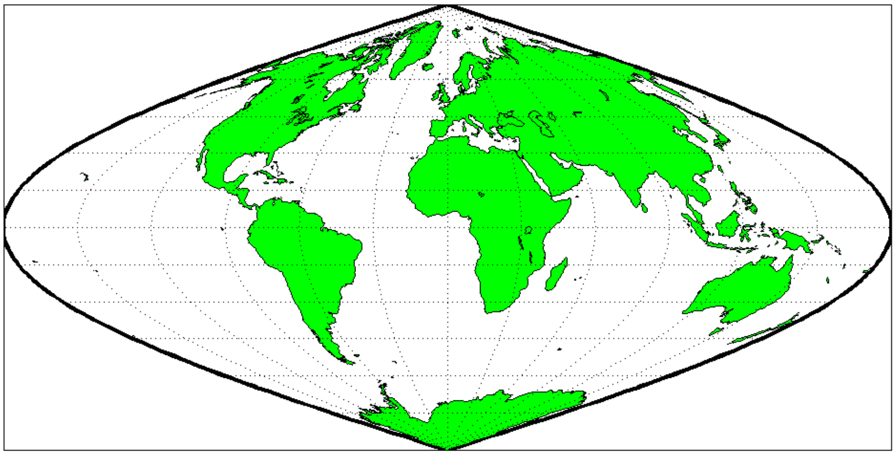
Exploring Projection Aspect

Perhaps the best way to gain an understanding of projection aspect is to experiment with orientation vectors. For the following exercise, use a pseudocylindrical projection, the sinusoidal.

1 Create a default map axes in a sinusoidal projection, turn on the graticule, and display the `coast` data set as filled polygons:

```
figure;
axesm sinusoid
framem on; gridm on; tightmap tight
load coast
patchm(lat, long,'g')
```

The continents and graticule appear in normal aspect, as shown below.



Normal aspect: origin at (0°,0°), orientation 0°
(orientation vector = [0 0 0])

2 Inspect the orientation vector from the map axes:

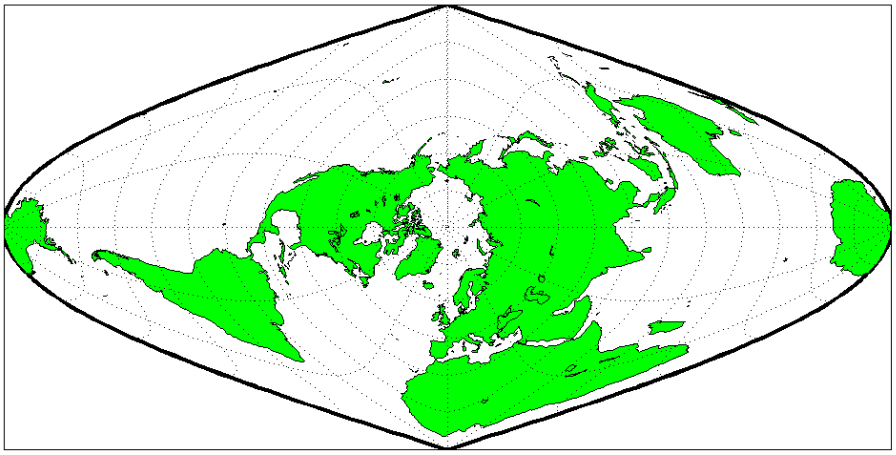
```
getm(gca,'Origin')

ans =
     0     0     0
```

By default, the origin is set at (0°E, 0°N), oriented 0° from vertical.

3 In the normal aspect, the North Pole is at the *top* of the image. To create a transverse aspect, imagine pulling the North Pole down to the center of the display, which was originally occupied by the point (0°,0°). Do this by setting the first element of `Origin` parameter to a latitude of 90°N:

```
setm(gca,'Origin',[90 0 0])
```

The shape of the frame is unaffected; this is still a sinusoidal projection.

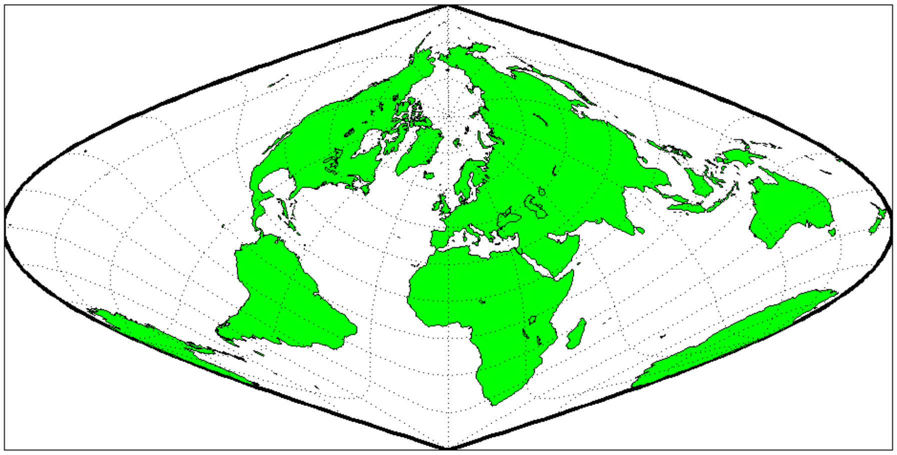


Transverse aspect: origin at (90°N,0°), orientation 0°
(orientation vector = [90 0 0])

4 The normal and transverse aspects can be thought of as limiting conditions. Anything else is an oblique aspect. Conceptually, if you push the North Pole halfway back to its original position (to the position originally occupied by the point (45ºN, 0ºE) in the normal aspect), the result is a simple oblique aspect.

```
setm(gca,'Origin',[45 0 0])
```

The oblique sinusoidal projection centered at (45ºN, 0ºE) is shown below.



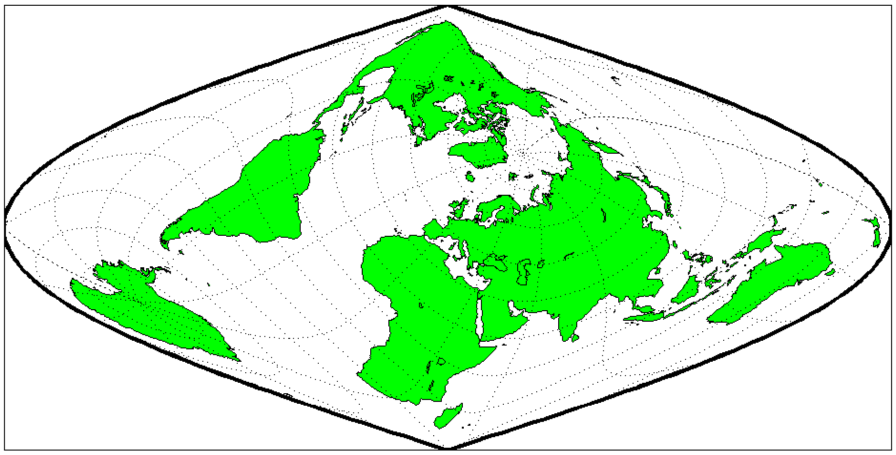
Oblique aspect: origin at (45°N,0°), orientation 0°
(orientation vector = [45 0 0])

You can think of this as pulling the new origin (45ºN, 0º) to the center of the image, the place that (0º,0º) occupied in the normal aspect.

5 The previous examples of projection aspect kept the aspect orientation at 0º. If the orientation is altered, an oblique aspect becomes a *skew-oblique*. Imagine the previous example with an orientation of 45º. Think of this as pulling the new origin (45ºN,0ºE), down to the center of the projection and then rotating the projection until the North Pole lies at an angle of 45º clockwise from straight up with respect to the new origin.

```
setm(gca,'Origin',[45 0 45])
```

As in the previous example, the location (45ºN,0ºE) still occupies the center of the map.



Skew-oblique aspect: origin at (45°N,0°), orientation 45°
(orientation vector = [45 0 45])

Any projection can be viewed in alternate aspects. Some of these are quite useful. For example, the transverse aspect of the Mercator projection is widely used in cartography, especially for mapping regions with predominantly north-south extent. One candidate for such handling might be Chile. Oblique Mercator projections might be used to map long regions that run neither north and south nor east and west, such as New Zealand.

Note The projection aspect discussed in this section is different from the map axes `Aspect` property. The map axes `Aspect` property controls the orientation of the figure axes. For instance, if a map is in a `normal` setting with a *landscape* orientation, a switch to a `transverse` aspect rotates the axes by 90º, resulting in a *portrait* orientation. To display a map in the transverse aspect, combine the `transverse` aspect property with a -90º skew angle. The skew angle is the last element of the `Origin` parameter. For example, a `[0 0 -90]` vector would produce a transverse map.

The base projection can be thought of as a standard coordinate system, and the normal aspect conforms to it. The features of a projection are maintained in any aspect, *relative to the base projection*. As the preceding illustrations show, the *outline* (frame) does not change. Nondirectional projection characteristics also do not change. For example, the sinusoidal projection is equal-area, no matter what its aspect. Directional characteristics must be considered carefully, however. In the normal aspect of the sinusoidal projection, scale is true along every parallel and the central meridian. This is not the case for the skew-oblique aspect; however, scale is true along the paths of the transformed parallels and meridian.

Projection Parameters

Every projection has at least one parameter that controls how it transforms geographic coordinates into planar coordinates. Some projections are rather fixed, and aside from the orientation vector and nominal scale factor, have no parameters that the user should vary, as to do so would violate the definition of the projection. For example, the Robinson projection has one standard parallel that is fixed by definition at 38º North and South; the Cassini and Wetch projections cannot be constructed in other than Normal aspect. In general, however, projections have several variable parameters. The following section discusses map projection parameters and provides guidance for setting them.

Projection Characteristics Maps Can Have

In addition to the name of the projection itself, the parameters that a map projection can have are

- *Aspect* — Orientation of the projection on the display surface
- *Center* or *Origin* — Latitude and longitude of the midpoint of the display
- *Scale Factor* — Ratio of distance on the map to distance on the ground
- *Standard Parallel(s)* — Chosen latitude(s) where scale distortion is zero
- *False Northing* — Planar offset for coordinates on the vertical map axis
- *False Easting* — Planar offset for coordinates on the horizontal map axis
- *Zone* — Designated latitude-longitude quadrangle used to systematically partition the planet for certain classes of projections

While not all projections require all these parameters, there will always be a projection aspect, origin, and scale.

Other parameters are associated with the graphic expression of a projection, but do not define its mathematical outcome. These include

- Map latitude and longitude limits
- Frame latitude and longitude limits

However, as certain projections are unable to map an entire planet, or become very distorted over large regions, these limits are sometimes a necessary part of setting up a projection.

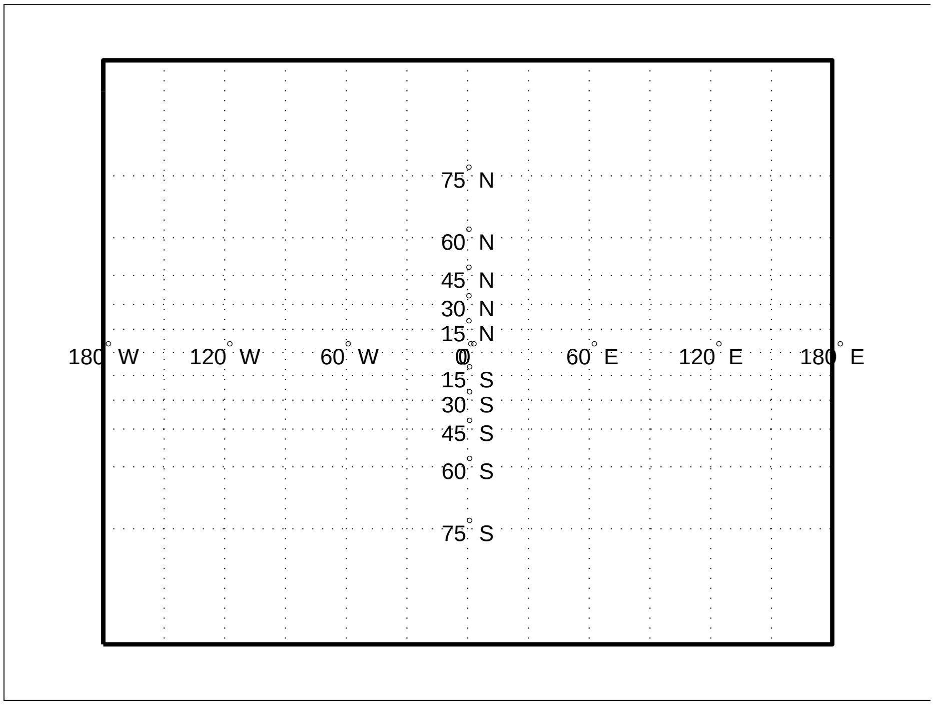
Determining Projection Parameters

In the following exercise, you define a map axes and examine default parameters for a cylindrical, a conic, and an azimuthal projection.

1 Set up a default Mercator projection (which is cylindrical) and pass its handle to the `getm` function to query projection parameters:

```
figure;
h=axesm('Mapprojection','mercator','Grid','on','Frame','on',...
'MlabelParallel',0,'PlabelMeridian',0,'mlabellocation',60,...
'meridianlabel','on','parallellabel','on')
```

The graticule and frame for the default map projection are shown below.



2 Query the map axes handle using `getm` to inspect the properties that pertain to map projection parameters. The principal ones are `aspect`, `origin`, `scalefactor`, `nparallels`, `mapparallels`, `falsenorthing`, `falseeasting`, `zone`, `maplatlimit`, `maplonlimit`, `rlatlimit`, and `flonlimit`:

```
getm(h,'aspect')

ans =
     normal

getm(h,'origin')

ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     1

getm(h,'mapparallels')

ans =
     0

getm(h,'falsenorthing')

ans =
     0

getm(h,'falseeasting')

ans =
     0
```

```
getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -86    86

getm(h,'maplonlimit')

ans =
  -180   180

getm(h,'Flatlimit')

ans =
   -86    86

getm(h,'Flonlimit')

ans =
  -180   180
```

For more information on these and other map axes properties, see the reference page for `axesm`.

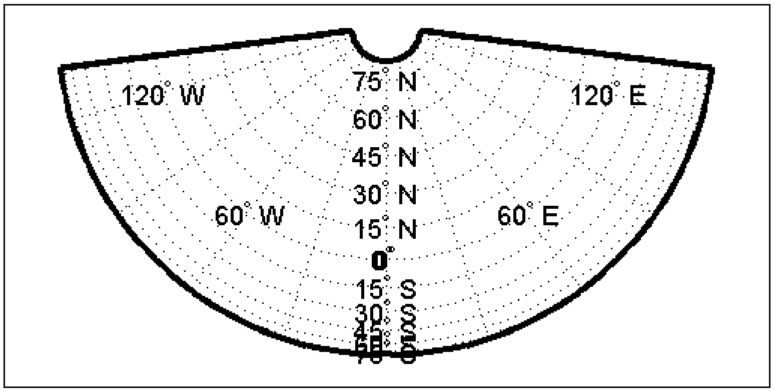
3 Reset the projection type to equal-area conic (`'eqaconic'`). The figure is redrawn to reflect the change. Determine the parameters that the toolbox changes in response:

```
setm(h,'Mapprojection', 'eqaconic')
getm(h,'aspect')

ans =
normal

getm(h,'origin')
```

```
ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     2

getm(h,'mapparallels')

ans =
    15    75

getm(h,'falsenorthing')

ans =
     0

getm(h,'falseeasting')

ans =
     0

getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -86    86

getm(h,'maplonlimit')
```

```
ans =
  -135   135

getm(h,'Flatlimit')

ans =
   -86    86

getm(h,'Flonlimit')

ans =
  -135   135
```

The eqaconic projection has two standard parallels, at 15º and 75º. It also has reduced longitude limits (covering 270º rather than 360º). The resulting eqaconic graticule is shown below.



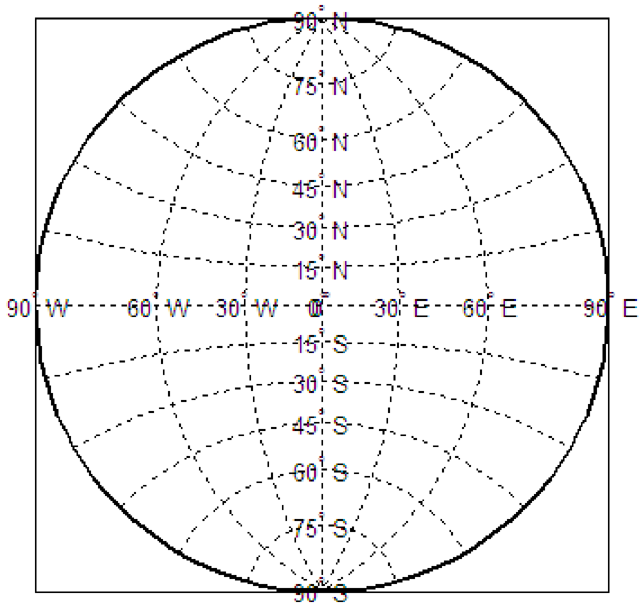
4 Now set the projection type to Stereographic ('stereo') and examine the same properties as you did for the previous projections:

```
setm(h,'Mapprojection','stereo')
setm(gca,'MLabelParallel',0,'PLabelMeridian',0)
getm(h,'aspect')

ans =
normal
```

```
getm(h,'origin')

ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     0

getm(h,'mapparallels')

ans =
     []

getm(h,'falsenorthing')

ans =
     0

getm(h,'falseeasting')

ans =
     0

getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -90    90
```

```
getm(h,'maplonlimit')

ans =
  -180   180

getm(h,'Flatlimit')

ans =
   -Inf    90

getm(h,'Flonlimit')

ans =
  -180   180
```

The stereographic projection, being azimuthal, does not have standard parallels, so none are indicated. The map limits do not change from the previous projection. The map figure is shown below.



Chapter 14, “Map Projections Reference” (available online and in the PDF version of this document) lists and illustrates all supported Mapping Toolbox map projections, including suggestions for parameter usage.

Visualizing and Quantifying Projection Distortions

In this section...

Displays of Spatial Error in Maps

Because no projection can preserve all directional and nondirectional geographic characteristics, it is useful to be able to estimate the degree of error in direction, area, and scale for a particular projection type and parameters used. Several Mapping Toolbox functions display projection distortions, and one computes distortion metrics for specified locations.

A standard method of visualizing the distortions introduced by the map projection is to display small circles at regular intervals across the globe. After projection, the small circles appear as ellipses of various sizes, elongations, and orientations. The sizes and shapes of the ellipses reflect the projection distortions. Conformal projections have circular ellipses, while equal-area projections have ellipses of the same area. This method was invented by Nicolas Tissot in the 19th century, and the ellipses are called *Tissot indicatrices* in his honor. The measure is a tensor function of location that varies from place to place, and reflects the fact that, unless a map is conformal, map scale is different in every direction at a location.

Visualizing Projection Distortions via Tissot Indicatrices

As the following example illustrates, you can add the indicatrices to a map display with the command `tissot` and remove them with `clmo tissot`:

1 Set up a Sinusoidal projection in a skewed aspect, plotting the graticule:

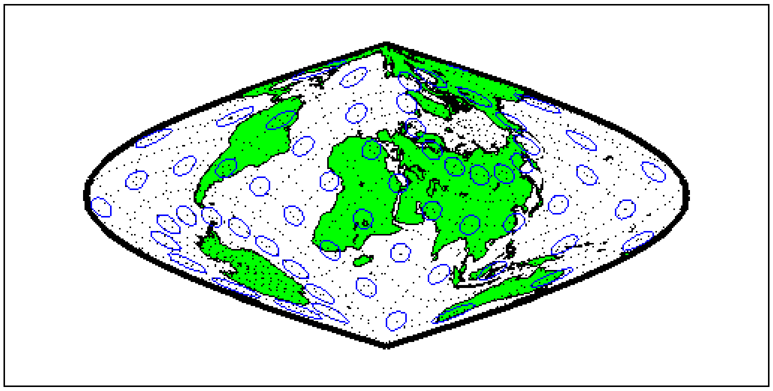
```
figure;
axesm sinusoid
gridm on;framem on;
setm(gca,'Origin', [20 30 45])
```

2 Load the `coast` data set and plot it as green patches:

```
load coast
patchm(lat,long,'g')
```

3 Plot the default Tissot diagram, shown below:

```
tissot
```



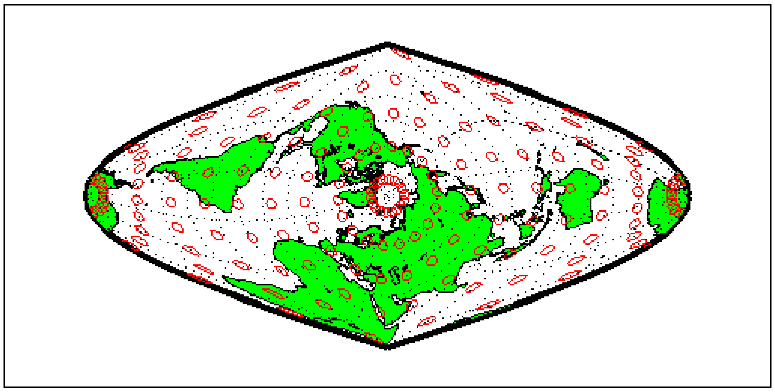
Notice that the circles vary considerably in shape. This indicates that the Sinusoidal projection is not conformal. Despite the distortions, however, the circles all cover equal amounts of area on the map, because the projection has the equal-area property.

Default Tissot diagrams are drawn with blue unfilled 100-point circles spaced 30 degrees apart in both directions. The default circle radius is 1/10 of the current radius of the reference ellipsoid (by default that radius is 1).

4 Now clear the Tissot diagram, rotate the projection to a polar aspect, and plot a new Tissot diagram using circles paced 20 degrees apart, half as big as before, drawn with 20 points, and drawn in red:

```
clmo tissot
setm(gca,'Origin', [90 0 45])
tissot([20 20 .05 20],'Color','r')
```

The result is shown below. Note that circles are drawn faster because fewer points are computed for each one. Also note that the distortions are still smallest close to the map origin, and still greatest near the map frame.



Try changing the map projection to a conformal one such as Mercator or Stereographic to see what Tissot indicatrices look like on shape-preserving maps.

For further information, see the reference page for `tissot`.

Visualizing Projection Distortions via Isolines

Most map projection distortions are rather orderly and vary continuously, making them suitable for display via isolines (contour lines). In addition to Tissot diagrams, the toolbox can plot isolines of variations of several parameters associated with map projections, using `mdistort`.

The `mdistort` function can plot variations in angles, areas, maximum and minimum scale, and scale along parallels and meridians, in units of percent deviation (except for angles, for which degrees are used). Use this function in selecting projections and projection parameters when you are concerned about keeping specific types of distortion within limits. Below are some examples of `mdistort` using the Hammer modified azimuthal projections and the Bonne pseudoconic projection.

1 Create a Hammer projection map axes in normal aspect, and plot a graticule, frame, and coastlines on it:

```
figure;
axesm('MapProjection','hammer','Grid','on','Frame','on')
```

2 Load the `coast` data set and plot it as green patches:

```
load coast
patchm(lat,long,'g')
```

3 Call `mdistort` to plot contours of minimum-to-maximum scale ratios:

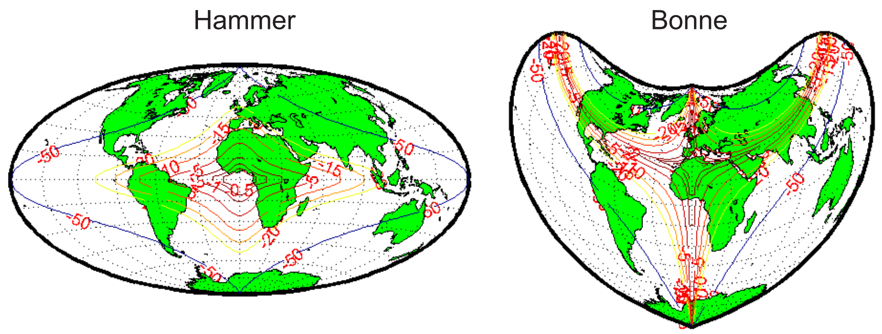
```
mdistort('scaleratio')
```

Notice that the region of minimum distortion is centered around (`0,0`).

4 Repeat this diagram with a Bonne projection in a new figure window:

```
figure;
axesm('MapProjection','bonne','Grid','on','Frame','on')
patchm(lat,long,'g')
mdistort('scaleratio')
```

Notice that the region of minimum distortion is centered around (`30,0`), which is where the single standard parallel is.



Isolines of maximum/minimum scale ratio

5 You can toggle the isolines by typing `mdistort` or `mdistort off`. Look at some other types of distortion. The types you can request are

- `area` — Percent departures from equal area
- `angles` — Angular distortion of right angles
- `scale` or `maxscale` — Percent of maximum scale

- `minscale` — Percent of minimum scale
- `parscale` — Percent of scale along the parallels
- `merscale` — Percent of scale along the meridians
- `scaleratio` — Percent of maximum-to-minimum scale ratio

For further information see the reference page for `mdistort`.

Quantifying Map Distortions at Point Locations

The `tissot` and `mdistort` functions described above provide synoptic visual overviews of different forms of map projection error. Sometimes, however, you need numerical estimates of error at specific locations in order to quantify or correct for map distortions. This is useful, for example, if you are sampling environmental data on a uniform basis across a map, and want to know precisely how much area is associated with each sample point, a statistic that will vary by location and be projection dependent. Once you have this information, you can adjust environmental density and other statistics you collect for areal variations induced by the map projection.

A Mapping Toolbox function returns location-specific map error statistics from the current projection or an mstruct. The `distortcalc` function computes the same distortion statistics as `mdistort` does, but for specified locations provided as arguments. You provide the latitude-longitude locations one at a time or in vectors. The general form is

```
[areascale,angdef,maxscale,minscale,merscale,parscale] = ...
    distortcalc(mstruct,lat,long)
```

However, if you are evaluating the current map figure, omit the mstruct. You need not specify any return values following the last one of interest to you.

Using distortcalc to Determine Map Projection Geometric Distortions

The following exercise uses `distortcalc` to compute the maximum area distortion for a map of Argentina from the landareas data set.

1 Read the North and South America polygon:

```
Americas = shaperead('landareas','UseGeoCoords',true, ...
    'Selector', {@(name) ...
    strcmpi(name,{'north and south america'}),'Name'});
```

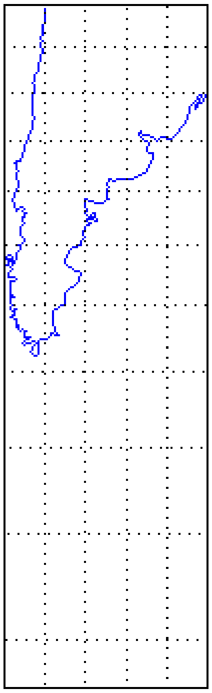
2 Set the spatial extent (map limits) to contain the southern part of South America and also include an area closer to the South Pole:

```
mlatlim = [-72.0 -20.0];
mlonlim = [-75.0 -50.0];
[alat, alon] = maptriml([Americas.Lat], ...
    [Americas.Lon], mlatlim, mlonlim);
```

3 Create a Mercator cylindrical conformal projection using these limits, specify a five-degree graticule, and then plot the outline for reference:

```
figure;
axesm('MapProjection','mercator','grid','on', ...
    'MapLatLimit',mlatlim,'MapLonLimit',mlonlim,...
    'MLineLocation',5, 'PLineLocation',5)
plotm(alat,alon,'b')
```

The map looks like this:



4 Sample every tenth point of the patch outline for analysis:

```
alats = alat(1:10:numel(alat));
alons = alon(1:10:numel(alat));
```

5 Compute the area distortions (the first value returned by distortcalc) at the sample points:

```
adistort = distortcalc(alats, alons);
```

6 Find the range of area distortion across Argentina (percent of a unit area on, in this case, the equator):

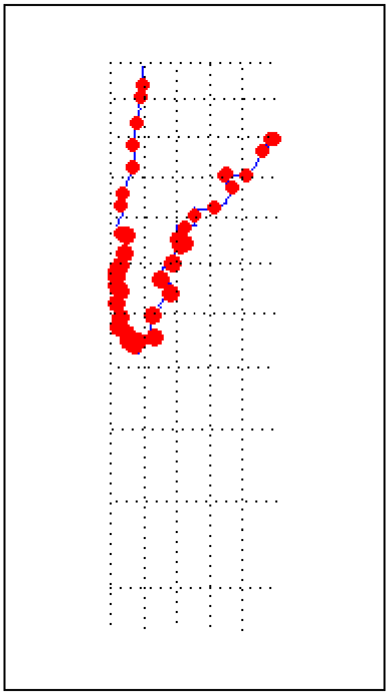
```
adistortmm = [min(adistort) max(adistort)]

adistortmm =
    1.1790    2.7716
```

As Argentina occupies mid southern latitudes, its area on a Mercator map is overstated, and the errors vary noticeably from north to south.

7 Remove any NaNs from the coordinate arrays and plot symbols to represent the relative distortions as proportional circles, using `scatterm`:

```
nanIndex = isnan(adistort);
alats(nanIndex) = [];
alons(nanIndex) = [];
adistort(nanIndex)  = [];
scatterm(alats,alons,20*adistort,'red','filled')
```

The resulting map is shown below:



8 The degree of area overstatement would be considerably larger if it extended farther toward the pole. To see how much larger, get the area distortion for 50°S, 60°S, and 70°S:

```
a=distortcalc(-50,-60)

a =
       2.4203

a=distortcalc(-60,-60)

a =
            4

>> a=distortcalc(-70,-60)

a =
       8.5485
```

Note You can only use `distortcalc` to query locations that are within the current map frame or mstruct limits. Outside points yield `NaN` as a result.

9 Using this technique, you can write a simple script that lets you query a map repeatedly to determine distortion at any desired location. You can select locations with the graphic cursor using `inputm`. For example,

```
[plat plon] = inputm(1)

plat =
      -62.225
plon =
      -72.301
>> a=distortcalc(plat,plon)

a =
       4.6048
```

Naturally the answer you get will vary depending on what point you pick. Using this technique, you can write a simple script that lets you query a map repeatedly to determine any distortion statistic at any desired location.

Try changing the map projection or even the orientation vector to see how the choice of projection affects map distortion. For further information, see the reference page for `distortcalc`.

Accessing, Computing, and Inverting Map Projection Data

In this section...

Accessing Projected Coordinate Data

Most of the examples in this document assume that the end product of a map projection is a graphical representation as a map, and that the planar coordinates yielded by projection are of little interest. However, there might be times when you need access to projected coordinate data. You might also have projected data that you want to transform back to latitude and longitude (assuming you know its projection parameters). The following sections describe how to retrieve projected data, project it without displaying it, and invert projections.

A MATLAB figure generally contains coordinate data only in its axes child object and in children of axes objects, such as line, patch, and surface objects. See the reference page for `axes` for an overview of this object hierarchy. Note that a map axes can have multiple patch children objects when created with `patchesm`.

You can retrieve projected data from a map axes, but you can also obtain it without having to plot the data or even creating a map axes. The following two exercises illustrate each of these approaches.

Retrieving Projected Coordinates from a Figure

An easy way to retrieve the projected coordinates of a map occupying a figure window is with the MATLAB `get` command. The projected coordinates are stored in the object's `XData` and `YData` properties. The `XData` and `YData` can belong to a child object rather than to the axes themselves, however, as the following exercise demonstrates.

1 Create a Mollweide projection map axes and obtain its handle:

```
figure;
ha = axesm('mollweid')
```

2 Observe that the axes has no `XData`, `YData`, or `children` information:

```
get(ha,'XData')

??? Error using ==> get
Invalid axes property: 'XData'.

get(ha,'YData')

??? Error using ==> get
Invalid axes property: 'YData'.

get(ha,'children')

ans =
   Empty matrix: 0-by-1
```

3 Display a map frame for the Mollweide projection, obtaining its handle. Confirm that the frame is a child of the axes:

```
hf = framem

hf =
          105

get(ha,'children')

ans =
          105
```

4 Use `get` to extract the *x-y* coordinates of the map frame:

```
xf = get(hf,'XData');
yf = get(hf,'YData');
```

The `xf` and `yf` coordinates are 398-by-1 column vector arrays.

5 Load the `coast` data set and render it with `plotm`, obtaining a handle:

```
load coast
hl = plotm(lat,long)

hl =
         106

get(ha, 'children')

ans =
         106
         105
```

Note that the line data is also a child of the axes.

6. Retrieve the projected coastline coordinates using handle `hl`:

   ```
   xline = get(hl,'XData');
   yline = get(hl,'YData');
   ```

   The `xline` and `yline` coordinates are 1-by-9591 row vector arrays. Inspect their contents before proceeding.

7. The units for projected coordinates are established by the ellipsoid vector. By default, these units are Earth radii, but you can change them at any time using `setm` to control the `geoid` property. For example, set the units to kilometers on a spherical earth with

   ```
   setm(gca,'Geoid', almanac('earth','sphere','kilometers'))
   ```

   Repeat step 6 above to see how this affects coordinate values. See "The Ellipsoid Vector" on page 3-4 for further information on specifying coordinate units and ellipsoids.

Projecting Coordinates Without a Map Axes

You do not need to display a map object to obtain its projected coordinates. You can perform the same projection computations that are done within Mapping Toolbox display commands by calling the `defaultm` and `mfwdtran` functions.

Using mfwdtran with a Map Projection Structure

Before projecting the data, you must define projection parameters, just as you would prepare a map axes with `axesm` before displaying a map. The projection parameters are stored in a map projection structure that is stored within a map axes object, but you can directly create and use such a structure for projection computations without involving a map axes or a graphical display.

1 Begin by using `defaultm` to create an empty map projection structure for a Sinusoidal projection.

```
mstruct = defaultm('sinusoid');
```

The structure mstruct appears in the workspace. Use the property editor to view its fields and contents.

2 Set the map limits for the mstruct. You must invoke `defaultm` a second time to fully populate the fields of the map projection structure and to ensure that the effects of property settings are properly implemented.

```
mstruct.maplonlimit = [-150 -30];
mstruct.geoid = almanac('earth','grs80','kilometers');
mstruct = defaultm(mstruct);
```

3 Note that the origin longitude is centered between the longitude limits.

```
mstruct.origin
```

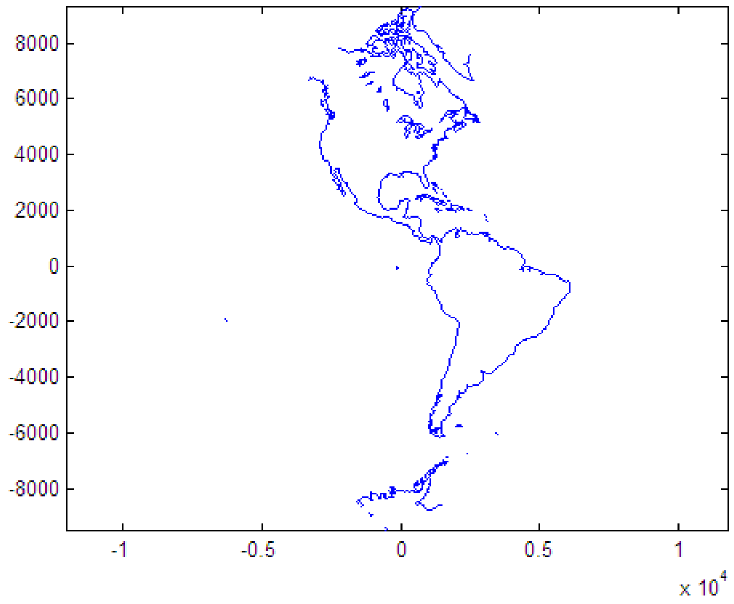
4 Trim the coast to the map limits set above.

```
load coast
[latt,lont] = maptriml(lat,long, ...
     mstruct.maplatlimit,mstruct.maplonlimit);
```

5 Having defined the map projection parameters, project the latitude and longitude vectors into plane coordinates with the Sinusoidal projection and display the result using nonmapping MATLAB graphic commands.

```
[x,y] = mfwdtran(mstruct,latt,lont);
figure
plot(x,y)
axis equal
```

The plot shows that resulting data are projected in the specified aspect.



For additional information, see the reference pages for `defaultm` and `mfwdtran`. It is also possible to reverse the process using `minvtran`, as the next section, "Inverse Map Projection" on page 8-41, describes. You may also use `projfwd` and `projinv`, which are newer Mapping Toolbox functions that use the `PROJ.4` map projection library to do forward and inverse projections, respectively. See the references pages for `projfwd` and `projinv` for details.

Inverse Map Projection

The process of obtaining latitudes and longitudes from geodata with planar coordinates is called *inverse projection*. Most, but not all, map projections have inverses. Mapping Toolbox function `minvtran` transforms plane coordinates into geodetic coordinates; it is a mirror image of `mfwdtran`, which is described in "Using mfwdtran with a Map Projection Structure" on page 8-40. Like its twin, `minvtran` operates on a geographic data structure that you can explicitly create. If the coordinate data originates from an external source or vendor, you need to know its correct projection parameters in order for inverse projection to be successful.

Recovering Geodetic Coordinates with minvtran

In the following exercise exploring the use of `minvtran`, you again work with the `coast` data set, using the projected coordinates created in the previous exercise, "Using mfwdtran with a Map Projection Structure" on page 8-40.

1. If you do not have the results of the previous exercise in the workspace, perform it now and go on to step 2. You have the following variables:

   ```
   Name          Size                        Bytes  Class

     lat       9589x1                        76712  double array
     long      9589x1                        76712  double array
     mstruct      1x1                         7360  struct array
     x         9599x1                        76792  double array
     y         9599x1                        76792  double array

   Grand total is 38563 elements using 314368 bytes
   ```

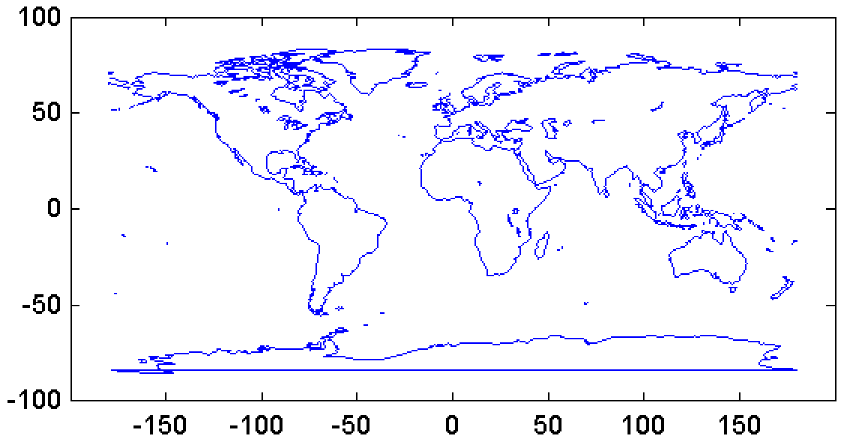
   The difference in size between `lat` and `long` and `x` and `y` are due to clipping the *x*-*y* data to the map frame (`NaN`s are inserted at clip locations).

2. Transform the projected *x*-*y* data back into geographic coordinates with the inverse transformation function:

   ```
   [lat2,long2] = minvtran(mstruct,x,y);
   ```

3. In a new figure, plot the resulting latitudes and longitudes as if they were plane coordinates, and set the frame larger than default:

   ```
   figure; plot(long2,lat2); axis equal
   set(gca,'XLim',[-200 200],'YLim',[-100 100])
   ```



Notice the wraparound in Antarctica. This occurred because its coastline crosses the International Date Line. In the projection transformation process, longitude data outside `[-180 180]` degrees is projected back into this range because angles differing by 360º are geographically equivalent. The data from the inverse transformation process therefore jumps from 180º to -180º, as depicted by the horizontal lines in the figure above.

Obtaining Angular Directions in a Projection Space

In addition to projecting geographic positions into Cartesian coordinates, you can project angles between the sphere and the plane. For cylindrical projections in normal aspect, north maps to up on the *y*-axis, and east maps to right on the *x*-axis. This is not necessarily true of other projection types. In the normal aspect of conic projections, for example, north may skew to the left or right of vertical, depending on longitude. The `vfwdtran` function, which takes latitudes, longitudes, and azimuths, computes angles that geographic vectors make on the projection plane.

To illustrate, define vectors pointing north (0º) and east (90º) at three locations and use `vfwdtran` to compute the angles of north and east in projected coordinates on an equidistant conic projection.

Note Geographic angles are measured clockwise from north, while projected angles are measured counterclockwise from the *x*-axis.

1 Set up an equidistant conic projection for the northern hemisphere:

```
figure;
axesm('eqdconic','MapLatLimit',[-10 45],'MapLonLimit',[-55 55])
gridm; framem; mlabel; plabel; tightmap
```

2 Define three locations along the equator:

```
lats = [0 0 0];
lons = [-45 0 45];
```

3 Define north and east azimuths for each point:

```
northazs = [0 0 0];
eastazs = [90 90 90];
```

4 Compute the projected direction of north for each location:

```
pnorth = vfwdtran(lats,lons,northazs)

ans =
       59.614            90       120.39
```

North varies from about 60º from the *x*-axis, to vertical, to 120º from the *x*-axis, quite symmetrically.

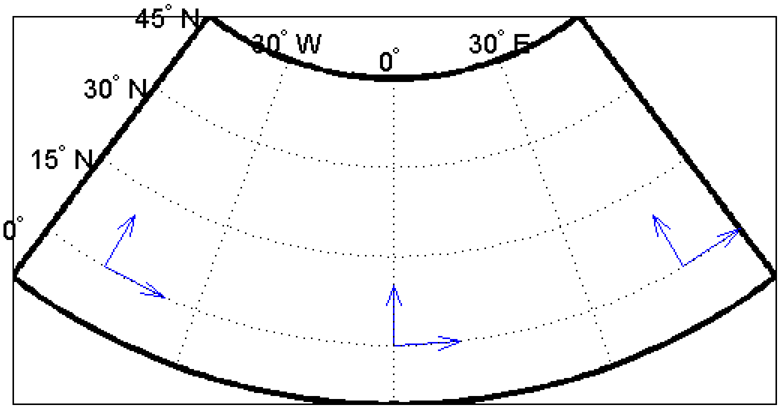
5 Compute projected direction of east for each location:

```
peast = vfwdtran(lats,lons,eastazs)

ans =
      -30.385    0.0001931       30.386

pnorth - peast

ans =
           90            90           90
```

The projected east vectors show a similar symmetry, and as expected form complementary angles to north.

6 Use `quiverm` to plot the six vectors on the projection; note their plane angles:

```
quiverm(lats, lons, [0 0 0], [10 10 10], 0)
quiverm(lats, lons, [10 10 10], [0 0 0], 0)
```



For more information, see the reference pages for `vfwdtran` and `quiverm`.

Coordinate Transformations

In “The Orientation Vector” on page 8-10, you explored the concept of altering the aspect of a map projection in terms of pushing the North Pole to new locations. Another way to think about this is to redefine the coordinate system, and then to compute a normal aspect projection based on the new system. For example, you might redefine a spherical coordinate system so that your home town occupies the origin. If you calculated a map projection in a normal aspect with respect to this *transformed* coordinate system, the resulting display would look like an oblique aspect of the *true* coordinate system of latitudes and longitudes.

This transformation of coordinate systems can be useful independent of map displays. If you transform the coordinate system so that your home town is the new *North Pole*, then the transformed coordinates of all other points will provide interesting information.

Note The types of coordinate transformations described here are appropriate for the spherical case only. Attempts to perform them on an ellipsoid will produce incorrect answers on the order of several to tens of meters.

When you place your home town at a pole, the spherical distance of each point from your hometown becomes 90º minus its transformed latitude (also known as a *colatitude*). The point antipodal to your town would become the *South Pole*, at -90º. Its distance from your hometown is 90º-(-90º), or 180º, as expected. Points 90º distant from your hometown all have a transformed latitude of 0º, and thus make up the transformed *equator*. Transformed longitudes correspond to their respective great circle azimuths from your home town.

Reorienting Vector Data with rotatem

The `rotatem` function uses an orientation vector to transform latitudes and longitudes into a new coordinate system. The orientation vector can be produced by the `newpole` or `putpole` functions, or can be specified manually.

As an example of transforming a coordinate system, suppose you live in Midland, Texas, at (32ºN,102ºW). You have a brother in Tulsa (36.2ºN,96ºW) and a sister in New Orleans (30ºN,90ºW).

1 Define the three locations:

```
midl_lat = 32;   midl_lon = -102;
tuls_lat = 36.2; tuls_lon = -96;
newo_lat = 30;   newo_lon = -90;
```

2 Use the `distance` function to determine great circle distances and azimuths of Tulsa and New Orleans from Midland:

```
[dist2tuls az2tuls] = distance(midl_lat,midl_lon,...
                                    tuls_lat,tuls_lon)

dist2tuls =
   6.5032

az2tuls =
  48.1386

[dist2neworl az2neworl] = distance(midl_lat,midl_lon,...
                                   newo_lat,newo_lon)

dist2neworl =
  10.4727

az2neworl =
  97.8644
```

Tulsa is about 6.5 degrees distant, New Orleans about 10.5 degrees distant.

3 Compute the absolute difference in azimuth, a fact you will use later.

```
azdif = abs(az2tuls-az2neworl)

azdif =
   49.7258
```

4 Today, you feel on top of the world, so make Midland, Texas, the *north pole* of a transformed coordinate system. To do this, first determine the origin required to put Midland at the pole using `newpole`:

```
origin = newpole(midl_lat,midl_lon)

origin =
 58    78     0
```

The origin of the new coordinate system is (58ºN, 78ºE). Midland is now at a *new latitude* of 90º.

5 Determine the transformed coordinates of Tulsa and New Orleans using the `rotatem` command. Because its units default to radians, be sure to include the `degrees` keyword:

```
[tuls_lat1,tuls_lon1] = rotatem(tuls_lat,tuls_lon,...
                                origin,'forward','degrees')
```

```
tuls_lat1 =
 83.4968
tuls_lon1 =
 -48.1386

[newo_lat1,newo_lon1] = rotatem(newo_lat,newo_lon,...
                                origin,'forward','degrees')

newo_lat1 =
 79.5273
newo_lon1 =
 -97.8644
```

6 Show that the new colatitudes of Tulsa and New Orleans equal their distances from Midland computed in step 2 above:

```
tuls_colat1 = 90-tuls_lat1

tuls_colat1 =
    6.5032

newo_colat1 = 90-newo_lat1

newo_colat1 =
   10.4727
```

7 Recall from step 4 that the absolute difference in the azimuths of the two cities from Midland was 49.7258º. Verify that this equals the difference in their new longitudes:

```
tuls_lon1-newo_lon1

ans =
   49.7258
```

You might note small numerical differences in the results (on the order of 10^{-6}), due to roundoff error and trigonometric functions.

For further information, see the reference pages for `rotatem`, `newpole`, `putpole`, `neworig`, and `org2pol`.

Reorienting Gridded Data with neworig

You can transform coordinate systems of data grids as well as vector data. When regular data grids are manipulated in this manner, distance and azimuth calculations with the map variable become row and column operations.

It is easy to transform a regular data grid to create a new one with its data rearranged to correspond to a new coordinate system using the `neworig` function. To demonstrate this, do the following:

1 Load the `topo` data set and transform it to a new coordinate system in which a point in Sri Lanka (7ºN, 80ºE) is the *north pole*:

```
figure;
load topo
origin = newpole(7,80)

origin =
   83.0000 -100.0000 0
```

2 Reorient the data grid with `neworig`, using this orientation vector:

```
[Z,lat,lon] = neworig(topo,topolegend,origin);
```

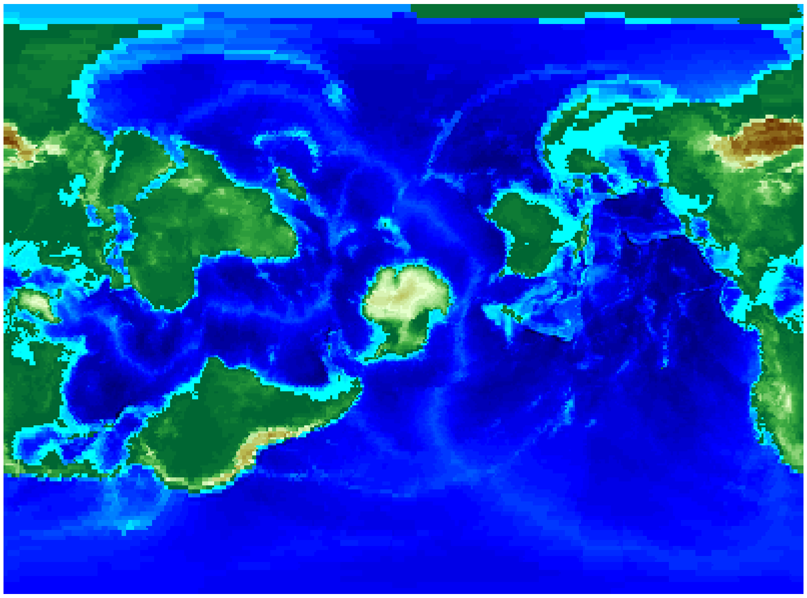
Note that the result, `[Z,lat,lon]`, is a *geolocated data grid*, not a regular data grid like the original `topo` data.

3 Display the new map:

```
axesm miller
latlim = [ -90  90];
lonlim = [-180 180];
gratsize = [90 180];
[lat,lon] = meshgrat(latlim,lonlim,gratsize);
surfm(lat,lon,Z);
demcmap(topo)
mstruct = getm(gca);
mstruct.origin
```

4 This map is displayed in normal aspect, as its orientation vector shows:

```
mstruct = getm(gca);
mstruct.origin

ans =
     0     0     0
```



An interesting feature of this new grid is that every cell in its first row is 0°–1° distant from the point (7°N, 80°E), and every cell in its second row is 1°–2° distant, etc. Another feature is that every cell in a particular column has the same great circle azimuth from the new origin.

Working with the UTM System

In this section...

What Is the Universal Transverse Mercator System?

So far, this chapter has described types and parameters of specific projections, treating each in isolation. The following sections discuss how the Transverse Mercator and Polar Stereographic projections are used to organize a worldwide coordinate grid. This system of projections is generally called Universal Transverse Mercator (UTM). This system supports many military, scientific, and surveying applications.

The UTM system divides the world into a regular nonoverlapping grid of quadrangles, called *zones*, each 8 by 6 degrees in extent. Each zone uses formulas for a transverse version of the Mercator projection, with projection and ellipsoid parameters designed to limit distortion. The Transverse Mercator projection is defined between 80 degrees south and 84 degrees north. Beyond these limits, the Universal Polar Stereographic (UPS) projection applies.

The UPS has two zones only, `north` and `south`, which also have special projection and ellipsoid parameters.

In addition to the zone identifier—a grid reference in the form of a number followed by a letter (e.g., `31T`)—each UTM zone has a *false northing* and a *false easting*. These are offsets (in meters) that enable each zone to have positive coordinates in both directions. For UTM, they are constant, as follows:

- False easting (for every zone): 500,000 m
- False northing (all zones in the Northern Hemisphere): 0 m

- False northing (all zones in the Southern Hemisphere): 1,000,000 m

For UPS (in both the `north` and `south` zones), the false northing and false easting are both 2,000,000.

Understanding UTM Parameters

You can create UTM maps with `axesm`, just like any other projection. However, unlike other projections, the map frame is limited to an 8-by-6 degree map window (the UTM zone), as the following steps illustrate.

1. Create a UTM map axes:

   ```
   axesm utm
   ```

2. Get the map axes properties and inspect them in the Command Window or with the Variable Editor. The first few illustrate the projection defaults:

   ```
   h = getm(gca)
   mapprojection: 'utm'
              zone: '31N'
        angleunits: 'degrees'
            aspect: 'normal'
     falsenorthing: 0
      falseeasting: 500000
       fixedorient: []
             geoid: [6.3782e+006 0.082483]
       maplatlimit: [0 8]
       maplonlimit: [0 6]
      mapparallels: []
        nparallels: 0
            origin: [0 3 0]
       scalefactor: 0.9996
           trimlat: [-80 84]
           trimlon: [-180 180]
             frame: 'off'
             ffill: 100
        fedgecolor: [0 0 0]
        ffacecolor: 'none'
         flatlimit: [0 8]
        flinewidth: 2
   ```

```
             flonlimit: [-3 3]
                      ...
```

Note that the default `zone` is `31N`. This is selected because the map `origin` defaults to [0 3 0], which is on the equator and at a longitude of 3º E. This is the center longitude of zone `31N`, which has a latitude limit of [0 8], and a longitude limit of [0 6].

3 Move the `zone` one to the east, and inspect the other parameters again:

```
setm(gca,'zone','32n')
h = getm(gca)
mapprojection: 'utm'
              zone: '32N'
        angleunits: 'degrees'
            aspect: 'normal'
     falsenorthing: 0
      falseeasting: 500000
       fixedorient: []
             geoid: [6.3782e+006 0.082483]
       maplatlimit: [0 8]
       maplonlimit: [6 12]
      mapparallels: []
        nparallels: 0
            origin: [0 9 0]
       scalefactor: 0.9996
           trimlat: [-80 84]
           trimlon: [-180 180]
             frame: 'off'
             ffill: 100
        fedgecolor: [0 0 0]
        ffacecolor: 'none'
         flatlimit: [0 8]
        flinewidth: 2
         flonlimit: [-3 3]
                  ...
```

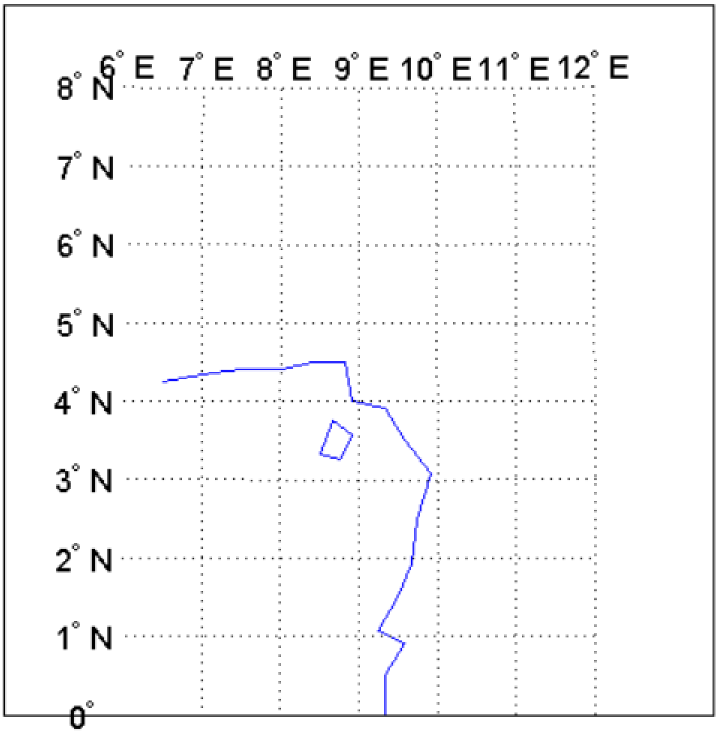
Note that the map origin and limits are adjusted for zone `32N`.

4 Draw the map grid and label it:

```
setm(gca,'grid','on','meridianlabel','on','parallellabel','on')
```

5 Load and plot the `coast` data set to see a close-up of the Gulf of Guinea and Bioko Island in UTM:

```
load coast
plotm(lat,long)
```



Setting UTM Parameters with a GUI

The easiest way to use the UTM projection is through a graphical user interface. You can create or modify a UTM area of interest with the `axesmui` projection control panel, and get further assistance form the `utmzoneui` control panel.

1. You can **Shift**+click in a map axes window, or type `axesmui` to display the projection control panel. Here you start from scratch:

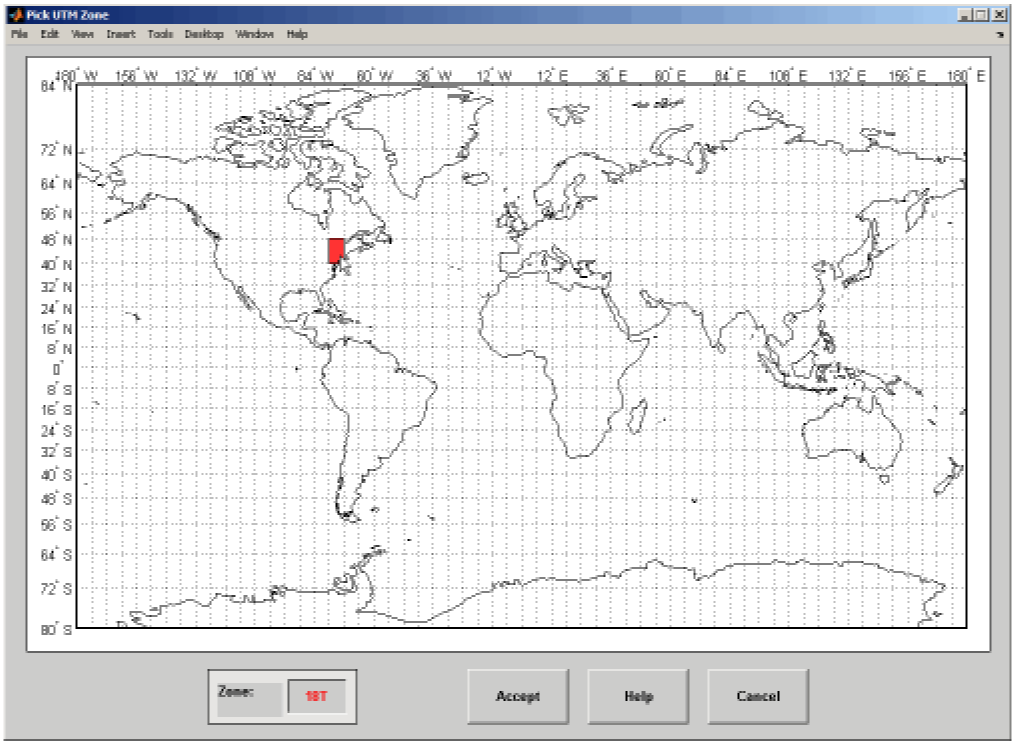
   ```
   figure;
   axesm utm
   axesmui
   ```

   The **Map Projection** field is set to `cyln:  Universal Transverse Mercator (UTM)`.

   Note For UTM and UPS maps, the **Aspect** field is set to `normal` and cannot be changed. If you attempt to specify `transverse`, an error results.

   ---

2. Click the **Zone** button to open the `utmzoneui` panel. Click the map near your area of interest to pick the zone:

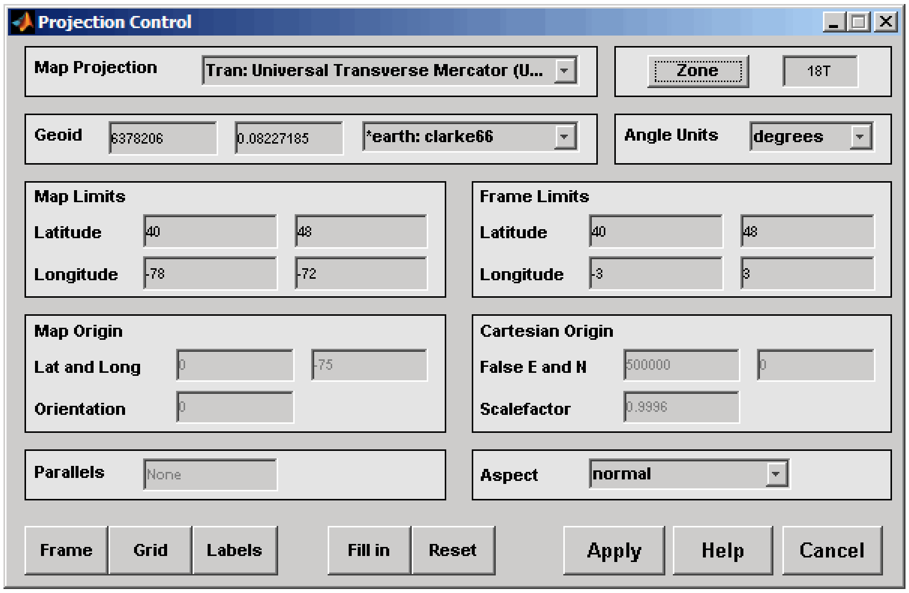


Note that while you can open the `utmzoneui` control panel from the command line, you then have to manually update the figure with the zone name it returns with a `setm` command:

```
setm(gca,'zone',ans)
```

3 Click the **Accept** button.

The `utmzoneui` panel closes, and the `zone` field is set to the one you picked. The map limits are updated accordingly, and the geoid parameters are automatically set to an appropriate ellipsoid definition for that zone. You can override the default choice by selecting another ellipsoid from the list or by typing the parameters in the **Geoid** field.



Map Projection: Tran: Universal Transverse Mercator (U...
Zone: 18T
Geoid: 6378206 | 0.08227185 | *earth: clarke66
Angle Units: degrees

Map Limits
Latitude: 40 | 48
Longitude: -78 | -72

Frame Limits
Latitude: 40 | 48
Longitude: -3 | 3

Map Origin
Lat and Long: 0 | -75
Orientation: 0

Cartesian Origin
False E and N: 500000 | 0
Scalefactor: 0.9996

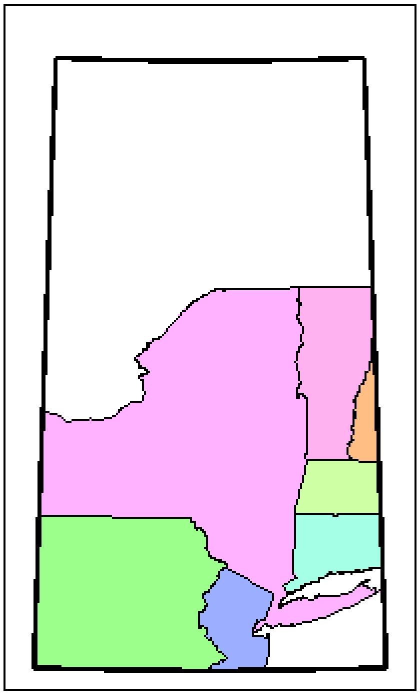
Parallels: None
Aspect: normal

4 Click **Apply** to close the projection control panel.

The projection is then ready for projection calculations or map display commands.

5 Now view a choropleth base map from the `usstatehi` demo shapefile for the area within the zone that you just selected:

```
states = shaperead('usastatehi', 'UseGeoCoords', true);
framem
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)],...
     'FaceColor', polcmap(numel(states))});
geoshow(states,'DisplayType', 'polygon',...
    'SymbolSpec', faceColors)
```



What you see depends on the zone you selected. The preceding display is for zone 18T, which contains portions of New England and the Middle Atlantic states.

You can also calculate projected UTM grid coordinates from latitudes and longitudes:

```
[latlim, lonlim] = utmzone('15S')

latlim =
    32    40
lonlim =
   -96   -90

[x,y] = mfwdtran(latlim, lonlim)

x =
 -1.5029e+006 -7.8288e+005
y =
  3.7403e+006  4.5369e+006
```

Working in UTM Without a Map Axes

You can set up UTM to calculate coordinates without generating a map display, using the `defaultm` function. The `utmzone` and `utmgeoid` functions help you select a zone and an appropriate ellipsoid. In the following exercise, you generate UTM coordinate data for a location in New York City, using that point to define the projection itself.

1 Define a location in New York City:

```
p1 = [40.7, -74.0];
```

2 Obtain the UTM zone for this point:

```
z1 = utmzone(p1)

z1 =
18T
```

3 Obtain the suggested ellipsoid vector and name for this zone:

```
[ellipsoid,estr] = utmgeoid(z1)

ellipsoid =
  6.3782e+006     0.082272
estr =
clarke66
```

4 Set up the UTM coordinate system based on this information:

```
utmstruct = defaultm('utm');
utmstruct.zone = '18T';
utmstruct.geoid = ellipsoid;
utmstruct = defaultm(utmstruct)
```

The empty latitude limits will be set properly by `defaultm`.

5 Now you can calculate the grid coordinates, without a map display:

```
    [x,y] = mfwdtran(utmstruct,p1(1),p1(2))

    x =
      5.8448e+005
    y =
      4.5057e+006
```

More on utmzone

You can also use the `utmzone` function to compute the zone limits for a given zone name. For example, using the preceding data, the latitude and longitude limits for zone 18T are

```
utmzone('18T')

ans =
    40    48   -78   -72
```

Therefore, you can call `utmzone` recursively to obtain the limits of the UTM zone within which a point location falls:

```
[zonelats zonelons] = utmzone(utmzone(40.7, -74.0))

zonelats =
    40    48
zonelons =
   -78   -72
```

For further information, see the reference pages for `utmzone`, `utmgeoid`, and `defaultm`.

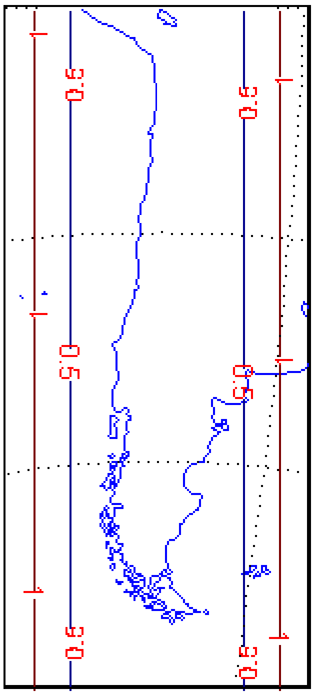
Mapping Across UTM Zones

Because UTM is a zone-based coordinate system, it is designed to be used like a map series, selecting from the appropriate sheet. While it is possible to extend one zone's coordinates into a neighboring zone's territory, this is not normally done.

To display areas that extend across more than one UTM zone, it might be appropriate to use the Mercator projection in a transverse aspect. Of course, you do not obtain coordinates in meters that would match those of a UTM projection, but the results will be nearly as accurate. Here is an example of a transverse Mercator projection appropriate to Chile. Note how the projection's line of zero distortion is aligned with the predominantly north-south axis of the country. The zero distortion line could be put exactly on the midline of the country by a better choice of the orientation vector's central meridian and orientation angle.

```
figure;
latlim = [-60 -15];centralMeridian = -70; width = 20;
axesm('mercator',...
    'Origin',[0 centralMeridian -90],...
    'Flatlimit',[-width/2 width/2],...
    'Flonlimit',sort(-latlim),...
    'Aspect','transverse')
land = shaperead('landareas.shp', 'UseGeoCoords', true);
geoshow([land.Lat], [land.Lon]);
framem
gridm; setm(gca,'plinefill',1000)
tightmap
mdistort scale
```

Note You might receive warnings about points from `landareas.shp` falling outside the valid projection region. You can ignore such warnings.


0.5
0.5
0.5
0.5
0.5
0.5
1
1
1
1
1
1

Summary and Guide to Projections

Cartographers often choose map projections by determining the types of distortion they want to minimize or eliminate. They can also determine which of the three projection types (cylindrical, conic, or azimuthal) best suits their purpose and region of interest. They can attach special importance to certain projection properties such as equal areas, straight rhumb lines or great circles, true direction, conformality, etc., further constricting the choice of a projection.

The toolbox has about 60 different built-in map projections. To list them all, type `maps`. The following table also summarizes them and identifies their properties. Notes for Special Features are located at the end of the table. Detailed information on all Mapping Toolbox map projections can be found in Chapter 14, "Map Projections Reference" (available online and in the PDF version of this document).

Projection	Syntax	Type	Equal--Area	Con-formal	Equi-distant	Special Features
Balthasart	`balthsrt`	Cylindrical	•			
Behrmann	`behrmann`	Cylindrical	•			
Bolshoi Sovietskii Atlas Mira	`bsam`	Cylindrical				
Braun Perspective	`braun`	Cylindrical				
Cassini	`cassini`	Cylindrical			•	
Central	`ccylin`	Cylindrical				
Equal-Area Cylindrical	`eqacylin`	Cylindrical	•			
Equidistant Cylindrical	`eqdcylin`	Cylindrical			•	
Gall Isographic	`giso`	Cylindrical			•	
Gall Orthographic	`gortho`	Cylindrical	•			
Gall Stereographic	`gstereo`	Cylindrical				
Lambert Equal-Area Cylindrical	`lambcyln`	Cylindrical	•			

Projection	Syntax	Type	Equal--Area	Con-formal	Equi-distant	Special Features
Mercator	`mercator`	Cylindrical		•		1
Miller	`miller`	Cylindrical				
Plate Carrée	`pcarree`	Cylindrical			•	
Trystan Edwards	`trystan`	Cylindrical	•			
Universal Transverse Mercator (UTM)	`utm`	Cylindrical		•		
Wetch	`wetch`	Cylindrical				
Apianus II	`apianus`	Pseudo-cylindrical				
Collignon	`collig`	Pseudo-cylindrical	•			
Craster Parabolic	`craster`	Pseudo-cylindrical	•			
Eckert I	`eckert1`	Pseudo-cylindrical				
Eckert II	`eckert2`	Pseudo-cylindrical	•			
Eckert III	`eckert3`	Pseudo-cylindrical				
Eckert IV	`eckert4`	Pseudo-cylindrical	•			
Eckert V	`eckert5`	Pseudo-cylindrical				
Eckert VI	`eckert6`	Pseudo-cylindrical	•			
Fournier	`fournier`	Pseudo-cylindrical	•			
Goode Homolosine	goode	Pseudo-cylindrical	•			

Projection	Syntax	Type	Equal--Area	Con-formal	Equi-distant	Special Features
Hatano Asymmetrical Equal-Area	`hatano`	Pseudo-cylindrical	•			
Kavraisky V	`kavrsky5`	Pseudo-cylindrical	•			
Kavraisky VI	`kavrsky6`	Pseudo-cylindrical	•			
Loximuthal	`loximuth`	Pseudo-cylindrical				
McBryde-Thomas Flat-Polar Parabolic	`flatplrp`	Pseudo-cylindrical	•			
McBryde-Thomas Flat-Polar Quartic	`flatplrq`	Pseudo-cylindrical	•			
McBryde-Thomas Flat-Polar Sinusoidal	`flatplrs`	Pseudo-cylindrical	•			
Mollweide	`mollweid`	Pseudo-cylindrical	•			
Putnins P5	`putnins5`	Pseudo-cylindrical				
Quartic Authalic	`quartic`	Pseudo-cylindrical	•			
Robinson	`robinson`	Pseudo-cylindrical				
Sinusoidal	`sinusoid`	Pseudo-cylindrical	•			
Tissot Modified Sinusoidal	`modsine`	Pseudo-cylindrical	•			
Wagner IV	`wagner4`	Pseudo-cylindrical	•			
Winkel I	`winkel`	Pseudo-cylindrical				

Projection	Syntax	Type	Equal--Area	Con-formal	Equi-distant	Special Features
Albers Equal-Area Conic	`eqaconic`	Conic	•			
Equidistant Conic	`eqdconic`	Conic			•	
Lambert Conformal Conic	`lambert`	Conic		•		
Murdoch I Conic	`murdoch1`	Conic			•	3
Murdoch III Minimum Error Conic	`murdoch3`	Conic			•	3
Bonne	`bonne`	Pseudoconic	•			
Werner	`werner`	Pseudoconic	•			
Polyconic	`polycon`	Polyconic				
Van Der Grinten I	`vgrint1`	Polyconic				
Breusing Harmonic Mean	`breusing`	Azimuthal				
Equidistant Azimuthal	`eqdazim`	Azimuthal			•	
Gnomonic	`gnomonic`	Azimuthal				4
Lambert Azimuthal Equal-Area	`eqaazim`	Azimuthal	•			
Orthographic	`ortho`	Azimuthal				
Stereographic	`stereo`	Azimuthal		•		5
Universal Polar Stereographic (UPS)	`ups`	Azimuthal		•		5
Vertical Perspective Azimuthal	`vperspec`	Azimuthal				
Wiechel	`wiechel`	Pseudo-azimuthal	•			
Aitoff	`aitoff`	Modified Azimuthal				

Projection	Syntax	Type	Equal--Area	Con-formal	Equi-distant	Special Features
Briesemeister	`bries`	Modified Azimuthal	•			
Hammer	`hammer`	Modified Azimuthal	•			
Globe	`globe`	Spherical	•	•	•	6

1 Straight rhumb lines.

2 Rhumb lines from central point are straight, true to scale, and correct in azimuth.

3 Correct total area.

4 Straight line great circles.

5 Great and small circles appear as circles or lines.

6 Three-dimensional display (not a map projection).

Creating Web Map Service Maps

Introduction to Web Map Service

In this section...

What Web Map Service Servers Provide

Web Map Service (WMS) servers follow a standard developed by the Open Geospatial Consortium, Inc.® (OGC) and provide access to a wealth of geospatial information. With maps from WMS servers, you can:

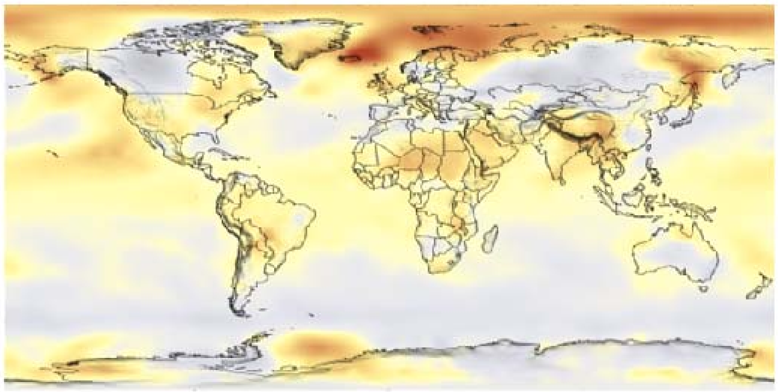
- Use any publicly available WMS data
- Easily adjust colors and styles to more clearly display information
- Update your map to reflect the most recent data
- Share your map with others

Mapping Toolbox software simplifies the process of WMS map creation by using a stored database of WMS servers. You can search the database for layers and servers that are of interest to you.

As an example, the WMS Global Mosaic map displays data from Landsat7 satellite scenes.

Courtesy NASA/JPL-Caltech

The Ozone Effect on Global Warming map displays data from the NASA Goddard Institute for Space Studies (GISS) computer model study.


Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

Basic WMS Terminology

- **Open Geospatial Consortium, Inc. (OGC)**— An organization comprising companies, government agencies, and universities that defines specifications for providers of geospatial data and developers of software designed to access that data. The specifications ensure that providers and clients can talk to each other and thus promote the sharing of geospatial data worldwide. You can access the Web Map Server Implementation Specification at the OGC Web site.
- **Web Map Service** — The OGC® defines a Web Map Service (WMS) as an entity that "produces maps of spatially referenced data dynamically from geographic information."
- **WMS server**— A server that follows the guidelines of the OGC to render maps and return them to clients.
- **georeferenced** — Tied to a specific location on the Earth.
- **raster data** — Data represented as a matrix in which each element corresponds to a specific rectangular or quadrangular geographic area.
- **map** — The OGC defines a map as "a portrayal of geographic information as a digital image file suitable for display on a computer screen."
- **raster map** — Geographically referenced information stored as a regular array of cells.
- **layer** — A data set containing a specific type of geographic information. Information can include temperature, elevation, weather, orthoimagery, boundaries, demographics, topography, transportation, environmental measurements, or various data from satellites.
- **capabilities document** — An XML document containing metadata describing the geographic content offered by a server.

Basic Workflow for Creating WMS Maps

Workflow Summary

1. Search WMS Database.
2. Refine search.
3. Update layer.
4. Modify request.
5. Retrieve map.
6. Display map.

Creating a Map of Elevation in Europe

Follow the example to learn the basic steps in creating a WMS map.

1. Search the local WMS Database for a layer. WMS servers store map data in units called layers.

   ```
   elevation = wmsfind('elevation');
   ```

   `wmsfind` returns an array of hundreds of `WMSLayer` objects.

2. Refine your search. Because you want a map of Europe, use the `WMSLayer.refine` method to search your results for layers on the European Space Agency (ESA) WMS server.

   ```
   euroheight = elevation.refine('esa.int', 'SearchField', 'serverurl');
   ```

   Refine your search again to include only the layer with DEM (Digital Elevation Model) in the `LayerName` field.

   ```
   demlayer = euroheight.refine('DEM', 'SearchField', 'layername');
   ```

3. Update your layer. You can skip this optional step for this example. The `wmsupdate` function accomplishes two tasks:

- Updates your `WMSLayer` object to include the most recent data
- Fills in its `Details`, `CoordRefSysCodes`, and `Abstract` fields

4 Modify your request. Specify geographic limits, image dimensions, background color, and other properties of the map. In this simple example, you can skip this step.

5 Retrieve your map:

First, set up a map axes with projection and geographic limits appropriate for Europe.

```
figure
worldmap europe;
```

Then, return the map axes map structure, which contains the settings for all the current map axes properties.

```
mstruct = gcm;
```

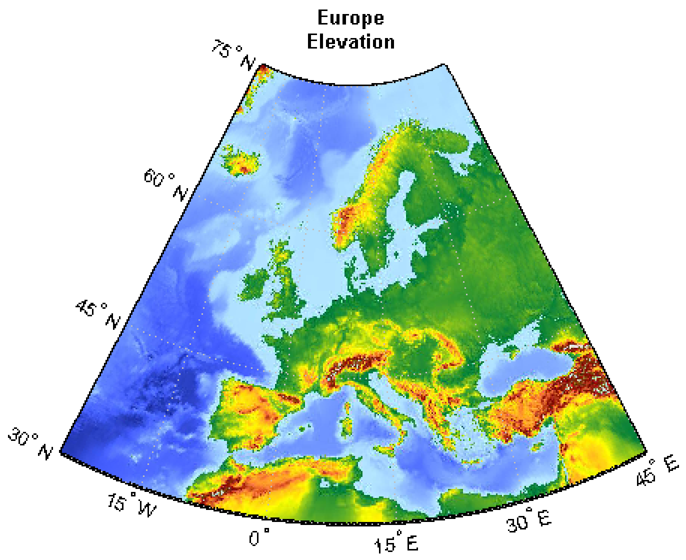
Use the `WMSLayer` object `elevation` as input for `wmsread`. Set the `wmsread` longitude and latitude limit parameters to the current map axes limits.

```
[elevationImage,R] = wmsread(demlayer, 'Latlim', ...
   mstruct.maplatlimit, 'Lonlim', mstruct.maplonlimit);
```

The `wmsread` function returns a map called `elevationImage` and a referencing matrix `R`, which ties the map to a specific location on Earth.

6 Display your map.

```
geoshow(elevationImage,R);
title({'Europe','Elevation'}, 'FontWeight', 'bold')
```



Courtesy European Space Agency (ESA)

Searching the WMS Database

In this section...
"Introduction to the WMS Database" on page 9-8
"Finding Temperature Data" on page 9-9

Introduction to the WMS Database

The Mapping Toolbox contains a database of over 1,000 stored WMS servers and over 100,000 layers. This database, called the *WMS Database*, updates at the time of the software release and includes a subset of available WMS servers. The MathWorks™ created the database by conducting a series of Internet searches and qualifying the search results.

Note The MathWorks cannot guarantee the stability and accuracy of WMS data, as the servers listed in the WMS Database are located on the Internet and are independent from the MathWorks. Occasionally, you may receive error messages from servers experiencing difficulties. The WMS Database changes at the beginning of each new software release. Servers can go down or become unavailable.

The WMS Database contains the following fields.

Field Name	Data Type	Field Content
`ServerTitle`	String	Title of the WMS server, descriptive information about the server
`ServerURL`	String	URL of the WMS server
`LayerTitle`	String	Title of the layer, descriptive information about the layer
`LayerName`	String	Name of the layer, keyword the server uses to retrieve the layer

Field Name	Data Type	Field Content
`Latlim`	Two-element vector	Southern and northern latitude limits of the layer
`Lonlim`	Two-element vector	Western and eastern longitude limits of the layer

The `LayerTitle` and `LayerName` fields sometimes have the same values. The `LayerName` indicates a code used by the servers, such as `'29:2'`, while the `LayerTitle` provides more descriptive information. For instance, `'Elevation and Rivers with Backdrop'` is a `LayerTitle`.

`wmsfind` is the only WMS function that accesses the stored WMS Database. The following example illustrates how to use `wmsfind` to find a layer.

Finding Temperature Data

For this example, assume that you work as a research scientist and study the relationship between global warming and plankton growth. Increased plankton growth leads to increased carbon dioxide absorption and reduced global warming. The sea surface temperature is already rising, however, which may reduce plankton growth in some areas. You begin investigating this complex relationship by mapping sea surface temperature.

1 Search the WMS Database for temperature data.

```
layers = wmsfind('temperature');
```

By default, `wmsfind` searches both the `LayerName` and `LayerTitle` fields of the WMS Database for partial matches. The function returns an array of class `WMSLayer`, which contains one `WMSLayer` object for each layer whose name or title partially matches `'temperature'`.

2 Click `layers` in the Workspace browser and then click one of the objects labeled `<1x1 WMSLayer>`.

Sample Output:

```
ServerTitle: 'NASA SVS Image Server'
  ServerURL: 'http://aes.gsfc.nasa.gov/cgi-bin/wms?'
```

```
         LayerTitle: 'Background Image for Global Sea Surface ...
                     Temperature from June, 2002 to September,
                     2003 (WMS)'
          LayerName: '2905_17492_bg'
             Latlim: [-90.0000 90.0000]
             Lonlim: [-180.0000 180.0000]
           Abstract: '<Update using WMSUPDATE>'
    CoordRefSysCodes: '<Update using WMSUPDATE>'
            Details: '<Update using WMSUPDATE>'
```

A `WMSLayer` object contains three fields that do not appear in the WMS Database—`Abstract`, `CoordRefSysCodes`, and `Details`. (By default, these fields do not display in the command window if they are not populated with `wmsupdate`. For more information, see "Updating Your Layer" on page 9-13 in the *Mapping Toolbox User's Guide*.)

Note `WMSLayer` is one of several classes related to WMS. If you are new to object-oriented programming, you can learn more about classes, methods, and properties in the *Object-Oriented Programming* section of the MATLAB documentation.

Refining Your Search

In this section...

Refining by Text String

Your initial search may return hundreds or even thousands of layers. Scanning all these layers to find the most relevant one could take a long time. You need to refine your search.

1 Refine your search to receive only layers that include sea surface temperature.

```
layers = wmsfind('temperature');
sst = layers.refine('sea surface');
```

2 Refine the search again to include only layers that contain the term "global."

```
global_sst = sst.refine('global');
```

3 Display one of the layers.

```
global_sst(4).disp
```

Sample Output:

```
       Index: 4
 ServerTitle: 'NASA SVS Image Server'
   ServerURL: 'http://aes.gsfc.nasa.gov/cgi-bin/wms?'
  LayerTitle: 'Background Image for Global Sea Surface ...
              Temperature Anomalies from June, 2002 ...
              to September, 2003 (WMS)'
   LayerName: '2906_17499_bg'
      Latlim: [-90.0000 90.0000]
      Lonlim: [-180.0000 180.0000]
```

Refining by Geographic Limits

You can search for layers in a specific geographic area.

1 First, find hurricane layers.

```
layers = wmsfind('hurricane');
```

2 Refine your search by selecting layers that are in the western hemisphere.

```
western_hemisphere = layers.refineLimits ...
   ('Latlim',[-90 90], 'Lonlim', [-180 0]);
```

3 Refine again to include only layers in the western hemisphere that include temperature data.

```
temp_and_west = western_hemisphere.refine('temperature');
```

Updating Your Layer

After you find your specific layer of interest, you can leave the local WMS Database and work with a WMS server. In this section, you learn how to synchronize your layer with the WMS source server.

Note When working with the Internet, you may have to wait several minutes for information to download, or servers can become unavailable. If you encounter problems, refer to "Common Problems with WMS Servers" on page 9-58 for tips.

Use the `wmsupdate` function to synchronize a `WMSLayer` object with the corresponding WMS server. This synchronization populates the `Abstract`, `CoordRefSysCodes`, and `Details` fields.

1 Find all layers in the WMS Database with the title "Global Sea Surface Temperature."

```
global_sst = wmsfind ('Global Sea Surface Temperature', ...
   'SearchField', 'LayerTitle');
```

2 Use the `WMSLayer` `servers` method to determine the number of unique servers.

```
global_sst.servers
```

3 If your search returns more than one server, consider setting the `wmsupdate` `'AllowMultipleServers'` property to `true`. (However, be aware that if you have many servers, updating them could take a long time.)

```
global_sst = wmsupdate(global_sst, 'AllowMultipleServers', true);
```

4 Now that you have updated all the fields in your `WMSLayer` objects, you can search by the `Abstract` field.

```
daily = global_sst.refine ('daily', 'SearchField', 'abstract')
```

Type `daily.Abstract` at the command line to view the abstract. The data for this layer updates daily; thus the name of the layer.

Sample Output:

```
Global sea surface temperature. The data used are from the
DUE/Medspiration project which is a real-time service for
the production and delivery of high-resolution sea surface
temperature from all available satellite sensors. The data
cover the period from 2007-10-01 and is updated daily.
```

5 Type `daily.CoordRefSysCodes` at the command line to view the coordinate reference system codes. In this example, `'AUTO:42003'` and `'EPSG:4326'` are given as the `CoordRefSysCodes`.

The Mapping Toolbox supports `EPSG:4326` but does not support auto codes. For more information, see "Understanding Coordinate Reference System Codes" on page 9-15 in the *Mapping Toolbox User's Guide*.

6 To view the contents of the `Details` field, type `daily.Details` at the command line.

Sample Output:

```
ans =

      MetadataURL: [1x75 char]
       Attributes: [1x1 struct]
      BoundingBox: [1x1 struct]
        Dimension: [1x1 struct]
     ImageFormats: {4x1 cell}
      ScaleLimits: [1x1 struct]
            Style: [1x1 struct]
          Version: '1.3.0'
```

You can go to the URL listed in the `MetadataURL` field to learn more about when and how this data was collected. The `Style` field covers a wide range of information, such as the line styles used to render vector data, the background color, the numeric format of data, the month of data collection, or the dimensional units. In this case, the WMS server provides no style information.

Retrieving Your Map

In this section...

Ways to Retrieve Your Map

To retrieve a map from a WMS server, use the function `wmsread` or, in a few specific situations, the `WebMapServer.getMap` method. Use the `getMap` method when:

- Working with non-EPSG:4326 reference systems
- Creating an animation of a specific geographic area over time
- Retrieving multiple layers from a WMS server

In most cases, use `wmsread` to retrieve your map. To use `wmsread`, specify either a `WMSLayer` object or a map request URL. Obtain a `WMSLayer` object by using `wmsfind` to search the WMS Database. Obtain a map request URL from:

- The output of `wmsread`
- The `RequestURL` property of a `WMSMapRequest` object
- An Internet search

The map request URL string is composed of a WMS server URL with additional WMS parameters. The map request URL can be inserted into a browser to make a request to a server, which then returns a raster map.

Understanding Coordinate Reference System Codes

When using `wmsread`, request a map that uses the EPSG:4326 coordinate reference system. EPSG stands for European Petroleum Survey Group. This group, an organization of specialists working in the field of oil exploration, developed a database of coordinate reference systems. Coordinate reference systems identify position unambiguously. Coordinate reference system codes are numbers that stand for specific coordinate reference systems.

EPSG:4326 is based on the 1984 World Geodetic System (WGS84) datum and the latitude and longitude coordinate system, with angles in degrees and Greenwich as the central meridian. All servers in the WMS Database, and presumably all WMS servers in general, use the EPSG:4326 reference system. This system is a requirement of the OGC WMS specification. If a layer does not use EPSG:4326, Mapping Toolbox software uses the next available coordinate reference system code. The Mapping Toolbox does not support automatic coordinate reference systems (systems in which the user chooses the center of projection). For more information about coordinate reference system codes, please see the Spatial Reference Web site.

Retrieving Your Map with wmsread

NASA's Blue Marble layer shows the Earth's surface for each month of 2004 at high resolution (500 meters/pixel). Read and display the Blue Marble layer.

1 Search the WMS Database for all layers with `'nasa'` in the `ServerURL` field.

```
nasa = wmsfind('nasa', 'SearchField', 'serverurl');
```

2 Use the `WMSLayer.refine` method to refine your search to include only those layers with the phrase `'bluemarble'` in the `LayerName` field. This syntax creates an exact search.

```
layer = nasa.refine('bluemarble',  'SearchField', 'layername', ...
   'MatchType', 'exact');
```

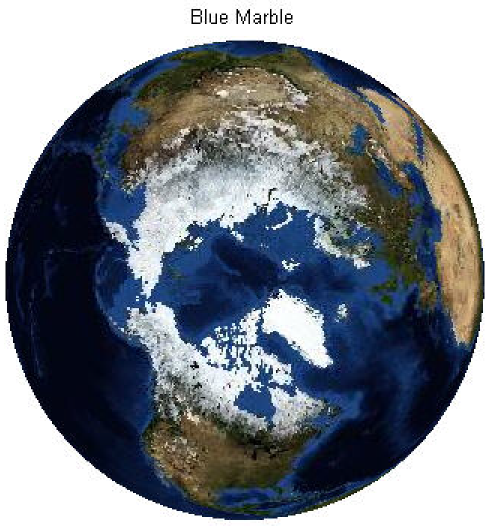
3 Use the `wmsread` function to retrieve the Blue Marble layer.

```
[A, R] = wmsread(layer);
```

The `wmsread` function returns `A`, a geographically referenced raster map, and `R`, a referencing matrix that ties `A` to the EPSG:4326 geographic coordinate system. The geographic limits of `A` span the full latitude and longitude extent of `layer`.

4 Open a figure window, set up your map axes, and display your map.

```
figure
axesm globe
axis off
geoshow(A, R)
title('Blue Marble')
```



Courtesy NASA/JPL-Caltech

Setting Optional Parameters

The `wmsread` function allows you to set many optional parameters, such as geographic limits, image height and width, and background color. This example demonstrates how to view an elevation map in 0.5-degree resolution by changing the cell size and how to display the ocean in light blue by setting the background color. For a complete list of parameters, see the `wmsread` reference page.

1 The European Space Agency (ESA), like NASA, hosts a wide variety of layers. Search the WMS Database for layers that contain the string `'esa.int'` in the `ServerURL` field.

```
esa = wmsfind('esa.int', 'SearchField', 'serverurl');
```

2. GTOPO30, a digital elevation model developed by the United States Geological Survey (USGS), has a horizontal grid spacing of 30 arc seconds. Refine your search to include only layers that contain the string `'gtopo30'` in the `LayerName` and `LayerTitle` fields.

   ```
   gtopo30Layer = esa.refine('gtopo30');
   ```

   The refined search returns one layer.

3. Choose red, green, and blue levels to define a background color.

   ```
   oceanColor = [0 170 255];
   ```

4. Use the `BackgroundColor` and `CellSize` parameters of the `wmsread` function to determine the background color and cell size as you retrieve your map.

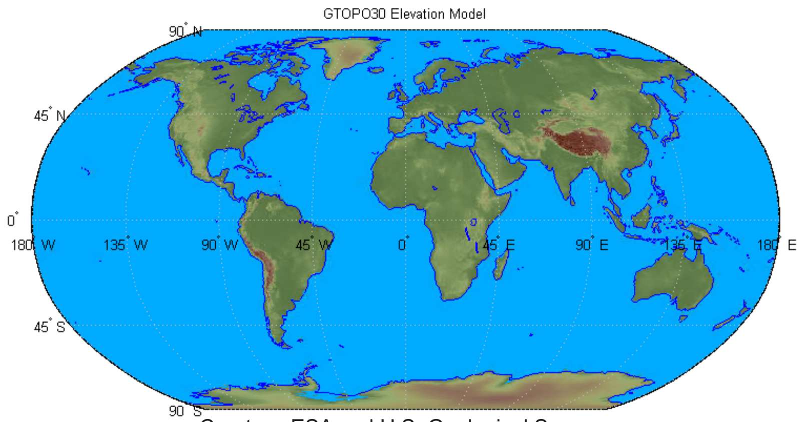
   ```
   [A,R] = wmsread(gtopo30Layer, 'BackgroundColor', oceanColor, ...
      'CellSize', 0.5);
   ```

5. Open a figure window, set up a world map axes, and display your map.

   ```
   figure
   worldmap world
   geoshow(A, R)
   ```

6. Load and plot coastline data and title your map.

```
coast = load('coast');
plotm(coast.lat, coast.long)
title('GTOPO30 Elevation Model')
```



Courtesy ESA and U.S. Geological Survey

Retrieving Your Map with WebMapServer.getMap

If you want to retrieve a layer that does not use the EPSG:4326 reference system code, use the `WebMapServer.getMap` method.

The Shaded Red Mars map is a 24-bit color image in the Plate Carree Equirectangular projection, representing Mars as viewed from space. The data, hosted on the CubeWerx® Web server, was derived from NASA's Jet Propulsion Laboratory imagery.

1 Find the Shaded Red layer.

```
mars = wmsfind('shadedred');
```

2 Update the results from multiple servers by setting the `AllowMultipleServers` parameter to `true`.

```
mars = wmsupdate(mars, 'AllowMultipleServers', true);
```

3 A `WebMapServer` object is a handle object that represents a WMS server. `ServerURL` is one of the properties of `WMSLayer`. Create a `WebMapServer` object from the `ServerURL` of the first `mars` layer.

```
mars = mars(1);
server = WebMapServer(mars.ServerURL);
```

4 Use the WebMapServer handle object and the WMSLayer object to create a WMSMapRequest object.

```
request = WMSMapRequest(mars, server);
```

5 Use the WMSMapRequest.boundImageSize method to bound the size of the output map such that the longest dimension has a size of 2048 pixels.

```
request = request.boundImageSize(2048);
```

6 The WMSMapRequest class includes a RequestURL property composed of the server URL with additional WMS parameters and values. Obtain your map of Mars by passing the WMSMapRequest.RequestURL of request to the WebMapServer.getMap method of server.

```
A = server.getMap(request.RequestURL);
```

7 Open a figure window and set the window to black.

```
figure
set(gcf,'color','black')
```

8 Set the axes.

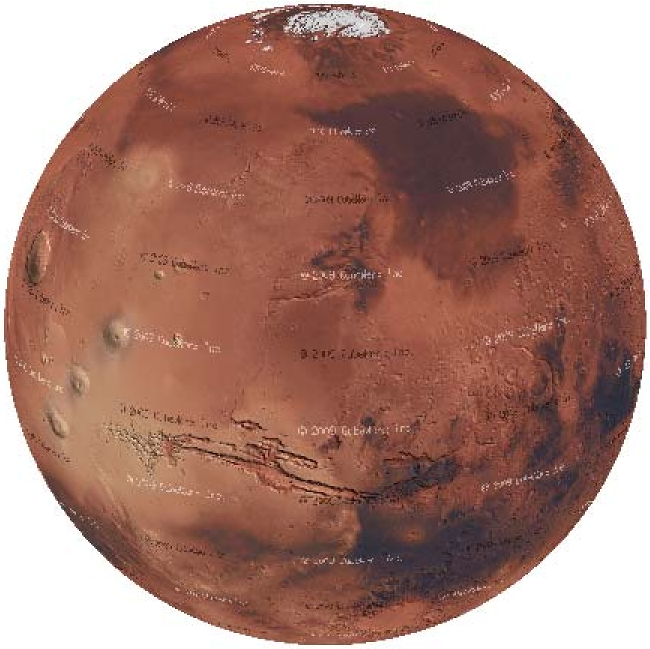
```
axesm globe
axis off
```

9 Display the map. The geoshow function accepts an image and a referencing matrix. The WMSMapRequest.RasterRef property references the raster map to an intrinsic coordinate system.

```
geoshow(A, request.RasterRef);
```

10 Set the point of view by adjusting the CameraPosition and CameraViewAngle axes properties.

```
cameraPosition = [7.5531 -14.677 5.3633];
cameraViewAngle = 5.6024;
set(gca,'cameraposition', cameraPosition)
set(gca,'cameraviewangle', cameraViewAngle)
```



Courtesy NASA/JPL-Caltech and Cubewerx.com

Modifying Your Request

In this section...

Setting the Geographic Limits and Background Color

A `WMSMapRequest` object contains properties to modify the geographic extent, rendering, and size of the requested map. You can use this request to set geographic limits and background color in the next example. See the `WMSMapRequest` reference page for a complete list of properties.

Map sea surface temperature for the ocean surrounding the southern tip of Africa. Set the color of the land areas (background) to black.

1 Search the WMS Database for all layers on the European Space Agency (ESA) server.

```
esa = wmsfind('esa.int', 'SearchField', 'serverurl');
```

2 Refine your search to include only layers with `'global sea surface'` in the layer title or layer name fields of the WMS database.

```
sst = esa.refine('global sea surface');
```

3 Construct a `WebMapServer` object from the server URL stored in the `ServerURL` property of the `WMSLayer` object `sst`.

```
server = WebMapServer(sst(1).ServerURL);
```

4 Construct a `WebMapRequest` object from a `WMSLayer` array and a `WebMapServer` object.

```
mapRequest = WMSMapRequest(sst, server);
```

5 Use the `Latlim` and `Lonlim` properties of `WMSMapRequest` to set the latitude and longitude limits.

```
mapRequest.Latlim = [-45 -25];
mapRequest.Lonlim = [15 35];
```

6 Set the background color.

```
mapRequest.BackgroundColor = [0 0 0];
```

7 Send your request to the server with the `WebMapServer.getMap` method. Pass in a `WMSMapRequest.RequestURL`.

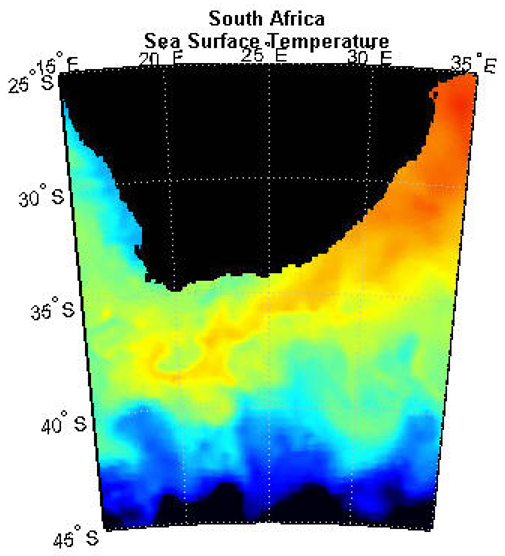
```
sstImage = server.getMap(mapRequest.RequestURL);
```

8 Set up empty map axes with the specified geographic limits.

```
figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim);
```

9 Project and display an image georeferenced to latitude and longitude. Use the referencing matrix provided by the `RasterRef` property of the `WMSMapRequest` object.

```
geoshow(sstImage, mapRequest.RasterRef);
title({'South Africa','Sea Surface Temperature'}, ...
   'FontWeight', 'bold')
```



Courtesy ESA

Setting the Geographic Limits, Image Dimension, Style, and Image Format

The Shuttle Radar Topography Mission (SRTM) is a project led by the U.S. National Geospatial-Intelligence Agency (NGA) and NASA. SRTM has created a high-resolution, digital, topographic database of Earth. The JPL `'srtmplus'` layer includes this SRTM data plus GTOPO30 data and UCSD Sandwell bathymetry data. Follow this example to read and display a global elevation and bathymetry layer for the Gulf of Maine at a 30 arc-second sampling interval.

1 Find the `'srtmplus'` layer in the WMS Database.

```
layer = wmsfind('srtmplus', 'MatchType', 'exact');
layer = layer(1);
```

2 Construct a `WebMapServer` object to represent the WMS server.

```
server = WebMapServer(layer.ServerURL);
```

3 Construct a `WMSMapRequest` object from the `WMSLayer` object and the `WebMapServer` object.

```
mapRequest = WMSMapRequest(layer, server);
```

4 Set the geographic limits of the `WMSMapRequest` object.

```
mapRequest.Latlim = [40 46];
mapRequest.Lonlim = [-71 -65];
```

5 Set the sampling interval to 30 arc-seconds.

```
samplesPerInterval = dms2degrees([0 0 30]);
```

6 Set the `WMSMapRequest.ImageHeight` property.

```
mapRequest.ImageHeight = ...
   round(diff(mapRequest.Latlim)/samplesPerInterval);
```

7 Set the `ImageWidth` property.

```
mapRequest.ImageWidth = ...
   round(diff(mapRequest.Lonlim)/samplesPerInterval);
```

8 Set the `StyleName` property to `'short_int'` for meters. (The `Abstract` states, "It is possible to request the elevation data in meters by the short_int style....")

```
mapRequest.StyleName = 'short_int';
```

9 Set the `ImageFormat` to GeoTIFF.

```
mapRequest.ImageFormat = 'image/geotiff';
```

10 After setting the `WMSMapRequest` properties, send the `RequestURL` of your `WMSMapRequest` object to the WMS server via the `WebMapServer.getMap` method.

```
Z = server.getMap(mapRequest.RequestURL);
```

11 Open a figure window and set up a map axes with geographic limits that match the limits in your `WMSMapRequest` object.

```
figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim)
```

12 Convert your map data to `double` precision and display it on your map axes. The referencing matrix `mapRequest.RasterRef` ties the intrinsic coordinates of the raster map to the EPSG:4326 geographic coordinate system.

```
geoshow(double(Z), mapRequest.RasterRef, ...
   'DisplayType', 'texturemap')
```

13 Create a colormap appropriate for elevation data.

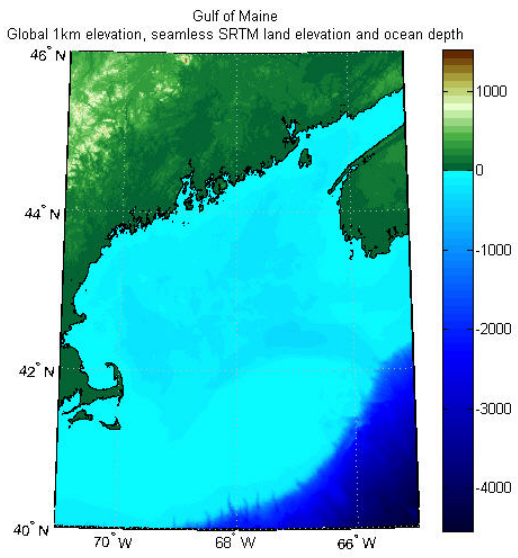
```
demcmap(double(Z))
```

14 Use the `zbuffer` renderer setting because this example uses lighting.

```
set(gcf,'Renderer','zbuffer')
```

15 Display and contour the map at sea level (0 m).

```
contourm(double(Z), mapRequest.RasterRef, [0 0], 'Color', 'black')
colorbar
title ({'Gulf of Maine', mapRequest.Layer.LayerTitle}, ...
   'Interpreter','none')
```



Courtesy NASA/JPL

Manually Editing a URL

You can modify a map request URL manually.

1 Obtain the map request URL.

```
nasa = wmsfind('nasa', 'SearchField', 'serverurl');
layer = nasa.refine('bluemarble', 'SearchField', 'layername', ...
   'MatchType', 'exact');
layer = layer(1);
mapRequest = WMSMapRequest(layer);
```

2 View the map request URL by typing `mapRequest.RequestURL` at the command line.

Sample Output:

```
ans =

http://g0hep12u.ecs.nasa.gov/mapserv-bin/wms_ogc? ...
   SERVICE=WMS ...
```

```
      &LAYERS=bluemarble ...
      &EXCEPTIONS=application/vnd.ogc.se_xml ...
      &FORMAT=image/jpeg&TRANSPARENT=FALSE ...
      &HEIGHT=256 ...
      &BGCOLOR=0xFFFFFF ...
      &REQUEST=GetMap ...
      &WIDTH=512 ...
      &BBOX=-180.0,-90.0,180.0,90.0 ...
      &STYLES=&SRS=EPSG:4326 ...
      &VERSION=1.1.1
```

3. Modify the bounding box to include the southern hemisphere by directly changing the `mapRequest.RequestURL`. Enter the following at the command line:

   ```
   modifiedURL =
   ```

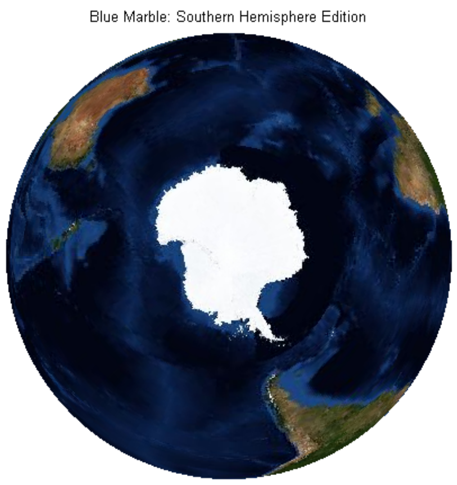
   Then enter the lengthy URL as shown in the previous code sample, but change the bounding box:

   ```
   &BBOX=-180.0,-90.0,180.0,0.0
   ```

   Enter the URL as one continuous string.

4. Display the modified map.

   ```
   [A, R] = wmsread(modifiedURL);
   figure
   axesm globe
   axis off
   geoshow(A, R)
   title('Blue Marble: Southern Hemisphere Edition')
   ```



Courtesy NASA/JPL-Caltech

Overlaying Multiple Layers

In this section...

Creating a Composite Map of Multiple Layers from One Server

The WMS specification allows the server to merge multiple layers into a single raster map. NASA's Globe Visualization server contains many data layers, such as coastlines, national boundaries, and the EGM96 model of the Earth's gravitational potential. Read and display a composite of multiple layers from the Globe Visualization server. The rendered map has a spatial resolution of 0.5 degrees.

1 Find the coastline, national boundaries, and EGM96 layers of the Globe Visualization server.

```
vizglobe = wmsfind('viz.globe', 'SearchField', 'serverurl');
coastlines = vizglobe.refine('coastline');
national_boundaries = vizglobe.refine('national*bound');
base_layer = vizglobe.refine('egm96');
```

2 Concatenate the results into a single `WMSLayer` array.

```
layers = [base_layer;coastlines;national_boundaries];
```

3 Construct a `WMSMapRequest` object from the `WMSLayer` array. (For this to work, the layers must all have the same server.)

```
request = WMSMapRequest(layers);
```

4 Request a transparent map background. All pixels not representing features or data values in a layer are set to a transparent value in the resulting image, making it possible to produce a composite map.

```
request.Transparent = true;
```

5 Use the `WMSMapRequest.boundImageSize` method to bound the size of the raster map.

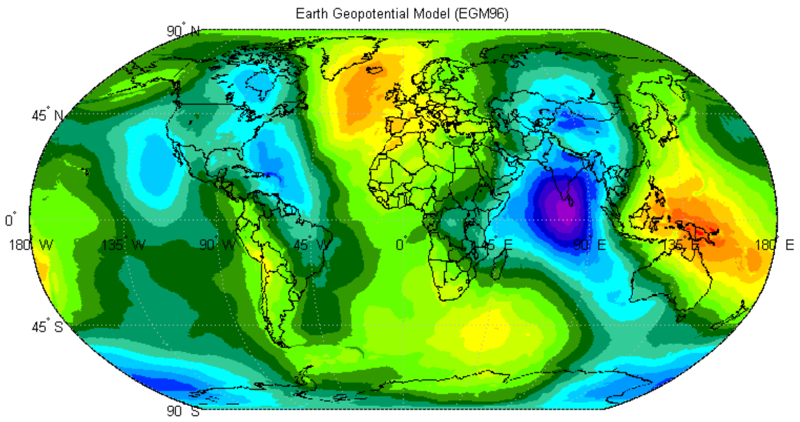
```
request = request.boundImageSize(720);
```

6 Pass the `RequestURL` of `request` to `WebMapServer.getMap` to retrieve your composite map.

```
overlayImage = request.Server.getMap(request.RequestURL);
```

7 Display your composite map.

```
figure
worldmap('world')
geoshow(overlayImage, request.RasterRef);
title(base_layer.LayerTitle)
```



Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

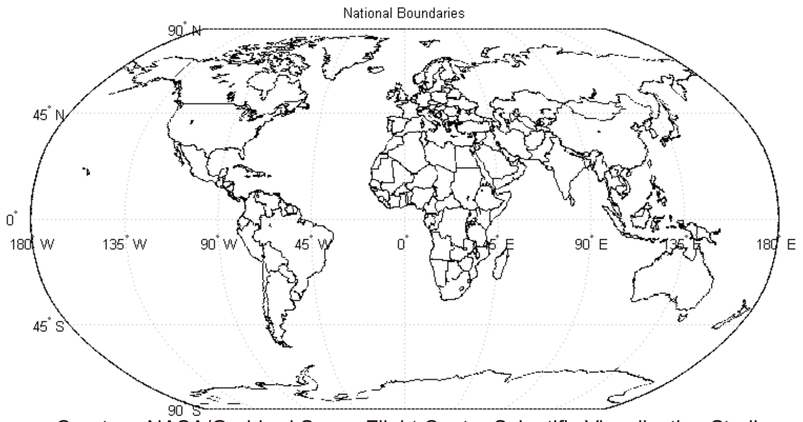
8 Read and display only the coastlines and national boundaries.

```
boundaries = [coastlines; national_boundaries];
```

```
[boundariesImage, R] = wmsread(boundaries, 'CellSize', .5, ...
   'Transparent', true);
```

9 Display the raster map.

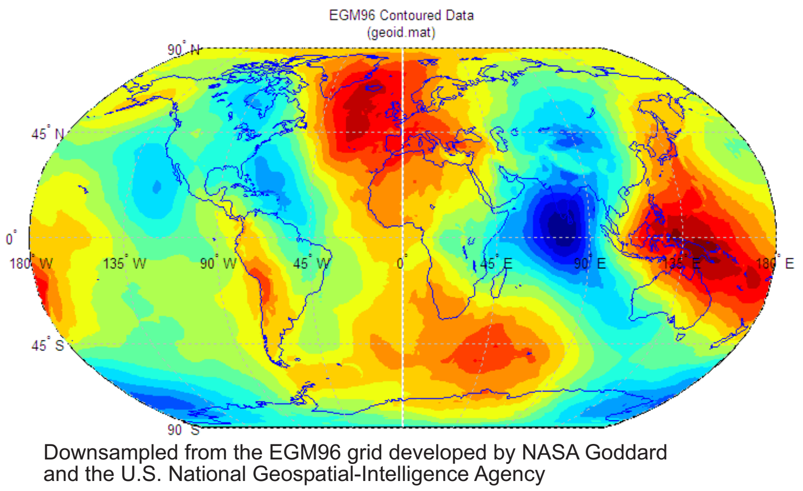
```
figure
worldmap('world')
geoshow(boundariesImage, R);
title(national_boundaries.LayerTitle)
```



Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

10 Compare the composite map with all three layers with the contoured data from `'geoid.mat'`. (The colormaps differ, but the information presented in both cases is the same.)

```
geoid = load('geoid');
coast = load('coast');
figure
worldmap('world')
contourfm(geoid.geoid, geoid.geoidrefvec, 15)
geoshow(coast.lat, coast.long)
title({'EGM96 Contoured Data', '(geoid.mat)'})
```


EGM96 Contoured Data
(geoid.mat)
90° N
45° N
0°
45° S
90° S
180° W
135° W
90° W
45° W
0°
45° E
90° E
135° E
180° E
Downsampled from the EGM96 grid developed by NASA Goddard and the U.S. National Geospatial-Intelligence Agency

Combining Layers from One Server with Data from Other Sources

This example, a continuation of the one preceding it, illustrates how you can merge the boundaries raster map with vector data. Combine two separate calls to `vec2mtx` to create a 4-color raster map showing interior land areas, coastlines, national boundaries, oceans, and world rivers. The `vec2mtx` function converts latitude-longitude vectors to a regular data grid.

1 Read the `landareas` vector shapefile and convert it to an image.

```
land = shaperead('landareas', 'UseGeoCoords', true);
lat = [land.Lat];
lon = [land.Lon];
[landImage, refvec] = ...
   vec2mtx(lat, lon, 2, [-90, 90], [-180,180], 'filled');
mergedImage = landImage;
```

2 Read the `worldrivers` vector shapefile and convert it to an image.

```
rivers = shaperead('worldrivers.shp','UseGeoCoords',true);
riverImage = vec2mtx([rivers.Lat], [rivers.Lon], landImage, refvec);
```

3 Merge the rivers with the land.

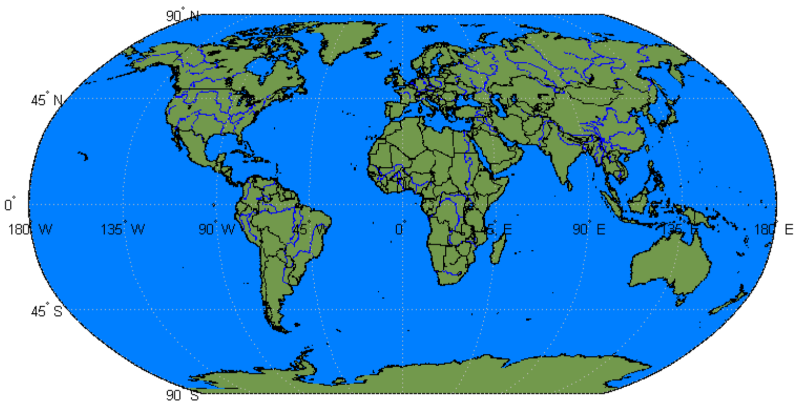
```
mergedImage(riverImage == 1) = 3;
```

4 Merge the rivers and land with the boundaries. You must first flip `mergedImage` because it has a reference defined by a referencing vector, where columns run from south to north. (The columns of WMS images run from north to south.)

```
mergedImage = flipud(mergedImage);
mergedImage(boundariesImage(:,:,1) == 0) = 1;
```

5 Display the result.

```
figure
worldmap(mergedImage, R)
geoshow(mergedImage, R, 'DisplayType', 'texturemap')
colormap([.45 .60 .30; 0 0 0; 0 0.5 1; 0 0 1])
```



Courtesy U.S. National Geospatial-Intelligence Agency (NGA)

Draping Topography and Ortho-Imagery Layers over a Digital Elevation Model Layer

Read and display an aerial image overlapping the same region found in the San Francisco South USGS 24 K Digital Elevation Model (DEM) file.

1 Uncompress the zip file and read it with the `ugs24kdem` function. Set the geographic limits to the minimum and maximum values in the DEM file.

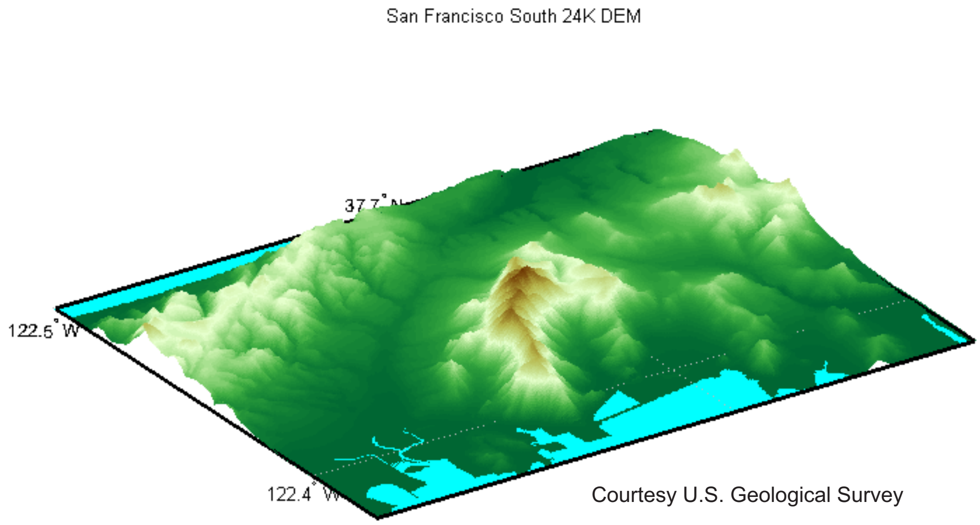
```
filenames = gunzip('sanfranciscos.dem.gz', tempdir);
demFilename = filenames{1};
[lat,lon,Z,header,profile] = usgs24kdem(demFilename, 1);
delete(demFilename);
Z(Z==0) = -1;
latlim = [min(lat(:)) max(lat(:))];
lonlim = [min(lon(:)) max(lon(:))];
```

2 Display the USGS 24K DEM data. Create map axes for the United States and assign an appropriate elevation colormap. Use `daspectm` to display the elevation data.

```
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, 'DisplayType','surface')
demcmap(Z)
daspectm('m',1)
title('San Francisco South 24K DEM');
```

3 Set the point of view by adjusting the `CameraPosition`, `CameraTarget`, and `CameraAngle` axes properties.

```
cameraPosition = [0.0102972 0.697919 39980.8];
cameraTarget = [-0.000307875 0.705072 231.46];
cameraAngle = 5.57428;
set(gca,'CameraPosition', cameraPosition, ...
   'CameraTarget', cameraTarget, ...
   'CameraViewAngle', cameraAngle)
```



4 The Microsoft TerraServer provides ortho-imagery and topography maps from various regions of the United States. The ortho-imagery layer name is UrbanArea, and the topographic layer name is DRG (short for Digital Raster Graphic).

```
terraserver = wmsfind('terraservice.net','search','serverurl');
orthoLayer = terraserver.refine('UrbanArea');
topoLayer  = terraserver.refine('DRG');
```

5 Construct a `WebMapServer` object for the ortho-imagery layer.

```
server = WebMapServer(orthoLayer.ServerURL);
```

6 Construct a `WMSMapRequest` object from the `WebMapServer` object and the ortho-imagery layer. Use `WMSMapRequest` properties to modify different aspects of your map request, such as geographic limits and image size. Set the geographic limits to cover the same region as found in the DEM file.

```
mapRequest = WMSMapRequest(orthoLayer, server);
mapRequest.ServerURL = 'http://terraservice.net/ogcmap6.ashx?';
mapRequest.Latlim = latlim;
mapRequest.Lonlim = lonlim;
mapRequest.ImageHeight = size(Z,1);
mapRequest.ImageWidth  = size(Z,2);
```

7 Request a map of the ortho-imagery layer.

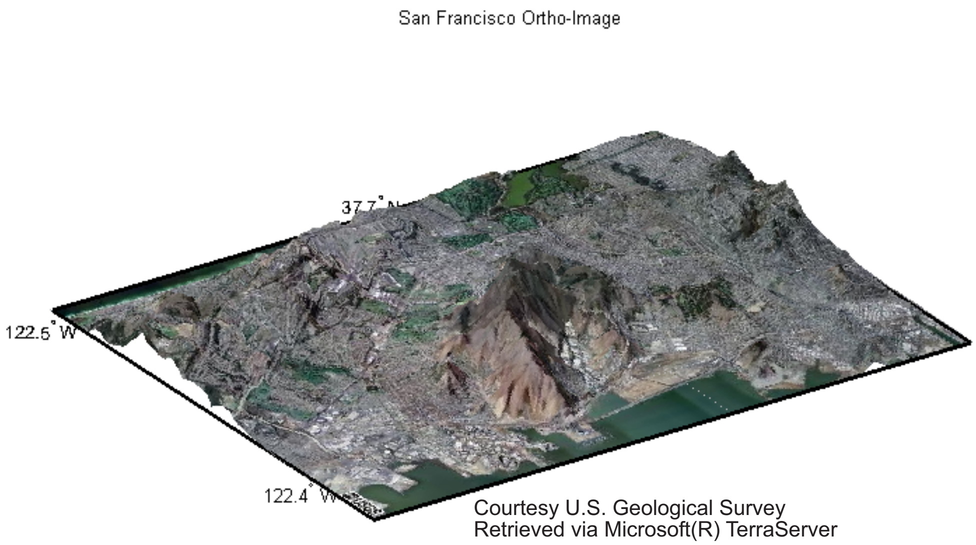
```
orthoImage = server.getMap(mapRequest.RequestURL);
```

8 Request a map of the topographic layer.

```
mapRequest.Layer = topoLayer;
topoMap = server.getMap(mapRequest.RequestURL);
```

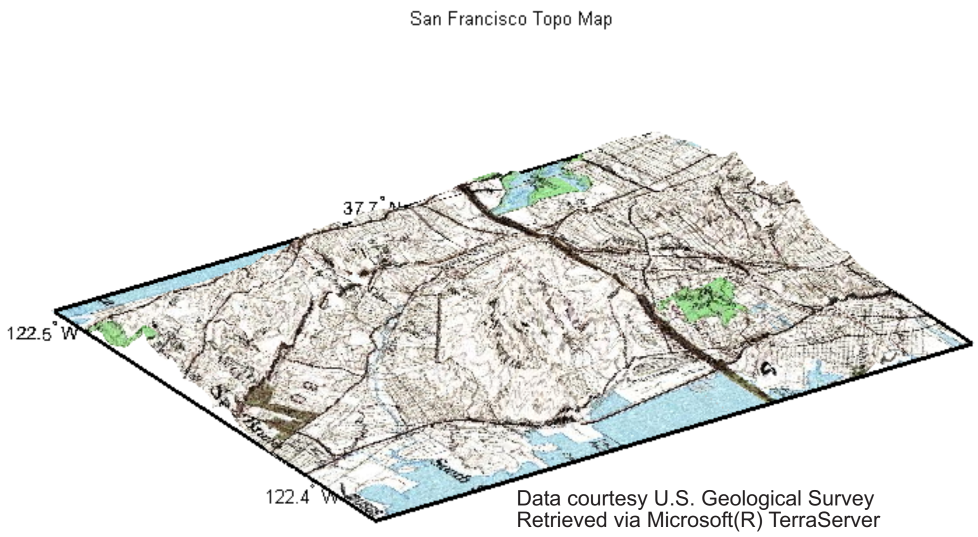
9 Drape the ortho-image onto the elevation data.

```
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, ...
   'DisplayType', 'surface', 'CData', orthoImage);
daspectm('m',1)
title('San Francisco Ortho-Image');
axis vis3d
set(gca,'CameraPosition', cameraPosition, ...
   'CameraTarget', cameraTarget, ...
   'CameraViewAngle', cameraAngle)
```


San Francisco Ortho-Image
37.7° N
122.5° W
122.4° W
Courtesy U.S. Geological Survey
Retrieved via Microsoft(R) TerraServer

10 Drape the topographic map onto the elevation data.

```
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, ...
   'DisplayType', 'surface', 'CData', topoMap);
daspectm('m',1)
title('San Francisco Topo Map');
axis vis3d
set(gca,'CameraPosition', cameraPosition, ...
   'CameraTarget', cameraTarget, ...
'CameraViewAngle', cameraAngle)
```


San Francisco Topo Map
37.7° N
122.5° W
122.4° W
Data courtesy U.S. Geological Survey
Retrieved via Microsoft(R) TerraServer

Animating Data Layers

In this section...

Creating Movie of Daily Planet Images for One Month

You can create maps of the same geographic region at different times and view them as a movie. Read and display a daily composite of visual images from NASA's Moderate Resolution Imaging Spectroradiometer (MODIS) scenes for the month of January 2009. This composite is referred to as the *Daily Planet*.

1 Search the WMS Database for the Daily Planet layer.

```
daily = wmsfind('daily');
daily_planet = daily.refine('planet');
```

2 Construct a `WebMapServer` object.

```
server = WebMapServer(daily_planet.ServerURL);
```

3 Construct a `WMSMapRequest` object.

```
mapRequest = WMSMapRequest(daily_planet, server);
```

4 Set the value for the `WMSMapRequest.Time` property to January 1, 2009. Save the time as a serial date number.

```
time = '2009-01-01';
mapRequest.Time = time;
dtime = datenum(time);
```

5 Set the total number of frames equal to the number of days in the month of January.

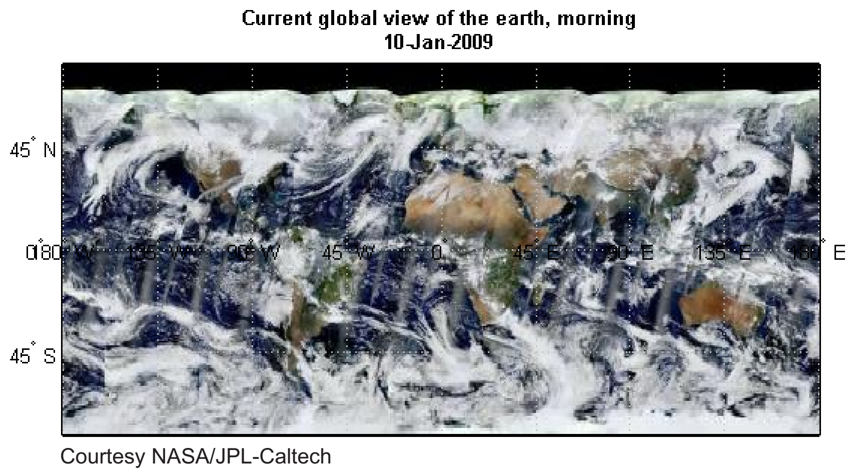
```
numberOfFrames = 31;
```

6 Open a figure window with axes appropriate for the region specified by the `daily_planet` layer.

```
figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim);
```

7 Retrieve a map of the `daily_planet` layer for each day of the month of January. Set the `Time` field to a number.

```
for k=1:numberOfFrames
   dailyImage = server.getMap(mapRequest.RequestURL);
   geoshow(dailyImage, mapRequest.RasterRef);
   title({mapRequest.Layer.LayerTitle, datestr(dtime)}, ...
      'Interpreter', 'none', 'FontWeight', 'bold')
   dtime = dtime + 1;
   mapRequest.Time = dtime;
   drawnow
   shg
end
```



Snapshot from Animation of Daily Planet

Creating an Animated GIF File

Read and display an animation of the Larsen Ice Shelf experiencing a dramatic collapse between January 31 and March 7, 2002.

1. Search the WMS Database for the phrase “Larsen Ice Shelf.”

   ```
   iceLayer = wmsfind('Larsen Ice Shelf');
   ```

   Try the first layer.

2. Construct a `WebMapServer` object.

   ```
   server = WebMapServer(iceLayer(1).ServerURL);
   ```

3. Use the `WebMapServer.updateLayers` method to synchronize the layer with the WMS source server. Retrieve the most recent data and fill in the `Abstract`, `CoordRefSysCodes`, and `Details` fields.

   ```
   iceLayer = server.updateLayers(iceLayer(1));
   ```

4. View the abstract.

   ```
   fprintf('%s\n', iceLayer(1).Abstract)
   ```

5. Create the `WMSMapRequest` object.

   ```
   request = WMSMapRequest(iceLayer(1), server);
   ```

6. Because you have updated your layer, the `Details` field now has content. Click `Details` in the Variable Editor. Then, click `Dimension`. The name of the dimension is `'time'`. Click `Extent`. The `Extent` field provides the available values for a dimension, in this case time. Save this information by entering the following at the command line:

   ```
   extent = [',' iceLayer.Details.Dimension.Extent, ','];
   ```

7. Calculate the number of required frames. (The `extent` contains a comma before the first frame and after the last frame. To obtain the number of frames, subtract 1.)

   ```
   frameIndex = findstr(extent, ',');
   numFrames = numel(frameIndex) - 1;
   ```

8 Open a figure window and set up a map axes with appropriate geographic limits.

```
h = figure;
worldmap(request.Latlim, request.Lonlim)
```

9 Set the map axes properties. `MLineLocation` establishes the interval between displayed grid meridians. `MLabelParallel` determines the parallel where the labels appear.

```
setm(gca,'MLineLocation', 1, 'MLabelLocation', 1, ...
   'MLabelParallel',-67.5, 'LabelRotation', 'off');
```

10 Initialize the value of `animated` to 0.

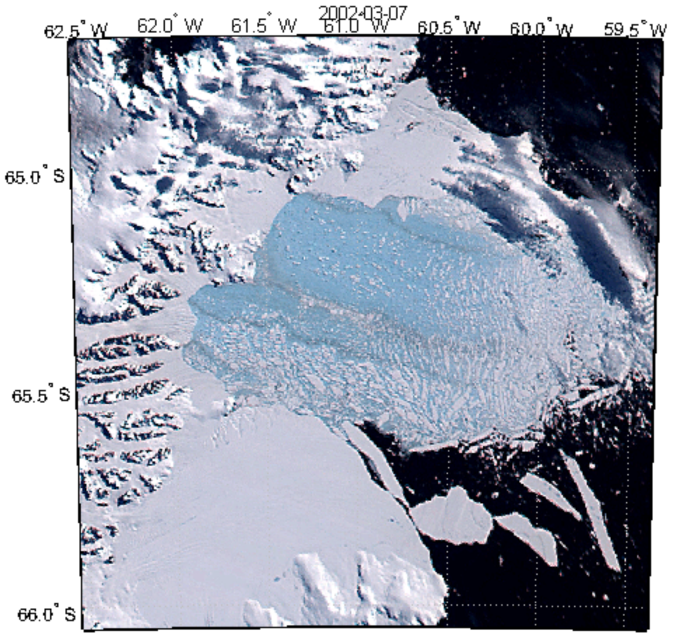
```
animated(1,1,1,numFrames) = 0;
```

11 Display the image of the Larsen Ice Shelf on different days.

```
for k=1:numFrames
   request.Time = extent(frameIndex(k)+1:frameIndex(k+1)-1);
   iceImage = server.getMap(request.RequestURL);
   geoshow(iceImage, request.RasterRef)
   title(request.Time, 'Interpreter', 'none')
   drawnow
   shg
   frame = getframe(h);
   if k == 1
      [animated, cmap] = rgb2ind(frame.cdata, 256, 'nodither');
   else
      animated(:,:,1,k) = rgb2ind(frame.cdata, cmap, 'nodither');
   end
end
```

12 Save and then view the animated GIF file.

```
filename = 'wmsanimated.gif';
imwrite(animated, cmap, filename, 'DelayTime', 1.5, ...
   'LoopCount', inf);
web(filename)
```



Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

Snapshot from Animation of Larsen Ice Shelf

Animating Time-Lapse Radar Observations

Display Next-Generation Radar (NEXRAD) images for the United States using data from the Iowa Environmental Mesonet (IEM) Web map server. The server stores layers covering the past 45 minutes up to the present time in increments of 5 minutes. Read and display the merged layers.

1 Find layers in the WMS Database that include `'mesonet'` and `'nexrad'` in their `ServerURL` fields.

   ```
   mesonet = wmsfind('mesonet*nexrad', 'SearchField', 'serverurl');
   ```

2 NEXRAD Base Reflect Current (`'nexrad-n0r'`) measures the intensity of precipitation. Refine your search to include only layers with this phrase in one of the search fields.

   ```
   nexrad = mesonet.refine('nexrad-n0r', 'SearchField', 'any');
   ```

3 Update your nexrad layer to fill in all fields and obtain most recent data.

```
nexrad = wmsupdate(nexrad, 'AllowMultipleServers', true);
```

4 Remove the 900913 layer because it is intended for Google Maps overlay. Also remove the WMST layer because it contains data for different times.

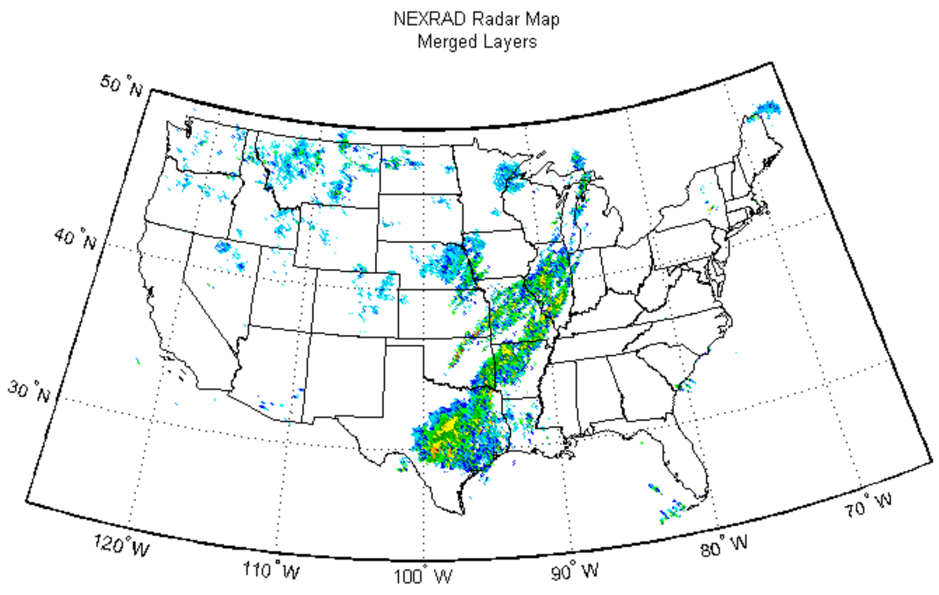
```
index = strcmpi('nexrad-n0r-900913',{nexrad.LayerName});
nexrad(index) = [];
index = strcmpi('nexrad-n0r-wmst',{nexrad.LayerName});
nexrad(index) = [];
```

5 'conus' represents the conterminous 48 U.S. states (all except Hawaii and Alaska). Use the usamap function to construct a map axes for the conterminous states. Read in the nexrad layers.

```
region = 'conus';
figure
usamap(region)
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
[A, R] = wmsread(nexrad, 'Latlim', latlim, 'Lonlim', lonlim, ...
   'Transparent', true);
```

6 Display the NEXRAD merged layers map. Overlay with United States state boundary polygons.

```
geoshow(A, R);
geoshow('usastatehi.shp', 'FaceColor', 'none');
title({'NEXRAD Radar Map', 'Merged Layers'});
```



Courtesy NOAA and Iowa State University

7 Loop through the sequence of time-lapse radar observations.

```
hfig = figure;
usamap(region)
geoshow('usastatehi.shp', 'FaceColor', 'none');
numFrames = numel(nexrad);
frames = struct('cdata', [], 'colormap', []);
frames(numFrames) = frames;
hmap = [];
frameIndex = 0;
for k = numFrames:-1:1
   frameIndex = frameIndex + 1;
   delete(hmap)
   [A, R] = wmsread(nexrad(k), 'Latlim', latlim, 'Lonlim', lonlim);
   hmap = geoshow(A, R);
   title(nexrad(k).LayerName)
   drawnow
   frames(frameIndex) = getframe(hfig);
```

```
end
```

8 Create an array to write out as an animated GIF file.

```
animated(1,1,1,numFrames) = 0;
for k=1:numFrames
   if k == 1
      [animated, cmap] = rgb2ind(frames(k).cdata, 256, 'nodither');
   else
      animated(:,:,1,k) = ...
         rgb2ind(frames(k).cdata, cmap, 'nodither');
      end
   end
```

9 View the animated GIF file.

```
filename = 'wmsnexrad.gif';
imwrite(animated, cmap, filename, 'DelayTime', 1.5, ...
   'LoopCount', inf);
web(filename)
```

Displaying Animation of Radar Images over Daily Planet Backdrop

Display NEXRAD radar images for the past 24 hours, sampled at one-hour intervals, for the United States using data from the IEM WMS server. Use the JPL Daily Planet layer as the backdrop.

1 Find the `'nexrad-n0r-wmst'` layer and update it.

```
wmst = wmsfind('nexrad-n0r-wmst', 'SearchField', 'layername');
wmst = wmsupdate(wmst);
```

2 Find the Daily Planet layer and update it.

```
jpl = wmsfind('jpl.nasa.gov', 'SearchField', 'serverurl');
backdrop = jpl.refine('daily_planet');
backdrop = wmsupdate(backdrop);
```

3 Create a figure with the desired geographic extent.

```
region = 'conus';
```

```
hfig = figure;
usamap(region)
mstruct = gcm;
```

4 Obtain geographic limits and read the backdrop image.

```
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
cellsize = .1;
backdrop = wmsread(backdrop, 'ImageFormat', 'image/png', ...
   'Latlim', latlim, 'Lonlim', lonlim, 'Cellsize', cellsize);
```

5 Calculate current time minus 24 hours and set up frames to hold the data from `getframe`.

```
now_m24 = datestr(now-1);
hour_m24 = [now_m24(1:end-5) '00:00'];
hour = datenum(hour_m24);
hmap = [];
numFrames = 24;
frames = struct('cdata', [], 'colormap', []);
frames(numFrames) = frames;
```

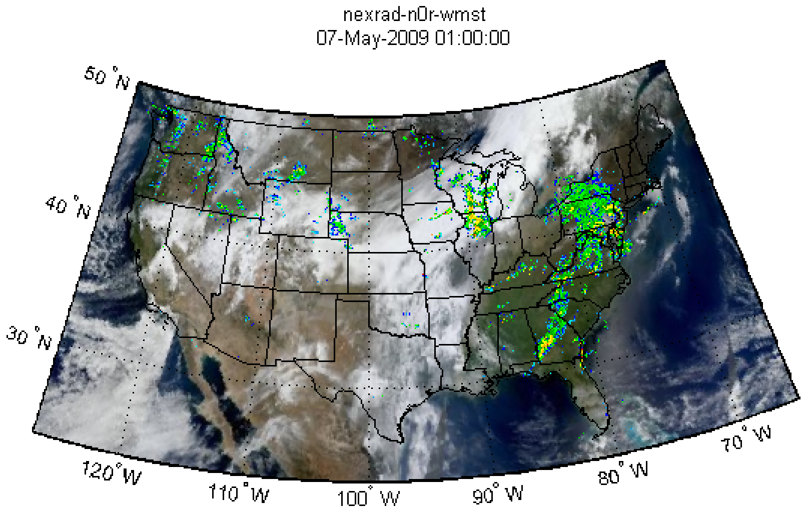
6 For each hour, obtain the hourly NEXRAD map data and combine it with a copy of the backdrop. Because of how this Web server handles PNG format, the resulting map data has an image with class `double`. Thus, you must convert it to `uint8` before merging.

```
geoshow('usastatehi.shp', 'FaceColor', 'none');
black = [0,0,0];
threshold = 0;
for k=1:numFrames
   time = datestr(hour);
   [A, R] = wmsread(wmst, 'Latlim', latlim, 'Lonlim', lonlim, ...
     'Time', time, 'CellSize', cellsize, ...
     'BackgroundColor', black, 'ImageFormat', 'image/png');
   delete(hmap)
   index = any(A > threshold, 3);
   combination = backdrop;
   index = cat(3,index,index,index);
   combination(index) = uint8(255*A(index));
```

```
       hmap = geoshow(combination, R);
       title({wmst.LayerName, time})
       drawnow
       frames(k) = getframe(hfig);
       hour = hour + 1/24;
    end
```

7 View the movie loop.

```
numTimes = 10;
fps = 1.5;
movie(hfig, frames, numTimes, fps);
```



Courtesy NOAA and Iowa State University

Snapshot from NEXRAD Animation

Saving Favorite Servers

You can save your favorite layers for easy access in the future. Use `wmsupdate` to fill in the `Abstract`, `CoordRefSysCodes`, and `Details` fields, and then save the layers. The next example demonstrates how to make a mini-database from the NASA, ESA, and WHOI servers.

1 Find the servers and update all fields.

```
nasa = wmsfind('nasa','SearchField','serverurl');
esa  = wmsfind('esa.int','SearchField','serverurl');
whoi = wmsfind('whoi','SearchField','serverurl');
favoriteLayers = [nasa; esa; whoi];
favoriteLayers = wmsupdate(favoriteLayers, ...
   'AllowMultipleServers', true);
favoriteServers = favoriteLayers.servers;
```

2 Save your favorite layers in a MAT-file.

```
save favorites favoriteLayers
```

3 Search within your favorite layers for `'wind speed'`. You have updated all fields, so you can search within any field, including the `Abstract`.

```
windSpeed = favoriteLayers.refine('wind speed','SearchFields','any')
```

In the following output, the phrase *wind speed* does not occur in the `LayerTitle` or `LayerName` fields, but it does occur in the `Abstract`.

Sample Output:

```
             Index: 14
       ServerTitle: 'NASA SVS Image Server'
         ServerURL: 'http://svs.gsfc.nasa.gov/cgi-bin/wms?'
        LayerTitle: 'Hurricane Dennis (Sequence)'
         LayerName: '3194_22037'
            Latlim: [8.9033 42.1490]
            Lonlim: [-95.7348 -62.7839]
          Abstract: 'The formation of Hurricane Dennis on July 5 ...
made that the earliest date on record that four named storms ...
formed in the Atlantic basin....After re-emerging over open ...
```

```
water, Dennis re-strengthened into a dangerous Category 4 ...
hurricane with top wind speeds of 233 kilometers per hour (145 mph)...
CoordRefSysCodes: {'EPSG:4326'}
         Details: [1x1 struct]
```

Exploring Other Layers from a Server

You may find a layer you like in the WMS Database and then want to find other layers on the same server.

1 Use the `wmsinfo` function to return the contents of the capabilities document as a `WMSCapabilities class` object. A *capabilities document* is an XML document containing metadata describing the geographic content offered by a server.

```
serverURL = 'http://webapps.datafed.net/AQS_H.ogc?';
capabilities = wmsinfo(serverURL);
```

2 View the layer names.

```
capabilities.LayerNames
```

Sample Output:

```
ans =

    'CO'
    'NO2'
    'NOX'
    'NOY'
    'O3'
    'SO2'
    'pm10'
```

3 Read the Carbon Monoxide (`'CO'`) layer.

```
layer = capabilities.Layer.refine('CO');
[A,R] = wmsread(layer,'cellsize',.1,'ImageFormat','image/png');
```

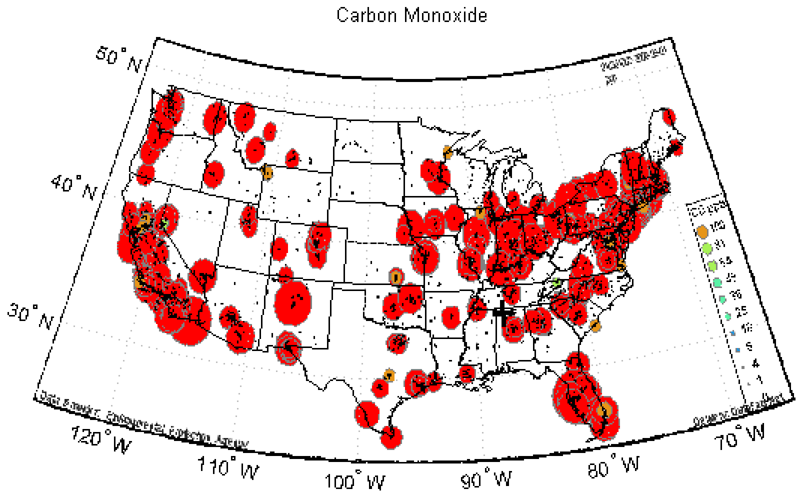
4 Set the longitude and latitude limits to the values specified for the layer.

```
latlim = layer.Latlim;
lonlim = layer.Lonlim;
```

5 Display the map.

```
figure
```

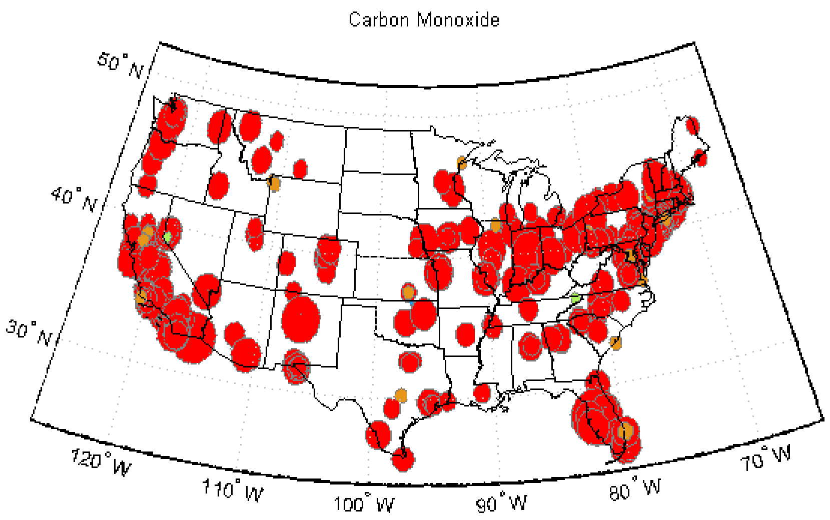
```
usamap(layer.Latlim, layer.Lonlim)
geoshow('usastatehi.shp','FaceColor','none','EdgeColor','black')
geoshow(A,R)
title(layer.LayerTitle)
```



Courtesy EPA

6 Examine the `Style` field. (Open `layer` and then `Details` and then `Style`.) There are two structures. The style of the first one is set to `'data'`. Read the layer again with the `StyleName` set to `'data'` and the cell size set to 0.1 degree resolution. (When the style is set to `'data'`, the map does not include a legend.)

```
[A,R] = wmsread(layer,'cellsize',.1, ...
  'ImageFormat','image/png','StyleName','data');
figure
usamap(layer.Latlim, layer.Lonlim)
geoshow('usastatehi.shp','FaceColor','none','EdgeColor','black')
geoshow(A,R)
title(layer.LayerTitle)
```



Courtesy EPA

Writing a KML File

Some servers, such as the JPL Web map server, render their maps in a nonimage format, such as KML. *KML* is an XML dialect used by Google Earth and Google Maps browsers. The `WMSMapRequest.getMap` method and the `wmsread` function do not allow you to set the KML format because they import only standard graphics image formats. Work around this limitation by using the `WMSMapRequest.RequestURL` property.

1 Search the WMS Database for layers on the JPL server and update these layers. Refine to include only `'landsat'` layers. Landsat satellites take pictures of the Earth from space.

```
jpl = wmsfind('jpl.nasa.gov', 'SearchField', 'serverurl');
jpl = wmsupdate(jpl);
landsat = jpl.refine('landsat');
```

2 Construct a `WMSMapRequest` object and set geographic limits.

```
request = WMSMapRequest(landsat);
request.Latlim = [36.042423, 36.161439];
request.Lonlim = [-113.358918, -113.129789];
```

3 Request the pseudo-color image with infrared and visual bands.

```
request.StyleName = 'pseudo';
```

4 Request an image format that opens in Google Earth.

```
request.ImageFormat = 'application/vnd.google-earth.kml+xml';
```

5 Use the `urlwrite` function to write out a KML file.

```
filename = 'landsat.kml';
urlwrite(request.RequestURL, filename);
```

6 Open the file with Google Earth to view.

Searching for Layers Outside the Database

You can search for layers by using your Web browser rather than by using the WMS Database. For example, this approach allows you to view layers developed more recently than the last software release.

1 To search for layers outside the WMS Database, use your favorite search engine. If you are using Google, select **Images** and enter the following in the search box: `getmap wms`.

2 View the images to choose a map. Click the map link and find the WMS GetCapabilities request somewhere on the page. If you cannot find a GetCapabilities request, try another map.

   For this example, the syntax for the URL of the WMS GetCapabilities request appears as follows:

```
url = ['http://sampleserver1.arcgisonline.com/' ...
   'ArcGIS/services/Specialty/ESRI_StatesCitiesRivers_USA/' ...
   'MapServer/WMSServer?service=WMS&request=GetCapabilities' ...
   '&version=1.3.0'];
```

3 After you obtain the URL, you can use `wmsinfo` to return the capabilities document.

```
c = wmsinfo(url);
```

4 Next, read in a layer and display it as a map.

```
[A,R] = wmsread(c.Layer(1), ...
   'BackgroundColor', [0,0,255], 'ImageFormat', 'image/png');
figure
usamap(c.Layer(1).Latlim, c.Layer(1).Lonlim)
geoshow(A,R)
```

Hosting Your Own WMS Server

You can host your own WMS server and share the maps you create with others. For free software and instructions, see the GeoServer or MapServer Web sites.

Common Problems with WMS Servers

In this section...

Connection Errors

One of the challenges of working with WMS is that sometimes you can have trouble connecting to a server.

Time-Out Error

A server may issue a time-out error such as:

```
Connection timed out: connect
```

Or

```
Read timed out
```

Workaround: Try setting the `'TimeoutInSeconds'` parameter to a larger value. The time-out setting defaults to 60 seconds. (The functions `wmsread`, `wmsinfo`, and `wmsupdate` all have `'TimeoutInSeconds'` parameters.)

System Overload

The NASA Jet Propulsion Laboratory (JPL) server may issue an error message due to server overloading:

```
Service denied due to system overload. Please try again later.
```

Workaround: Try connecting again in a few minutes.

If you fail to connect to the server on a second try, another approach is to request maps from the OnEarth_JPL DataFed Web Map Server. This server hosts most of the layers from the NASA JPL server. However, values for `Style` and `ImageFormats` for a specific layer will be slightly different. With the JPL server, you can request non-scaled data. With the DataFed server, returned data is class `uint8`.

This example compares maps of the same layer from the JPL server and the DataFed server:

```
jpl = wmsfind('jpl','search','serverurl');
jpl.servers'
us_ned = jpl.refine('us_ned')

layer = wmsupdate(us_ned(1));
layer_datafed = wmsupdate(us_ned(2));

latlim = [35, 40];
lonlim = [-110, -105];

% Read and display the layer from the JPL server.
[Z, R] = wmsread(layer, 'ImageFormat', 'image/geotiff', ...
   'StyleName', 'default', 'Latlim', latlim, 'Lonlim', lonlim);
figure
usamap(Z,R)
geoshow(Z,R,'DisplayType', 'texturemap');
demcmap(Z)

% Read and display the layer from the DataFed server.
[Z_datafed, R] = wmsread(layer_datafed, 'ImageFormat', 'GeoTIFF', ...
   'StyleName', 'data', 'Latlim', latlim, 'Lonlim', lonlim);
figure
usamap(Z_datafed,R)
geoshow(Z_datafed,R,'DisplayType','texturemap')
demcmap(Z_datafed)
```

HTTP Response Code 500

In some cases, the server becomes temporarily unavailable or the WMS server application experiences some type of issue. The server issues an HTTP response code of 500, such as:

```
Server returned HTTP response code: 500 for URL: http://xyz.com ...
```

Workaround: Try again later. Also try setting a different `'ImageFormat'` parameter.

WMSServlet Removed

If the `columbo.nrlssc.navy.mil` server issues an error such as:

```
WebMapServer cannot communicate to the host columbo.nrlssc.navy.mil.
The host is unknown.
```

This message indicates that the server it is trying to access is no longer available.

Workaround: Choose a different layer.

Wrong Scale

The `columbo.nrlssc.navy.mil` server often throws this error message:

```
This layer is not visible for this scale. The maximum valid scale
is approximately X. Zoom in and try again if desired. The scale of
the image requested is Y.
```

`X` and `Y` represent specific values that vary from layer to layer.

Workaround: Some of the WMS sources this server accesses have map layers sensitive to the requested scale. Zoom in (choose a smaller region of interest), or zoom out (choose a larger region of interest). Alternatively, you can select a larger output image size to view the layer at the appropriate scale.

Problems with Geographic Limits

Some servers do not follow the guidelines of the OGC specification regarding latitude and longitude limits.

Latlim and Lonlim in Descending Order

The OGC specification requires, and the WMS functions expect, that the limits are ascending. Some sites, however, have descending limits. As a result, you may get this error message:

```
 ??? Error using ==> WMSMapRequest>validateLimit at 1313
Expected the elements of 'Latlim' to be in ascending order.
```

Workaround: To address this problem, set the `Latlim` and `Lonlim` properties of `WMSLayer`:

```
layer = wmsfind('SampleServer.com', 'SearchField', 'serverurl');
layer = wmsupdate(layer);
latlim = [min(layer.Latlim), max(layer.Latlim)];
lonlim = [min(layer.Lonlim), max(layer.Lonlim)];
layer.Latlim = [max([ -90, latlim(1)]), min([ 90, latlim(2)])];
layer.Lonlim = [max([-180, lonlim(1)]), min([180, lonlim(2)])];
[A,R] = wmsread(layer);
```

Update your layer before setting the limits. Otherwise, `wmsread` updates the limits from the server, and you once again have descending limits.

Limits Exceed Bounds

Some servers have limits that exceed the bounds of [-180 180] for longitude and [-90 90] for latitude.

Workaround: To address this problem, follow the same procedure outlined in "Latlim and Lonlim in Descending Order" on page 9-61.

Problems with Server Changing LayerName

In most cases, the updated layer returned by `wmsupdate` should have `ServerURL` and `LayerName` properties that match those of the layer you enter as input. In some cases when the layer is updated from the `columbo.nrlssc.navy.mil` server, the server returns a layer with a different `LayerName`, but the `ServerURL` and `LayerTitle` are the same. The layers from the `columbo.nrlssc.navy.mil` server have names such as `'X:Y'`, where `X` and `Y` are ASCII numbers. Since the time of your last update, a layer has been added to or removed from the server causing a shift in the sequence of layers. Since the `LayerName` property is constructed with ASCII numbers based on the layer's position in this sequence, the `LayerName` property has changed. For layers from the `columbo.nrlssci.navy.mil` server, `wmsupdate` matches the `LayerTitle` property rather than the `LayerName` property.

Non-EPSG:4326 Coordinate Reference Systems

Some layers, such as the CubeWerx Mars Topography Layer, do not use the EPSG:4326 coordinate reference system.

Workaround: The toolbox uses the first available coordinate reference system code. See "Understanding Coordinate Reference System Codes" on page 9-15 for more information.

Map Not Returned

Sometimes you can connect to the WMS server, but you do not receive the map you are expecting.

Blank Map Returned

A server may return a blank map.

Workaround: You can change the scale of your map; either increase the image height and width or change the geographic bounds. Another possibility is that your requested geographic extent lies outside the extent of the layer, in which case you should change the extent of your request. A third possibility is that you have the wrong image format selected; in this case, change the `'ImageFormat'` parameter.

HTML File Returned

You may receive this error message:

```
The server returned an HTML file instead of an image file.
```

Workaround: Follow the directions in the error message. The following example, which uses a sample URL, illustrates the type of error message you receive.

```
% Example command.
>> [A,R] = wmsread('http://www.mathworks.com?&BBOX=-180,-90,180,90...
    &CRS=EPSG:4326');
```

Sample error message:

```
??? Error using ==> WebMapServer>issueReadGetMapError at 832
The server returned an HTML file instead of an image file.
You may view the complete error message by issuing the command,
 web('http://www.mathworks.com?&BBOX=-180,-90,180,90&CRS=EPSG:4326')
 or
 urlread('http://www.mathworks.com?&BBOX=-180,-90,180,90...
      &CRS=EPSG:4326').
```

XML File Returned

The server issues a very long error message, beginning with the following phrase:

```
An error occurred while attempting to get the map from the server.
The error returned is <?xml version="1.0" encoding="utf-8"?> ...
```

Workaround: This problem occurs because the server breaks with the requirements of the OGC standard and returns the XML capabilities document rather than the requested map. Choose a different layer or server.

Unsupported WMS Version

In rare cases, the server uses a different and unsupported WMS version. In this case, you receive an error message such as:

```
The WMS version, '1.2.0', listed in layer.Details.Version is not
supported by the server. The supported versions are: '1.0.0' '1.1.0'
'1.1.1' '1.3.0' .
```

Workaround: Choose a different server.

Other Unrecoverable Server Errors

The server issues an error indicating that no correction or workaround exists. These cases result in the following types of error messages:

```
Server redirected too many  times (20)
```

```
An error occurred while attempting to parse the XML capabilities
document from the server.

Unexpected end of file from server

An error occurred while attempting to get the map from the server.
The server returned a map containing no data.
```

Mapping Applications

This chapter describes several types of numerical applications for geospatial data, including computing and spatial statistics, and calculating tracks, routes, and other information useful for solving navigation problems.

Geographic Statistics

In this section...

Statistics for Point Locations on a Sphere

Certain Mapping Toolbox functions compute basic geographical measures for spatial analysis and for filtering and conditioning data. Since MATLAB functions can compute statistics such as means, medians, and variances, why not use those functions in the toolbox? First of all, classical statistical formulas typically assume that data is one-dimensional (and, often, normally distributed). Because this is not true for geospatial data, spatial analysts have developed statistical measures that extend conventional statistics to higher dimensions.

Second, such formulas generally assume that data occupies a two-dimensional Cartesian coordinate system. Computing statistics for geospatial data with geographic coordinates as if it were in a Cartesian framework can give statistically inappropriate results. While this assumption can sometimes yield reasonable numerical approximations within small geographic regions, for larger areas it can lead to incorrect conclusions because of distance measures and area assumptions that are inappropriate for spheres and spheroids. Mapping Toolbox functions appropriately compute statistics for geospatial data, avoiding these potential pitfalls.

Geographic Means

Consider the problem of calculating the mean position of a collection of geographic points. Taking the arithmetical mean of the latitudes and longitudes using the standard MATLAB `mean` function may seem reasonable, but doing this could yield misleading results.

Take two points at the same latitude, 180º apart in longitude, for example (30ºN,90ºW) and (30ºN,90ºE). The *mean* latitude is (30+30)/2=30, which seems right. Similarly, the mean longitude must be (90+(-90))/2=0. However, as one can also express 90ºW as 270ºE, (90+270)/2=180 is also a valid mean longitude. Thus there are two correct answers, the prime meridian and the dateline. This demonstrates how the sphericity of the Earth introduces subtleties into spatial statistics.

This problem is further complicated when some points are at different latitudes. Because a degree of longitude at the Arctic Circle covers a much smaller distance than a degree at the equator, distance between points having a given difference in longitude varies by latitude.

Is in fact 30ºN the right mean latitude in the first example? The mean position of two points should be equidistant from those two points, and should also minimize the total distance. Does (30ºN,0º) satisfy these criteria?

```
dist1 = distance(30,90,30,0)
dist1 =
 75.5225
dist2 = distance(30,-90,30,0)
dist2 =
 75.5225
```

Consider a third point, (`lat`,`lon`), that is also equidistant from the above two points, but at a lesser distance:

```
dist1 = distance(30,90,lat,lon)
dist1 =
 60.0000
dist2 = distance(30,-90,lat,lon)
dist2 =
 60.0000
```

What is this mystery point? The `lat` is 90ºN, and any `lon` will do. The North Pole is the true geographic mean of these two points. Note that the great circle containing both points runs through the North Pole (a great circle represents the shortest path between two points on a sphere).

The Mapping Toolbox function meanm determines the geographic mean of any number of points. It does this using three-dimensional vector addition of all the points. For example, try the following:

```
lats = [30 30];
longs = [-90 90];
[latbar,longbar] = meanm(lats,longs)
latbar =
 90
longbar =
 0
```

This is the answer you now expect. This geographic mean can result in one oddity; if the vectors all cancel each other, the mean is the center of the planet. In this case, the returned mean point is (NaN,NaN) and a warning is displayed. This phenomenon is highly improbable in *real* data, but can be easily constructed. For example, it occurs when all the points are equally spaced along a great circle. Try taking the geographic mean of (0º,0º), (0º,120º), and (0º,240º), which trisect the equator.

```
elats = [0 0 0];
elons = [60 120 240];
meanm(elats, elons)
ans =
         0  120.0000
```

Geographic Standard Deviation

As you might now expect, the Cartesian definition of standard deviation provided in the standard MATLAB function std is also inappropriate for geographic data that is unprojected or covers a significant portion of a planet. Depending upon your purpose, you might want to use the separate geographic deviations for latitude and longitude provided by the function stdm, or the single standard distance provided in stdist. Both methods measure the deviation of points from the mean position calculated by meanm.

The Meaning of stdm

The stdm function handles the latitude and longitude deviations separately.

```
[latstd,lonstd] = stdm(lat,lon)
```

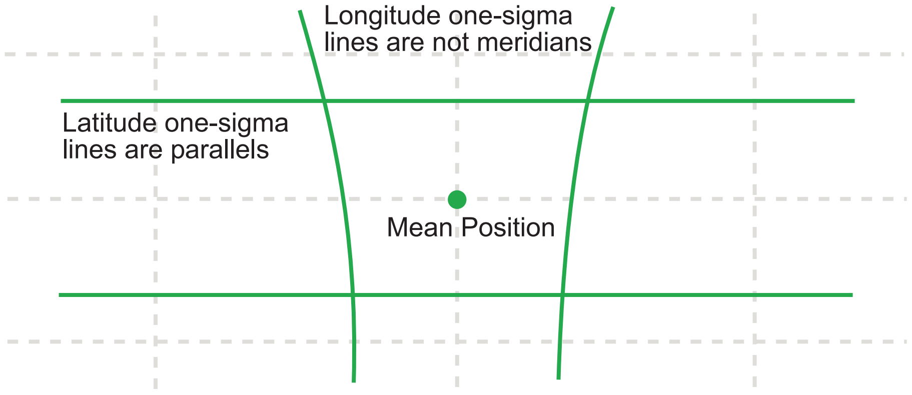
The function returns two deviations, one for latitudes and one for longitudes.

Latitude deviation is a straightforward standard deviation calculation from the mean latitude (mean parallel) returned by `meanm`. This is a reasonable measure for most cases, since on a sphere at least, a degree of latitude always has the same arc length.

Longitude deviation is another matter. Simple calculations based on sum-of-squares angular deviation from the mean longitude (mean meridian) are misleading. The arc length represented by a degree of longitude at extreme latitudes is significantly smaller than that at low latitudes.

The term *departure* is used to represent the arc length distance along a parallel of a point from a given meridian. For example, assuming a spherical planet, the departure of a degree of longitude at the Equator is a degree of arc length, but the departure of a degree of longitude at a latitude of 60º is one-half a degree of arc length. The `stdm` function calculates a sum-of-squares departure deviation from the mean meridian.

If you want to plot the one-sigma lines for `stdm`, the latitude sigma lines are parallels. However, the longitude sigma lines are not meridians; they are lines of constant departure from the mean parallel.


Longitude one-sigma lines are not meridians
Latitude one-sigma lines are parallels
Mean Position

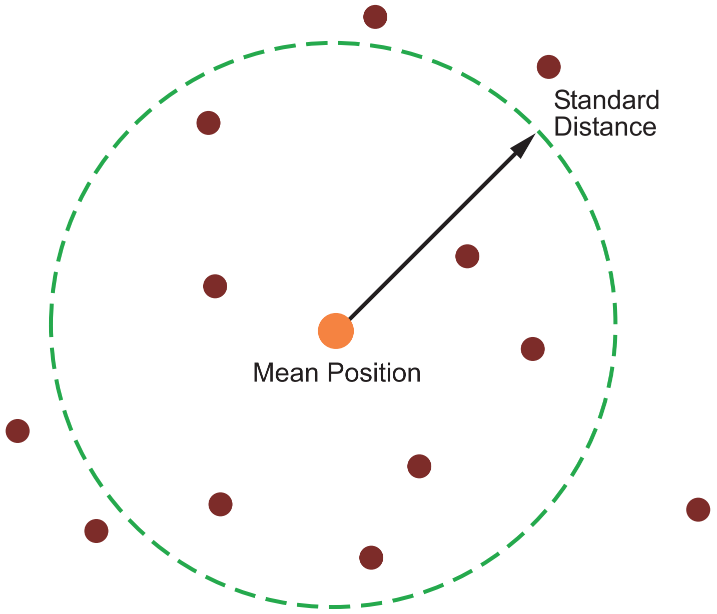
This handling of deviation has its problems. For example, its dependence upon the logic of the coordinate system can cause it to break down near the poles. For this reason, the standard distance provided by `stdist` is often a better measure of deviation. The `stdm` handling is useful for many applications, especially when the data is not global. For instance, these potential difficulties would not be a danger for data points confined to the country of Mexico.

The Meaning of stdist

The standard distance of geographic data is a measure of the dispersion of the data in terms of its distance from the geographic mean. Among its advantages are its applicability anywhere on the globe and its single value:

```
dist = stdist(lat,lon)
```

In short, the standard distance is the average, norm, or *cubic norm* of the distances of the data points in a great circle sense from the mean position. It is probably a superior measure to the two deviations returned by `stdm` except when a particularly latitude- or longitude-dependent feature is under examination.



Equal-Areas in Geographic Statistics

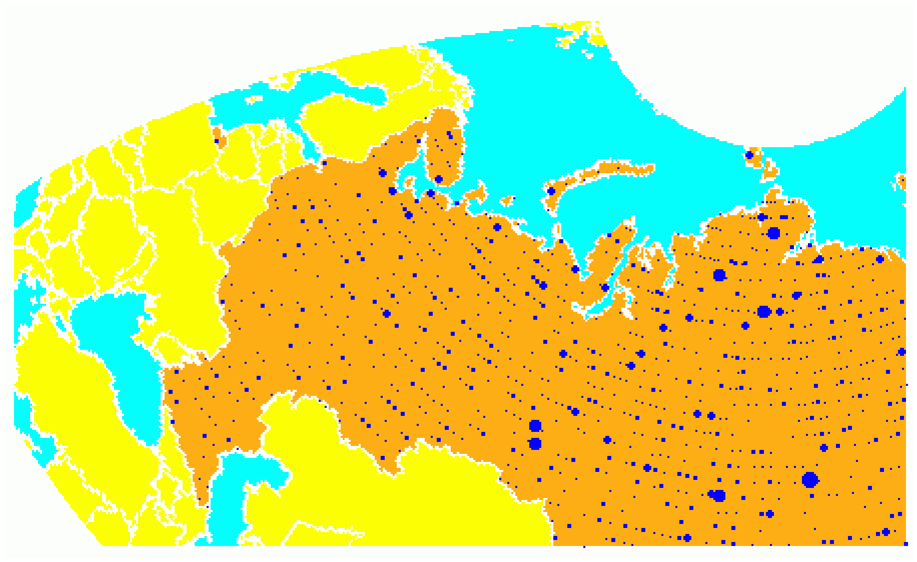
A common error in applying two-dimensional statistics to geographic data lies in ignoring equal-area treatment. It is often necessary to *bin* data to statistically analyze it. In a Cartesian plane, this is easily done by dividing the space into equal *x-y* squares. The geographic equivalent of this is to bin up the data in equal latitude-longitude *squares*. Since such squares at high latitudes cover smaller areas than their low-latitude counterparts, the observations in these regions are underemphasized. The result can be conclusions that are biased toward the equator.

Geographic Histograms

The geographic histogram function `histr` allows you to display *binned-up* geographic observations. The `histr` function results in equirectangular binning. Each bin has the same angular measurement in both latitude and longitude, with a default measurement of 1 degree. The center latitudes and longitudes of the bins are returned, as well as the number of observations per bin:

```
[binlat,binlon,num] = histr(lats,lons)
```

As previously noted, these equirectangular bins result in counting bias toward the equator. Here is a display of the one-degree-by-one-degree binning of approximately 5,000 random data points in Russia. The relative size of the circles indicates the number of observations per bin:

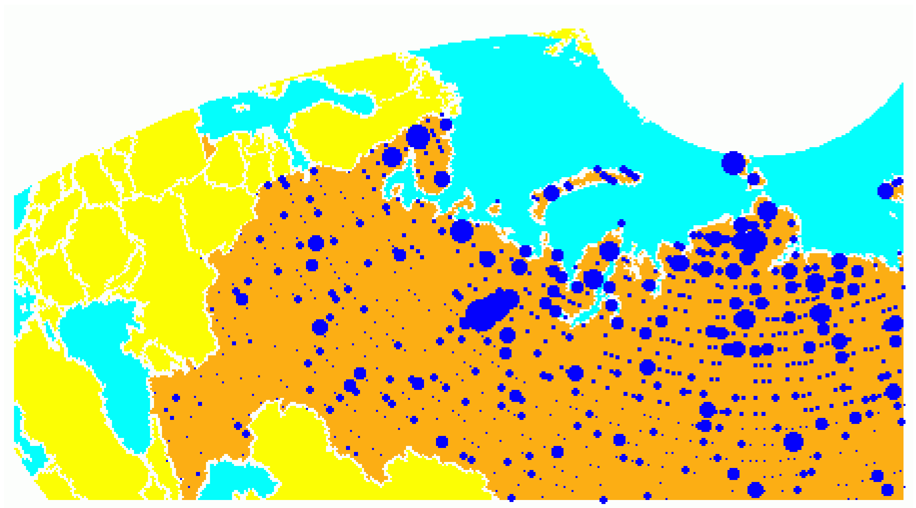


This is a portion of the whole map, displayed in an equal-area Bonne projection. The first step in creating data displays without area bias is to choose an equal-area projection. The proportionally sized symbols are a result of the specialized display function `scatterm`.

You can eliminate the area bias by adding a fourth output argument to `histr`, that will be used to weight each bin's observation by that bin's area:

```
[binlat,binlon,num,wnum] = histr(lats,lons)
```

The fourth output is the weighted observation count. Each bin's observation count is divided by its normalized area. Therefore, a high-latitude bin will have a larger weighted number than a low-latitude bin with the same number of actual observations. The same data and bins look much different when they are area-weighted:



Notice that there are larger symbols to the north in this display. The previous display suggested that the data was relatively uniformly distributed. When equal-area considerations are included, it is clear that the data is skewed to the north. In fact, the data used is northerly skewed, but a simple equirectangular handling failed to demonstrate this.

The `histr` function, therefore, does provide for the display of area-weighted data. However, the actual bins used are of varying areas. Remember, the one-degree-by-one-degree bin near a pole is much smaller than its counterpart near the equator.

The `hista` function provides for actual equal-area bins.

Converting to an Equal-Area Coordinate System

The actual data itself can be converted to an equal-area coordinate system for analysis with other statistical functions. It is easy to convert a collection of geographic latitude-longitude points to an equal-area *x*-*y* Cartesian coordinate system. The `grn2eqa` function applies the same transformation used in calculating the Equal-Area Cylindrical projection:

```
[x,y] = grn2eqa(lat,lon)
```

For each geographic `lat` - `lon` pair, an equal-area `x` - `y` is returned. The variables `x` and `y` can then be operated on under the equal-area assumption, using a variety of two-dimensional statistical techniques. Tools for such analysis can be found in the Statistics Toolbox™ software and elsewhere. The results can then be converted back to geographic coordinates using the `eqa2grn` function:

```
[lat,lon] = eqa2grn(x, y)
```

Remember, when converting back and forth between systems, latitude corresponds to *y* and longitude corresponds to *x*.

Navigation

In this section...

What Is Navigation?

Navigation is the process of planning, recording, and controlling the movement of a craft or vehicle from one location to another. The word derives from the Latin roots *navis* ("ship") and *agere* ("to move or direct"). Geographic information—usually in the form of latitudes and longitudes—is at the core of navigation practice. The toolbox includes specialized functions for navigating across expanses of the globe, for which projected coordinates are of limited use.

Navigating on land, over water, and through the air can involve a variety of tasks:

- Establishing position, using known, fixed landmarks (piloting)
- Using the stars, sun, and moon (celestial navigation)
- Using technology to fix positions (inertial guidance, radio beacons, and satellite navigation, including GPS)
- Deducing net movement from a past known position (dead reckoning)

Another navigational task involves planning a voyage or flight, which includes determining an efficient route (usually by great circle approximation), weather avoidance (optimal track routing), and setting out a plan of intended movement (track laydown). Mapping Toolbox functions support these navigational activities as well.

Conventions for Navigational Functions

Units

You can use and convert among several angular and distance measurement units. The navigational support functions are

- `dreckon`
- `gcwaypts`
- `legs`
- `navfix`

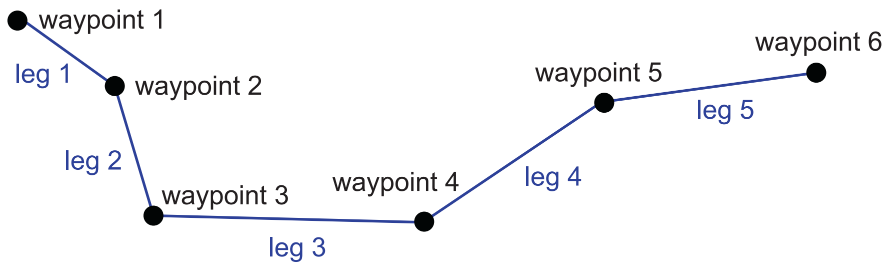
To make these functions easy to use, and to conform to common navigational practice, *for these specific functions only*, certain conventions are used:

- Angles are always in degrees.
- Distances are always in nautical miles.
- Speeds are always in knots (nautical miles per hour).

Related functions that *do not* carry this restriction include `rhxrh`, `scxsc`, `gcxgc`, `gcxsc`, `track`, `timezone`, and `crossfix`, because of their potential for application outside navigation.

Navigational Track Format

Navigational track format requires column-vector variables for the latitudes and longitudes of track waypoints. A *waypoint* is a point through which a track passes, usually corresponding to a course (or speed) change. Navigational tracks are made up of the line segments connecting these waypoints, which are called *legs*. In this format, therefore, *n* legs are described using *n+1* waypoints, because an endpoint for the final leg must be defined. Mapping Toolbox navigation functions always presume angle units are always given in degrees.



Here, five track legs require six waypoints. In navigational track format, the waypoints are represented by two 6-by-1 vectors, one for the latitudes and one for the longitudes.

Fixing Position

The fundamental objective of navigation is to determine at a given moment how to proceed to your destination, avoiding hazards on the way. The first step in accomplishing this is to establish your current position. Early sailors kept within sight of land to facilitate this. Today, navigation within sight (or radar range) of land is called *piloting*. Positions are fixed by correlating the bearings and/or ranges of landmarks. In real-life piloting, all sighting bearings are treated as rhumb lines, while in fact they are actually great circles.

Over the distances involved with visual sightings (up to 20 or 30 nautical miles), this assumption causes no measurable error and it provides the significant advantage of allowing the navigator to plot all bearings as straight lines on a Mercator projection.

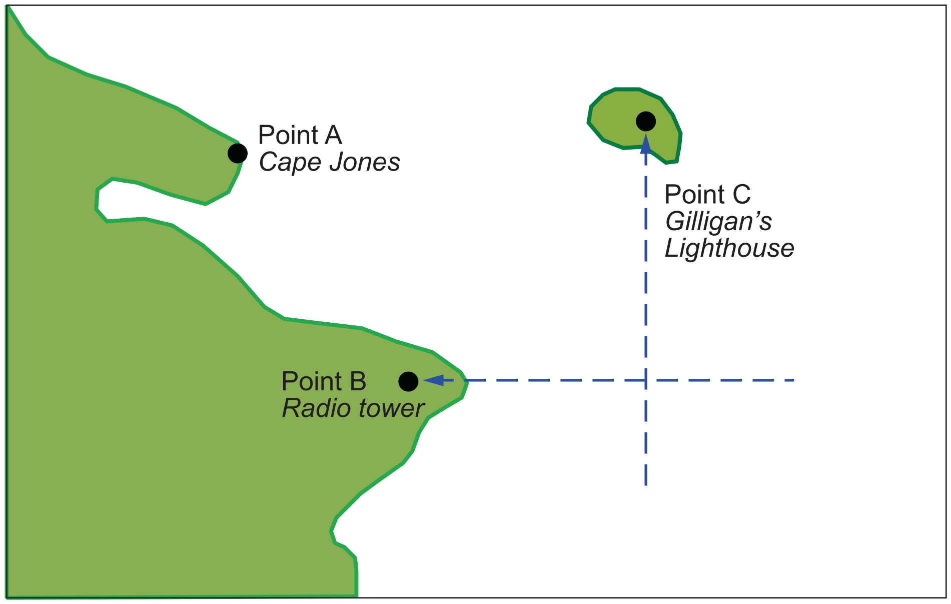
The Mercator was designed exactly for this purpose. Range circles, which might be determined with a radar, are assumed to plot as true circles on a Mercator chart. This allows the navigator to manually draw the range arc with a compass.

These assumptions also lead to computationally efficient methods for fixing positions with a computer. The toolbox includes the `navfix` function, which mimics the manual plotting and fixing process using these assumptions.

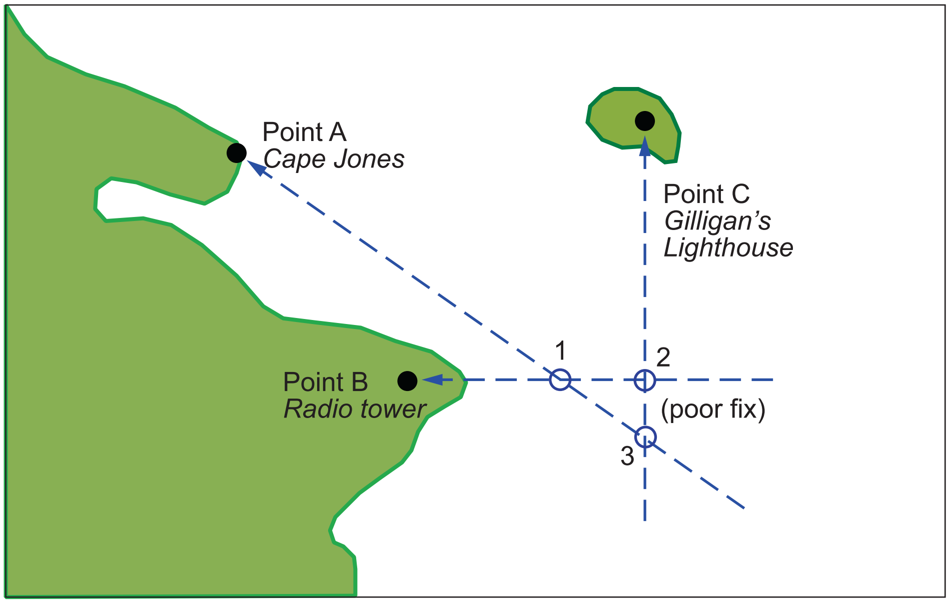
To obtain a good navigational fix, your relationship to at least three known points is considered necessary. A questionable or poor fix can be obtained with two known points.

Some Possible Situations

In this imaginary coastal region, you take a visual bearing on the radio tower of 270º. At the same time, Gilligan's Lighthouse bears 0º. If you plot a 90º-270º line through the radio tower and a 0º-180º line through the lighthouse on your Mercator chart, the point at which the lines cross is a fix. Since you have used only two lines, however, its quality is questionable.



But wait; your port lookout says he took a bearing on Cape Jones of 300º. If that line exactly crosses the point of intersection of the first two lines, you will have a perfect fix.



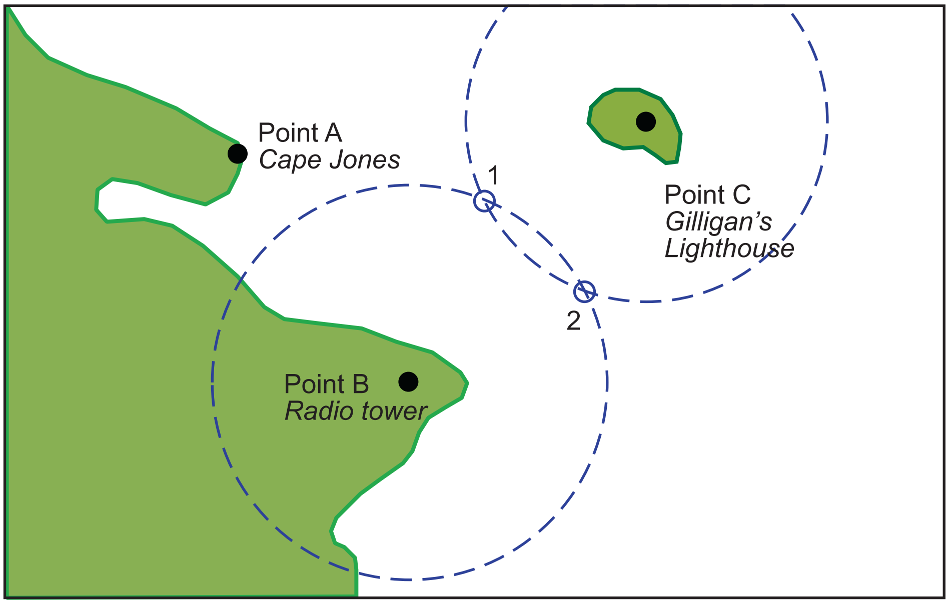
Whoops. What happened? Is your lookout in error? Possibly, but perhaps one or both of your bearings was slightly in error. This happens all the time. Which point, 1, 2, or 3, is correct? As far as you know, they are all equally valid.

In practice, the little triangle is plotted, and the fix position is taken as either the center of the triangle or the vertex closest to a danger (like shoal water). If the triangle is large, the quality is reported as *poor*, or even as *no fix*. If a fourth line of bearing is available, it can be plotted to try to resolve the ambiguity. When all three lines appear to cross at exactly the same point, the quality is reported as *excellent* or *perfect*.

Notice that three lines resulted in three intersection points. Four lines would return six intersection points. This is a case of combinatorial counting. Each intersection corresponds to choosing two lines to intersect from among n lines.

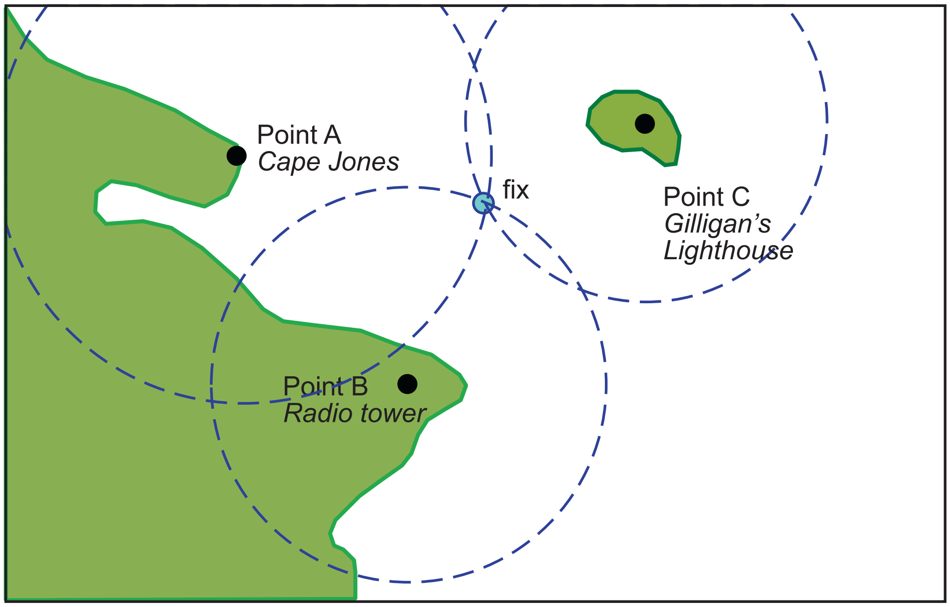
The next time you traverse these straits, it is a very foggy morning. You can't see any landmarks, but luckily, your navigational radar is operating. Each of these landmarks has a good radar signature, so you're not worried.

You get a range from the radio tower of 14 nautical miles and a range from the lighthouse of 15 nautical miles.



Now what? You took ranges from only two objects, and yet you have two possible positions. This ambiguity arises from the fact that circles can intersect twice.

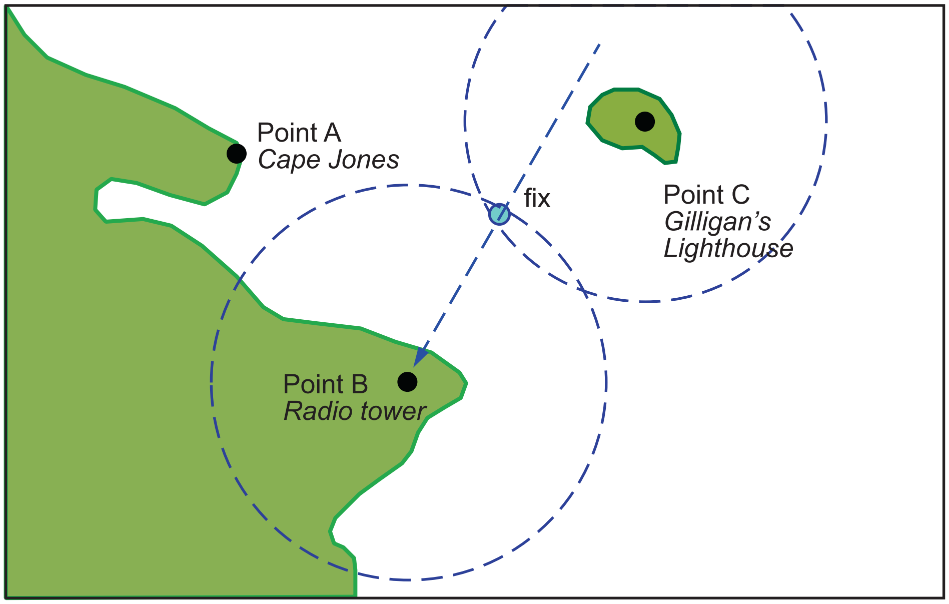
Luckily, your radar watch reports that he has Cape Jones at 18 nautical miles. This should resolve everything.



You were lucky this time. The third range resolved the ambiguity and gave you an excellent fix. Three intersections practically coincide. Sometimes the ambiguity is resolved, but the fix is still poor because the three closest intersections form a sort of circular triangle.

Sometimes the third range only adds to the confusion, either by bisecting the original two choices, or by failing to intersect one or both of the other arcs at all. In general, when *n* arcs are used, 2x(*n*-choose-2) possible intersections result. In this example, it is easy to tell which ones are *right*.

Bearing lines and arcs can be combined. If instead of reporting a third range, your radar watch had reported a bearing from the radar tower of 20º, the ambiguity could also have been resolved. Note, however, that in practice, lines of bearing for navigational fixing should only be taken visually, except in desperation. A radar's beam width can be a degree or more, leading to uncertainty.



As you begin to wonder whether this manual plotting process could be automated, your first officer shows up on the bridge with a laptop and Mapping Toolbox software.

Using navfix

The `navfix` function can be used to determine the points of intersection among any number of lines and arcs. Be warned, however, that due to the combinatorial nature of this process, the computation time grows rapidly with the number of objects. To illustrate this function, assign positions to the landmarks. Point A, Cape Jones, is at (`latA`,`lonA`). Point B, the radio tower, is at (`latB`,`lonB`). Point C, Gilligan's Lighthouse, is at (`latC`,`lonC`).

For the bearing-lines-only example, the syntax is:

```
[latfix,lonfix] = navfix([latA latB latC],[lonA lonB lonC],...
                         [300 270 0])
```

This defines the three points and their bearings as taken *from the ship*. The outputs would look something like this, with actual numbers, of course:

```
latfix =
 latfix1       NaN          % A intersecting B
 latfix2       NaN          % A intersecting C
 latfix3       NaN          % B intersecting C
lonfix =
 lonfix1       NaN          % A intersecting B
 lonfix2       NaN          % A intersecting C
 lonfix3       NaN          % B intersecting C
```

Notice that these are two-column matrices. The second column consists of `NaN`s because it is used only for the two-intersection ambiguity associated with arcs.

For the range-arcs-only example, the syntax is

```
[latfix,lonfix] = navfix([latA latB latC],[lonA lonB lonC],...
                         [16 14 15],[0 0 0])
```

This defines the three points and their ranges as taken from the ship. The final argument indicates that the three cases are all ranges.

The outputs have the following form:

```
latfix =
 latfix11  latfix12          % A intersecting B
 latfix21  latfix22          % A intersecting C
 latfix31  latfix32          % B intersecting C
lonfix =
 lonfix11  lonfix12          % A intersecting B
 lonfix21  lonfix22          % A intersecting C
 lonfix31  lonfix32          % B intersecting C
```

Here, the second column is used, because each pair of arcs has two potential intersections.

For the bearings and ranges example, the syntax requires the final input to indicate which objects are lines of bearing (indicated with a 1) and which are range arcs (indicated with a 0):

```
[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                         [20 14 15],[1 0 0])
```

The resulting output is mixed:

```
latfix =
 latfix11      NaN          % Line B intersecting Arc B
 latfix21  latfix22         % Line B intersecting Arc C
 latfix31  latfix32         % Arc B intersecting Arc C
lonfix =
 lonfix11      NaN          % Line B intersecting Arc B
 lonfix21  lonfix22         % Line B intersecting Arc C
 lonfix31  lonfix32         % Arc B intersecting Arc C
```

Only one intersection is returned for the line from B with the arc about B, since the line originates inside the circle and intersects it once. The same line intersects the other circle twice, and hence it returns two points. The two circles taken together also return two points.

Usually, you have an idea as to where you are before you take the fix. For example, you might have a dead reckoning position for the time of the fix (see below). If you provide `navfix` with this estimated position, it chooses from each pair of ambiguous intersections the point closest to the estimate. Here's what it might look like:

```
[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                         [20 14 15],[1 0 0],drlat,drlon)
latfix =
 latfix11                   % the only point
 latfix21                   % the closer point
 latfix31                   % the closer point
lonfix =
 lonfix11                   % the only point
 lonfix21                   % the closer point
 lonfix31                   % the closer point
```

A Numerical Example of Using navfix

1 Define some specific points in the middle of the Atlantic Ocean. These are strictly arbitrary; perhaps they correspond to points in Atlantis:

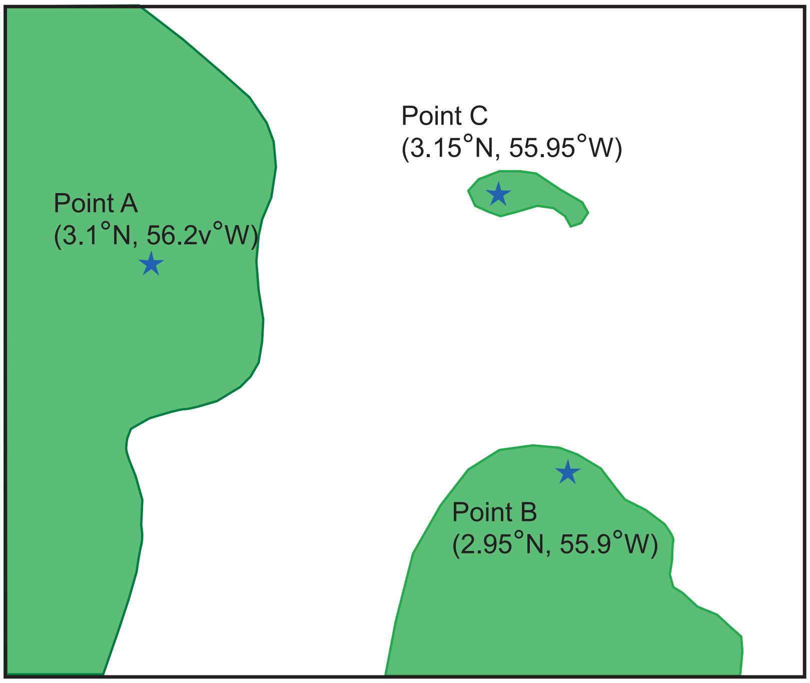
```
lata = 3.1;  lona = -56.2;
latb = 2.95; lonb = -55.9;
```

```
latc = 3.15; lonc = -55.95;
```

2 Plot them on a Mercator projection:

```
axesm('MapProjection','mercator','Frame','on',...
 'MapLatLimit',[2.8 3.3],'MapLonLimit',[-56.3 -55.8])
plotm([lata latb latc],[lona lonb lonc],...
 'LineStyle','none','Marker','pentagram',...
 'MarkerEdgeColor','b','MarkerFaceColor','b',...
 'MarkerSize',12)
```

Here is what it looks like (the labeling and imaginary coastlines are added after the fact for illustration).



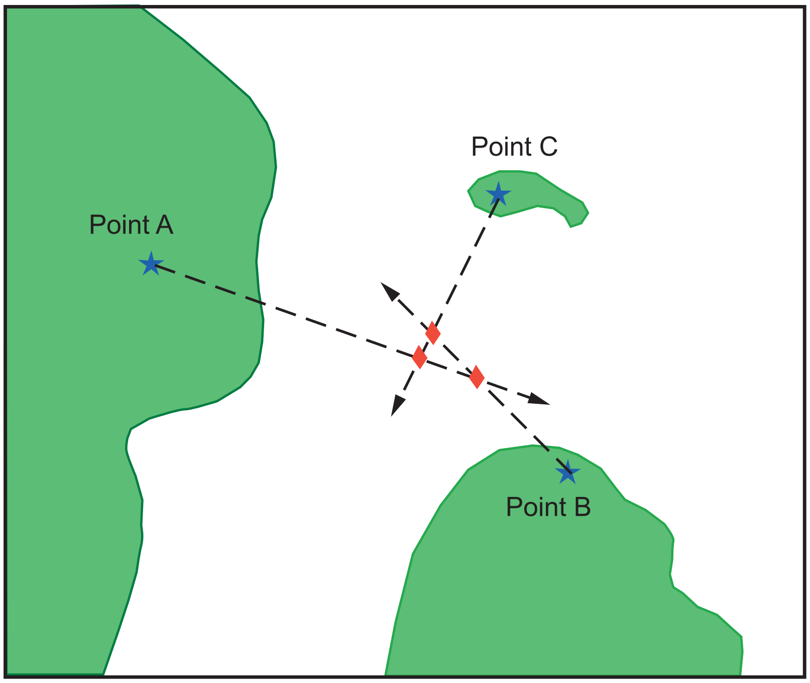
3 Take three visual bearings: Point A bears 289º, Point B bears 135º, and Point C bears 026.5º. Calculate the intersections:

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [289 135 26.5],[1 1 1])
```

```
newlat =
 3.0214       NaN
 3.0340       NaN
 3.0499       NaN
newlong =
 -55.9715       NaN
 -56.0079       NaN
 -56.0000       NaN
```

4 Add the bearing lines and intersection points to the map:

```
plotm(newlat,newlong,'LineStyle','none',...
 'Marker','diamond','MarkerEdgeColor','r',...
 'MarkerFaceColor','r','MarkerSize',9)
```

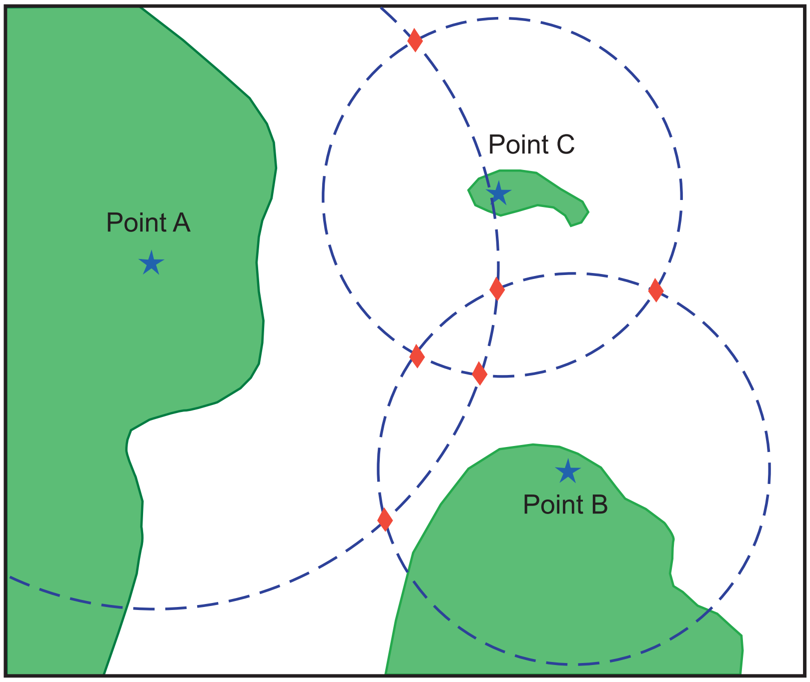


Notice that each pair of objects results in only one intersection, since all are lines of bearing.

5 What if instead, you had ranges from the three points, A, B, and C, of 13 nmi, 9 nmi, and 7.5 nmi, respectively?

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [13 9 7.5],[0 0 0])
newlat =
 3.0739    2.9434
 3.2413    3.0329
 3.0443    3.0880
newlong =
 -55.9846  -56.0501
 -56.0355  -55.9937
 -56.0168  -55.8413
```

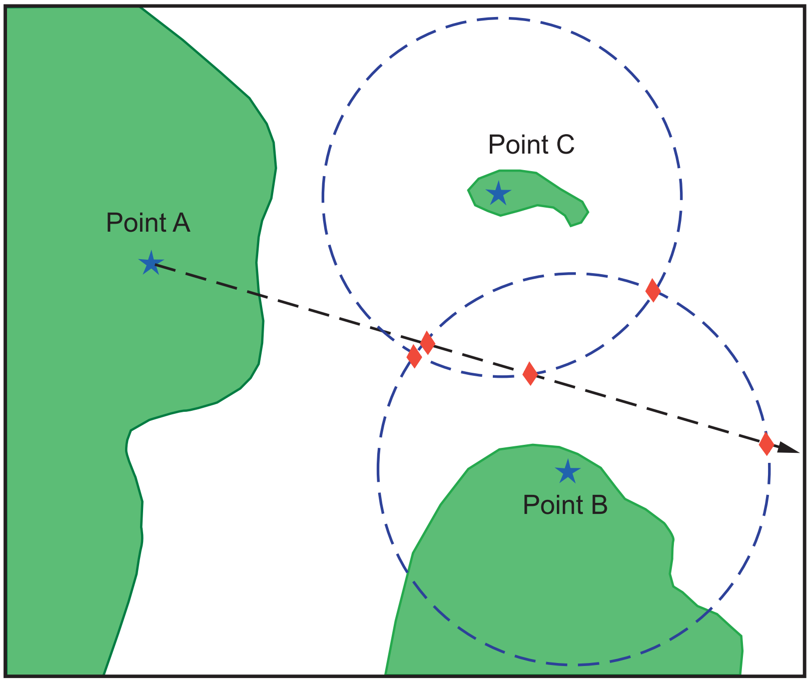
Here's what these points look like:



Three of these points look reasonable, three do not.

6 What if, instead of a range from Point A, you had a bearing to it of 284º?

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [284 9 7.5],[1 0 0])
newlat =
 3.0526    2.9892
 3.0592    3.0295
 3.0443    3.0880
newlong =
 -56.0096  -55.7550
 -56.0360  -55.9168
 -56.0168  -55.8413
```



Again, visual inspection of the results indicates which three of the six possible points seem like *reasonable* positions.

7 When using the dead reckoning position (3.05ºN,56.0ºW), the closer, more reasonable candidate from each pair of intersecting objects is chosen:

```
drlat = 3.05; drlon = -56;
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [284 9 7.5],[1 0 0],drlat,drlon)
newlat =
 3.0526
 3.0592
 3.0443
newlong =
 -56.0096
 -56.0360
 -56.0168
```

Planning

You know that the shortest path between two geographic points is a great circle. Sailors and aviators are interested in minimizing distance traveled, and hence time elapsed. You also know that the rhumb line is a path of constant heading, the *natural* means of traveling. In general, to follow a great circle path, you would have to continuously alter course. This is impractical. However, you can approximate a great circle path by rhumb line segments so that the added distance is minor and the number of course changes minimal.

Surprisingly, very few rhumb line *track legs* are required to closely approximate the distance of the great circle path.

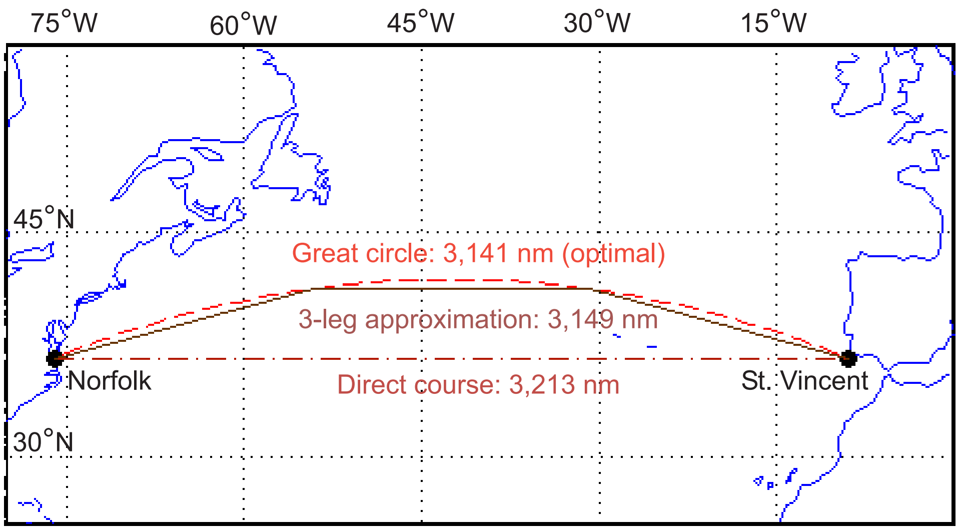
Consider the voyage from Norfolk, Virginia (37ºN,76ºW), to Cape St. Vincent, Portugal (37ºN,9ºW), one of the most heavily trafficked routes in the Atlantic. A due-east rhumb line track is 3,213 nautical miles, while the optimal great circle distance is 3,141 nautical miles.

Although the rhumb line path is only a little more than 2% longer, this is an additional 72 miles over the course of the trip. For a 12-knot tanker, this results in a 6-hour delay, and in shipping, time is money. If just three rhumb line segments are used to approximate the great circle, the total distance of the trip is 3,147 nautical miles. Our tanker would suffer only a half-hour delay compared to a continuous rhumb line course. Here is the code for computing the three types of tracks between Norfolk and St. Vincent:

```
figure('color','w');
ha = axesm('mapproj','mercator',...
           'maplatlim',[25 55],'maplonlim',[-80 0]);
```

```
axis off, gridm on, framem on;
setm(ha,'MLineLocation',15,'PLineLocation',15);
mlabel on, plabel on;
load coast;
hg = geoshow(lat,long,'displaytype','line','color','b');
% Define point locs for Norfolk, VA and St. Vincent Portugal
norfolk = [37,-76];
stvincent = [37, -9];
geoshow(norfolk(1),norfolk(2),'DisplayType','point',...
    'markeredgecolor','k','markerfacecolor','k','marker','o')
geoshow(stvincent(1),stvincent(2),'DisplayType','point',...
    'markeredgecolor','k','markerfacecolor','k','marker','o')
% Compute and draw 100 points for great circle
gcpts = track2('gc',norfolk(1),norfolk(2),...
               stvincent(1),stvincent(2));
geoshow(gcpts(:,1),gcpts(:,2),'DisplayType','line',...
    'color','red','linestyle','--')
% Compute and draw 100 points for rhumb line
rhpts = track2('rh',norfolk(1),norfolk(2),...
                stvincent(1),stvincent(2));
geoshow(rhpts(:,1),rhpts(:,2),'DisplayType','line',...
    'color',[.7 .1 0],'linestyle','-.')
[latpts,lonpts] = gcwaypts(norfolk(1),norfolk(2),...
    stvincent(1),stvincent(2),3);   % Compute 3 waypoints
geoshow(latpts,lonpts,'DisplayType','line',...
    'color',[.4 .2 0],'linestyle','-')
```

The resulting tracks and distances are shown below:



The Mapping Toolbox function `gcwaypts` calculates waypoints in navigation track format in order to approximate a great circle with rhumb line segments. It uses this syntax:

```
[latpts,lonpts] = gcwaypts(lat1,lon1,lat2,lon2,numlegs)
```

All the inputs for this function are scalars a (starting and an ending position). The `numlegs` input is the number of equal-length legs desired, which is 10 by default. The outputs are column vectors representing waypoints in navigational track format (`[heading distance]`). The size of each of these vectors is `[(numlegs+1) 1]`. Here are the points for this example:

```
[latpts,lonpts] = gcwaypts(norfolk(1),norfolk(2),...
stvincent(1),stvincent(2),3)   % Compute 3 waypoints
latpts =
   37.0000
   41.5076
   41.5076
   37.0000

lonpts =
  -76.0000
  -54.1777
  -30.8223
   -9.0000
```

These points represent waypoints along the great circle between which the approximating path follows rhumb lines. Four points are needed for three legs, because the final point at Cape St. Vincent must be included.

Now we can compute the distance in nautical miles (nm) along each track and via the waypoints:

```
drh = distance('rh',norfolk,stvincent); % Get rhumb line dist (deg)
dgc = distance('gc',norfolk,stvincent); % Get gt. circle dist (deg)
% Compute headings and distances for the waypoint legs
[course distnm] = legs(latpts,lonpts,'rh');
```

Finally, compare the distances:

```
distrhnm = deg2nm(drh)             % Nautical mi along rhumb line
distgcnm = deg2nm(dgc)             % Nautical mi along great circle
distlegsnm = sum(distnm)           % Total dist along the 3 legs
rhgcdiff = distrhnm - distgcnm     % Excess rhumb line distance
trgcdiff = distlegsnm - distgcnm   % Excess distance along legs

distrhnm =
  3.2127e+003

distgcnm =
  3.1407e+003

distlegsnm =
  3.1490e+003

rhgcdiff =
   71.9980

trgcdiff =
    8.3446
```

Following just three rhumb line legs reduces the distance travelled from 72 nm to 8.3 nm compared to a great circle course.

Track Laydown – Displaying Navigational Tracks

Navigational tracks are most useful when graphically displayed. Traditionally, the navigator identifies and plots waypoints on a Mercator projection and then connects them with a straightedge, which on this projection results in rhumb line tracks. In the previous example, waypoints were chosen to approximate a great circle route, but they can be selected for a variety of other reasons.

Let's say that after arriving at Cape St. Vincent, your tanker must traverse the Straits of Gibraltar and then travel on to Port Said, the northern terminus of the Suez Canal. On the scale of the Mediterranean Sea, following great circle paths is of little concern compared to ensuring that the many straits and passages are safely transited. The navigator selects appropriate waypoints and plots them.

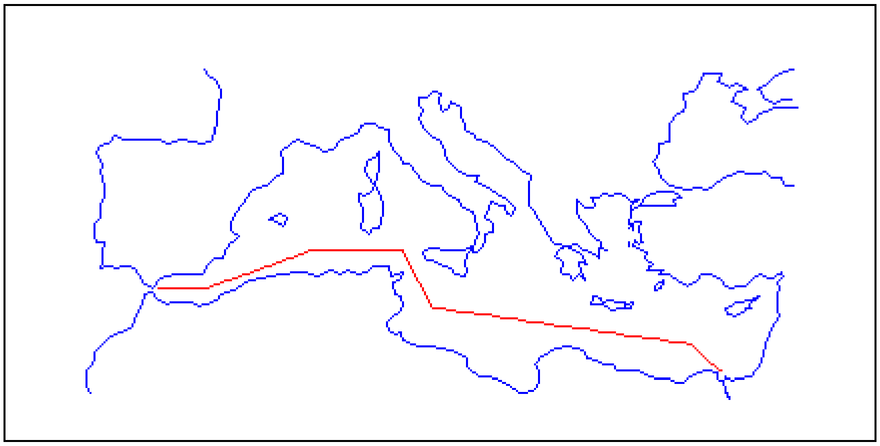
To accomplish this with Mapping Toolbox functions, you can display a map axes with a Mercator projection, select appropriate map latitude and longitude limits to isolate the area of interest, plot coastline data, and interactively mouse-select the waypoints with the `inputm` function. The `track` function will generate points to connect these waypoints, which can then be displayed with `plotm`.

For illustration, assume that the waypoints are known (or were gathered using `inputm`). To learn about using `inputm`, see "Interacting with Displayed Maps" on page 4-76, or `inputm` in the Mapping Toolbox reference pages.

```
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
load coast
axesm('MapProjection','mercator',...
'MapLatLimit',[30 47],'MapLonLimit',[-10 37])
framem
plotm(lat,long)
```

```
[lttrk,lntrk] = track(waypoints);
plotm(lttrk,lntrk,'r')
```

Although these track segments are straight lines on the Mercator projection, they are curves on others:



The segments of a track like this are called *legs*. Each of these legs can be described in terms of course and distance. The function `legs` will take the waypoints in navigational track format and return the course and distance required for each leg. Remember, the order of the points in this format determines the direction of travel. Courses are therefore calculated from each waypoint to its successor, not the reverse.

```
[courses,distances] = legs(waypoints)
courses =
 90.0000
 70.3132
 90.0000
 151.8186
 98.0776
 131.5684
distances =
 145.6231
```

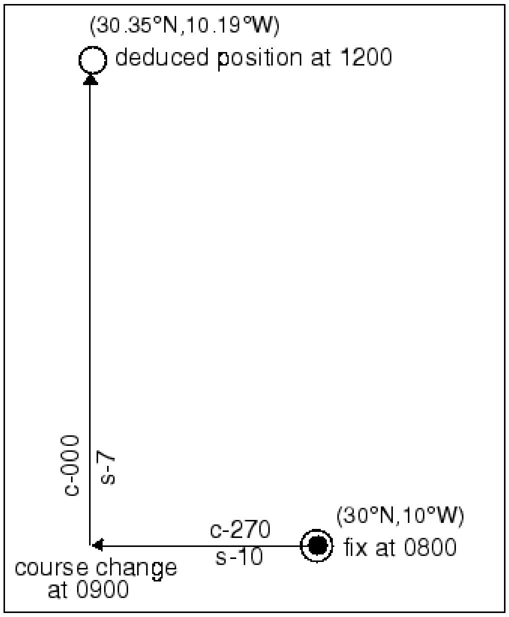
```
   356.2117
   283.6839
   204.2073
   854.0092
   135.6415
```

Since this is a navigation function, the courses are all in degrees and the distances are in nautical miles. From these distances, speeds required to arrive at Port Said at a given time can be calculated. Southbound traffic is allowed to enter the canal only once per day, so this information might be economically significant, since unnecessarily high speeds can lead to high fuel costs.

Dead Reckoning

When sailors first ventured out of sight of land, they faced a daunting dilemma. How could they find their way home if they didn't know where they were? The practice of *dead reckoning* is an attempt to deal with this problem. The term is derived from *deduced reckoning*.

Briefly, dead reckoning is vector addition plotted on a chart. For example, if you have a fix at (30ºN,10ºW) at 0800, and you proceed due west for 1 hour at 10 knots, and then you turn north and sail for 3 hours at 7 knots, you should be at (30.35ºN,10.19ºW) at 1200.



However, a sailor *shoots the sun* at local apparent noon and discovers that the ship's latitude is actually 30.29ºN. What's worse, he lives before the invention of a reliable chronometer, and so he cannot calculate his longitude at all from this sighting. What happened?

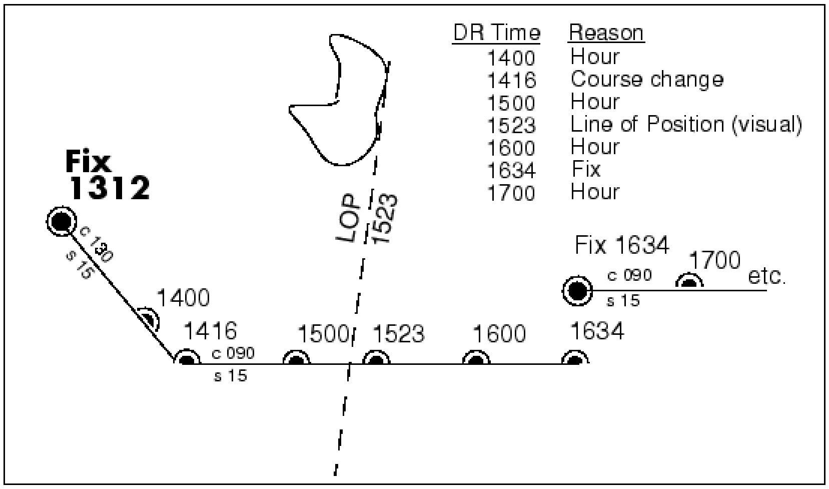
Leaving aside the difficulties in speed determination and the need to tack off course, even modern craft have to contend with winds and currents. However, despite these limitations, dead reckoning is still used for determining position between fixes and for forecasting future positions. This is because dead reckoning provides a certainty of assumptions that estimations of wind and current drift cannot.

When navigators establish a fix from some source, be it from piloting, celestial, or satellite observations, they plot a dead reckoning (DR) track, which is a plot of the intended positions of the ship forward in time. In practice, dead reckoning is usually plotted for 3 hours in advance, or for the time period covered by the next three expected fixes. In open ocean conditions, hourly fixes are sufficient; in coastal pilotage, three-minute fixes are common.

Specific DR positions, which are sometimes called *DR*s, are plotted according to the *Rules of DR*:

- DR at every course change
- DR at every speed change
- DR every hour on the hour
- DR every time a fix or running fix is obtained
- DR 3 hours ahead or for the next three expected fixes
- DR for every line of position (LOP), either visual or celestial

For example, the navigator plots these DRs:



DR Time	Reason
1400	Hour
1416	Course change
1500	Hour
1523	Line of Position (visual)
1600	Hour
1634	Fix
1700	Hour

Notice that the 1523 DR does not coincide with the LOP at 1523. Although note is taken of this variance, one line is insufficient to calculate a new fix.

Mapping Toolbox function `dreckon` calculates the DR positions for a given set of courses and speeds. The function provides DR positions for the first three rules of dead reckoning. The approach is to provide a set of waypoints in navigational track format corresponding to the plan of intended movement.

The time of the initial waypoint, or fix, is also needed, as well as the speeds to be employed along each leg. Alternatively, a set of speeds and the times for which each speed will apply can be provided. `dreckon` returns the positions and times required of these DRs:

- `dreckon` calculates the times for position of each course change, which will occur at the waypoints
- `dreckon` calculates the positions for each whole hour
- If times are provided for speed changes, `dreckon` calculates positions for these times if they do not occur at course changes

Imagine you have a fix at midnight at the point (10ºN,0º):

```
waypoints(1,:) = [10 0]; fixtime = 0;
```

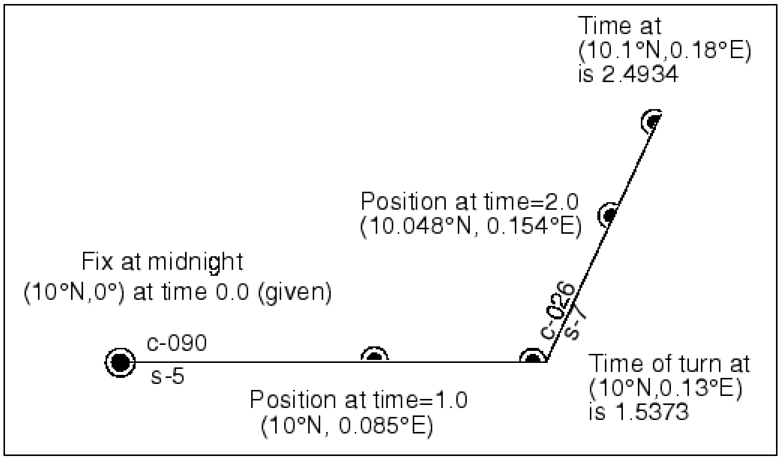
You intend to travel east and alter course at the point (10ºN,0.13ºE) and head for the point (10.1ºN,0.18ºE). On the first leg, you will travel at 5 knots, and on the second leg you will speed up to 7 knots.

```
waypoints(2,:) = [10 .13];
waypoints(3,:) = [10.1 .18];
speeds = [5;7];
```

To determine the DR points and times for this plan, use `dreckon`:

```
[drlat,drlon,drtime] = dreckon(waypoints,fixtime,speeds);
[drlat drlon drtime]
ans =
 10.0000    0.0846    1.0000     % Position at 1 am
 10.0000    0.1301    1.5373     % Time of course change
 10.0484    0.1543    2.0000     % Position at 2 am
 10.1001    0.1801    2.4934     % Time at final waypoint
```

Here is an illustration of this track and its DR points:

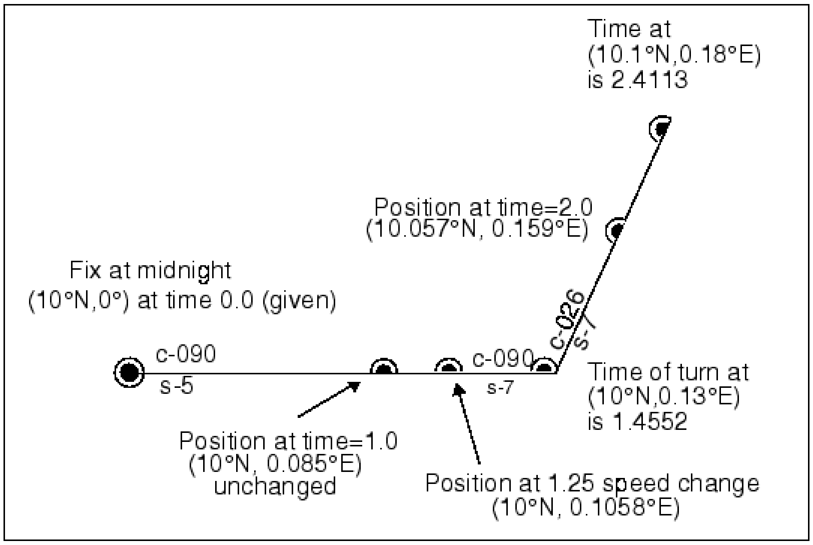


However, you would like to get to the final point a little earlier to make a rendezvous. You decide to recalculate your DRs based on speeding up to 7 knots a little earlier than planned. The first calculation tells you that you were going to increase speed at the turn, which would occur at a time 1.5373 hours after midnight, or 1:32 a.m. (at time 0132 in navigational time format). What time would you reach the rendezvous if you increased your speed to 7 knots at 1:15 a.m. (0115, or 1.25 hours after midnight)?

To indicate times for speed changes, another input is required, providing a time interval after the fix time at which each ordered speed is to end. The first speed, 5 knots, is to end 1.25 hours after midnight. Since you don't know when the rendezvous will be made under these circumstances, set the time for the second speed, 7 knots, to end at infinity. No DRs will be returned past the last waypoint.

```
spdtimes = [1.25; inf];
[drlat,drlon,drtime] = dreckon(waypoints,fixtime,...
                               speeds,spdtimes);
[drlat,drlon,drtime]
ans =
 10.0000    0.0846    1.0000   % Position at 1 am
 10.0000    0.1058    1.2500   % Position at speed change
 10.0000    0.1301    1.4552   % Time of course change
 10.0570    0.1586    2.0000   % Position at 2 am
 10.1001    0.1801    2.4113   % Time at final waypoint
```

This following illustration shows the difference:

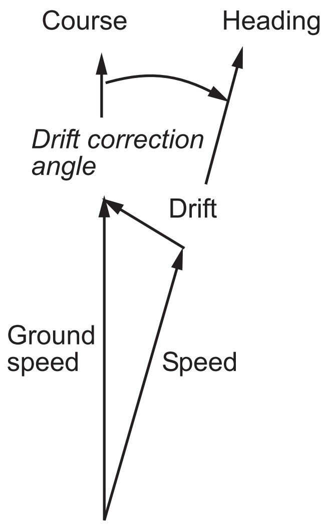


The times at planned positions after the speed change are a little earlier; the position at the known time (2 a.m.) is a little farther along. With this plan, you will arrive at the rendezvous about 4 1/2 minutes earlier, so you may want to consider a greater speed change.

Drift Correction

Dead reckoning is a reasonably accurate method for predicting position if the vehicle is able to maintain the planned course. Aircraft and ships can be pushed off the planned course by winds and current. An important step in navigational planning is to calculate the required drift correction.

In the standard drift correction problem, the desired course and wind are known, but the heading needed to stay on course is unknown. This problem is well suited to vector analysis. The wind velocity is a vector of known magnitude and direction. The vehicle's speed relative to the moving air mass is a vector of known magnitude, but unknown direction. This heading must be chosen so that the sum of the vehicle and wind velocities gives a resultant in the specified course direction. The ground speed can be larger or smaller than the air speed because of headwind or tailwind components. A navigator would like to know the required heading, the associated wind correction angle, and the resulting ground speed.



What heading puts an aircraft on a course of 250° when the wind is 38 knots from 285°? The aircraft flies at an airspeed of 145 knots.

```
course = 250; airspeed = 145; windfrom = 285; windspeed = 38;
[heading,groundspeed,windcorrangle] = ...
driftcorr(course,airspeed,windfrom,windspeed)

heading =
        258.65

groundspeed =
        112.22

windcorrangle =
          8.65
```

The required heading is about 9° to the right of the course. There is a 33-knot headwind component.

A related problem is the calculation of the wind speed and direction from observed heading and course. The wind velocity is just the vector difference of the ground speed and the velocity relative to the air mass.

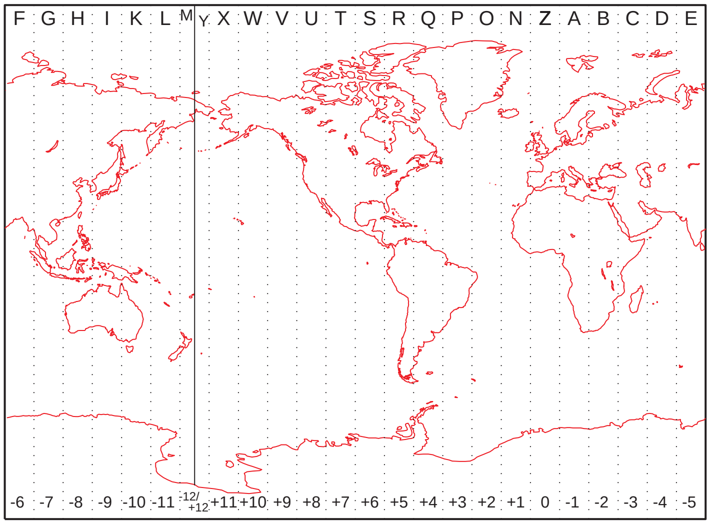
```
[windfrom,windspeed] = ...
driftvel(course,groundspeed,heading,airspeed)

windfrom =
        285.00

windspeed =
         38.00
```

Time Zones

Time zones used for navigation are uniform 15º extents of longitude. The `timezone` function returns a navigational time zone, that is, one based solely on longitude with no regard for statutory divisions. So, for example, Chicago, Illinois, lies in the statutory U.S. Central time zone, which has irregular boundaries devised for political or convenience reasons. However, from a navigational standpoint, Chicago's longitude places it in the *S* (Sierra) time zone. The zone's *description* is +6, which indicates that 6 hours must be added to local time to get Greenwich, or *Z* (Zulu) time. So, if it is noon, standard time in Chicago, it is 12+6, or 6 p.m., at Greenwich.

Each 15º navigational time zone has a distinct description and designating letter. The exceptions to this are the two zones on either side of the date line, *M* and *Y* (Mike and Yankee). These zones are only 7-1/2º wide, since on one side of the date line, the description is +12, and on the other, it is -12.

Navigational time zones are very important for celestial navigation calculations. Although there are no Mapping Toolbox functions designed specifically for celestial navigation, a simple example can be devised.



It is possible with a sextant to determine *local apparent noon*. This is the moment when the Sun is at its zenith from your point of view. At the exact center longitude of a time zone, the phenomenon occurs exactly at noon, local time. Since the Sun traverses a 15° time zone in 1 hour, it crosses one degree every 4 minutes. So if you observe local apparent noon at 11:54, you must be 1.5° east of your center longitude.

You must know what time zone you are in before you can even attempt a fix. This concept has been understood since the spherical nature of the Earth was first accepted, but early sailors had no ability to keep accurate time on ship, and so were unable to determine their longitude. The invention of accurate chronometers in the 18th century solved this problem.

The `timezone` function is quite simple. It returns the description, `zd`, an integer for use in calculations, a string, `zltr`, of the zone designator, and a string fully naming the `zone`. For example, the information for a longitude 123°E is the following:

```
[zd,zltr,zone] = timezone(123)
zd =
  -8
zltr =
H
zone =
  -8 H
```

Returning to the simple celestial navigation example, the center longitude of this zone is:

```
-(zd*15)
ans =
  120
```

This means that at our longitude, 123°E, we should experience local apparent noon at 11:48 a.m., 12 minutes early.

Function Reference

Geospatial Data Import and Access

Standard File Formats (p. 11-2)	Reading and writing vector and raster geodata in widely used exchange formats
Gridded Terrain and Bathymetry Products (p. 11-3)	For reading raster data products distributed in special file formats
Vector Map Products (p. 11-4)	For reading vector data products distributed in special file formats
Miscellaneous Data Sets (p. 11-5)	For reading other data products distributed in special file formats
GUIs for Data Import (p. 11-5)	GUIs for browsing data products and selecting areas and objects of interest
File Reading Utilities (p. 11-5)	Low-level access functions for text and other data files
Ellipsoids, Radii, Areas, and Volumes (p. 11-5)	Geometric parameters of Earth, planets, Sun, and Moon

Standard File Formats

arcgridread	Read gridded data set in Arc ASCII Grid Format
geotiffinfo	Information about GeoTIFF file
geotiffread	Read georeferenced image from GeoTIFF file
getworldfilename	Derive worldfile name from image filename
kmlwrite	Write geographic data to KML file
makeattribspec	Attribute specification from geographic data structure
sdtsdemread	Read data from SDTS raster/DEM data set

sdtsinfo	Information about SDTS data set
shapeinfo	Information about shapefile
shaperead	Read vector features and attributes from shapefile
shapewrite	Write geographic data structure to shapefile
worldfileread	Read worldfile and return referencing matrix
worldfilewrite	Construct worldfile from referencing matrix

Gridded Terrain and Bathymetry Products

dted	Read U.S. Department of Defense Digital Terrain Elevation Data (DTED)
dteds	DTED filenames for latitude-longitude quadrangle
etopo	Read global 5-min or 2-min digital terrain data
globedem	Read Global Land One-km Base Elevation (GLOBE) data
globedems	GLOBE data filenames for latitude-longitude quadrangle
gtopo30	Read 30-arc-second global digital elevation data (GTOPO30)
gtopo30s	GTOPO30 data filenames for latitude-longitude quadrangle
satbath	Read 2-minute terrain/bathymetry from Smith and Sandwell
tbase	Read 5-minute global terrain elevations from TerrainBase

usgs24kdem	Read USGS 7.5-minute (30-m or 10-m) Digital Elevation Models
usgsdem	Read USGS 1-degree (3-arc-second) Digital Elevation Model
usgsdems	USGS 1-degree (3-arc-sec) DEM filenames for latitude-longitude quadrangle

Vector Map Products

dcwdata	Read selected DCW worldwide basemap data
dcwgaz	Search DCW worldwide basemap gazette file
dcwread	Read DCW worldwide basemap file
dcwrhead	Read DCW worldwide basemap file headers
fipsname	Read Federal Information Processing Standard (FIPS) name file used with TIGER thinned boundary files
gshhs	Read Global Self-Consistent Hierarchical High-Resolution Shoreline
tgrline	Read TIGER/Line data
vmap0data	Read selected data from Vector Map Level 0
vmap0read	Read Vector Map Level 0 file
vmap0rhead	Read Vector Map Level 0 file headers

Miscellaneous Data Sets

avhrrgoode	Read AVHRR data product stored in Goode Projection
avhrrlambert	Read AVHRR data product stored in eqaazim projection
egm96geoid	Read 15-minute gridded geoid heights from EGM96
readfk5	Read Fifth Fundamental Catalog of Stars

GUIs for Data Import

demdataui	UI for selecting digital elevation data
vmap0ui	UI for selecting data from Vector Map Level 0

File Reading Utilities

grepfields	Identify matching fields in fixed record length files
readfields	Read fields or records from fixed-format files
readmtx	Read matrix stored in file
spcread	Read columns of data from ASCII text file

Ellipsoids, Radii, Areas, and Volumes

almanac	Parameters for Earth, planets, Sun, and Moon

Web Map Service

WMS Server and Layer Information (p. 11-6)	For searching local database for relevant layers and servers
WMS Capabilities Information (p. 11-6)	For retrieving capabilities information from WMS server
WMS Map Rendering (p. 11-7)	For rendering WMS map

WMS Server and Layer Information

disp (WMSLayer)	Display properties
refine (WMSLayer)	Refine search
refineLimits (WMSLayer)	Refine search based on geographic limits
servers (WMSLayer)	Return URLs of unique servers
serverTitles (WMSLayer)	Return titles of unique servers
updateLayers (WebMapServer)	Update layer properties
WebMapServer	Web map server object
WMSCapabilities	Web Map Service capabilities object
wmsfind	Search local database for Web map servers and layers
WMSLayer	Web Map Service layer object
wmsupdate	Synchronize WMSLayer object with server

WMS Capabilities Information

disp (WMSCapabilties)	Display properties
getCapabilities (WebMapServer)	Get capabilities document from server
WebMapServer	Web map server object

WMSCapabilities	Web Map Service capabilities object
wmsinfo	Information about WMS server from capabilities document

WMS Map Rendering

boundImageSize (WMSMapRequest)	Bound size of raster map
getMap (WebMapServer)	Get raster map from server
WebMapServer	Web map server object
WMSMapRequest	Web Map Service map request object
wmsread	Retrieve WMS map from server

Vector Map Data and Geographic Data Structures

Geographic Data Structures (p. 11-7)	For updating and obtaining fields from data structures
Data Manipulation (p. 11-8)	For altering, combining, and analyzing polygon and line data
Utilities for NaN-Separated Polygons and Lines (p. 11-8)	For structuring vectors defining multiple line or polygon objects

Geographic Data Structures

extractfield	Field values from structure array
extractm	Coordinate data from line or patch display structure
updategeostruct	Convert line or patch display structure to geostruct

Data Manipulation

bufferm	Buffer zones for latitude-longitude polygons
flatearthpoly	Insert points along date line to pole
interpm	Densify latitude-longitude sampling in lines or polygons
intrplat	Interpolate latitude at given longitude
intrplon	Interpolate longitude at given latitude
ispolycw	True if polygon vertices are in clockwise order
nanclip	Clip vector data with NaNs at specified pen-down locations
poly2ccw	Convert polygon contour to counterclockwise vertex ordering
poly2cw	Convert polygon contour to clockwise vertex ordering
poly2fv	Convert polygonal region to patch faces and vertices
polycut	Polygon branch cuts for holes
polymerge	Merge line segments with matching endpoints
reducem	Reduce density of points in vector data

Utilities for NaN-Separated Polygons and Lines

closePolygonParts	Close all rings in multipart polygon
isShapeMultipart	True, if polygon or line has multiple parts

polyjoin	Convert line or polygon parts from cell arrays to vector form
polysplit	Convert line or polygon parts from vector form to cell arrays
removeExtraNanSeparators	Clean up `NaN` separators in polygons and lines

Georeferenced Images and Data Grids

Spatial Referencing (p. 11-9)	Computing bounds and converting between geographic and raster coordinates for spatially referenced images and grids
Terrain Analysis (p. 11-10)	Computing slope, aspect, lines of sight, and terrain visibility
Other Analysis/Access (p. 11-11)	Computing areas and profiles, and selecting subsets of values from data grids
Construction and Modification (p. 11-11)	Constructing, encoding, seeding, reorienting, and converting data grids
Initialization (p. 11-12)	Generating data grids containing uniform values

Spatial Referencing

latlon2pix	Convert latitude-longitude coordinates to pixel coordinates
limitm	Determine latitude and longitude limits of regular data grid
makerefmat	Construct affine spatial-referencing matrix

map2pix	Convert map coordinates to pixel coordinates
mapbbox	Compute bounding box of georeferenced image or data grid
mapoutline	Compute outline of georeferenced image or data grid
meshgrat	Construct map graticule for surface object display
pix2map	Convert pixel coordinates to map coordinates
pixcenters	Compute pixel centers for georeferenced image or data grid
refmat2vec	Convert referencing matrix to referencing vector
refvec2mat	Convert referencing vector to referencing matrix
setltln	Convert data grid rows and columns to latitude-longitude
setpostn	Convert latitude-longitude to data grid rows and columns

Terrain Analysis

gradientm	Calculate gradient, slope and aspect of data grid
los2	Line-of-sight visibility between two points in terrain
viewshed	Areas visible from point on terrain elevation grid

Other Analysis/Access

areamat	Surface area covered by nonzero values in binary data grid
filterm	Filter latitudes and longitudes based on underlying data grid
findm	Latitudes and longitudes of nonzero data grid elements
ltln2val	Extract data grid values for specified locations
mapprofile	Interpolate heights between waypoints on regular data grid

Construction and Modification

changem	Substitute values in data array
encodem	Fill in regular data grid from seed values and locations
geoloc2grid	Convert geolocated data array to regular data grid
imbedm	Encode data points into regular data grid
neworig	Orient regular data grid to oblique aspect
resizem	Resize regular data grid
sizem	Row and column dimensions needed for regular data grid
vec2mtx	Convert latitude-longitude vectors to regular data grid

Initialization

nanm	Construct regular data grid of NaNs
onem	Construct regular data grid of 1s
spzerom	Construct sparse regular data grid of 0s
zerom	Construct regular data grid of 0s

Map Projections and Coordinates

Available Map Projections (p. 11-13)	Lists of map projections and characteristics
Map Projection Transformations (p. 11-13)	Forward and inverse map projection functions
Map Trimming (p. 11-13)	For trimming lines, polygons, and data grids to latitude-longitude quadrangles
Angles, Scales, and Distortions (p. 11-14)	Computing directions, angles, and distortions on projected maps
Visualizing Map Distortions (p. 11-14)	Generating displays of distortion statistics and Tissot ellipses
UTM System (p. 11-14)	Selecting zones and ellipsoids for the Universal Transverse Mercator system
Coordinate Rotation on the Sphere (p. 11-14)	Reorienting map data by solid-body rotations on the sphere
Trimming and Clipping (p. 11-15)	Removing and replacing data that extends outside a map frame

For specific map projections, see Chapter 14, “Map Projections Reference”.

Available Map Projections

maplist	Available Mapping Toolbox map projections
maps	List available map projections and verify names
projlist	Map projections supported by `projfwd` and `projinv`

Map Projection Transformations

mfwdtran	Project geographic features to map coordinates
minvtran	Unproject features from map to geographic coordinates
projfwd	Forward map projection using `PROJ.4` map projection library
projinv	Inverse map projection using `PROJ.4` map projection library

Map Trimming

maptriml	Trim lines to latitude-longitude quadrangle
maptrimp	Trim polygons to latitude-longitude quadrangle
maptrims	Trim regular data grid to latitude-longitude quadrangle

Angles, Scales, and Distortions

distortcalc	Distortion parameters for map projections
vfwdtran	Direction angle in map plane from azimuth on ellipsoid
vinvtran	Azimuth on ellipsoid from direction angle in map plane

Visualizing Map Distortions

mdistort	Display contours of constant map distortion
tissot	Project Tissot indicatrices on map axes

UTM System

utmgeoid	Select ellipsoids for given UTM zone
utmzone	Select UTM zone given latitude and longitude

Coordinate Rotation on the Sphere

newpole	Origin vector to place specific point at pole
org2pol	Location of north pole in rotated map
putpole	Origin vector to place north pole at specified point

Trimming and Clipping

clipdata	Clip data at `+/-pi` in longitude, `+/-pi` in latitude
trimcart	Trim graphic objects to map frame
trimdata	Trim map data exceeding projection limits
undoclip	Remove object clips introduced by `clipdata`
undotrim	Remove object trims introduced by `trimdata`

Map Display and Interaction

Map Creation and High-Level Display (p. 11-16)	Top-level functions that create map axes, project map data onto them, and control symbolization
Vector Symbolization (p. 11-17)	Functions that draw symbols for points, lines, and polygons (coordinate lists and geostructs)
Lines and Contours (p. 11-17)	Lower level line plotting and higher level contour plotting functions
Patch Data (p. 11-17)	Lower-level functions for plotting polygons as patches on map axes
Data Grids (p. 11-18)	For mapping regular and geolocated data grids in 2-D and 3-D
Light Objects and Lighted Surfaces (p. 11-18)	For mapping regular and geolocated data grids using lighting and shading
Thematic Maps (p. 11-18)	For making scatter, quiver, comet, and stem maps

Map Creation and High-Level Display

axesm	Define map axes and set map properties
displaym	Display geographic data from display structure
geoshow	Display map latitude and longitude data
grid2image	Display regular data grid as image
mapview	Interactive map viewer

usamap	Construct map axes for United States of America
worldmap	Construct map axes for given region of world

Vector Symbolization

makesymbolspec	Construct vector layer symbolization specification

Lines and Contours

contour3m	Project 3-D contour plot of map data
contourfm	Project filled 2-D contour plot of map data
contourm	Project 2-D contour plot of map data
linem	Project line object on map axes
plot3m	Project 3-D lines and points on map axess
plotm	Project 2-D lines and points on map axes

Patch Data

fill3m	Project filled 3-D patch objects on map axes
fillm	Project filled 2-D patch objects on map axes
patchesm	Project patches on map axes as individual objects
patchm	Project patch objects on map axes

Data Grids

meshm	Project regular data grid on map axes
pcolorm	Project regular data grid on map axes in z = 0 plane
surfacem	Project and add geolocated data grid to current map axes
surfm	Project geolocated data grid on map axes

Light Objects and Lighted Surfaces

lightm	Project light objects on map axes
meshlsrm	3-D lighted shaded relief of regular data grid
shaderel	Construct `cdata` and colormap for shaded relief
surflm	3-D shaded surface with lighting on map axes
surflsrm	3-D lighted shaded relief of geolocated data grid

Thematic Maps

comet3m	Project 3-D comet plot on map axes
cometm	Project 2-D comet plot on map axes
quiverm	Project 2-D quiver plot on map axes
scatterm	Project point markers with variable color and area

stem3m	Project stem plot map on map axes
symbolm	Project point markers with variable size

Map Annotation

clabelm	Add contour labels to map contour display
framem	Toggle and control display of map frame
gridm	Toggle and control display of graticule lines
lcolorbar	Colorbar with text labels
mlabel	Toggle and control display of meridian labels
mlabelzero22pi	Convert meridian labels to 0-360 degree range
northarrow	Add graphic element pointing to geographic north pole
plabel	Toggle and control display of parallel labels
rotatetext	Rotate text to projected graticule
scaleruler	Add or modify graphic scale on map axes
textm	Project text annotation on map axes

Colormaps for Map Displays

contourcmap	Contour colormap and colorbar current axes
demcmap	Colormaps appropriate to terrain elevation data
polcmap	Colormaps appropriate to political regions

Interactive Map Positions

gcpmap	Current mouse point from map axes
gtextm	Place text on map using mouse
inputm	Latitudes and longitudes of mouse-click locations

Interactive Track and Circle Definition

scircleg	Small circle defined via mouse input
sectorg	Sector of small circle defined via mouse input
trackg	Great circle or rhumb line defined via mouse input

GUIs

clrmenu	Add colormap menu to figure window
colorm	Create index map colormaps
colorui	Interactively define RGB color
getseeds	Interactively assign seeds for data grid encoding

lightmui	Control position of lights on globe or 3-D map
maptool	Add menu activated tools to map figure
maptrim	Interactively trim and convert map data from vector to raster format
mlayers	GUI to control plotting of display structure elements
mobjects	Manipulate object sets displayed on map axes
originui	Interactively modify map origin
panzoom	Pan and zoom on map axes
parallelui	Interactively modify map parallels
qrydata	GUI to interactively perform data queries
rootlayr	Construct cell array of workspace variables for `mlayers` tool
seedm	GUI to fill data grids with seeded values
surfdist	Interactive distance, azimuth, and reckoning calculations
uimaptbx	Handle buttondown callbacks for mapped objects
utmzoneui	Choose or identify UTM zone by clicking map

Map Object and Projection Properties

cart2grn	Transform projected coordinates to Greenwich system
defaultm	Initialize or reset map projection structure

gcm	Current map projection structure
geotiff2mstruct	Convert GeoTIFF information to map projection structure
getm	Map object properties
handlem	Handles of displayed map objects
ismap	True for axes with map projection
ismapped	True, if object is projected on map axes
makemapped	Convert ordinary graphics object to mapped object
namem	Determine names of valid graphics objects
project	Project displayed map graphics object
restack	Restack objects within map axes
rotatem	Transform vector map data to new origin and orientation
setm	Set properties of map axes and graphics objects
tagm	Set property of map graphics object
zdatam	Adjust z-plane of displayed map objects

Map Appearance

axesscale	Resize axes for equivalent scale
camposm	Set camera position using geographic coordinates
camtargm	Set camera target using geographic coordinates

camupm	Set camera up vector using geographic coordinates
daspectm	Control vertical exaggeration in map display
paperscale	Set figure properties for printing at specified map scale
previewmap	View map at printed size
tightmap	Remove white space around map

Display Clearing

clma	Clear current map axes
clmo	Clear specified graphics objects from map axes
hidem	Hide specified graphic objects on map axes
showaxes	Toggle display of map coordinate axes
showm	Specify graphic objects to display on map axes

Geographic Calculations

Geometry of Sphere and Ellipsoid (p. 11-24)	Distances, deviations, areas, and curves on the sphere or ellipsoid
3-D Coordinates (p. 11-25)	For converting between different 3-D coordinate systems
Ellipsoids and Latitudes (p. 11-25)	For converting ellipsoid parameters and auxiliary latitudes

Geometric Object Overlay (p. 11-26)	For determining if, how, and where points, lines, circles, and areas intersect
Geographic Statistics (p. 11-27)	For computing geographic means, standard deviations, and histograms
Navigation (p. 11-27)	For determining positions, headings, drift, and navigational fixes and way points

Geometry of Sphere and Ellipsoid

antipode	Point on opposite side of globe
areaint	Surface area of polygon on sphere or ellipsoid
areaquad	Surface area of latitude-longitude quadrangle
azimuth	Azimuth between points on sphere or ellipsoid
departure	Departure of longitudes at specified latitudes
distance	Distance between points on sphere or ellipsoid
ellipse1	Geographic ellipse from center, semimajor axes, eccentricity, and azimuth
gc2sc	Center and radius of great circle
meridianarc	Ellipsoidal distance along meridian
meridianfwd	Reckon position along meridian
reckon	Point at specified azimuth, range on sphere or ellipsoid
scircle1	Small circles from center, range, and azimuth

scircle2	Small circles from center and perimeter
track1	Geographic tracks from starting point, azimuth, and range
track2	Geographic tracks from starting and ending points

3-D Coordinates

ecef2geodetic	Convert geocentric (ECEF) to geodetic coordinates
ecef2lv	Convert geocentric (ECEF) to local vertical coordinates
elevation	Local vertical elevation angle, range, and azimuth
geodetic2ecef	Convert geodetic to geocentric (ECEF) coordinates
lv2ecef	Convert local vertical to geocentric (ECEF) coordinates

Ellipsoids and Latitudes

axes2ecc	Eccentricity of ellipse with given axis lengths
convertlat	Convert between geodetic and auxiliary latitudes
ecc2flat	Flattening of ellipse with given eccentricity
ecc2n	n-value of ellipse with given eccentricity
flat2ecc	Eccentricity of ellipse with given flattening

geocentric2geodeticLat	Convert geocentric to geodetic latitude
geodetic2geocentricLat	Convert geodetic to geocentric latitude
majaxis	Semimajor axis of ellipse given semiminor axis and eccentricity
minaxis	Semiminor axis of ellipse given semimajor axis and eccentricity
n2ecc	Eccentricity of ellipse with given n-value
rcurve	Radii of curvature of ellipsoid
rsphere	Radii of auxiliary spheres

Geometric Object Overlay

circcirc	Intersections of circles in Cartesian plane
gcxgc	Intersection points for pairs of great circles
gcxsc	Intersection points for great and small circle pairs
ingeoquad	True for points inside or on lat-lon quadrangle
intersectgeoquad	Intersection of two latitude-longitude quadrangles
linecirc	Intersections of circles and lines in Cartesian plane
outlinegeoquad	Polygon outlining geographic quadrangle
polybool	Set operations on polygonal regions
polyxpoly	Intersection points for lines or polygon edges

rhxrh	Intersection points for pairs of rhumb lines
scxsc	Intersection points for pairs of small circles

Geographic Statistics

combntns	All possible combinations of set of values
eqa2grn	Convert from equal area to Greenwich coordinates
grn2eqa	Convert from Greenwich to equal area coordinates
hista	Histogram for geographic points with equal-area bins
meanm	Mean location of geographic coordinates
stdist	Standard distance for geographic points
stdm	Standard deviation for geographic points

Navigation

crossfix	Cross-fix positions from bearings and ranges
dreckon	Dead reckoning positions for track
driftcorr	Heading to correct for wind or current drift
driftvel	Wind or current from heading, course, and speeds

gcwaypts	Equally spaced waypoints along great circle
legs	Courses and distances between navigational waypoints
navfix	Mercator-based navigational fix
timezone	Time zone based on longitude
track	Track segments to connect navigational waypoints

Utilities

Angle Conversions (p. 11-29)	For converting angles between different units and encodings
Conversion Factors for Angles and Distances (p. 11-29)	Function to compute factor for converting between units of distance and angles
Data Precision (p. 11-29)	For managing data precision
Distance Conversions (p. 11-30)	For converting distances between different units and encodings
Image Conversion (p. 11-30)	Function for changing indexed images to uint8 true-color images
String Formatters (p. 11-30)	For formatting angles and distances as text suitable for annotations
Longitude or Azimuth Wrapping (p. 11-30)	For forcing angles to lie within specified intervals

Angle Conversions

degrees2dm	Convert degrees to degrees-minutes
degrees2dms	Convert degrees to degrees-minutes-seconds
degtorad	Convert angles from degrees to radians
dm2degrees	Convert degrees-minutes to degrees
dms2degrees	Convert degrees-minutes-seconds to degrees
fromDegrees	Convert angles from degrees
fromRadians	Convert angles from radians
radtodeg	Convert angles from radians to degrees
str2angle	Convert strings to angles in degrees
toDegrees	Convert angles to degrees
toRadians	Convert angles to radians

Conversion Factors for Angles and Distances

unitsratio	Unit conversion factors

Data Precision

epsm	Accuracy in angle units for certain map computations
roundn	Round to multiple of 10

Distance Conversions

deg2km, deg2nm, deg2sm	Convert distance from degrees to kilometers, nautical miles, or statute miles
km2deg, nm2deg, sm2deg	Convert from distance units to degrees
km2nm, km2sm, nm2km, nm2sm, sm2km, sm2nm	Convert distance between kilometers and miles
km2rad, nm2rad, sm2rad	Convert from distance units to radians
rad2km, rad2nm, rad2sm	Convert distance from radians to kilometers, nautical miles, or statute miles

Image Conversion

ind2rgb8	Convert indexed image to uint8 RGB image

String Formatters

angl2str	Format angle strings
dist2str	Format distance strings

Longitude or Azimuth Wrapping

unwrapMultipart	Unwrap vector of angles with NaN-delimited parts
wrapTo180	Wrap angle in degrees to [-180 180]
wrapTo2Pi	Wrap angle in radians to [0 2*pi]

wrapTo360	Wrap angle in degrees to [0 360]
wrapToPi	Wrap angle in radians to [-pi pi]

GUIs

Map Definition Tools (p. 11-31)	Selecting vector and raster data, defining map axes, and projection parameters
Mapping Tools (p. 11-32)	Displaying maps, manipulating layers, and querying map objects
Display Manipulation Tools (p. 11-32)	Controlling zoom levels, colormaps, and lighting
Object Property Tools (p. 11-33)	Showing, hiding, tagging, and clearing objects, and customizing colormaps
Track Tools (p. 11-33)	Plotting small and great circles, rhumb lines, and other navigational tracks
Map Data Construction Tools (p. 11-34)	Setting limits, trimming maps, and seeding grid values

Map Definition Tools

axesm, axesmui	Define map axes and modify map projection and display properties
demdataui	UI for selecting digital elevation data
originui	Interactively modify map origin
parallelui	Interactively modify map parallels

utmzoneui	Choose or identify UTM zone by clicking map
vmap0ui	UI for selecting data from Vector Map Level 0

Mapping Tools

maptool	Add menu activated tools to map figure
maptrim	Interactively trim and convert map data from vector to raster format
mapview	Interactive map viewer
mlayers	GUI to control plotting of display structure elements
mobjects	Manipulate object sets displayed on map axes
qrydata	GUI to interactively perform data queries

Display Manipulation Tools

clrmenu	Add colormap menu to figure window
hidem-ui	Hide specified mapped objects
lightmui	Control position of lights on globe or 3-D map
panzoom	Pan and zoom on map axes

Object Property Tools

clmo	Clear specified graphics objects from map axes
colorui	Interactively define RGB color
handlem	Handles of displayed map objects
handlem-ui	GUI for handles of specified mapped objects
hidem	Hide specified graphic objects on map axes
property editors	GUIs to edit properties of mapped objects
showm	Specify graphic objects to display on map axes
tagm	Set property of map graphics object
zdatam	Adjust *z*-plane of displayed map objects

Track Tools

scircleg	Small circle defined via mouse input
scirclui	GUI to display small circles on map axes
sectorg	Sector of small circle defined via mouse input
surfdist	Interactive distance, azimuth, and reckoning calculations
trackg	Great circle or rhumb line defined via mouse input
trackui	GUI to display great circles and rhumb lines on map axes

Map Data Construction Tools

colorm	Create index map colormaps
seedm	GUI to fill data grids with seeded values

Functions — Alphabetical List

almanac

Purpose 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` 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'`	1849 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.

Examples

The radius of the Earth (treated as a sphere) in kilometers is

```
almanac('earth','radius')

ans =
 6371
```

The default ellipsoid model for the Earth (`[semimajor axis eccentricity]`) is

```
almanac('earth','ellipsoid')

ans =
 1.0e+03 *
  6.3781    0.0001
```

Note that the radius returned for any ellipsoid model reference body is the semimajor axis:

```
almanac('earth','radius','kilometers','ellipsoid')

Warning: Semimajor axis returned for radius parameter
ans =
   6.3781e+03
```

Compare the tabulated values of the Earth's surface area with a spherical assumption and with the 1966 World Geodetic System ellipsoid model:

```
almanac('earth','surfarea','statutemiles','actual')

ans =
     1.969499232704451e+008

almanac('earth','surfarea','statutemiles','sphere')

ans =
     1.969362058529953e+008
```

```
almanac('earth','surfarea','statutemiles','wgs66')

ans =
     1.969371331484438e+008
```

Note that these values are so close that long notation is required to differentiate them.

Some lunar measurements are

```
almanac('moon','radius')

ans =
        1738

almanac('moon','surfarea')

ans =
   3.7959e+07

almanac('moon','volume')

ans =
   2.1991e+10
```

Remarks

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`

angl2str

Purpose

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
`'degrees2dm'`	degrees	degrees + decimal minutes

Units	Units of Angle	Output Format
`'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`.

Remarks

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`.

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

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

See Also `str2angle`, `dist2str`

Purpose

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 `degtorad`, `radtodeg`, 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

Example

Convert from degrees to radians:

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

ans =
    0.4093
```

See Also

`degrees2dms`, `degtorad`, `fromDegrees`, `fromRadians`, `toDegrees`, `toRadians`, `radtodeg`, `unitsratio`

Purpose

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

Example 1

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

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

or (43ºS, 165ºW).

Example 2

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.

Example 3

Find the antipode of the location of the Mathworks corporate headquarters in Natick, Massachusetts. Map the headquarters location and its antipode in an orthographic projection. Begin by specifying 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

[amwlat amwlon] = antipode(mwlat,mwlon)

amwlat =
  -42.3007
amwlon =
  108.6478
```

Prove that these points are antipodes:

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

dist =
  180.0000
```

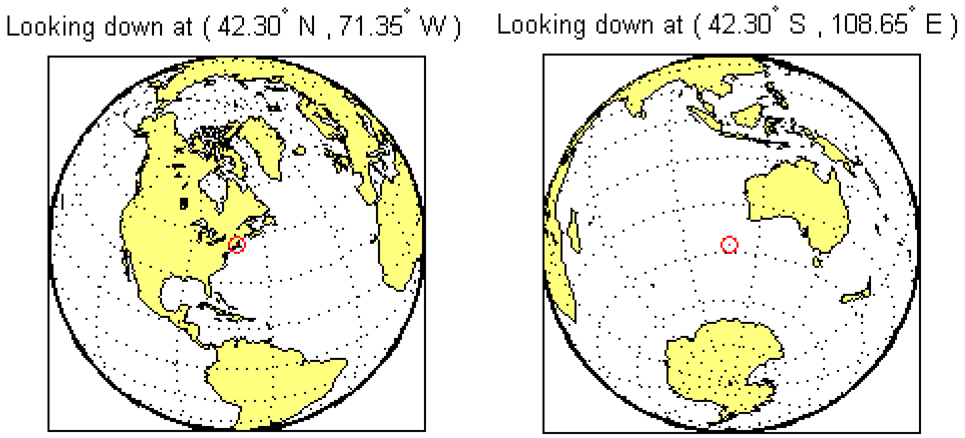
The `distance` function shows them to be 180 degrees apart.

Generate a map centered on the original point:

```
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')
s = ['Looking down at (' angl2str(mwlat,'ns') ...
    ',' angl2str(mwlon,'ew') ')'];
title(s)
```

Add a second map centered on the computed antipodal point:

```
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')
t = ['Looking down at (' angl2str(amwlat,'ns') ...
    ',' angl2str(amwlon,'ew') ')'];
title(t)
```


Looking down at (42.30° N , 71.35° W)
Looking down at (42.30° S , 108.65° E)

Purpose

Read gridded data set in Arc ASCII Grid Format

Syntax

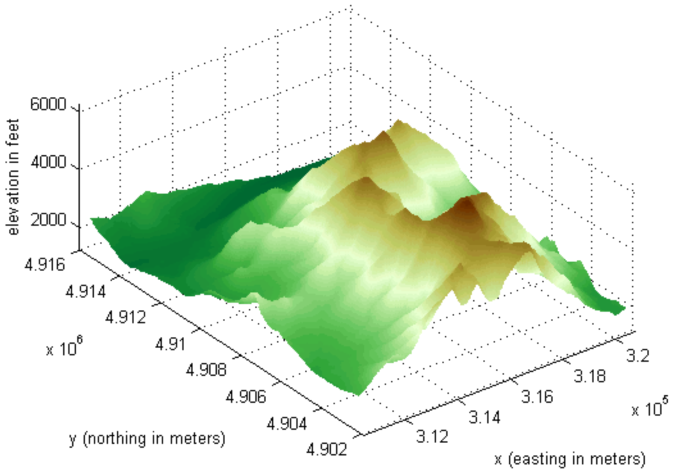
`[Z,R] = arcgridread(filename)`

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.

Example

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

Purpose

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)` uses the two-element ellipsoid vector `ellipsoid` to describe the sphere or ellipsoid. The output, `area`, is in square 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)]';
```

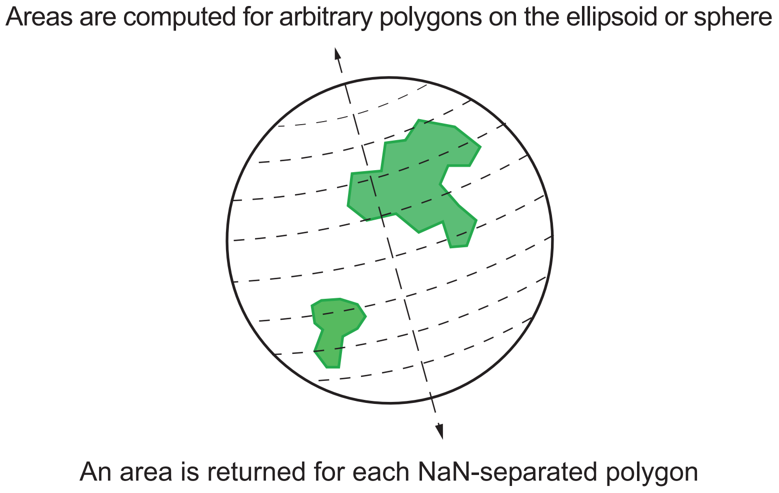
```
area = areaint(lats,lons)
area =
    0.0792
```

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

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

Algorithm

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`

Purpose

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` is 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
```

where `lat` and `lon` are in units of degrees. 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 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)` uses the input `ellipsoid` vector to describe the sphere or reference ellipsoid. `ellipsoid` has the form `[semi-major-axis-length, eccentricity]`. The units of the output, `A`, are the square of the length units in which the semi-major axis is provided. For example, if `ellipsoid` is replaced with `almanac('earth','wgs84','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`.

Remarks

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.

Examples

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

area =
    0.2411
```

Approximately 24% of the Earth has an altitude greater than 127 meters. What is the surface area of this portion of the Earth in square kilometers if a spherical ellipsoid is assumed? (Use the `almanac` function with the sphere as its reference body.)

```
earthgeoid = almanac('earth','ellipsoid','km','sphere');
area = areamat((topo>127),topolegend,earthgeoid)

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.

See Also

`almanac`, `areaint`, `areaquad`

Purpose

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

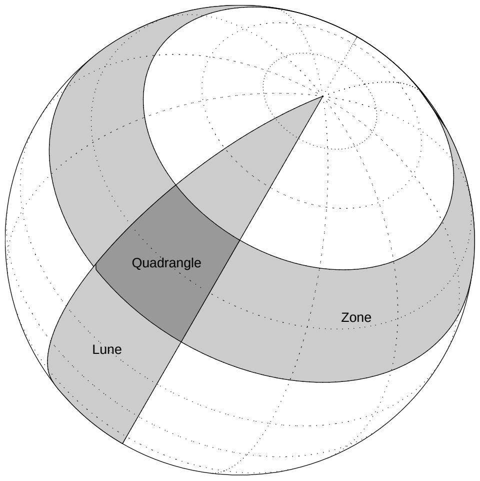
`area = areaquad(lat1,lon1,lat2,lon2)` returns the surface area bounded by the parallels `lat1` and `lat2` and the meridians `lon1` and `lon2`. The output `area` is a fraction of the unit sphere's area of 4π, so the result ranges from 0 to 1.

`area = areaquad(lat1,lon1,lat2,lon2,ellipsoid)` allows the specification of the ellipsoid model with the two-element ellipsoid vector `ellipsoid`. When `ellipsoid` is input, the resulting `area` is given in terms of the (squared) units of the ellipsoid. For example, if the ellipsoid `almanac('earth','ellipsoid','kilometers')` is used, the resulting area is in km^2. The default ellipsoid is the unit sphere.

`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 30°N and 45°N, and also between 25°W and 60°E:

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

area =
    0.0245
```

Assuming a spherical ellipsoid, find the surface area of the Earth in square kilometers. (Use the `almanac` function with the sphere as its reference body.)

```
earthellipsoid = almanac('earth','ellipsoid','km','sphere');
area = areaquad(-90,-180,90,180,earthellipsoid)

area =
   5.1006e+08
```

For comparison,

```
   almanac('earth','surfarea','km')

   ans =
      5.1006e+08
```

Algorithm

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

Purpose 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 100th 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 10th 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 `precision` specified. If empty, `'uint8'` is assumed. `'uint16'` is appropriate for some files. Check the metadata (`.txt` or `README`) file in the GLCC `ftp` directory 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.

Remarks

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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

Limitations

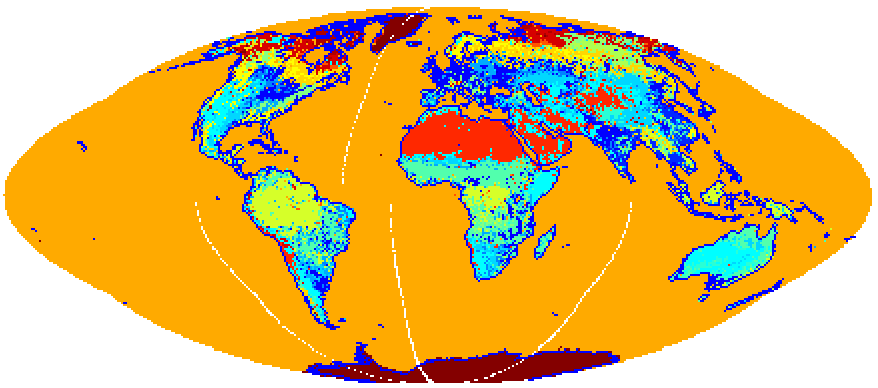
Most files store the data in scaled integers. Though this function returns the data as double, the scaling from integer to float is not performed. Check the data's README file for the appropriate scaling parameters.

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_0g.img`. (To run the example, you must first download the file.)

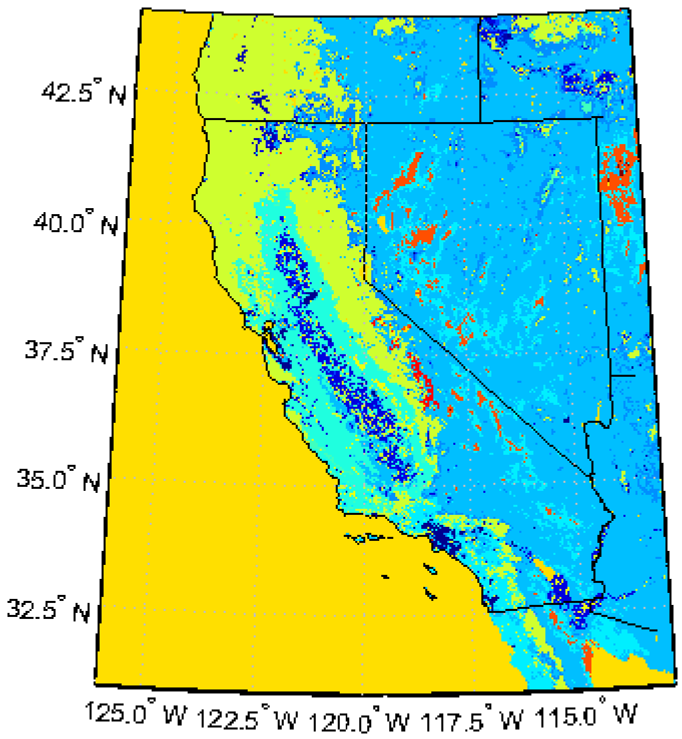
```
[latgrat, longrat, Z] = avhrrgoode('global','gusgs2_0g.img',50);

% 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('Renderer','zbuffer')
axesm('MapProjection', 'goode', 'Origin', origin, 'Geoid', ellipsoi
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');
```



See Also

`avhrrlambert`

Purpose

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 directory 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.

Remarks

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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

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 0];
ellipsoid = [6370997 0];
figure
axesm('MapProjection','eqaazim', 'Origin',origin, 'Geoid',
 ellipsoid)
geoshow(latgrat, longrat, Z, 'DisplayType', 'texturemap');
```

Example 2

Read and display every 50th point from the Global Land Cover Characteristics (GLCC) file covering Europe with the USGS classification scheme, named `eausgs1_2le.img`.

```
figure
worldmap europe
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
```

```
scalefactor = 50;
[latgrat, longrat, Z] = ... avhrrlambert('e', ...
   'eausgs1_2le.img', scalefactor, latlim, lonlim);
geoshow(latgrat, longrat, Z, 'DisplayType', 'texturemap');
```

Overlay the coastline.

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

See Also

`avhrrgoode`

Purpose

Eccentricity of ellipse with given axis lengths

Syntax

```
eccentricity = axes2ecc(semimajor,semiminor)
eccentricity = axes2ecc(axes)
```

Description

Eccentricity, the second element of the standard Mapping Toolbox ellipsoid vector, can be determined given both the semimajor and semiminor axes.

`eccentricity = axes2ecc(semimajor,semiminor)` returns the eccentricity associated with the input axes.

`eccentricity = axes2ecc(axes)` allows the axes inputs to be packed into a single two-column input of the form `[semimajor, semiminor]`.

Examples

Using the axes for the default GRS 80 Earth model,

```
ecc = axes2ecc(6378.1370,6356.7523)
ecc =
   0.08181921804834
```

This is the eccentricity returned by `almanac('earth','ellipsoid')`.

See Also

`almanac`, `ecc2n`, `majaxis`, `minaxis`

Purpose

Define map axes and set map properties

Syntax

```
axesm
axesm(PropertyName,PropertyValue,...)
axesm(ProjectionFcn,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(`*`ProjectionFcn`*`,`*`PropertyName`*`,PropertyValue,...)` allows omission of the `MapProjection` property name. The first input must be the identifying string of an available projection. For a list of these strings, see the table of projections in "Summary and Guide to Projections" on page 8-63.

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 data 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 properties `XColor`, `YColor`, and `ZColor` are set to the background color.
- The `hold` mode is on.

The map data 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 12-36, below, for descriptions of the map axes properties.

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 M-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 12-37
- "Properties That Control the Frame" on page 12-42
- "Properties That Control the Grid" on page 12-45
- "Properties That Control Grid Labeling" on page 12-48

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" on page 3-18 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 {0}`

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]`

Planet ellipsoid definition — Sets the ellipsoid for calculating the projections of any displayed map objects. In the toolbox, the ellipsoid is approximated by a spheroid. The default ellipsoid is a sphere with a radius of 1. This is represented as `[1 0]`. Any semimajor axis, in any distance units, can be entered; eccentricity lies between 0 and 1.

`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" on page 4-14 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" on page 4-14 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. The projection name is a string corresponding to an M-file appropriate to the projection. 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° 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°, 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 | {[0 0 0]}`

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 (`[0 0 0]`).

`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'` (`[0 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 | {[0 0 0]}`

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 (`[0 0 0]`).

`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 {0}`

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` (MATLAB function), `gcm`, `getm`, `setm`

Purpose

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.

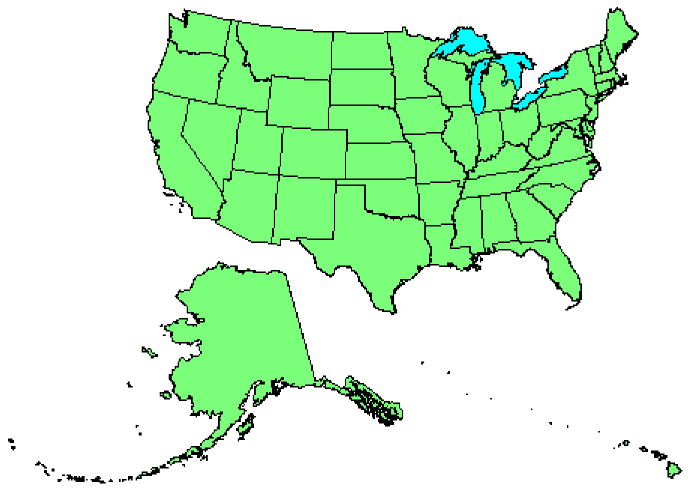
`axesscale(hbase,hother)` uses the axes `hbase` as the base axes, and rescales only the axes in `hother`.

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(strmatch('Alaska', statenames));
hawaii = states(strmatch('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.

Remarks

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

See Also

`paperscale`

azimuth

Purpose

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`. The `ellipsoid` vector is of the form `[semimajor_axis_length, eccentricity]`. The default ellipsoid is a unit sphere ([1 0]).

`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.

Definitions

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.

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 on the distinction between great circles and rhumb lines, see "Great Circles, Rhumb Lines, and Small Circles" on page 3-32 in the *Mapping Toolbox* documentation.

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.

Algorithm

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).

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.

See Also

`distance` | `elevation` | `reckon` | `track` | `track1` | `track2`

How To

- "Great Circles, Rhumb Lines, and Small Circles" on page 3-32

Purpose

Buffer zones for latitude-longitude polygons

Syntax

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

Description

`[latb,lonb] = bufferm(lat,lon,dist,*direction*)` computes the buffer zone around a polygon. A buffer zone for a closed polygon is defined as the locus of points that are a certain distance in or out of the polygon. A buffer zone for an open polygon is the locus of points a certain distance out from the polygon. The polygon is specified as vectors of latitude and longitude in units of degrees. The distance is a scalar specified in degrees of arc along the surface. Valid direction strings are `'in'` and `'out'`. The result is returned as `NaN`-clipped vectors in units of degrees.

`[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 omitted, 13 points per circle are used.

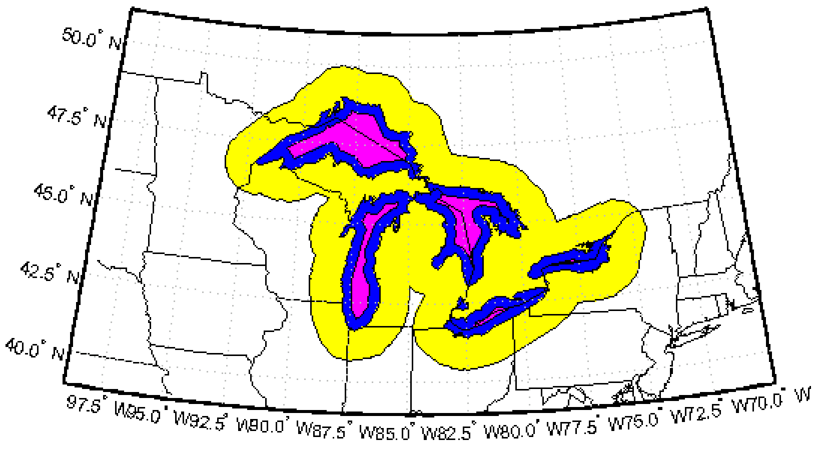
`[latb,lonb] = bufferm(lat,lon,dist,*direction*,npts,*outputformat*)` controls the format of the returned buffer zones. `outputformat 'vector'` returns `NaN`-clipped vectors. `outputformat 'cutvector'` returns `NaN`-clipped vectors with cuts connecting holes to the exterior of the polygon. `outputformat 'cell'` returns cell arrays in which each element of the cell array is a separate polygon. Each polygon can consist of an outer contour followed by holes separated with `NaN`s.

Examples

Load the coordinates for the conterminous U.S. and its great lakes. Construct a 1-degree buffer zone around the great lakes, another buffer one-third of a degree wide inside the great lakes, and display the resulting buffers over the lake and state boundaries using `geoshow`:

```
load conus
tol = 0.1; % Tolerance for simplifying polygon outlines
```

```
[reducedlat, reducedlon] = reducem(gtlakelat, gtlakelon, tol);
dist = 1;  % Buffer distance in degrees
[latb, lonb] = bufferm(reducedlat, reducedlon, dist, 'out');
[lati, loni] = bufferm(reducedlat, reducedlon, 0.3*dist, 'in');
usamap({'MN','NY'})
geoshow(latb, lonb, 'DisplayType', 'polygon',...
        'FaceColor', 'yellow')
geoshow(gtlakelat, gtlakelon,...
        'DisplayType', 'polygon', 'FaceColor', 'blue')
geoshow(lati, loni, 'DisplayType', 'polygon',...
        'FaceColor', 'magenta')
geoshow(uslat, uslon)
geoshow(statelat, statelon)
```



See Also

`polybool`

Purpose

Set camera position using geographic coordinates

Syntax

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

Description

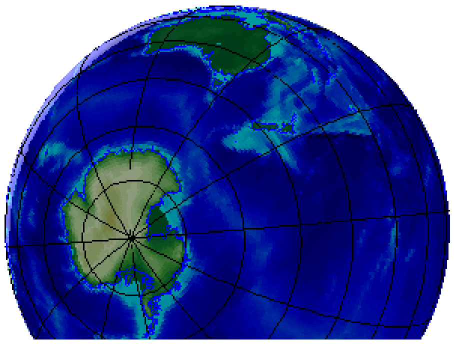
`camposm(lat,long,alt)` sets the axes `CameraPosition` property of the current map axes to the position specified in geographic coordinates. The inputs `lat` and `long` are assumed to be in the angle units of the current map axes.

`[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`

Purpose

Set camera target using geographic coordinates

Syntax

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

Description

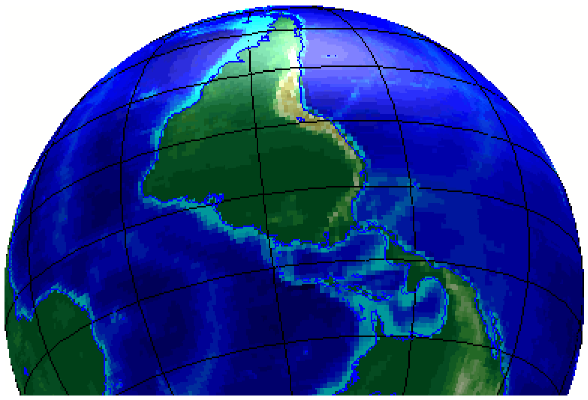
`camtargm(lat,long,alt)` sets the axes `CameraTarget` property of the current map axes to the position specified in geographic coordinates. The inputs `lat` and `long` are assumed to be in the angle units of the current map axes.

`[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`

Purpose

Set camera up vector using geographic coordinates

Syntax

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

Description

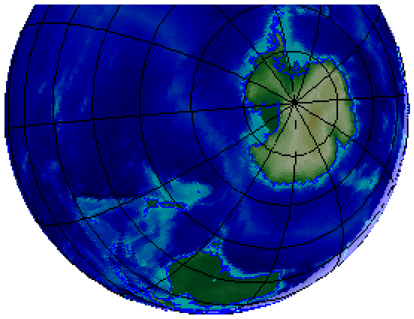
`camupm(lat,long)` sets the axes `CameraUpVector` property of the current map axes to the position specified in geographic coordinates. The inputs `lat` and `long` are assumed to be in the angle units of the current map axes.

`[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`

Purpose

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`

Purpose

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
```

Purpose

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, `NaN`s are returned.

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

See Also

`linecirc`

Purpose

Add contour labels to map contour display

Syntax

```
h1 = clabelm(c,h)
h1 = clabelm(c,h,v)
h1 = clabelm(c,h,'manual')
h1 = clabelm(c), h1 = clabelm(c,v)
```

Description

`h1 = clabelm(c,h)` rotates the labels and inserts them in line with the contour lines. The handles of the labels can be returned in `h1`.

`h1 = clabelm(c,h,v)` creates inline labels only for those levels specified in the vector `v`.

`h1 = clabelm(c,h,'manual')` places contour labels at locations you select with a mouse. You press the left mouse button (the only mouse button on a single-button mouse), or the space bar to label a contour at the closest location beneath the center of the cursor. Press the **Return** key while the cursor is within the figure window to terminate labeling. The labels are inserted in line with the contour lines.

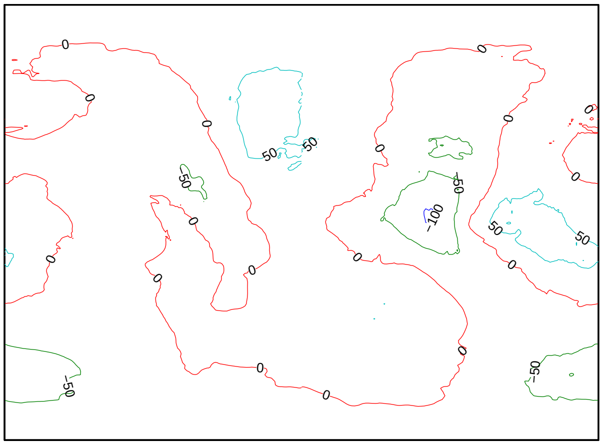
`h1 = clabelm(c)`, `h1 = clabelm(c,v)`, and `h1 = clabelm(c,'manual')` operate as above, except that instead of rotating the labels and placing them in line with the contours, the labels are upright, and a + indicates the contour line the label is annotating.

The `clabelm` function adds height labels to a two-dimensional contour plot. By default, `clabelm` labels all displayed contours and randomly selects label positions.

`c` is the contour matrix as described on the `contourm` reference page of this guide; `h` is the vector of handles for the displayed contours.

Example

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



See Also `clegendm`, `contourm`, `contour3m`, `clabel` (MATLAB function)

Purpose

Add legend labels to map contour display

Syntax

```
clegendm(cs,h)
clegendm(cs,h,pos)
clegendm(...,unitstr)
clegendm(...,str)
```

Description

The `clegendm` function displays a legend for a displayed contour map.

`clegendm(cs,h)` displays a legend for the contour map defined by the two-column contour definition matrix, `cs`, and the handle(s) `h`. Both of these inputs are produced as the outputs of either `contourm` or `contour3m`.

`clegendm(cs,h,pos)` allows you to specify the position of the legend in the display. The input `pos` can be any of the following integers, with the indicated result:

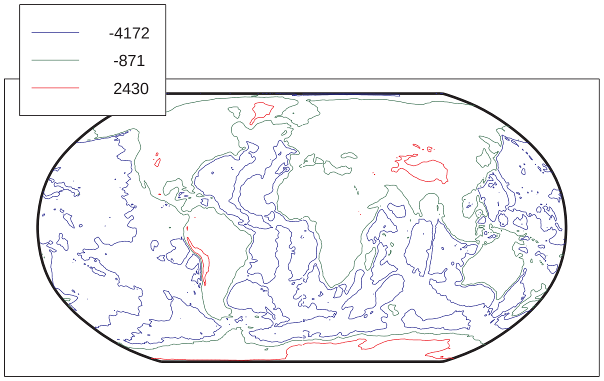
0	Automatic placement (this is the default)
1	Upper right corner
2	Upper left corner
3	Lower left corner
4	Lower right corner
-1	To the right of the plot

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

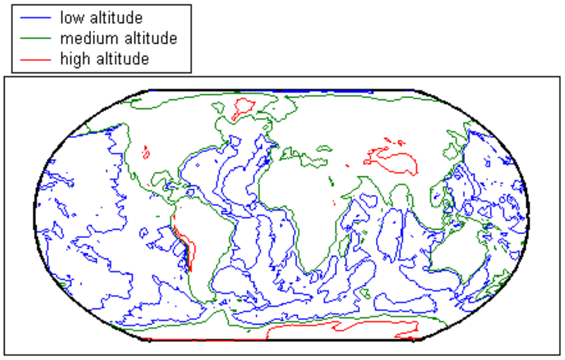
`clegendm(...,str)` uses the strings specified in cell array `str` to label the legend. The cell array must have same number of entries as `h`.

Examples

```
load topo
axesm robinson; framem
[cs,h] = contourm(topo,topolegend,3);
clegendm(cs,h,2)
```



```
% Example showing legend string usage
% Load topographic data measured in meters
load topo;
axesm robinson; framem
[cs,h] = contourm(topo,topolegend,3);
% Create Legend with user specified string
str = {'low altitude','medium altitude','high altitude'}
clegendm(cs,h,2,str);
```



See Also `clabelm`, `contourm`, `contour3m`, `contourc` (MATLAB function)

Purpose

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

Purpose

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` clears all displayed map objects and converts the map axes to standard axes. This is equivalent to `cla reset`.

See Also

`cla` (MATLAB function), `clmo`, `handlem`, `hidem`, `namem`, `showm`, `tagm`

Purpose

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

Purpose

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

Closing a polygon in plane coordinates

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

whos
```

```
  Name          Size            Bytes  Class     Attributes

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

Closing a polygon in latitude-longitude coordinates

```
% Construct a two-part polygon based on coast.mat.  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.
c = load('coast.mat');
[latparts, lonparts] = polysplit(c.lat, c.long);
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
```

See Also

`isshapemultipart`, `removeextrananseparators`

Purpose

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`

Purpose

All possible combinations of set of values

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
```

Remarks

This is a recursive function.

Purpose

Project 3-D comet plot on map axes

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`

Purpose

Project 2-D comet plot on map axes

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`

Purpose

Project 3-D contour plot of map data

Syntax

```
contour3m(Z,R)
contour3m(lat,lon,Z)
contour3m(Z,R,n) or contour3m(lat,lon,Z,n)
contour3m(Z,V,R) or contour3m(lat,lon,Z,V)
contour3m(..., linespec)
contour3m(..., prop1, val1, prop2, val2,...)
C = contour3m(...)
[C,h] = contour3m(...)
```

Description

`contour3m(Z,R)` displays a contour plot of the regular M-by-N data grid, `Z`. `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 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. 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. For more information about referencing vectors and matrices, see the section “Understanding Raster Geodata” on page 2-33 in the User’s Guide.

`contour3m(lat,lon,Z)` displays a contour plot of the geolocated M-by-N data grid, `Z`. `lat` and `lon` can be the size of `Z` or can specify the corresponding row and column dimensions for `Z`.

`contour3m(Z,R,n) or contour3m(lat,lon,Z,n)` where `n` is a scalar, draws `n` contour levels.

`contour3m(Z,V,R)` or `contour3m(lat,lon,Z,V)` where `V` is a vector, draws contours at the levels specified by the input vector `v`. Use `V = [v v]` to compute a single contour at level `v`.

`contour3m(..., linespec)` uses any valid LineSpec string to draw the contour lines.

`contour3m(..., prop1, val1, prop2, val2,...)` specifies property/value pairs that modify LINE graphics properties. Property names can be abbreviated and are case-insensitive.

`C = contour3m(...)` returns a standard contour matrix, `C`, with the first row representing longitude data and the second row representing latitude data.

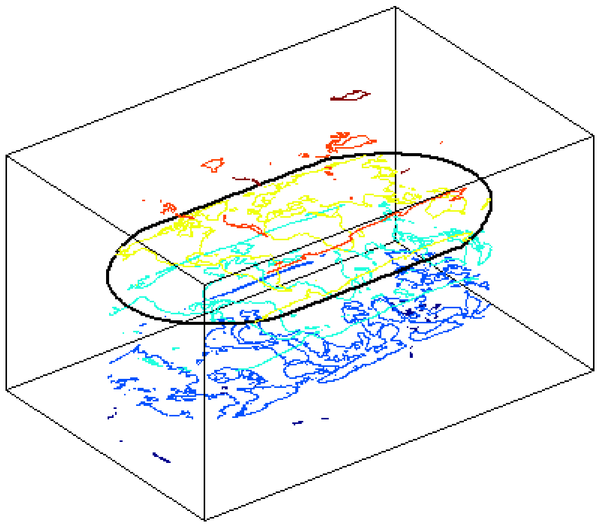
`[C,h] = contour3m(...)` returns the contour matrix and the line handles to the contour lines drawn onto the current axes.

Examples

Example 1

Make a default contour map of world topography data

```
load topo
axesm robinson; framem; view(3)
contour3m(topo,topolegend)
set(gca,'DataAspectRatio',[1 1 3000])
```

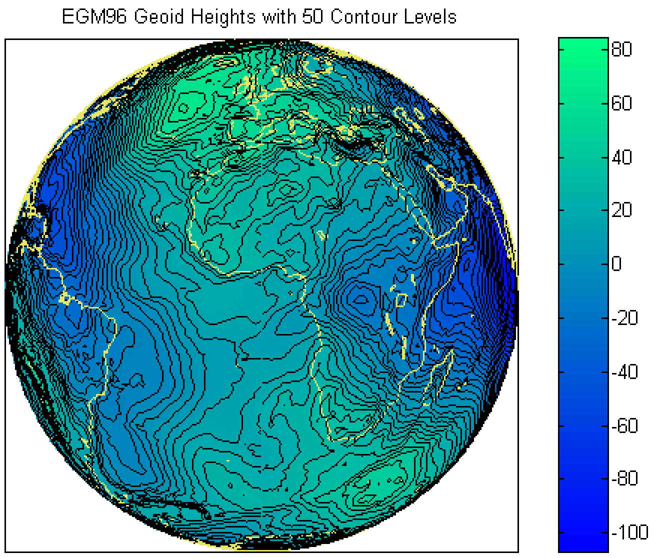


Example 2

Contour EGM96 geoid heights as a 3-D surface with 50 levels, set contour patch edge color to black, show the geoid surface under and coastlines above the contour lines on an orthographic projection.

```
load geoid
axesm ortho
% Contour the geoid surface in black using 50 levels
[c,h]=contour3m(geoid, geoidrefvec, 50,'EdgeColor','black');
% Add the geoid surface.
hold on
geoshow(geoid,geoidrefvec,'DisplayType','surface')
% Add a title and colorbar.
title('EGM96 Geoid Heights with 50 Contour Levels');
colorbar
% Set the colormap to blue - green
colormap('winter')
% Set the Z-datum so that all contours show
zdatam(handlem('surface'),min(geoid(:)));
```

```
% Get world coastlines and plot them in gold
landareas = shaperead('landareas.shp','UseGeoCoords',true);
geoshow(landareas,'DisplayType','Polygon',...
  'FaceColor','None','EdgeColor',[.9 .9 .4])
```

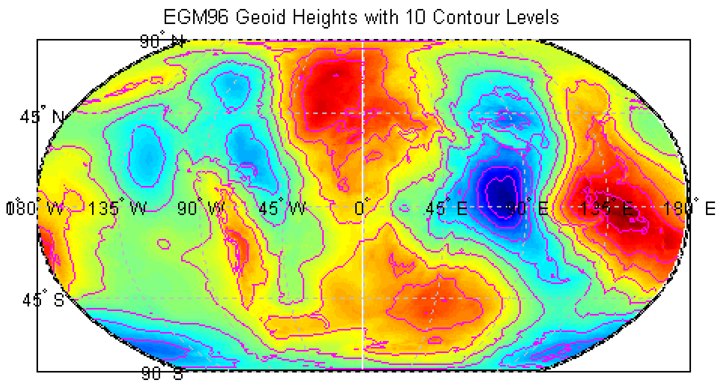


Example 3

Display the EGM96 geoid height contours in a default world map.

```
load geoid
figure
worldmap('world');

% Contour the geoid height with 10 levels and
% set the color to magenta.
[c,h]=contour3m(geoid, geoidrefvec, 10,'m');
```

```
% Add the geoid surface.
hold on
geoshow(geoid,geoidrefvec,'DisplayType','surface')

% Set the surface to the minimum height of the geoid.
% to keep the contours visible.
zdatam(handlem('surface'),min(geoid(:)));

% Add a title.
title('EGM96 Geoid Heights with 10 Contour Levels');
```



See Also

clabel, clabelm, clegendm, contour, contour3, contourm, geoshow, plot

Purpose Contour colormap and colorbar current axes

Syntax

```
contourcmap(cdelta,cmap)
contourcmap(cdelta,cmap,property,value,...)
hcb = contourcmap(...)
```

Description

`contourcmap(cdelta,cmap)` creates a contour colormap for the current axes. A contour colormap is a colormap with color changes aligned to the color data. If `cdelta` is a scalar, contours are generated at multiples of `cdelta`. If `cdelta` is a vector of evenly spaced values, contours are generated at those values. The string input *`cmap`* is the name of the colormap function used in the surface. Valid entries for *`cmap`* include `'pink'`, `'hsv'`, `'jet'`, or any similar colormap function.

`contourcmap(cdelta,cmap,property,value,...)` allows you to add a colorbar and control the colorbar's properties. You turn the colorbar on with the property-value pair `'Colorbar'` and `'on'`. 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'` or `'ends'`. Property `'SourceObject'` controls which object is used to determine the color limits for the colormap. The `SourceObject` value is the handle of a currently displayed object. If omitted, `gca` is used. Other valid property-value pairs are any properties and values that can be applied to the title and labels of the colorbar axes.

`hcb = contourcmap(...)` returns a handle to the colorbar.

Example

Create a colormap and set color limits to make the color changes occur at multiples of 20 for the geoid.

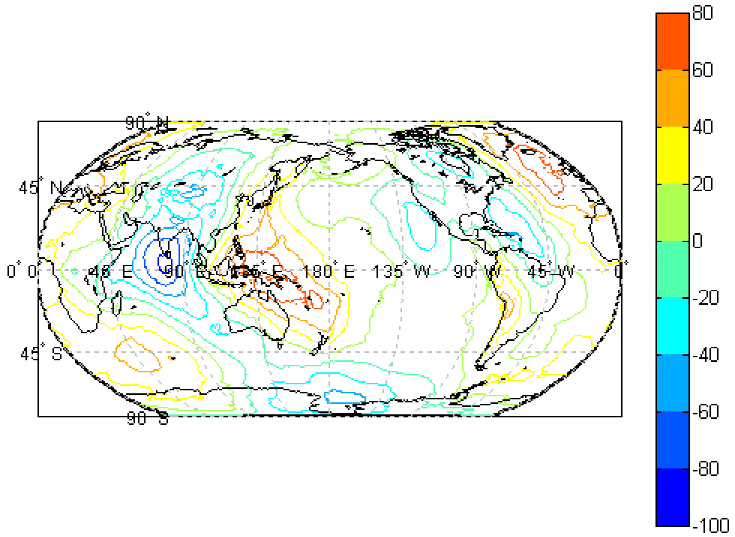
```
load geoid
figure
worldmap(geoid, geoidrefvec)
contourm(geoid, geoidrefvec, -120:20:100);
```

Add a colorbar, controlling the labels and font properties.

```
contourcmap(20, 'jet', 'colorbar', 'on');
```

Load and plot coastlines on top.

```
load coast
plotm(lat, long, 'k')
```



See Also

`contourfm`, `contourm`, `lcolorbar`, `demcmap`

Purpose

Project filled 2-D contour plot of map data

Syntax

```
contourfm(lat,lon,Z)
contourfm(Z,R)
contourfm(lat,lon,Z,n,...)
contourfm(...,v,...)
contourfm(...,LineSpec)
c = contourfm(...)
[c,h] = contourfm(...)
```

Description

`contourfm(lat,lon,Z)` produces a contour plot of map data projected onto the current map axes. The input latitude and longitude vectors can be the size of `Z` (as in a geolocated data grid), or can specify the corresponding row and column dimensions for the map.

`contourfm(Z,R)` creates a contour plot of the regular data grid, `Z`. `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 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. For more information about referencing vectors and matrices, see the section "Understanding Raster Geodata" on page 2-33 in the User's Guide.

`contourfm(lat,lon,Z,n,...)` draws `n` contour levels, where `n` is a scalar.

`contourfm(...,v,...)` draws contours at the levels specified by the input vector `v`.

`contourfm(...,`*`LineSpec`*`)` uses any valid *`LineSpec`* string to draw the contour lines.

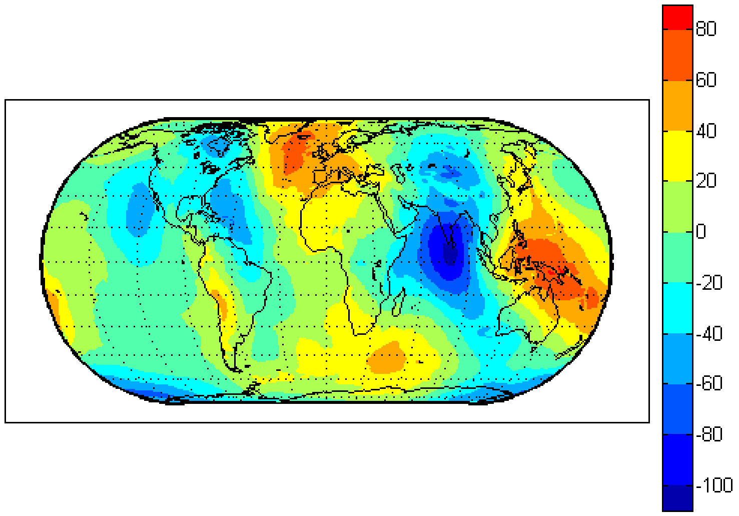
`c = contourfm(...)` returns a standard contour matrix, with the first row representing longitude data and the second row representing latitude data.

`[c,h] = contourfm(...)` returns the contour matrix and an array of handles to the contour patches drawn.

Examples

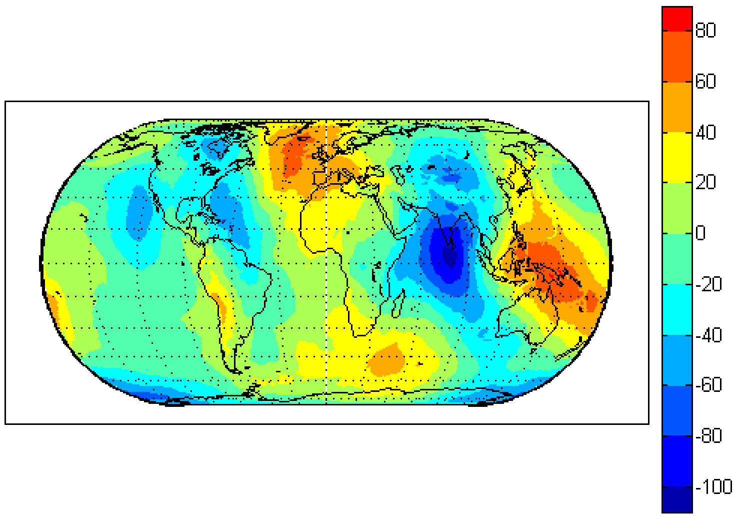
Plot the Earth's geoid with filled contours. The data is in meters.

```
load geoid
figure
axesm eckert4
framem;gridm
load coast
plotm(lat,long,'k')
caxis([-120 100]);colormap(jet(11));colorbar
contourfm(geoid,geoidrefvec,-120:20:100);
```



You can reproduce the filled contour display by using a surface instead of the patches created by `contourfm`.

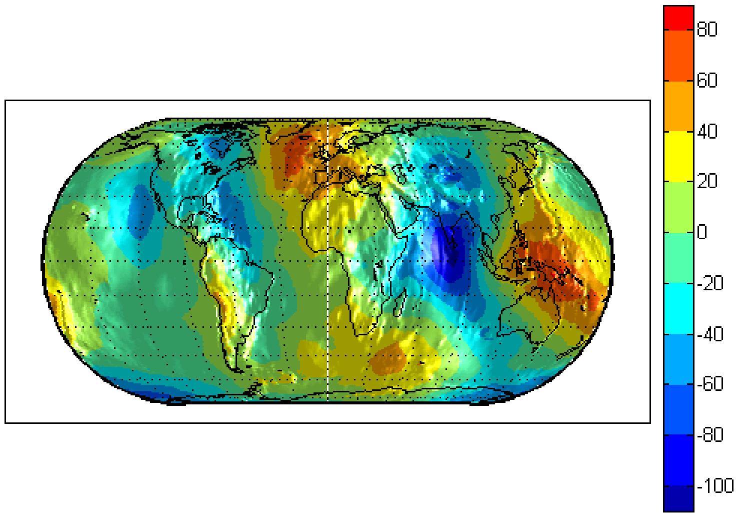
```
figure
axesm eckert4
framem;gridm
load coast
plotm(lat,long,'k')
meshm(geoid,geoidrefvec,size(geoid),'Facecolor','interp')
contourcmap(20,'jet');colorbar
```



Surfaces also allow use of lighting to bring out the smaller variations in the data.

```
clmo surface
meshm(geoid,geoidrefvec,size(geoid),geoid,'Facecolor','interp')
light;lighting phong; material(0.6*[ 1 1 1])
```

```
set(gca,'dataaspectratio',[ 1 1 200])
gridm reset
zdatam(handlem('line'),max(geoid(:)))
```



Limitations

`contourfm` might not fill properly with azimuthal projections.

Remarks

By default, filled contour patches are displayed with no edge lines. To add contour lines, supply a `lineSpec` or specify an `EdgeColor` to `contourfm`. An `EdgeColor` may also be `set` later.

In most circumstances, contour plots made with surfaces are preferable to the filled patches created by `contourfm`. Surfaces are rendered more quickly and take less time to project and reproject. The use of surfaces also allows surface lighting to create shaded 3-D maps.

See Also

`contourm`, `contour3m`, `clabelm`, `meshm`, `surfm`

Purpose

Project 2-D contour plot of map data

Syntax

```
contourm(Z, R)
contourm(lat, lon, Z)
contourm(Z, R, n)
contourm(Z, R, V) or contourm(lat, lon, Z, V)
contourm(..., linespec)
contourm(..., prop1, val1, prop2, val2,...)
C = contourm(...)
[C,h] = contourm(...)
```

Description

`contourm(Z, R)` creates a contour plot of the regular M-by-N data grid, `Z`. `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 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. For more information about referencing vectors and matrices, see the section "Understanding Raster Geodata" on page 2-33 in the User's Guide.

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.

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

`contourm(Z, R, n)` or `contourm(lat,lon,Z,n)` where `n` is a scalar, draws `n` contour levels.

`contourm(Z, R, V)` or `contourm(lat, lon, Z, V)` where `V` is a vector, 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(..., prop1, val1, prop2, val2,...)` specifies property/value pairs that modify `contourgroup` graphics properties. Property names can be abbreviated and are case-insensitive.

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

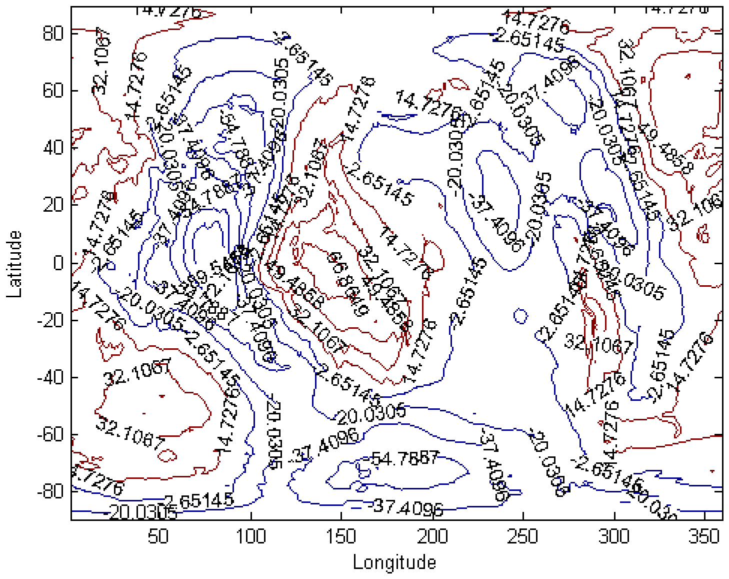
`[C,h] = contourm(...)` returns the contour matrix and the handle to the contour patches drawn onto the current axes. The handle is type `hggroup`.

Examples

Example 1

Contour EGM96 geoid heights as dotted lines and with 10 levels and set the contour labels on.

```
load geoid
figure
contourm(geoid, geoidrefvec, 10, ':','ShowText','on');
xlabel('Longitude');
ylabel('Latitude');
```

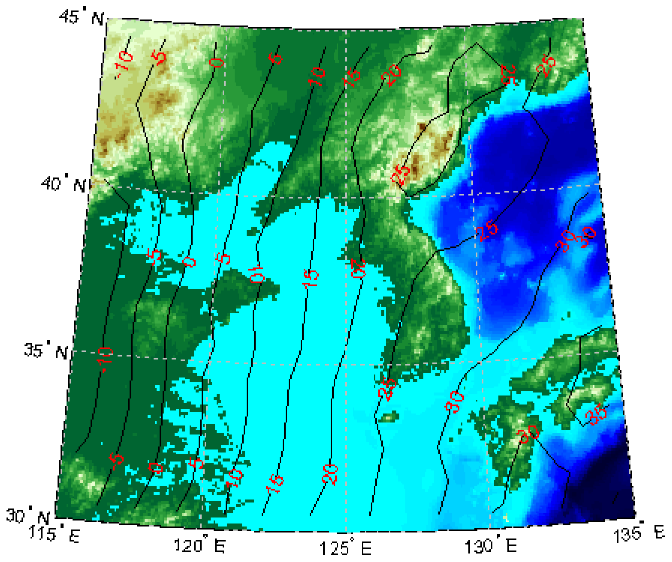


Example 2

Contour the Korean bathymetry and elevation data:

```
% Load the data.
load korea;
load geoid;

% Create a worldmap of Korea.
figure
worldmap(map, refvec);

% Display the digital elevation data and colormap.
geoshow(map, refvec, 'DisplayType', 'surface');
colormap(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 to the contours.
ht=clabel(c,h);
set(ht,'Color','r');
```

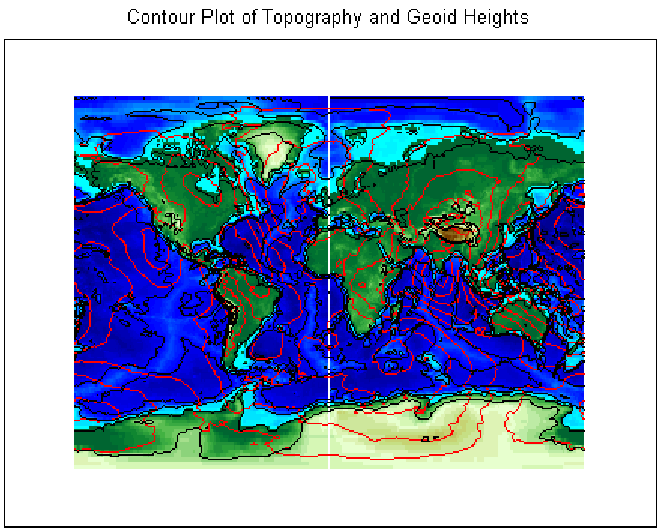


Example 3

Contour the geoid and topography heights:

```
% Load the data.
load topo
load geoid

% Create a Miller projection with geoid contours as red lines,
% and topography contours as black lines.
figure; axesm miller
hold on
contourm(geoid, geoidrefvec, 'r');
contourm(topo,  topolegend, 'k');
```

```
% Add the topograpy surface and color map.
geoshow(topo, topolegend, 'DisplayType', 'surface')
colormap(demcmap(topo))

% Set the surface as the lowest value of topo
% to keep the contour lines visible.
zdatam(handlem('surface'), min(topo(:)))

% Add a title
title('Contour Plot of Topography and Geoid Heights');
```



See Also

clabelm, clegendm, contour, contourc, contour3, contour3m, geoshow, plot

Purpose

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 1-by-2 ellipsoid vector of the form `[semimajoraxis eccentricity]`. (The `almanac` function offers a set of built-in ellipsoids covering most widely available map data.)

`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`.

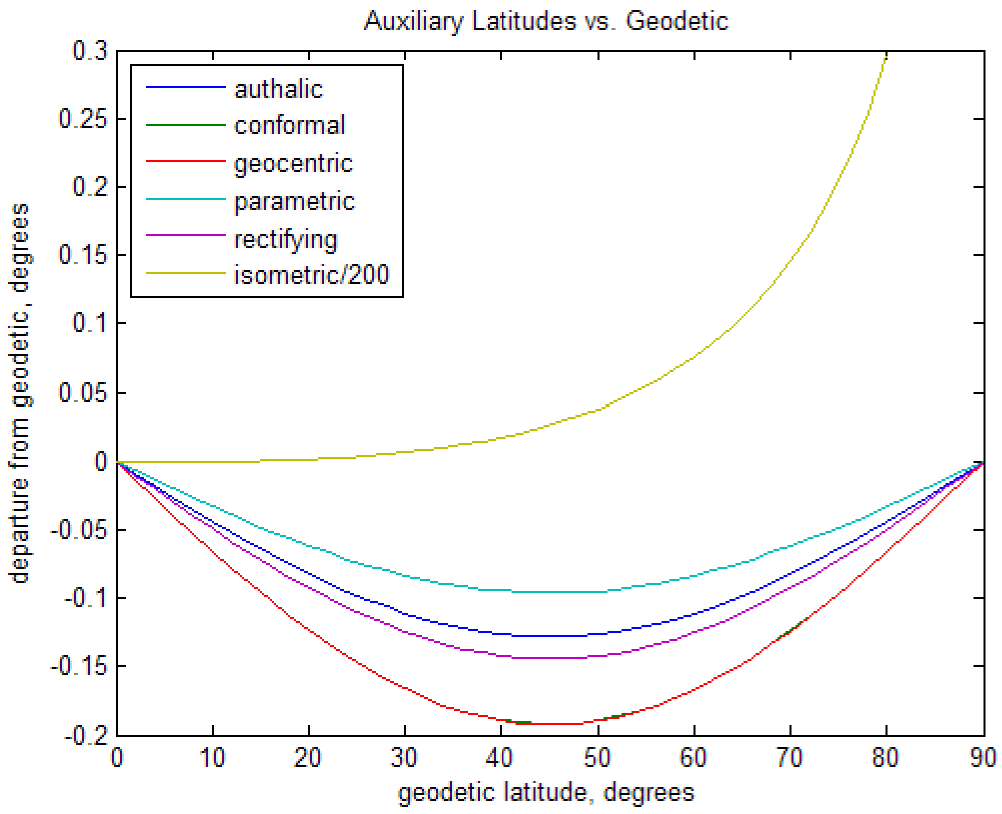
Latitude Type	Description
`geodetic`	The geodetic latitude is the angle that a line perpendicular to the surface of the ellipsoid at the given point makes with the equatorial plane.
`authalic`	The authalic latitude maps an ellipsoid to a sphere while preserving surface area. Authalic latitudes are used 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`.

Example

```
% 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 = almanac('earth','ellipsoid','m','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

`almanac`, `rsphere`

Purpose

Cross-fix positions from bearings and ranges

Syntax

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

Description

`[newlat,newlon] = crossfix(lat,long,az)` returns the intersection points of all pairs of great circles passing through the points given by the column vectors `lat` and `long` 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,long,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,long,az_range,case,drlat,drlong)` resolves the ambiguities when there is more than one intersection between two objects. The scalar-valued `drlat` and `drlong` provide the location of an estimated (dead reckoned) position. The outputs `newlat` and `newlong` 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,long,az,`*`units`*`)`, `[newlat,newlon] = crossfix(lat,long,az_range,case,`*`units`*`)`, `[newlat,newlon] = crossfix(lat,long,az_range,drlat,drlong,`*`units`*`)`, and `[newlat,newlon] = crossfix(lat,long,az_range,case,drlat,drlong,`*`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`.

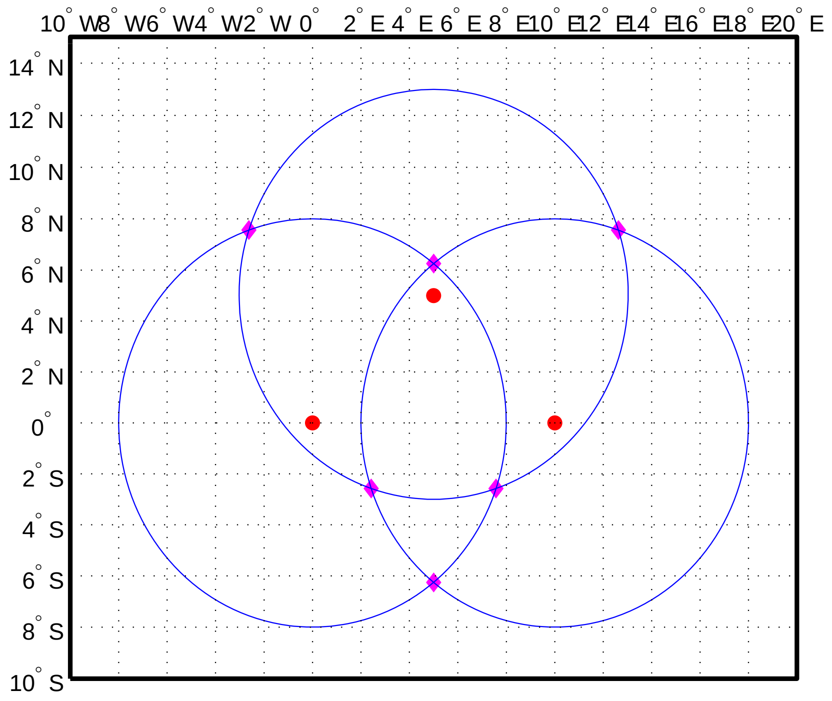
This function calculates the points of intersection between a set of objects taken in pairs. Given great circle azimuths and/or ranges from input points, the locations of the possible intersections are returned. This is different from the navigational function `navfix` in that `crossfix` uses great circle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

Example

Where do the small circles defined as all points 8º in distance from the points (0º,0º), (5ºN,5ºE), and (0º,10ºE)" intersect?

```
figure('color','w');
ha = axesm('mapproj','mercator', ...
    'maplatlim',[-10 15],'maplonlim',[-10 20],...
    'MLineLocation',2,'PLineLocation',2);
axis off, gridm on, framem on;
mlabel on, plabel on;
latpts = [0;5;0];          % Define latitudes of three arbitrary points
lonpts = [0;5;10];         % Define longitudes of three arbitrary points
radii = [8;8;8];           % Define three radii, all 8 degrees

% Obtain intersections of imagined small circles around these points
[newlat,newlon] = crossfix(latpts,lonpts,radii,[0;0;0])

% Draw red circle markers at the given points
geoshow(latpts,lonpts,'DisplayType','point',...
    'markeredgecolor','r','markerfacecolor','r','marker','o')

% Draw magenta diamond markers at intersection points just found
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','-')
```

The diagram shows why there are six intersections:



If a dead reckoning position is provided, say (0º,5ºE), then one from each pair is returned (the closest one):

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

newlat =
   -2.5744
    6.2529
   -2.5744

newlong =
    7.5770
    5.0000
    2.4230
```

See Also

`gcxgc`, `gcxsc`, `scxsc`, `rhxrh`, `polyxpoly`, `navfix`

daspectm

Purpose

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

Description

`daspectm(zunits)` sets the figure `'DataAspectRatio'` property 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 string 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.

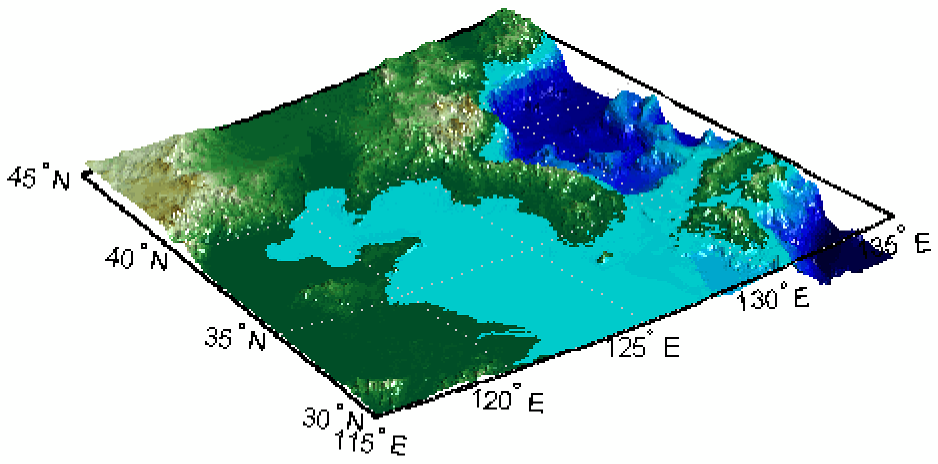
`daspectm(zunits,vfac,lat,long,az,gunits)` also specifies the units in which the geographic position and direction are given. If omitted, `'degrees'` is assumed.

`daspectm(zunits,vfac,lat,long,az,gunits,radius)` uses the last input to determine the radius of the sphere. If `radius` is a string, then it is evaluated as an `almanac` body to determine the spherical radius. If numerical, it is the radius of the desired sphere in `zunits`. If omitted, the default radius of the Earth is used.

Examples

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

```
load korea
[latlim,lonlim] = limitm(map,refvec);
worldmap(latlim,lonlim)
meshm(map,refvec,size(map),map)
demcmap(map)
view(3)
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

Purpose

Read selected DCW worldwide basemap data

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).

Remarks

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 12-142 in the reference page for `displaym`.

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'});
```

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

Purpose

Search DCW worldwide basemap gazette file

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.

Remarks

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

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
```

See Also `dcwdata`, `dcwread`, `dcwrhead`, `mlayers`, `updategeostruct`

Purpose

Read DCW worldwide basemap file

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.

Remarks

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.

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: '014'
          ELLIPSOID: 'WGS 84'
   ELLIPSOID_DETAIL: 'A=6378137,B=6356752 Meters'
     VERT_DATUM_REF: 'MEAN SEA LEVEL'
    VERT_DATUM_CODE: '015'
        SOUND_DATUM: 'MEAN SEA LEVEL'
   SOUND_DATUM_CODE: '015'
     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
```

See Also dcwdata, dcwgaz, dcwrhead

Purpose

Read DCW worldwide basemap file headers

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.

Remarks

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.

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,-,-,:;
```

See Also

dcwdata, dcwgaz, dcwread

Purpose

Initialize or reset map projection structure

Syntax

```
mstruct = defaultm(projection)
mstruct = defaultm(mstruct)
```

Description

`mstruct = defaultm(`*`projection`*`)` initializes a map projection structure. *`projection`* is a string containing the name of a projection M-function.

`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" on page 8-59 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`

Purpose

Convert distance from degrees to kilometers, nautical miles, or statute miles

Syntax

```
km = deg2km(deg)
nm = deg2nm(deg)
sm = deg2sm(deg)
km = deg2km(deg,radius)
nm = deg2nm(deg,radius)
sm = deg2sm(deg,radius)
km = deg2km(deg,sphere)
nm = deg2nm(deg,sphere)
sm = deg2sm(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.

`nm = deg2nm(deg)` and `sm = deg2sm(deg)` work identically, except that the output units are nautical miles and statute miles, respectively.

`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.

For `nm = deg2nm(deg,radius)` and `sm = deg2sm(deg,radius)`, make sure your input radius is in the appropriate units.

`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.

`nm = deg2nm(deg,`*`sphere`*`)` and `sm = deg2sm(deg,`*`sphere`*`)` work identically, except that the output units are nautical miles and statute miles, respectively.

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

`deg2nm`, `degtorad`, `deg2sm`, `km2deg`, `sm2deg`

Purpose

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.

Example

```
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`, `degtorad` `degrees2dms`, `radtodeg`

degrees2dms

Purpose

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.

Example

```
format long g
angleInDegrees = [ 30.8457722555556; ...
                  -82.0444189583333; ...
                   -0.504756513888889;...
                    0.00411666666666667];
dms = degrees2dms(angleInDegrees)

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{:})

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');
a = strread(str,'%s',-1,'delimiter',newline);
for k = 1:numel(a)
    str2angle(a{k})
end

ans =
          30.8457722555556

ans =
         -82.0444189583333

ans =
        -0.504756513888889

ans =
       0.00411666666666667
```

See Also dms2degrees, degtorad degrees2dm, radtodeg

Purpose

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`

Purpose

Colormaps appropriate to terrain elevation data

Syntax

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

Description

`demcmap(Z)` creates and assigns a colormap for elevation data grid `Z`. The colormap has the number of land and sea colors in the same proportions as the maximum elevations and depths in the data grid. With no output arguments, the colormap is applied to the current figure and the color axis is set so that the interface between the land and sea is in the right place.

`demcmap(Z,ncolors)` makes a colormap with a length of `ncolors`. The default value is 64.

`demcmap(Z,ncolors,cmapsea,cmapland)` allows the default colors for sea and land to be replaced. The colors in the created colormap are interpolated from the RGB color matrix inputs, which can be of any length. You can retain default colors for either land or sea by providing an empty matrix in place of the color matrices. You can specify the current figure colormap by entering the string `'window'` in place of either RGB matrix.

`demcmap(`*`color`*`,Z,spec)` uses the *`color`* string to define a colormap. If the string is set to `'size'`, `spec` is the length of the colormap. If it is set to `'inc'`, `spec` is the size of the altitude range assigned to each color. If omitted, *`color`* is `'size'` by default.

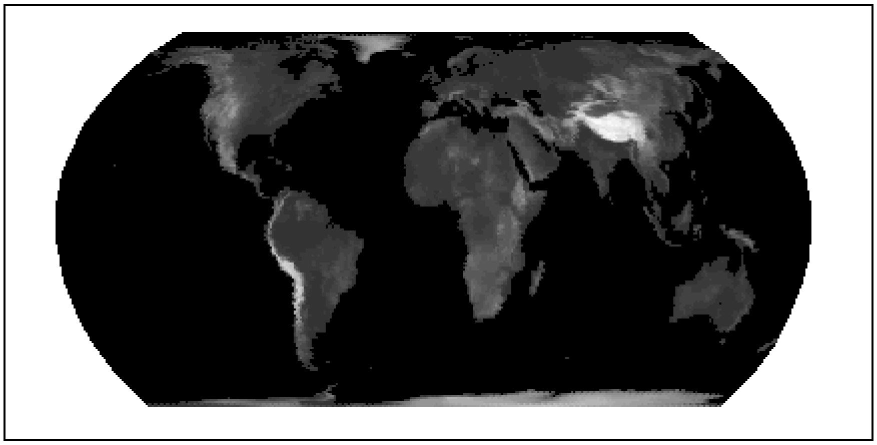
`demcmap(`*`color`*`,Z,spec,cmapsea,cmapland)` allows for both coloring options along with specified colors.

Examples

Display the world topographical map using grayscale colors:

```
load topo
axesm hatano
meshm(topo,topolegend)
```

```
demcmap(topo,64,[0 0 0],[.2 .2 .2; 1 1 1])
```



See Also `caxis`, `colormap`, `meshlsrm`, `meshm`, `surflsrm`, `surfm`

Purpose

Departure of longitudes at specified latitudes

Syntax

```
dist = departure(long1,long2,lat)
dist = departure(long1,long2,lat,geoid)
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,geoid)` computes the departure assuming that the input points lie on the ellipsoid defined by the input `geoid`. The `geoid` vector is of the form `[semimajor axes, 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 `geoid` vector.

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 60º is not exactly twice the 0º value:

```
distance = departure(0, 10, 0, almanac('earth', 'ellipsoid', 'nm'))

distance =
  601.0772
distance = departure(0, 10, 60, almanac('earth', 'ellipsoid', 'nm'))

distance =
  299.7819
```

See Also

`distance` | `stdm`

Purpose

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 12-142 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 (similar to the MATLAB `strmatch` function)
- `'findstr'` — Search within the tag (similar to the MATLAB `findstr` function)
- `'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" on page 2-16.

Remarks

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`

Purpose

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`

Purpose

Distance between points on sphere or ellipsoid

Syntax

```
[dist,az] = distance(lat1,lon1,lat2,lon2)
[dist,az] = distance(lat1,lon1,lat2,lon2,ellipsoid)
[dist,az] = distance(lat1,lon1,lat2,lon2,units)
[dist,az] = distance(lat1,lon1,lat2,lon2,ellipsoid,units)
[dist,az] = distance(pt1,pt2)
[dist,az] = distance(pt1,pt2,ellipsoid)
[dist,az] = distance(pt1,pt2,units)
[dist,az] = distance(pt1,pt2,ellipsoid,units)
[dist,az] = distance(track,...)
```

Description

`[dist,az] = distance(lat1,lon1,lat2,lon2)` computes the great circle distance(s) and azimuth(s) between pairs of points on the surface of a sphere. The input latitudes and longitudes, `lat1`, `lon1`, `lat2`, and `lon2`, are in degrees and can be scalars or arrays of equal size. The distance `dist` is expressed in degrees of arc length and will have the same size as the input arrays. Azimuth `az` is clockwise from north, from the first point to the second point. When given a combination of scalar and array inputs, the scalar inputs are automatically expanded to match the size of the arrays.

`[dist,az] = distance(lat1,lon1,lat2,lon2,ellipsoid)` computes the geodesic distance and azimuth assuming that the points lie on the reference ellipsoid defined by the input `ellipsoid`. The ellipsoid vector is of the form `[semimajor axis,eccentricity]`. The output `dist` is expressed in the same distance units as the semimajor axis of the ellipsoid vector.

`[dist,az] = distance(lat1,lon1,lat2,lon2,`*`units`*`)` uses the string *`units`* to define the angle units of the input latitudes and longitudes and the outputs `dist` and `az`. The *`units`* string may equal `'degrees'` (the default value) or `'radians'`.

`[dist,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 vector.

`[dist,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 `dist = distance(pt1(:,1),pt1(:,2),pt2(:,1),pt2(:,2))`.

`[dist,az] = distance(pt1,pt2,ellipsoid)`,

`[dist,az] = distance(pt1,pt2,`*`units`*`)`, and

`[dist,az] = distance(pt1,pt2,ellipsoid,`*`units`*`)` are all valid calling forms.

`[dist,az] = distance(`*`track`*`,...)` specifies whether great circle distances or rhumb line distances are desired. Great circle distances, the default, are indicated with the standard *`track`* string `'gc'`. Rhumb line distances are indicated with the standard *`track`* string `'rh'`.

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.

```
dist = distance('gc',[37,-76],[37,-9])

dist =
   52.3094

dist = distance('rh',[37,-76],[37,-9])

dist =
    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 90º, 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
dist = distance(37,-76,67,-76)
dist =
     30.0000

% rhumb line distance
dist = distance('rh',37,-76,67,-76)

dist =
     30.0000
```

The distances are the same, 30º, or about 1800 nautical miles (there are about 60 nautical miles in a degree of arc length).

Algorithm

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 `dist` and `az` are set to NaN for the "problem pairs."

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

```
[dist az] = distance(...);
```

rather than the slower

```
dist = distance(...)
az = azimuth(...)
```

To express the output `dist` as an arc length in either degrees or radians, omit the `ellipsoid` argument. This is possible only on a sphere. If `ellipsoid` is supplied, `dist` is a distance expressed in the same units as the semimajor axis of the ellipsoid. Specify `ellipsoid` as `[R 0]` to compute `dist` as a distance on a sphere of radius `R`, with `dist` having the same units as `R`.

See Also

`almanac` | `azimuth` | `elevation` | `reckon` | `track` | `track1` | `track2` | `trackg`

How To

- "Great Circles, Rhumb Lines, and Small Circles" on page 3-32

Purpose

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
```

Remarks

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`

Purpose

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 12-155 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.

Remarks

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 the there is no appropriate direct conversion utility, or you won't know the valus 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')
```

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
```

See Also

`deg2km`, `deg2nm`, `deg2sm`, `km2deg`, `km2nm`, `km2rad`, `km2sm`, `nm2deg`, `nm2km`, `nm2rad`, `nm2sm`, `rad2km`, `rad2nm`, `rad2sm`, `sm2deg`, `sm2km`, `sm2nm`, `sm2rad`, `unitsratio`

dm2degrees

Purpose

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.

Example

```
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`, `degtorad` `dms2degrees`, `str2angle`

dms2degrees

Purpose

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.

Example

```
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`, `degtorad` `dm2degrees`, `str2angle`

dreckon

Purpose

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” on page 10-11 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`

Purpose

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.

Example

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`

Purpose

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`).

Example

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`

Purpose

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(dirname, 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(dirname, samplefactor, latlim, lonlim)` reads and concatenates data from multiple files within a DTED CD-ROM or directory structure. The `dirname` input is a string with the name of a directory containing the DTED directory. Within the DTED directory are subdirectories for each degree of longitude, each of which contain files for each degree of latitude. For DTED CD-ROMs, `dirname` 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.

Remarks

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 directory 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 digitial 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 the following documentation at the MathWorks Web site:

```
http://www.mathworks.com/support/tech-notes/2100/2101.html
```

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.

Examples

```
[Z,refvec] = dted('n38.dt0');
[Z,refvec,UHL,DSI,ACC] = dted('n38.dt0',1,[38.5 38.8],...
  [-76.8 -76.6]);
[Z,refvec,UHL,DSI,ACC] = dted('f:',1,[38.5 38.8],...
  [-76.8 -76.6]);
```

See Also

`usgsdem`, `gtopo30`, `tbase`, `etopo`

Purpose

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 (directory 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`

eastof

Purpose

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)`.

Purpose

Flattening of ellipse with given eccentricity

Syntax

```
flattening = ecc2flat(eccentricity)
```

Description

`flattening = ecc2flat(eccentricity)` returns the equivalent flattening for the input eccentricities. If the input, `eccentricity`, is a two-column vector, only the second column is used. This allows the standard two-element ellipsoid vectors to be used as rows of the input, because the second element of these vectors is the eccentricity. In all other cases, all columns of the input are used.

Flattening and eccentricity are two methods of defining an ellipsoid.

Example

```
flattening = ecc2flat(almanac('earth','ellipsoid'))

flattening =
    0.0034
```

See Also

`almanac`, `ecc2n`, `majaxis`, `flat2ecc`

Purpose

n-value of ellipse with given eccentricity

Syntax

```
n = ecc2n(eccentricity)
```

Description

`n = ecc2n(eccentricity)` returns the equivalent *n* for the input eccentricities. If the input, `eccentricity`, is a two-column vector, only the second column is used. This allows the standard two-element ellipsoid vectors to be used as rows of the input, because the second element of these vectors is the eccentricity. In all other cases, all columns of the input are used.

Eccentricity and the parameter *n* are two methods of defining an ellipsoid. The definition of *n* is

(semimajor axis – semiminor axis)/(semimajor axis + semiminor axis)

Example

```
n = ecc2n(almanac('earth','ellipsoid'))
n =
    0.00167922039463
```

See Also

`almanac`, `ecc2flat`, `majaxis`, `n2ecc`

Purpose

Convert geocentric (ECEF) to geodetic coordinates

Syntax

```
[phi,lambda,h] = ecef2geodetic(x,y,z,ellipsoid)
```

Description

`[phi,lambda,h] = ecef2geodetic(x,y,z,ellipsoid)` converts geocentric Cartesian coordinates, stored in the coordinate arrays `x`, `y`, `z`, to geodetic coordinates `phi` (geodetic latitude in radians), `lambda` (geodetic longitude in radians), and `h` (height above the ellipsoid). The geodetic coordinates refer to the reference ellipsoid specified by `ellipsoid` (a row vector with the form `[semimajor axis, eccentricity])`. Arrays `x`, `y`, `z`, and `h` must use the same units as the semimajor axis. `x`, `y`, `z`, `phi`, `lambda`, and `h` must have the same shape.

Definitions

For a definition of the geocentric system, also known as Earth-Centered, Earth-Fixed (ECEF), see the help for `geodetic2ecef`.

See Also

`ecef2lv` | `geodetic2ecef` | `lv2ecef`

Purpose

Convert geocentric (ECEF) to local vertical coordinates

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 row vector with 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

`ecef2geodetic` | `elevation` | `geodetic2ecef` | `lv2ecef`

Purpose

Read 15-minute gridded geoid heights from EGM96

Syntax

```
[N, refvec] = egm96geoid(samplefactor)
[N, refvec] = egm96geoid(samplefactor,latlim,lonlim)
```

Description

`[N, 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 `N` along with referencing vector `refvec`. At full resolution (a `samplefactor` of 1), `N` will be 721-by-1441.

The gridded EGM96 data set must be on your path in a file named `'WW15MGH.GRD'`.

`[N, 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
```

Remarks

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:
`http://www.mathworks.com/support/tech-notes/2100/2101.html`

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`

Purpose

Local vertical elevation angle, range, and azimuth

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) as defined by `almanac`.

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 the vector `ellipsoid`, with form `[semimajor axis,eccentricity]`, to specify the ellipsoid. 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 earth ellipsoid defined by `azimuth` is used, and 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.

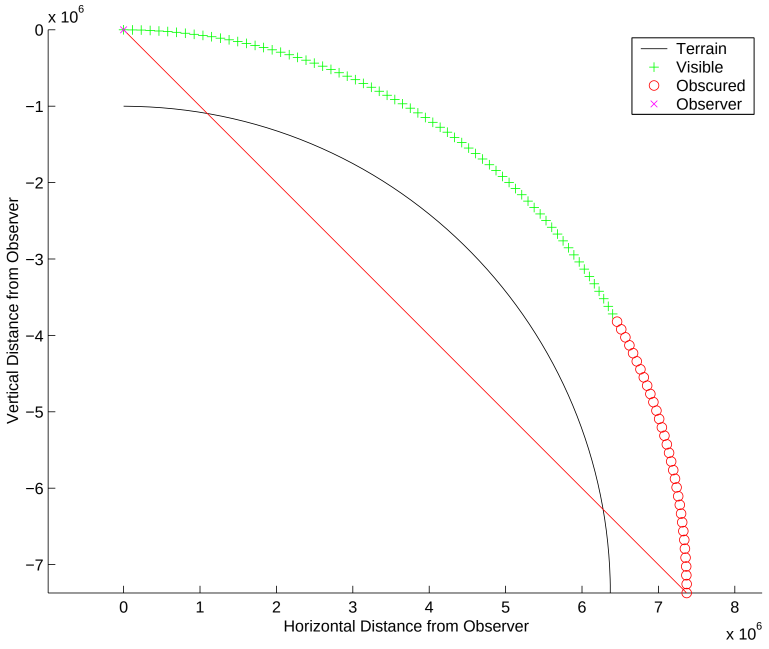
Examples

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

`almanac` | `azimuth` | `distance`

ellipse1

Purpose

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(lat0,lon0,ellipse,offset,units)

[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,

[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,
   units,npts)
[lat,lon] = ellipse1(track,...)
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` vector, of the form `[semimajor axis,eccentricity]`. If omitted, the unit sphere, `ellipsoid = [1 0]`, 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(lat0,lon0,ellipse,offset,`*`units`*`)`, `[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,`*`units`*), and `[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,`*`units`*) are all valid calling forms, which use the input *`units`* to define the angle units of the inputs and outputs. If the *`units`* string is omitted, `'degrees'` is assumed.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,`*`units`*`,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(`*`track`*`,...)` uses the *`track`* string to define either great circle or rhumb line distances from the ellipse center. If *`track`* = `'gc'`, then great circle distances are computed (the default). If *`track`* = `'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);
```

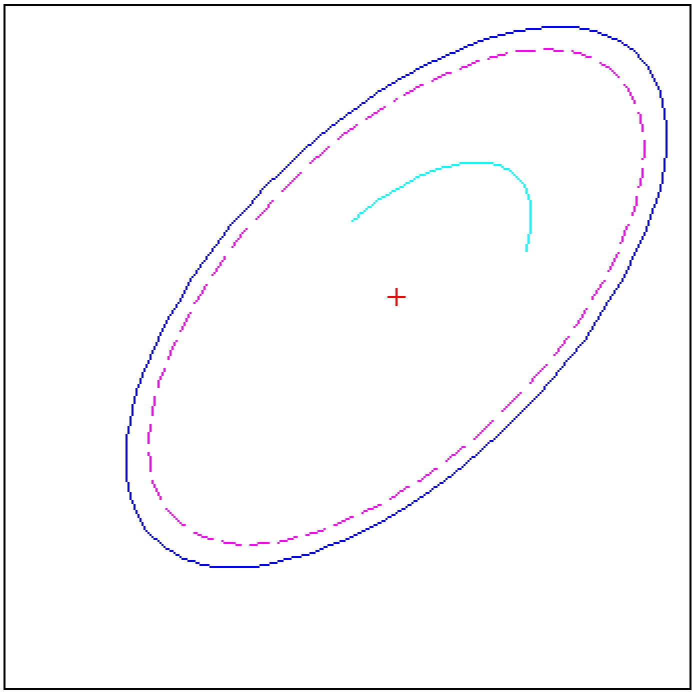
```
plotm(elat,elon)
```

If the desired radius is known in some nonangular distance unit, use the radius returned by the `almanac` function as the ellipsoid input to set the range units. (Use an empty azimuth entry to specify a full ellipse.)

```
earthradius = almanac('earth','radius','nm');
[elat,elon] = ellipse1(0,0,[550 ecc],45,[],earthradius);
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`

Purpose

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`

Purpose

Accuracy in angle units for certain map computations

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`

Purpose

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 two-element ellipsoid vector describing the ellipsoidal model of the figure of the Earth. The `ellipsoid` is spherical 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`

Purpose

Read global 5-min or 2-min digital terrain data

Syntax

```
[Z, refvec] = etopo
[Z, refvec] = etopo(samplefactor)
[Z, refvec] = etopo(samplefactor, latlim, lonlim)
[Z, refvec] = etopo(directory, ...)
[Z, refvec] = etopo(file, ...)
```

Description

`[Z, refvec] = etopo` reads the ETOPO data for the entire world from the ETOPO data in the current directory. The current directory 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] = etopo(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 for ETOPO5 data and 5400 by 10800 values for ETOPO2 data). When `samplefactor` is an integer n greater than one, every n^{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] = etopo(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. `lonlim` must be specified in the range `[0 360]` for ETOPO5 data and `[-180 180]` for ETOPO2 data. If `latlim` is empty the latitude limits are [-90 90]. If `lonlim` is empty, the longitude limits are determined by the file type.

`[Z, refvec] = etopo(directory, ...)` allows the path for the ETOPO data file to be specified by `directory` rather than the current directory.

`[Z, refvec] = etopo(file, ...)` reads the ETOPO data from `file`, where `file` is a string or a cell array of strings containing the name or names of the ETOPO data files.

Background

ETOPO5 is a global database of elevations and depths on a regular 5-minute grid. It is a compilation of data from a variety of different sources, including the U.S. Naval Oceanographic Office, U.S. Defense Mapping Agency, U.S. Navy Fleet Numerical Oceanographic Center, Bureau of Mineral Resources, Australia, and the Department of Industrial and Scientific Research, New Zealand. These databases were assembled by Margo Edwards at Washington University, St. Louis, Missouri.

Remarks

ETOPO5 data values are in whole meters, representing the elevation of the center of each cell. Some parts of the world are represented by data with a horizontal resolution as coarse as 1 degree by 1 degree. The vertical resolution varies from 1 meter for Australia and New Zealand to as much as 150 meters for parts of Africa, Asia, and South America. Oceanographic data in areas shallower than 200 meters contains little detail, because of how depth contours were converted to gridded depths.

ETOPO5 is superseded by ETOPO2 and the TerrainBase digital terrain model. See the `tbase` external interface function for more information.

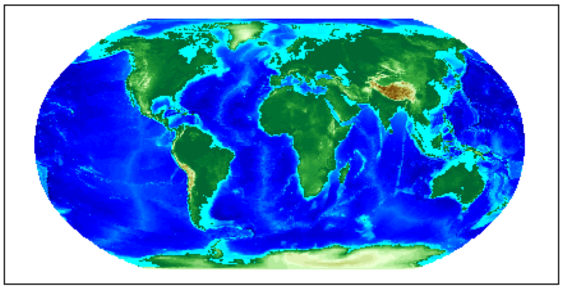
Note You can find links to more information about ETOPO files in the following page at the MathWorks Web site: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

Examples

Example 1

```
% Read and display the ETOPO5 data from the directory 'etopo5'
% downsampled by a factor of 10.
```

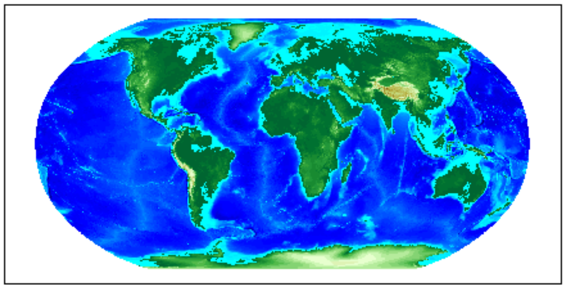
```
[Z, refvec] = etopo('etopo5',10);
whos

  Name         Size                     Bytes  Class
  Z          216x432                   746496  double array
  refvec       1x3                         24  double array

Grand total is 93315 elements using 746520 bytes

axesm robinson
geoshow(Z, refvec, 'DisplayType', 'surface');
colormap(demcmap(Z));
```



Example 2

```
% From the current directory, read and display the
% ETOPO2 binary data downsampled by a factor of 10.
cd etopo2
[Z, refvec] = etopo('ETOPO2.dos.bin', 10);
whos

  Name         Size                     Bytes  Class
  Z          540x1080                 4665600  double array
  refvec       1x3                         24  double array

figure; axesm robinson
geoshow(Z, refvec, 'DisplayType', 'surface');
```

```
colormap(demcmap(Z));
```



See Also gtopo30, tbase, usgsdem

Purpose

Read global 5-min digital terrain data

Syntax

Note `etopo5` is obsolete; use `etopo`.

```
[Z, refvec] = etopo5
[Z, refvec] = etopo5(samplefactor)
[[Z, refvec] = etopo5(samplefactor, latlim, lonlim)
[Z, refvec] = etopo5(directory, ...)
[Z, refvec] = etopo5(file, ...)
```

Description

`[Z, refvec] = etopo5` reads the topography data for the entire world for the data in the current directory. The current directory 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 nth 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(directory, ...)` allows the path for the data file to be specified by `directory` rather than the current directory.

`[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:
`http://www.mathworks.com/support/tech-notes/2100/2101.html`

Examples

Example 1

Read every tenth point in the data set:

```
% Read and display the ETOPO5 data from the directory 'etopo5'
% downsampled by a factor of 10.
[Z, refvec] = etopo5('etopo5',10);
axesm merc
geoshow(Z, refvec, 'DisplayType', 'surface');
colormap(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`

Purpose

Field values from structure array

Syntax

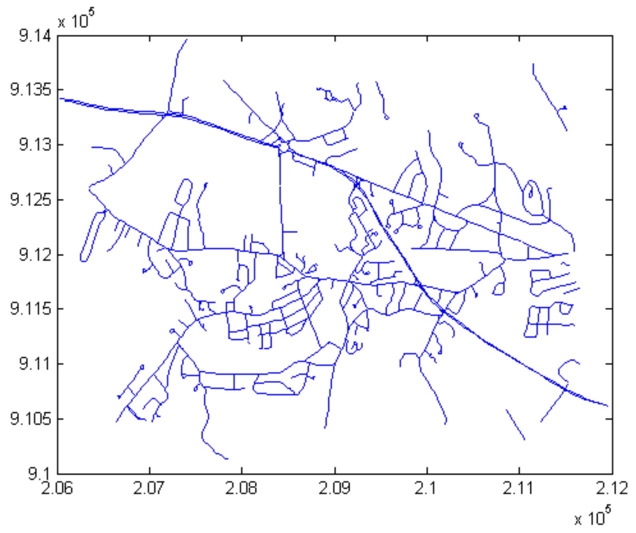
```
a = extractfield(s, name)
```

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`

Purpose

Coordinate data from line or patch display structure

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 (similar to the `strmatch` function)
`'findstr'`	Search within the tag (similar to the `findstr` function)
`'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.

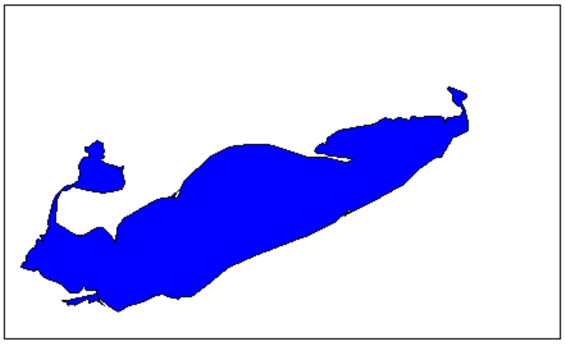
`[lat,lon,indx] = extractm(...)` also returns the vector `indx` identifying which elements of `display_struct` met the selection criteria.

`mat = extractm(...)` returns the vector data in a single matrix, where `mat = [lat lon]`.

Example

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')
```



Remarks

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 12-142 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`, `geoshow`, `mapshow`, `updategeostruct`, `mlayers`

Purpose

Project filled 3-D patch objects on map axes

Syntax

```
h = fill3m(lat,lon,z,cdata)
h = fill3m(lat,lon,z,PropertyName,PropertyValue,...)
```

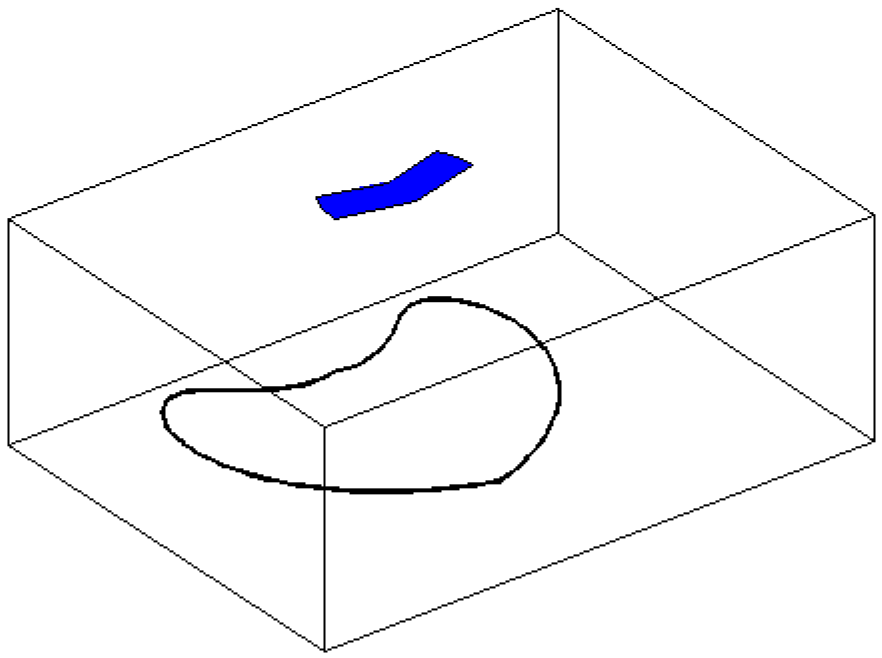
Description

`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

Purpose

Project filled 2-D patch objects on map axes

Syntax

```
h = fillm(lat,lon,cdata)
h = fillm(lat,lon,'PropertyName',PropertyValue,...)
```

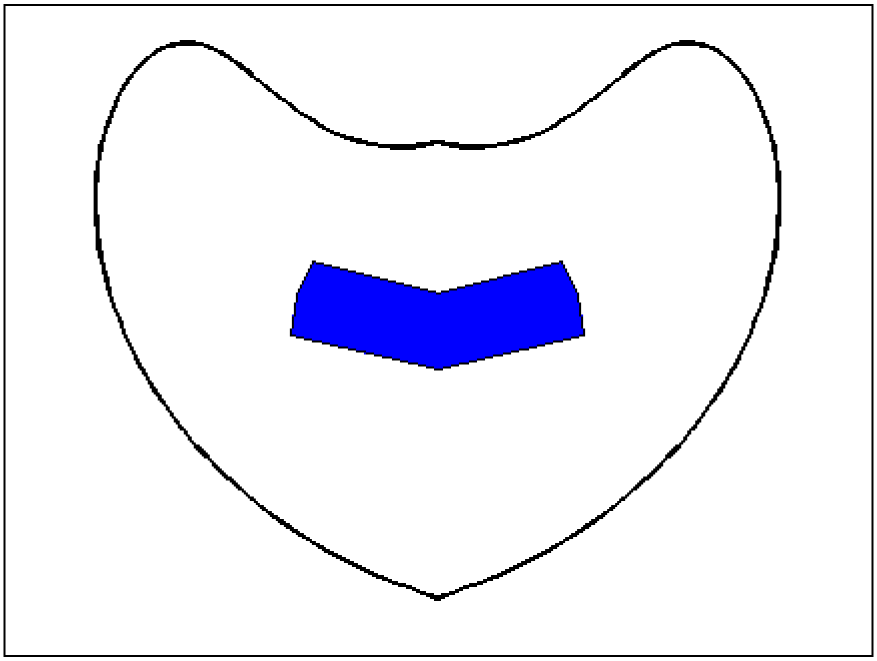
Description

`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

Purpose

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` 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.

`[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

Purpose

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` 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. All input and output 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.

Example

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` (MATLAB function)

fipsname

Purpose

Read Federal Information Processing Standard (FIPS) name file used with TIGER thinned boundary files

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`*.

Remarks

The FIPS name files, along with TIGER thinned boundary files, 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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

Example

```
struc = fipsname('st_name.dat')

struc =
1x57 struct array with fields:
    name
    id

s(1)

ans =
    name: 'Alabama'
      id: 1
```

Purpose

Eccentricity of ellipse with given flattening

Syntax

```
eccentricity = flat2ecc(flattening)
```

Description

`eccentricity = flat2ecc(flattening)` returns the equivalent eccentricity for the input `flattening`. If the input, `flattening`, is a two-column vector, only the second column is used. This allows two-element vectors to be used as rows of the input, since the form `[semimajor-axis, flattening]` is a complete representation of an ellipsoid (but is not the standard form for ellipsoid vectors in the toolbox). In all other cases, all columns of the input are used.

Flattening and eccentricity are two methods of defining an ellipsoid.

Example

```
e = flat2ecc(0.003353)

e =
   0.08182149712026
```

This eccentricity is the default value for the Earth.

See Also

`almanac`, `ecc2flat`, `ecc2n`, `majaxis`

flatearthpoly

Purpose

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`.

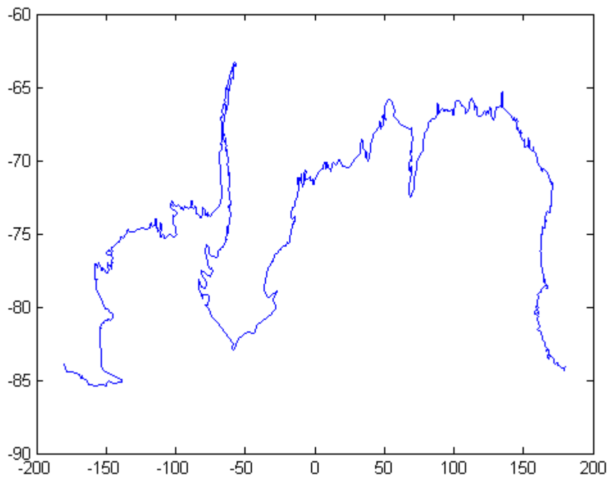
Remarks

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.

Example

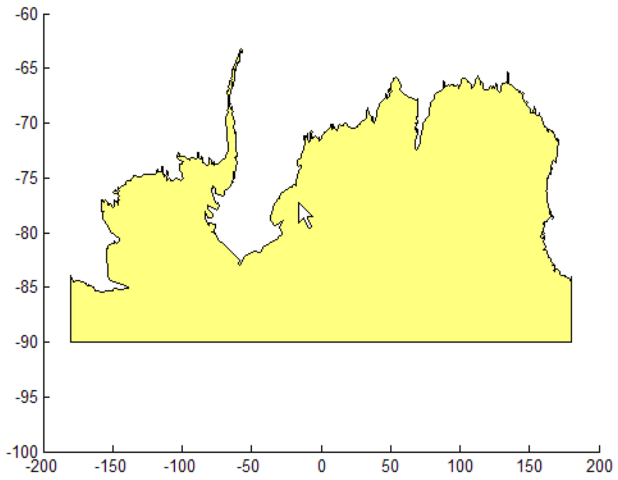
Vector data for geographic objects that encompass a pole will inevitably encounter or cross the date line. While the toolbox properly displays such polygons, they can cause problems for functions like the polygon intersection and Boolean operations that work with Cartesian coordinates. When these polygons 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])
```



See Also `ispolycw`, `maptrimp`, `poly2cw`, `poly2ccw`

Purpose

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.

Remarks

You can also create or alter map frame properties using the `axesm` or `setm` functions.

See Also

`axesm`, `setm`

Purpose

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

`degtorad`, `fromRadians`, `toDegrees`, `toRadians`

Purpose 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`, `radtodeg`, `toDegrees`, `toRadians`

Purpose

Center and radius of great circle

Syntax

```
[centerlat,centerlong,radius] = gc2sc(lat,long,az)
[centerlat,centerlong,radius] = gc2sc(lat,long,az,units)
mat = gc2sc(...)
```

Description

`[centerlat,centerlong,radius] = gc2sc(lat,long,az)` converts a great circle (i.e., latitude, longitude, azimuth, where latitude/longitude is on the circle) to a small circle (i.e., latitude, longitude, range, where latitude/longitude is the center of the circle, and range is 90º). A great circle has two possible centers (or zeniths). One is given; its antipode is the other.

`[centerlat,centerlong,radius] = gc2sc(lat,long,az,*units*)` uses the input *`units`* to define the angle units of the inputs and outputs. The default is `'degrees'`.

`mat = gc2sc(...)` returns a single output, where `mat = [lat long rng]`.

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:

```
[newlat,newlong,range] = gc2sc(-25,-70,45)

newlat =
  -39.8557
newlong =
   42.9098
range =
    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:

```
[newlat, newlong, range] = gc2sc(0,-70,270)
newlat =
    90
newlong =
 -145.9638
range =
    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`

Purpose

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
```

Remarks

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`

Purpose

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.

Remarks

`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]`.

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.

Example

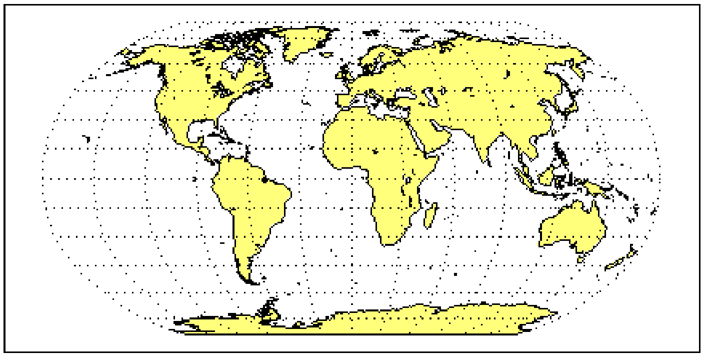
Set up a map axes with a graticule and display a world map:

```
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
```



See Also `inputm`, Axes Properties

Purpose

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

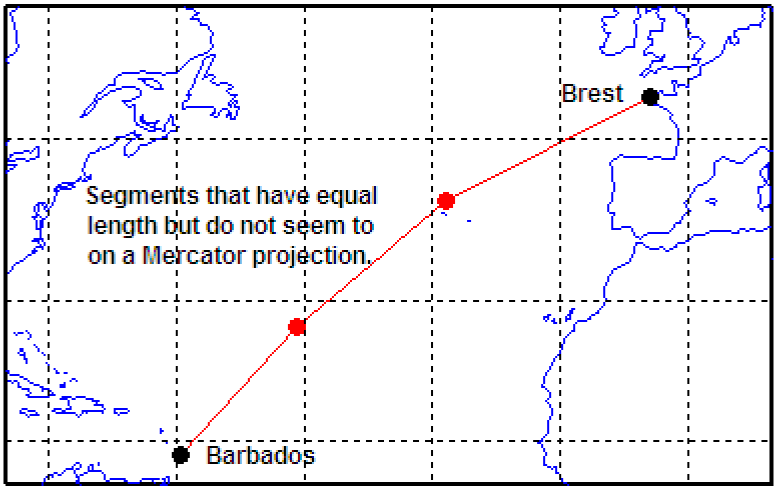
This is a navigational function. It assumes that all latitudes and longitudes are in degrees.

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')
```


Brest
Segments that have equal length but do not seem to on a Mercator projection.
Barbados

See Also

dreckon, legs, navfix, track

Purpose

Intersection points for pairs of great circles

Syntax

```
[newlat,newlong] = gcxgc(lat1,long1,az1,lat2,long2,az2)
[newlat,newlong] =
gcxgc(lat1,long1,az1,lat2,long2,az2,units)
```

Description

`[newlat,newlong] = gcxgc(lat1,long1,az1,lat2,long2,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,newlong] = gcxgc(lat1,long1,az1,lat2,long2,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,newlong] = gcxgc(10,13,10,0,20,-23)

newlat =
   14.3105  -14.3105
newlong =
   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,newlong] = gcxgc(10,13,0,0,20,180)

newlat =
   -90    90
newlong =
   -174.4504   12.5094
```

The two meridians intersect at the North and South Poles, which is exactly correct.

See Also

`antipode`, `gc2sc`, `scxsc`, `gcxsc`, `rhxrh`, `crossfix`, `polyxpoly`

Purpose

Intersection points for great and small circle pairs

Syntax

```
[newlat,newlong] = gcxsc(gclat,gclong,gcaz,sclat,sclong,
   scrange)
[newlat,newlong] = gcxsc(...,units)
```

Description

`[newlat,newlong] = gcxsc(gclat,gclong,gcaz,sclat,sclong,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,newlong] = 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,newlong] = gcxsc(43,0,10,47,3,12)

newlat =
   35.5068   58.9143
newlong =
   -1.6159    5.4039
```

See Also `gc2sc`, `gcxgc`, `scxsc`, `rhxrh`, `crossfix`, `polyxpoly`

Purpose

Convert geocentric to geodetic latitude

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`.

For conversion to/from other types of auxiliary latitude and, optionally, to work in degrees, use Mapping Toolbox function `convertlat`. For conversion from 3-D geocentric coordinates, see `ecef2geodetic`.

See Also

`convertlat`, `ecef2geodetic`, `geodetic2geocentricLat`

geodetic2ecef

Purpose

Convert geodetic to geocentric (ECEF) coordinates

Syntax

```
[x,y,z] = geodetic2ecef(phi,lambda,h,ellipsoid)
```

Description

`[x,y,z] = geodetic2ecef(phi,lambda,h,ellipsoid)` converts geodetic point locations specified by the coordinate arrays `phi` (geodetic latitude in radians), `lambda` (longitude in radians), and `h` (ellipsoidal height) to geocentric Cartesian coordinates `x`, `y`, and `z`. The geodetic coordinates refer to the reference ellipsoid specified by `ellipsoid` (a row vector with the form `[semimajor axis, eccentricity]`). `h` must use the same units as the semimajor axis; `x`, `y`, and `z` will be expressed in these units, also.

Definitions

The geocentric Cartesian coordinate system is fixed with respect to the Earth, with its origin at the center of the ellipsoid and its *x*-, *y*-, and *z*-axes intersecting the surface at the locations listed in the table below.

Axis	Latitude where axis intersects surface	Longitude where axis intersects surface	Description
x	0	0	Equator/Prime Meridian
y	0	90º E	Equator/90º E meridian
z	90º N	NA	North Pole

A common synonym is Earth-Centered, Earth-Fixed coordinates, or ECEF.

See Also

`ecef2geodetic` | `ecef2lv` | `geodetic2geocentricLat` | `lv2ecef`

geodetic2geocentricLat

Purpose

Convert geodetic to geocentric latitude

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`.

For conversion to/from other types of auxiliary latitude and, optionally, to work in degrees, use Mapping Toolbox function `convertlat`. For conversion to 3-D geocentric coordinates, see `geodetic2ecef`.

See Also

`convertlat`, `geocentric2geodeticLat`, `geodetic2ecef`

geoloc2grid

Purpose

Convert geolocated data array to regular data grid

Syntax

`[Z, refvec] = geoloc2grid(lat, lon, A, cellsize)`

Description

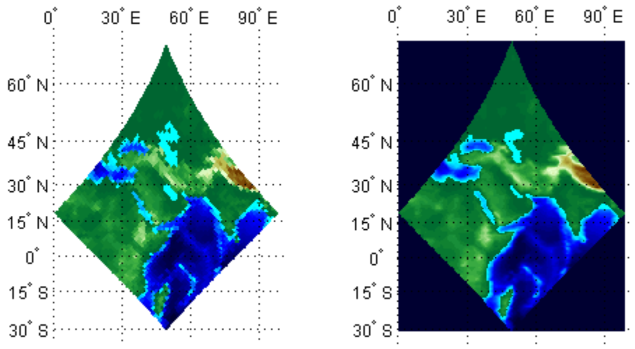
`[Z, refvec] = geoloc2grid(lat, lon, A, cellsize)` converts the geolocated data array `A`, given geolocation points in `lat` and `lon`, to produce a regular data grid, `Z`, and the corresponding three-element referencing vector `refvec`. `cellsize` is a scalar that specifies the width and height of data 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`.

Remarks

`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.

Example

```
% 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');
```



Purpose

Display map latitude and longitude data

Syntax

```
geoshow(lat,lon)
geoshow(lat,lon, ..., 'DisplayType', displaytype, ...)
geoshow(lat,lon,Z, ..., 'DisplayType', displaytype, ...)
geoshow(Z,R, ..., 'DisplayType', displaytype,...),
geoshow(lat,lon,I),geoshow(lat,lon,BW),geoshow(lat,lon,X,
   cmap), geoshow(lat,lon,RGB),
geoshow(...'DisplayType', ...)
geoshow(I,R),geoshow(BW,R),geoshow(RGB,R),geoshow(A,CMAP,R),
geoshow(... `DisplayType', ...)
geoshow(s)
geoshow(s, ..., `SymbolSpec', symspec)
geoshow(filename)
geoshow(ax, ...)
geoshow(..., 'Parent', ax, ...)
h = geoshow(...)
geoshow(..., param1, val1, param2, val2, ...)
```

Description

`geoshow(lat,lon)` or `geoshow(lat,lon, ..., 'DisplayType', displaytype, ...)` project and display the latitude and longitude vectors, `lat` and `lon`, using the projection stored in the axes. If there is no projection, the latitudes and longitudes are projected using a default Plate Carree projection. `lat` and `lon` must be of equal lentgth, and may contain embedded NaNs, delimiting individual lines or polygon parts. `DisplayType` can be `'point'`, `'line'`, or `'polygon'`, and defaults to `'line'`.

`geoshow(lat,lon,Z, ..., 'DisplayType', displaytype, ...)`, projects and displays a geolocated data grid. `lat` and `lon` are M-by-N latitude-longitude arrays and Z is an M-by-N array of class `double`. `lat`, `lon`, and `Z` may contain NaN values. `DisplayType` must be set to `'surface'`, `'mesh'`, `'texturemap'`, or `'contour'`.

`geoshow(Z,R, ..., 'DisplayType', *displaytype*,...)`, projects and displays a regular data grid. `Z` is a 2-D array of class `double`. `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 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. For more information about referencing vectors and matrices, see the section "Understanding Raster Geodata" on page 2-33 in the User's Guide.

`DisplayType` must be set to `'surface'`, `'mesh'`, `'texturemap'`, or `'contour'`. If `DisplayType` is `'texturemap'`, `geoshow` constructs a surface with `ZData` values set to `0`.

`geoshow(lat,lon,I)`,`geoshow(lat,lon,BW)`,`geoshow(lat,lon,X,cmap)`, `geoshow(lat,lon,RGB)`, or `geoshow(...'DisplayType', ...)` projects and display a geolocated image as a texturemap on a zero-elevation surface. lat and lon are latitude-longitude geolocation arrays and I is a grayscale image, BW is a logical image, X is an indexed image with colormap cmap, or RGB is a truecolor image. lat, lon, and the image array must match in size. If specified, `DisplayType` must be set to `'image'`. Examples of geolocated images include a color composite from a satellite swath or an image originally referenced to a different coordinate system.

`geoshow(I,R)`,`geoshow(BW,R)`,`geoshow(RGB,R)`,`geoshow(A,CMAP,R)`, or `geoshow(...  `DisplayType', ...)` project and display an image georeferenced to latitude-longitude through the referencing matrix `R`. The image is shown as a texturemap on a zero-elevation surface. If specified, `DisplayType` must be set to `'image'`.

`geoshow(s)` or `geoshow(s, ..., `SymbolSpec', symspec)` display 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 `Lat` and `Lon` fields, then the coordinate values are projected to map coordinates. If `s` includes `X` and `Y` fields they are plotted as (preprojected) map coordinates and a warning is issued. symspec is a structure returned by `makesymbolspec` that specifies the symbolization rules to be used for displaying vector data.

`geoshow(filename)` projects and displays data from `filename` according to the type of file format. The `DisplayType` parameter is automatically set, according to the following table:

Format	DisplayType
Shape file	`'point'`, `'line'`, or `'polygon'`
GeoTIFF	`'image'`
TIFF/JPEG/PNG with a world file	`'image'`
ARC ASCII GRID	`'surface'` (can be overridden)
SDTS raster	`'surface'` (can be overridden)

`geoshow(ax, ...)` and `geoshow(..., 'Parent', ax, ...)` set the parent axes to `ax`.

`h = geoshow(...)` returns a 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.

`geoshow(..., param1, val1, param2, val2, ...)` specifies parameter/value pairs that modify the type of display or set MATLAB graphics properties. Refer to the MATLAB Graphics documentation for line, patch, image, surface, mesh, and contour properties for full descriptions of these object properties and their values.

Parameters

Parameter names can be abbreviated and are case insensitive. Parameters include

- DisplayType: The DisplayType parameter specifies the type of graphic display for the data. The value must be consistent with the type of data being displayed, as shown in the following table:

Data Type	Value(s)
Vector	'point', 'multipoint', 'line', or 'polygon'
Image	'image'
Grid	'surface', 'mesh', 'texturemap', or 'contour'

- SymbolSpec: The SymbolSpec parameter specifies the symbolization rules used for vector data through a structure returned by makesymbolspec. 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 property name/property value pair in the symbolspec, prefix the word 'Default' to the graphics property name (listed in the preceding table). See Example 2 below.

Note If you display a polygon, do not set 'EdgeColor' to either 'flat' or 'interp'. This combination may result in a warning.

Graphics Properties

In addition to specifying a parent axes, you can set any appropriate property for a point, line, and polygon DisplayType, as follows:

DisplayType	Properties
'line'	Any MATLAB line property

DisplayType	Properties
`'point'`	Any MATLAB line marker property
`'polygon'`	Any MATLAB patch property

See the MATLAB Graphics Reference documentation for line, patch, image, and surface properties for complete descriptions of these properties and their values.

Remarks

`geoshow` is often used to display vector geodata previously read from shapefiles using `shaperead`. 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 (or specify `'UseGeoCoords',false`), `shaperead` will create a mapstruct, with coordinate fields labelled `X` and `Y` instead of `Lon` and `Lat`, causing `geoshow` to assume that the geostruct is in fact a mapstruct containing projected coordinates. In such cases, `geoshow` warns and calls `mapshow` to display the geostruct data without projecting it.

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.

Note 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" on page 4-40.

`geoshow` adds graphics to the current map axes (it does not clear it first), enabling you to create multiple raster and vector map layers. If you do not want `geoshow` to draw on top of an existing map, create a new figure or subplot before calling it.

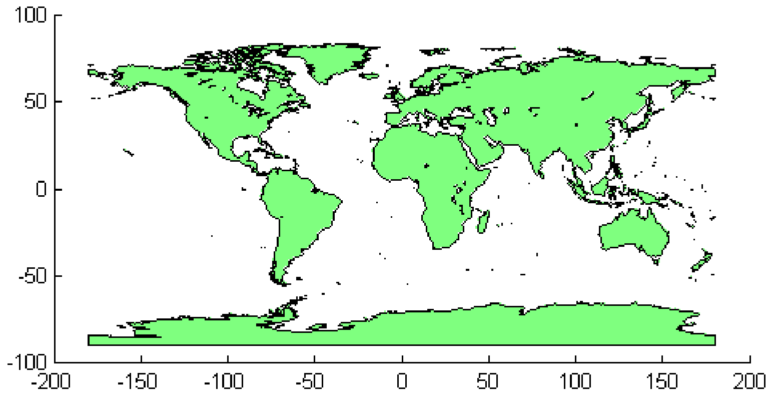
`geoshow` can generally be substituted for `displaym`. However, there are limitations where display of specific objects is concerned. See the remarks under `updategeostruct` for further information.

Examples

Example 1

Display world land areas using a default Plate Carree projection:

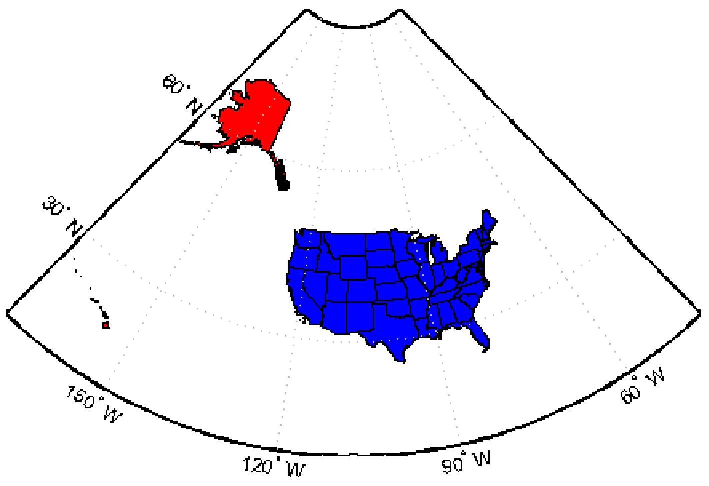
```
figure
geoshow('landareas.shp', 'FaceColor', [0.5 1.0 0.5]);
```



Example 2

Override the symbolspec default rule:

```
% Create a worldmap of North America
figure
worldmap('na');

% Read the USA high resolution data
states = shaperead('usastatehi', 'UseGeoCoords', true);

% Create a symbolspec to make 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');
```

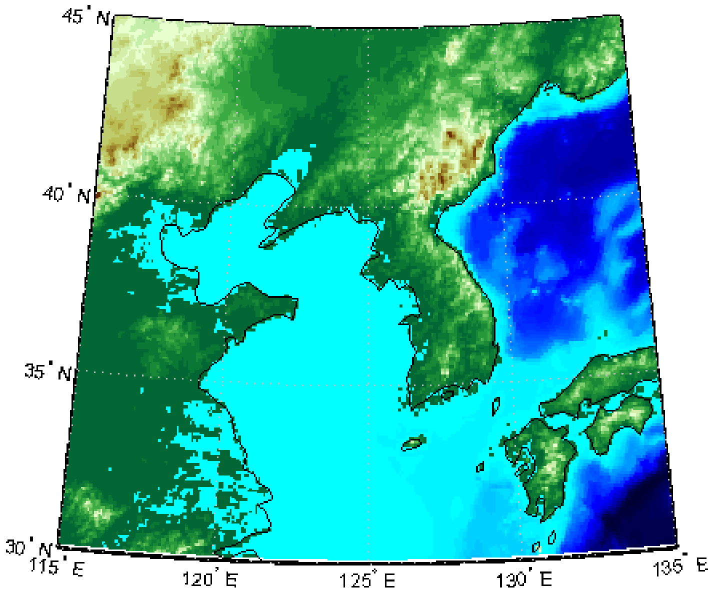


Example 3

Create a `worldmap` of Korea and display the `korea` data grid as a texture map:

```
load korea
figure;
worldmap(map, refvec)

% Display the Korean data grid as a texture map.
geoshow(gca,map,refvec,'DisplayType','texturemap');
colormap(demcmap(map))

% Display the land area boundary as black lines.
```

```
S = shaperead('landareas','UseGeoCoords',true);
geoshow([S.Lat], [S.Lon],'Color','black');
```

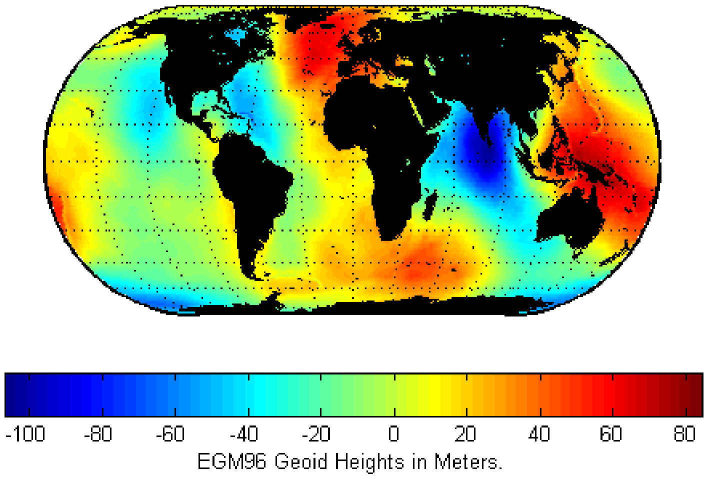


Example 4

Display the EGM96 geoid heights, masking out land areas:

```
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('horiz');
set(get(hcb,'Xlabel'),'String','EGM96 Geoid Heights in Meters.')

% Mask out all the land.
geoshow('landareas.shp', 'FaceColor', 'black');
```

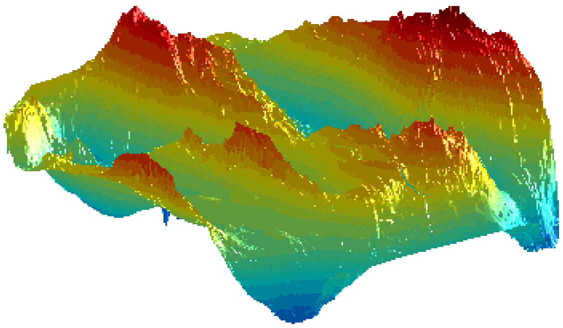


Example 5

Display the EGM96 geoid heights as a 3-D surface using the Eckert IV projection:

```
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.
```

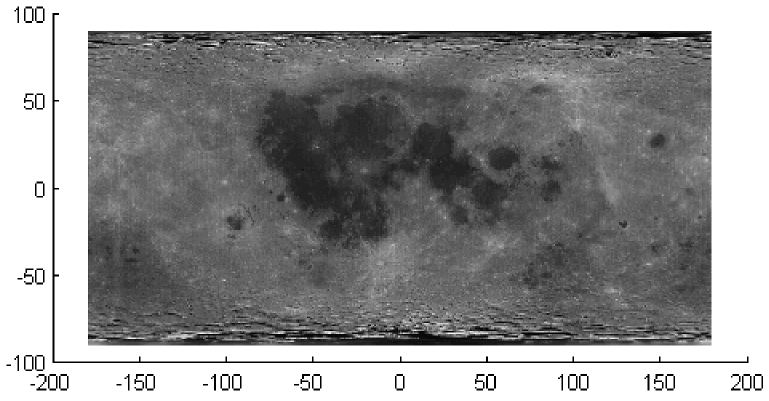
```
light; material(0.6*[ 1 1 1]);

% View as a 3-D surface.
view(3)
axis normal
tightmap
```



Example 6

Display the moon albedo image projected using Plate Carree and in an orthographic projection.

```
load moonalb

% Projection not specified -- uses Plate Carree
figure
geoshow(moonalb,moonalbrefvec)
```



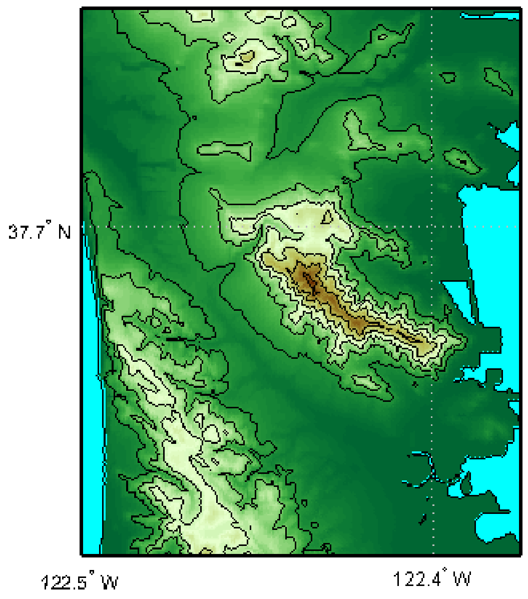
```
% Orthographic projection
figure
axesm ortho
geoshow(moonalb, moonalbrefvec, 'DisplayType', 'texturemap')
colormap(gray(256))
axis off
```

Example 7

Read and display the San Francisco South 24K DEM data:

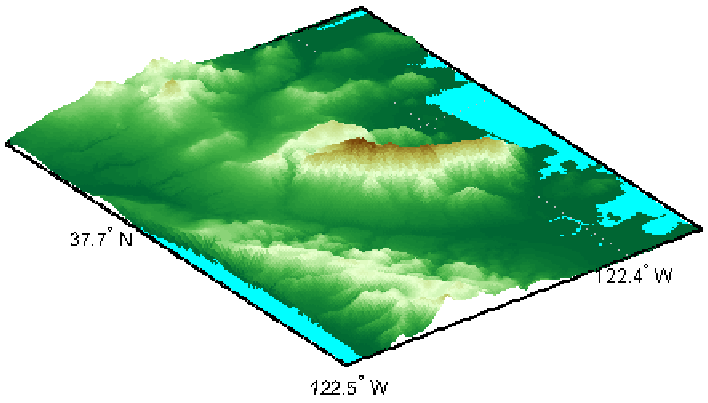
```
filenames = gunzip('sanfranciscos.dem.gz', tempdir);
demFilename = filenames{1};

% Read every point of the 1:24,000 DEM file.
[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)
```



See Also

`axesm`, `makesymbolspec`, `mapshow`, `mapview`, `updategeostruct`

Purpose

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.

Example

```
% 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`

Purpose

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.

`filename` is a string that specifies the name of the GeoTIFF file. `filename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path. If the named file includes the extension `.TIF` or `.TIFF` (either upper- or lowercase), the extension can be omitted from `filename`.

If `filename` is a file containing more than one GeoTIFF image, `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 more than one image exists in the file, it is assumed that each image will have the same cartographic information and the same image width and height.

`info = geotiffinfo(url)` reads the GeoTIFF image from an Internet URL. The `url` must include the protocol type (e.g., “`http://`”).

Field Description

The `info` structure contains the following fields:

`Filename`	String containing the name of the file
`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'`, or `''`
`PCS`	String describing 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`	An 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`	An N-by-1 cell array listing the projection parameter identifier for each corresponding numerical element of `ProjParm`: • `'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` — A structure containing row and column coordinates of each image tiepoint. The `ImagePoints` structure contains these fields: - `Row` — A double array of size 1-by-N. - `Col` — A double array of size 1-by-N. • `WorldPoints` — A structure containing the *x* and *y* world coordinates of the tiepoints. The `WorldPoints` structure contains these fields: - `X` — A double array of size 1-by-N - `Y` — A double array of size 1-by-N
`PixelScale`	3-by-1 double array that specifies the X, Y, Z pixel scale values

`RefMatrix`	3-by-2 double referencing matrix that must be unambiguously defined by the GeoTIFF file; otherwise it is returned 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`	A 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:

- X — Horizontal coordinates in the projected coordinate system. `X` equals `Lon` (below) if *`ModelType`* is `'ModelTypeGeographic'`.
- Y — Vertical 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. The `GeoTIFFCodes` fields are: • `Model` • `PCS` • `GCS` • `UOMLength` • `UOMAngle` • `Datum` • `PM` • `Ellipsoid` • `ProjCode` • `Projection` • `CTProjection` • `ProjParmId` • `MapSys` Each is scalar, except for `ProjParmId`, which is a column vector.
`ImageDescription`	String describing the image; if no description is included in the file, the field is omitted.

Example

```
info = geotiffinfo('boston.tif')

info =
                Filename: [1x78 char]
```

```
          FileModDate: '31-May-2007 03:25:30'
             FileSize: 38729900
               Format: 'tif'
        FormatVersion: []
               Height: 2881
                Width: 4481
             BitDepth: 24
            ColorType: 'truecolor'
            ModelType: 'ModelTypeProjected'
                  PCS: 'NAD83 / Massachusetts Mainland'
           Projection: 'SPCS83 Massachusetts Mainland zone (m)'
               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+006
                   PM: 'Greenwich'
    PMLongToGreenwich: 0
            UOMLength: 'US survey foot'
    UOMLengthInMeters: 0.3048
             UOMAngle: 'degree'
    UOMAngleInDegrees: 1.0000
            TiePoints: [1x1 struct]
           PixelScale: [3x1 double]
            RefMatrix: [3x2 double]
          BoundingBox: [2x2 double]
         CornerCoords: [1x1 struct]
          GeoTIFFCodes: [1x1 struct]
     ImageDescription: '"GeoEye"'
```

See Also

imfinfo, geotiffread, makerefmat, projfwd, projinv, projlist

Purpose

Read georeferenced image from GeoTIFF file

Syntax

```
A = geotiffread(filename)
[X, cmap] = geotiffread(filename)
[X, cmap, R, bbox] = geotiffread(filename)
[A, R, bbox] = geotiffread(filename)
[...] = geotiffread(filename, idx)
[...] = geotiffread(url, ...)
```

Description

`A = geotiffread(filename)` reads the GeoTIFF image in `filename` into `A`. If the file contains a grayscale image, `A` is a two-dimensional array. If the file contains a true-color (RGB) image, `A` is a three-dimensional (M-by-N-by-3) array.

`filename` is a string that specifies the name of the GeoTIFF file. `filename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path. If the named file includes the extension `.TIF` or `.TIFF` (either upper- or lowercase), the extension can be omitted from `filename`.

`[X, cmap] = geotiffread(filename)` reads the indexed image in `filename` into `X` and its associated colormap into `cmap`. Colormap values in the image file are automatically rescaled into the range `[0,1]`.

`[X, cmap, R, bbox] = geotiffread(filename)` reads the indexed image into `X`, the associated colormap into `cmap`, the referencing matrix into `R`, 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 ([]).

`[A, R, bbox] = geotiffread(filename)` reads the grayscale or RGB image into `A`, the referencing matrix into `R`, and the bounding box into `bbox`.

`[...]  = geotiffread(filename, idx)` reads in one image from a multiimage GeoTIFF file. `idx` is an 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. If you omit this argument, `geotiffread` reads the first image in the file.

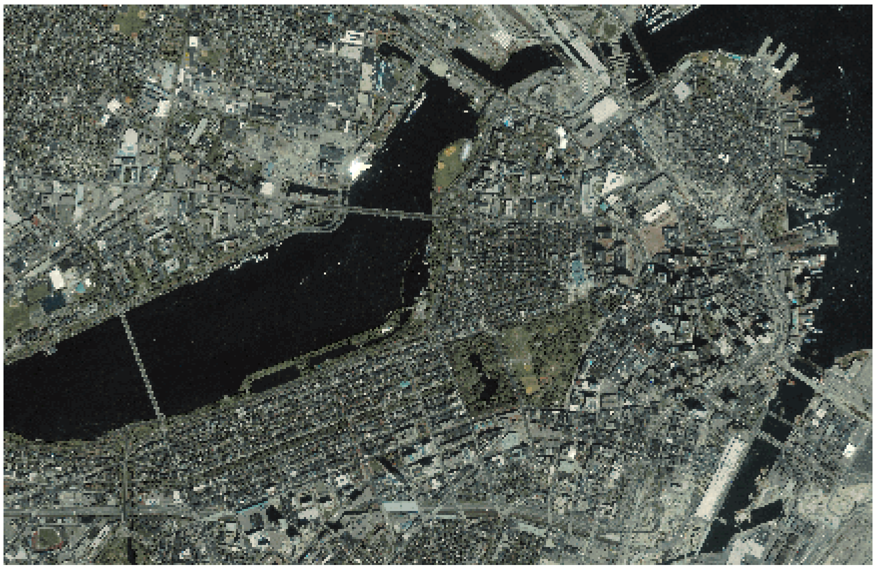
`[...] = geotiffread(url, ...)` reads the GeoTIFF image from an Internet URL. The URL must include the protocol type (e.g., "`http://`").

Note `geotiffread` imports pixel data using the TIFF-reading capabilities of the MATLAB function `imread`. Consequently, it shares the limitations of `imread`. Consult the `imread` documentation for details on the types of TIFF images that `imread` can import.

Example

Read and display the Boston GeoTIFF image:

```
[boston, R, bbox] = geotiffread('boston.tif');
figure
mapshow(boston, R);
axis image off
```



boston.tif copyright © GeoEye, all rights reserved.

See Also

`geotiffinfo`, `imread`, `mapview`, `mapshow`, `geoshow`

getm

Purpose

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`

Purpose

Interactively assign seeds for data grid encoding

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`

Purpose

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

Purpose

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(dirname,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 subdirectory `'/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(dirname,scalefactor,latlim,lonlim)` reads and concatenates data from multiple files within a GLOBE directory tree. The `dirname` input is a string with the name of the directory 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.

Remarks

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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

Examples

Determine the file that contains the area around Cape Cod.

```
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. Provide an empty file name, and select the 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 directory 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
```

References

See Web site for the National Oceanic and Atmospheric Administration, National Geophysical Data Center

See Also

demdataui, dted, gtopo30, satbath, tbase, usgsdem

Purpose

GLOBE data filenames for latitude-longitude quadrangle

Syntax

```
fname = globedems(latlim,lonlim)
```

Description

`fname = globedems(latlim,lonlim)` returns a cell array of the filenames covering the geographic region for GLOBE DEM digital elevation maps. The region is specified by scalar latitude and longitude points, or two-element vectors of latitude and longitude limits 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 files 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 filenames for a given geographic region.

Remarks

The `globedems` function reads data from GLOBE Version 1.0. GLOBE DEM first reads the corresponding ESRI header file found in the subdirectory `'/esri/hdr/'` and then the binary data file (with no extension).

Examples

Which files are needed for southern Louisiana?

```
latlim =[28.61 31.31]; lonlim = [-91.24 -88.62];
globedems(latlim,lonlim)

ans =
    'e10g'
    'f10g'
```

References

See Web site for the National Oceanic and Atmospheric Administration, National Geophysical Data Center

See Also

`globedem`

Purpose

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` 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 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.

`[...]  = 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 specified ellipsoid vector, `ellipsoid`, a 1-by-2 vector of the form [semimajor-axis, eccentricity]. If the map contains elevations in the same units as `ellipsoid(1)`, 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 `ellipsoid(1)`.

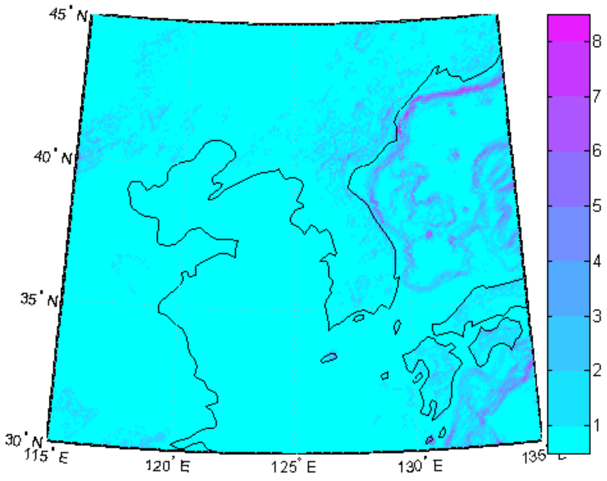
Remarks

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.

Example

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')
```



See Also viewshed

Purpose

Identify matching fields in fixed record length files

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.

Example

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.

Remarks

See `readfields` for a complete discussion of the format and contents of the `fields` argument.

See Also

`readfields`, `fopen`

gridm

Purpose

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 12-45 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.

Remarks

You can also create or alter map grid properties using the `axesm` or `setm` functions.

See Also

`axesm`, `setm`

Purpose

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 as an image. `Z` can be a matrix of dimension M-by-N or M-by-N-by-3, and can contain double, `uint8`, or `uint16` data. `R` is a 1-by-3 referencing vector defined as [cells/angle units north-latitude west-longitude], or a 3-by-2 referencing matrix, defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. The displayed map is a Plate Carrée projection, treating longitude as X and latitude as Y. This projection produces significant distortion near the poles.

`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.

See Also

`image`, `mapshow`, `mapview`, `meshm`, `surfacem`, `surfm`

grn2eqa

Purpose 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 (0°,0°).

`[x,y] = grn2eqa(lat,lon,origin,ellipsoid)` specifies the two-element ellipsoid vector describing the ellipsoidal model of the figure of the Earth. 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`

Purpose

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 version 1.3 and earlier vector data for the entire world from `filename`. GSHHS files have names of the form `gshhs_X.b`, where `X` is one of the letters `c`, `l`, `i`, `h` and `f`, corresponding to increasing resolution (and file size). The result returned in `S` is a polygon geographic data structure array (geostruct).

`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 directory 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 geostruct output structure `S` contains the following fields; all latitude and longitude values are in degrees:

Field Name	Field Contents
`Geometry`	`'Polygon'`

(Continued)

Field Name	Field Contents
`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'` or `'lake'`, or `'island_in_lake'`, or `'pond_in_island_in_lake'` or `'other'`
`NumPoints`	Number of points in the polygon
`FormatVersion`	Format version of data: empty if unspecified
`Source`	Source of data: `'WDBII'` or `'WVS'`
`CrossGreenwich`	Scalar flag: true if the polygon crosses the prime meridian, false otherwise
`GSHHS_ID`	Unique polygon scalar id number, starting at 0

Remarks

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 when mapping very large amounts of data, you might want to plot the data as `NaN`-clipped lines rather than as patches.

Note that 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.

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 1.3 of the `gshhs_c.b` (coarse) data set ships with the toolbox in the `toolbox/map/mapdemos` directory. For details, type

```
type gshhs_c.txt
```

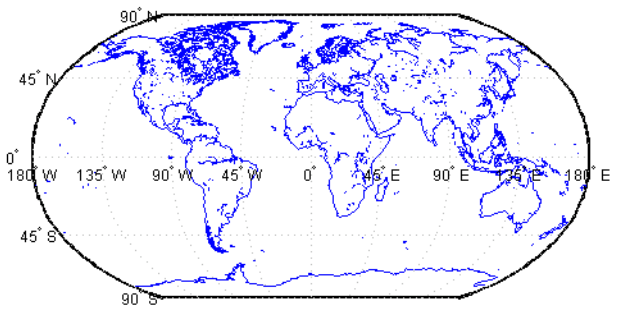
at the MATLAB command prompt.

Examples

Example 1

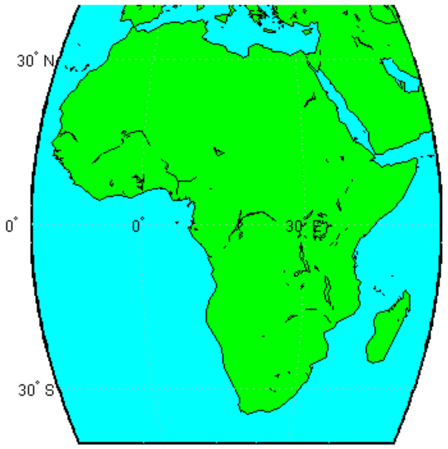
Read the entire coarse data set (located on the MATLAB path in *`matlabroot`*`/toolbox/map/mapdemos`) and display as a coastline:

```
filename = gunzip('gshhs_c.b.gz', tempdir);
world = gshhs(filename{1});
delete(filename{1})
figure
worldmap world
geoshow([world.Lat], [world.Lon])
```



After creating an index file, read and display Africa as a green polygon; note that `gshhs` detects and uses the index file automatically:

```
filename = gunzip('gshhs_c.b.gz', tempdir);
indexname = gshhs(filename{1}, 'createindex');
figure
worldmap Africa
projection = gcm;
latlim = projection.maplatlimit;
lonlim = projection.maplonlimit;
africa = gshhs(filename{1}, latlim, lonlim);
delete(filename{1})
delete(indexname)
geoshow(africa, 'FaceColor', 'green')
setm(gca, 'FFaceColor', 'cyan')
```



Note 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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

Example 2

Read all of the lowest resolution database:

```
s = gshhs('gshhs_c.b')
```

Example 3

Read the intermediate resolution database for South America:

```
s = gshhs('gshhs_i.b',[-60 -15],[-90 -30])
```

Example 4

Read the full-resolution file for East and West Falkland Islands (Islas Malvinas):

```
s = gshhs('gshhs_f.b',[-55 -50],[-65 -55])
```

Example 5

Create the index file for the high-resolution database:

```
gshhs('gshhs_h.b','createindex')
```

See Also

`dcwdata`, `geoshow`, `maptrimp`, `shaperead`, `vmap0data`, `worldmap`

gtextm

Purpose 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.

Example Create map axes:

```
axesm('sinusoid','FEdgeColor','red')
gtextm('hello world','FontWeight','bold')
```

Click inside the frame and the text appears.

See Also `axesm`, `textm`

Purpose

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(dirname,samplefactor,latlim,lonlim)
```

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 directory 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`. 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 `[-180 180]`. 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(dirname,samplefactor,latlim,lonlim)` reads and concatenates data from multiple tiles within a GTOPO30 CD-ROM or directory structure. The `dirname` input is a string with the name of the directory which contains the GTOPO30 tile directories or GTOPO30 tiles. Within the tile directories are the uncompressed data files. The dirname for CD-ROMs distributed by the USGS is the device name of the CD-ROM drive. As in 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 dirname.

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].

When directory dirname contains no GTOPO30 data or tilename identifies a file with a .DEM extension that is not a GTOPO30 file, the function returns a single NaN in Z, a canonical refvec, and issues a warning.

The data and documentation are available over the Internet via http and anonymous ftp, as well as for purchase on CD-ROM.

Note For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: http://www.mathworks.com/support/tech-notes/2100/2101.html.

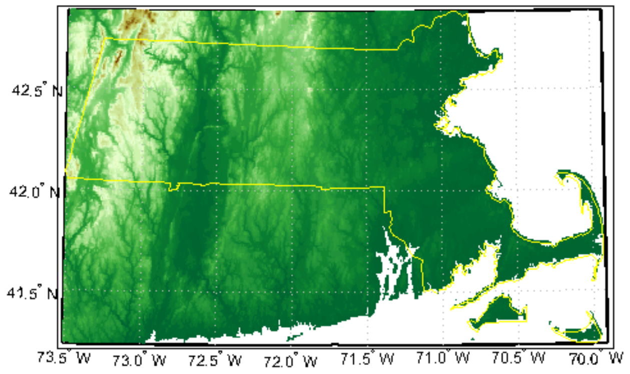
Examples

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')
colormap(demcmap(Z))
geoshow(Massachusetts,'DisplayType','polygon',...
  'facecolor','none','edgecolor','y')
```



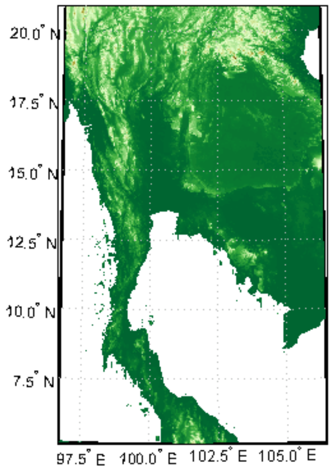
Example 2

```
% 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 directory or mapped drive if not "F:\'
[Z,refvec] = gtopo30('F:\',5,latlim,lonlim);
worldmap(Z,refvec);
```

```
geoshow(Z,refvec,'DisplayType','surface')
colormap(demcmap(Z))
```



Example 4

```
% Extract every 10th point from a column of data 5 degrees around
% the prime meridian. The current directory contains GTOPO30 data.
[Z,refvec] = gtopo30(pwd,10,[],[-5 5]);
```

See Also

gtopo30s, globedem, dted, satbath, tbase, usgsdem

Purpose

GTOPO30 data filenames for latitude-longitude quadrangle

Syntax

`fname = gtopo30s(latlim,lonlim)`

Description

`fname = gtopo30s(latlim,lonlim)` returns a cell array of the filenames covering the geographic region for GTOPO30 digital elevation maps (also referred to as “30-arc second” DEMs). `latlim` and `lonlim` specify the region as scalar latitude and longitude points, or two-element vectors of latitude and longitude limits in units of degrees.

Remarks

The data and documentation are available over the Internet via http and anonymous ftp.

Note For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

See Also

`gtopo30`

handlem

Purpose

Handles of displayed map objects

Syntax

```
handlem or handlem('taglist')
h = handlem('prompt')
h = handlem(object)
handlem('object',axesh)
handlem('object',axesh,'searchmethod')
h = handlem(handles)
```

Description

`handlem` or `handlem('taglist')` displays a dialog box for selecting the objects for which you want handles.

`h = handlem('prompt')` displays another dialog box, which allows greater control of object selection.

`h = handlem(`*`object`*`)` returns the handles of those objects specified by the input string. The options for the *`object`* string are

`'all'`	All children of the current axes
`'clabels'`	Contour labels on the current map axes
`'contour'`	Contourgroup for contours on the current map axes
`'contour3d'`	3-D contour lines on the current map axes
`'cpatches'`	Filled contour patches on the current map axes
`'frame'`	Map frame
`'grid'`	Map grid lines
`'hggroup'`	All hggroup objects
`'hidden'`	Hidden objects on the current axes
`'image'`	Image objects on the current axes
`'light'`	Light objects on the current axes
`'line'`	Line objects on the current axes

`'map'`	All objects on the map, excluding the frame (default)
`'meridian'`	Longitude grid lines
`'mlabel'`	Longitude labels
`'parallel'`	Latitude grid lines
`'patch'`	Patch objects on the current axes
`'plabel'`	Latitude labels
`'scaleruler'`	Scaleruler objects
`'surface'`	Surface objects on the current axes
’`'text'`’	Text objects on the current axes
`'tissot'`	Tissot indicatrices on the current map axes
`'visible'`	Visible objects on the current axes

Or any user-defined object tag string.

A prefix of `'all'` can be applied to strings defining a Handle Graphics object type (`'allimage'`, `'allline'`, `'allsurface'`, `'allpatch'`, `'alltext'`) to determine all object handles that meet the type criteria. Without the `'all'` prefix, those objects named by the user with the `tagm` function are not included (e.g., a line with the tag `'route'` would not be included for object string `'line'`, but would be for `'allline'`).

`handlem('`*`object`*`',axesh)` searches within the axes specified by the input handle `axesh`.

`handlem('`*`object`*`',axesh,'`*`searchmethod`*`')` controls the method used to match the `'str'` input. If omitted, `'exact'` is assumed. Search method `'strmatch'` searches for matches at the beginning of the tag, similar to the MATLAB `strmatch` function. Search method `'findstr'` searches within the tag, similar to the MATLAB `findstr` function.

`h = handlem(handles)` returns those elements of an input vector of handles that are still valid.

See Also `clma`, `clmo`, `hidem`, `namem`, `showm`, `tagm`

Purpose

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

Purpose

Histogram for geographic points with equal-area bins

Syntax

```
[lat,lon,num] = hista(lats,lons)
[lat,lon,num] = hista(lats,lons,binarea)
[lat,lon,num] = hista(lats,lons,binarea,ellipsoid)
[lat,lon,num] = hista(lats,lons,binarea,units)
```

Description

`[lat,lon,num] = hista(lats,lons)` returns the center coordinates of equal-area bins and the number of observations falling in each based on the geographically distributed input data.

`[lat,lon,num] = hista(lats,lons,binarea)` specifies the equal-area bin size, in square kilometers. It is 100 km^2 by default.

`[lat,lon,num] = hista(lats,lons,binarea,ellipsoid)` specifies the elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. The default ellipsoid model is a spherical Earth, which is sufficient for most applications.

`[lat,lon,num] = hista(lats,lons,binarea,`*`units`*`)` specifies the standard angle unit string. The default value is `'degrees'`.

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 50-by-50 km cells (2500 sq km):

```
[lat,lon,num] = hista(lats,longs,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

Purpose

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

Purpose

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` 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. The entries to be reset correspond to the locations defined by the latitude and longitude position vectors `lat` and `lon`. The entries are reset to the same number if `value` is a scalar, or to individually specified numbers if `value` is a vector the same size as `lat` and `lon`. If any points lie outside the input grid, a warning is issued. All input angles are in 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

Purpose

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.

Example

```
% 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`

Purpose

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.

Example

1. Load and display a digital elevation model (DEM) including the Korean Peninsula:

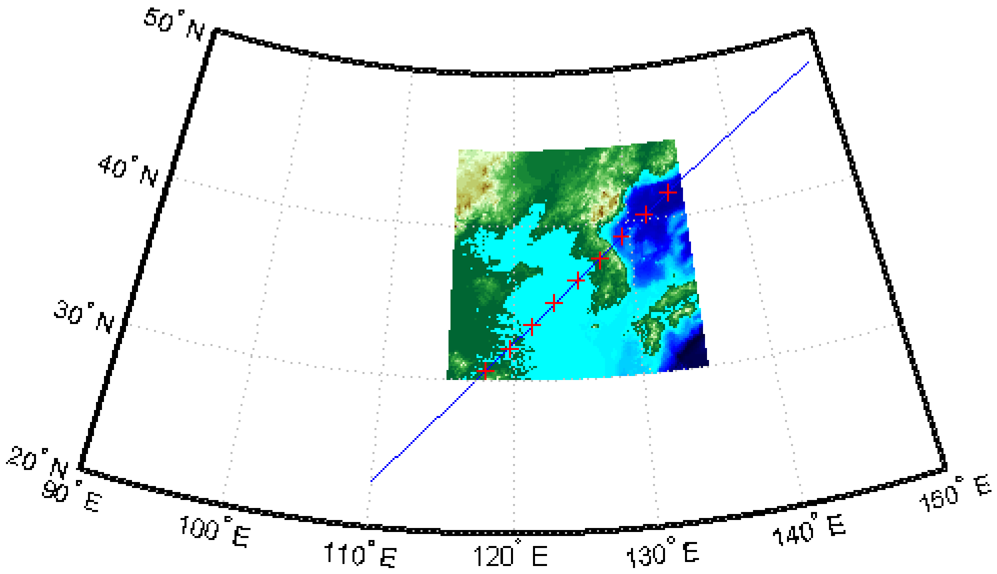
   ```
   korea = load('korea');
   [latlim, lonlim] = limitm(korea.map, korea.refvec);
   figure('Color','white')
   worldmap([20 50],[90 150])
   geoshow(korea.map, korea.refvec, 'DisplayType', 'texturemap');
   colormap(demcmap(korea.map))
   ```

2. Generate a track that crosses the DEM:

   ```
   [lat, lon] = track2(23, 110, 48, 149, [1 0], 'degrees', 20);
   geoshow(lat, lon, 'DisplayType', 'line')
   ```

3. Identify and mark points on the track that fall within the quadrangle outlining the DEM:

   ```
   tf = ingeoquad(lat, lon, latlim, lonlim);
   geoshow(lat(tf), lon(tf), 'DisplayType', 'point')
   ```



See Also `inpolygon`, `intersectgeoquad`

intersectgeoquad

Purpose

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.

Remarks

`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.

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:

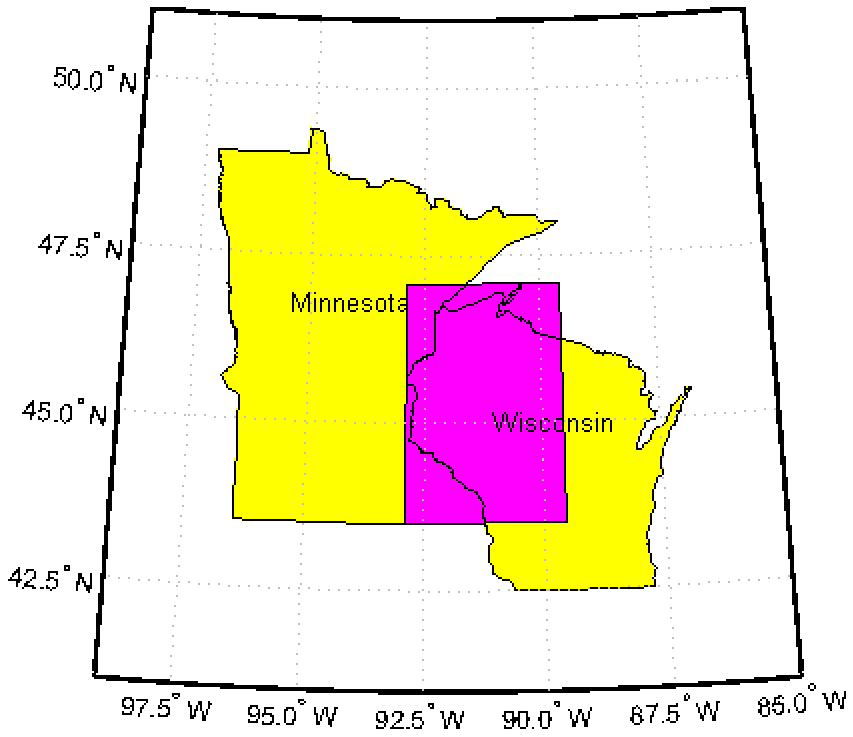
```
[latlim, lonlim] = intersectgeoquad( ...
                   [-30 90],[-180 180],[-90 30],[0 360])

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')
```



See Also `ingeoquad`, `outlinegeoquad`

Purpose

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]`.

Remarks

`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` (MATLAB function)

interpm

Purpose

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`

Purpose

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. The default value of the *`method`* string is `'linear'`, which results in linear, or Cartesian, interpolation between the numerical values entered. This is really just a pass-through to the MATLAB `interp1` function. Similarly, `'spline'` and `'cubic'` perform cubic spline and cubic interpolation, respectively. The `'rh'` method returns interpolated points that lie on rhumb lines between input data. Similarly, the `'gc'` method 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
```

Remarks

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`

Purpose

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. The default value of the *`method`* string is `'linear'`, which results in linear, or Cartesian, interpolation between the numerical values entered. This is really just a pass-through to the MATLAB `interp1` function. Similarly, `'spline'` and `'cubic'` perform cubic spline and cubic interpolation, respectively. The `'rh'` method returns interpolated points that lie on rhumb lines between input data. Similarly, the `'gc'` method 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
```

Remarks

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`

Purpose

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

Purpose

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`

Purpose

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.

Example

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`

Purpose

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

```
isShapeMultipart([0 0 1],[0 1 0])

ans =
     0

isShapeMultipart([0 0 1 NaN 2 2 3 3],[0 1 0 NaN 2 3 3 2])

ans =
     1

load coast
isShapeMultipart(lat, long)

ans =
     1

S = shaperead('concord_hydro_area');
isShapeMultipart( S(1).X,  S(1).Y)

ans =
     0

isShapeMultipart(S(14).X, S(14).Y)

ans =
     1
```

See Also

`polysplit`

km2deg, nm2deg, sm2deg

Purpose

Convert from distance units to degrees

Syntax

```
deg = km2deg(km)
deg = nm2deg(nm)
deg = sm2deg(sm)
deg = km2deg(km,radius)
deg = nm2deg(nm,radius)
deg = sm2deg(sm,radius)
deg = km2deg(km,sphere)
deg = nm2deg(nm,sphere)
deg = sm2deg(sm,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 = nm2deg(nm)` converts distances from nautical miles to degrees as measured along a great circle on a sphere with a radius of 6371 km (3440.065 nm), the mean radius of the Earth.

`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 = 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 = nm2deg(nm,radius)` and `deg = sm2deg(sm,radius)` work identically, except that both the input distance and radius must be in nautical miles and statute miles, respectively.

`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.

`deg = nm2deg(nm,`*`sphere`*`)` and `deg = sm2deg(sm,`*`sphere`*`)` work identically, except that the input units are nautical miles and statute miles, respectively.

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

`degtorad`, `radtodeg`, `deg2km`, `km2rad`, `km2nm`, `km2sm`, `deg2nm`, `nm2deg`, `nm2km`, `nm2sm`, `deg2sm`, `sm2deg`, `sm2km`, `sm2nm`

km2rad, nm2rad, sm2rad

Purpose

Convert from distance units to radians

Syntax

```
rad = km2rad(km)
rad = nm2rad(nm)
rad = sm2rad(sm)
rad = km2rad(km,radius)
rad = nm2rad(nm,radius)
rad = sm2rad(sm,radius)
rad = km2rad(km,sphere)
rad = nm2rad(nm,sphere)
rad = sm2rad(sm,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 = nm2rad(nm)`, and `rad = sm2rad(sm)` work identically, except that the input units are nautical miles and statute miles, respectively.

`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 = nm2rad(nm,radius)` and `rad = sm2rad(sm,radius)` work identically, except that both the input distance and radius must be in nautical miles and statute miles, respectively.

`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.

`rad = nm2rad(nm,`*`sphere`*`)` and `rad = sm2rad(sm,`*`sphere`*`)` work identically, except that the input units must be nautical miles and statute miles, respectively.

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

`degtorad`, `radtodeg`, `rad2km`, `km2deg`, `km2nm`, `km2sm`, `rad2nm`, `nm2deg`, `nm2km`, `nm2sm`, `rad2sm`, `sm2deg`, `sm2km`, `sm2nm`

km2nm, km2sm, nm2km, nm2sm, sm2km, sm2nm

Purpose

Convert distance between kilometers and miles

Syntax

```
nm = km2nm(km)
sm = km2sm(km)
km = nm2km(nm)
sm = nm2sm(nm)
km = sm2km(sm)
nm = sm2nm(sm)
```

Description

`nm = km2nm(km)` converts distances from kilometers to nautical miles.

`sm = km2sm(km)` works identically, except that the output units are statute miles.

`km = nm2km(nm)` converts distances from nautical miles to kilometers.

`sm = nm2sm(nm)` works identically, except that the output units are statute miles.

`km = sm2km(sm)` converts distances from statute miles to kilometers.

`nm = sm2nm(sm)` works identically, except that the output units are nautical miles.

Examples

How many statute miles is a *10k run*?

```
sm = km2sm(10)

sm =
    6.2137
```

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`

Purpose

Write geographic data to KML file

Syntax

```
kmlwrite(filename, lat, lon)
kmlwrite(filename, S)
kmlwrite(filename, address)
kmlwrite(..., param1, val1, param2, val2, ...)
```

Description

`kmlwrite(filename, lat, lon)` writes the latitude and longitude points `lat` and `lon` to disk in KML format. KML stands for Keyhole Markup Language. It is an XML dialect used by the Google Earth and Google Maps mapping services and similar applications. `lat` and `lon` are numeric vectors, specified in degrees. `lat` must be in the range `[-90, 90]`. There is no range constraint on `lon`; all longitudes are automatically wrapped to the range `[-180, 180]`, to adhere to the KML specification. `filename` must be a character string specifying the output file name and location. If an extension is included, it must be `.kml`.

`kmlwrite(filename, S)` writes a point or multipoint geostruct to disk in KML format. The `Geometry` field of `S` must be either `'Point'` or `'Multipoint'`. `S` must include `Lat` and `Lon` fields. (If `S` includes `X` and `Y` fields an error is issued). The attribute fields of `S` are presented as a table in the description tag of the placemark displayed for each element of `S`, in the same order as they appear in `S`.

`kmlwrite(filename, address)` specifies the location of a KML Placemark via an `address` string or cell array of strings. Each string represents an unstructured address with city, state, and/or postal code. If `address` is a cell array, each cell contains the address of a unique point.

`kmlwrite(..., param1, val1, param2, val2, ...)` specifies parameter-value pairs that set additional KML feature properties. Parameter names can be abbreviated and are case-insensitive.

The parameter-value pairs are listed below:

- `Name` — A string or cell array of strings that specifies a name displayed in the viewer as the label for the object. If the value is a string, the name is applied to all objects. If the value is a cell array, it must match in size to `lat` and `lon`, `S`, or `address`.

- `Description` — A string, cell array of strings, or an attribute spec, that specifies the contents to be displayed in the feature's description tag(s). The description appears in the description balloon when the user clicks either the feature name in the Google Earth application Places panel or clicks the placemark icon in the viewer window. If the value is a string, the description is applied to all objects. If the value is a cell array, it must match the size of `lat` and `lon`, `S`, or `address`. Use a cell array to customize descriptive tags for different placemarks.

  Description elements can be either plain text or marked up with HTML. When it is plain text, the Google Earth application applies basic formatting, replacing each `newline` with `<br>` and giving anchor tags to all valid URLs for the World Wide Web. The URL strings are converted to hyperlinks. This means that you do not need to surround a URL with `<A HREF>` tags in order to create a simple link. Examples of HTML tags recognized by the Google Earth application are provided on its Web site, `http://earth.google.com`.

- `Icon` — A string or cell array of strings that specifies a custom icon filename. If the value is a string, the value is applied to all objects. If the value is a cell array, it must have the same size as `lat` and `lon`, `S`, or `address`. If the icon filename is not in the current directory, or in a directory on the MATLAB path, specify a full or relative path name. The string can be an Internet URL. The URL must include the protocol prefix (e.g., `http://`).

- `IconScale` — A positive numeric scalar or array that specifies a scaling factor for the icon. If the value is a scalar, the value is applied to all objects. If the value is an array, it must have the same size as `lat` and `lon`, `S`, or `address`.

Remarks

Using an Attribute Spec to Control Formatting of Attributes

An attribute spec is a structure with field names of attributes that controls how the table is displayed in its description balloon. In it, each field name you want to display has two fields, `Format` and `AttributeLabel`.

When you provide geostruct, `S`, to `kmlwrite`, then the `Description` parameter can be an attribute spec. In this case, the attribute fields of `S` are displayed as a table in the description tag of the placemark for each element of `S`. (If you specify an attribute spec with `lat` and `lon` input syntax, the attribute spec is ignored.) The attribute spec can control:

- Which attributes are included in the table
- The name for the attribute
- The order in which attributes appear
- The formatting of attributes

The easiest way to construct an attribute spec is to call `makeattribspec`, and then modify the output to remove attributes or change the Format field for one or more attributes. The `lat` and `lon` fields of `S` are never treated as attributes.

Viewing the KML file with the Google Earth browser

A KML file may be displayed in a Google Earth browser. The Google Earth application must be installed on the system. On Microsoft Windows platforms you can display the KML file with:

```
winopen(filename)
```

For Unix and MAC users, display the KML file with:

```
cmd = 'googleearth ';
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

Viewing the KML file with a Web Browser

You can view KML files using the Google Maps mapping service in addition to using an installed Google Earth application. To do so, the file must be located on a web server that is accessible from the Internet.

A private intranet server will not suffice, because the Google Maps server must be able to access the URL that you provide to it. Here is a template for viewing your KML in a browser window via the Google Maps mapping service:

```
GMAPS_URL = 'http://maps.google.com/maps?q=';
KML_URL = 'http://<your web server and path to your KML file>';
web([GMAPS KML_URL])
```

You can only display a limited number of placemarks on a Google Maps page, and all placemarks must be geolocated using latitude-longitude coordinates (address-based placemarks are not supported). Google Mobile has further restrictions. See the Google KML documentation for more information.

Examples

Example 1 — Write a single point to a KML file

Add a description containing HTML markup, a name, and provide the location of an icon to display. Specifying an icon as a URL from the Web (as opposed to specifying one from a local file) makes the icon accessible to users of Google Maps service as well as to Google Earth users.

```
% Write a single point to a KML file.
% Add a description containing HTML, a name and an icon.
lat =  42.299827;
lon = -71.350273;
description = sprintf('%s<br>%s</br><br>%s</br>', ...
       '3 Apple Hill Drive', 'Natick, MA. 01760', ...
       'http://www.mathworks.com');
name = 'The MathWorks, Inc.';
filename = 'The_MathWorks.kml';
kmlwrite(filename, lat, lon, ...
       'Description', description, 'Name', name, 'Icon', ...
'http://www.mathworks.com/products/product_listing/images/ml_icon.gif');
```

Example 2 — Write the locations of major European cities to a KML file

Include 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]);
filename = 'European_Cities.kml';
kmlwrite(filename, cities, 'Name', {cities.Name}, 'Description',{});
```

Example 3 — Write the locations of several Australian cities to a KML file

List the addresses to be displayed in a cell array:

```
address = {'Perth, Australia', ...
              'Melbourne, Australia', ...
              'Sydney, Australia'};
filename = 'Australian_Cities.kml';
kmlwrite(filename, address, 'Name', address);
```

Example 4 — Unproject locations of Boston landmarks and write to a KML file

The Boston placenames file contains points stored in projected coordinates of meters, but Earth browsers require geographic coordinates (latitudes and longitudes). Begin by converting coordinates from meters to survey feet, inverting the projection to latitudes and longitudes, and then adding the latitudes and longitudes to the geostruct. To unproject properly, use the projection information extracted from the GeoTIFF file `boston.tif`:

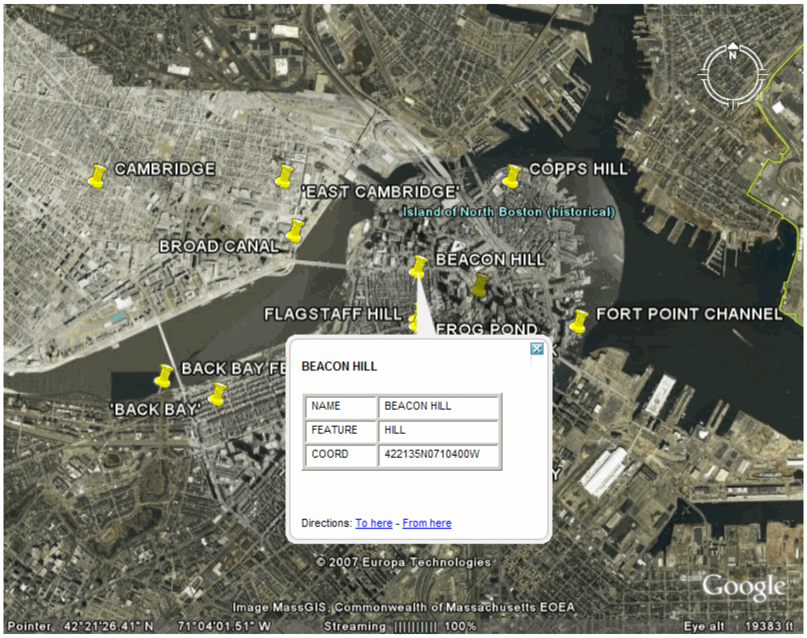
```
S = shaperead('boston_placenames');
proj = geotiffinfo('boston.tif');
surveyFeetPerMeter = unitsratio('sf','meter');
for k=1:numel(S)
    x =  surveyFeetPerMeter * S(k).X;
    y =  surveyFeetPerMeter * S(k).Y;
    [S(k).Lat, S(k).Lon] = projinv(proj, x, y);
end
filename = 'Boston_Placenames.kml';
kmlwrite(filename, S, 'Name', {S.NAME});
```

If you have the Google Earth application installed, you can view the file on Microsoft Windows as follows:

```
winopen(filename)
```

On UNIX or MAC, use:

```
cmd = 'googleearth ';
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```


CAMBRIDGE
'EAST CAMBRIDGE'
COPPS HILL
Island of North Boston (historical)
BROAD CANAL
BEACON HILL
FLAGSTAFF HILL
FROG POND
FORT POINT CHANNEL
BACK BAY FE
'BACK BAY'
© 2007 Europa Technologies
Image MassGIS, Commonwealth of Massachusetts EOEA
Google
Pointer 42°21'26.41" N 71°04'01.51" W Streaming ||||||||| 100% Eye alt 19383 ft

BEACON HILL

NAME	BEACON HILL
FEATURE	HILL
COORD	422135N0710400W

Directions: To here - From here

For a different view of this location and placename data, see “Tour Boston with the Map Viewer” on page 1-9.

See Also

`geoshow`, `makeattribspec`, `shaperead`, `shapewrite`

latlon2pix

Purpose

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 a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. `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.

Example

```
% 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

`makerefmat`, `pix2latlon`, `map2pix`

Purpose

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.

Example

```
figure; colormap(jet(5))
labels = {'apples','oranges','grapes','peachs','melons'};
lcolorbar(labels,'fontweight','bold');
```

See Also

`contourcmap`, `colormapeditor` (MATLAB function)

Purpose

Courses and distances between navigational waypoints

Syntax

```
[course,dist] = legs(lat,lon)
[course,dist] = legs(lat,lon,method)
[course,dist] = legs(pts) and [course,dist] = legs(pts,
   method)
mat = legs(lat,...)
```

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)` and `[course,dist] = legs(pts,method)` allow you to input the waypoints in a single two-column matrix, `pts`.

`mat = legs(lat,...)` 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

Purpose

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

```
load topo
axesm globe; view(120,30)
meshm(topo,topolegend); demcmap(topo)
lightm(0,90,'color','yellow')
material([.5 .5 1]); lighting phong
```

See Also

`light` (MATLAB function), `lightmui`

limitm

Purpose

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` with referencing vector or matrix `R`. `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. 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

Using a familiar data grid,

```
load topo
[latlimits,lonlimits] = limitm(topo,topolegend)
latlimits =
   -90    90
lonlimits =
     0   360
```

Which is expected, because `topo` covers the whole globe.

See Also

`makerefmat`

linecirc

Purpose

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`

Purpose

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`.

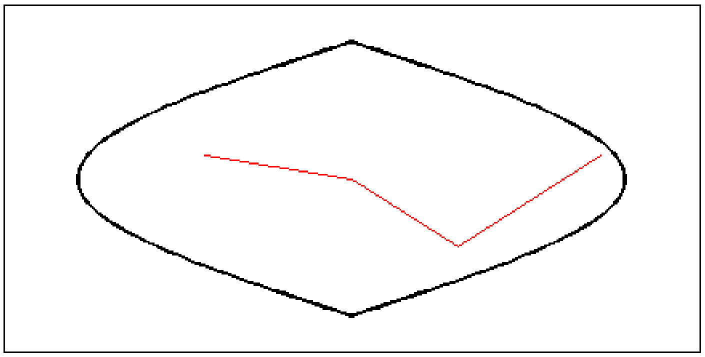
`h = linem(lat,lon,z)` displays a line object in three dimensions, where `z` is the same size as `lat` and `lon` and contains the desired altitude data. `z` is independent of `AngleUnits`. If omitted, all points are assigned a z-value of 0 by default.

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 for a reference ellipsoid vector of `[1 0]` (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`

Purpose

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` 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 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.

`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.

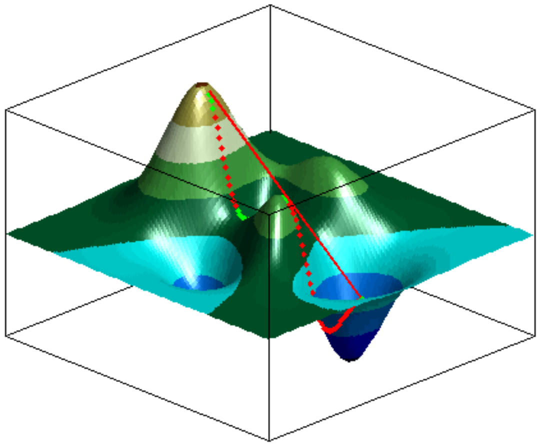
`vis = los2(Z,R,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt, ...`
`alt2opt,actualradius,effectiveradius)` assumes a larger radius for propagation of the line of sight. This can account for the curvature of the signal path due to refraction in the atmosphere. For example, radio propagation in the atmosphere 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.

Example

```
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',almanac('earth','sphere','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`

Purpose

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` 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. 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`

Purpose

Convert local vertical to geocentric (ECEF) coordinates

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 row vector with 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

`ecef2geodetic` | `ecef2lv` | `elevation` | `geodetic2ecef`

Purpose

Semimajor axis of ellipse given semiminor axis and eccentricity

Syntax

```
semimajor = majaxis(semiminor,eccentricity)
semimajor = majaxis([semiminor eccentricity])
```

Description

`semimajor = majaxis(semiminor,eccentricity)` returns the semimajor axis length corresponding to the input semiminor axis and eccentricity.

`semimajor = majaxis([semiminor eccentricity])` allows the inputs to be packed into a single two-column input of the form `[semiminor eccentricity]`.

The semimajor axis, the first element of a standard Mapping Toolbox ellipsoid vector, can be determined given both the semiminor axis and the eccentricity.

Examples

Using the default values for the Earth,

```
semimajor = majaxis(6356.7523,0.0818192)
semimajor =
   6.3781e+03
```

This is the default semimajor axis.

See Also

`almanac`, `axes2ecc`, `minaxis`

Purpose

Attribute specification from geographic data structure

Syntax

```
attribspec = makeattribspec(S)
```

Description

`attribspec = makeattribspec(S)` analyzes a geographic data structure `S` and constructs an attribute specification suitable for use with `kmlwrite`. `kmlwrite`, given geostruct input, 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 `attribspec`, and then pass it to `kmlwrite` to exert control over which geostruct attribute fields are written to the HTML table and the format of the string conversion.

`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, on the next level, contains a scalar structure with a fixed pair of fields:

`AttributeLabel`	A string that corresponds to the name of the attribute field in the geographic data structure. 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.

Example

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

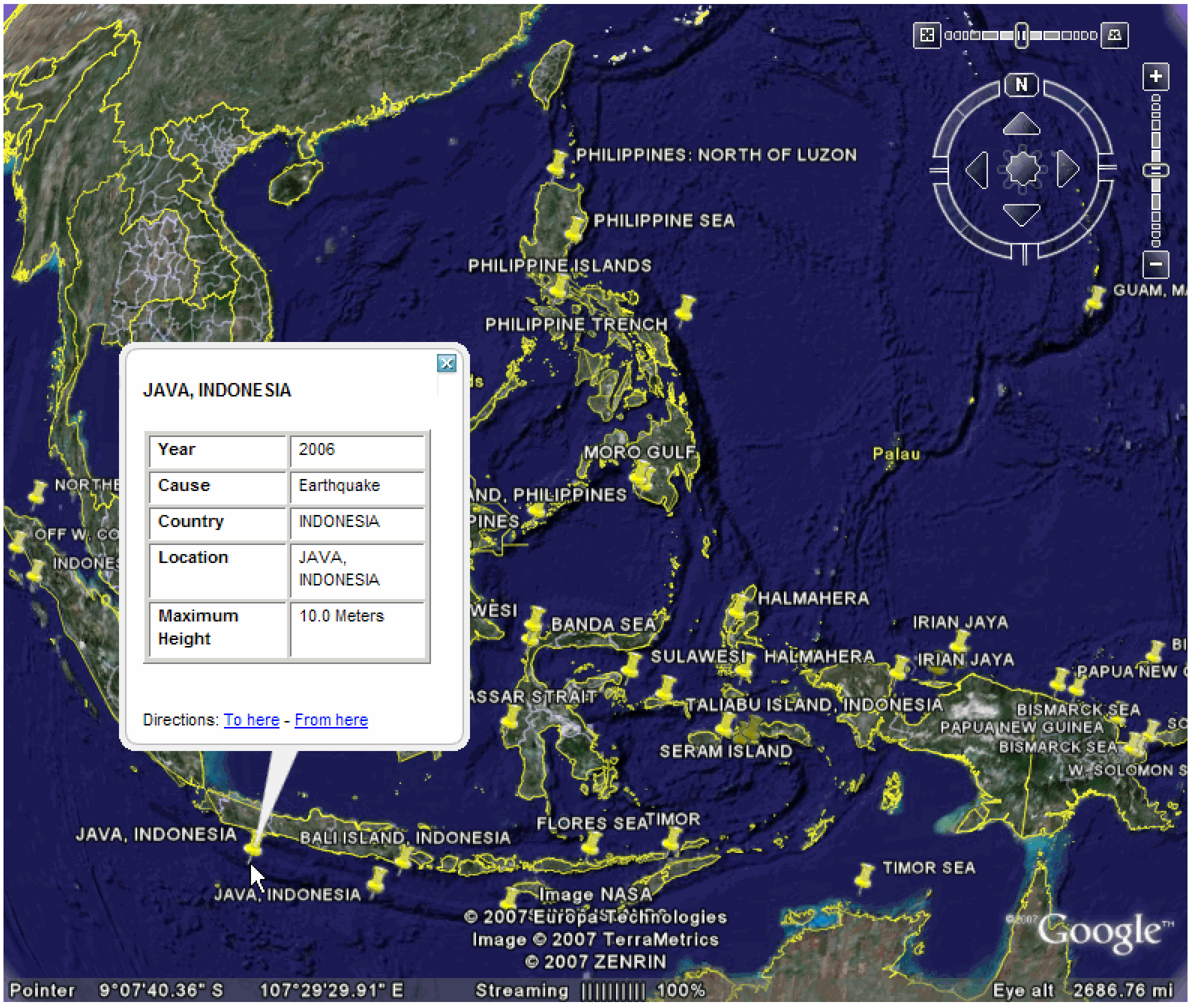
```
desiredAttributes = ...
       {'Max_Height', 'Cause', 'Year', 'Location', 'Country'};
allAttributes = fieldnames(attribspec);
attributes = setdiff(allAttributes, desiredAttributes);
```

```
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})
```

A view of Southeast Asia produced by the Google Earth application shows the selected, formatted attributes displayed for a 2006 tsunami in Indonesia.


JAVA, INDONESIA

Year	2006
Cause	Earthquake
Country	INDONESIA
Location	JAVA, INDONESIA
Maximum Height	10.0 Meters

Directions: To here - From here

See also

`kmlwrite`, `makedbfspec`, `shapewrite`

Purpose

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 character fields. The default value for noninteger numeric fields is 6.

Example

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

Purpose

Convert ordinary graphics object to mapped object

Syntax

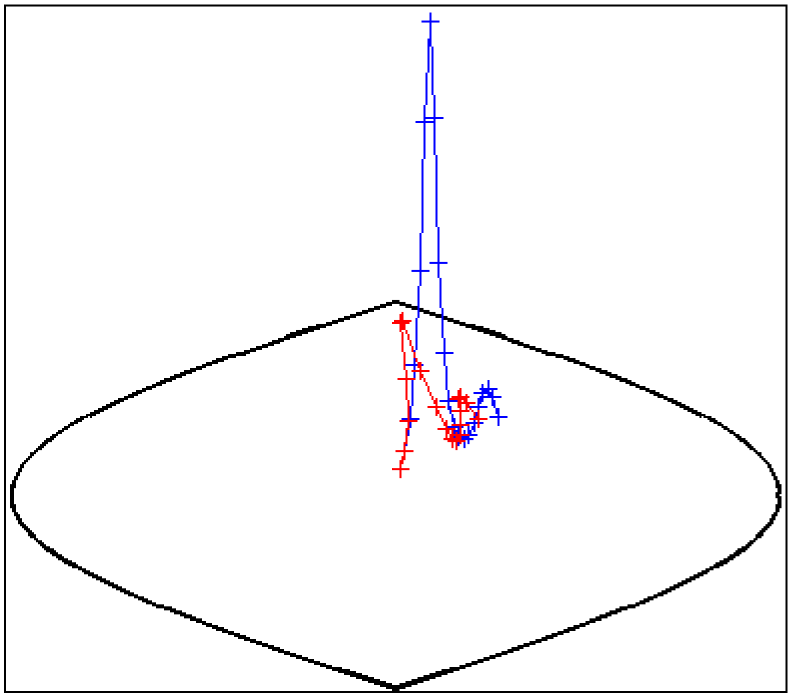
```
makemapped(h)
```

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`.

Example

```
axesm('miller','geoid',[25 0])
framem
plot(humps,'b+-')

h = plot(humps,'r+-');
trimcart(h)
makemapped(h)

setm(gca,'MapProjection','sinusoid')
```



Remarks Objects should first be trimmed to the map frame using `trimcart`. This avoids problems in taking inverse map projections with out-of-range data.

See Also `trimcart`, `handlem`, `cart2grn`

Purpose

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.
		
		With `'RasterSize'`, you may also provide a size vector having more than two elements. This enables usage such as:
		
		`R = makerefmat('RasterSize', ...`
		`   size(RGB), ...)`
		
		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].

`Latlim`	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].
`Lonlim`	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 `'Lonlim'` vector must be ascending in value. In other words, the limits must be unwrapped. The default value is [0 1].

Parameter Name	Data Type	Value
`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'`.
`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'`.

Definition

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.

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))
```

See Also

`latlon2pix` | `map2pix` | `pix2latlon` | `pix2map` | `refvec2mat` | `worldfileread` | `worldfilewrite`

Tutorials

- Creating a Half-Resolution Georeferenced Image

How To

- "Understanding Raster Geodata" on page 2-33

Purpose

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'`

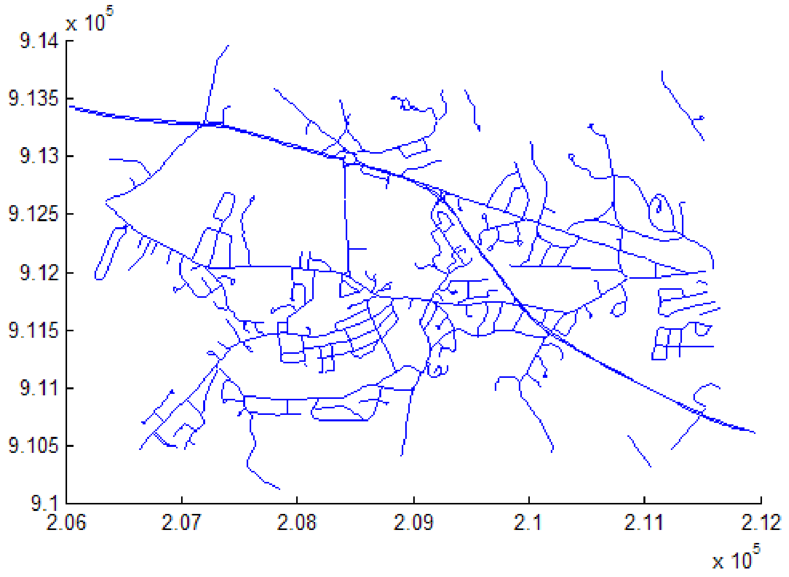
- Polygons: `'FaceColor'`, `'FaceAlpha'`, `'LineStyle'`, `'LineWidth'`, `'EdgeColor'`, `'EdgeAlpha'`, and `'Visible'`

Examples

The following examples import a shapefile containing road data and symbolize it in several ways using symbol specifications.

Example 1 — Default Color

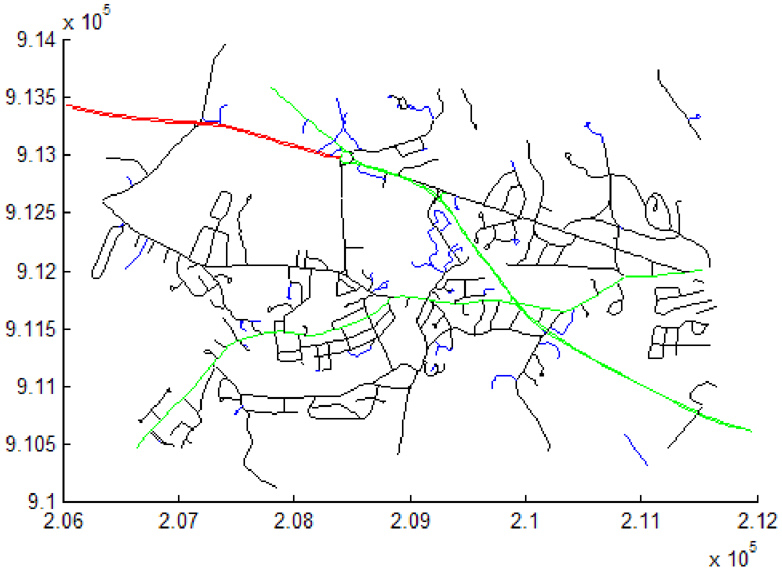
```
roads = shaperead('concord_roads.shp');
blueRoads = makesymbolspec('Line',{'Default','Color',[0 0 1]});
mapshow(roads,'SymbolSpec',blueRoads);
```



Example 2 — Discrete Attribute Based

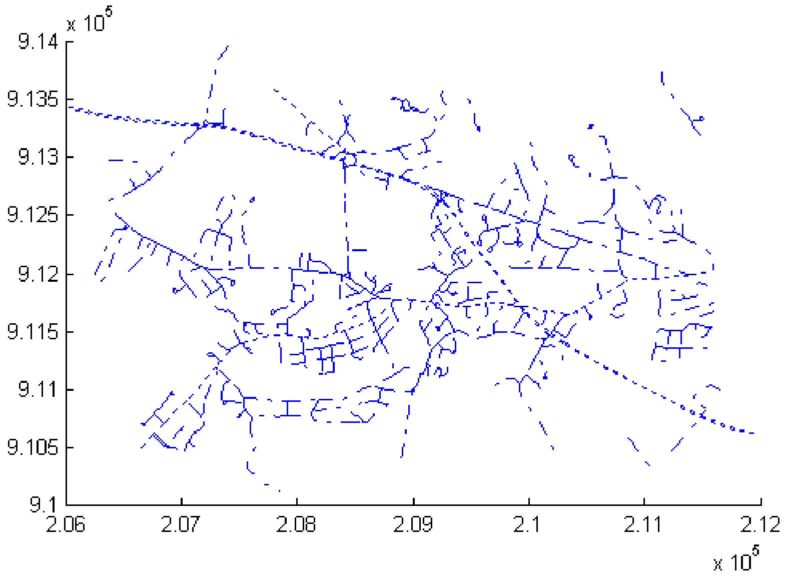
```
roads = shaperead('concord_roads.shp');
roadColors = ...
makesymbolspec('Line',{'CLASS',2,'Color','r'},...
                      {'CLASS',3,'Color','g'},...
                      {'CLASS',6,'Color','b'},...
```

```
                     {'Default','Color','k'});
mapshow(roads,'SymbolSpec',roadColors);
```



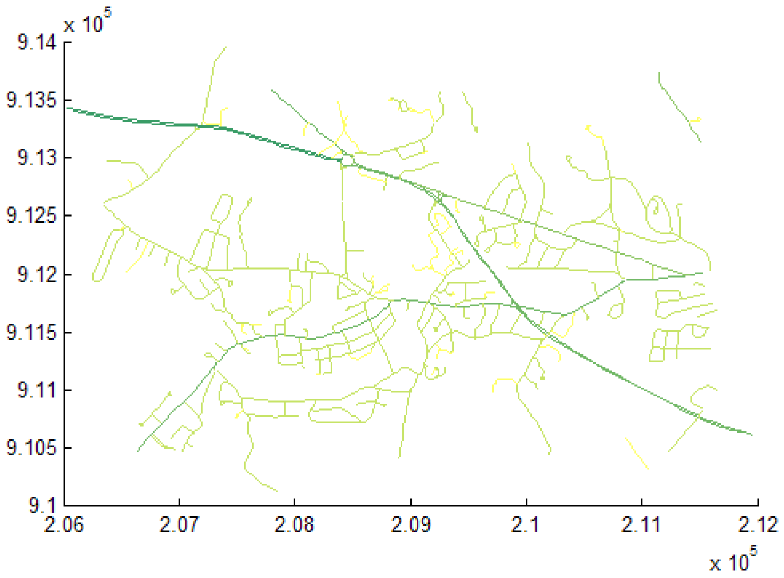
Example 3 — Using a Range of Attribute Values

```
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`

map2pix

Purpose

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 a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to map coordinates. `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.

Example

```
% 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`, `makerefmat`, `pix2map`, `worldfileread`

Purpose

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 a 3-by-2 affine referencing matrix. `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

Purpose

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` defines the name of the M-file 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

When map projections are added to the toolbox, the list structure needs to be extended. For example, if a new projection is added to the default list, then a new entry in the list structure would be

```
list.Name(61)           = 'My Projection'
list.IdString(61)       = 'newprojection';
list.Classification(61) = 'New Projection Type';
list.ClassCode(61)      = 'Code';
```

See Also

`maps`, `axesmui`

mapoutline

Purpose

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 a 3-by-2 affine referencing matrix. `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.

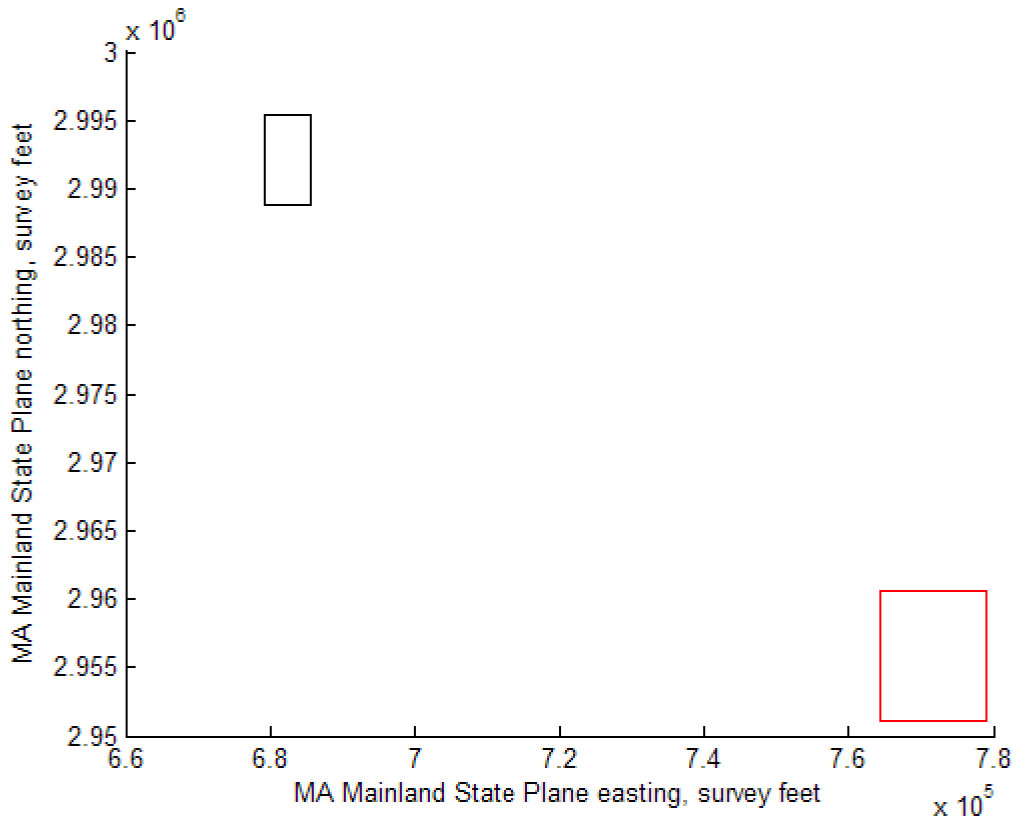
Example

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 `makerefmat`, `mapbbox`, `pixcenters`, `pix2map`

Purpose

Interpolate heights between waypoints on regular data grid

Syntax

```
[zi,rng,lat,lon] = mapprofile
[zi,rng,lat,lon] = mapprofile(Z,R,lat,lon)
[zi,rng,lat,lon] = mapprofile(Z,R,lat,lon,rngunits)
[zi,rng,lat,lon] = mapprofile(Z,R,lat,lon,ellipsoid)
[zi,rng,lat,lon] = ...    mapprofile(Z,R,lat,lon,rngunits,
   trackmethod,interpmethod)
[zi,rng,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,rng,lat,lon] = mapprofile` returns the values of the profile without displaying them. The output `zi` contains interpolated values from map along great circles between the waypoints. `rng` 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,rng,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` 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.

`[zi,rng,lat,lon] = mapprofile(Z,R,lat,lon,rngunits)` 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 grid. If omitted, `'degrees'` is assumed.

`[zi,rng,lat,lon] = mapprofile(Z,R,lat,lon,ellipsoid)` uses the provided ellipsoid definition in computing the range along the profile. The ellipsoid vector is of the form `[semimajor axes, eccentricity]`. The output range is reported in the same distance units as the semimajor axes of the ellipsoid vector. If omitted, the range vector is for a sphere.

`[zi,rng,lat,lon] = ...`
`mapprofile(Z,R,lat,lon,rngunits,trackmethod,interpmethod)`
and `[zi,rng,lat,lon] = ...`
`mapprofile(Z,R,lat,lon,ellipsoid,trackmethod,interpmethod)`
control the interpolation methods used. Valid `trackmethods` are `'gc'` for great circle tracks between waypoints, and `'rh'` for rhumb lines. Valid `interpmethods` for interpolation within the data grid are `'bilinear'` for linear interpolation, `'bicubic'` for cubic interpolation, and `'nearest'` for nearest neighbor interpolation. If omitted, `'gc'` and `'bilinear'` are assumed.

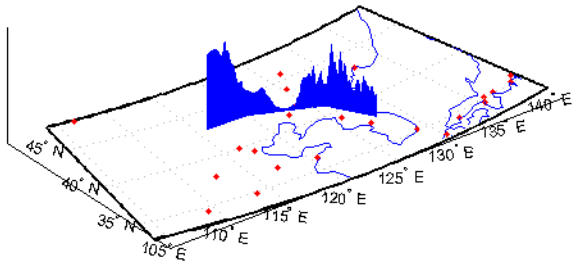
Examples

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)
```

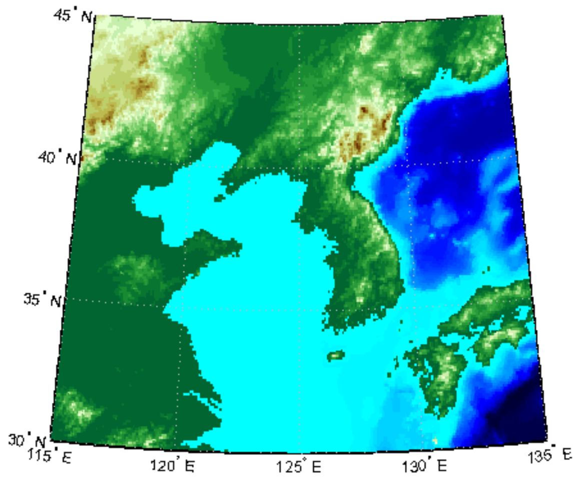
```
load coast
plotm(lat, long)
geoshow('worldcities.shp', 'Marker', '.', 'Color', 'red')
```



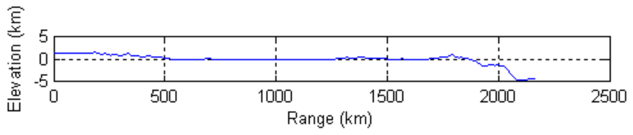
When you select more than two waypoints, the automatically generated figure displays the result in three dimensions. The following example shows the relative sizes of the mountains in northern China compared to the depths of the Sea of Japan. The call to `mapprofile` without input arguments requires 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)
[z,rng,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.



```
figure
plot(deg2km(rng),z/1000)
daspect([ 1 1/20 1 ]);
xlabel 'Range (km)'
ylabel 'Elevation (km)'
```



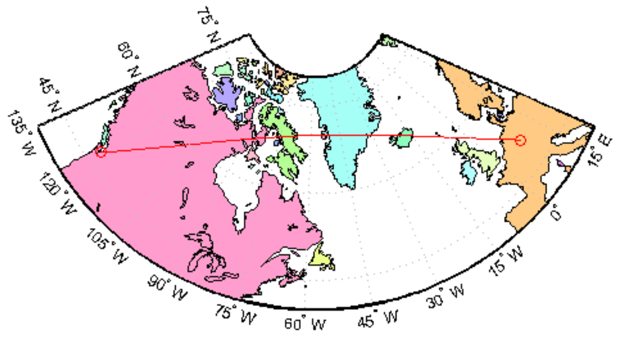
Naturally, the profile you get depends on the transect locations you pick.

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 = strmatch('Seattle', {cities(:).Name});
Frankfurt = strmatch('Frankfurt', {cities(:).Name});
lat = [cities(Seattle).Lat cities(Frankfurt).Lat]
lon = [cities(Seattle).Lon cities(Frankfurt).Lon]
load topo
[valp,rngp,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`

maps

Purpose

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`

mapshow

Purpose

Display map data without projection

Syntax

```
mapshow(x,y)
mapshow(x,y, ..., 'DisplayType', displaytype, ...)
mapshow(x,y,z, ..., 'DisplayType', displaytype, ...)
mapshow(Z,R, ..., 'DisplayType', displaytype,...)
mapshow(x,y,I)
mapshow(x,y,BW)
mapshow(x,y,A,cmap)
mapshow(x,y,RGB)
mapshow(I,R)
mapshow(BW,R)
mapshow(RGB,R)
mapshow(A,cmap,R)
mapshow(s)
mapshow(s,...,'SymbolSpec',symspec, ...)
mapshow(filename)
mapshow(..., param1, val1, param2, val2, ...)
mapshow(ax, ...)
mapshow(..., 'Parent', ax, ...)
h = mapshow(...)
```

Description

`mapshow(x,y)` or `mapshow(x,y, ..., 'DisplayType', displaytype, ...)` displays the coordinate vectors `x` and `y`. `x` and `y` can contain embedded NaNs, delimiting individual lines or polygon parts. `displaytype` can be `'point'`, `'line'`, or `'polygon'` and defaults to `'line'`.

`mapshow(x,y,z, ..., 'DisplayType', displaytype, ...)` displays a geolocated data grid. `x` and `y` are M-by-N coordinate arrays, `z` is an M-by-N array of class `double`, and `displaytype` is `'surface'`, `'mesh'`, `'texturemap'`, or `'contour'`. `z` can contain `NaN` values.

`mapshow(Z,R, ..., 'DisplayType', displaytype,...)` displays a regular data grid, `Z`. `Z` is class `double` and `displaytype` can be `'surface'`, `'mesh'`, `'texturemap'`, or `'contour'`. `R` is a referencing matrix that relates the subscripts of `Z` to map coordinates. If

DisplayType is 'texturemap', mapshow constructs a surface with ZData values set to 0.

mapshow(x,y,I), mapshow(x,y,BW), mapshow(x,y,A,cmap), and mapshow(x,y,RGB) display a geolocated image as a texturemap on a zero-elevation surface. x and y are geolocation arrays in map coordinates; I is a grayscale image, BW is a logical image, A is an indexed image with colormap cmap, or rgb is a truecolor image. x, y, and the image array must match in size. If specified, DisplayType must be set to 'image'. Examples of geolocated images include a color composite from a satellite swath or an image originally referenced to a different coordinate system.

mapshow(I,R), mapshow(BW,R), mapshow(RGB,R), and mapshow(A,cmap,R) display an image georeferenced to map coordinates through the referencing matrix R. It constructs an image object if the display geometry permits; otherwise, the image is shown as a texturemap on a zero-elevation surface. If specified, 'DisplayType' must be set to 'image'.

mapshow(s) or mapshow(s,...,'SymbolSpec',symspec, ...) display 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 they are used directly to plot features in map coordinates. If Lat and Lon fields are present in s instead, the coordinates are projected using the Plate Carree projection and a warning is issued. symspec specifies the symbolization rules used for the vector data through a structure returned by makesymbolspec.

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.

mapshow(filename) displays data from filename, according to the type of file format. 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'`
ARC ASCII GRID	`'surface'` (can be overridden)
SDTS raster	`'surface'` (can be overridden)

`mapshow(..., param1, val1, param2, val2, ...)` specifies parameter/value pairs that modify the type of display or set MATLAB graphics properties. Parameter names can be abbreviated and are not case-sensitive. Refer to the MATLAB Graphics documentation on line, patch, image, surface, mesh, and contour Handle Graphics object properties for a complete description of these properties and their values.

`mapshow(ax, ...)` and `mapshow(..., 'Parent', ax, ...)` set the axes parent to `ax`.

`h = mapshow(...)` returns a 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.

Parameters

Parameters for `mapshow` include

- `DisplayType`: The `DisplayType` parameter specifies the type of graphic display for the data. The value must be consistent with the type of data being displayed, as shown in the following table:

Data Type	Value
`vector`	`'point'`, `'line'`, or `'polygon'`

Data Type	Value
`image`	`'image'`
`grid`	`'surface'`, `'mesh'`, `'texturemap'`, or `'contour'`

- `SymbolSpec`: The `SymbolSpec` parameter specifies the symbolization rules used for vector data through a structure returned by `makesymbolspec`. It is used only 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 property name/property value pair in the symbolspec, prefix the word `'Default'` to the graphics property name (listed in the preceding table).

If `PropertyN` is `'SymbolSpec'`, then `ValueN` must be `symspec`. `symspec` should conform to the structure returned by `makesymbolspec`.

When you use `'SymbolSpec'/symspec` and other property name/property value pairs together, the property name/property value pairs override any settings in `symspec`.

Note If you display a polygon, do not set `'EdgeColor'` to either `'flat'` or `'interp'`. This combination may result in a warning.

Graphics Properties

In addition to specifying a parent axes, you can set any appropriate property for a `point`, `line`, and `polygon` `DisplayType`, as follows:

DisplayType	Properties
`'line'`	Any MATLAB line property
`'point'`	Any MATLAB line marker property
`'polygon'`	Any MATLAB patch property

See the MATLAB Graphics documentation for line, patch, image, and surface properties for complete descriptions of these properties and their values.

Remarks

You can use mapshow to display vector data in an axesm figure. However, you should not subsequently change the map projection using setm.

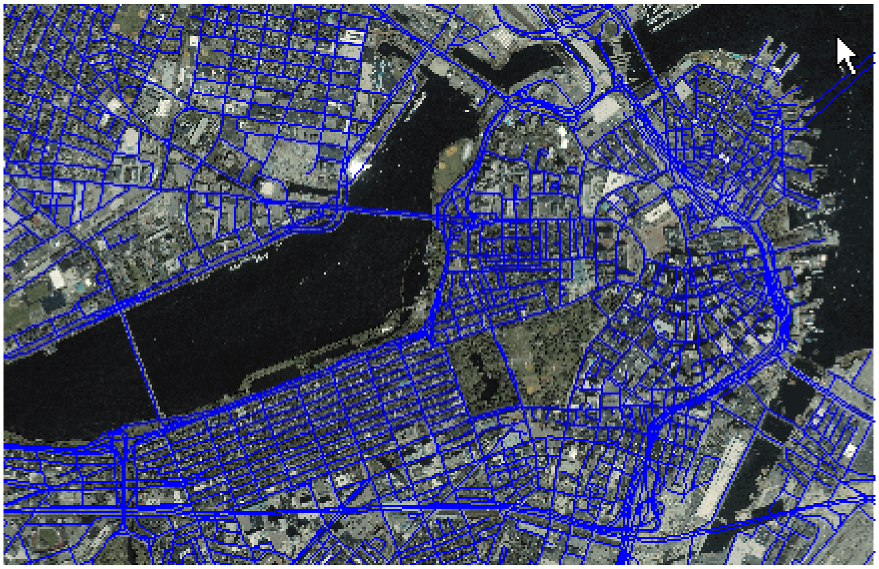
mapshow adds graphics to the current axes (it does not clear it first), enabling you to create multiple raster and vector map layers. If you do not want mapshow to draw on top of an existing map, create a new figure or subplot before calling it.

Examples

Example 1

Overlay Boston roads on an orthophoto. You need to convert Boston road vectors to units of survey feet before overlaying them on the image. Note that mapshow draws a new layer in the axes rather than replacing its contents:

```
figure
mapshow boston.tif
axis image off

% The orthophoto is in survey feet, the roads are in meters;
% convert the road units to feet before overlaying them.
S = shaperead('boston_roads.shp');
surveyFeetPerMeter = unitsratio('sf','meter');
x = surveyFeetPerMeter * [S.X];
y = surveyFeetPerMeter * [S.Y];
mapshow(x,y)
```

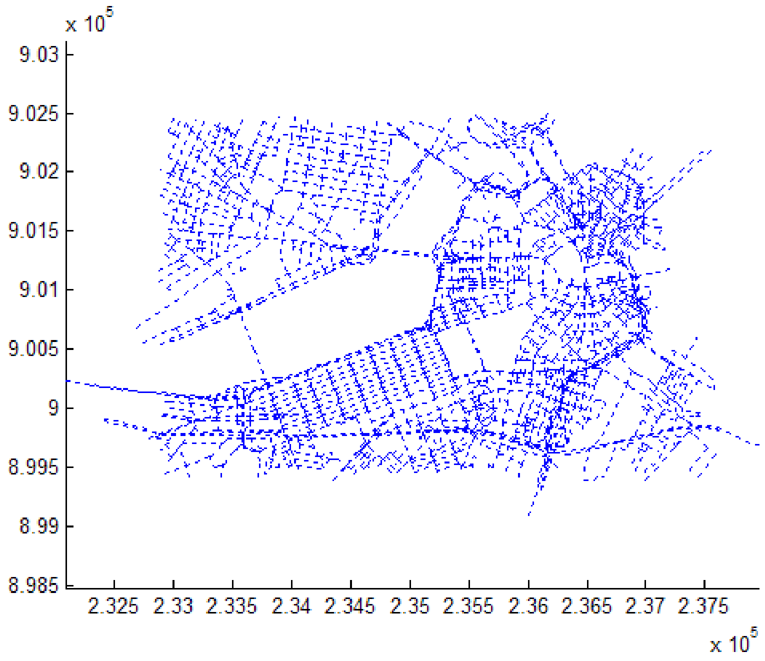


boston.tif image copyright © GeoEye, all rights reserved.

Example 2

Display Boston roads and change the line style:

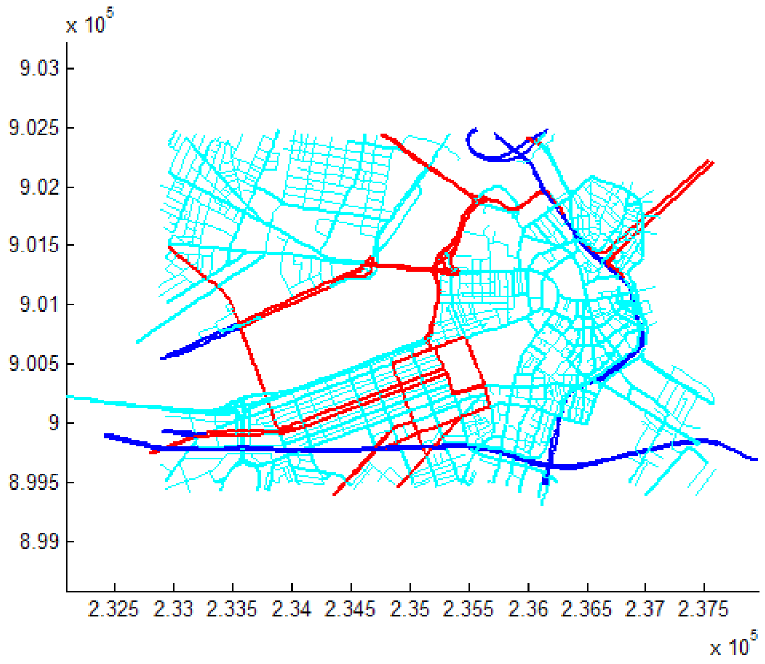
```
roads = shaperead('boston_roads.shp');
figure
mapshow(roads,'LineStyle',':');
```



Example 3

Display the Boston roads shapes using a symbolspec:

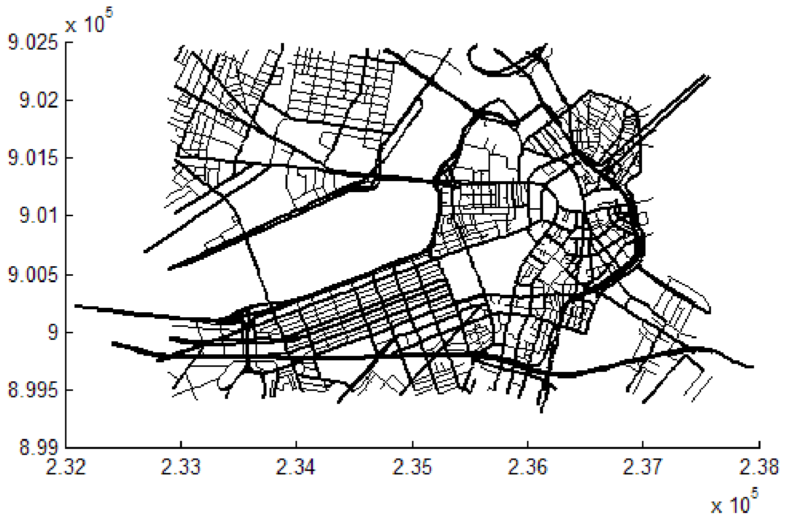
```
% Create a SymbolSpec to color local roads:
%   (ADMIN_TYPE=0) cyan, state roads (ADMIN_TYPE=3) red.
% Hide very minor roads (CLASS=6).
% Make all roads that are major or larger (CLASS=1-4)
%   have a LineWidth of 2.
roadspec = makesymbolspec('Line',...
                          {'ADMIN_TYPE',0,'Color','cyan'}, ...
                          {'ADMIN_TYPE',3,'Color','red'},...
                          {'CLASS',6,'Visible','off'},...
                          {'CLASS',[1 4],'LineWidth',2});
figure
mapshow('boston_roads.shp','SymbolSpec',roadspec);
```



Example 4

Override default properties in combination with a symbolspec.

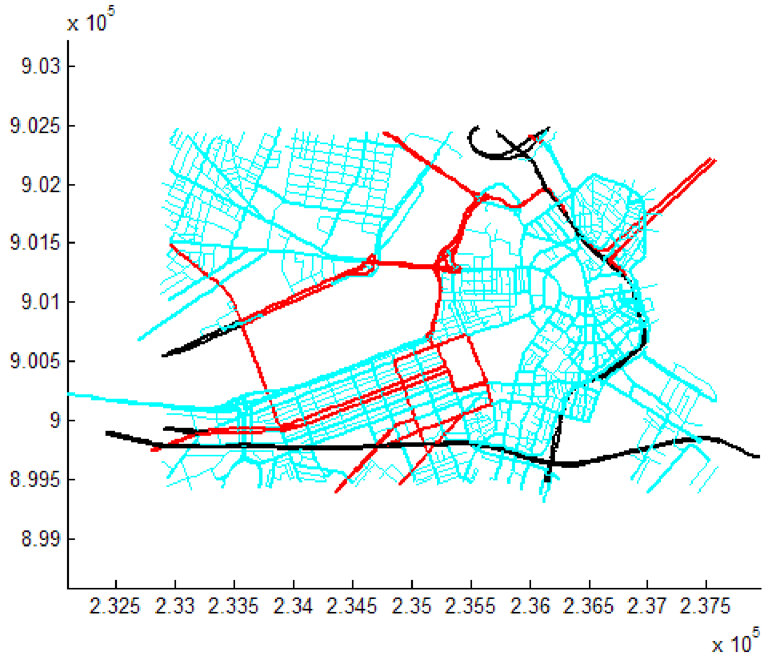
```
roadspec = makesymbolspec('Line',...
                          {'Default', 'Color', 'yellow'}, ...
                          {'ADMIN_TYPE',0,'Color','c'}, ...
                          {'ADMIN_TYPE',3,'Color','r'},...
                          {'CLASS',6,'Visible','off'},...
                          {'CLASS',[1 4],'LineWidth',2});
figure
mapshow('boston_roads.shp', 'Color', 'black', ...
'SymbolSpec', roadspec);
```



Example 5

Override default properties of the line with a symbolspec:

```
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});
figure
mapshow('boston_roads.shp','SymbolSpec',roadspec);
```

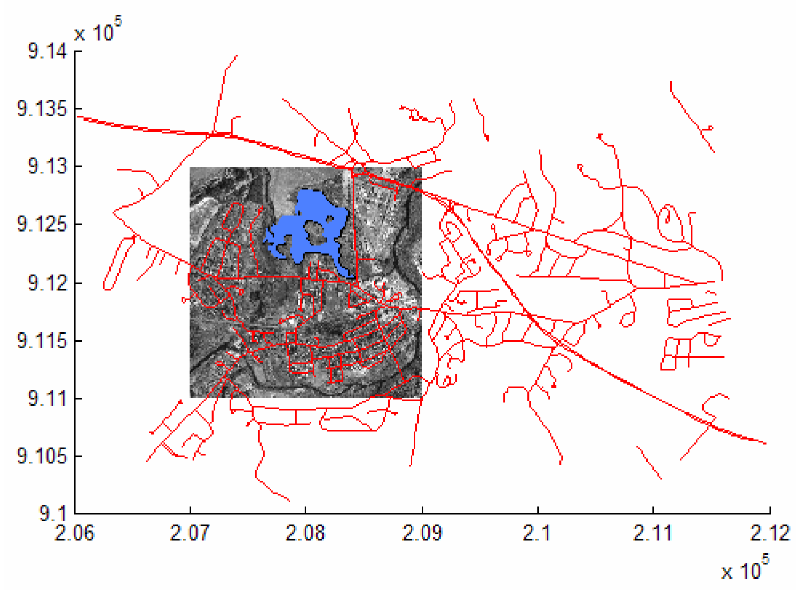


Example 6

Display an orthophoto of Concord, MA, including a pond with three large islands:

```
[ortho, cmap] = imread('concord_ortho_w.tif');
R = worldfileread('concord_ortho_w.tfw');
figure
mapshow(ortho, cmap, R)

% Overlay a polygon representing the same pond
% (feature 14 in the concord_hydro_area shapefile).
% Note that the islands are visible in the orthophoto
% through three "holes" in the pond polygon.
pond = shaperead('concord_hydro_area.shp', 'RecordNumbers', 14);
mapshow(pond, 'FaceColor', [0.3 0.5 1], 'EdgeColor', 'black')
```

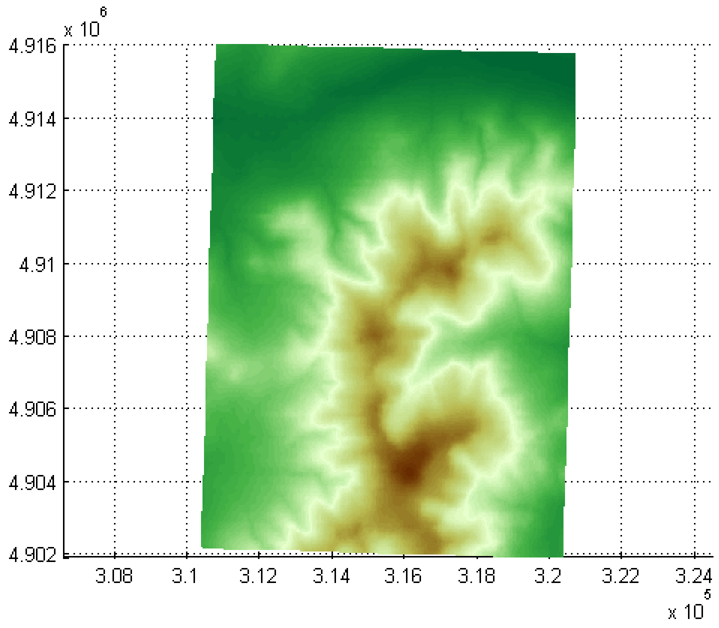
```
% Overlay roads in the same figure.
mapshow('concord_roads.shp', 'Color', 'red', 'LineWidth', 1);
```



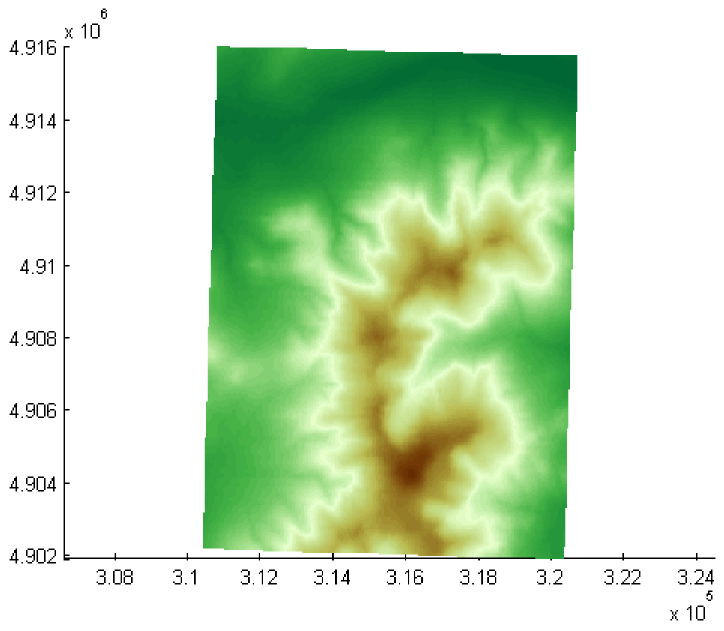
Example 7

Read and view the Mount Washington SDTS DEM terrain data three different ways:

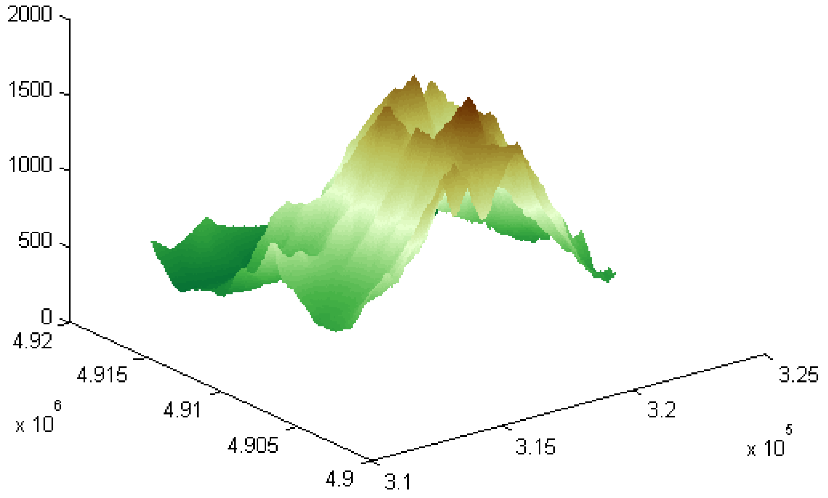
```
[Z, R] = sdtsdemread('9129CATD.DDF');

% View the Mount Washington terrain data as a mesh.
figure
mapshow(Z, R, 'DisplayType', 'mesh');
colormap(demcmap(Z))
```



```
% View the Mount Washington terrain data as a surface.
figure
mapshow(Z, R, 'DisplayType', 'surface');
colormap(demcmap(Z))
```



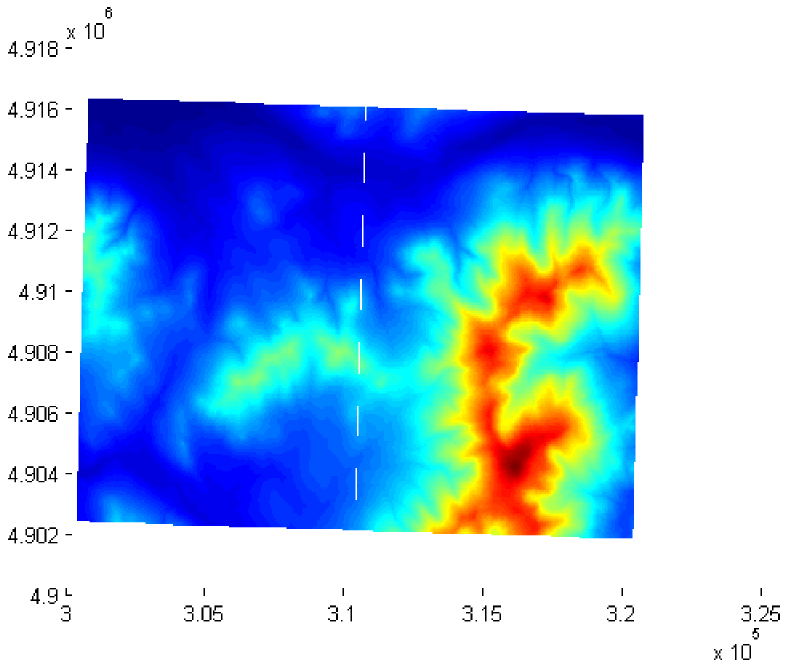
```
% View as a 3-D surface.
view(3);
axis normal
```



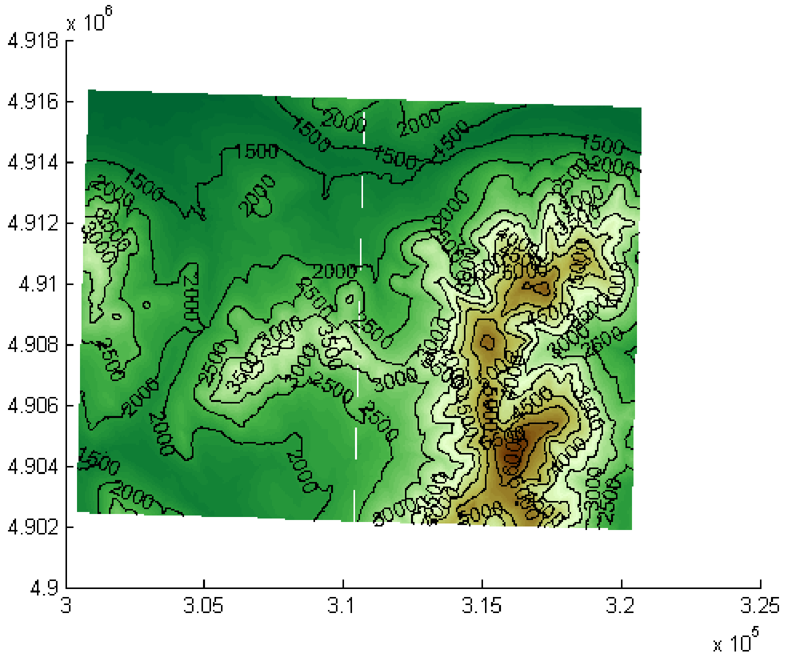
Example 8

Display the grid and contour lines of Mount Washington and Mount Dartmouth.

```
% Read the terrain data files.
[Z_W, R_W] = arcgridread('MtWashington-ft.grd');
[Z_D, R_D] = arcgridread('MountDartmouth-ft.grd');

% Display the terrain data as a texture map.
figure
hold on
h1 = mapshow(Z_W, R_W, 'DisplayType', 'texturemap');
h2 = mapshow(Z_D, R_D, 'DisplayType', 'texturemap');
set([h1, h2],'FaceColor','flat');
```



```
% Overlay black contour lines with labels onto the texturemap.
mapshow(Z_W, R_W, 'DisplayType', 'contour', ...
    'LineColor','black', 'ShowText', 'on');
mapshow(Z_D, R_D, 'DisplayType', 'contour', ...
    'LineColor','black', 'ShowText', 'on');

% Set the colormap appropriate to terrain elevation.
colormap(demcmap(Z_W))
```



See Also `geoshow`, `makesymbolspec`, `mapview`, `shaperead`

maptriml

Purpose

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 `NaN`s inserted at trim points.

See Also

`maptrimp`, `maptrims`

Purpose

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.

Remarks

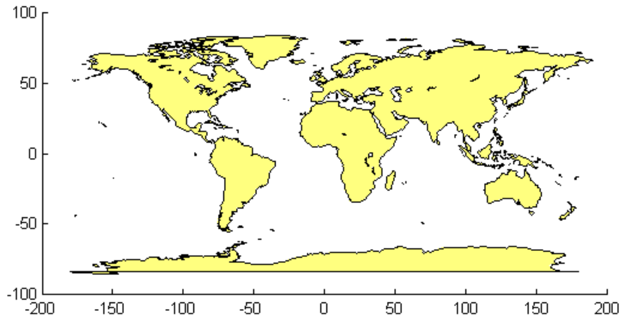
`maptrimp` conditions the longitude limits such that:

- `lonlim(2)` always exceeds `lonlim(1)`
- `lonlim(2)` never exceeds `lonlim(1)` by more than `360`
- `lonlim(1) < 180` or `lonlim(2) > -180`
- Should the quadrangle span the Greenwich meridian, then that meridian appears at longitude = 0.

Example

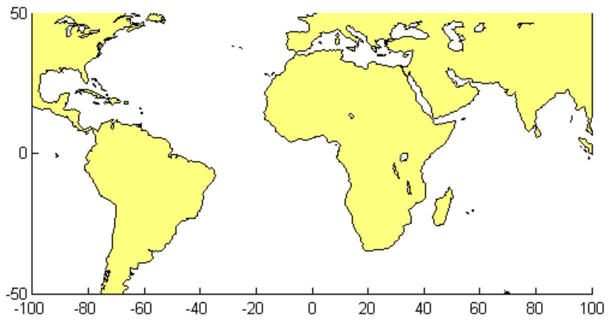
Display a world map of coastline data, trim the dataset to a specific geographic area, and display a map of this trimmed data.

```
coast = load('coast.mat');
figure
mapshow(coast.long, coast.lat, 'DisplayType', 'polygon');
```



Original Map

```
latlim = [-50 50];
lonlim = [-100 50];
[latTrimmed, lonTrimmed] = maptrimp(coast.lat, coast.long, ...
   latlim, lonlim);
figure
mapshow(lonTrimmed, latTrimmed, 'DisplayType', 'polygon');
```



Map with Trimmed Data

See Also

`maptriml`, `maptrims`

Purpose

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`. `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. `latlim` and `lonlim` are two-element vectors, defining the latitude and longitude limits respectively. The output grid `Z_trimmed` has the same sample size as the input.

`[Z_trimmed] = maptrims(Z,R,latlim,lonlim,cellDensity)` uses `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 or matrix for the trimmed data grid. If `R` is a referencing vector, then `R_trimmed` is a referencing vector. Likewise, if `R` is a referencing matrix, then `R_trimmed` is a referencing matrix.

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`

Purpose

Interactive map viewer

Syntax

```
mapview
```

Description

Use the Map Viewer to work with vector, image, and raster data grids in a map coordinate system: 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.

`mapview` (with no arguments) starts a new Map Viewer in an empty state. The Map Viewer is a self-contained GUI for viewing geospatial data in map (*x*-*y*) coordinates. For usage information, see the following sections. You can also work through the Map Viewer tutorial, "Tour Boston with the Map Viewer" on page 1-9.

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 directory or navigate the Map Viewer Open dialog to

```
matlabroot/toolbox/map/mapdemos
```

Import From Workspace

Images. Use the **Raster Data > Image** import dialog to select a **referencing matrix and 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.

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 `NaN`s 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`
	`Marker = 'x'`
	`MarkerEdgeColor =` <randomly generated value>
	`MarkerFaceColor = 'none'`

Line (line objects)	`Color =` <randomly generated value>
	`LineStyle = '-'`
	`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 and/or line width and/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 an M-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. In addition, the **Insert** menu allows you to insert axis labels and a title. 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`

Purpose

Display contours of constant map distortion

Syntax

```
mdistort
mdistort off
mdistort(parameter)
mdistort parameter
mdistort(parameter,levels)
mdistort(parameter,levels,gsize)
[h,ht] = 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,ht] = mdistort(...)` returns the handles to the line and text objects.

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.

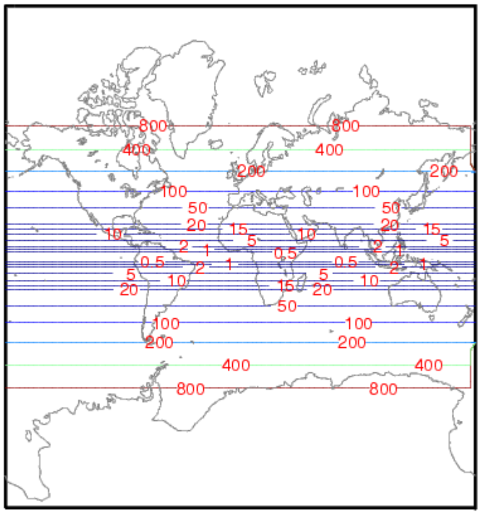
`mdistort` is not intended for use with UTM. Distortion is minimal within a given UTM zone. `mdistort` issues a warning if a UTM projection is encountered.

Examples

Example 1

Note the extreme area distortion of the Mercator projection. This makes it ill-suited for global displays.

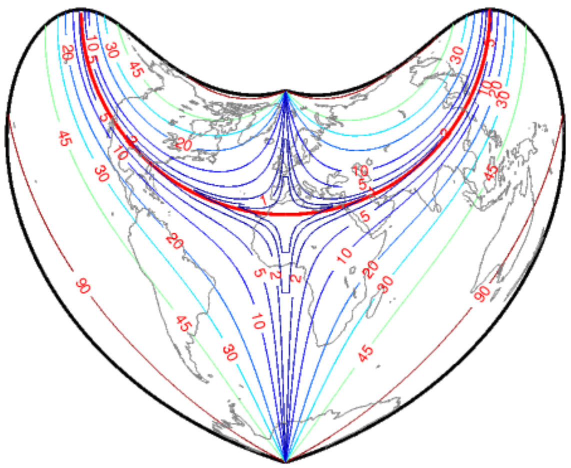
```
figure
axesm mercator
load coast
framem;plotm(lat, long,'color',.5*[1 1 1])
mdistort area
```



Example 2

The lines of zero distortion for the Bonne projection follow the central meridian and the standard parallel.

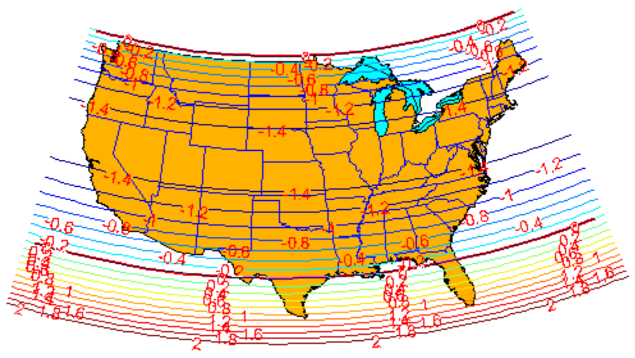
```
figure
axesm bonne
load coast
framem;plotm(lat, long,'color',.5*[1 1 1])
mdistort angles
parallelui
```



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:.2:2)
parallelui
```



Remarks

`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

Purpose

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 elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. 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" on page 10-2 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

Purpose

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 the 1-by-2 vector `ellipsoid`. 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`

Purpose

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`. Latitudes `phi1` and `phi2` are in radians, and `s` has the same units as the semimajor axis of the ellipsoid.

See Also

`meridianarc`

meshgrat

Purpose

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 `Z`. `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.

`[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`

Purpose

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. By default, shading is based on a light to the east (90°) 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. `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.

`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.

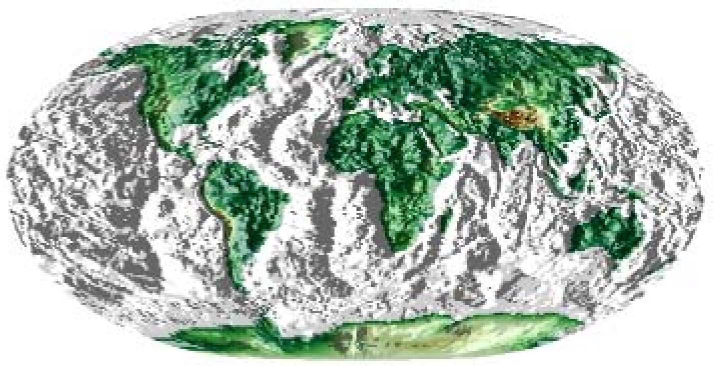
Remarks

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.

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.

```
load topo
[cmap,clim] = demcmap(topo,[],[1 1 1],[]);
axesm loximuth
meshlsrm(topo,topolegend,[],cmap,clim)
```



See Also

`meshgrat`, `meshm`, `pcolorm`, `surfacem`, `surflm`, `surflsrm`

Purpose

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` 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. 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).

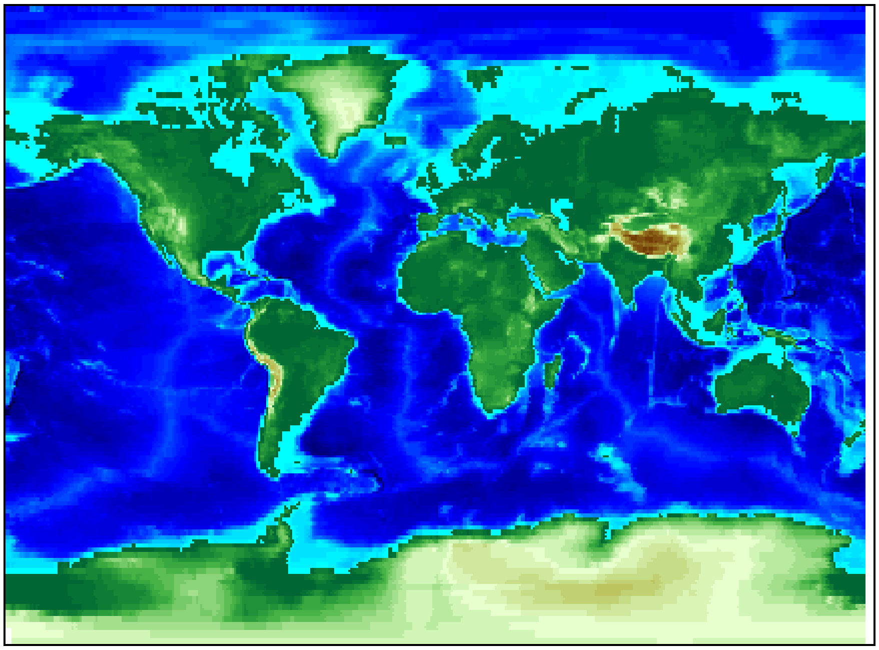
`meshm(Z, R, gratsize, alt)` displays the regular surface map at the altitude specified by `alt`. If `alt` is a scalar, then the grid is drawn in the z = `alt` plane. If `alt` is a matrix, then `size(alt)` must equal `gratsize`, and the graticule mesh is drawn at the altitudes specified by `alt`. If 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.

Example

```
load topo
axesm miller
meshm(topo,topolegend,[90 180])
demcmap(topo)
tightmap
```



See Also

`geoshow`, `mapshow`, `meshgrat`, `pcolorm`, `surfacem`, `surfm`

Purpose

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` demo 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`

Purpose

Semiminor axis of ellipse given semimajor axis and eccentricity

Syntax

```
semiminor = minaxis(semimajor,eccentricity)
semiminor = minaxis([semimajor,eccentricity])
```

Description

`semiminor = minaxis(semimajor,eccentricity)` returns the semiminor axis length corresponding to the input semimajor axis and eccentricity.

`semiminor = minaxis([semimajor,eccentricity])` allows the inputs to be packed into a single two-column input of the form `[semimajor, eccentricity]`.

The semiminor axis can be determined given both the semimajor axis and the eccentricity, the two elements of a standard Mapping Toolbox ellipsoid vector.

Examples

Using the default values for the Earth,

```
semiminor = minaxis(almanac('earth','ellipsoid'))
semiminor =
   6.3568e+03
```

See Also

`almanac`, `axes2ecc`, `majaxis`

Purpose

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`

Purpose

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

Purpose

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.

Example

```
% create a map
figure('color','w'); axesm('miller','grid','on'); tightmap;
mlabel on; plabel on
```

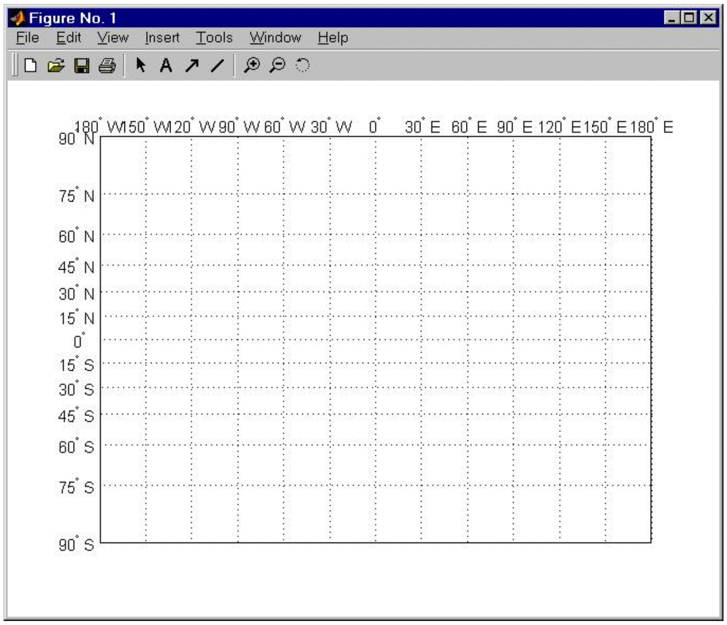


Figure No. 1
File Edit View Insert Tools Window Help
180° W 150° W 120° W 90° W 60° W 30° W 0° 30° E 60° E 90° E 120° E 150° E 180° E
90° N
75° N
60° N
45° N
30° N
15° N
0°
15° S
30° S
45° S
60° S
75° S
90° S

```
% Display longitude labels in the range of 0 to 360 degrees
mlabelzero22pi
```

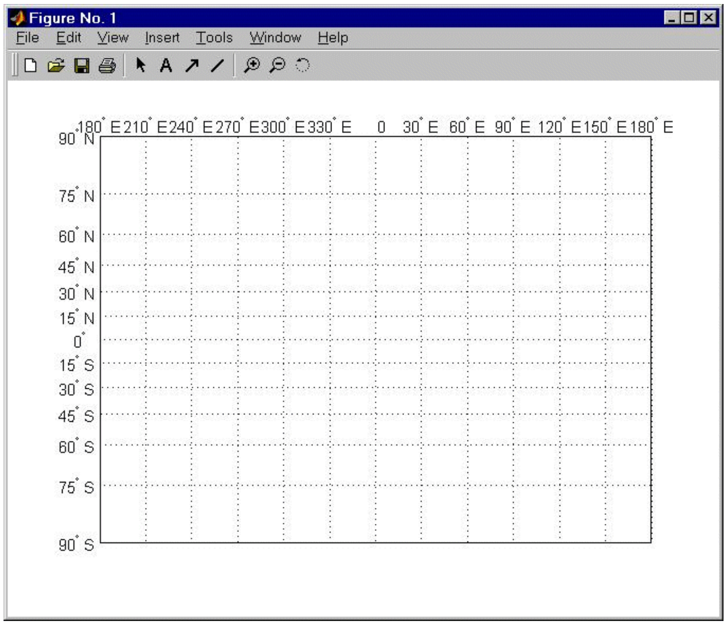


Figure No. 1
File Edit View Insert Tools Window Help
180° E 210° E 240° E 270° E 300° E 330° E 0 30° E 60° E 90° E 120° E 150° E 180° E
90° N
75° N
60° N
45° N
30° N
15° N
0°
15° S
30° S
45° S
60° S
75° S
90° S

See Also `mlabel`

n2ecc

Purpose Eccentricity of ellipse with given n-value

Syntax `eccentricity = n2ecc(n)`

Description `eccentricity = n2ecc(n)` returns the equivalent eccentricities for the input *n* parameters. If the input `n` is a two-column vector, only the second column is used. This allows two-element vectors to be used as rows of the input, because the form `[semimajor-axis, n]` is a complete representation of an ellipsoid (but is not the standard form for Mapping Toolbox ellipsoid vectors). In all other cases, all columns of the input are used.

Eccentricity and the parameter *n* are two methods of defining an ellipsoid. The definition of *n* is

(semimajor axis – semiminor axis)/(semimajor axis + semiminor axis)

Example

```
ecc = n2ecc(0.00167922039463)

ecc =
   0.08181919104285
```

This eccentricity is the default value for the Earth.

See Also `almanac`, `ecc2flat`, `majaxis`, `ecc2n`

Purpose

Determine names of valid graphics objects

Syntax

```
objects = namem
objects = namem(handles)
```

Description

`objects = namem` returns the object names for all objects on the current axes. The object name is defined as its tag, if the object `Tag` property is supplied. Otherwise, it is the object `Type`. Duplicate object names are removed from the output string matrix.

`objects = namem(handles)` returns the object names for the objects specified by the input `handles`.

The names returned are either set at object creation or defined by the user with the `tagm` function.

See Also

`clma`, `clmo`, `handlem`, `hidem`, `showm`, `tagm`

Purpose

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`

Purpose

Construct regular data grid of NaNs

Syntax

```
[Z,refvec] = nanm(latlim,lonlim,scale)
```

Description

`[Z,refvec] = nanm(latlim,lonlim,scale)` returns a regular data grid consisting entirely of `NaN`s 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.

Example

```
[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`

Purpose

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 `0`s 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" on page 10-11 in the *Mapping Toolbox User's Guide* for a more complete description of the use of this function.

Remarks

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.

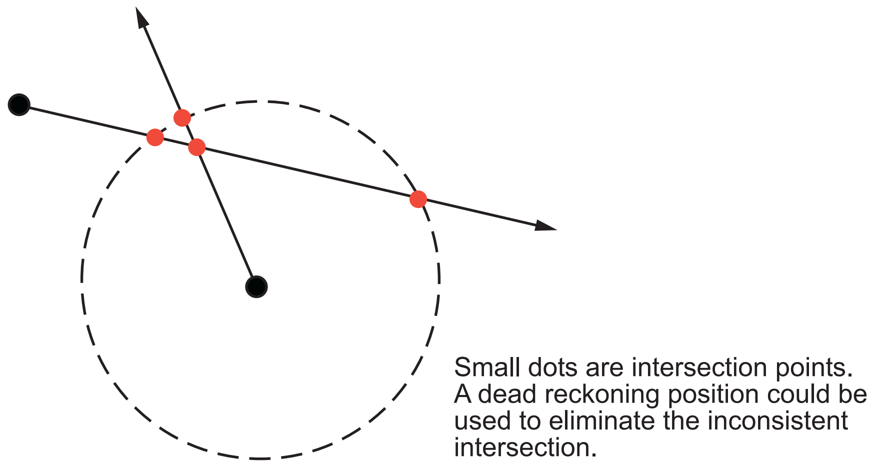
Examples

For a fully illustrated example of the application of this function, refer to the "Navigation" on page 10-11 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.

See Also

`crossfix`, `gcxgc`, `gcxsc`, `scxsc`, `rhxrh`, `polyxpoly`, `dreckon`, `gcwaypts`, `legs`, `track`

neworig

Purpose

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` 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.

`[Z,lat,lon] = neworig(Z0,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.

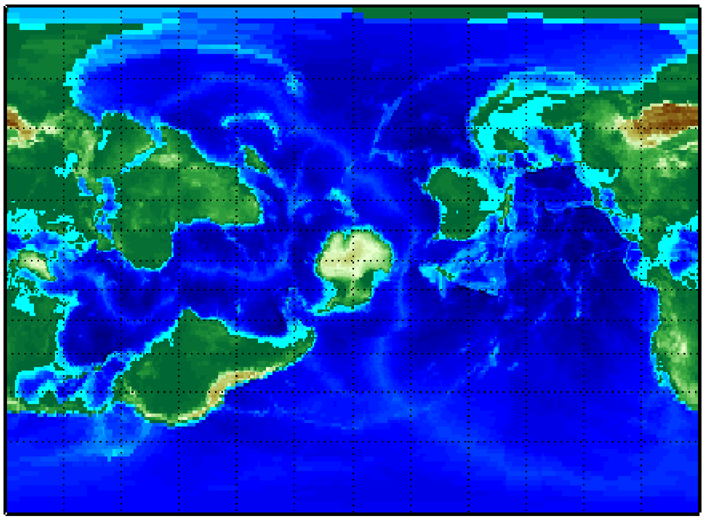
Limitations

neworig only supports data grids that cover the entire globe.

Example

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         0
[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`

Purpose

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 0°.

`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 30° beyond the true North Pole on the original origin's meridian is pulled up to become the *pole*, the point originally 30° above the origin is pulled down into the origin spot.

See Also

`neworig`, `org2pol`, `putpole`

northarrow

Purpose

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.

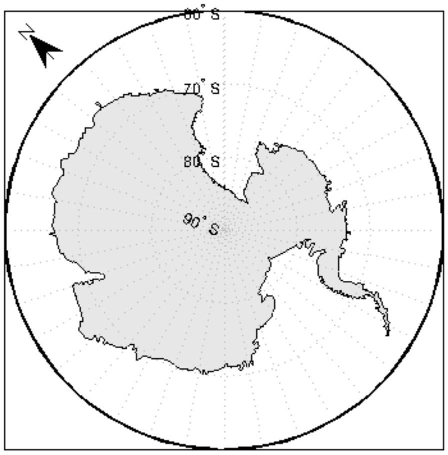
`northarrow` creates a north arrow symbol at the map origin on the displayed map. You can reposition the north arrow symbol by clicking and dragging its icon. Alternate clicking the icon creates an input dialog box that you can also use to change the location of the north arrow.

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.

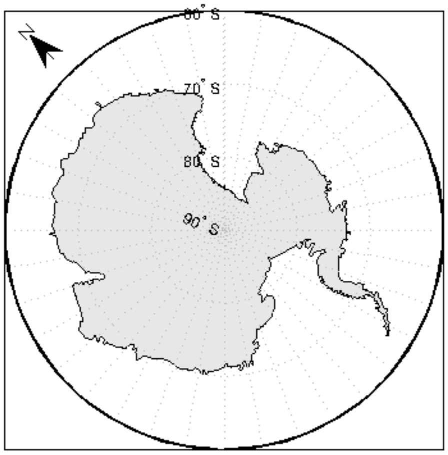
Examples

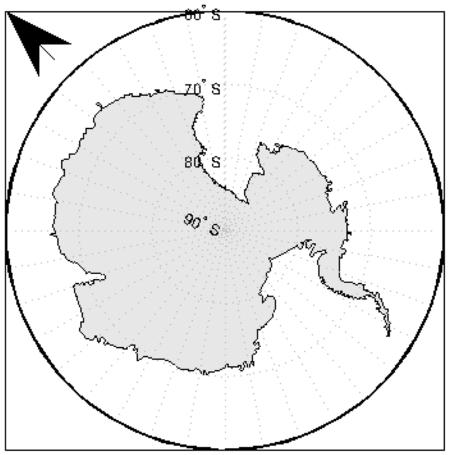
Create a map of the South Pole and then add the north arrow in the upper left of the map.

```
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);
```



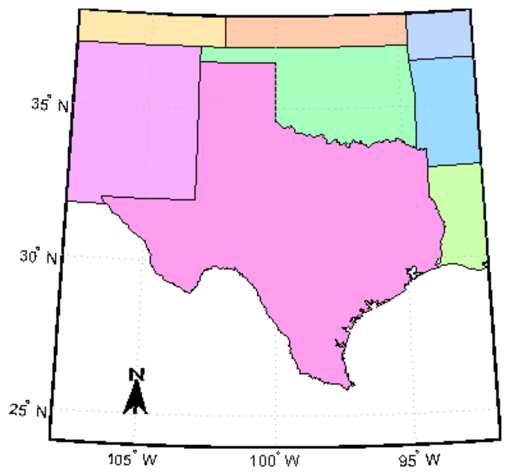
Right-click the north arrow icon to activate the input dialog box. Increase the size of the north arrow symbol by changing the `'ScaleRatio'` property.





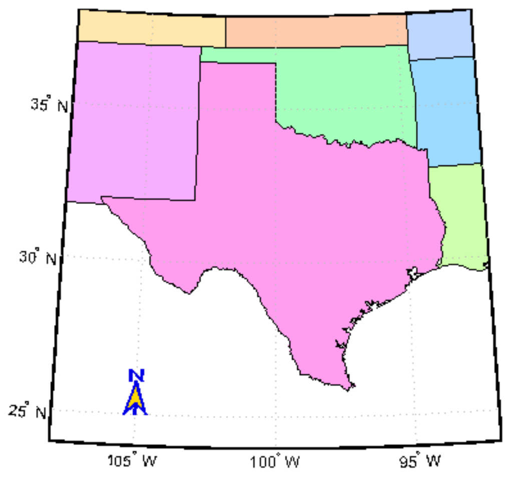
Create a map of Texas and add the north arrow in the lower left of the map.

```
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`

Purpose

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`

Purpose

Construct regular data grid of 1s

Syntax

```
[Z,refvec] = onem(latlim,lonlim,scale)
```

Description

`[Z,refvec] = onem(latlim,lonlim,scale)` returns a regular data grid consisting entirely of `1`s 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`

Purpose

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 30º down to the origin, and the North Pole is pulled down 30º. A little less obvious example is the following:

```
pole = org2pol([5 40 30])

pole =
   59.6245   80.0750   40.0000
```

See Also

neworig, putpole

Purpose

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 two-element vectors 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.

Remarks

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

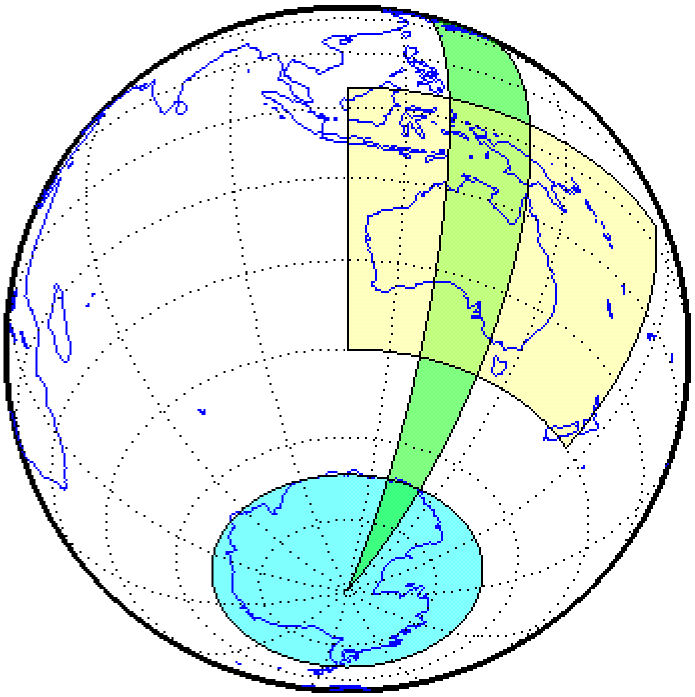
- `latlim(1) == -90` *or* `latlim(2) == 90`, so that only one part is needed—a polygon that follows a parallel clockwise around one of the poles.

- `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.

Example

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

% 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);
```



See Also `ingeoquad`, `intersectgeoquad`

Purpose

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. If `radius` is a string, then it is evaluated as an `almanac` body to determine the spherical radius. If numerical, it is the radius of the desired sphere in the same units as the surface distance. 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

`pagesetupdlg`, `axesscale`, `daspectm`

patchesm

Purpose

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.

Remarks

Differences between patchesm and patchm

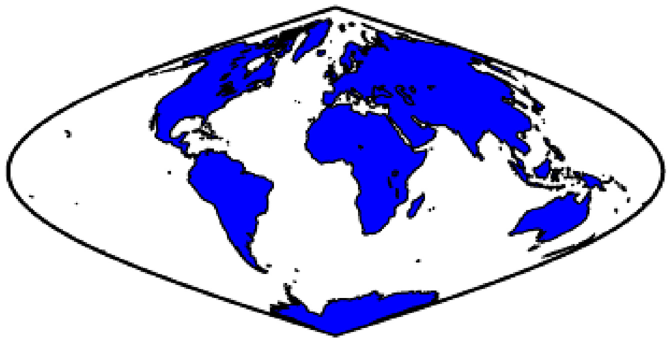
The `patchesm` function is very similar to the `patchm` function. The significant difference is that in `patchesm`, separate patches (delineated by `NaN`s in the inputs `lat` and `lon`) are separated and plotted as distinct patch objects on the current map axes. The advantage to this is that less memory 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.

Examples

```
load coast
axesm sinusoid; framem
h = patchesm(lat,long,'b');
```



```
length(h)

ans =
   238
```

See Also

`geoshow`, `fill3m`, `fillm`, `patchm`

patchm

Purpose

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`.

Remarks

How patchm Works

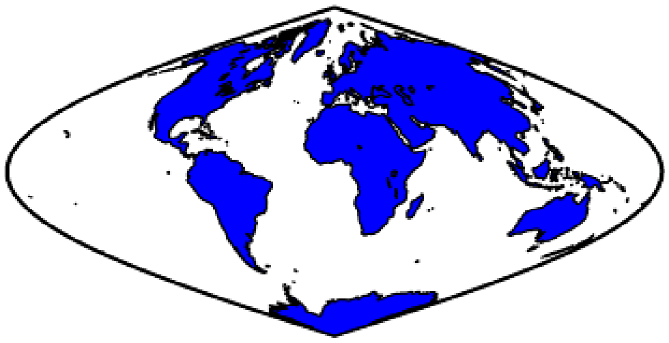
This Mapping Toolbox function is very similar to the standard MATLAB `patch` function. Like its analog, and unlike higher level functions such as `fillm` and `fill3m`, `patchm` adds patch objects to the current map axes regardless of hold state. Except for `XData`, `YData`, and `ZData`, all line properties 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.

Examples

```
load coast
axesm sinusoid; framem
h = patchm(lat,long,'b');
```



```
length(h)

ans =
     1
```

See Also

`patchesm`, `fill3m`, `fillm`

pcolorm

Purpose

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.

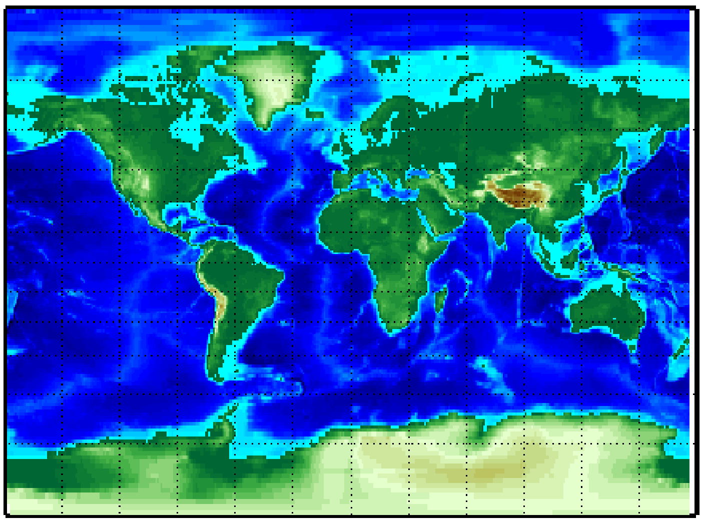
Remarks

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.

Example

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
```



See Also

geoshow, meshgrat, meshm, surfacem, surfm

Purpose

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 a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. `row` and `col` are vectors or arrays of matching size. The outputs `lat` and `lon` have the same size as `row` and `col`.

Example

```
% Find the lat and lon 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_lat, UL_lon] = pix2latlon(R, .5, .5)

UL_lat =
    88
UL_lon =
     0

[LR_lat, LR_lon] = pix2latlon(R, 90.5, 180.5)

LR_lat =
   268
LR_lon =
   360
```

See Also

`latlon2pix`, `makerefmat`, `pix2map`

Purpose

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 a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. `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.

Example

```
% Find the map coordinates for the pixel at (100,50).
R = worldfileread('concord_ortho_w.tfw');
[x,y] = pix2map(R,100,50)

x =
        2.070495000000000e+005
y =
        9.129005000000000e+005
```

See Also

`makerefmat`, `map2pix`, `pix2latlon`, `worldfileread`

pixcenters

Purpose

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 the 3-by-2 affine referencing matrix. `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 `height`-by-1 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.

Remarks

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.

Example

```
[Z,R] = arcgridread('MtWashington-ft.grd');
[x,y] = pixcenters(R, size(Z));
```

```
h = surf(x,y,Z); axis equal; colormap(demcmap(Z))
set(h,'EdgeColor','none')
xlabel('x (easting in meters)')
ylabel('y (northing in meters')
zlabel('elevation in feet')colormap(terrain)
```

See Also

`arcgridread`, `makerefmat`, `mapbbox`, `mapoutline`, `pix2map`, `worldfileread`

The help for `mapshow` provides an alternative version of the preceding example.

plabel

Purpose

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`

Purpose

Project 3-D lines and points on map axess

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 for a reference ellipsoid vector of `[1 0]` (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.

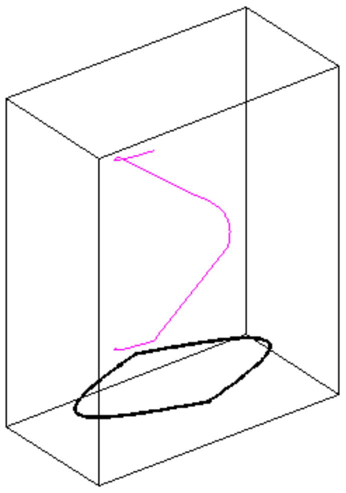
`h = plot3m(lat,lon,`*`PropertyName`*`,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB `line` function except for `XData`, `YData`, and `ZData`.

Remarks

`plot3m` is the mapping equivalent of the MATLAB `plot3` function.

Example

```
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')
```



See Also `linem`, `plot3`, `plotm`

Purpose

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.

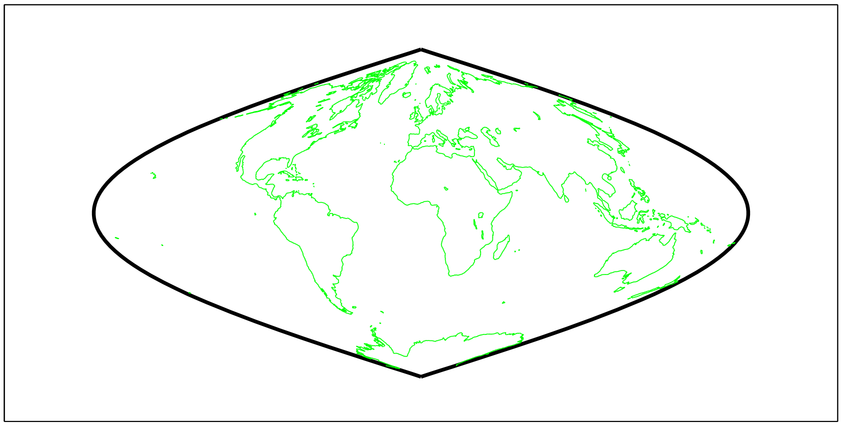
`h = plotm(lat,lon,`*`PropertyName`*`,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB `line` function except for `XData`, `YData`, and `ZData`.

`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.

Example

```
load coast
axesm sinusoid; framem
plotm(lat,long,'g')
```



See Also `linem`, `plot`, `plot3m`

Purpose

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.

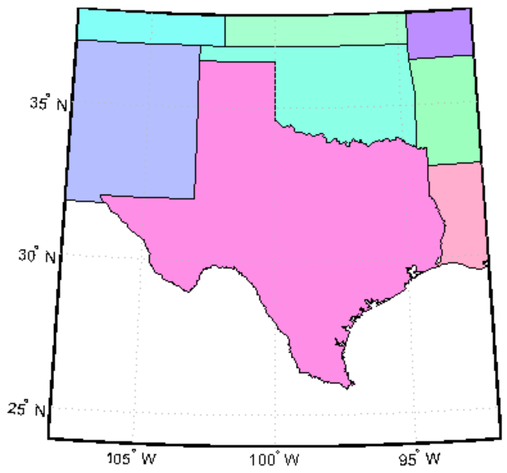
Remarks

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.

Example

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.

See Also

`demcmap`, `colormap`

Purpose

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`.

Example

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`

Purpose

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`.

Example

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`

Purpose

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. The contour vertices can be represented either in `NaN`-separated vector format or cell array format.

Individual contours in `x` and `y` are assumed to be external contours if their vertices are arranged in clockwise order; otherwise they are assumed to be internal contours. Use `poly2cw` or `poly2ccw`, if necessary, to achieve the desired vertex ordering.

Example

Display a rectangular region with two holes using a single patch object.

```
% External contour, rectangle, clockwise ordered.
x1 = [0 0 6 6 0];
y1 = [0 3 3 0 0];

% First hole contour, square, counterclockwise ordered.
x2 = [1 2 2 1 1];
y2 = [1 1 2 2 1];

% Second hole contour, triangle, counterclockwise ordered.
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 `polybool` for additional examples illustrating `poly2fv`.

See also

`ispolycw`, `patch`, `poly2cw`, `poly2ccw`, `polybool`

Purpose

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'`

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.

`polybool` assumes that individual contours whose vertices are clockwise ordered are external contours, and that contours whose vertices are counterclockwise ordered are internal contours. You can use `poly2cw` to convert a polygonal contour to clockwise ordering.

Limitations

Polygons processed via `polybool` are assumed to be in a Cartesian coordinate system. Therefore, geographic data that encompasses a pole cannot be used directly. Use `flatearthpoly` to convert polygons that contain a pole to Cartesian coordinates.

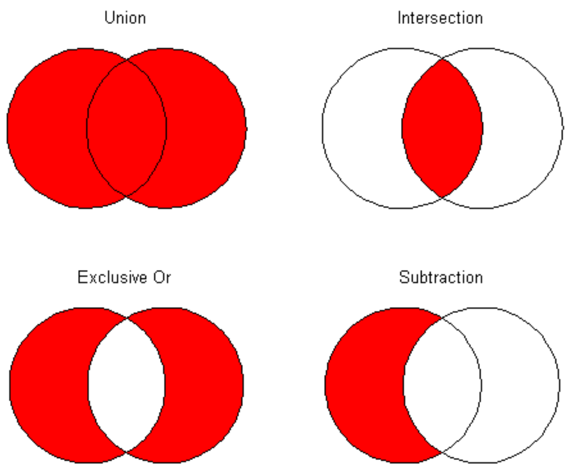
Examples

Example 1

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
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')
```

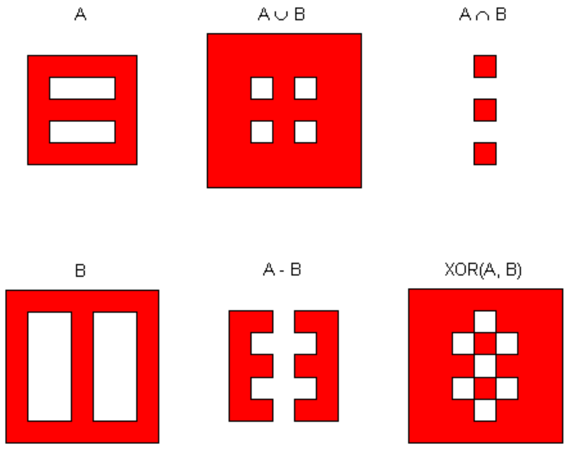


Example 2

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]};
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`

Purpose

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`

Purpose

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.

Remarks

A polygon may consist of an outer contour followed by holes separated with `NaN`s. In vector format, each vector may contain multiple faces separated by `NaN`s. There is no structural distinction between outer contours and holes in vector format.

Example

```
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
```

See Also

`polybool`, `polycut`, `polysplit`

polymerge

Purpose

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'`).

Example

```
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`

Purpose 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.

Example

```
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`

Purpose

Intersection points for lines or polygon edges

Syntax

```
[xi,yi] = polyxpoly(x1,y1,x2,y2)
[xi,yi] = polyxpoly(...,'unique')
[xi,yi,ii] = polyxpoly(...)
```

Description

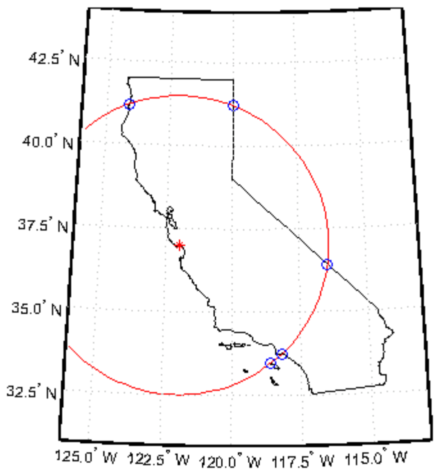
`[xi,yi] = polyxpoly(x1,y1,x2,y2)` returns the intersection points of two sets of lines and/or polygons.

`[xi,yi] = polyxpoly(...,'unique')` returns only unique intersections.

`[xi,yi,ii] = polyxpoly(...)` also returns a two-column index of line segment numbers corresponding to the intersection points.

Example

```
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')

[lat, lon] = reducem(california.Lat', california.Lon');
[loni, lati] = polyxpoly(lon, lat, lonc, latc);
plotm(lati, loni, 'bo')
```



See Also `crossfix`, `gcxgc`, `gcxsc`, `navfix`, `rhxrh`, `scxsc`

Purpose

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.

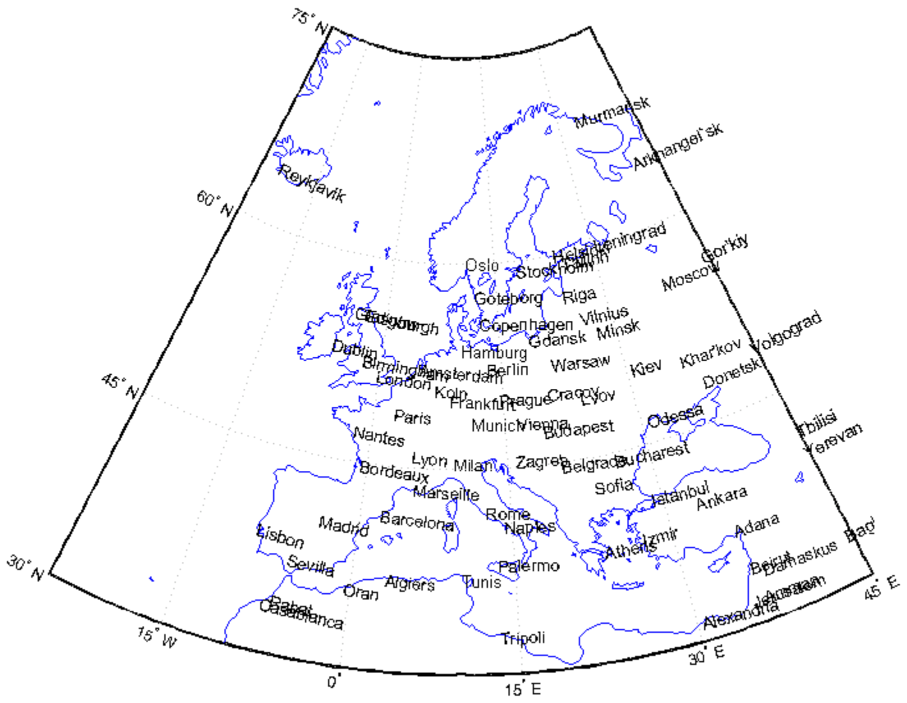
Remarks

`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.

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.

See Also `pagesetupdlg`, `paperscale`, `axesscale`

Purpose

Project displayed map graphics object

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!

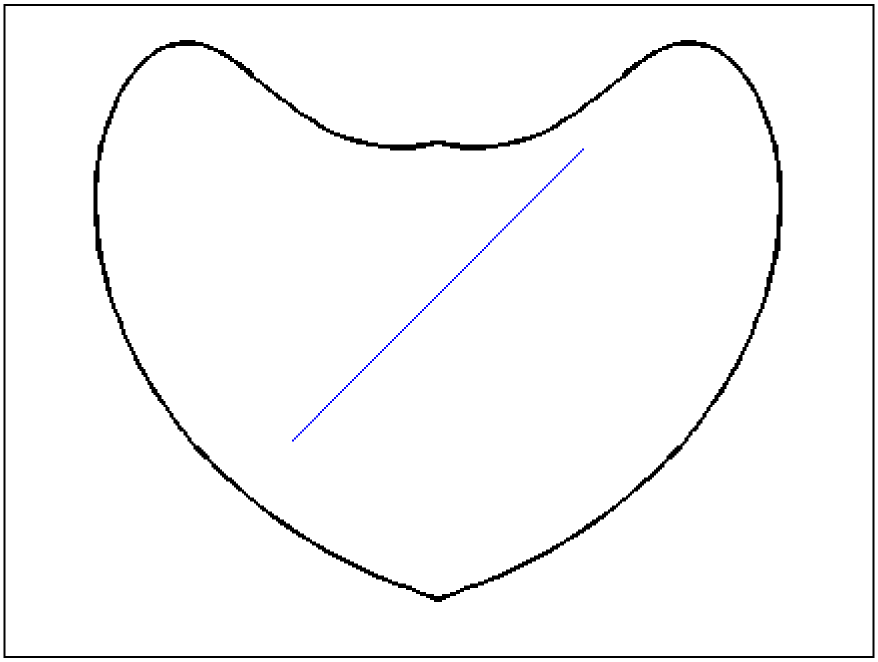
`project(h,'xy')` specifies that the `XData` of the unprojected objects corresponds to longitudes and the `YData` to latitudes. This is the default assumption.

`project(h,'yx')` specifies that the `XData` of the unprojected objects corresponds to latitudes and the `YData` to longitudes.

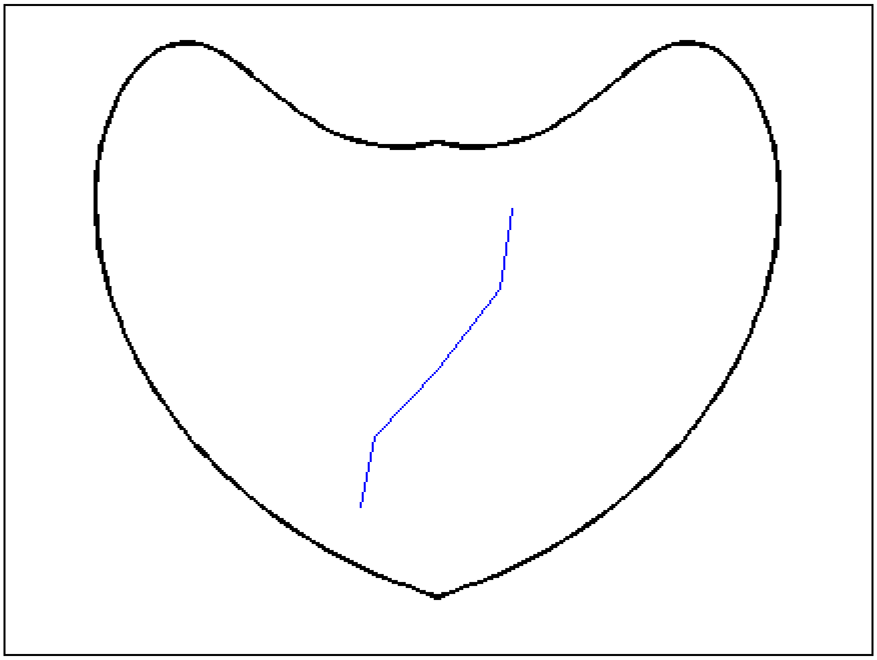
Example

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.

See Also

`linem`, `patchm`, `surfacem`, `textm`

projfwd

Purpose

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 `projfwd`, see the reference page for `projlist`.

Example

Overlay the boundary of Massachusetts on an 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);
```

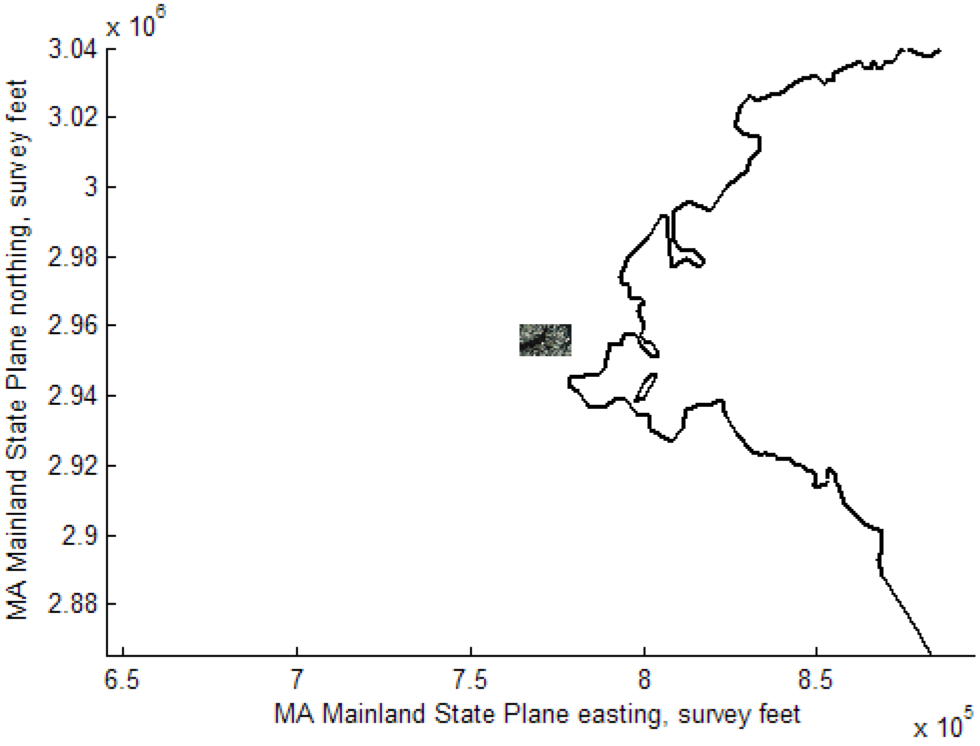
Read and display the `'boston.tif'` orthophoto image:

```
[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.


x 10^6
3.04
3.02
3
2.98
2.96
2.94
2.92
2.9
2.88
MA Mainland State Plane northing, survey feet
6.5
7
7.5
8
8.5
MA Mainland State Plane easting, survey feet
x 10^5

See Also `geotiffinfo`, `mfwdtran`, `minvtran`, `projinv`, `projlist`

Purpose

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`.

Example

Display Boston Orthophoto on a Mercator projection

1 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');
```

2 Convert the road coordinates to the projection's length unit. As shown by the `UOMLength` field of the projection structure, the units of length in the projected coordinate system is `US Survey Feet`. Coordinates in the roads shapefile are in meters:

```
proj.UOMLength

ans =
US survey foot

x = [roads.X] * unitsratio('survey feet','meter');
y = [roads.Y] * unitsratio('survey feet','meter');

% Now convert the scaled coordinates of the roads
% to latitude and longitude.
[roadsLat, roadsLon] = projinv(proj, x, y);
```

3 Read the `boston_ovr.jpg` image and worldfile:

```
RGB = imread('boston_ovr.jpg');
R = worldfileread(getworldfilename('boston_ovr.jpg'));
```

4 Read state boundary vectors for Massachusetts from the `usastatehi` shapefile using a selector to eliminate other states:

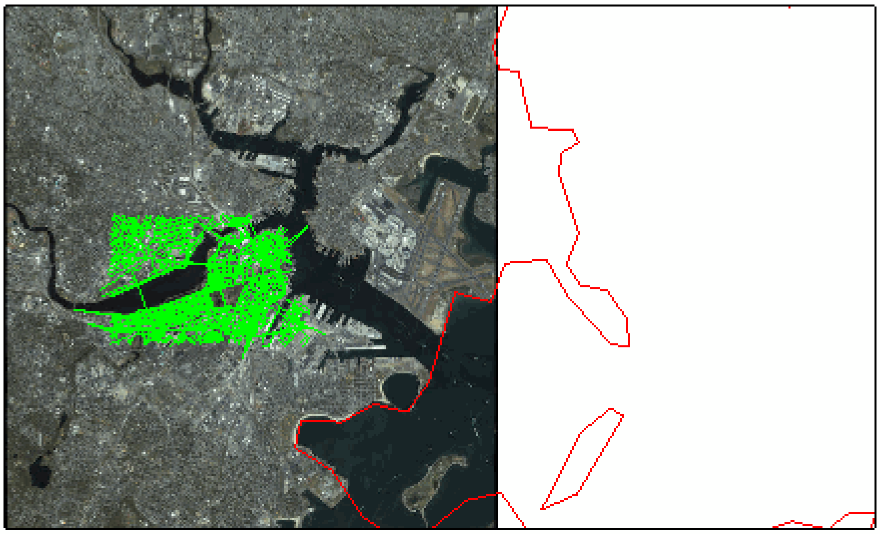
```
S = shaperead('usastatehi', 'UseGeoCoords', true, ...
    'Selector',{@(name) strcmpi(name,'Massachusetts'), 'Name'});
```

5 Open a figure with a Mercator projection and display the state boundary, image, and roads:

```
figure
axesm('mercator')

geoshow(S.Lat, S.Lon, 'Color','red')
geoshow(RGB, R)
geoshow(roadsLat, roadsLon, 'Color', 'green')
```

6 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
```



boston_ovr.jpg image copyright © GeoEye, all rights reserved.

See Also

`geotiffinfo`, `mfwdtran`, `minvtran`, `projfwd`, `projlist`

Purpose

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

Remarks

`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.

Examples

```
s=projlist

s =
1x19 struct array with fields:
    Name
    MapProjection

s=projlist('geotiff2mstruct')

s =
1x19 struct array with fields:
    GeoTIFF
    MapProjection
```

See Also

`geotiff2mstruct`, `projfwd`, `projinv`, `maplist`, `maps`

Purpose

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'`.

Remarks

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.

Examples

Pull the North Pole down the 0° meridian by 30° to 60°N. 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
```

See Also

`neworig`, `org2pol`

Purpose

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`*.

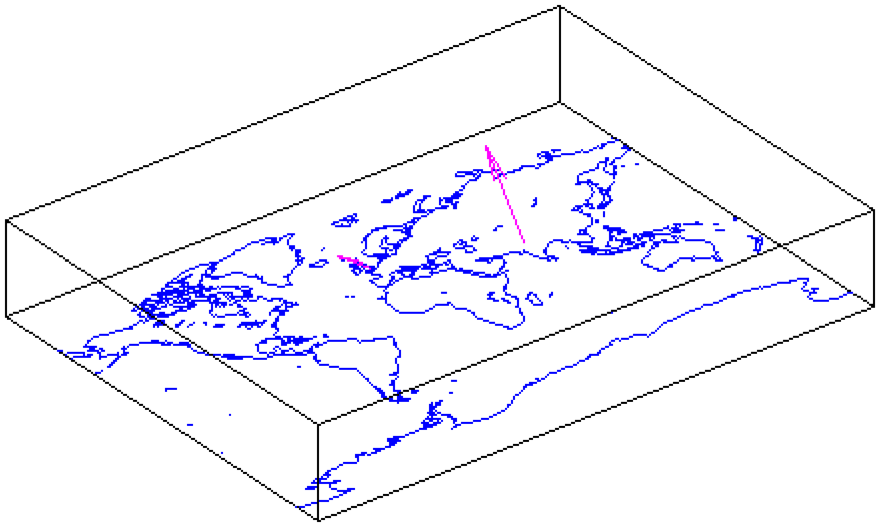
`h = quiver3m(lat,lon,alt,u,v,w,scale)`, `h = quiver3m(lat,lon,alt,u,v,w,linespec,scale)` and `h = quiver3m(lat,lon,alt,u,v,w,linespec,scale,'filled')` alter the automatically calculated vector lengths by multiplying them by the scalar value `scale`. For example, if `scale` is 2, the displayed vectors 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

Purpose

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`).

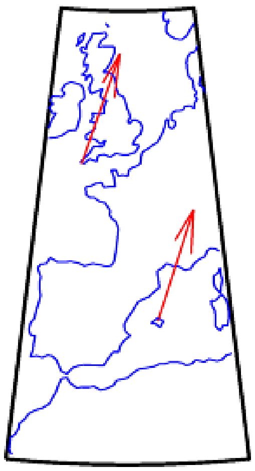
`h = quiverm(lat,lon,u,v,`*`linespec`*`,'filled')` results in the filling in of any symbols specified by *`linespec`*.

`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`).

Example

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, rad2nm, rad2sm

Purpose

Convert distance from radians to kilometers, nautical miles, or statute miles

Syntax

```
km = rad2km(rad)
nm = rad2nm(rad)
sm = rad2sm(rad)
km = rad2km(rad,radius)
nm = rad2nm(rad,radius)
sm = rad2sm(rad,radius)
km = rad2km(rad,sphere)
nm = rad2nm(rad,sphere)
sm = rad2sm(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.

`nm = rad2nm(rad)` and `sm = rad2sm(rad)` work identically, except that the output units are nautical miles and statute miles, respectively.

`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.

For `nm = rad2nm(rad,radius)` and `sm = rad2sm(rad,radius)`, make sure your input radius is in the appropriate units.

`km = rad2km(rad,`*`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.

`nm = rad2nm(rad,`*`sphere`*`)` and `sm = rad2sm(rad,`*`sphere`*`)` work identically, except that the output units are nautical miles and statute miles, respectively.

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`, `degtorad`, `radtodeg`, `deg2km`, `km2deg`, `km2nm`, `km2sm`, `deg2nm`, `nm2rad`, `nm2km`, `nm2sm`, `deg2sm`, `sm2rad`, `sm2km`, `sm2nm`

Purpose

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`

Purpose

Radii of curvature of ellipsoid

Syntax

```
r = rcurve(ellipsoid,lat)
r = rcurve('parallel',ellipsoid,lat)
r = rcurve(ellipsoid,lat,units)
r = rcurve('meridian',ellipsoid,lat,units)
r = rcurve('transverse',ellipsoid,lat,units)
```

Description

`r = rcurve(ellipsoid,lat)` and `r = rcurve('parallel',ellipsoid,lat)` return the parallel radius of curvature at the latitude `lat` for a given elliptical definition, where `ellipsoid` is a two-element ellipsoid vector. This is the radius of the small circle encompassing the ellipsoid at the given latitude. The radius is a distance in units consistent with the semimajor axis, the first element of `ellipsoid`.

`r = rcurve(ellipsoid,lat,`*`units`*`)` specifies the units of the input `lat`, where *`units`* is any valid angle units string. The default is `'degrees'`.

`r = rcurve('meridian',ellipsoid,lat,`*`units`*`)` returns the meridianal radius, which is the radius of curvature at the latitude `lat` for the ellipse described by a meridian on the ellipsoid.

`r = rcurve('transverse',ellipsoid,lat,`*`units`*`)` returns the transverse radius, which is the radius of a curve described by the intersection of the ellipsoid with a plane normal to the surface of the ellipsoid at the latitude `lat`.

Examples

The radii of curvature of the default ellipsoid at 45º, in kilometers:

```
r = rcurve('transverse',almanac('earth','ellipsoid','km'),...
            45,'degrees')

r =
   6.3888e+03

r = rcurve('meridian',almanac('earth','ellipsoid','km'),...
            45,'degrees')

r =
   6.3674e+03

r = rcurve('parallel',almanac('earth','ellipsoid','km'),...
            45,'degrees')

r =
   4.5024e+03
```

See Also `rsphere`

Purpose

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.

Remarks

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, readmtx, textread, spcread, dlmread

Purpose

Read Fifth Fundamental Catalog of Stars

Syntax

```
struc = readfk5(filename)
struc = readfk5(filename,struc)
```

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.

Remarks

Positions are given in terms of right ascension and declination. Chapter 8, "Using Map Projections and Coordinate Systems" in 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.

Note For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

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'
```

References

See references `[5]` and `[6]` in the Bibliography located at the end of this chapter.

See Also

`dms2degrees`, `scatterm`

Purpose

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.

Remarks

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`, `textread`, `spcread`, `dlmread`

reckon

Purpose

Point at specified azimuth, range on sphere or ellipsoid

Syntax

```
[latout,lonout] = reckon(lat,lon,rng,az)
[latout,lonout] = reckon(lat,lon,rng,az,units)
[latout,lonout] = reckon(lat,lon,rng,az,ellipsoid)
[latout,lonout] = reckon(lat,lon,rng,az,ellipsoid,units)
[latout,lonout] = reckon(track,...)
```

Description

`[latout,lonout] = reckon(lat,lon,rng,az)`, for scalar inputs, calculates a position (`latout,lonout`) at a given range `rng` and azimuth `az` along a great circle from a starting point defined by `lat` and `lon`. The range is in degrees of arc length on a sphere, `lat` and `lon` are in degrees, and the input azimuth is also in degrees, measured clockwise from due north. `reckon` calculates multiple positions when given four non-scalar inputs 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,rng,az,`*`units`*`)`, where *`units`* is any valid angle units string, specifies the angular units of the inputs and outputs, including `rng`. The default value is `'degrees'`.

`[latout,lonout] = reckon(lat,lon,rng,az,ellipsoid)` calculates positions along a geodesic on an ellipsoid, as specified by the two-element vector `ellipsoid`. The range, `rng`, is in linear distance units matching the units of the semimajor axis of the ellipsoid (the first element of `ellipsoid`).

`[latout,lonout] = reckon(lat,lon,rng,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

Purpose

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.

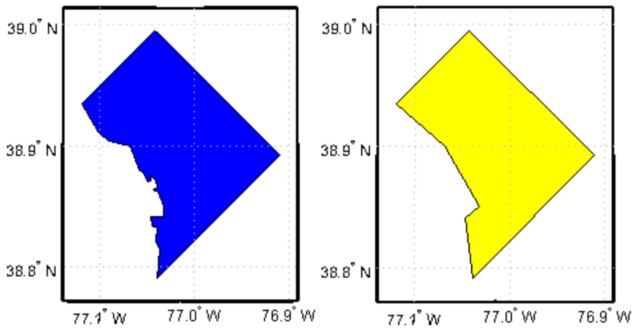
`[latout,lonout,cerr,tol] = reducem(...)` also returns the tolerance used in the reduction, which is useful when the tolerance is computed automatically.

Example

Compare the original and reduced outlines of the District of Columbia from the `usastatehi` demo 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')
```



Remarks

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`	Interpolate vector data to a specified data separation
`resizem`	Resize a data grid

Purpose

Convert referencing matrix to referencing vector

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.

Example

```
% 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`, `refvec2mat`

refvec2mat

Purpose

Convert referencing vector to referencing matrix

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 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.

Example

```
% Convert the geoid referencing vector to a referencing matrix
 load geoid;
R = refvec2mat(geoidlegend, size(geoid));
```

See Also

`makerefmat`, `refmat2vec`

Purpose

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
```

Purpose

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 with a referencing vector or matrix, `R1`, and returns a new grid and its referencing vector or matrix, `R`. `R1` 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] * 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. If `R1` is a referencing vector, then `R` is a referencing vector. Likewise, if `R1` is a referencing matrix, then `R` is a referencing matrix. If `R1` is a referencing vector, then `scale` must be a scalar resizing factor; the form `[numrows numcols]` is supported only for referencing matrices.

`[...]  = 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'`.

Example

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` (MATLAB function)

restack

Purpose

Restack objects within map axes

Syntax

`restack(h,position)`

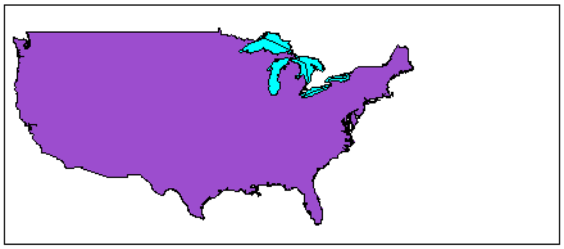
Description

`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'`.

Examples

Restack the great lakes to lie on top of `conus`:

```
figure; axesm miller
load conus
h = geoshow(gtlakelat, gtlakelon,...
    'DisplayType', 'polygon', 'FaceColor', 'cyan');
geoshow(uslat, uslon,...
    'DisplayType', 'polygon', 'FaceColor', [0.6 0.3 0.8])
% The great lakes were plotted first but need to be on top
% Cast handle to great lakes object to double in call to RESTACK
restack(double(h),'top')
```



Remarks

This function is the command line equivalent of the stacking buttons in the `mobjects` graphical user interface. The stacking order is the order of the children of the axes.

See Also

`mobjects`

Purpose

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 `NaN`s 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`

Purpose

Construct cell array of workspace variables for `mlayers` tool

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

Purpose

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`

Purpose

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.

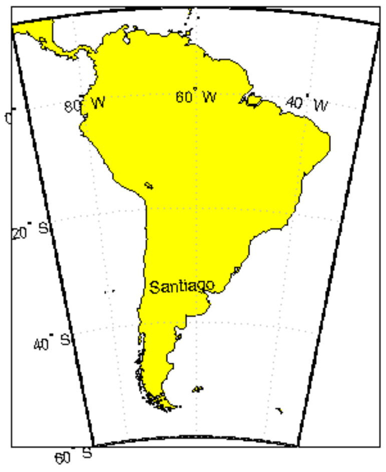
`rotatetext(objects)` rotates the selected objects. `objects` can be a name string recognized by `handlem` or a vector of handles to displayed text objects.

`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 = strmatch('Santiago',{cities(:).Name});
h=textm(cities(Santiago).Lat, cities(Santiago).Lon, ...
     'Santiago');
rotatetext(h)
```



Remarks You can rotate meridian and parallel labels automatically by setting the map axes `LabelRotation` property to `'on'`.

See Also `vfwdtran`, `vinvtran`

roundn

Purpose

Round to multiple of 10^n

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`

Purpose

Radii of auxiliary spheres

Syntax

```
r = rsphere('biaxial',ellipsoid)
r = rsphere('biaxial',ellipsoid,method)
r = rsphere('triaxial',ellipsoid)
r = rsphere('eqavol',ellipsoid)
r = rsphere('authalic',ellipsoid)
r = rsphere('rectifying',ellipsoid)
r = rsphere('curve',ellipsoid,lat,method,units)
r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid)
r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid,units)
```

Description

`r = rsphere('biaxial',ellipsoid)` calculates the radius of a biaxial auxiliary sphere for the ellipsoid specified by the two-element ellipsoid vector `ellipsoid`. The output, `r`, is the radius of this sphere in units consistent with the semimajor axis, that is, the first element of `ellipsoid`. The biaxial radius is calculated by averaging the semimajor and semiminor axes of the ellipsoid, giving each equal weight.

`r = rsphere('biaxial',ellipsoid,`*`method`*`)` specifies the averaging method. If the string *`method`* is `'mean'` (the default), an arithmetic mean is used. If *`method`* is `'norm'`, a geometric mean is used.

`r = rsphere('triaxial',ellipsoid)` results in a triaxial radius, which is calculated by averaging the ellipsoidal axes while giving double weight to the semimajor axis to reflect its role in two of the ellipsoid's three dimensions.

`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,`*`method`*`,`*`units`*`)` returns a radius that is the result of averaging the meridional and transverse radii of curvature at the specified latitude, `lat`. The units of the input `lat` can be specified by the valid angle units string *`units`*. The default units are `'degrees'`, the default averaging method is `'mean'`, and the default latitude is 45°.

`r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid)` and `r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid,`*`units`*`)` calculate a radius using Euler's Theorem. This calculation requires the specification of an arc that is defined by its endpoints, (`lat1`,`lon1`) and (`lat2`,`lon2`).

The `rsphere` function calculates the radii of auxiliary spheres for the ellipsoid. An auxiliary sphere is a sphere that shares certain desired characteristics with the ellipsoid.

Examples

Different criteria result in different spheres:

```
r = rsphere('biaxial',almanac('earth','ellipsoid','km'))

r =
   6.3674e+03

r = rsphere('triaxial',almanac('earth','ellipsoid','km'))

r =
   6.3710e+03

r = rsphere('curve',almanac('earth','ellipsoid','km'))

r =
   6.3781e+03
```

See Also

`rcurve`

Purpose

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 50th 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.

Remarks

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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

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
```

See Also

`tbase`, `gtopo30`, `egm96geoid`

scaleruler

Purpose

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

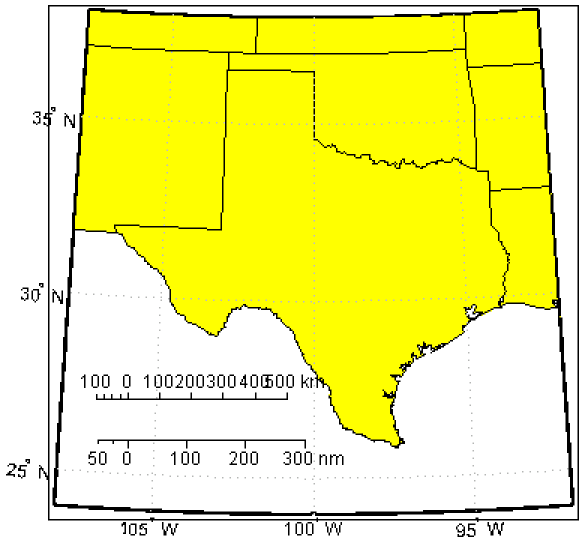
Cartographers often add graphic elements to the map to indicate its scale. Perhaps the most commonly used is the graphic scale, a ruler-like object that shows distances on the ground at the correct size for the projection.

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'),'YLoc',.5)
scaleruler('units','nm')
setm(handlem('scaleruler2'), ...
    'YLoc', .48, ...
    'MajorTick', 0:100:300,...
    'MinorTick', 0:25:50, ...
    'TickDir', 'down', ...
    'MajorTickLength', km2nm(25),...
    'MinorTickLength', km2nm(12.5))
```



Remarks

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`.

Modifying the properties of the graphic scale results in the replacement of the original object (dragging a scaleruler, however, does not replace it). For this reason, handles to the graphic scale object will change.

Use `handlem('scaleruler')` to get a list of the current handles to all graphic scale objects. Use `handlem('scalerulerN')`, where `N` is an integer, to get the handle to a particular graphic scale. Use `namem` to see the names of existing graphic scale objects. The name of a graphic scale object is also stored in the read-only `'Children'` property, which is accessed using `getm`.

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.

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`

`almanac body or scalar`

Planetary radius — The radius property controls the scaling between angular and surface distances. If `radius` is a string, then it is evaluated as an `almanac` body to determine the spherical radius. If numerical, it is the 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.

See Also `distance`, `surfdist`, `axesscale`, `paperscale`, `distortcalc`, `mdistort`

Purpose

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')
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'`.

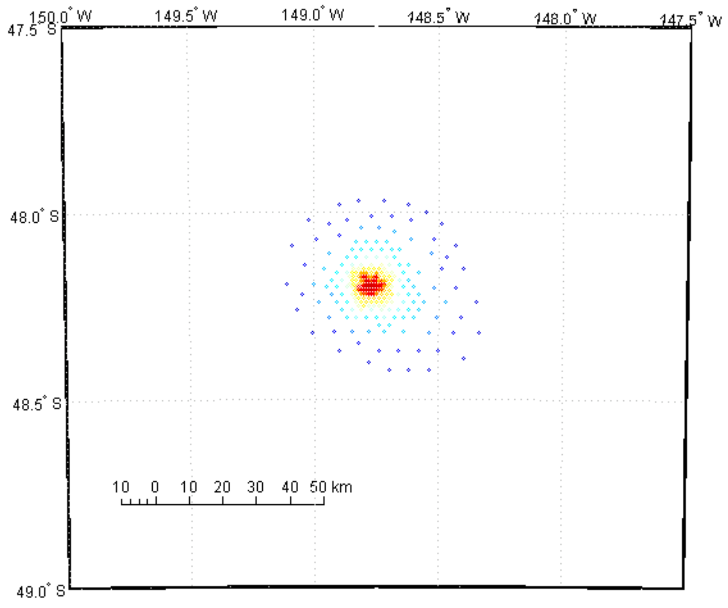
`scatterm(...,'filled')` fills the markers.

`h = scatterm(...)` returns handles of patches created.

Examples

Plot the `seamount` MATLAB demo 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

Purpose

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,geoid)
[lat,lon] = scircle1(lat0,lon0,rad,units)
[lat,lon] = scircle1(lat0,lon0,rad,az,units)
[lat,lon] = scircle1(lat0,lon0,rad,az,geoid,units)
[lat,lon] = scircle1(lat0,lon0,rad,az,geoid,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,geoid)` computes small circles on the ellipsoid defined by the input `geoid`, rather than by assuming a sphere. The `geoid` vector is 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 `geoid = []`, 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,geoid,`*`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,geoid,*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 90° 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 (0°,0°) with a radius of 10°.

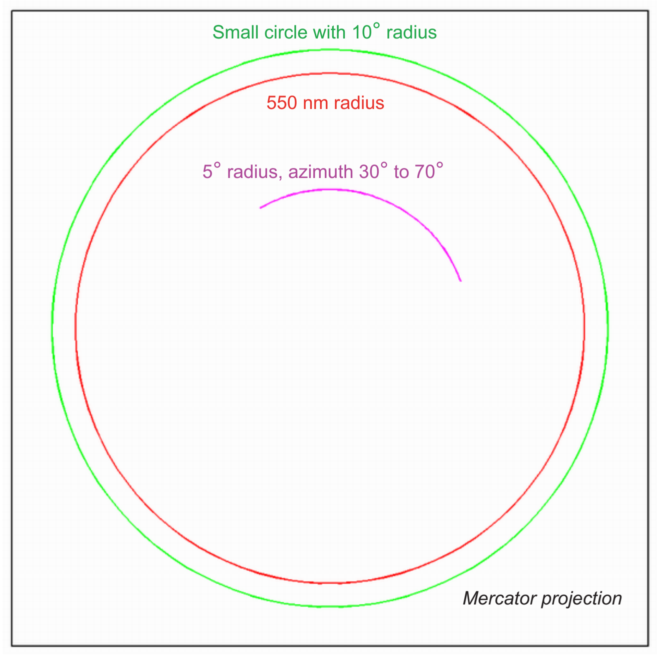
```
axesm('mercator','MapLatLimit',[30 -30],'MapLonLimit',[-30 30]);
[latc,longc] = scircle1(0,0,10);
plotm(latc,longc,'g')
```

If the desired radius is known in some nonangular distance unit, use the radius returned by the `almanac` function as the ellipsoid to set the range units. (Use an empty azimuth entry to indicate a full circle.)

```
earthradius = almanac('earth','radius','nm');
[latc,longc] = scircle1(0,0,550,[],earthradius);
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`

Purpose

Small circles from center and perimeter

Syntax

```
[lat,lon] = scircle2(lat1,lon1,lat2,lon2)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,geoid)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,units)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,geoid,units)
[lat,lon] = scircle2(lat1,lon1,lat2,lon2,geoid,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,geoid)` computes the small circle on the ellipsoid defined by the input `geoid`, rather than by assuming a sphere. The `geoid` vector is of the form `[semimajor axis, eccentricity]`. If `geoid = []`, a sphere is assumed.

`[lat,lon] = scircle2(lat1,lon1,lat2,lon2,`*`units`*`)` and `[lat,lon] = scircle2(lat1,lon1,lat2,lon2,geoid,`*`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,geoid,`*`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`.

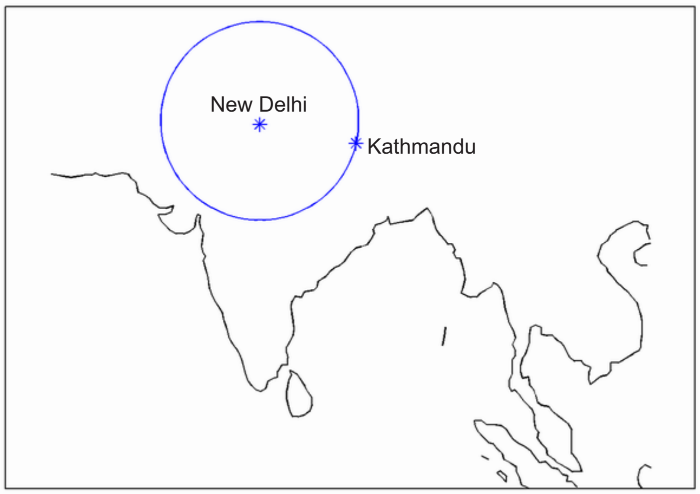
Definitions

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`

Purpose

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`

Purpose

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,newlong] = scxsc(-10,-170,20,3,179,15)

newlat =
   -8.8368    9.8526
newlong =
   169.7578 -167.5637
```

Note that in this example, the two small circles cross the date line.

Remarks

Great circles are a subset of small circles—a great circle is just a small circle with a radius of 90º. 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 90º (or its equivalent in radians).

See Also

`gc2sc`, `gcxgc`, `gcxsc`, `rhxrh`, `crossfix`, `polyxpoly`

Purpose

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`). `NaN`s 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.

Remarks

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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

Example

```
[Z, R] = sdtsdemread('9129CATD.ddf');
mapshow(Z,R,'DisplayType','contour')
```

See Also

`arcgridread`, `makerefmat`, `mapshow`, `sdtsinfo`

sdtsinfo

Purpose

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

Example

```
info = sdtsinfo('9129CATD.DDF');
```

See Also

`sdtsdemread`, `makerefmat`

Purpose

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.

Remarks

Sector control windows are superimposed at the same location. A valid map axes must exist prior to running this function.

See also

`scircleg`, `trackg`

Purpose

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` 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. All input and output angles are in units of 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`

Purpose

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.

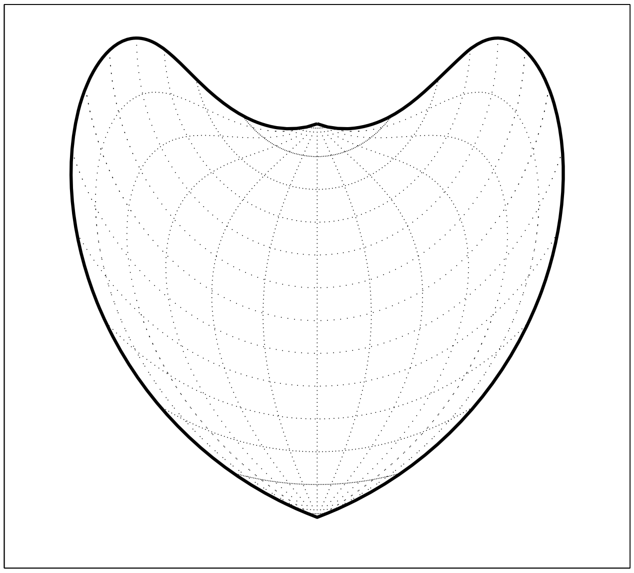
`setm(surfhndl,'MeshGrat',npts,alt)` alters the mesh graticule of projected surface objects displayed using the `meshm` function. In this case, the two-element vector `npts` specifies the graticule size in the manner described under `meshm`. The altitude can be a scalar or a matrix with a size corresponding to `npts`.

Examples

Display a map axes and alter it:

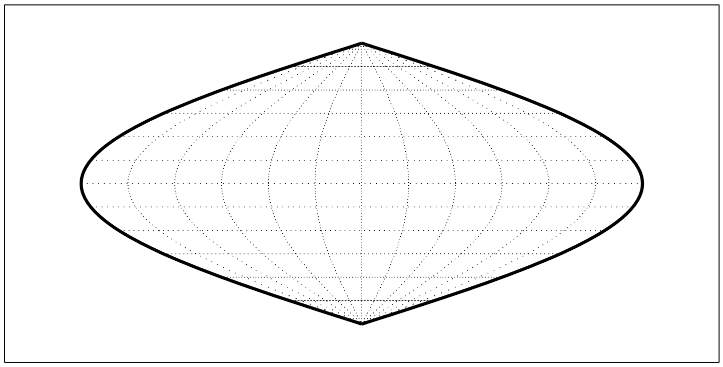
```
axesm('bonne','Frame','on','Grid','on')
```

The standard Bonne projection has a standard parallel at 30ºN.



Setting this standard parallel to 0° results in a Sinusoidal projection:

```
setm(gca,'MapParallels',0)
```



See Also

`axesm`, `getm`

Purpose

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` 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. 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`

Purpose

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 (90º azimuth) at an elevation of 45º.

`[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`.

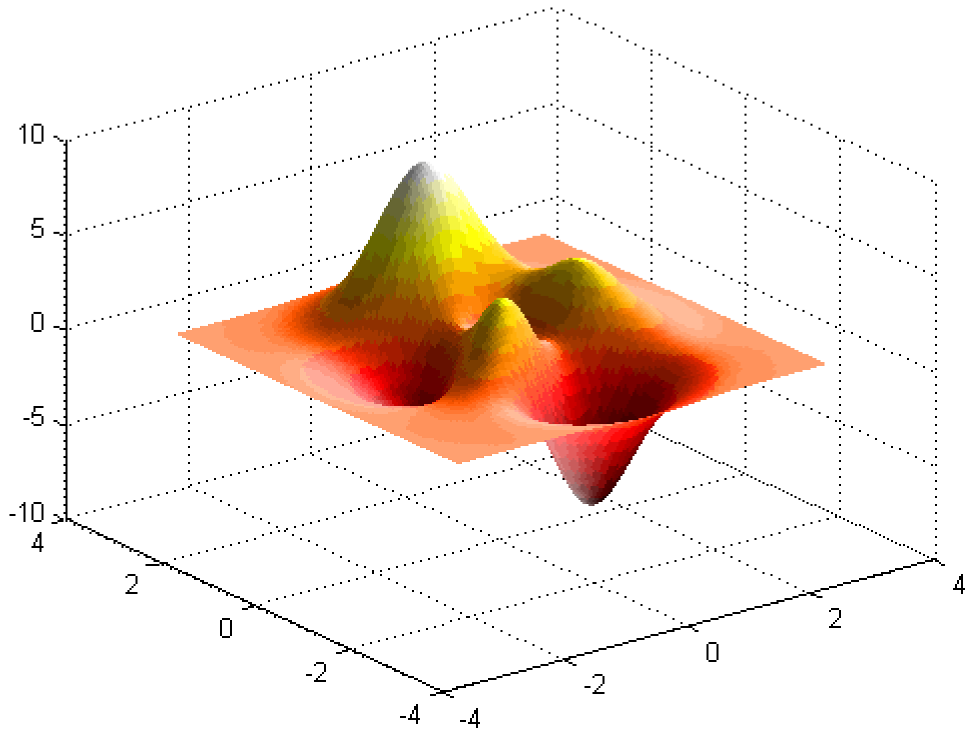
Remarks

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 color map. 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.

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
```



See Also

caxis, colormap, light, meshlsrm, surf, surflsrm

shapeinfo

Purpose

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.

The shapefile format was defined by the Environmental Systems Research Institute (ESRI) to store nontopological vector geometry and attribute information for spatial features. A shapefile consists of a main file, an index file, and an xBASE table. All three files have the same base name and are distinguished by the extensions `.SHP`, `.SHX`, and `.DBF`, respectively (e.g., given the base name `'roads'` the shapefile filenames would be `'roads.SHP'`, `'roads.SHX'`, and `'roads.DBF'`).

`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 directory and have valid file extensions. 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.

Field Descriptions

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.

Note `shapeinfo` cannot tell you what coordinate system data in a shapefile use. Coordinates can be either planar (*x, y*) or geographic (lat, lon) and have a variety of units. Because shapefiles do not document coordinate systems, `shapeinfo` cannot tell you what map projection coordinate data may be in or what the projection's parameters are. You need to obtain this information from your shapefile vendor. This information can be critical to the proper display of shapefile vector data, because by default `shaperead` will generate mapstructs that describe map coordinates (projected *x,y* pairs) unless you specify that the shapefile contains geographic coordinates using the optional `'UseGeoCoords',true` parameter/value pair.

See Also

`shaperead`

shaperead

Purpose

Read vector features and attributes from shapefile

Syntax

```
s = shaperead(filename)
[s, a] = shaperead(filename)
[s, a] = shaperead(filename, param1, val1, param2, val2,...)
```

Description

`s = shaperead(filename)` returns an N-by-1 geographic data structure (mapstruct or geostruct) array, `S`, containing an element for each nonnull spatial feature in the shapefile. `S` combines feature coordinates/geometry and attribute values. The extension of `filename` can be `.shp`, `.dbf` or `.shx`, or be omitted (see Remarks, below).

`[s, a] = shaperead(filename)` returns an N-by-1 mapstruct or geostruct array, `s`, and a parallel N-by-1 attribute structure array, `a`. Each array contains an element for each nonnull spatial feature in the shapefile. The attributes are read from the file `filename.dbf` (see Remarks, below).

`[s, a] = shaperead(filename, param1, val1, param2, val2,...)` returns a subset of the shapefile contents in `s` or `[s,a]`, as determined by the parameters `'RecordNumbers'`, `'BoundingBox'`, `'Selector'`, or `'Attributes'`. In addition, the parameter `'UseGeoCoords'` can be used in cases where you know that the *x*- and *y*-coordinates in the shapefile actually represent longitude and latitude.

Remarks

The shapefile format was defined by the Environmental Systems Research Institute (ESRI) to store nontopological vector geometry and attribute information for spatial features. A shapefile consists of a main file, an index file, and an xBASE table. All three files have the same base name and are distinguished by the extensions `.shp`, `.shx`, and `.dbf`, respectively (e.g., given the base name `'concord_roads'` the shapefile filenames would be `'concord_roads.shp'`, `'concord_roads.shx'`, and `'concord_roads.dbf'`).

`filename` can be the base name or the full name of any one of the component files. `shaperead` reads all three files as long as they exist in the same directory and have valid file extensions. If the main file (with extension .SHP) is missing, shaperead returns an error. If either of the other files is missing, shaperead returns a warning.

Supported Shape Types

shaperead supports the ordinary 2-D shape types: 'Point', 'Multipoint', 'PolyLine', and 'Polygon'. ('Null Shape' features can also be present in a Point, Multipoint, PolyLine, or Polygon shapefile, but are ignored.) shaperead does *not* support any 3-D or "measured" shape types: 'PointZ', 'PointM', 'MultipointZ', 'MultipointM', 'PolyLineZ', 'PolyLineM', 'PolygonZ', 'PolylineM', or 'Multipatch'.

Output Structure

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 the attributes included in the file:

Field Name	Field Contents	Comment
Geometry	Shape type string	'Point', 'Multipoint', 'PolyLine', or 'Polygon'
BoundingBox	[minX minY; maxX maxY]	Omitted for shape type 'Point'
X or Lon	Coordinate vector	NaN separators used in multipart PolyLine and Polygon shapes
Y or Lat	Coordinate vector	NaN separators used in multipart PolyLine and Polygon shapes
attr1	Value of first attribute	Included in output s if output a is omitted
attr2	Value of second attribute	Included in output s if output a is omitted
...	...	...

The names of the attribute fields (listed above as `Attr1, Attr2, ...`) are determined at run-time from the xBASE table (with extension `.DBF`) and/or optional, user-specified parameters. There can be many attribute fields, or none at all.

Field Descriptions

- **Geometry** — String with one of the following values: `'Point'`, `'MultiPoint'`, `'Line'`, or `'Polygon'`. (Note that these match the standard shapefile types, except that for shape type `'Polyline'` the value of the *`Geometry`* field is simply `'Line'`.)
- **BoundingBox** — Contains a 2-by-2 numerical array specifying the minimum and maximum feature coordinate values in each dimension (`min([x, y]); max([x, y]`, where x and y are N-by-1 and contain the combined coordinates of all parts of the feature). In the absence of documentation or metadata for the geodata in a shapefile, you can use this information to decide what kind of coordinate system (map or geographic) the shapes have.
- **X and Y / Lon and Lat (Coordinate vector)** — 1-by-N arrays of class double. For `'Point'` shapes, they are 1-by-1. In the case of multipart `'Polyline'` and `'Polygon'` shapes, `NaN`s are added to separate the lines or polygon rings. In addition, one terminating `NaN` is added to support horizontal concatenation of the coordinate data from multiple shapes.
- **Attribute fields** — Attribute names, types, and values are defined within a given shapefile. The following four types are supported: Numeric, Floating, Character, and Date. `shaperead` skips over other attribute types. The field names in the output shape structure are taken directly from the shapefile if they contain no spaces or other illegal characters and there is no duplication of field names (e.g., an attribute named `'BoundingBox'`, `'PointData'`, etc., or two attributes with the same names).

  Otherwise, the following “name mangling” is applied: Illegal characters are replaced by `'_'`. If the first character in the attribute name is illegal, a leading `'Z'` is added. Numerals are appended if needed to avoid duplicate names. The attribute values for a feature are taken from the shapefile and stored as doubles or character arrays:

Attribute Type in Shapefile	MATLAB Storage
Numeric	double (scalar)
Float	double (scalar)
Character	char array
Date	char array

Parameter-Value Options

By default, `shaperead` returns an entry for every nonnull feature and creates a field for every attribute. Use the first three parameters below (`RecordNumbers`, `BoundingBox`, and `Selector`) to be selective about which features to read. Use the fourth parameter (`Attributes`) to control the attributes to keep. Use the fifth (`UseGeoCoords`) to control the output field names.

Name	Description of Value	Purpose
`RecordNumbers`	Integer-valued vector, class double	Screen out features whose record numbers are not listed.
`BoundingBox`	2-by-(2, 3, or 4) array, class double	Screen out features whose bounding boxes fail to intersect the selected box. No trimming of features that partially intersect the box is performed.

Name	Description of Value	Purpose
`Selector`	Cell array containing a function handle and one or more attribute names. Function must return a logical scalar.	Screen out features for which the function, when applied to the corresponding attribute values, returns false.
`Attributes`	Cell array of attribute names	Omit attributes that are not listed. Use `{}` to omit all attributes. Also sets the order of attributes in the structure array.
`UseGeoCoords`	Scalar logical	If `true`, replace `X` and `Y` field names with `Lon` and `Lat`, respectively. Defaults to `false`.

Note If you do not know whether a shapefile uses latitude and longitude or map (planar) coordinates (i.e. contains unprojected or projected geodata), you can read it (or use `shapeinfo`) to obtain the `BoundingBox`; the ranges of coordinates may be sufficient information for you to decide what kind of coordinates you have. In some cases you may need to reread the shapefile with or without the `'UseGeoCoords',true` argument (it defaults to `false`), depending on whether you believe the coordinates are geographic latitude and longitude or map x and y, respectively. If you do not specify `UseGeoCoords`, the mapstruct is returned by `shaperead` containing `X` and `Y` fields rather than `Lon` and `Lat` fields. The `geoshow` function can sense such situations; it issues a warning and calls `mapshow` to plot mapstructs. The `mapshow` function refuses to draw geostructs, as it only accepts geographic data structures that have `X` and `Y` fields.

Examples

Example 1

Read the entire `concord_roads.shp` shapefile, including the attributes, in `concord_roads.dbf`:

```
S = shaperead('concord_roads.shp');
% Restrict output based on a bounding box and read only two
% of the feature attributes.
bbox = [2.08 9.11; 2.09 9.12] * 1e5;
S = shaperead('concord_roads','BoundingBox',bbox,...
              'Attributes',{'STREETNAME','CLASS'});

% Select the 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'});
N = hist([S.CLASS],1:7)
hist([S.LENGTH])
```

Example 2

Read worldwide city names and locations in latitude and longitude. Note presence of `'Lat'` and `'Lon'` fields:

```
S = shaperead('worldcities.shp', 'UseGeoCoords', true)

S =
318x1 struct array with fields:
    Geometry
    Lon
    Lat
    Name
```

Note For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

See Also `shapeinfo`, `shapewrite`, `updategeostruct`

Purpose

Write geographic data structure to shapefile

Syntax

```
shapewrite(S, filename)
shapewrite(S, filename, 'DbfSpec', dbfspec)
```

Description

`shapewrite(S, filename)` writes a geographic data structure to disk in shapefile format. `S` must be a valid mapstruct or geostruct with specific 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.

`filename` must be a character string specifying the output file name and location. If an extension is included, it must be `'.shp'` or `'.SHP'`.

`shapewrite` creates three output files,

- `[basename '.shp']`
- `[basename '.shx']`
- `basename '.dbf']`

where `basename` is `filename` without its extension.

If a given attribute is integer-valued for all features, then it is written to the `[basename '.dbf']` file as an integer. If an attribute is noninteger for any feature, then it is written as a fixed point decimal value with six digits to the right of the decimal place.

`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`. A DBF specification is a scalar MATLAB structure with one field for each feature attribute to be included in the output shapefile. To include an attribute in the output, make sure to provide a field in `dbfspec` with a field name identical to the attribute name (the corresponding field name in S), and 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' or 'C')
- FieldLength — The field length in the file, in bytes
- FieldDecimalCount — For numeric fields, the number of digits to the right of the decimal place

When a DBF spec is provided, a given attribute will be included in the output file only if it matches the name of a field in the spec.

The easiest way to construct a DBF spec is to call makedbfspec, 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.

Remarks

geostruct and mapstruct attribute names that are longer than 11 characters are truncated to 11 characters in copying as DBF field names in order to adhere to dBASE (.dbf) file specifications. Consider shortening long field names in your data structure before calling shapewrite to make field names in the DBF file more readable and to avoid introducing duplicate names as a result of truncation.

Example

Derive a shapefile from concord_roads.shp in which roads of CLASS 5 and greater are omitted. Note the use of the 'Selector' option in shaperead, together with an anonymous function, to read only the main roads from the original shapefile.

```
shapeinfo('concord_roads')  % 609 features

ans =
       Filename: [3x67 char]
      ShapeType: 'PolyLine'
    BoundingBox: [2x2 double]
    NumFeatures: 609
```

```
     Attributes: [5x1 struct]

S = shaperead('concord_roads', 'Selector', ...
    {@(roadclass) roadclass < 4, 'CLASS'});
shapewrite(S, 'main_concord_roads.shp')
shapeinfo('main_concord_roads')  % 107 features

ans =
       Filename: [3x24 char]
      ShapeType: 'PolyLine'
    BoundingBox: [2x2 double]
    NumFeatures: 107
     Attributes: [5x1 struct]
```

See Also

makedbfspec, shapeinfo, shaperead, updategeostruct

showaxes

Purpose

Toggle display of map coordinate axes

Syntax

```
showaxes
showaxes('on')
showaxes('off')
showaxes('hide')
showaxes('show')
showaxes('reset')
showaxes(color)
showaxes(color)
```

Description

`showaxes` toggles the visibility of the axes between the `'on'` and `'off'` conditions.

`showaxes('on')` sets the color of the axes (the `XColor` and `YColor` properties) to black.

`showaxes('off')` sets the color of the axes (the `XColor` and `YColor` properties) to the background color, effectively making them invisible. This is the default condition for map axes.

`showaxes('hide')` sets the `Visible` property of the axes to `'on'`.

`showaxes('show')` sets the `Visible` property of the axes to `'off'`.

`showaxes('reset')` sets the axes properties to the default map axes settings.

`showaxes(`*`color`*`)` sets the color of the axes (the `XColor` and `YColor` properties) to the color specified by any valid color string.

`showaxes(color)` sets the color of the axes (the `XColor` and `YColor` properties) to the color specified by the input RGB triple.

See Also

`axesm`

Purpose

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`

Purpose

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`

Purpose

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 `radtodeg(unwrapMultipart(degtorad(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

Purpose

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.

Remarks

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`

Purpose

Construct sparse regular data grid of 0s

Syntax

```
[Z,refvec] = spzerom(latlim,lonlim,scale)
```

Description

`[Z,refvec] = spzerom(latlim,lonlim,scale)` returns a sparse regular data grid consisting entirely of `0`s 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

Purpose

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)` returns a row vector of the latitude and longitude geographic standard distance for the data points specified by the columns of `lat` and `lon`.

`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 elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. The default ellipsoid model is a spherical Earth, which is sufficient for most applications. Output measurements are in terms of the distance units of the `ellipsoid` vector.

`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 = strmatch('Paris',{cities(:).Name});
London = strmatch('London',{cities(:).Name});
Rome = strmatch('Rome',{cities(:).Name});
Madrid = strmatch('Madrid',{cities(:).Name});
Berlin = strmatch('Berlin',{cities(:).Name});
Athens = strmatch('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`

Purpose

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 elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. The default ellipsoid model is a spherical Earth, which is sufficient for most applications. 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" on page 10-2 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 = strmatch('Paris',{cities(:).Name});
London = strmatch('London',{cities(:).Name});
Rome = strmatch('Rome',{cities(:).Name});
Madrid = strmatch('Madrid',{cities(:).Name});
Berlin = strmatch('Berlin',{cities(:).Name});
Athens = strmatch('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`

Purpose

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`.

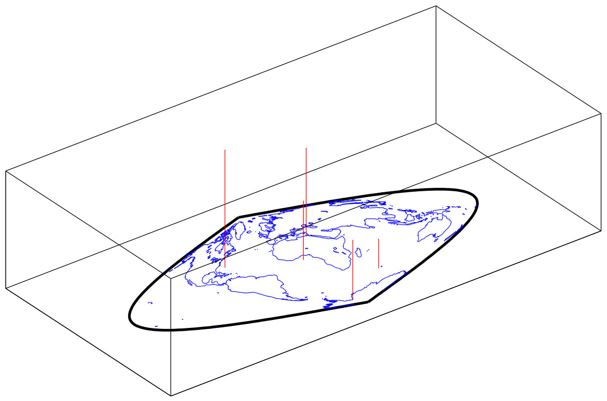
`h = stem3m(lat,lon,z,`*`LineType`*`)` allows the style of the stem plot's lines to be specified with any string *`LineType`* recognized by the MATLAB `line` function.

`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

Purpose

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.

Example

```
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

strs = strvcat(strs{:})

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

Purpose

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`.

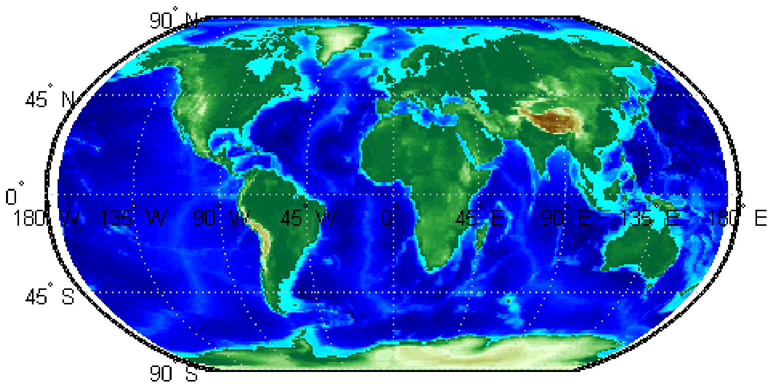
`h = surfacem(...)` returns a handle to the surface object.

Note Unlike `meshm` and `surfm`, `surfacem` always adds a surface to the current axes, regardless of hold state.

Example

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`

Purpose

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.

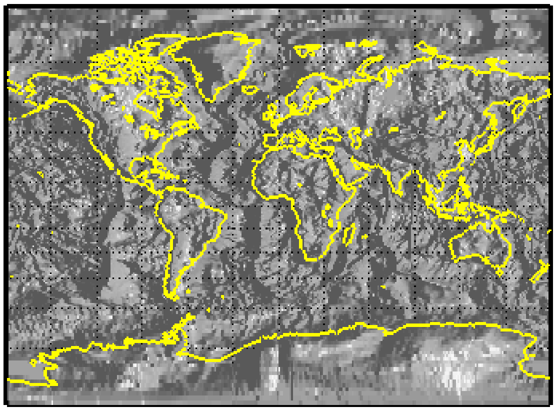
Remarks

`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.

Example

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')
```



See Also surfm

surflsrm

Purpose

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.

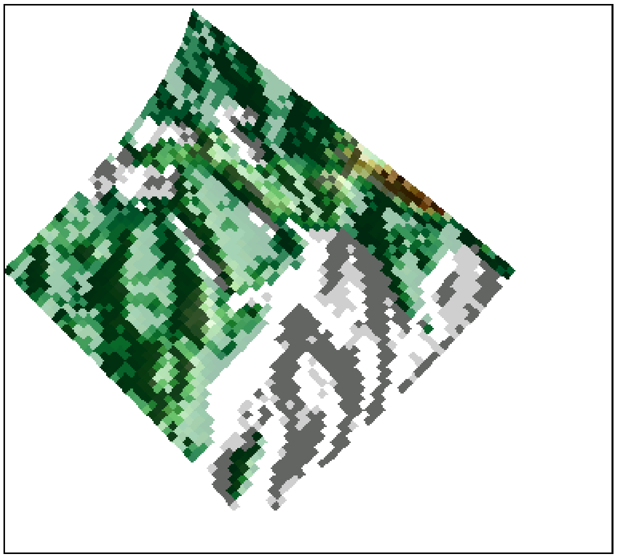
Remarks

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.

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)
```



See Also

`meshlsrm`, `meshm`, `pcolorm`, `shaderel`, `surfacem`, `surflm`, `surfm`

surfm

Purpose 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.

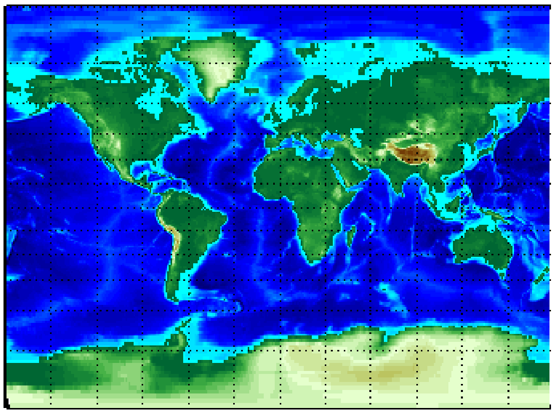
Remarks

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.

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)
```



See Also `geoshow` `meshgrat`, `meshm`, `pcolorm`, `surfacem`

Purpose

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`

Purpose

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`

Purpose

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 best available public domain data from almost 20 different sources, including the DCW-DEM and ETOPO5. The model is currently under development and will be updated as new data sources become available. The data set was created by the National Geophysical Data Center and World Data Center-A for Solid Earth Geophysics in Boulder, Colorado.

Remarks

Elevations and depths are given in meters above or below mean sea level.

The `tbase.bin` file is available on CD-ROM from

```
NOAA/NGDC
Mail Code E/GC3
325 Broadway
Boulder, CO  80303
USA
Tel: (303) 497-6338
Fax: (303) 497-6513
```

The data and documentation are available over the Internet via http and anonymous ftp.

Note For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

No byte-swapping or line-ending conversion is required.

Examples

Read every 10th point in the data set:

```
[Z,refvec] = tbase(10);
whos

  Name                Size          Bytes  Class

  Z                 216x432        746496  double array
  refvec              1x3              24  double array

limitm(Z,refvec)

ans =
   -90    90     0   360
```

Read data for Korea and Japan at the full resolution:

```
scalefactor = 1; latlim = [30 45]; lonlim = [115 145];
[Z,refvec] = tbase(scalefactor,latlim,lonlim);
whos datagrid

  Name       Size          Bytes  Class

  Z        180x360        518400  double array
```

See Also

`gtopo30`, `etopo`, `usgsdem`

Purpose

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.

Remarks

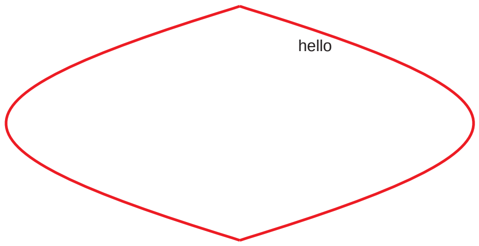
You may be working with scalar `lat` and `lon` data or vector `lat` and `lon` data. If you are in scalar mode and you enter a cell array of strings, you will get a text object with a multiline string. Also note that vertical slash characters, rather than producing multiline strings, will yield a single 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.

Example

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.

See Also

`axesm`, `text` (MATLAB function)

tgrline

Purpose

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.

Note For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

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.

Remarks

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 12-142 in the reference page for `displaym`. The `updategeostruct` function performs such conversions.

Examples

Read from the data for Washington, D.C.:

```
[CL,PR,SR,RR,H,AL,PL] = tgrline('TGR11001',1994,'Wash,DC');
```

See Also

`shaperead`, `updategeostruct`

Purpose

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
```

Limitations

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.

See Also

`panzoom`, `zoom`, `paperscale`, `axesscale`, `previewmap`

timezone

Purpose

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

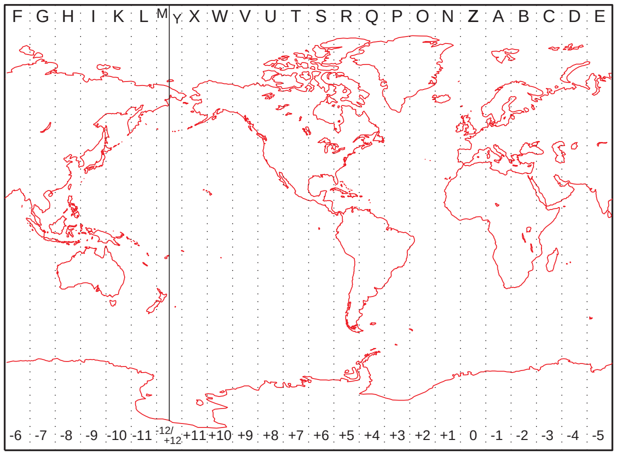
Given that it is locally 1330 (1:30 p.m.) at a longitude of 75°W, determine GMT:

```
[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 (180º 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.

Purpose

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. Chapter 14, "Map Projections Reference" of this guide includes Tissot diagrams for every projection on a global scale.

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.

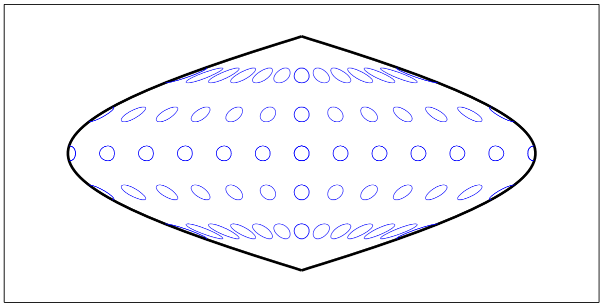
`Latint` is the latitude interval between indicatrix circle centers. If entered it should be in the map axes `AngleUnits`. The default value is one circle every 30º of latitude (that is, 0º, +/-30º, etc.).

`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.

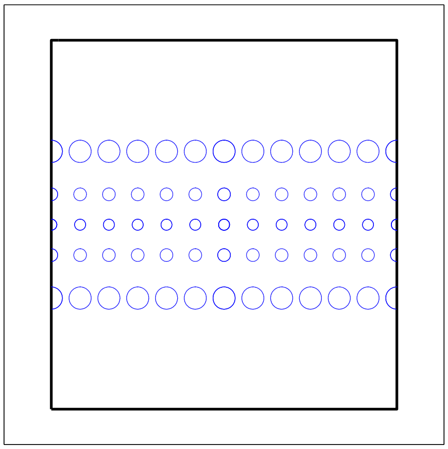
Examples

```
axesm sinusoid; framem
tissot
```



The Sinusoidal projection is equal area.

```
setm(gca,'MapProjection','Mercator')
```



The Mercator projection is conformal.

See Also

`mdistort`, `distortcalc`

See Chapter 14, "Map Projections Reference"

toDegrees

Purpose 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`, `radtodeg`, `toRadians`

Purpose

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

`degtorad`, `fromDegrees`, `fromRadians`, `toDegrees`

Purpose

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 `NaN`s.

`[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 elliptical definition of the Earth with a two-element ellipsoid model vector `ellipsoid`. The default ellipsoid is a spherical Earth, which is sufficient for most applications.

`[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')
```

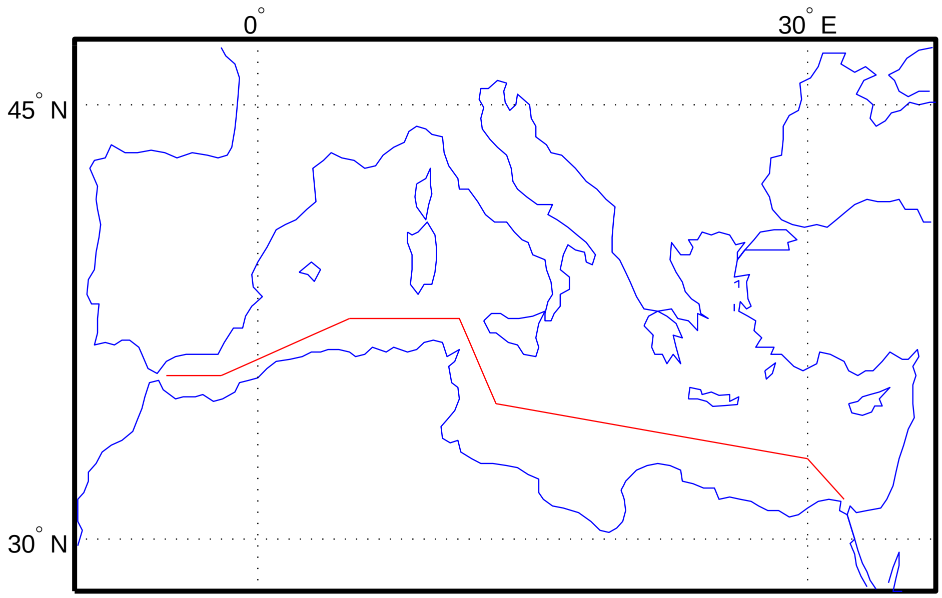
These are the waypoints Lt. Sextant has selected:

```
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`

Purpose

Geographic tracks from starting point, azimuth, and range

Syntax

```
[lat,lon] = track1(lat0,lon0,az)
[lat,lon] = track1(lat0,lon0,az,rng)
[lat,lon] = track1(lat0,lon0,az,rng,geoid)
[lat,lon] = track1(lat0,lon0,az,units)
[lat,lon] = track1(lat0,lon0,az,rng,units)
[lat,lon] = track1(lat0,lon0,az,rng,geoid,units)
[lat,lon] = track1(lat0,lon0,az,rng,geoid,units,npts)
[lat,lon] = track1(track,...)
mat = track1(...)
```

Description

`[lat,lon] = track1(lat0,lon0,az)` computes complete great circle tracks on a sphere starting at the point `lat0,lon0` and bearing along the input azimuth, `az`. The inputs can be scalar or column vectors.

`[lat,lon] = track1(lat0,lon0,az,rng)` uses the input `rng` to define the range of the great circle computed. The range input is in degrees of arc length along a sphere. If range is a column vector, then the track is computed from the starting point, with positive distance measured easterly. If range 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 `rng = []`, then the complete track is computed.

`[lat,lon] = track1(lat0,lon0,az,rng,geoid)` computes the great circle track on the ellipsoid defined by the input `geoid`. The `geoid` vector is of the form `[semimajor axis,eccentricity]`. If the semimajor axis is non-zero, `rng` is assumed to be in distance units matching the units of the semimajor axis. However, if `geoid = []`, or if the semimajor axis is zero, then `rad` is interpreted as an angle and the tracks are computed on a sphere as in the preceding syntax.

`[lat,lon] = track1(lat0,lon0,az,`*`units`*`)`, `[lat,lon] = track1(lat0,lon0,az,rng,`*`units`*`)`, and `[lat,lon] = track1(lat0,lon0,az,rng,geoid,`*`units`*`)` are all 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] = track1(lat0,lon0,az,rng,geoid,`*`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] = track1(`*`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 = track1(...)` 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 `lat0,lon0` inputs and column vectors for `az` and `rng` if desired.

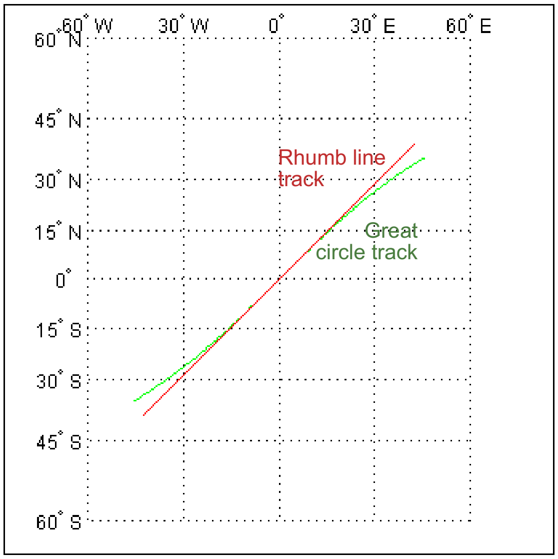
Definitions

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])

% 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

Purpose

Geographic tracks from starting and ending points

Syntax

```
[lat,lon] = track2(lat1,lon1,lat2,lon2)
[lat,lon] = track2(lat1,lon1,lat2,lon2,geoid)
[lat,lon] = track2(lat1,lon1,lat2,lon2,units)
[lat,lon] = track2(lat1,lon1,lat2,lon2,geoid,units)
[lat,lon] = track2(lat1,lon1,lat2,lon2,geoid,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,geoid)` computes the great circle track on the ellipsoid defined by the input `geoid`. The `geoid` vector is of the form `[semimajor axis,eccentricity]`. If `geoid = []`, a sphere is assumed.

`[lat,lon] = track2(lat1,lon1,lat2,lon2,`*`units`*`)` and `[lat,lon] = track2(lat1,lon1,lat2,lon2,geoid,`*`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,geoid,`*`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.

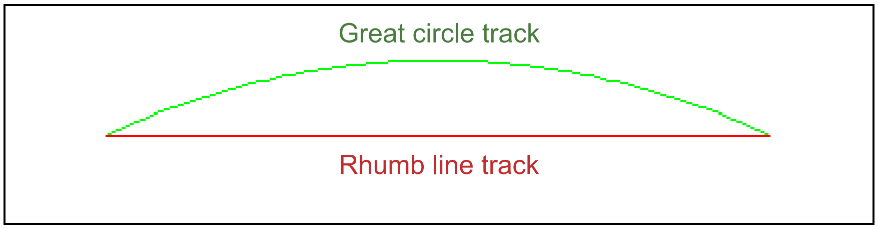
`mat = track2(...)` returns a single output argument where `mat = [lat lon]`. This is useful if a single track is computed. Multiple tracks can be defined from a single starting point by providing scalar inputs for `lat1,lon1` and column vectors for `lat2,lon2`.

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`

Purpose

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`

Purpose

Trim graphic objects to map frame

Syntax

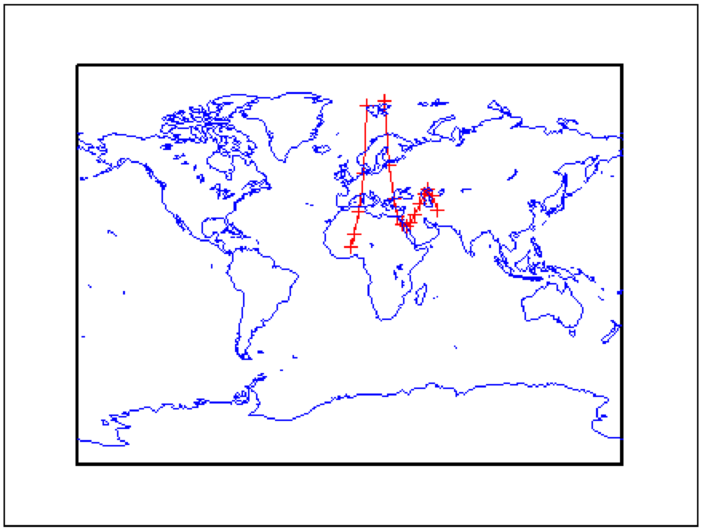
`trimcart(h)`

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`

Purpose

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`

Purpose

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

Purpose

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`

Purpose

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. `unitsratio` 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). `to` and `from` are case insensitive and can be either singular or plural.

Units of Length

`unitsratio` recognizes the following identifiers for converting units of length:

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)'`
Nautical mile	`'nm'`, `'nautical mile(s)'`
International foot	`'ft'`, `'international ft'`, `'foot'`, `'international foot'`, `'feet'`, `'international feet'`
Inch	`'in'`, `'inch'`, `'inches'`
Yard	`'yd'`, `'yard(s)'`

Unit Name	String(s)
international mile	`'mi', 'mile(s)', 'international mile(s)'`
U.S. survey foot	`'sf', 'survey ft', 'U.S. survey ft', 'survey foot', 'U.S. survey foot', 'survey feet', 'U.S. survey feet'`
U.S. survey mile (statute mile)	`'sm', 'survey mile(s)', 'statute mile(s)', 'U.S. survey mile(s)'`

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)
```

Purpose

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`

Purpose

Unwrap vector of angles with NaN-delimited parts

Syntax

```
unwrapped = unwrapMultipart(p)
```

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.

Examples

Example 1

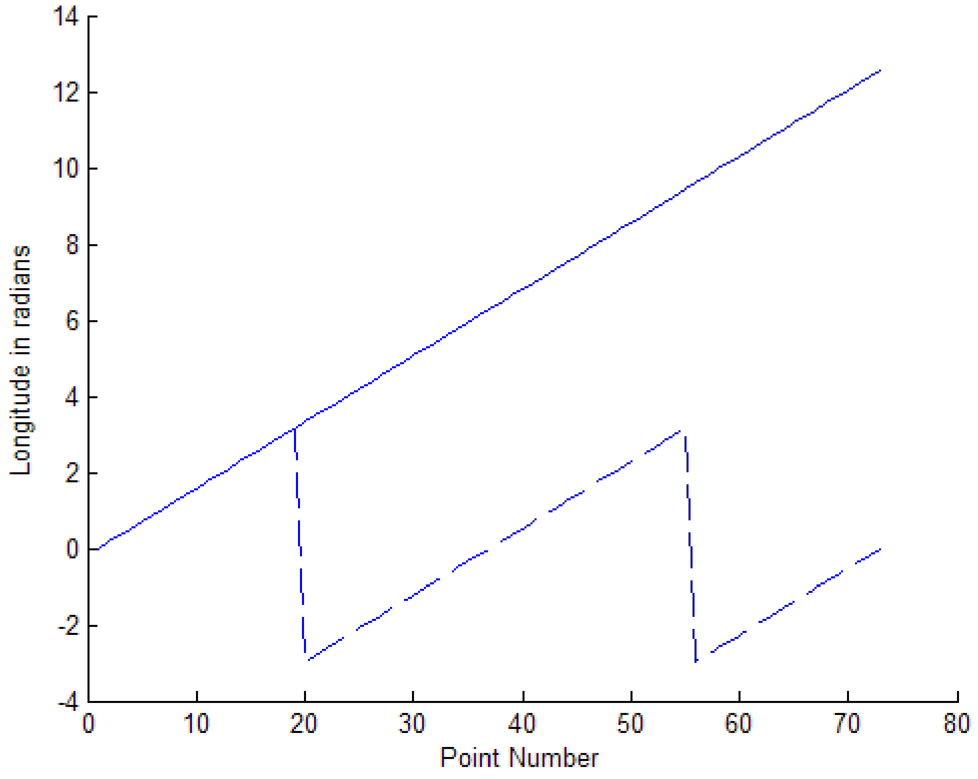
Compare the behavior `unwrapMultipart` to that of `unwrap`. The output of `unwrapMultipart` starts over again at 6.11 following the NaN, unlike the output of `unwrap`. The output of `unwrapMultipart` is equivalent to a concatenation (with NaN-separator) of separate calls to `unwrap`:

```
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(degtorad(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

Purpose

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.

Remarks

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
- 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.

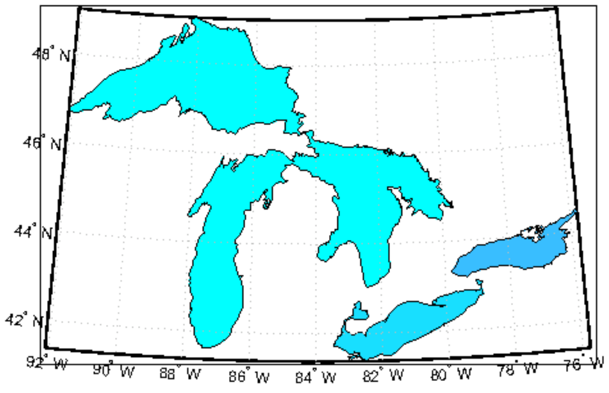
For information about the display structure format, see "Version 1 Display Structures" on page 12-142 in the reference page for `displaym`. For a discussion of the characteristics of geographic data structures, see "Mapping Toolbox Geographic Data Structures" on page 2-16 in the *Mapping Toolbox User's Guide*.

Example

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)
```



See Also

displaym, geoshow, makesymbolspec, mapshow, mapview, shaperead

Purpose

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')
h = usamap('allequal')
```

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'`, `'all'`, and `'allequal'`.

`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` 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.

`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, respectively, using projection parameters suggested by the U.S. Geological Survey. 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.

`h = usamap('allequal')` constructs the map axes with Alaska and Hawaii at the same scale as the conterminous states.

Remarks

`usamap` uses `tightmap` set 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.

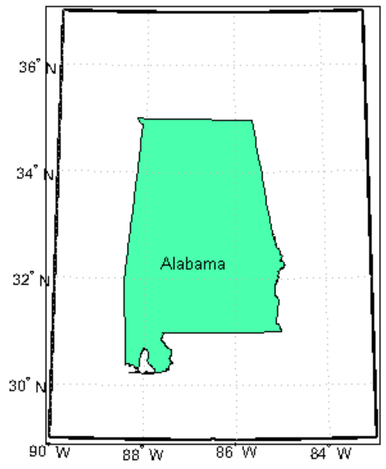
`axesscale(h(1))` resizes the axes containing Alaska and Hawaii to the same scale as the conterminous states.

Examples

Example 1

Make a map of Alabama only:

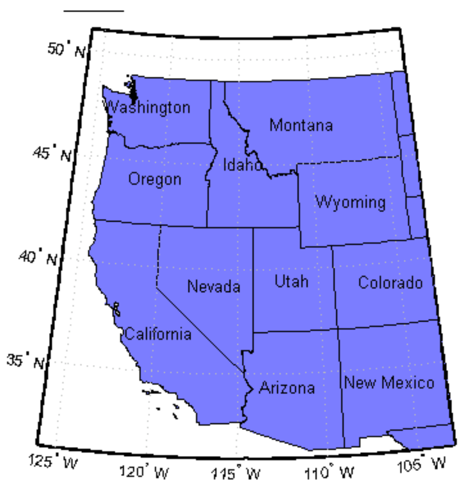
```
usamap('Alabama')
alabamahi = shaperead('usastatehi', 'UseGeoCoords', true,...
            'Selector',{@(name) strcmpi(name,'Alabama'), 'Name'});
geoshow(alabamahi, 'FaceColor', [0.3 1.0, 0.675])
textm(alabamahi.LabelLat, alabamahi.LabelLon, alabamahi.Name,...
  'HorizontalAlignment', 'center')
```



Example 2

Map a region extending from California to Montana:

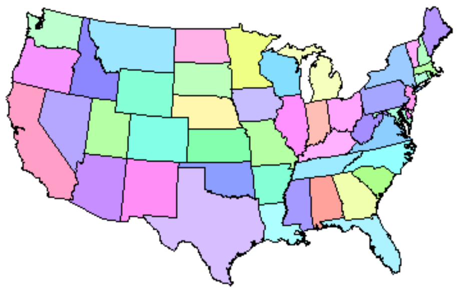
```
figure; ax = usamap({'CA','MT'});
set(ax, 'Visible', 'off')
latlim = getm(ax, 'MapLatLimit');
lonlim = getm(ax, 'MapLonLimit');
states = shaperead('usastatehi',...
        'UseGeoCoords', true, 'BoundingBox', [lonlim', latlim']);
geoshow(ax, states, 'FaceColor', [0.5 0.5 1])

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:

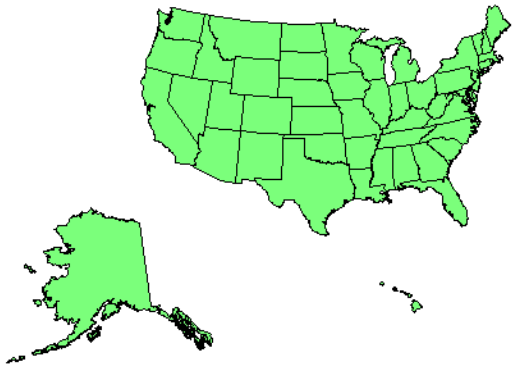
```
figure; ax = usamap('conus');
states = shaperead('usastatelo', 'UseGeoCoords', true,...
  'Selector',...
  {@(name) ~any(strcmp(name,{'Alaska','Hawaii'})), 'Name'});
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))}); %NOTE - colors 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('allequal');
set(ax, 'Visible', 'off')
states = shaperead('usastatelo', 'UseGeoCoords', true);
names = {states.Name};
indexHawaii = strmatch('Hawaii',names);
indexAlaska = strmatch('Alaska',names);
indexConus = 1:numel(states);
indexConus(indexHawaii) = [];
indexConus(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

usgs24kdem

Purpose

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.

Example

Use the archived San Francisco South 24K DEM file `sanfranciscos.dem.gz`, which is provided in the Mapping Toolbox `mapdemos` directory.

1. Gunzip the demo file to a temporary directory:

   ```
   filenames = gunzip('sanfranciscos.dem.gz', tempdir);
   demFilename = filenames{1};
   ```

2. Read every other point of the 1:24,000 DEM file.

   ```
   [lat, lon,Z,header,profile] = usgs24kdem(demFilename,2);
   ```

3. Delete the temporary gunzipped file.

```
    delete(demFilename);
```

4 As no negative elevations exist, move all points at sea level to -1 to color them blue:

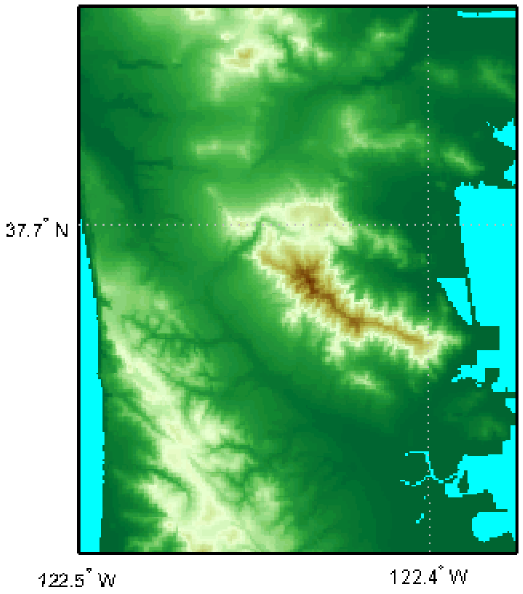
```
Z(Z==0) = -1;
```

5 Compute the latitude and longitude limits for the DEM:

```
latlim = [min(lat(:)) max(lat(:))]

latlim =
   37.6249   37.7504

lonlim = [min(lon(:)) max(lon(:))]

lonlim =
 -122.5008 -122.3740
```

6 Display the DEM values:

```
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, 'DisplayType','surface')
demcmap(Z)
daspectm('m',1)
```



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: NaN
           SmallestPrimary: NaN
        SourceMinCintUnits: NaN
            DataSourceDate: NaN
           DataInspRevDate: NaN
               InspRevFlag: ''
        DataValidationFlag: NaN
           SuspectVoidFlag: NaN
             VerticalDatum: NaN
           HorizontalDatum: NaN
               DataEdition: NaN
               PercentVoid: NaN
```

Remarks

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. 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

Purpose

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).

Remarks

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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

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
```

See Also

`usgs24kdem`, `gtopo30`, `etopo`, `tbase`, `usgsdems`

usgsdems

Purpose

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.

Remarks

This function only returns filenames for the contiguous United States.

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'
```

See Also

`usgsdem`

utmgeoid

Purpose

Select ellipsoids for given UTM zone

Syntax

```
ellipsoid = utmgeoid,
ellipsoid = utmgeoid(zone)
[ellipsoid,ellipsoidstr] = utmgeoid(...)
```

Description

`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 ellipsoid string used by the `almanac` function.

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

`utmzone`

Purpose

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`

Purpose

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` 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] * 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 either a referencing vector or a referencing 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.

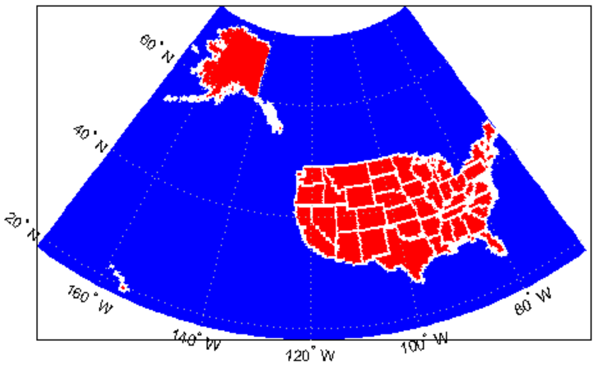
Notes

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.

Example 1

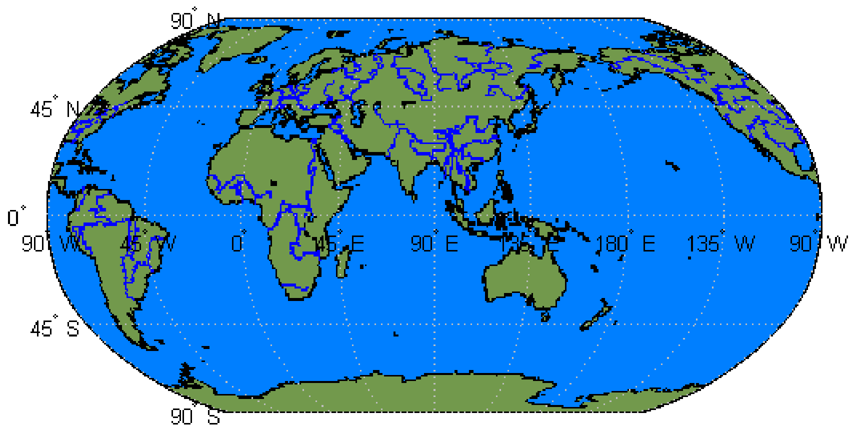
```
states = shaperead('usastatelo', 'UseGeoCoords', true);
lat = [states.Lat];
lon = [states.Lon];
[Z, R] = vec2mtx(lat, lon, 5, 'filled');
figure; worldmap(Z, R);
meshm(Z,R)
colormap(flag(3))
```



Example 2

Combine two separate calls to `vec2mtx` to create a 4-color raster map showing interior land areas, coastlines, oceans, and world rivers.

```
coast = load('coast.mat');
[Z, R] = vec2mtx(coast.lat, coast.long, ...
    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])
```



See Also

`imbedm`

Purpose

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.

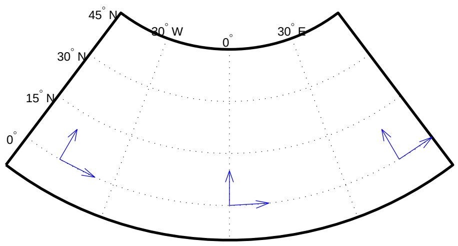
Background

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.

Remarks

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`

Purpose

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` 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. The value of `R` on output is identical to the value supplied as input.

`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.

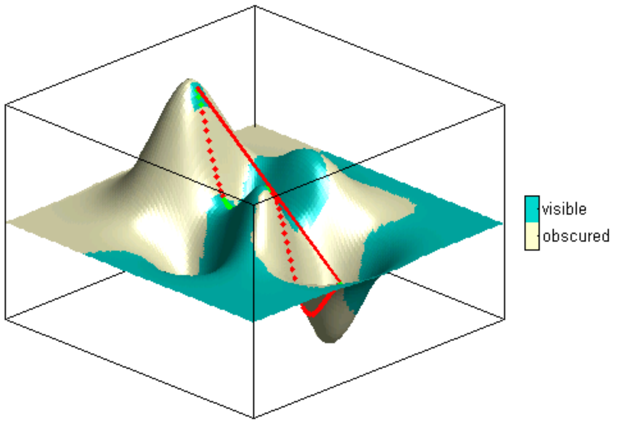
Remarks

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.

Example

Compute visibility for a point on the peaks map. Add the detailed information for the line of sight calculation between two points from `los2`.

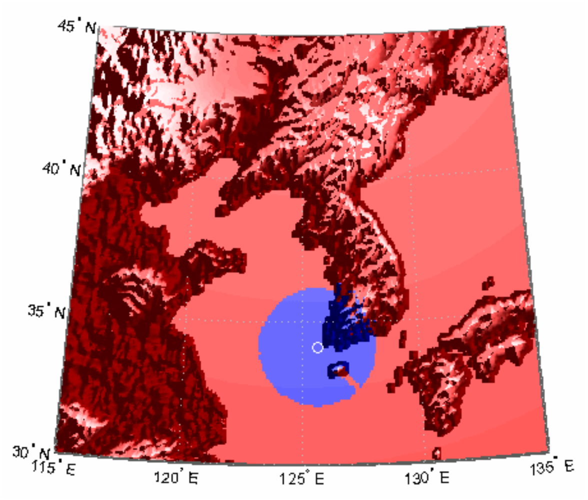
```
Z = 500*peaks(100);
refvec = [ 1000 0 0];
[lat1,lon1,lat2,lon2]=deal(-0.027,0.05,-0.093,0.042);

[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',almanac('earth','sphere','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'});
set(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 the surface areas visible by radar from an aircraft 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. Display the visible areas as blue and the obscured areas as red. Drape the visibility colors on an elevation map, and use lighting to bring out the surface topography. The aircraft's radar can see out a certain radius on the surface of the ocean, but 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.

```
load korea
map(map<0) = -1;
figure
worldmap(map,refvec)
da = daspect;
pba = pbaspect;
da(3) = 7.5*pba(3)/da(3);
daspect(da);
demcmap(map)
camlight(90,5);
camlight(0,5);
```

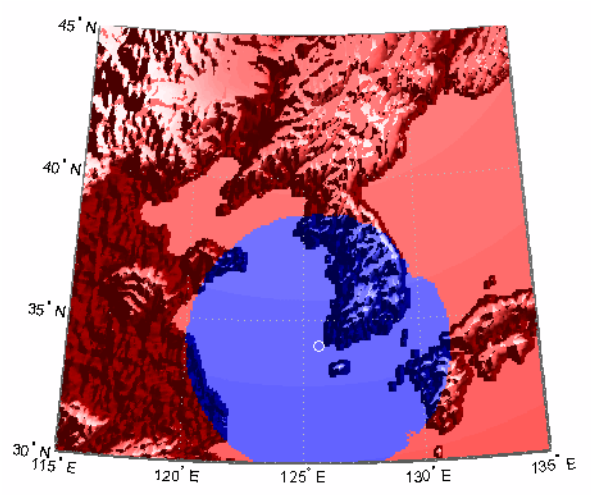
```
lighting phong
material([0.25 0.8 0])
lat = 34.0931; lon = 125.6578;
altobs = 3000; alttarg = 0;
plotm(lat,lon,'wo')
Re = almanac('earth','radius','m');
[vmap,vmapl] = viewshed( ...
   map,refvec,lat,lon,altobs,alttarg, ...
   'MSL','AGL',Re,4/3*Re);
meshm(vmap,vmapl,size(map),map)
caxis auto; colormap([1 0 0; 0 0 1])
lighting phong; material metal
axis off
```



Over what area can the radar plane flying at an altitude of 3000 meters have line-of-sight to other aircraft flying at 5000 meters? Now the area is much larger. 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)
material metal
lighting phong
```



See Also

`los2`

Purpose

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.

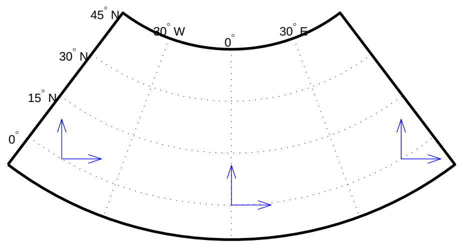
Background

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.

Remarks

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

Purpose

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.

Remarks

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 12-142 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: `http://www.mathworks.com/support/tech-notes/2100/2101.html`.

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'});
```

See Also

`vmap0read`, `vmap0rhead`, `geoshow`, `extractm`, `mlayers`, `updategeostruct`

vmap0read

Purpose

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.

Remarks

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.

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: '015'
     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
```

See Also

`vmap0data`, `vmap0rhead`

Purpose

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.

Remarks

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.

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,-,
```

See Also

`vmap0data`, `vmap0read`

Purpose

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.

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.

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 time-out mechanism is ignored)

`EnableCache`

Indicates if caching is allowed. If `true`, the `WMSCapabilites` object is cached and 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

Example

Construct a `WebMapServer` object that communicates with the Jet Propulsion Laboratory (JPL) WMS server and obtains its capabilities document.

```
jpl = wmsfind('jpl', 'SearchField', 'serverurl');
serverURL = jpl(1).ServerURL;
server = WebMapServer(serverURL);
capabilities = server.getCapabilities();
```

See Also

`WMSCapabilities` | `wmsfind` | `wmsinfo` | `WMSMapRequest` | `wmsread` | `wmsupdate`

Purpose

Get capabilities document from server

Syntax

```
capabilities = server.getCapabilities()
```

Description

`capabilities = server.getCapabilities()` retrieves the capabilities document from the server as a `WMSCapabilities` object and updates the `RequestURL` property. The `getCapabilities` method accesses the Internet to retrieve the document. The WMS server may periodically be unavailable, and several minutes may elapse before the document is retrieved.

Example

Retrieve the capabilities document from the Jet Propulsion Laboratory (JPL) server.

```
layers = wmsfind('jpl', 'SearchField', 'serverurl');
server = WebMapServer(layers(1).ServerURL);
capabilities = server.getCapabilities();
```

WebMapServer.getMap

Purpose

Get raster map from server

Syntax

`A = server.getMap(mapRequestURL)`

Description

`A = server.getMap(mapRequestURL)` dynamically renders and retrieves a color or grayscale, geographically referenced, raster map from the server and stores it in `A`. Parameters in the URL, `mapRequestURL`, define the map. The `getMap` method also updates the `WMSMapRequest.RequestURL` property `mapRequestURL`.

`getMap` accesses the Internet to retrieve the map, so several minutes may elapse before the map is retrieved, or the server may be unavailable.

Example

Retrieve a map of the `global_mosaic` layer.

```
layers = wmsfind('global_mosaic', ...
   'SearchField', 'layername', 'MatchType', 'exact');
layer = layers(1);
server = WebMapServer(layer.ServerURL);
mapRequest = WMSMapRequest(layer, server);
globalMosaic = server.getMap(mapRequest.RequestURL);
```

Purpose

Update layer properties

Syntax

`[updatedLayer, index] = server.updateLayers(layer)`

Description

`[updatedLayer, index] = server.updateLayers(layer)` returns a `WMSLayer` array with properties updated with values from the server. 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 may be unavailable, or several minutes may elapse before the properties are updated.

Examples

Update the properties of the `global_mosaic` layer.

```
layers = wmsfind('global_mosaic', ...
   'SearchField', 'layername', 'MatchType', 'exact');
layer = layers(1);
server = WebMapServer(layer.ServerURL);
updatedLayer = server.updateLayers(layer);
```

Update the properties of layers from multiple servers.

```
% Find layers with the name 'terra' in
% the server URL.
terra = wmsfind('terra', 'SearchField', 'serverurl');

% Find the layers for an individual server in the
% terra layer, update their properties, and append
% them to the updatedTerraLayers array.
servers = terra.servers();
updatedTerraLayers = [];
fprintf('Updating layer properties from %d servers.\n', ...
   numel(servers));
```

```
for k=1:numel(servers)
   serverLayers = ...
      terra.refine(servers{k}, 'SearchField', 'serverurl');
   fprintf('Updating properties from server %d:\n%s\n', ...
      k, serverLayers(1).ServerURL);
   server = WebMapServer(serverLayers(1).ServerURL);
   layers = server.updateLayers(serverLayers);

% Grow using concatenation; layers can have any length
% ranging from 0 to numel(serverLayers).
  updatedTerraLayers = [updatedTerraLayers; layers];
end
```

Purpose

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]`.

WMSCapabilities class

Purpose 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 E-mail 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
`E-mail`	String	E-mail 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

Example

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

`WebMapServer` | `wmsinfo` | `WMSLayer`

Purpose Display properties

Syntax

```
capabilities.disp()
```

Description `capabilities.disp()` is overloaded to remove hyperlinks and expand string and cell array properties.

WMSCapabilities.Layer property

Purpose 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
`Details`	Structure	Detailed information about layer

Purpose

Search local database for Web map servers and layers

Syntax

```
layers = wmsfind(querystr)
layers = wmsfind(..., param, val, ...)
```

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 these fields (`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(...,` *`param`*`,` *`val`*`, ...)` modifies the search of the WMS database based on the values of the parameters. You can abbreviate parameter names, and case does not matter. To modify the search, specify any of the parameters listed in the Inputs section.

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.

Inputs

querystr

Specifies the search string, such as `'temperature'`

Data Type: string

param, val

Modifies the search. Parameter names and values are shown below.

Parameter	Data Type	Value
`'IgnoreCase'`	Logical	Specifies whether to ignore case when performing string comparisons. Possible values are `true` or `false`; the default value is `true`.
`'Latlim'`	Two-element vector or scalar	A two-element vector of latitude 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. 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.

(Continued)

Parameter	Data Type	Value
`'Lonlim'`	Two-element vector or scalar	A two-element vector of longitude 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. 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	Has 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 `'*'`. The default value is `'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

(Continued)

Parameter	Data Type	Value
		partial match with querystr. If 'layer' is supplied, then both the 'layertitle' and 'layername' fields are searched. If 'server' is supplied, then both the 'serverurl' and 'servertitle' fields are searched. The layer information is returned if any supplied 'SearchFields' match. The default value is {'layer'}.

Examples

1. Using Basic Syntax

Find layers that contain temperature data and return a WMSLayer array.

```
layers = wmsfind('temperature');
```

2. Specifying Your Search

Find all layers that contain global temperature data and return a WMSLayer array.

```
layers = wmsfind('global*temperature');
```

3. Returning an Exact Match

Find all layers that contain an exact match for 'Major Rivers' in the LayerTitle field and return a WMSLayer array.

```
layers = wmsfind('Major Rivers', 'MatchType', 'exact', ...
   'IgnoreCase', false, 'SearchFields', 'layertitle');
```

4. Returning a Partial Match

Find all layers that contain a partial match for 'elevation' in the LayerName field and return a WMSLayer array.

```
layers = wmsfind('elevation', 'SearchField', 'layername');
```

5. Searching by Layer Name

Find all unique servers that contain `'global_mosaic'` as a layer name.

```
layers = wmsfind('global_mosaic', ...
   'SearchField', 'layername', 'MatchType', 'exact');
servers = layers.servers;
```

6. Finding Elevation for a Region

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);
```

7. Finding Elevation for a Specific Point

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);
```

8. Display Layer Properties

Find all layers provided by the Jet Propulsion Laboratory (JPL) server and display to the command window each layer title and layer name.

```
layers = wmsfind('jpl.nasa.gov', 'SearchField', 'serverurl');
layers.disp('Properties', {'layerTitle', 'layerName'});
```

9. Finding URLs of a Certain Type

Find all unique URLs of government servers.

```
layers = wmsfind('*.gov*', 'SearchField', 'serverurl');
```

```
servers = layers.servers;
```

10. Refining a Search

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')
```

11. Refining a Search of the Entire Database

Perform multiple searches and listings of the entire WMS Database. Please note that finding all layers from the WMS Database takes several minutes to execute. There are over 300,000 layers in the WMS Database.

```
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');
```

See Also

`wmsinfo` | `WMSLayer` | `wmsread` | `wmsupdate`

wmsinfo

Purpose

Information about WMS server from capabilities document

Syntax

```
[capabilities, infoRequestURL] = wmsinfo(serverURL)
[capabilities, infoRequestURL] = wmsinfo(infoRequestURL)
[capabilities, infoRequestURL] = wmsinfo(..., param, val)
```

Description

`[capabilities, infoRequestURL] = wmsinfo(serverURL)` accesses the Internet to read a capabilities document from a Web Map Service (WMS) server. (The capabilities document, an XML document, 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.

`[capabilities, infoRequestURL] = wmsinfo(infoRequestURL)` reads the capabilities document from a WMS `infoRequestURL` and returns the contents into `capabilities`.

`[capabilities, infoRequestURL] = wmsinfo(..., *param*, *val*)` specifies a parameter-value pair that modifies the request to the server. The parameter name can be abbreviated and is case-insensitive. The `'TimeoutInSeconds'` parameter, an integer-valued, scalar double, indicates the number of seconds to elapse before a server times out. The default value for this parameter is 60 seconds, and a value of 0 causes the time-out mechanism to be ignored.

`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.

Note about Internet Connection

The `wmsinfo` function requires an Internet connection. Several minutes may elapse before the document is returned, or a WMS server may be unavailable. 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.

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.
gsfcLayers = wmsfind('gsfc.nasa.gov', 'SearchField', 'serverurl');
capabilities = wmsinfo(gsfcLayers(1).ServerURL);

% Display the layer information in the command window.
capabilities.Layer
```

Sample Output

```
           Index: 304
     ServerTitle: 'NASA SVS Image Server'
       ServerURL: 'http://aes.gsfc.nasa.gov/cgi-bin/wms?'
      LayerTitle: '(2048x2048 Animation)'
       LayerName: '3352_24736'
          Latlim: [38.3727 39.3133]
          Lonlim: [-91.1052 -89.9017]
        Abstract: 'During the first half of 1993, ....'
CoordRefSysCodes: {'EPSG:4326'}
         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

```
Since measurements of Jakobshavn Isbrae were first taken....
```

See Also

`WebMapServer` | `WMSCapabilities` | `wmsfind` | `WMSLayer` | `wmsread`

WMSLayer class

Purpose

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 cannot set properties of the class except for `'Latlim'` and `'Lonlim'`, 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

Data Type: two-element vector

Default: `[]`

`Lonlim`

The western and eastern longitude limits of the layer in units of degrees

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

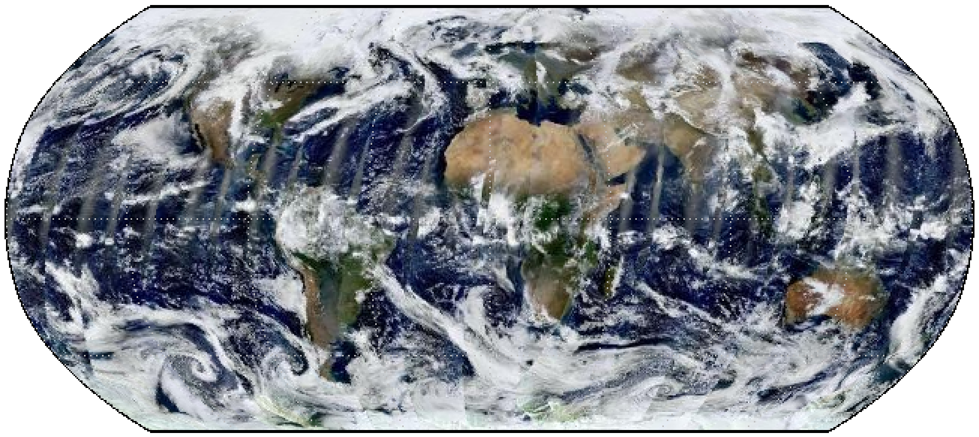
disp	Display properties
refine	Refine search
refineLimits	Refine search based on geographic limits
servers	Return URLs of unique servers
serverTitles	Return titles of unique servers

Example

Construct a `WMSLayer` object from a WMS `GetMap` request URL.

```
serverURL = 'http://onearth.jpl.nasa.gov/wms.cgi?';
url = [serverURL 'SERVICE=WMS' ...
   '&LAYERS=daily_planet&EXCEPTIONS=application/...
       vnd.ogc.se_xml', ...
   '&FORMAT=image/jpeg&TRANSPARENT=FALSE&HEIGHT=288', ...
   '&BGCOLOR=0xFFFFFF&REQUEST=GetMap&WIDTH=720&STYLES=' ...
   '&BBOX=-180.0,-72.0,180.0,72.0&SRS=EPSG:4326&VERSION=1.1.1'];

layer = WMSLayer('ServerURL', serverURL, ...
   'LayerName', 'daily_planet', ...
   'Latlim', [-72, 72], 'Lonlim', [-180,180]);
```

```
layer = wmsupdate(layer);
[A, R] = wmsread(layer, 'ImageHeight', 288, 'ImageWidth', 720);
figure
worldmap world
setm(gca,'maplatlimit',layer.Latlim, 'maplonlimit',layer.Lonlim)
mlabel off;plabel off
geoshow(A,R)
```



Courtesy NASA/JPL-Caltech

See Also

`WebMapServer` | `WMSCapabilities` | `wmsfind` | `wmsinfo` | `WMSMapRequest` | `wmsread` | `wmsupdate`

Purpose

Display properties

Syntax

```
layers.disp(..., param, val, ...)
```

Description

`layers.disp(..., param, val, ...)` is overloaded to display the index number followed by the property names and property values of the layer.

Use optional parameters for `disp` to modify the output display. You can abbreviate parameter names, and case does not matter.

Inputs

`param`, *`val`*

Modify output display. Parameter names and their permissible values appear in the table below.

Parameter	Data Type	Value	Default
`'Properties'`	String or cell array of strings	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'`. The values can be abbreviated and case does not matter. If `'Properties'` is `'all'`, then all the properties are listed.	`'all'`
`'Label'`	String	A case-insensitive string with permissible values of `'on'` or `'off'`. If `'Label'` is `'on'`, then the property name appears followed by its value. If `'Label'` is `'off'`, then only the property value appears in the output.	`'on'`
`'Index'`	String	A case-insensitive string with permissible values of `'on'` or `'off'`. If `'Index'` is `'on'`, then the element's index is listed in the output. If `'Index'` is `'off'`, then the index value is not listed in the output.	`'on'`

Examples

Display LayerTitle and LayerName properties to the command window.

```
layers = wmsfind('srtm30plus');
layers(1:5).disp( 'Index', 'off', ...
      'Properties',{'layertitle','layername'});
```

Sample Output

```
LayerTitle: 'SRTM30Plus World with Backdrop'
 LayerName: '10:4'
```

Sort and display the LayerName property and index.

```
layers = wmsfind('elevation');
[layerNames, index] = sort({layers.LayerName});
layers = layers(index);
layers.disp('Label','off', 'Properties', 'layername');
```

Sample Output

```
           Index: 1418
'topp:elevation_earth_300sec'

           Index: 1419
'topp:elevation_europe_150sec'

           Index: 1420
'topp:elevation_europe_150sec'
```

See Also

wmsfind

Purpose

Refine search

Syntax

`layers.refine(`*`querystr`*`, `*`param`*`, `*`val`*`, ...)`

Description

`layers.refine(`*`querystr`*`, `*`param`*`, `*`val`*`, ...)` searches for elements of `layers` in which values of the `Layer` or `LayerName` properties match the string *`querystr`*. Use the `'MatchType'` property to control whether the method uses partial or exact matching. The default is partial matching.

Inputs

`querystr`

Specifies the search string

Data Type: string

`param``, `*`val`*

Modifies the search by specifying any of the following optional parameters. Parameter names can be abbreviated and case does not matter.

Parameter	Data Type	Value	Default
`'SearchFields'`	Case-insensitive string or cell array of strings	Valid strings are `'abstract'`, `'layer'`, `'layertitle'`, `'layername'`, `'server'`, `'serverurl'`, `'servertitle'`, or `'any'`. The method searches the entries in the `'SearchFields'` properties of `layers` for a partial match of the entry with `querystr`. The layer information is returned if any supplied `'SearchFields'` match. If `'layer'` is supplied then both the `'LayerTitle'` and `'LayerName'` properties are searched. If	`{'layer'}`

(Continued)

Parameter	Data Type	Value	Default
		`'server'` is supplied, then both the `'ServerURL'` and `'ServerTitle'` fields are searched. If `'any'` is supplied, then the properties `'Abstract'`, `'LayerTitle'`, `'LayerName'`, `'ServerURL'`, and `'ServerTitle'` are searched.	
`'MatchType'`	Case-insensitive string with value `'partial'` or `'exact'`	If `'MatchType'` is `'partial'`, then a match is determined if the query string is found in the property value. If `'MatchType'` is `'exact'`, then a match is determined only if the query string exactly matches the property value. If `'MatchType'` is `'exact'` and `querystr` is `'*'`, a match is found if the property value matches the character `'*'`.	`'partial'`
`'IgnoreCase'`	Logical	If `'IgnoreCase'` is `true`, then case is ignored when performing string comparisons.	`true`

Example

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 = temperature.refine('annual');
sst = temperature.refine('sea surface');
annual_and_sst = sst.refine('annual');
annual_or_sst = [sst;annual];
```

See Also

`wmsfind` | `WMSLayer.refineLimits`

Purpose

Refine search based on geographic limits

Syntax

`layers.refineLimits(param, val, ...)`

Description

`layers.refineLimits(param, val, ...)` searches for elements of `layers` that match specific latitude or longitude limits. The results include a given layer only if its boundary quadrangle (as defined by the `Latlim` and `Lonlim` properties) is fully contained in the quadrangle specified by the optional `'Latlim'` or `'Lonlim'` parameters. Partial overlap does not result in a match. `'Latlim'` and `'Lonlim'` can be abbreviated and case does not matter. All angles are in units of degrees.

The default value of `[]` for either `'Latlim'` or `'Lonlim'` implies that all layers match the criteria. For example, if you specify the following:

```
layer.refineLimits('Latlim', [0 90], 'Lonlim', [])
```

then the results include all the layers that cover the northern hemisphere.

Inputs

param, *val*

Specifies geographic limits of search.

Parameter	Value
`'Latlim'`	A two-element vector of latitude 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 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.

Example

Find layers containing global elevation data.

```
    elevation = wmsfind('elevation');
    latlim = [-90, 90];
    lonlim = [-180, 180];
    globalElevation = ...
       elevation.refineLimits('Latlim', latlim, 'Lonlim', lonlim);

    % Print out the server titles from the unique servers.
   globalElevation.serverTitles'
```

Sample Output

```
   ans =

       'Global'
       'NRL GIDB Portal: Missouri CARES Maps'
       'NRL GIDB Portal: NOAA NGDC Maps'
```

See Also

`wmsfind`

Purpose

Return URLs of unique servers

Syntax

```
servers = layers.servers()
```

Description

`servers = layers.servers()` returns a cell array of URLs of unique servers.

Examples

Find all unique URLs of government servers.

```
layers = wmsfind('*.gov*','SearchField', 'serverurl');
servers = layers.servers;
sprintf('%s\n', servers{:})
```

Sample Output

```
http://www.ga.gov.au/bin/getmap.pl?dataset=national
http://www.geoportaligm.gov.ec/nacional/wms?
http://www.geoportaligm.gov.ec/regional/wms?
```

For each server that contains a temperature layer, list the server URL and the number of temperature layers.

```
temperature = wmsfind('temperature');
servers = temperature.servers;
for k=1:numel(servers)
   querystr = servers{k};
   layers = temperature.refine(querystr, ...
      'SearchFields', 'serverurl');
   fprintf('Server URL\n%s\n', layers(1).ServerURL);
   fprintf('Number of layers: %d\n\n', numel(layers));
end
```

Sample Output

```
Server URL
http://svs.gsfc.nasa.gov/cgi-bin/wms?
```

```
Number of layers: 36
```

See Also `wmsfind` | `WMSLayer.refine` | `WMSLayer.serverTitles`

Purpose

Return titles of unique servers

Syntax

```
serverTitles = layers.serverTitles()
```

Description

`serverTitles = layers.serverTitles()` returns a cell array of titles of unique servers.

Example

List titles of unique government servers.

```
layers = wmsfind('*.gov*', 'SearchField', 'serverurl');
titles = layers.serverTitles
sprintf('%s\n', titles{:})
```

Sample Output

```
High Resolution Nighttime Imagery (Las Vegas)
```

See Also

`wmsfind` | `WMSLayer.servers`

WMSLayer.Details property

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 the layer can be queried 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

Attributes Structure (Continued)

Field Name	Data Type	Field Content
`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

Dimension Structure (Continued)

Field Name	Data Type	Field Content
`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` 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 as single value (`value`); list of values (`value1, value2, ...`); interval defined by bounds and resolution (`min1/max1/res1`); or 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

Purpose

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 properties of `layer` are updated, 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. If set to a value other than `'EPSG:4326'`, `CoordRefSysCode` must be found in the `Details.BoundingBox` structure array found in the `Layer` property. When set to `'EPSG:4326'`, 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 set to a value other than `'EPSG:4326'`, 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'`

`RasterRef`

References the raster map to an intrinsic coordinate system

Data Type: 3-by-2 matrix

`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].

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. `XLim` may only be set if `CoordRefSysCode` is set 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. `YLim` may only be set if `CoordRefSysCode` is set 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 `ImageHeight` property is initialized 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 `ImageWidth` property is initialized 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 the property is set, `'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 to be made transparent or not. When `Transparent` is `true`, all pixels not representing features or data values in that layer are set to a transparent value, producing a composite map. When `Transparent` is `false`, non-data pixels are set 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 is `[255,255,255]`, which specifies the background color as white. `BackgroundColor` is stored as `uint8`. If `BackgroundColor` is set using non-`uint8` numeric values, the values are cast to `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` structure 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

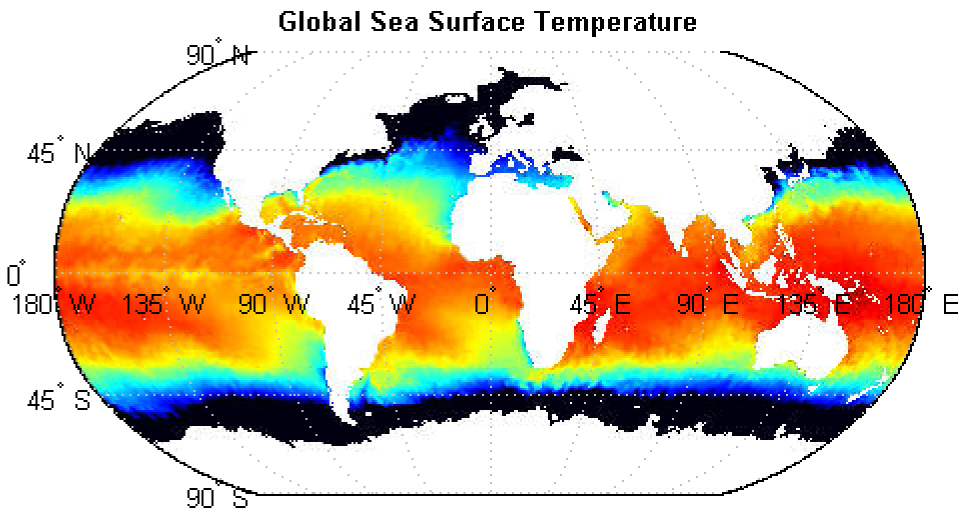
boundImageSize	Bound size of raster map

Examples

Basic Request

Read and display a global sea surface temperature map from the European Space Agency Web server.

```
esa = wmsfind('esa.int', 'SearchField', 'serverurl');
sst = esa.refine('global sea surface');
server = WebMapServer(sst.ServerURL);
mapRequest = WMSMapRequest(sst, server);
sstImage = server.getMap(mapRequest.RequestURL);

figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim);
geoshow(sstImage, mapRequest.RasterRef);
title(mapRequest.Layer.LayerTitle, ...
   'Interpreter', 'none', 'FontWeight', 'bold')
```

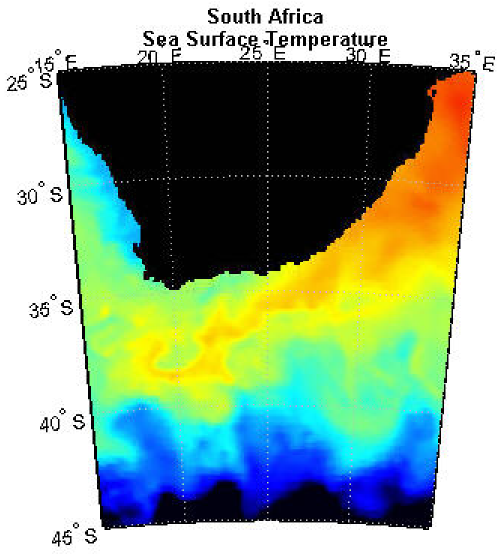


Courtesy European Space Agency

Geographic Limits Request

Read sea surface temperature for the ocean surrounding the southern tip of Africa and set the color of the land areas (background) to black.

```
esa = wmsfind('esa.int', 'SearchField', 'serverurl');
sst = esa.refine('global sea surface');
server = WebMapServer(sst.ServerURL);
mapRequest = WMSMapRequest(sst, server);

mapRequest.Latlim = [-45 -25];
mapRequest.Lonlim = [15 35];
mapRequest.BackgroundColor = [0 0 0];
sstImage = server.getMap(mapRequest.RequestURL);

figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim);
geoshow(sstImage, mapRequest.RasterRef);
title({'South Africa','Sea Surface Temperature'}, ...
   'FontWeight', 'bold')
```



Courtesy ESA

Style Request

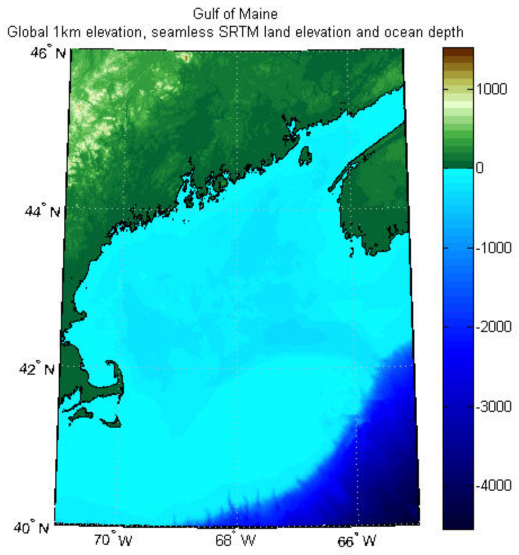
Read and display a global elevation and bathymetry layer for the Gulf of Maine at a 30 arc-seconds sampling interval. The values are in units of meters.

```
layers = wmsfind('srtmplus', 'MatchType', 'exact');
layer = layers(1);
server = WebMapServer(layer.ServerURL);
mapRequest = WMSMapRequest(layer, server);
mapRequest.Latlim = [40 46];
mapRequest.Lonlim = [-71 -65];
samplesPerInterval = 30/3600;
mapRequest.ImageHeight = ...
   round(diff(mapRequest.Latlim)/samplesPerInterval);
mapRequest.ImageWidth = ...
   round(diff(mapRequest.Lonlim)/samplesPerInterval);
mapRequest.StyleName = 'short_int';
mapRequest.ImageFormat = 'image/geotiff';
```

```
Z = server.getMap(mapRequest.RequestURL);
```

Display and contour the map at sea level (0 meters).

```
figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim)
geoshow(double(Z), mapRequest.RasterRef, ...
   'DisplayType', 'texturemap')
demcmap(double(Z))
set(gcf, 'renderer','zbuffer')
contourm(double(Z), mapRequest.RasterRef, [0 0], 'color', 'black')
colorbar
title ({'Gulf of Maine', mapRequest.Layer.LayerTitle}, ...
   'Interpreter','none')
```



Courtesy NASA/JPL

See Also

WebMapServer | wmsfind | wmsinfo | WMSLayer | wmsread

WMSMapRequest.boundImageSize

Purpose

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 row or column dimension length is calculated using the aspect ratio of the `Latlim` and `Lonlim` properties or the aspect ratio of the `XLim` and `YLim` properties, if they are set.

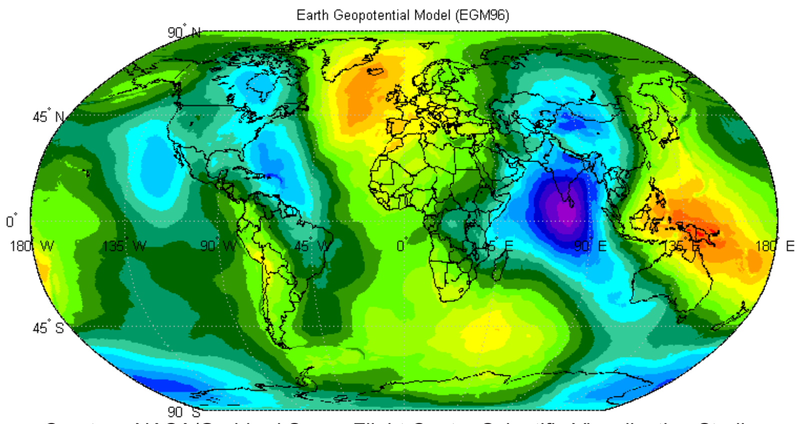
Image dimensions are measured in geographic or map coordinates. The longest image dimension is set to `imageLength`, and the shortest is set to the nearest integer value that preserves the aspect ratio, without changing the coordinate limits. The maximum value of `imageLength` is the maximum value of the `MaximumHeight` and `MaximumWidth` properties.

Example

Read and display a composite of multiple layers representing the EGM96 geopotential model of the Earth, coastlines, and national boundaries from the NASA Globe Visualization server. The rendered map has a spatial resolution of 0.5 degrees.

```
vizglobe = wmsfind('viz.globe', 'SearchField', 'serverurl');
coastlines = vizglobe.refine('coastline');
national_boundaries = vizglobe.refine('national*bound');
base_layer = vizglobe.refine('egm96');
layers = [base_layer;coastlines;national_boundaries];
request = WMSMapRequest(layers);
request.Transparent = true;
request = request.boundImageSize(720);
overlayImage = request.Server.getMap(request.RequestURL);

figure
worldmap('world')
geoshow(overlayImage, request.RasterRef);
```

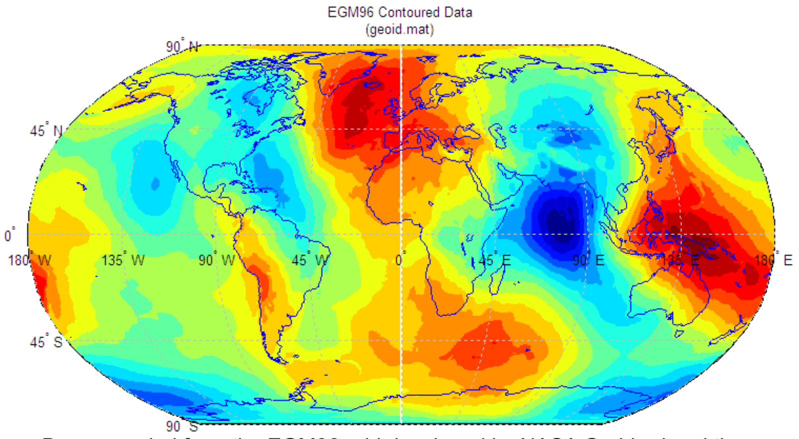
```
title(base_layer.LayerTitle)
```



Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

Compare the map with the contoured data from `'geoid.mat'`.

```
geoid = load('geoid');
coast = load('coast');
figure
worldmap('world')
contourfm(geoid.geoid, geoid.geoidrefvec, 15)
geoshow(coast.lat, coast.long)
title({'EGM96 Contoured Data', '(geoid.mat)'})
```



Downsampled from the EGM96 grid developed by NASA Goddard and the National Geospatial-Intelligence Agency (NGA)

Purpose

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 the property is set, `'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:

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.

Purpose

Retrieve WMS map from server

Syntax

```
[A, R] = wmsread(layer)
[A, R] = wmsread(mapRequestURL)
[A, R] = wmsread(layer, param, val, ...)
[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, a referencing matrix `R`, 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 larger spatial size of `A` is chosen to match its larger geographic dimension and 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. `N` is set to the integer value that provides the closest possible approximation to equal cell sizes in latitude and longitude, under the constraint (described earlier) that the map spans the full extent supported for the `layer`.

`[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` keyword. Obtain a `mapRequestURL` from the output of `wmsread`, the `RequestURL` property of a `WMSMapRequest` object, or an Internet search.

`[A, R] = wmsread(layer, param, val, ...)` specifies parameter-value pairs that modify the request to the server. Parameter names can be abbreviated and are case-insensitive. See the Inputs section for a list of available parameters.

`[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.

Note about Internet Connection

Establish an Internet connection to use `wmsread`. A WMS server may be unavailable, and several minutes may elapse before the map is returned. `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 and 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.

Inputs

`layer`

Contains information about the layer you are retrieving, such as the server URL

Data Type: `WMSLayer` object

`mapRequestURL`

Defines the request to the server

Data Type: string

`param``, `*`val`*

Specifies parameter-value pairs that modify the request to the server. See the table below for permissible values

Parameter	Data Type	Value
'Latlim'	Two-element vector	Specifies the latitude limits of the output image in the form [southern_limit northern_limit]. The limits are in degrees and must be ascending. By default, 'Latlim' is empty, and the full extent in latitude of layer is used. If Layer.Details.Attributes.NoSubsets is true, then 'Latlim' may not be modified.
'Lonlim'	Two-element vector	Specifies the longitude limits of the output image in the form [western_limit eastern_limit]. The limits are in degrees and 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, then 'Lonlim' may not be modified.
'ImageHeight'	Scalar, positive, integer-valued number	Specifies the desired height of the raster map in pixels. ImageHeight cannot exceed 8192. If layer.Details.Attributes. FixedHeight contains a positive number, then you cannot modify 'ImageHeight'.
'ImageWidth'	Scalar, positive, integer-valued number	Specifies the desired width of the raster map in pixels. ImageWidth cannot exceed 8192. If Layer.Details.Attributes.FixedWidth contains a positive number, then you cannot modify 'ImageWidth'.
'CellSize'	Scalar or two-element vector	Specifies the target size of the output pixels (raster cells) in units of degrees. If a scalar is specified, the value applies to both height and width dimensions. If a vector is specified, it is given in the form [height width]. An error is issued if both CellSize and ImageHeight or ImageWidth are specified. The output raster map must not exceed a size of [8192,8192].

Parameter	Data Type	Value
`'RelTolCellSize'`	Scalar or two-element vector	Specifies the relative tolerance for `'CellSize'`. If a scalar is specified, the value applies to both height and width dimensions. If a vector is specified, the tolerance appears in the order `[height width]`. The default value is `.001`.
`'ImageFormat'`	String	Specifies the desired image format for use in rendering the map as an image. If specified, the format must match an entry in the `Layer.Details.ImageFormats` cell array and must match one of the following supported formats: `'image/jpeg'`, `'image/gif'`, `'image/png'`, `'image/tiff'`, `'image/geotiff'`, `'image/geotiff8'`, `'image/tiff8'`, `'image/png8'`. If not specified, the format defaults to the first available format in the supported format list.
`'StyleName'`	String or cell array of strings	Specifies the style to use when rendering the image. By default, the style is set to the empty string. The `StyleName` must be a valid entry in the `Layer.Details.Style.Name` field. If multiple layers are requested, each with a different style, then `StyleName` must be a cell array of strings.
`'Transparent'`	Logical	Specifies if transparency is enabled. When `Transparent` is `true`, all pixels not representing features or data values are set to a transparent value. When `Transparent` is `false`, non-data pixels are set to the value of the background color. By default, the value is `false`.
`'BackgroundColor'`	Three-element vector	Specifies the color of the background (non-data) pixels of the map. If not specified, the default is `[255,255,255]`.

Parameter	Data Type	Value
'Elevation'	String	Indicates the desired elevation extent of the requested map. 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.
'Time'	String or numeric date number	Indicates the desired time extent of the requested map. 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, please see the WMSMapRequest.Time property reference page.
'SampleDimension'	Two-element cell array of strings	Indicates the name of a sample dimension (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

Parameter	Data Type	Value
		determines the permissible range of values for the second element of `'SampleDimension'`.
`'TimeoutInSeconds'`	Integer-valued, scalar double	Indicates the number of seconds to elapse before a server time-out is issued. By default, the value is 60 seconds. A value of 0 causes the time-out mechanism to be ignored.

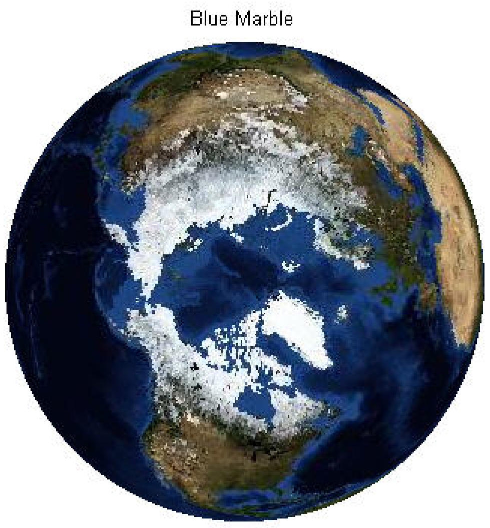
Definitions

The EPSG:4326 coordinate reference system is based on the WGS84 (1984 World Geodetic System) datum, with latitude and longitude in degrees and longitude referenced to the Greenwich Meridian.

Examples

Read and display the `'bluemarble'` layer from a NASA server.

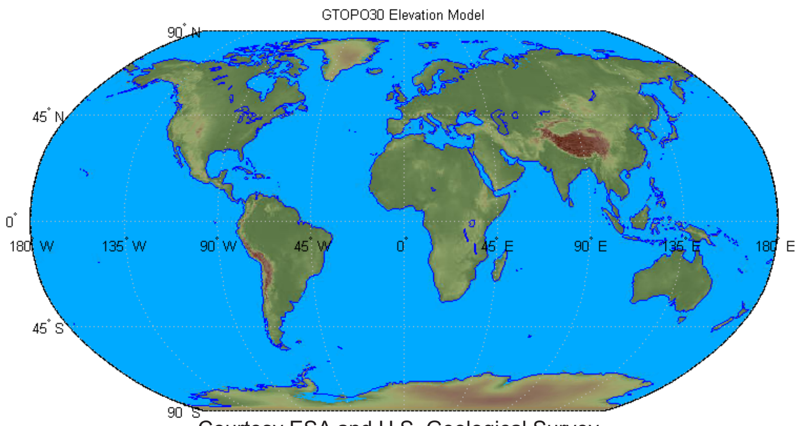
```
nasa = wmsfind('nasa', 'SearchField', 'serverurl');
layer = nasa.refine('bluemarble', 'SearchField', 'layername', ...
   'MatchType', 'exact');
[A, R] = wmsread(layer(1));
figure
axesm globe
axis off
geoshow(A, R)
title('Blue Marble')
```



Courtesy NASA/JPL-Caltech

Read and display a 0.5-degree resolution GTOPO30 layer from the European Space Agency Web server. Display the ocean in light blue by setting the background color.

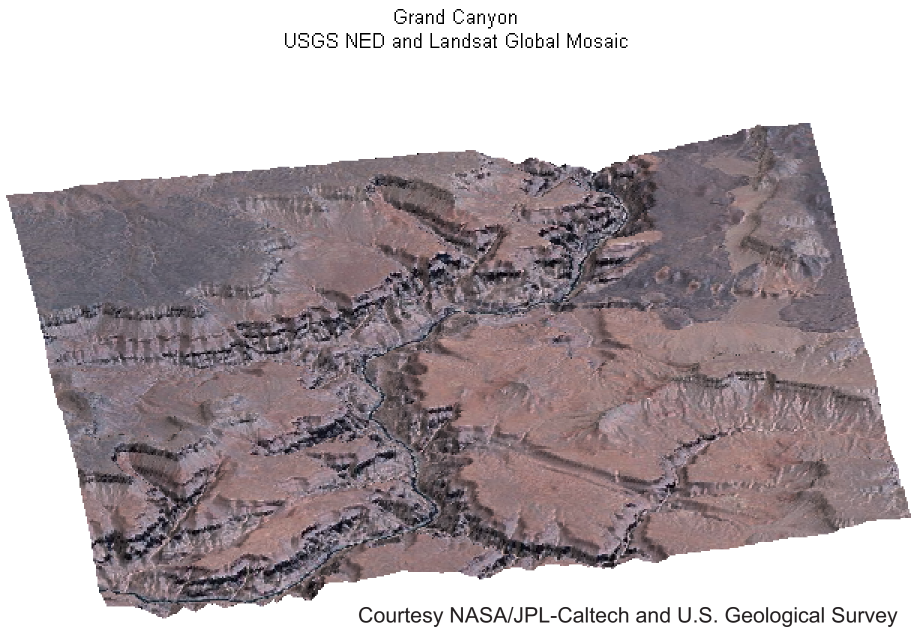
```
esa = wmsfind('esa.int', 'SearchField', 'serverurl');
gtopo30Layer = esa.refine('gtopo30');
oceanColor = [0 170 255];
[A,R] = wmsread(gtopo30Layer(1), 'BackgroundColor', oceanColor, ...
   'CellSize', .5);
figure
worldmap world
geoshow(A, R)
coast = load('coast');
plotm(coast.lat, coast.long)
title('GTOPO30 Elevation Model')
```



Courtesy ESA and U.S. Geological Survey

Drape Landsat imagery onto elevation data from the USGS National Elevation Dataset (NED) for an area surrounding the Grand Canyon. Read the `'global_mosaic'` and `'us_ned'` layers from the Web map server at the Jet Propulsion Laboratory.

```
% Obtain the layers of interest.
jpl = wmsfind('jpl.nasa.gov', 'SearchFields', 'serverurl');
jpl = wmsupdate(jpl);
global_mosaic = jpl.refine('global_mosaic', 'MatchType', ...
   'exact');
us_ned = jpl.refine('us_ned');

% Assign geographic extent and image size.
latlim = [36 36.23];
lonlim = [-113.36 -113.13];
imageHeight = 575;
imageWidth = 575;

% Read the global_mosaic layer.
[A, R] = wmsread(global_mosaic, 'StyleName', 'visual', ...
   'Latlim', latlim, 'Lonlim', lonlim, ...
   'ImageHeight', imageHeight, 'ImageWidth', imageWidth);

% Read the USGS NED layer.
[Z, R] = wmsread(us_ned, 'ImageFormat', 'image/geotiff', ...
   'StyleName', 'real', 'Latlim', latlim, 'Lonlim', lonlim, ...
   'ImageHeight', imageHeight, 'ImageWidth', imageWidth);

% Drape the Landsat image onto the elevation data.
figure
usamap(latlim, lonlim)
framem off; mlabel off; plabel off; gridm off
geoshow(double(Z), R, 'DisplayType', 'surface', 'CData', A);
daspectm('m',1)
title({'Grand Canyon', 'USGS NED and Landsat Global Mosaic'});
axis vis3d

% Assign camera parameters.
cameraPosition = [0.015136  0.67424  -72027];
cameraTarget = [-1.2904e-005 0.67187 3054.6];
cameraViewAngle = 8.1561;
cameraUpVector = [0.602132 0.0939748 5.05123e+006];

% Set camera and light parameters.
set(gca,'CameraPosition', cameraPosition, ...
   'CameraTarget', cameraTarget, ...
   'CameraViewAngle', cameraViewAngle, ...
   'CameraUpVector', cameraUpVector);

lightHandle = camlight;
camLightPosition = [0.0011253 0.22101 -4.1188e+006];
set(lightHandle, 'Position', camLightPosition);
```

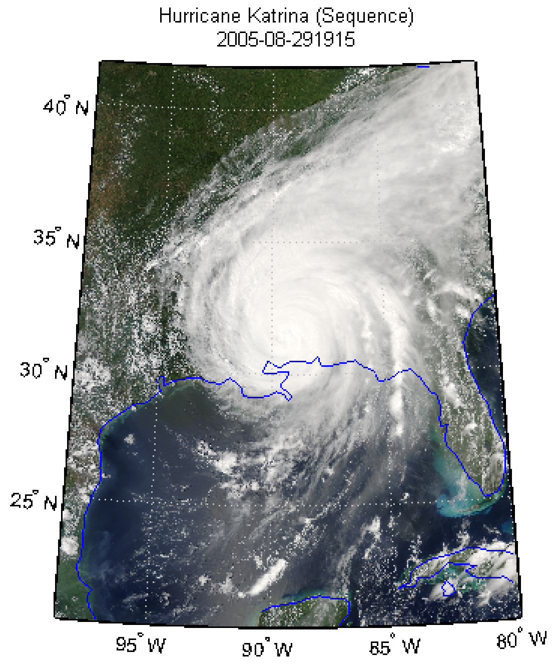

Grand Canyon
USGS NED and Landsat Global Mosaic
Courtesy NASA/JPL-Caltech and U.S. Geological Survey

Read and display a single sequence image from the MODIS instruments on the Aqua and Terra satellites that shows hurricane Katrina on August 29, 2005.

```
% Find the hurricane Katrina sequence layer.
katrina = wmsfind('Hurricane Katrina (Sequence)');
katrina = wmsupdate(katrina(1));

% The Dimension.Extent field shows a sequence delimited
% by commas. The sequence starts on August 24 and ends
% on August 31. The commas start at August 25 and end
% after August 30. Select the sequence corresponding to
% August 29.
commas = findstr(',', katrina.Details.Dimension.Extent);
extent = katrina.Details.Dimension.Extent;
```

```
sequence = extent(commas(end-2)+1:commas(end-1)-1);

% Obtain the time, latitude, and longitude limits
% from the values in the sequence. Split the sequence
% into a cell array of values by first finding
% all values between and including the parentheses,
% then remove the parentheses and split the values.
pat = '[(-.\d)]';
r = regexp(sequence, pat);
values = sequence(r);
values = strrep(values, '(', ' ');
values = strrep(values, ')', ' ');
values = regexp(values, '\s', 'split');
values = values(~cellfun('isempty', values));
time = values{1};
xmin = values{2};
ymin = values{3};
xmax = values{4};
ymax = values{5};

% Define latitude and longitude limits from the information
% in the sequence. The layer's geographic extent is assigned
% for the combined set of sequences. The requested map cannot
% be a subset of the layer's bounding box. In this rare case,
% set the layer's limits using the limits of the sequence.
latlim = [str2double(ymin) str2double(ymax)];
lonlim = [str2double(xmin) str2double(xmax)];
katrina.Latlim = latlim;
katrina.Lonlim = lonlim;

% Read and display the sequence map.
[A,R] = wmsread(katrina, 'SampleDimension', ...
   {katrina.Details.Dimension.Name, sequence});
figure
usamap(katrina.Latlim,katrina.Lonlim);
geoshow(A,R)
coast = load('coast');
```

```
plotm(coast.lat, coast.long)
title({katrina.LayerTitle, time})
```



Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

See Also

`WebMapServer` | `wmsfind` | `wmsinfo` | `WMSLayer` | `WMSMapRequest` | `wmsupdate`

Purpose

Synchronize WMSLayer object with server

Syntax

```
[updatedLayers, index] = wmsupdate(layers)
[...] = wmsupdate(layers, param, val, ...)
```

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, param, val, ...)` specifies parameter-value pairs that modify the request. Parameter names can be abbreviated and are case-insensitive.

Note about Internet Connection

The function accesses the Internet to update the properties. The WMS server may periodically be unavailable and several minutes may elapse before the layers are updated. The function 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.

Inputs

`layers`

Contains `WMSLayer` objects

Data Type: `WMSLayer` array

`param`, `val`

Modifies the request. See the table below for permissible values.

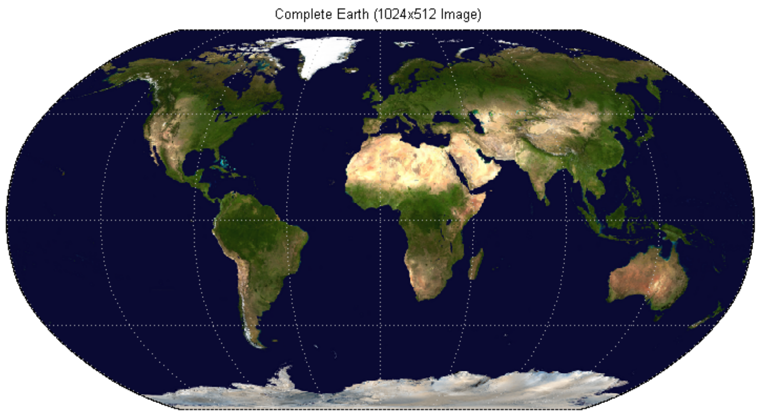
Parameter	Data Type	Value	Default
`'TimeoutInSeconds'`	Integer-valued, scalar double	Indicates the number of seconds before a server times out. A value of 0 causes the time-out mechanism to be ignored.	60 seconds
`'AllowMultipleServers'`	Logical scalar	Indicates whether the `layer` array may contain elements from multiple servers. Use caution when setting the value to `true`, since a request is made to each unique server and each request may take several minutes to finish.	`false` (indicates the array must contain elements from the same server)

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 blue marble layer containing the term
% '1024x512' in its LayerTitle.
gsfc = wmsfind('svs.gsfc.nasa.gov', 'SearchField', 'serverurl');
gsfc = wmsupdate(gsfc);
blue_marble = gsfc.refine('blue marble', 'SearchField', ...
   'abstract');
blueMarbleQuery = '1024x512';
layer = blue_marble.refine(blueMarbleQuery);

% Display the layer.
[A, R] = wmsread(layer);
figure
```

```
worldmap world
plabel off; mlabel off
geoshow(A, R);
title(layer.LayerTitle)
```



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);
```

See Also

`WebMapServer` | `wmsfind` | `wmsinfo` | `WMSLayer` | `wmsread`

worldfileread

Purpose

Read worldfile and return referencing matrix

Syntax

`R = worldfileread(worldfilename)`

Description

`R = worldfileread(worldfilename)` reads the worldfile data from `worldfilename` and constructs the referencing matrix `R`.

`R` is a 3-by-2 affine transformation matrix that is used in `pix2map` and `map2pix` to transform pixel row and column coordinates to/from map/model coordinates according to `[x y] = [row col 1] * R`.

Example

```
R = worldfileread('concord_ortho_w.tfw')

R =      1.0e+005 *
                         0  -0.000010000000000
          0.000010000000000                  0
          2.069995000000000   9.130005000000001
```

See Also

`getworldfilename`, `makerefmat`, `pix2map`, `map2pix`, `worldfilewrite`

Purpose

Construct worldfile from referencing matrix

Syntax

`worldfilewrite(R, worldfilename)`

Description

`worldfilewrite(R, worldfilename)` calculates the worldfile entries corresponding to referencing matrix `R` and writes them into the file `worldfilename`.

`R` is a 3-by-2 affine transformation matrix that is used in `pix2map` and `map2pix` to transform pixel row and column coordinates to/from map/model coordinates according to `[x y] = [row col 1] * R`.

Example

```
R = worldfileread('concord_ortho_w.tfw');
worldfilewrite(R,'concord_ortho_w_test.tfw');
```

constructs the referencing matrix `R` from `concord_ortho_w.tfw`, then reconstructs a copy of the worldfile from `R`.

See Also

`getworldfilename`, `pix2map`, `map2pix`, `worldfileread`

worldmap

Purpose

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` 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.

`h = worldmap(...)` returns the handle of the map axes.

For cylindrical projections, `worldmap` uses `tightmap` set 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`.

Examples

Example 1

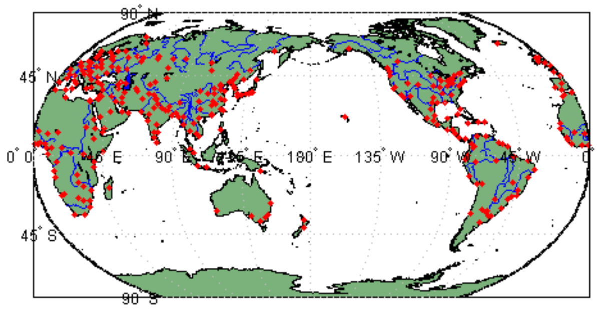
Set up a world map and draw coarse coastlines:

```
worldmap('World')
load coast
plotm(lat, long)
```

Example 2

Set up `worldmap` with land areas, major lakes and rivers, and cities and populated places:

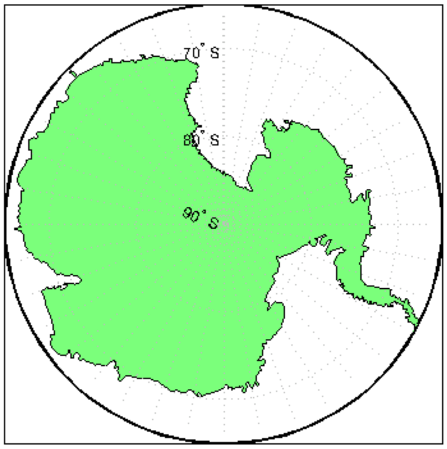
```
ax = worldmap('World');
setm(ax, 'Origin', [0 180 0])
land = shaperead('landareas', 'UseGeoCoords', true);
geoshow(ax, land, 'FaceColor', [0.5 0.7 0.5])
lakes = shaperead('worldlakes', 'UseGeoCoords', true);
geoshow(lakes, 'FaceColor', 'blue')
rivers = shaperead('worldrivers', 'UseGeoCoords', true);
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:

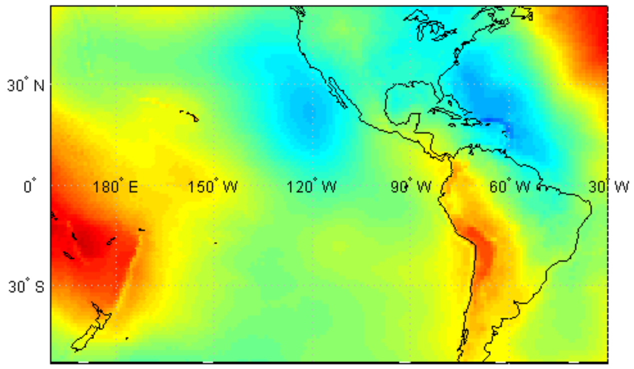
```
worldmap({'Africa','India'})
land = shaperead('landareas.shp', 'UseGeoCoords', true);
geoshow(land, 'FaceColor', [0.15 0.5 0.15])
cities = shaperead('worldcities', 'UseGeoCoords', true);
geoshow(cities, 'Marker', '.', 'Color', 'red')
```

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:

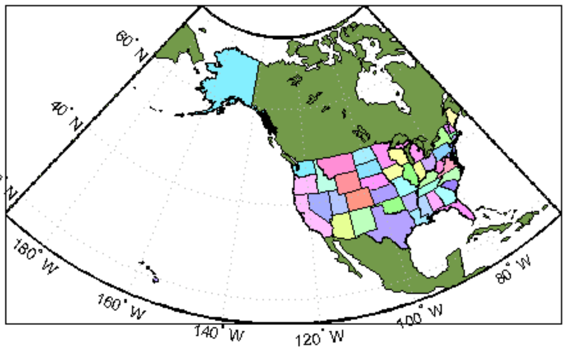
```
load korea
h = worldmap(map, refvec);
set(h, 'Visible', 'off')
geoshow(h, map, refvec, 'DisplayType', 'texturemap')
colormap(demcmap(map))
```

Example 7

Make a map of the United States of America, coloring state polygons:

```
ax = worldmap('USA');
load coast
geoshow(ax, lat, long,...
'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`

Purpose

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

Purpose

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`

Purpose

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 zero.)

See Also

`wrapTo180`, `wrapTo360`, `wrapToPi`

wrapToPi

Purpose

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]` such that 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`

Purpose

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 `handlem`, 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`

Purpose

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`

Purpose

Construct regular data grid of 0s

Syntax

```
[Z,refvec] = zerom(latlim,lonlim,scale)
```

Description

`[Z,refvec] = zerom(latlim,lonlim,scale)` returns a full regular data grid consisting entirely of `0`s 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`

Purpose

Define map axes and modify map projection and display properties

Activation

Command Line	Maptool	Map Display

`axesm`
`axesmui`
`c = axesmui(...)`
	Display > Projection	`extend-click map display`

Description

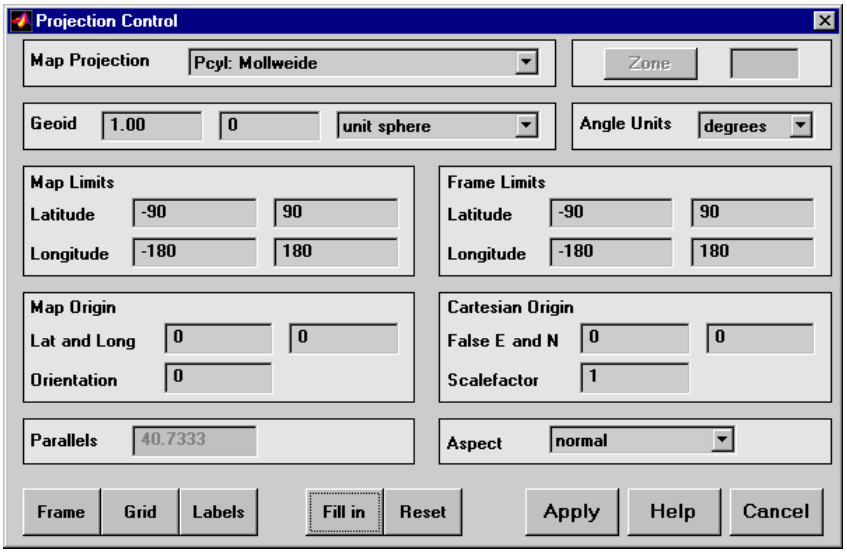
`axesm` activates a Projection Control dialog box, which allows map projection definition and property modification. If no map is currently defined, `axesm` creates a map axes with the Robinson projection as the default.

`axesmui` activates the Projection Control box for the current map axes.

`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



Map Projection: Pcyl: Mollweide

Geoid: 1.00 | 0 | unit sphere

Angle Units: degrees

Map Limits
Latitude: -90 | 90
Longitude: -180 | 180

Frame Limits
Latitude: -90 | 90
Longitude: -180 | 180

Map Origin
Lat and Long: 0 | 0
Orientation: 0

Cartesian Origin
False E and N: 0 | 0
Scalefactor: 1

Parallels: 40.7333

Aspect: normal

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'`.

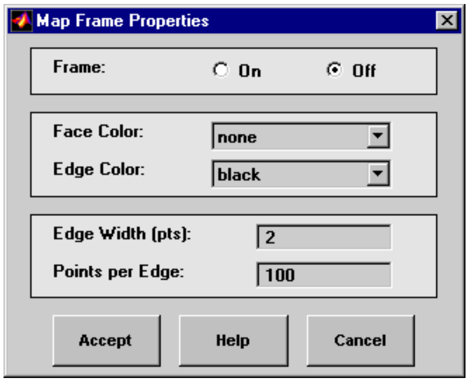
The **Apply** button is used to apply the projection and display settings to the current map, which results in the map being reprojected.

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.



Frame: On [✓ Off]

Face Color: none

Edge Color: black

Edge Width (pts): 2

Points per Edge: 100

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.

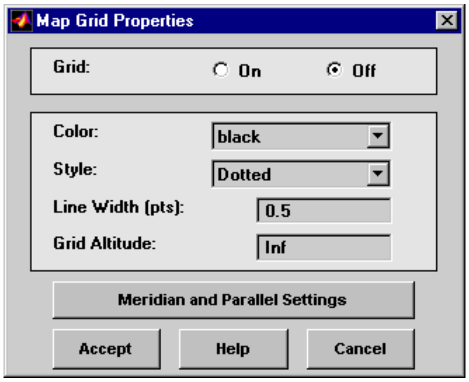
The **Points per Edge** edit box is used to enter the number of points used to display each edge of the map frame.

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.



Grid: On [✓ Off]

Color: black

Style: Dotted

Line Width (pts): 0.5

Grid Altitude: Inf

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.

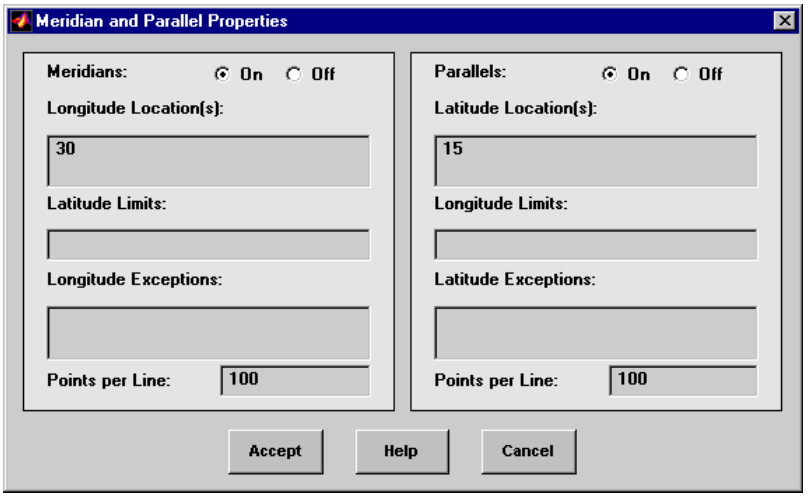
The **Meridian and Parallel Settings** button brings up the **Meridian and Parallel Properties** dialog box, which allows the properties of the meridian and parallel grid lines to be modified.

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.


Meridian and Parallel Properties

Meridians: [✓ On] Off

Longitude Location(s): 30

Latitude Limits:

Longitude Exceptions:

Points per Line: 100

Parallels: [✓ On] Off

Latitude Location(s): 15

Longitude Limits:

Latitude Exceptions:

Points per Line: 100

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 bedisplayed 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.

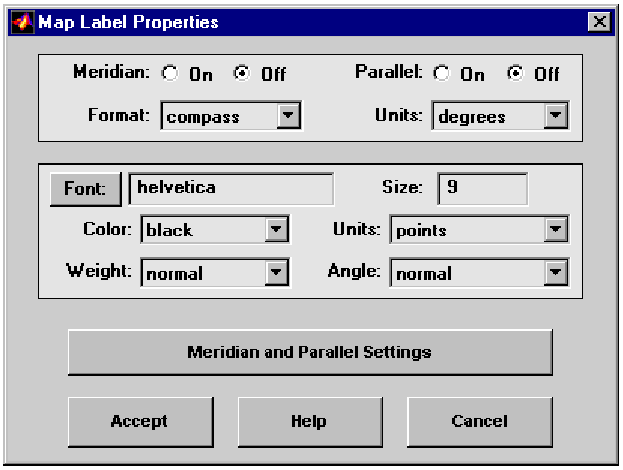
The **Accept** button accepts any modifications that have been made to the meridian and parallel grid line properties and return to the Map

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.


Map Label Properties

Meridian: On [✓ Off]   Parallel: On [✓ Off]

Format: compass   Units: degrees

Font: helvetica   Size: 9

Color: black   Units: points

Weight: normal   Angle: normal

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.

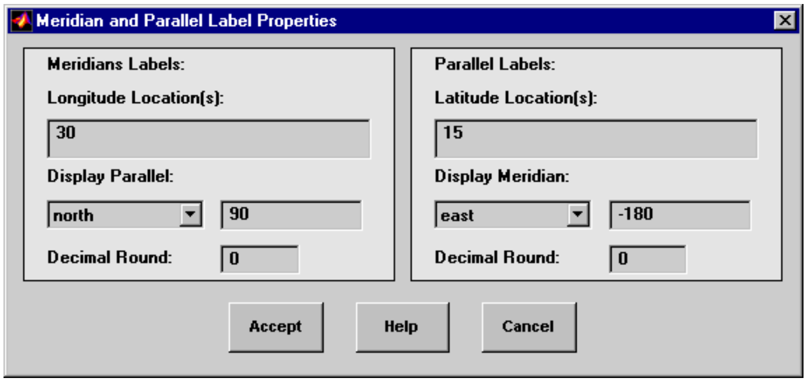
The **Meridian and Parallel Settings** button brings up the Meridian and Parallel Label Properties dialog box, which allows modification of properties specific to the meridian and parallel grid labels.

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

Purpose

GUI to clear mapped objects

Activation

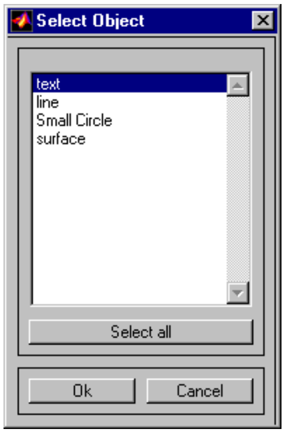
Command Line	Maptool
`clmo`	**Tools > Delete > Object**

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`

Purpose

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:

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.

Spin spins the colormap.

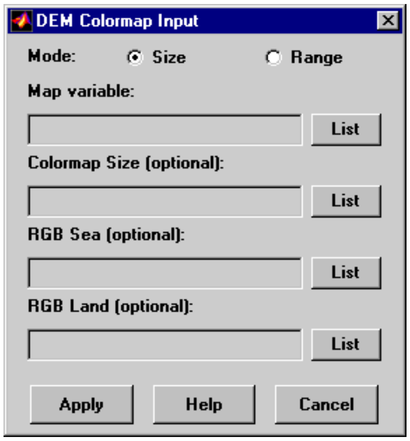
Define allows a workspace variable to be specified for the colormap.

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 Color Map 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 **Color Map 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 Color Map Input dialog box.

See Also

`colorm`, `demcmap`

Purpose

Create index map colormaps

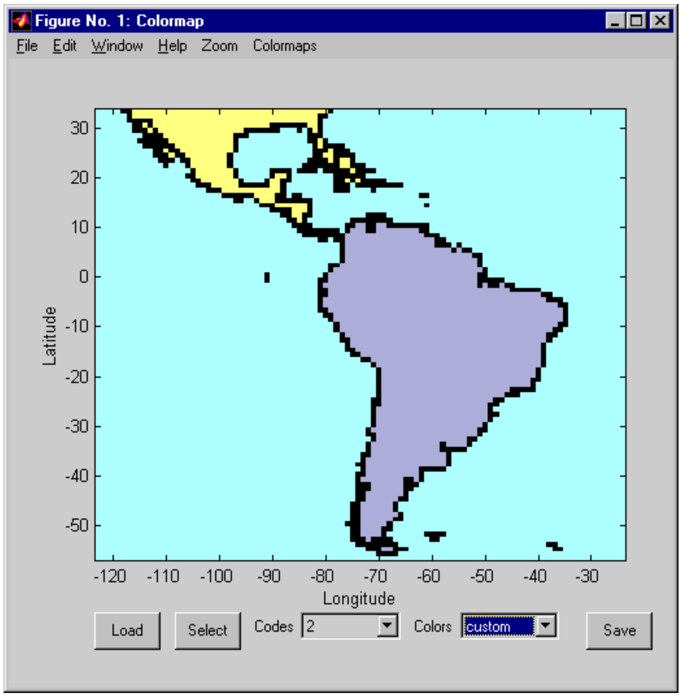
Activation

Command Line
`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 the `panzoom` 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.

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`, `panzoom`, `seedm`

demdataui

Purpose

UI for selecting digital elevation data

Activation

`demdataui`

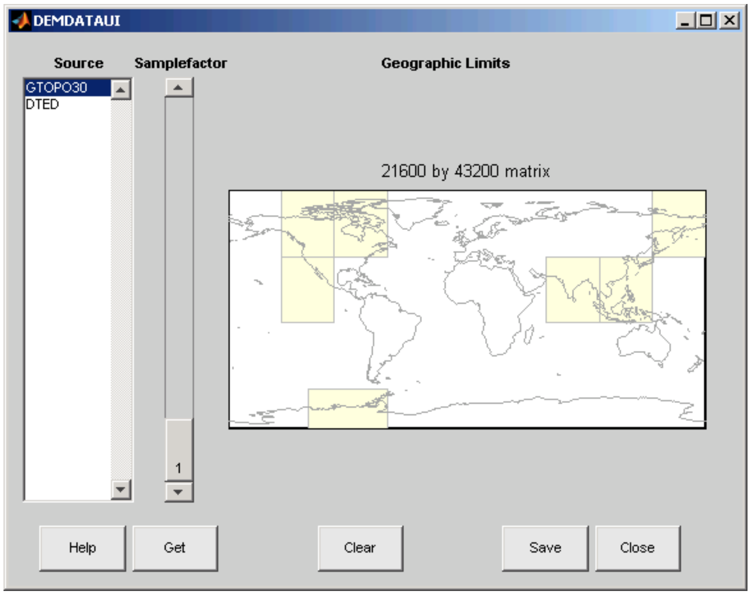
Description

`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 directory that contains subdirectories with the data files. For example, `demdataui` would detect `gtopo30` data if a directory on the path contained the directories E060S10 and E100S10, each of which holds the uncompressed data files. `globedem`: a directory that contains data files and in the subdirectory `/esri/hdr` and the `*.hdr` header files. `dted`: a directory that has a subdirectory named DTED. The contents of the DTED directory are more subdirectories 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 12-142 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

Purpose

GUI for handles of specified mapped objects

Activation

Command Line
`h = handlem`
`h = handlem('prompt')`

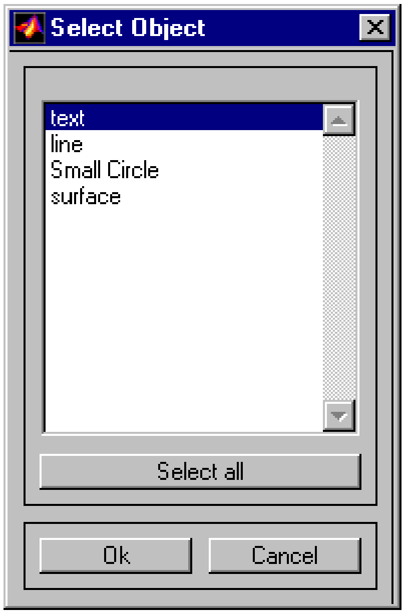
Description

`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.

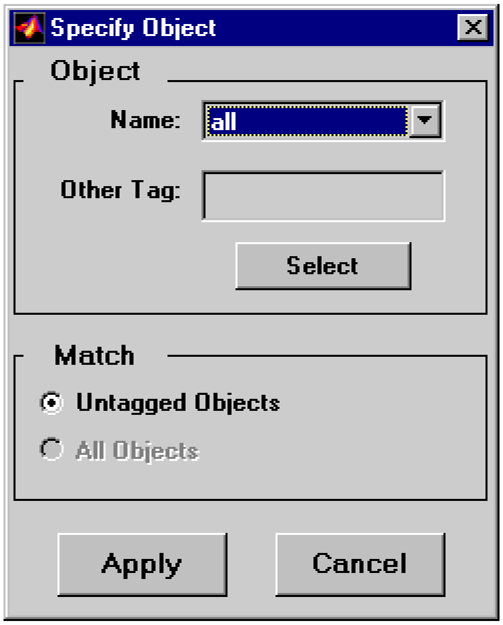
Controls

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



Name: all

Other Tag:

Untagged Objects [✓]

All Objects

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`

Purpose

Hide specified mapped objects

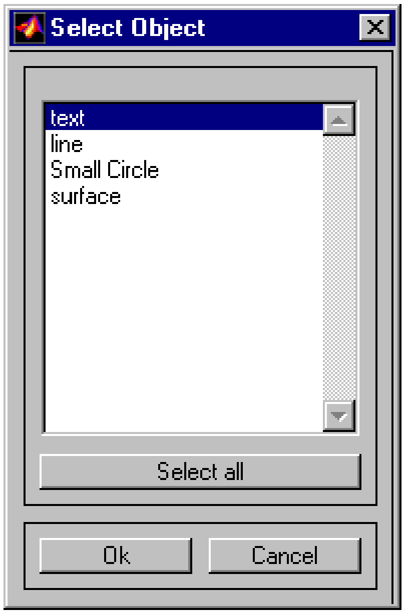
Activation

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

Purpose

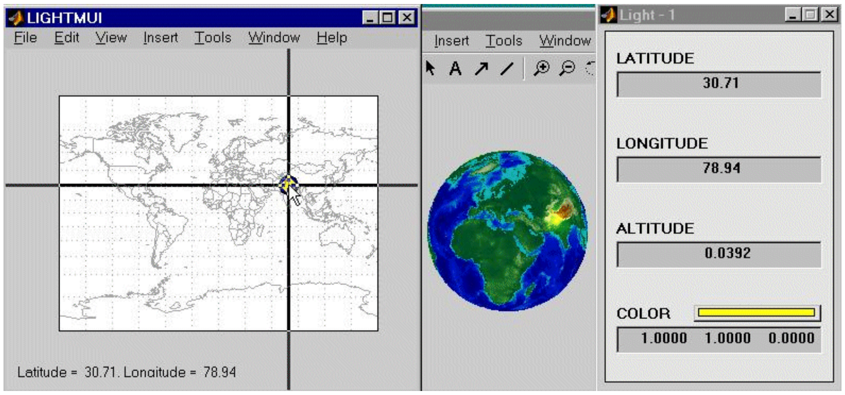
Control position of lights on globe or 3-D map

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.



Latitude = 30.71, Longitude = 78.94

LATITUDE 30.71

LONGITUDE 78.94

ALTITUDE 0.0392

COLOR 1.0000 1.0000 0.0000

See Also

`lightm`

Purpose

Add menu activated tools to map figure

Activation

Command Line
`maptool(PropertyName,PropertyValue)`
`maptool(ProjectionFile,...)`
`h = maptool(...)`

Description

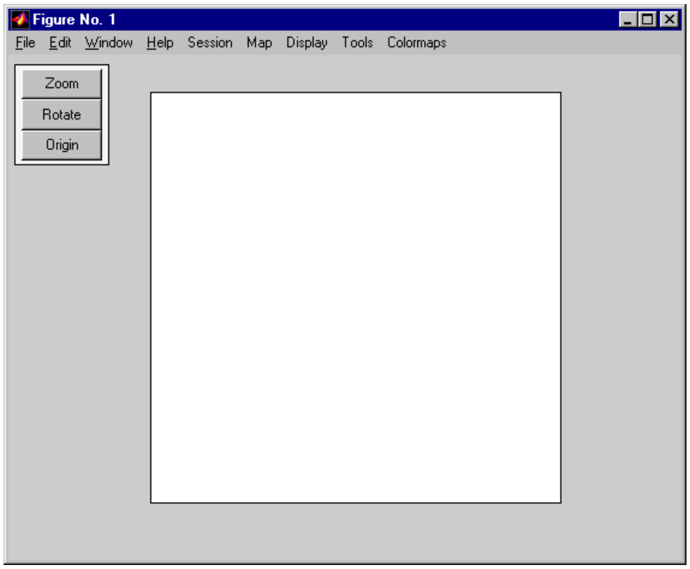
`maptool` creates a figure window with a map axes and activates the Projection Control dialog box for defining map projection and display properties. The figure window features a special menu bar that provides access to most of Mapping Toolbox GUIs.

`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



Session Menu

The **Load** option is used to load workspace data. Select from the workspace names provided, or use the **Specify Workspace** option to enter a different workspace.

The **Layers** option is used to load a map layers workspace and activate the `mlayers` tool. Select from the workspace names provided, or use the **Other** option to enter a different workspace. Choosing **Workspace** loads all structure variables in the current workspace.

The **Renderer** option is used to set the renderer for the maptool figure window. The Figure Renderer dialog box is activated when this option is selected.

The **Variables** option is used to view the current workspace variables.

The **Command** option brings up the Workspace Commands dialog box for entering commands to operate on the current workspace.

The **Clear** option is used to clear variables and functions from memory.

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 **Comet** option activates the Comet Map Input dialog box for a projecting two- or three-dimensional comet plot 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.

The **Light** option activates the Light Map Input dialog box for projecting light 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 **Zoom Tool** option is used to toggle Panzoom (`panzoom`) mode on and off. It is used for zooming in on a two-dimensional map display.

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 **Rotate** option is used to toggle Rotate 3-D (`rotate3d`) mode on and off. Rotate 3-D mode is used to interactively rotate the view of a three-dimensional plot.

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 from the **Current Object** or **Last Object** options, or choose the **Object** option to activate the Select Object dialog box.

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.

The **Zoom** button toggles Zoom mode on and off. Zoom mode is used for zooming in on a two-dimensional map display.

The **Rotate** button toggles Rotate 3-D mode on and off. Rotate 3-D mode is used to interactively rotate the view of a three-dimensional plot.

The **Origin** button toggles Origin mode on and off. Origin mode is used to interactively modify the map origin.

See Also

`axesm`

Purpose

Interactively trim and convert map data from vector to raster format

Activation

Command Line
`maptrim(lat,lon)`
`maptrim(lat,lon,`*`linespec`*`)`
`maptrim(datagrid,refvec)`
`maptrim(datagrid,refvec,`*`PropertyName`*`,PropertyValue,...)`

Description

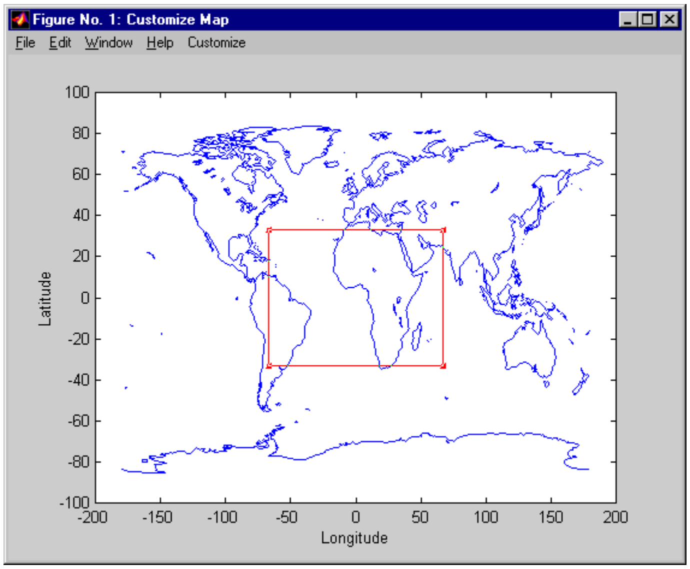
`maptrim(lat,lon)` displays the supplied map data in a new figure window and allows a region of the map to be selected and saved in the workspace. `lat` and `lon` must be vector map data. The output can be line, patch, or regular surface (matrix) data. If patch map output is selected, the inputs `lat` 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.

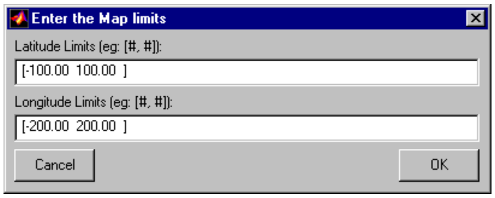
Controls



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 option toggles the `panzoom` 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.

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.



Latitude Limits (eg: [#, #]):
[-100.00 100.00]
Longitude Limits (eg: [#, #]):
[-200.00 200.00]

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`, `panzoom`

Purpose

GUI to control plotting of display structure elements

Activation

Command Line	Maptool

`mlayers('filename')`
`mlayers('filename',h)`
`mlayers(cellarray)`
`mlayers(cellarray,h)`
	Session > Layers

Description

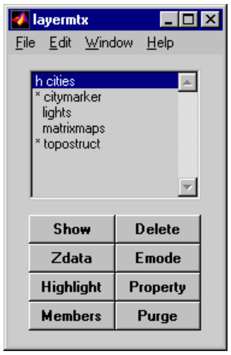
`mlayers('filename')` associates all display structures, which in this context are also called map layers, in the MAT-file `filename` with the current map axes. The display structure variables are accessible only through the `mlayers` tool, and not through the base workspace. `filename` must be a string.

`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`.

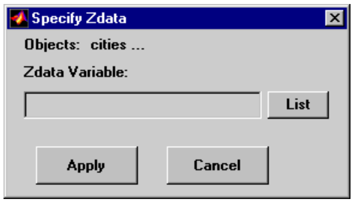
Controls



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.

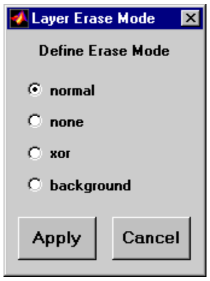


The **Highlight** button is used to toggle the selected map layer between highlighted and normal display.

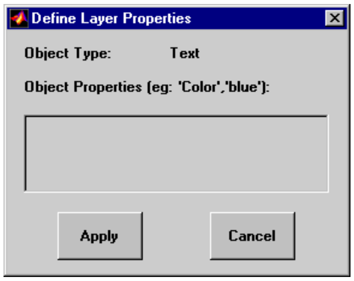
The **Members** button brings up a list of members of the selected map layer. Members of a layer are defined by their `Tag` property.

The **Delete** button deletes the selected map layer from the map.

The **Emode** button activates the Layer Erase Mode dialog box, which is used to specify the erase mode of the selected map layer.



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.



Object Type: Text

Object Properties (eg: 'Color','blue'):

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`

Purpose

Manipulate object sets displayed on map axes

Activation

Command Line	Maptool

`mobjects`
`mobjects(h)`
	Tools > Objects

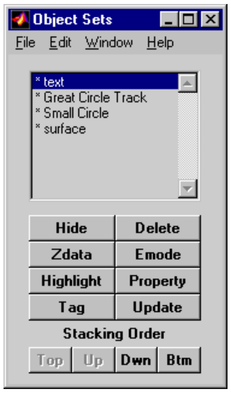
Description

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`.

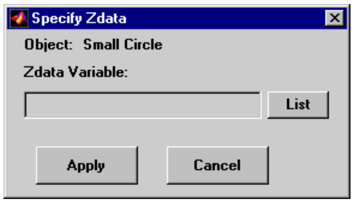
Controls



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.

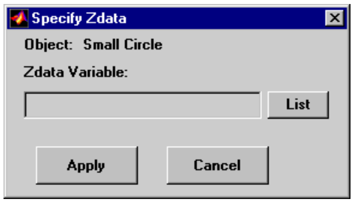


The **Highlight** button highlights all objects belonging to the selected object set.

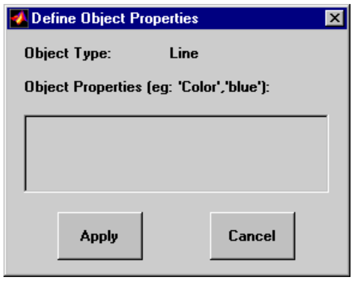
The **Tag** button brings up an Edit Tag dialog box, which allows the tag of all members of the selected object set to be modified.

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 **Emode** button activates the Object Erase Mode dialog box, which is used to specify the erase mode of the selected object set.



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

Purpose

Interactively modify map origin

Activation

Command Line	Maptool

`originui`
`originui on`
`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

Purpose

Pan and zoom on map axes

Activation

Command Line	Maptool

`panzoom`
`panzoom on`
`panzoom off`
`panzoom setlimits`
`panzoom out`
`panzoom fullview`
	Tools > Zoom Tool (menu) > Zoom (button)

Description

`panzoom` toggles the pan and zoom tool on and off.

`panzoom on` activates the pan and zoom tool.

`panzoom off` deactivates the pan and zoom tool.

`panzoom setlimits` sets the zoom out limits to the current settings on the map axes.

`panzoom out` zooms out to the current map axes limit settings.

`panzoom fullview` resets the axes to their full view range and resets the pan and zoom tool with these settings.

The pan and zoom tool provides an interactive means of defining zoom limits on a two-dimensional map display. A box that can be resized and moved appears on the map display and is used to define the zoom area. The box cannot be moved beyond the current axes limits.

Controls

Mouse Interaction

With the cursor inside the zoom box, a single-click and drag moves the box. The zoom box can be resized by dragging the corners of the box. A double-click in the center of the box zooms in to the current boundaries of the box. A single-click outside the zoom box moves the box to that location. An extend-click inside or outside of the zoom box zooms out by a factor of two. Alternate-click exits the pan and zoom tool.

Keyboard Interaction

The following keyboard interaction is enabled if the figure containing the map axes is made the active window.

Pressing the **Return** key sets the axes to the current zoom box and remains in pan and zoom mode. The **Enter** key sets the axes to the current zoom box and exits pan and zoom mode. Pressing the **Esc** or **Delete** keys exits pan and zoom mode.

See Also

`zoom`

parallelui

Purpose

Interactively modify map parallels

Activation

Command Line	Maptool

`parallelui`
`parallelui on`
`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`

Purpose

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

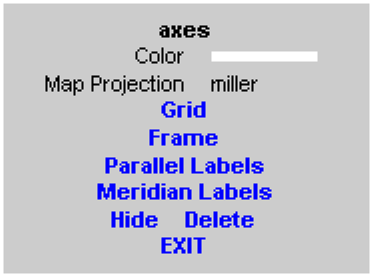
Alternate (e.g., **Ctrl**+clicking a mapped object activates a property editor, which allows modification of some basic properties of the object through simple mouse clicks and drags. The objects supported by this editor are map axes, lines, text, patches, and surfaces, and the properties supported for 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

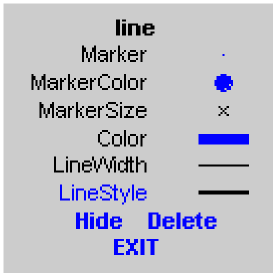
Click-and-Drag Property Editor

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'`.



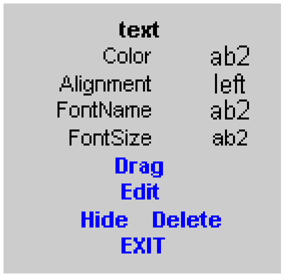
Click-and-Drag Editor for a map axes

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.



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.

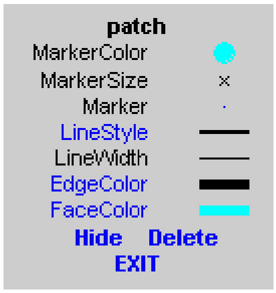


text

Color	ab2
Alignment	left
FontName	ab2
FontSize	ab2

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

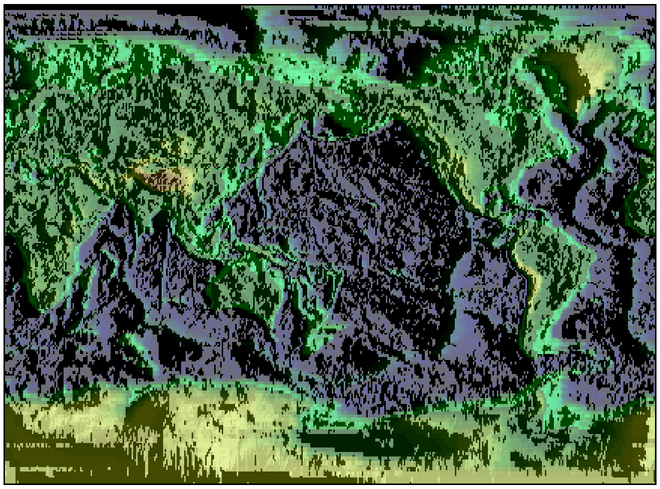
Click-and-Drag Editor for a patch object

The **Graticule** control on the surface editor activates a Graticule Mesh dialog box, which is used to alter the size of the graticule.

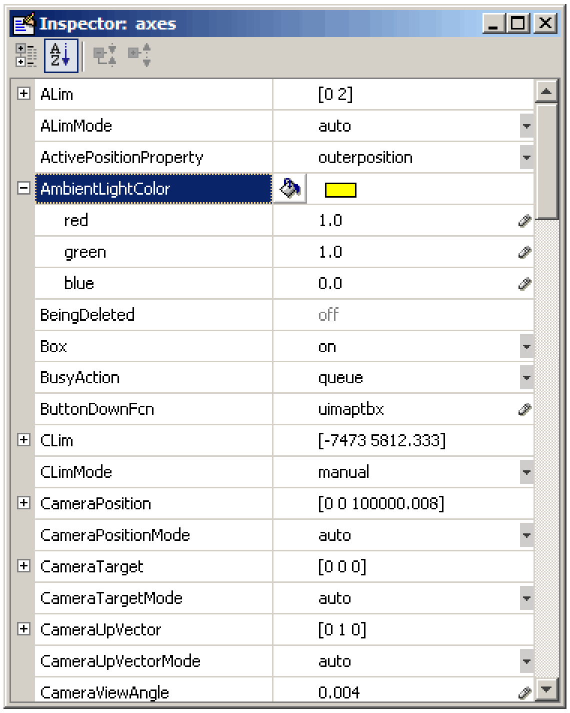
To move the property editor around the figure window, hold down the **Shift** key while dragging the editor box. Alternate-clicking the background of the property editor closes the **Click-and-Drag** editing session.

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



Inspector: axes

ALim	[0 2]
ALimMode	auto
ActivePositionProperty	outerposition
AmbientLightColor	🟨
red	1.0
green	1.0
blue	0.0
BeingDeleted	off
Box	on
BusyAction	queue
ButtonDownFcn	uimaptbx
CLim	[-7473 5812.333]
CLimMode	manual
CameraPosition	[0 0 100000.008]
CameraPositionMode	auto
CameraTarget	[0 0 0]
CameraTargetMode	auto
CameraUpVector	[0 1 0]
CameraUpVectorMode	auto
CameraViewAngle	0.004

Property Inspector view of axes object

See Also

`propedit`, `inspect`, `uimaptbx`

qrydata

Purpose

GUI to interactively perform data queries

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.

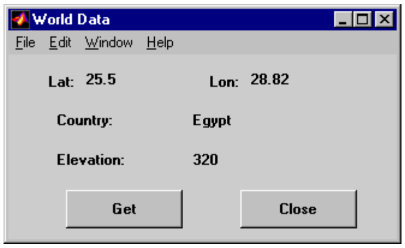
The `'mapmatrix'` query interpolates to find the value of the map at the input `(lat,lon)` point. The format of `cellarray` for this query is: `{'label','mapmatrix',datagrid,refvec,`*`method`*`}`. `datagrid` and `refvec` are the data grid and the corresponding referencing vector. *`method`* is an optional 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 an M-file of the form `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



Lat: 25.5 Lon: 28.82
Country: Egypt
Elevation: 320

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

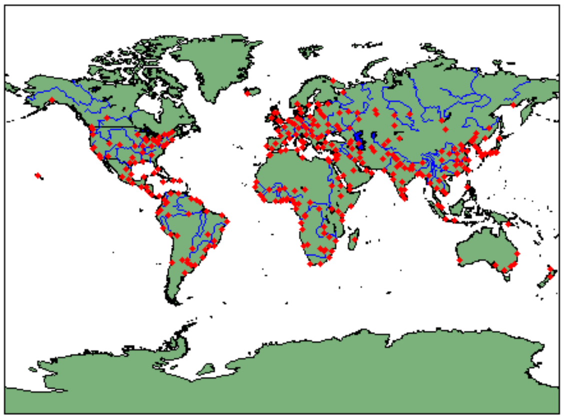
This example illustrates use of a user-defined query to display city names for map points specified by a mouse click. The query is evaluated by a user-supplied M-file called qrytest.m, described below:

```
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 = strvcat(cities.Name);
qrydata(gca,'City Data',{'City','qrytest',lat,lon,mat})
```

Create the M-file `qrytest` 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`

Purpose

GUI to display small circles on map axes

Note `scirclui` is obsolete. Use `scircleg` instead.

Activation

Command Line	Maptool

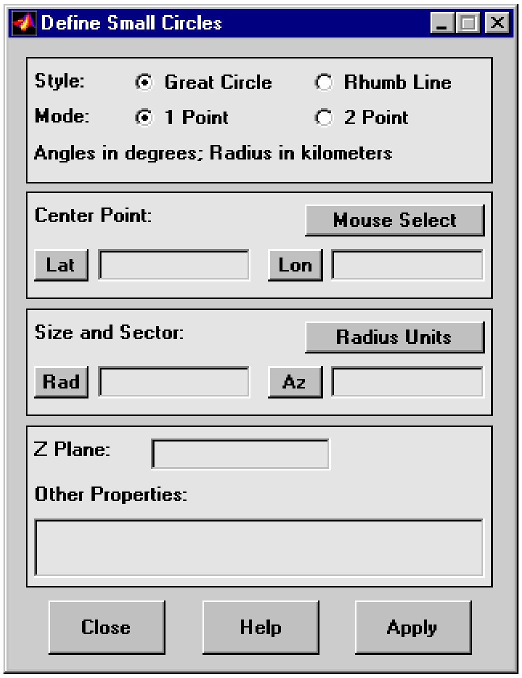
`scirclui`
`scirclui(h)`
	Display mall Circles

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.

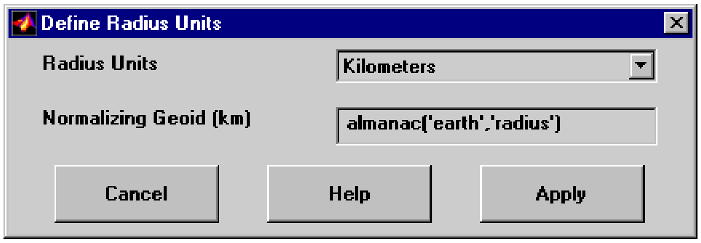
The **Size and Sector** controls are used only in one-point mode. The **Radius Units** button brings up a Define Radius Units dialog box, which allows for modification of the small circle radius units and the normalizing geoid. The **Rad** edit box is used to enter the radius of the small circle in the 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.



Kilometers

almanac('earth','radius')

Define Radius Units 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`

Purpose

GUI to fill data grids with seeded values

Activation

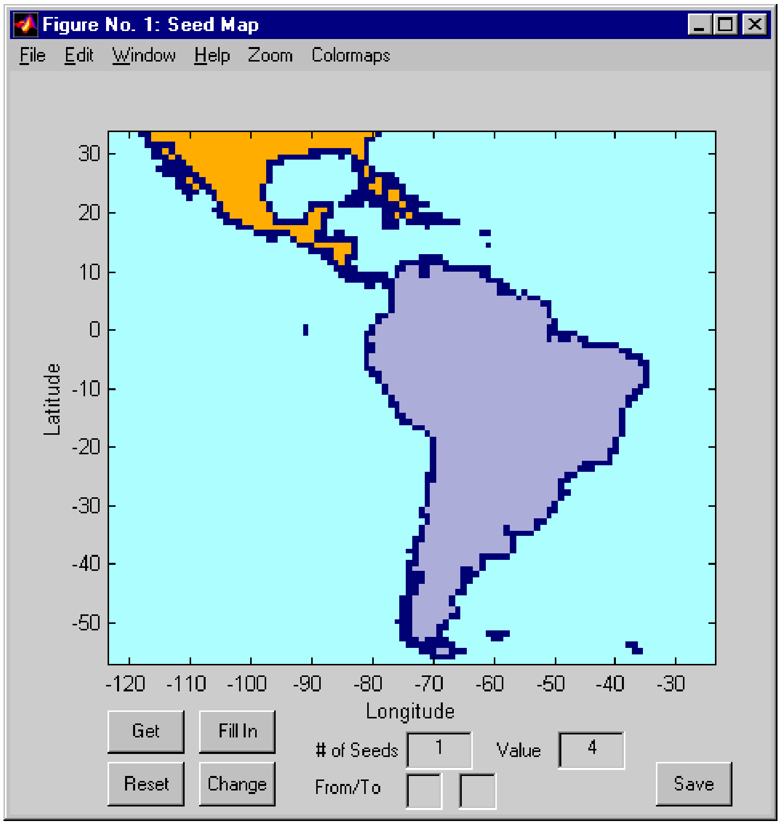
Command Line
`seedm(datagrid,refvec)`

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

Purpose

Show specified mapped objects

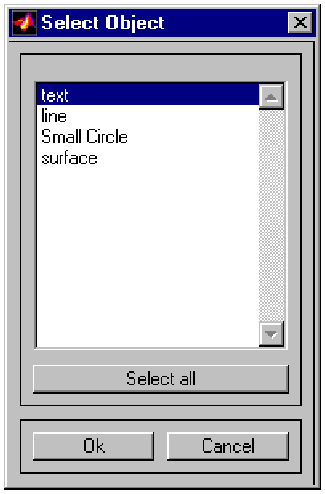
Activation

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`

Purpose

Interactive distance, azimuth, and reckoning calculations

Activation

Command Line	Maptool

`surfdist`
`surfdist(h)`
`surfdist([])`
	Display > Surface > Distances

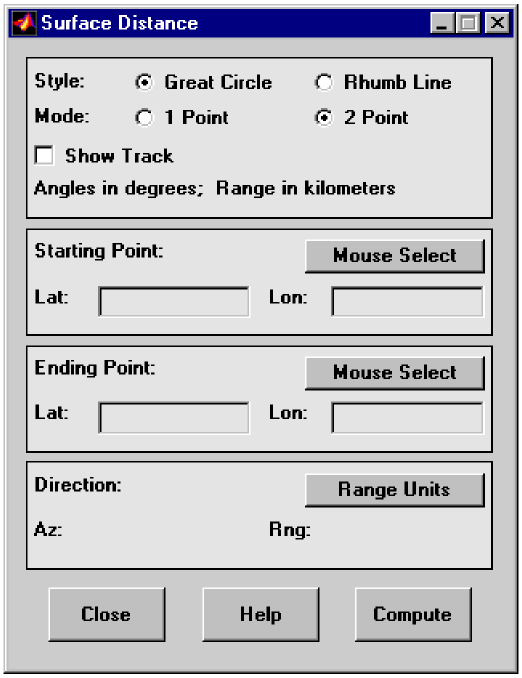
Description

`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.

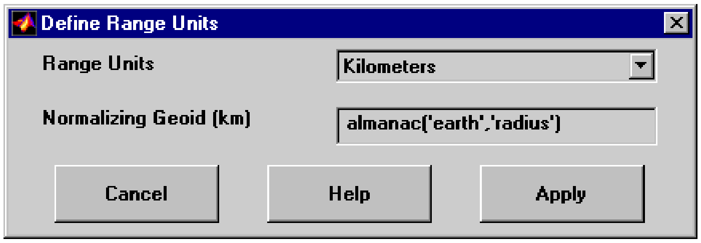
The **Direction** controls are enabled only for one-point mode. The **Range Units** button brings up a Define Range Units dialog box which allows for modification of the range units and the normalizing geoid. The **Az** edit box is used to enter the azimuth, which sets the initial direction of the track 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.



Range Units: Kilometers

Normalizing Geoid (km): almanac('earth','radius')

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.

Purpose

GUI to edit tag property of mapped object

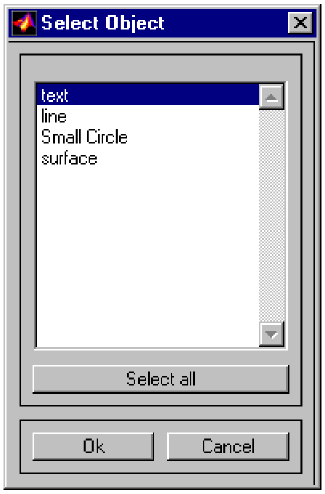
Activation

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.

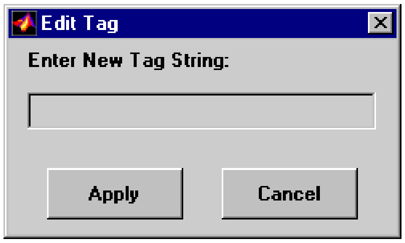
`tagm(h)` activates the Edit Tag dialog box for the objects specified by the handle `h`.



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`

Purpose

GUI to display great circles and rhumb lines on map axes

Note `trackui` is obsolete. Use `trackg` instead.

Activation

Command Line	Maptool

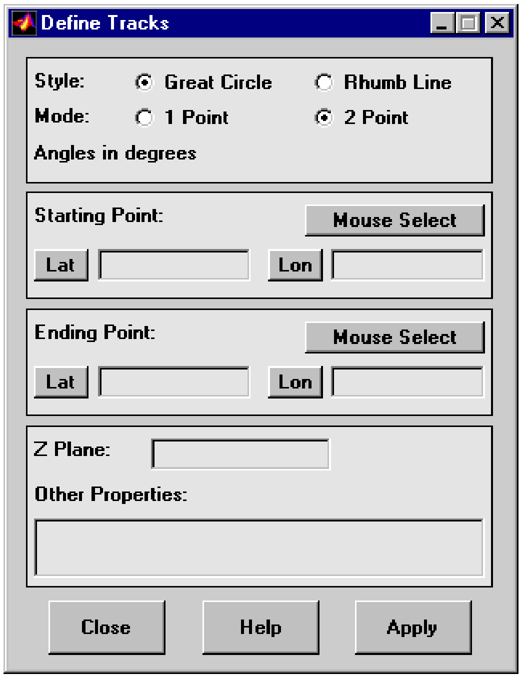
`trackui`
`trackui(h)`
	Display > Tracks

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.

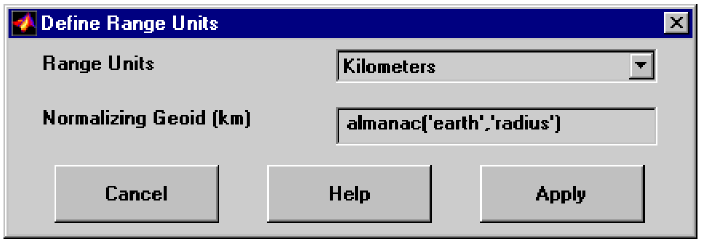
The **Direction** controls are used only for one-point mode. The **Range Units** button brings up a Define Range Units dialog box, which allows for modification of the range units and the normalizing geoid. The **Az** edit box is used to enter the azimuth, which sets the initial direction of the track 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`

Purpose

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

Purpose

Choose or identify UTM zone by clicking map

Activation

Command Line
`utmzoneui`
`utmzoneui(InitZone)`

Description

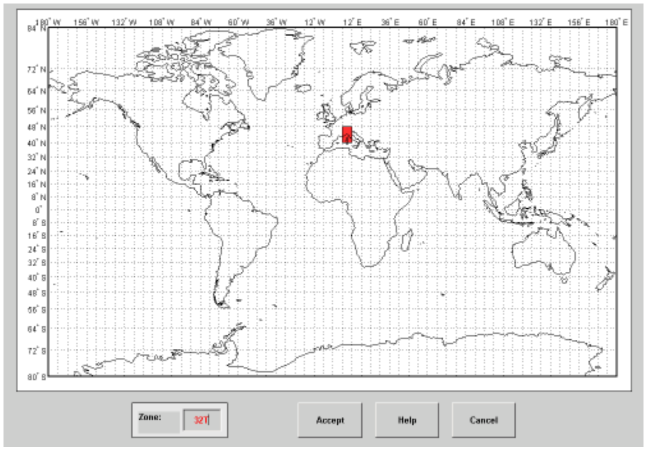
`zone = utmzoneui` will create an interface for choosing a UTM zone on a world display map. It allows for clicking an area for its appropriate zone, or entering a valid zone to identify the zone on the map.

`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



Remarks

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

`ups`	Universal Polar Stereographic (UPS) Projection.
`utm`	Universal Transverse Mercator (UTM) Projection.
`utmgeoid`	Select ellipsoid for a given UTM zone.
`utmzone`	Select a UTM zone.

vmap0ui

Purpose

UI for selecting data from Vector Map Level 0

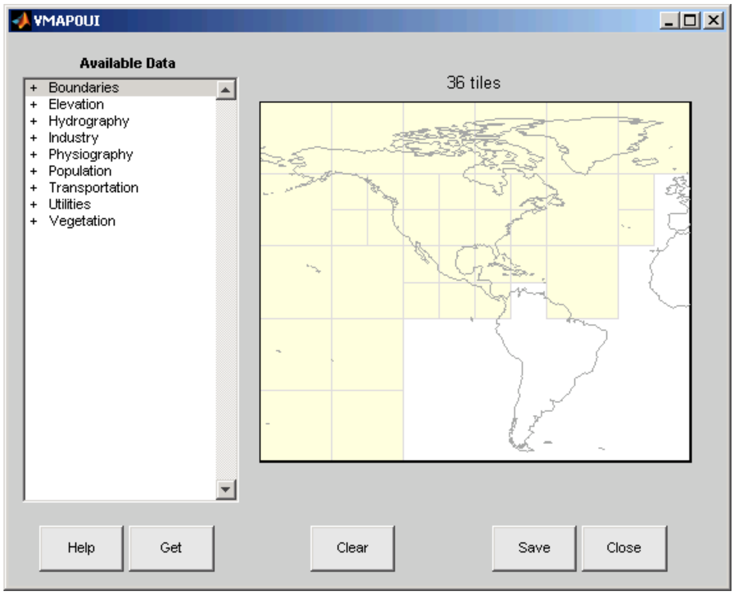
Description

`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 directory 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 `vmap0ui` 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. `vmap0ui` 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 12-142 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

Purpose

GUI to adjust *z*-plane of mapped objects

Activation

Command Line
`zdatam`
`zdatam(h)`
`zdatam(`*`str`*`)`

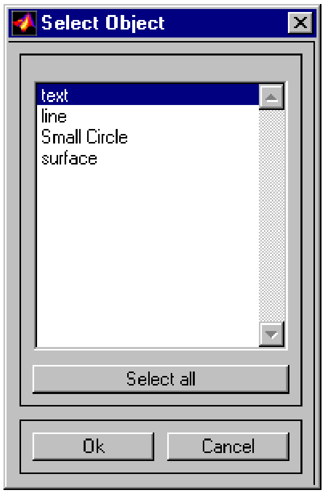
Description

`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`.

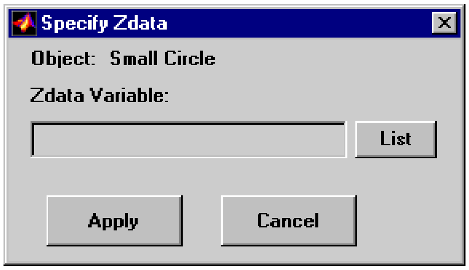
`zdatam(`*`str`*`)` activates the Specify Zdata dialog box for the objects identified by *`str`*, where *`str`* is any string recognized by `handlem`.

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`

Class Reference

Web Map Service

In this section...

WebMapServer

getCapabilities	Get capabilities document from server
getMap	Get raster map from server
updateLayers	Update layer properties
WebMapServer	Web map server object

WMSCapabilities

disp	Display properties
WMSCapabilities	Web Map Service capabilities object

WMSLayer

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
WMSLayer	Web Map Service layer object

WMSMapRequest

boundImageSize	Bound size of raster map
WMSMapRequest	Web Map Service map request object

Map Projections Reference

See Chapter 8, "Using Map Projections and Coordinate Systems" for a general discussion of map projections, and "Summary and Guide to Projections" on page 8-63 for a tabular comparison of their properties.

Cylindrical Projections

`balthsrt`	Balthasart Projection
`behrmann`	Behrmann Projection
`bsam`	Bolshoi Sovietskii Atlas Mira Projection
`braun`	Braun Perspective Projection
`cassini`	Cassini Projection
`cassinistd`	Cassini Projection — Standard
`ccylin`	Central Cylindrical Projection
`eqacylin`	Equal Area Projection
`edqcylin`	Equidistant Projection
`giso`	Gall Isographic Projection
`gortho`	Gall Orthographic Projection
`gstereo`	Gall Stereographic Projection
`lambcyln`	Lambert Projection
`mercator`	Mercator Projection
`miller`	Miller Projection
`pcarree`	Plate Carree Projection
`tranmerc`	Transverse Mercator Projection
`trystan`	Trystan Edwards Projection
`wetch`	Wetch Projection

Pseudocylindrical Projections

`apianus`	Apianus II Projection
`collig`	Collignon Projection
`craster`	Craster Parabolic Projection

`eckert1`	Eckert I Projection
`eckert2`	Eckert II Projection
`eckert3`	Eckert III Projection
`eckert4`	Eckert IV Projection
`eckert5`	Eckert V Projection
`eckert6`	EckertVI Projection
`flatplrp`	Flat-Polar Parabolic Projection
`flatplrq`	Flat-Polar Quartic Projection
`flatplrs`	Flat-Polar Sinusoidal Projection
`fournier`	Fournier Projection
`goode`	Goode Homolosine Projection
`hatano`	Hatano Assymmetrical Equal Area Projection
`kavrsky5`	Kavraisky V Projection
`kavrsky6`	Kavraisky VI Projection
`loximuth`	Loximuthal Projection
`modsine`	Modified Sinusoidal Projection
`mollweid`	Mollweide Projection
`putnins5`	Putnins P5 Projection
`quartic`	Quartic Authalic Projection
`robinson`	Robinson Projection
`sinusoid`	Sinusoidal Projection
`wagner4`	Wsgner IV Projection
`winkel`	Winkel I Projection

Conic Projections

`eqaconic`	Albers Equal Area Conic Projection
`eqaconicstd`	Albers Equal Conic Projection — Standard
`eqdconic`	Equidistant Conic Projection
`eqdconicstd`	Equidistant Conic Projection — Standard
`lambert`	Lambert Conformal Conic Projection
`lambertstd`	Lambert Conformal Conic Projection — Standard
`murdoch1`	Murdoch I Conic Projection
`murdoch3`	Murdoch III Minimum Error Conic Projection

Polyconic and Pseudoconic Projections

`bonne`	Bonne Projection
`polycon`	Polyconic Projection
`polyconstd`	Polyconic Projection — Standard
`vgrint1`	Van Der Grinten I Projection
`werner`	Werner Projection

Azimuthal, Pseudoazimuthal, and Modified Azimuthal Projections

`aitoff`	Aitoff Projection
`breusing`	Breusing Harmonic Mean Projection
`bries`	Briesemeister Projection
`eqaazim`	Lambert Equal Area Azimuthal Projection

`eqdazim`	Equidistant Azimuthal Projection
`gnomonic`	Gnomonic Azimuthal Projection
`hammer`	Hammer Projection
`ortho`	Orthographic Azimuthal Projection
`stereo`	Stereographic Azimuthal Projection
`vperspec`	Vertical Perspective Azimuthal Projection
`wiechel`	Wiechel Equal Area Projection

UTM and UPS Systems

`ups`	Universal Polar Stereographic (UPS) system
`utm`	Universal Transverse Mercator (UTM) system

3-D Globe Display

`globe`	Earth as sphere in 3–D graphics

Map Projections — Alphabetical List

Aitoff Projection

Classification

Modified Azimuthal

Syntax

`aitoff`

Graticule

Meridians: Central meridian is a straight line half the length of the Equator. Other meridians are complex curves, equally spaced along the Equator, and concave toward the central meridian.

Parallels: Equator is straight. Other parallels are complex curves, equally spaced along the central meridian, and concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is neither conformal nor equal area. The only point free of distortion is the center point. Distortion of shape and area are moderate throughout. This projection has less angular distortion on the outer meridians near the poles than pseudoazimuthal projections

Parallels

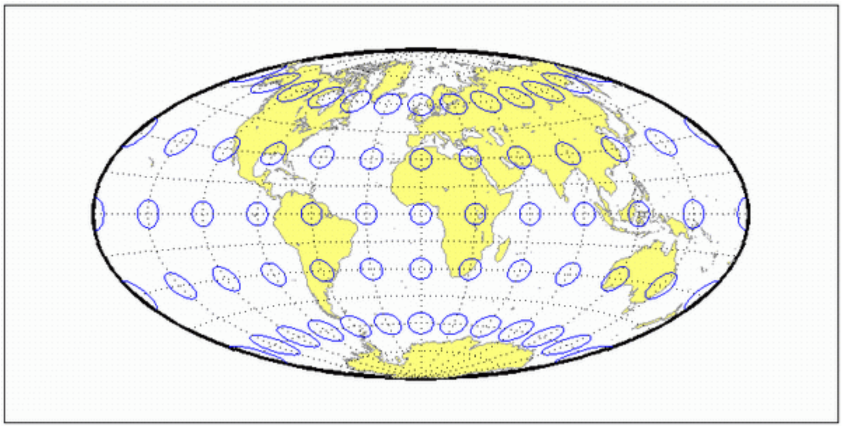
There is no standard parallel for this projection.

Remarks

This projection was created by David Aitoff in 1889. It is a modification of the Equidistant Azimuthal projection. The Aitoff projection inspired the similar Hammer projection, which is equal area.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('aitoff', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Albers Equal-Area Conic Projection

Classification Conic

Syntax

```
eqaconic
eqaconic
```

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the point of convergence. Spacing of parallels decreases away from the central latitudes.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

This is an equal-area projection. Scale is true along the one or two selected standard parallels. Scale is constant along any parallel; the scale factor of a meridian at any given point is the reciprocal of that along the parallel to preserve equal-area. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is neither conformal nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane and a Lambert Azimuthal Equal-Area projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Equal-Area Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Equal-Area Cylindrical projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

Remarks

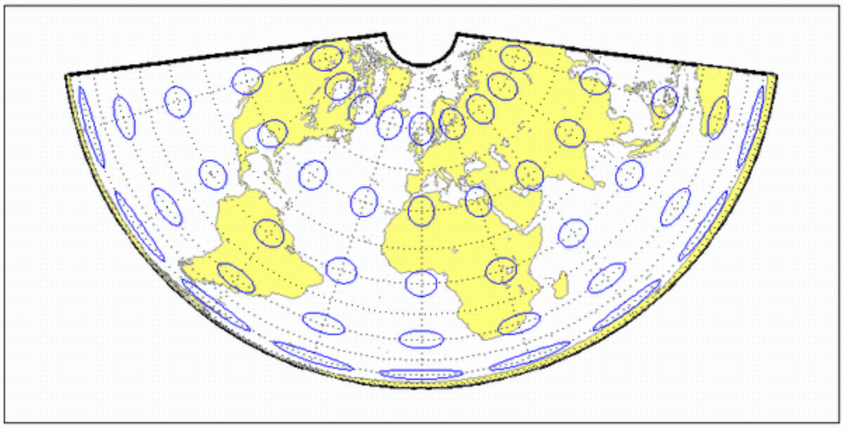
This projection was presented by Heinrich Christian Albers in 1805.

Limitations

Longitude data greater than 135° east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqaconic', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`eqaconicstd`

Albers Equal-Area Conic Projection — Standard

Classification

Conic

Syntax

`eqaconicstd`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the point of convergence. Spacing of parallels decreases away from the central latitudes.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

This function implements the Albers Equal Area Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `eqaconic` for an alternative implementation based on rotating the authalic sphere.

This is an equal area projection. Scale is true along the one or two selected standard parallels. Scale is constant along any parallel; the scale factor of a meridian at any given point is the reciprocal of that along the parallel to preserve equal area. The projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is neither conformal nor equidistant.

Parallels

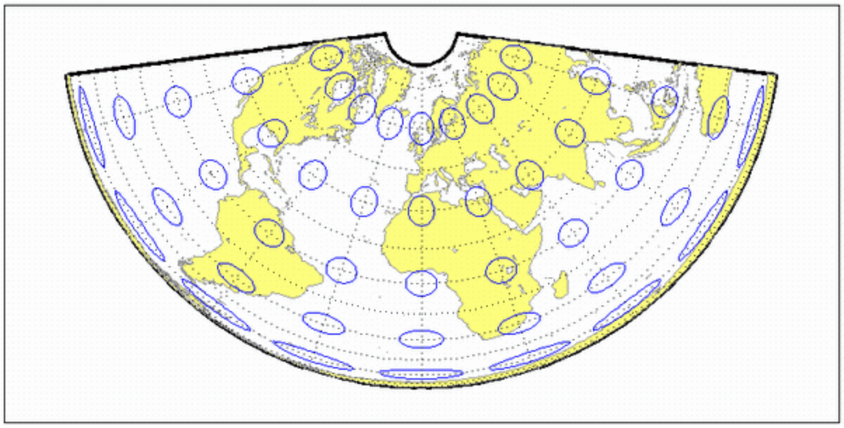
The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane and a Lambert Azimuthal Equal-Area projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Equal-Area Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Equal-Area Cylindrical projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

Remarks

This projection was presented by Heinrich Christian Albers in 1805 and it is also known as a Conical Orthomorphic projection. The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Lambert Equal Area Conic projection is the result. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Cylindrical Equal Area Projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other cylindrical equal area projection is the result.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqaconicstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See also

`eqaconic`

Apianus II Projection

Classification

Pseudocylindrical

Syntax

`apianus`

Graticule

Meridians: Equally spaced elliptical curves converging at the poles.

Parallels: Equally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

Scale is constant along any parallel or pair of parallels equidistant from the Equator, as well as along the central meridian. The Equator is free of angular distortion. This projection is not equal-area, equidistant, or conformal.

Parallels

There is no standard parallel for this projection.

Remarks

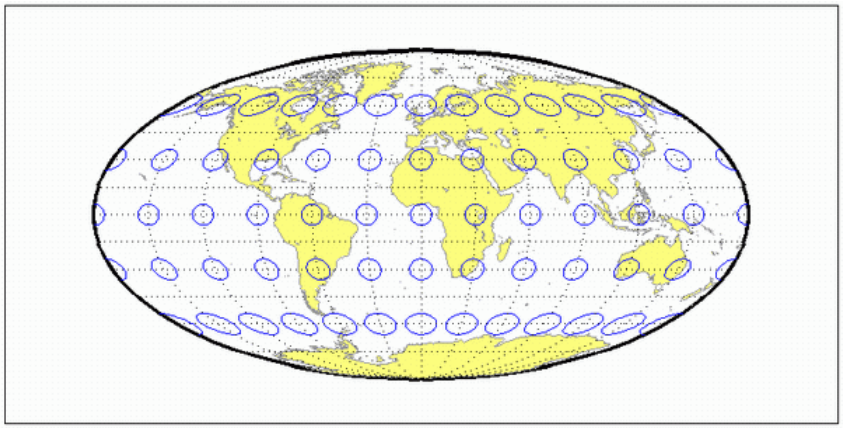
This projection was first described in 1524 by Peter Apian (or Bienewitz).

Limitations

This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('apianus', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Balthasart Cylindrical Projection

Classification

Cylindrical

Syntax

`balthsrt`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 50º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

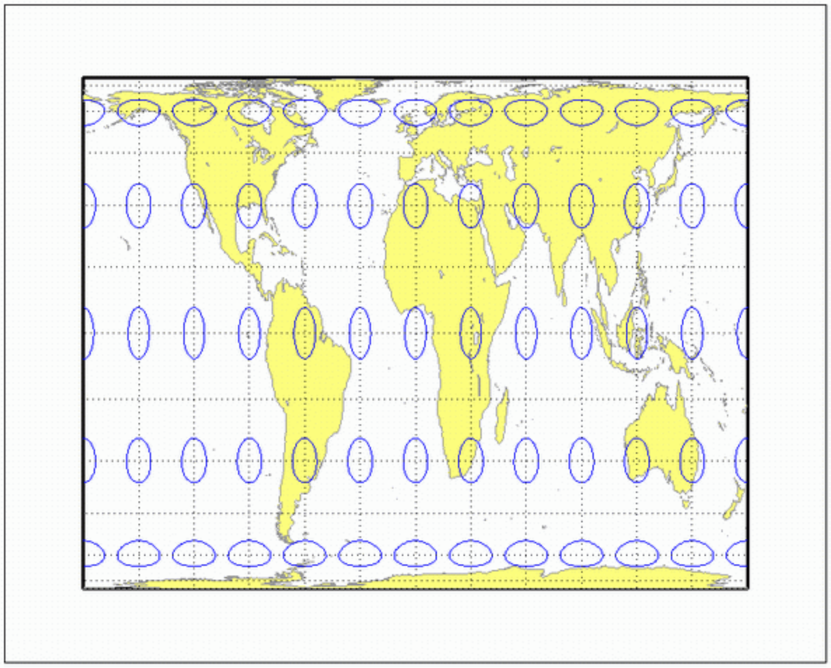
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 50º.

Remarks

The Balthasart Cylindrical projection was presented in 1935 and is a special form of the Equal-Area Cylindrical projection secant at 50ºN and S.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('balthsrt', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Behrmann Cylindrical Projection

Classification Cylindrical

Syntax `behrmann`

Graticule Meridians: Equally spaced straight parallel lines 0.42 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

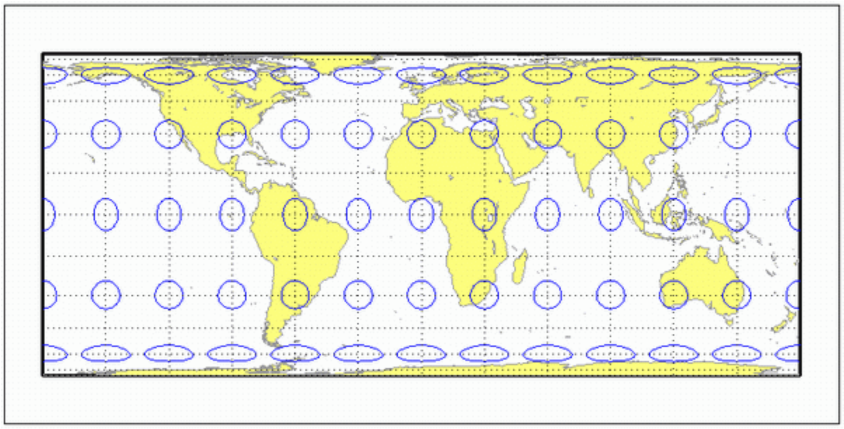
Features This is an orthographic projection onto a cylinder secant at the 30º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 30º.

Remarks This projection is named for Walter Behrmann, who presented it in 1910 and is a special form of the Equal-Area Cylindrical projection secant at 30ºN and S.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('behrmann', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Bolshoi Sovietskii Atlas Mira Projection

Classification Cylindrical

Syntax `bsam`

Graticule Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features This is a perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at the 30º parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

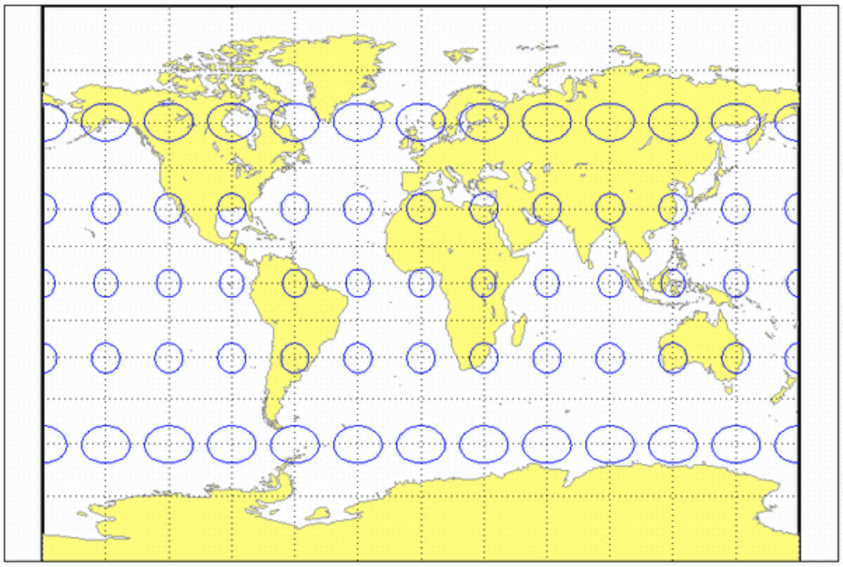
Parallels For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 30º.

Remarks This projection was first described in 1937, when it was used for maps in the *Bolshoi Sovietskii Atlas Mira* (Great Soviet World Atlas). It is commonly abbreviated as the BSAM projection. It is a special form of the Braun Perspective Cylindrical projection secant at 30ºN and S.

Limitations This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('bsam', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Bonne Projection

Classification

Pseudoconic

Syntax

`bonne`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves connecting points equally spaced along each parallel and concave toward the central meridian.

Parallels: Concentric circular arcs spaced at true distances along the central meridian.

Poles: Points.

Symmetry: About the central meridian.

Features

This is an equal-area projection. The curvature of the standard parallel is identical to that on a cone tangent at that latitude. The central meridian and the central parallel are free of distortion. This projection is not conformal.

Parallels

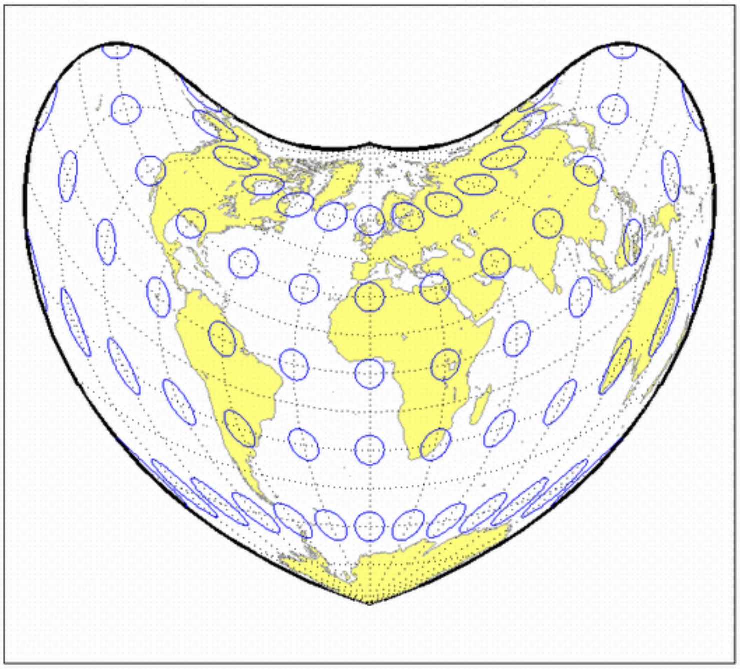
This projection has one standard parallel, which is 30ºN by default. It has two interesting limiting forms. If a pole is employed as the standard parallel, a Werner projection results; if the Equator is used, a Sinusoidal projection results.

Remarks

This projection dates in a rudimentary form back to Claudius Ptolemy (about A.D. 100). It was further developed by Bernardus Sylvanus in 1511. It derives its name from its considerable use by Rigobert Bonne, especially in 1752.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('bonne', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Braun Perspective Cylindrical Projection

Classification

Cylindrical

Syntax

`braun`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at standard parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude may be chosen; the default is arbitrarily set to 0º.

Remarks

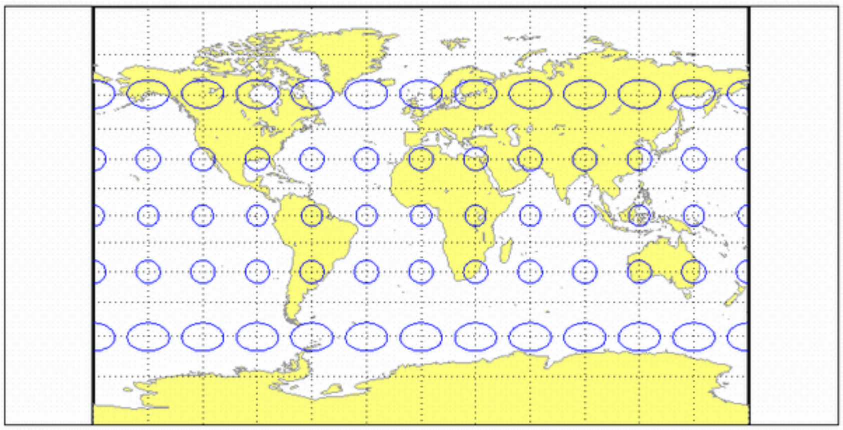
This projection was first described by Braun in 1867. It is less well known than the specific forms of it called the Gall Stereographic and the *Bolshoi Sovietskii Atlas Mira* projections.

Limitations

This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('braun', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Breusing Harmonic Mean Projection

Classification Azimuthal

Syntax `breusing`

Graticule The graticule described is for the polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole.

Parallels: Unequally spaced circles centered on the central pole. The opposite hemisphere cannot be shown. Spacing increases (slightly) away from the central pole.

Poles: The central pole is a point, while the opposite pole cannot be shown.

Symmetry: About any meridian.

Features This is a harmonic mean between a Stereographic and Lambert Equal-Area Azimuthal projection. It is not equal-area, equidistant, or conformal. There is no point at which scale is accurate in all directions. The primary feature of this projection is that it is minimum error—distortion is moderate throughout.

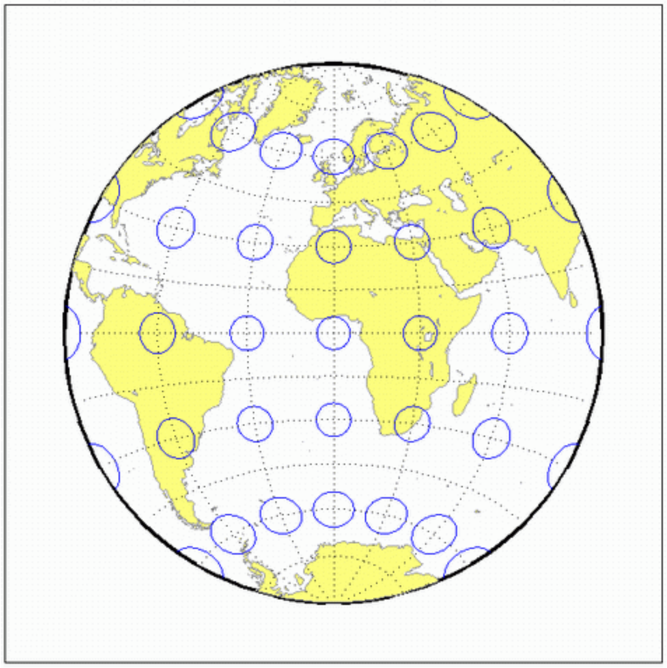
Parallels There are no standard parallels for azimuthal projections.

Remarks F. A. Arthur Breusing developed a geometric mean version of this projection in 1892. A. E. Young modified this to the harmonic mean version presented here in 1920. This projection is virtually indistinguishable from the Airy Minimum Error Azimuthal projection, presented by George Airy in 1861.

Limitations This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('breusing', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Briesemeister Projection

Classification Modified Azimuthal

Syntax `bries`

Graticule Meridians: Central meridian is straight. Other meridians are complex curves.

Parallels: Complex curves.

Poles: Points.

Symmetry: About the central meridian.

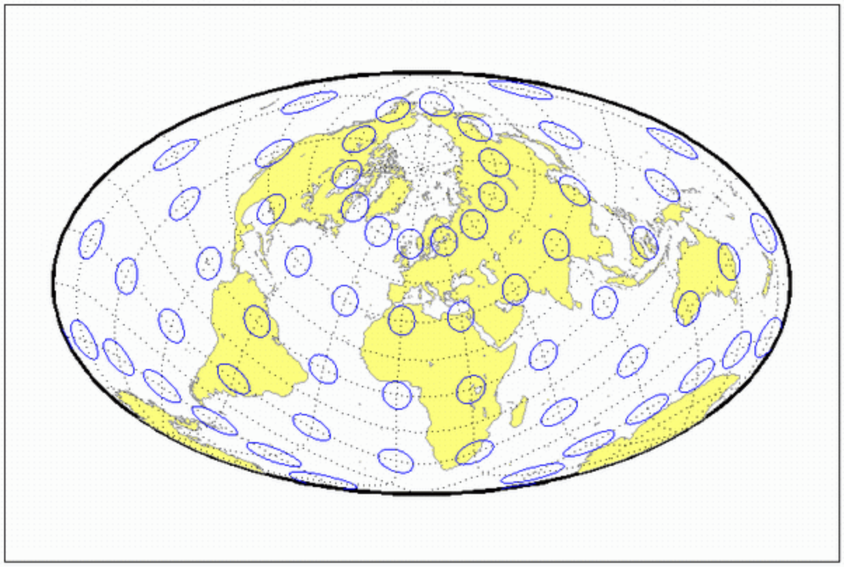
Features This equal-area projection groups the continents about the center of the projection. The only point free of distortion is the center point. Distortion of shape and area are moderate throughout.

Parallels There is no standard parallel for this projection.

Remarks This projection was presented by William Briesemeister in 1953. It is an oblique Hammer projection with an axis ratio of 1.75 to 1, instead of 2 to 1.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('bries', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Cassini Cylindrical Projection

Classification

Cylindrical

Syntax

`cassini`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90° from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the central meridian. Distortion of both shape and area are functions of distance from the central meridian. Scale is true along the central meridian and along any straight line perpendicular to the central meridian (i.e., it is equidistant).

Parallels

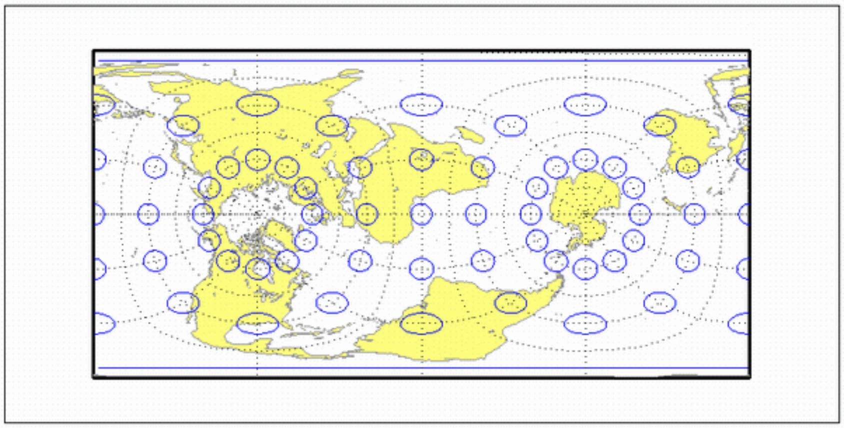
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel *of the base projection* is by definition fixed at 0°.

Remarks

This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('cassini', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See also `cassinistd`

Cassini Cylindrical Projection — Standard

Syntax

`cassinistd`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90º from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the central meridian. Distortion of both shape and area are functions of distance from the central meridian. Scale is true along the central meridian and along any straight line perpendicular to the central meridian (i.e., it is equidistant).

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel *of the base projection* is by definition fixed at 0º.

Remarks

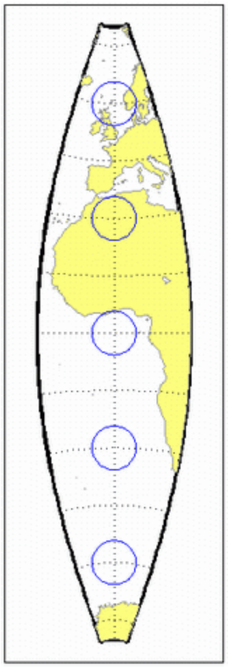
This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.

`cassinistd` implements the Cassini projection directly on a sphere or reference ellipsoid, as opposed to using the equidistant cylindrical projection in tranverse mode as in function `cassini`. Distinct forms are used for the sphere and ellipsoid, because approximations in the ellipsoidal formulation cause it to be appropriate only within a zone that extends 3 or 4 degrees in longitude on either side of the central meridian.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('cassinistd', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See also `cassini`

Central Cylindrical Projection

Classification Cylindrical

Syntax `ccylin`

Graticule Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles, more rapidly than that of the Mercator projection.

Poles: Cannot be shown.

Symmetry: About any meridian or the Equator.

Features This is a perspective projection from the center of the Earth onto a cylinder tangent at the Equator. It is not equal-area, equidistant, or conformal. Scale is true along the Equator and constant between two parallels equidistant from the Equator. Scale becomes infinite at the poles. There is no distortion along the Equator, but it increases rapidly away from the Equator.

Parallels For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

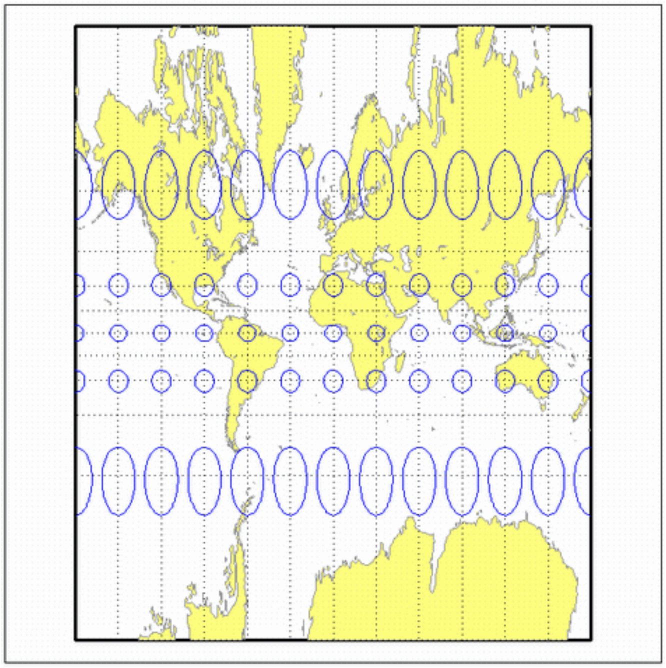
Remarks The origin of this projection is unknown; it has little use beyond the educational aspects of its method of projection and as a comparison to the Mercator projection, which is not perspective. The transverse aspect of the Central Cylindrical is called the Wetch projection.

Limitations This projection is available only on the sphere. Data at latitudes greater than 75º is trimmed to prevent large values from dominating the display.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('ccylin', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
```

```
tissot;
```



Collignon Projection

Classification Pseudocylindrical

Syntax `collig`

Graticule Meridians: Equally spaced straight lines converging at the North Pole.

Parallels: Unequally spaced straight parallel lines, farthest apart near the North Pole, closest near the South Pole

Poles: North Pole is a point, South Pole is a line 1.41 as long as the Equator.

Symmetry: About the central meridian.

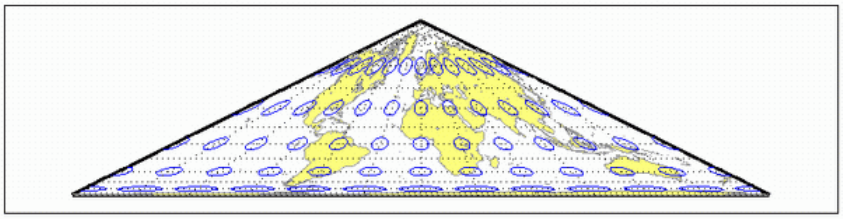
Features This is a novelty projection showing a straight-line, equal-area graticule. Scale is true along the 15°51'N parallel, constant along any parallel, and *different* for any pair of parallels. Distortion is severe in many regions, and is only absent at 15°51'N on the central meridian. This projection is not conformal or equidistant.

Parallels This projection has one standard parallel, which is by definition fixed at 15°51'.

Remarks This projection was presented by Édouard Collignon in 1865.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('collig', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification Pseudocylindrical

Syntax `craster`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced parabolas intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes very gradually and is greatest near the Equator.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 36º46’ parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than the Sinusoidal projection. This projection is free of distortion only at the two points where the central meridian intersects the 36º46’ parallels. This projection is not conformal or equidistant.

Parallels

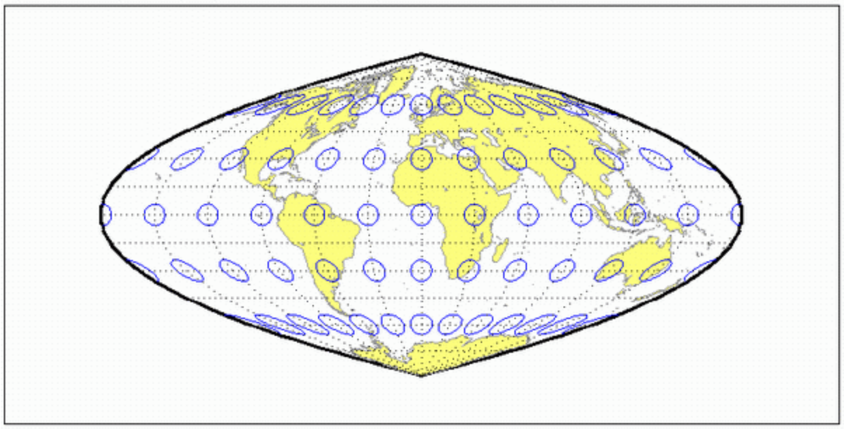
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 36º46’.

Remarks

This projection was developed by John Evelyn Edmund Craster in 1929; it was further developed by Charles H. Deetz and O.S. Adams in 1934. It was presented independently in 1934 by Putnins as his P_4 projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('craster', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Eckert I Projection

Classification

Pseudocylindrical

Syntax

```
eckert1
```

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced straight converging lines broken at the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 47º10' parallels and is constant along any parallel, between any pair of parallels equidistant from the Equator, and along any given meridian. It is not free of distortion at any point, and the break at the Equator introduces excessive distortion there; regardless of the appearance here, the Tissot indicatrices are of indeterminate shape along the Equator. This novelty projection is not equal-area or conformal.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 47º10'.

Remarks

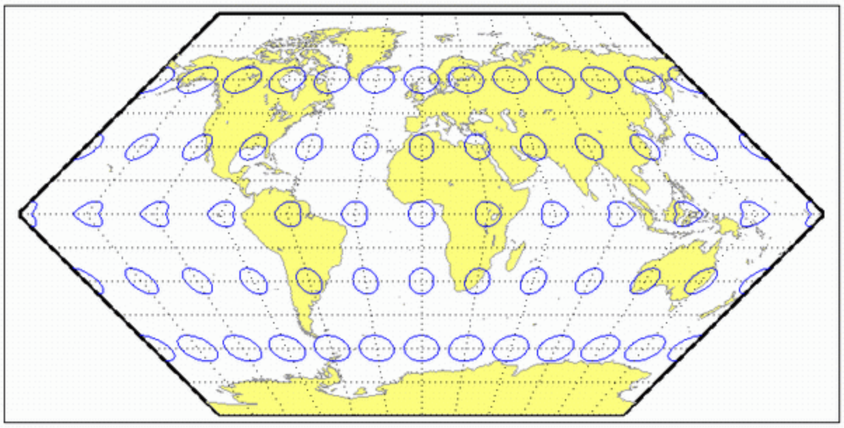
This projection was presented by Max Eckert in 1906.

Limitations

This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Eckert II Projection

Classification

Pseudocylindrical

Syntax

`eckert2`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced straight converging lines broken at the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 55º10' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is not free of distortion at any point except at 55º10'N and S along the central meridian; the break at the Equator introduces excessive distortion there. Regardless of the appearance here, the Tissot indicatrices are of indeterminate shape along the Equator. This novelty projection is not conformal or equidistant.

Parallels

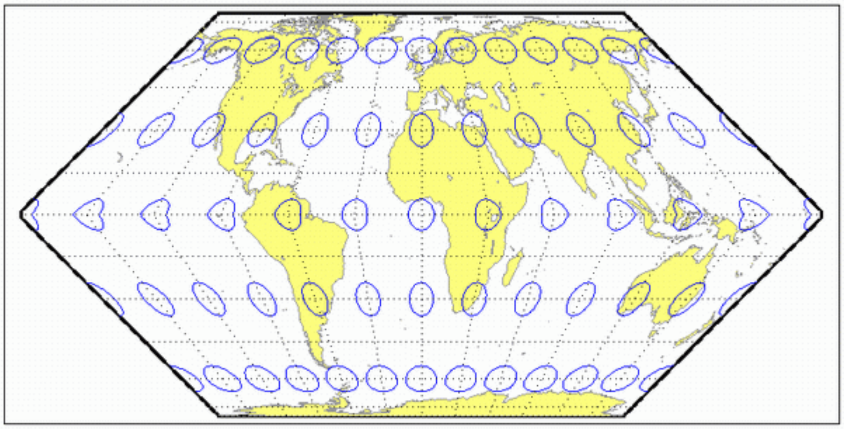
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 55º10'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert2', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Eckert III Projection

Classification

Pseudocylindrical

Syntax

`eckert3`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180º east and west of the central meridian, are semicircles.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 35º58' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. No point is free of all scale distortion, but the Equator is free of angular distortion. This projection is not equal-area, conformal, or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 35º58'.

Remarks

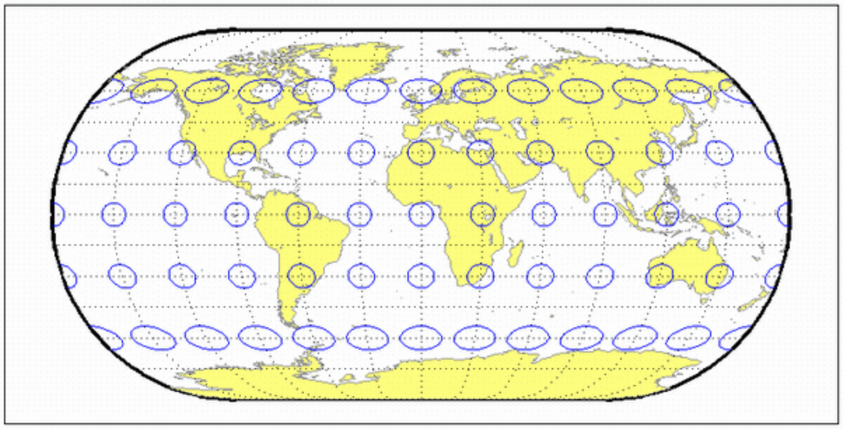
This projection was presented by Max Eckert in 1906.

Limitations

This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert3', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Eckert IV Projection

Classification

Pseudocylindrical

Syntax

`eckert4`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180º east and west of the central meridian, are semicircles.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 40º30' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 40º30' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

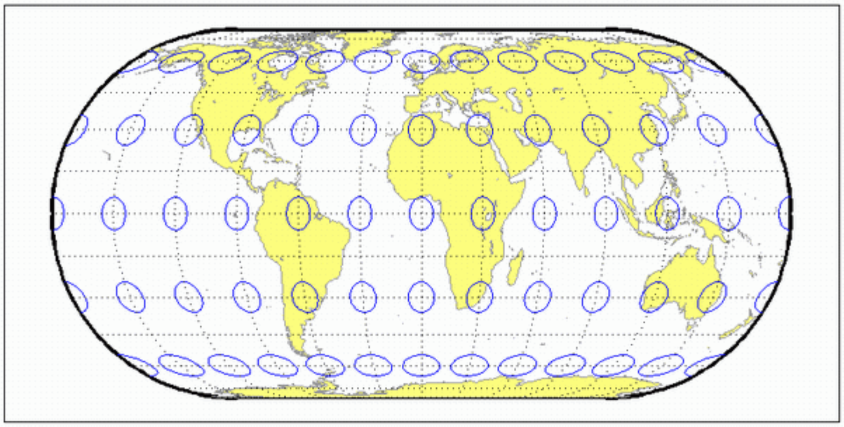
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 40º30'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert4', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Eckert V Projection

Classification

Pseudocylindrical

Syntax

```
eckert5
eckert5
```

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This projection is an arithmetic average of the *x* and *y* coordinates of the Sinusoidal and Plate Carrée projections. Scale is true along latitudes 37º55'N and S, and is constant along any parallel and between any pair of parallels equidistant from the Equator. There is no point free of all distortion, but the Equator is free of angular distortion. This projection is not equal-area, conformal, or equidistant.

Parallels

This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

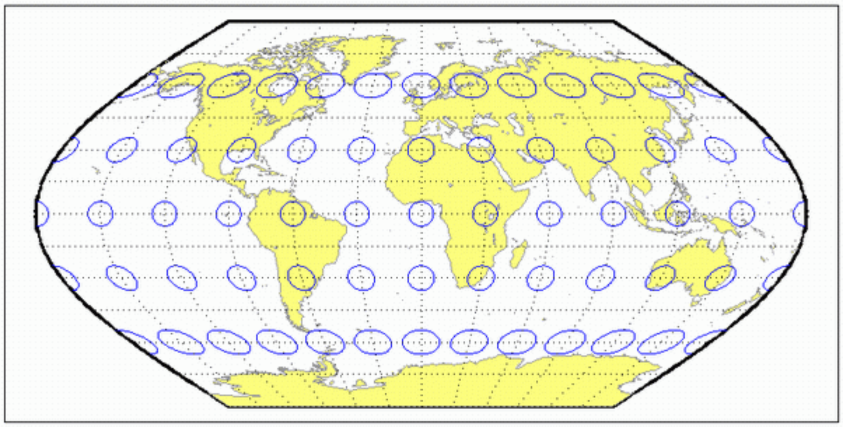
This projection was presented by Max Eckert in 1906.

Limitations

This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert5', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification Pseudocylindrical

Syntax `eckert6`

Graticule Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

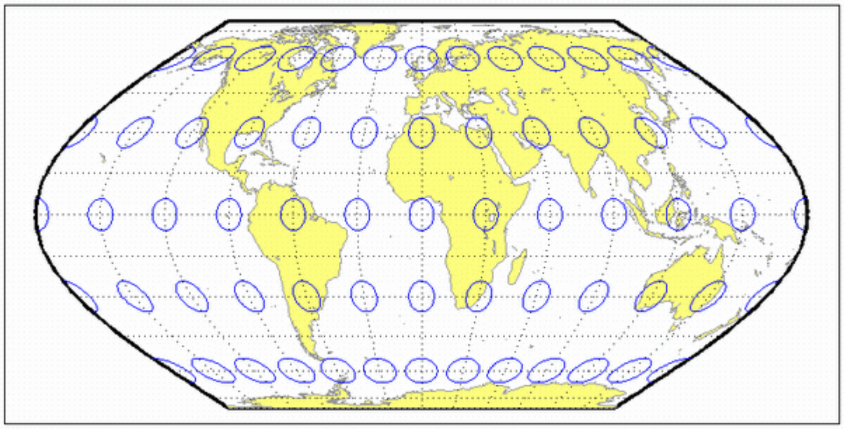
Features This is an equal-area projection. Scale is true along the 49º16’ parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 49º16’ parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 49º16’.

Remarks This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert6', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Equal-Area Cylindrical Projection

Classification

Cylindrical

Syntax

`eqacylin`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the standard parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

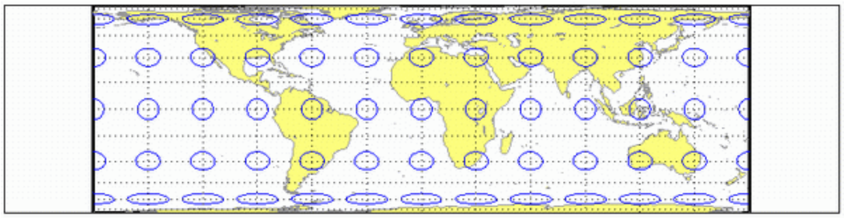
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude may be chosen; the default is arbitrarily set to 0º (the Lambert variation).

Remarks

This projection was proposed by Johann Heinrich Lambert (1772), a prolific cartographer who proposed seven different important projections. The form of this projection tangent at the Equator is often called the Lambert Equal-Area Cylindrical projection. That and other special forms of this projection are included separately in this guide, including the Gall Orthographic, the Behrmann Cylindrical, the Balthasart Cylindrical, and the Trystan Edwards Cylindrical projections.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqacylin', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Equidistant Azimuthal Projection

Classification Azimuthal

Syntax `eqdazim`

Graticule The graticule described is for the polar aspect.

Meridians: Equally spaced straight lines intersecting at a central pole. The angles between them are the true angles.

Parallels: Equally spaced circles, centered on the central pole. The entire Earth may be shown.

Poles: Central pole is a point. The opposite pole is a bounding circle with a radius twice that of the Equator.

Symmetry: About any meridian.

Features This is an equidistant projection. It is neither equal-area nor conformal. In the polar aspect, scale is true along any meridian. The projection is distortion free only at the center point. Distortion is moderate for the inner hemisphere, but it becomes extreme in the outer hemisphere.

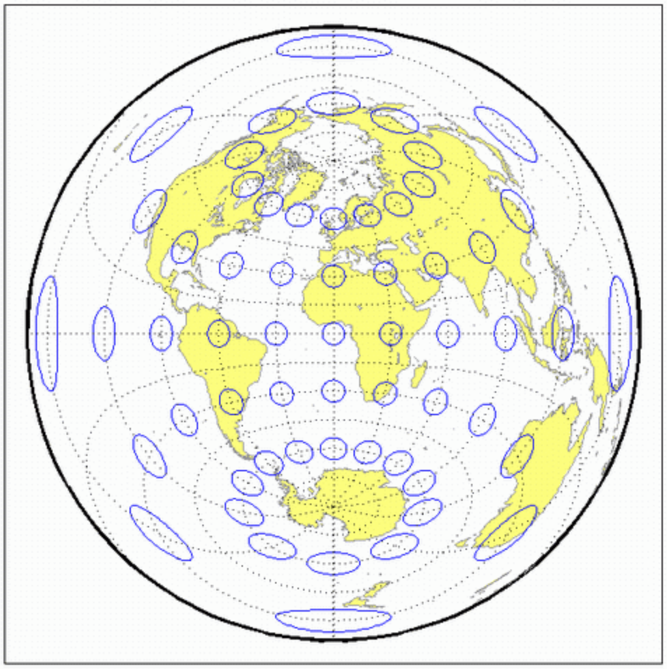
Parallels There are no standard parallels for azimuthal projections.

Remarks This projection may have been first used by the ancient Egyptians for star charts. Several cartographers used it during the sixteenth century, including Guillaume Postel, who used it in 1581. Other names for this projection include Postel and Zenithal Equidistant.

Limitations This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdazim', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification

Conic

Syntax

`eqdconic`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Equally spaced concentric circular arcs centered on the point of meridanal convergence.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

Scale is true along each meridian and the one or two selected standard parallels. Scale is constant along any parallel. This projection is free of distortion along the two standard parallels. Distortion is constant along any other parallel. This projection provides a compromise in distortion between conformal and equal-area conic projections, of which it is neither.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and an Equidistant Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then an Equidistant Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is so chosen, the cone becomes a cylinder and a Plate Carrée projection results. If two parallels equidistant from the Equator are chosen as the standard parallels, an Equidistant Cylindrical projection results. The default parallels are [15 75].

Remarks

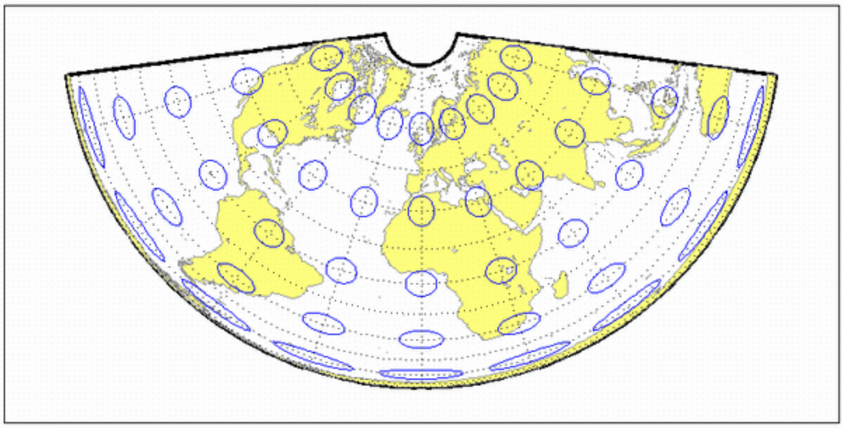
In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.

Limitations

Longitude data greater than 135° east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdconic', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`eqdconicstd`

Equidistant Conic Projection — Standard

Syntax

`eqdconicstd`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Equally spaced concentric circular arcs centered on the point of meridanal convergence.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

`eqdconicstd` implements the Equidistant Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `eqdconic` for an alternative implementation based on rotating the rectifying sphere.

Scale is true along each meridian and the one or two selected standard parallels. Scale is constant along any parallel. This projection is free of distortion along the two standard parallels. Distortion is constant along any other parallel. This projection provides a compromise in distortion between conformal and equal-area conic projections, of which it is neither.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and an Equidistant Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then an Equidistant Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is so chosen, the cone becomes a cylinder and a Plate Carrée projection results. If two parallels equidistant from the Equator are chosen as the standard parallels, an Equidistant Cylindrical projection results. The default parallels are [15 75].

Remarks

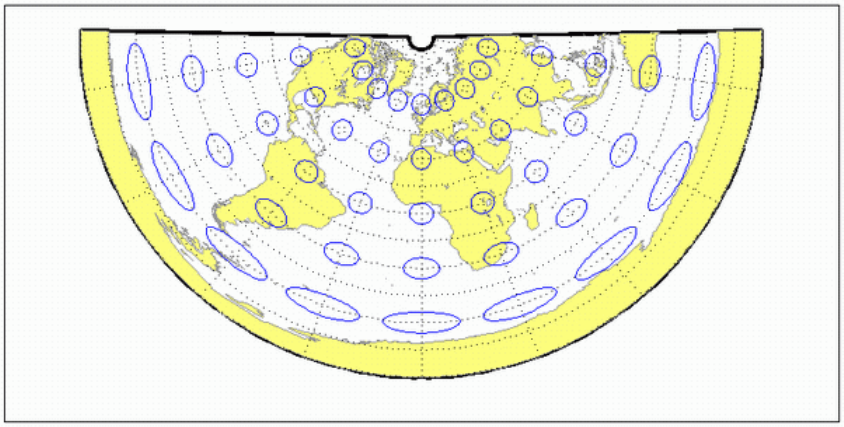
In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.

Limitations

Longitude data greater than 135º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdconicstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`eqdconic`

Equidistant Cylindrical Projection

Classification

Cylindrical

Syntax

`eqdcylin`

Graticule

Meridians: Equally spaced straight parallel lines more than half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder secant at the standard parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the standard parallels and is constant along any parallel and along the parallel of opposite sign.

Parallels

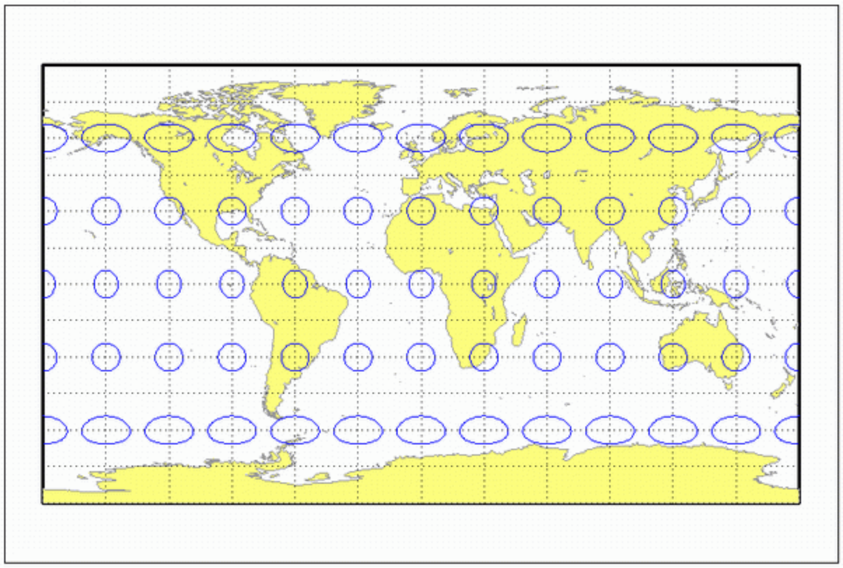
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude can be chosen; the default is arbitrarily set to 30º.

Remarks

This projection was first used by Marinus of Tyre about A.D. 100. Special forms of this projection are the Plate Carrée, with a standard parallel at 0º, and the Gall Isographic, with standard parallels at 45ºN and S. Other names for this projection include Equirectangular, Rectangular, Projection of Marinus, *La Carte Parallélogrammatique*, and *Die Rechteckige Plattkarte*.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdcylin', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Fournier Projection

Classification

Pseudocylindrical

Syntax

`fournier`

Graticule

Meridians: Equally spaced elliptical curves converging at the poles.

Parallels: Straight lines.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is equal-area. Scale is constant along any parallel or pair of parallels equidistant from the Equator. This projection is neither equidistant nor conformal.

Parallels

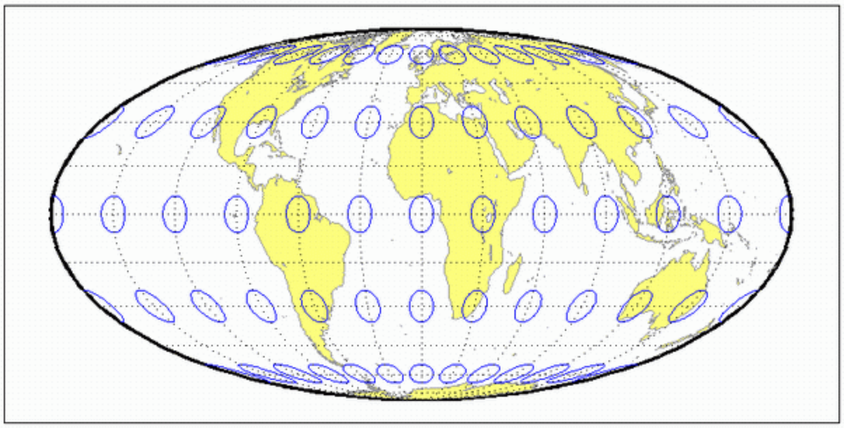
There is no standard parallel for this projection.

Remarks

This projection was first described in 1643 by Georges Fournier. This is actually his second projection, the Fournier II.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('fournier', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Gall Isographic Projection

Classification

Cylindrical

Syntax

`giso`

Graticule

Meridians: Equally spaced straight parallel lines more than half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder secant at the 45° parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the two standard parallels, and is constant along any parallel and along the parallel of opposite sign.

Parallels

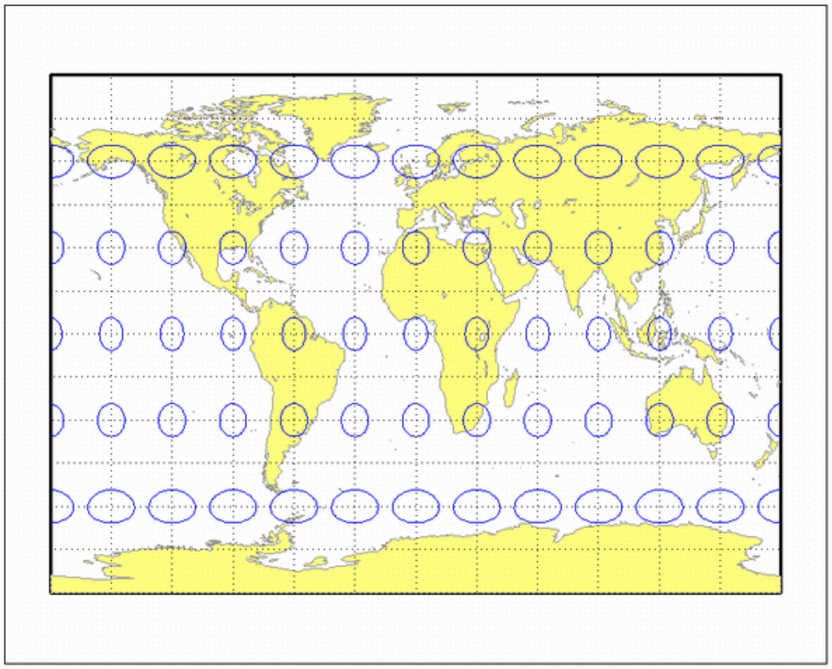
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45°.

Remarks

This projection is a specific case of the Equidistant Cylindrical projection, with standard parallels at 45°N and S.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('giso', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification

Cylindrical

Syntax

`gortho`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 45º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

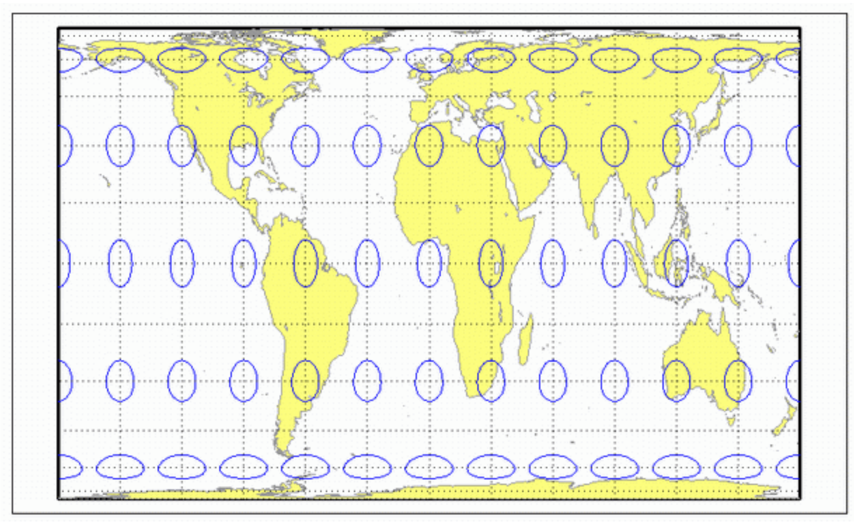
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

Remarks

This projection is named for James Gall, who originated it in 1855 and is a special form of the Equal-Area Cylindrical projection secant at 45ºN and S. This projection is also known as the Peters projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('gortho', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification Cylindrical

Syntax `gstereo`

Graticule

Meridians: Equally spaced straight parallel lines 0.77 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at the 45º parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

Remarks

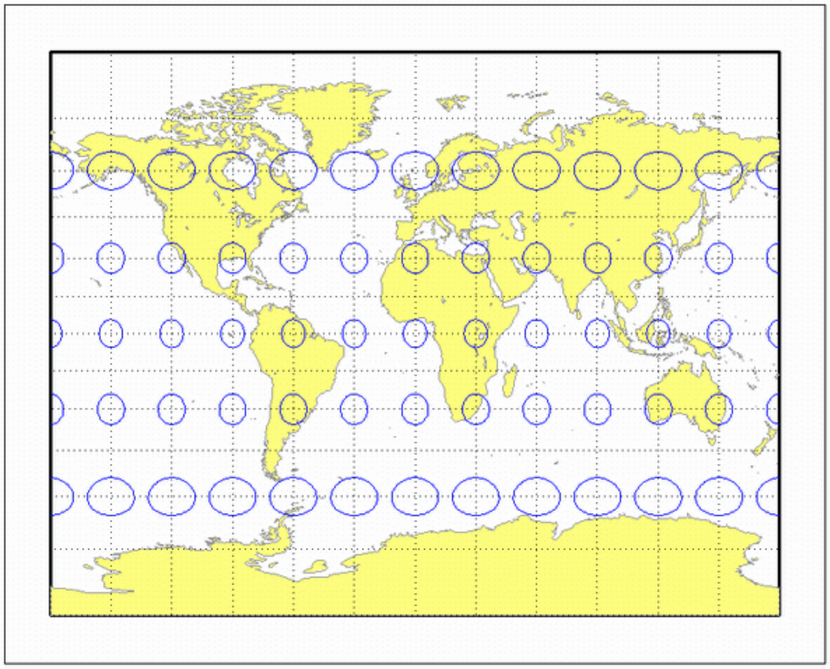
This projection was presented by James Gall in 1855. It is also known simply as the Gall projection. It is a special form of the Braun Perspective Cylindrical projection secant at 45ºN and S.

Limitations

This projection is available only on the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('gstereo', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Globe

Classification

Spherical

Syntax

`globe`

Graticule

This map display is not a true map projection. It is constructed by calculating a three-dimensional frame and displaying the map objects on the surface of this frame.

Features

In the three-dimensional sense, globe is true in scale, equal-area, conformal, minimum error, and equidistant everywhere. When displayed, however, it looks like an Orthographic azimuthal projection, provided that the MATLAB axes `Projection` property is set to `'orthographic'`.

Parallels

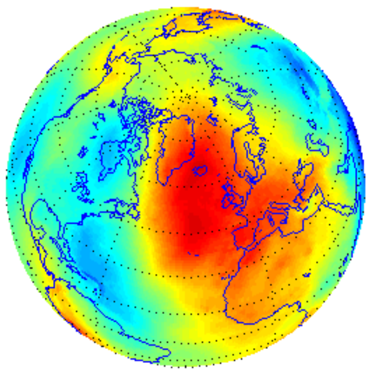
The globe requires no standard parallels.

Remarks

This is the only three-dimensional representation provided for display. Unless some other display purpose requires three dimensions, the Orthographic projection's display is equivalent.

Example

```
% Set up axes
axesm ('globe','Grid', 'on');
view(60,60)
axis off

% Display a surface
load geoid
meshm(geoid, geoidrefvec)

% Display coastline vectors
load coast
plotm(lat, long)
```



Classification Azimuthal

Syntax `gnomonic`

Graticule The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing increases rapidly away from this pole. The Equator and the opposite hemisphere cannot be shown

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features This is a perspective projection from the center of the globe on a plane tangent at the center point, which is a pole in the common polar aspect, but can be any point. Less than one hemisphere can be shown with this projection, regardless of its center point. The significant property of this projection is that all great circles are straight lines. This is useful in navigation, as a great circle is the shortest path between two points on the globe. Only the center point enjoys true scale and zero distortion. This projection is neither conformal nor equal-area.

Parallels There are no standard parallels for azimuthal projections.

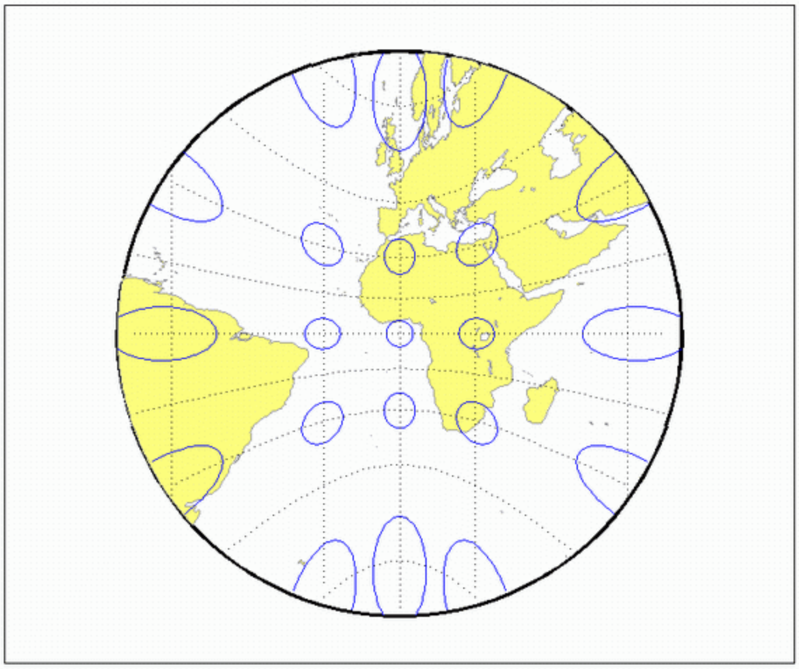
Remarks This projection may have been first developed by Thales around 580 B.C. Its name is derived from the gnomon, the face of a sundial, since the meridians radiate like hour markings. This projection is also known as a Gnomic or Central projection.

Limitations This projection is available only on the sphere. Data greater than 65° distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('gnomic', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification Pseudocylindrical

Syntax `goode`

Graticule Central Meridian: Straight line 0.44 as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves between the 40º44'11.8" parallels and elliptical arcs elsewhere, all concave toward the central meridian. The result is a slight, visible bend in the meridians at 40º44'11.8" N and S.

Parallels: Straight parallel lines, perpendicular to the central meridian. Equally spaced between the 40º44'11.8" parallels, with gradually decreasing spacing outside these parallels.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features This is an equal-area projection. Scale is true along all parallels and the central meridian between 40º44'11.8" N and S, and is constant along any parallel and between any pair of parallels equidistant from the Equator for all latitudes. Its distortion is identical to that of the Sinusoidal projection between 40º44'11.8" N and S, and to that of the Mollweide projection elsewhere. This projection is not conformal or equidistant.

Parallels This projection has one standard parallel, which is by definition fixed at 0º.

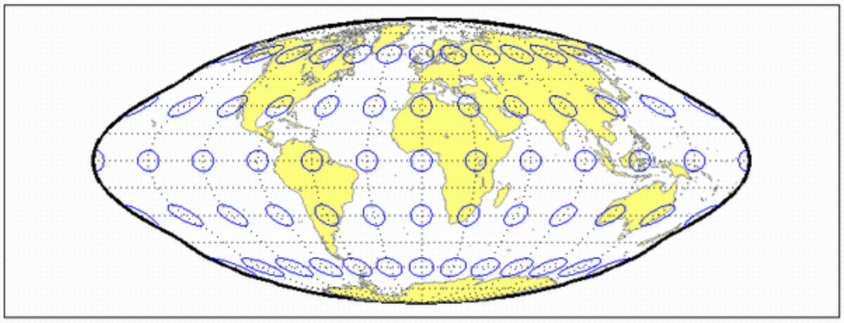
Remarks This projection was developed by J. Paul Goode in 1916. It is sometimes called simply the Homolosine projection, and it is usually used in an interrupted form. It is a merging of the Sinusoidal and Mollweide projections.

Limitations This projection is available in an uninterrupted form only.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('goode', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Hammer Projection

Classification

Modified Azimuthal

Syntax

`hammer`

Graticule

Meridians: Central meridian is a straight line half the length of the Equator. Other meridians are complex curves, equally spaced along the Equator, and concave toward the central meridian.

Parallels: Equator is straight. Other parallels are complex curves, equally spaced along the central meridian, and concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is equal-area. The only point free of distortion is the center point. Distortion of shape is moderate throughout. This projection has less angular distortion on the outer meridians near the poles than pseudoazimuthal projections

Parallels

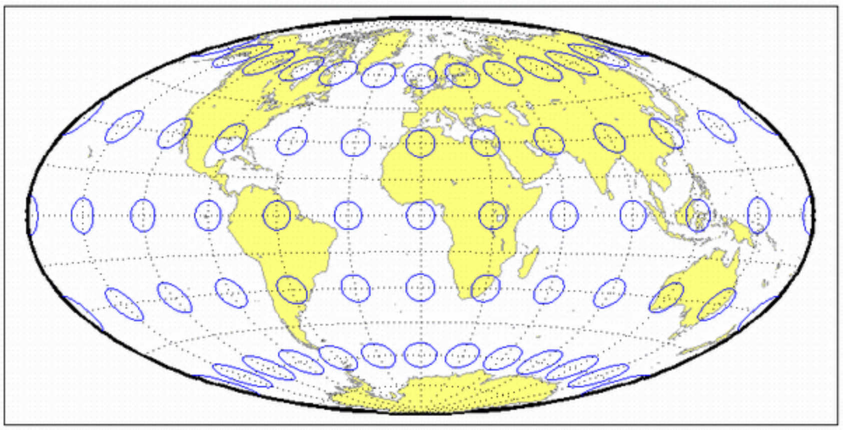
There is no standard parallel for this projection.

Remarks

This projection was presented by H. H. Ernst von Hammer in 1892. It is a modification of the Lambert Azimuthal Equal Area projection. Inspired by Aitoff projection, it is also known as the Hammer-Aitoff. It in turn inspired the Briesemeister, a modified oblique Hammer projection. John Bartholomew's Nordic projection is an oblique Hammer centered on 45 degrees north and the Greenwich meridian. The Hammer projection is used in whole-world maps and astronomical maps in galactic coordinates.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('hammer', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Hatano Asymmetrical Equal-Area Projection

Classification Pseudocylindrical

Syntax `hatano`

Graticule Central Meridian: Straight line 0.48 as long as the Equator.

Other Meridians: Equally spaced elliptical arcs concave toward the central meridian. The eccentricity of each ellipse changes at the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is not symmetrical about the Equator.

Poles: The North Pole is a line two-thirds the length of the Equator; the South Pole is a line three-fourths the length of the Equator.

Symmetry: About the central meridian but *not* the Equator.

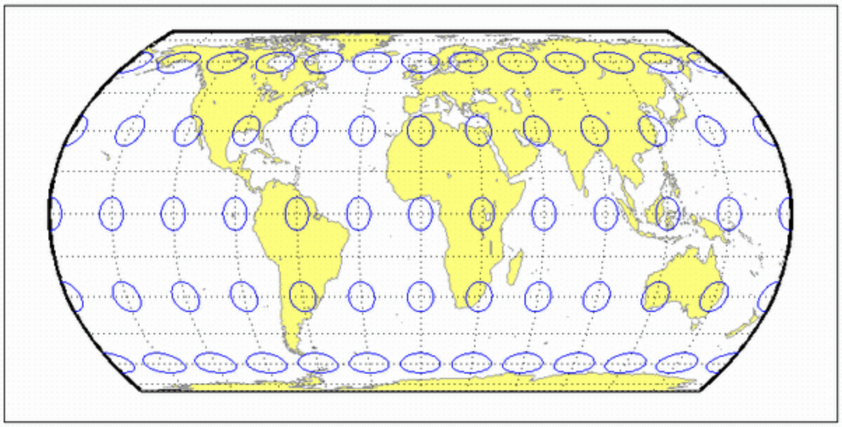
Features This is an equal-area projection. Scale is true along 40º42'N and 38º27'S, and is constant along any parallel but generally *not* between pairs of parallels equidistant from the Equator. It is free of distortion only along the central meridian at 40º42'N and 38º27'S. This projection is not conformal or equidistant.

Parallels Because of the asymmetrical nature of this projection, two standard parallels must be specified. The standard parallels are by definition fixed at 40º42'N and 38º27'S.

Remarks This projection was presented by Masataka Hatano in 1972.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('hatano', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Kavraisky V Projection

Classification

Pseudocylindrical

Syntax

`kavrsky5`

Graticule

Meridians: Complex curves converging at the poles. A sine function is used for *y*, but the meridians are not sine curves.

Parallels: Unequally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and the central meridian.

Features

This is an equal-area projection. Scale is true along the fixed standard parallels at 35º, and 0.9 true along the Equator. This projection is neither conformal nor equidistant.

Parallels

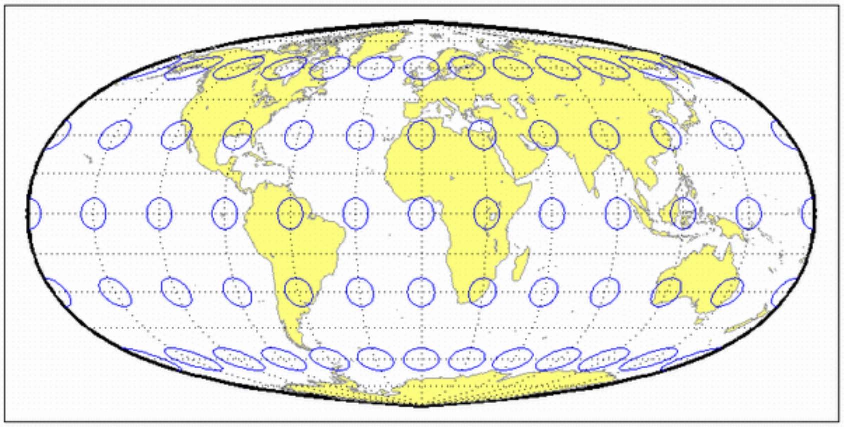
The fixed standard parallels are at 35º.

Remarks

This projection was described by V. V. Kavraisky in 1933.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('kavrsky5', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Kavraisky VI Projection

Classification

Pseudocylindrical

Syntax

`kavrsky6`

Graticule

Central Meridian: Straight line half the length of the Equator.

Meridians: Sine curves (60° segments).

Parallels: Unequally spaced straight lines.

Poles: Straight lines half the length of the Equator.

Symmetry: About the Equator and the central meridian.

Features

This is an equal-area projection. Scale is constant along any parallel or pair of equidistant parallels. This projection is neither conformal nor equidistant.

Parallels

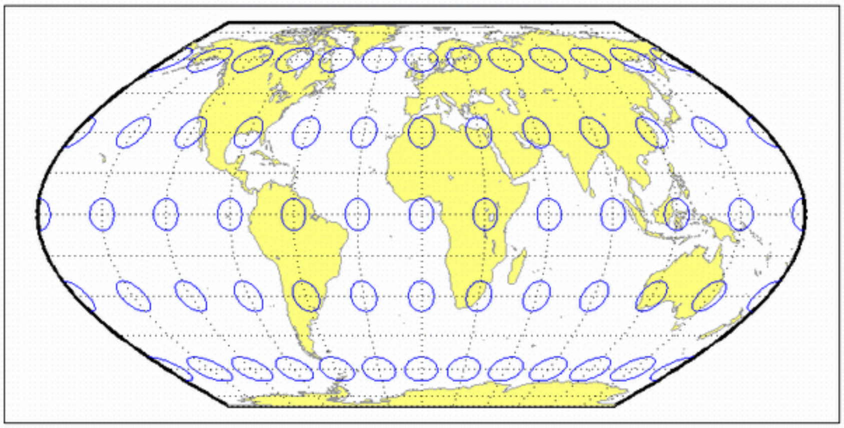
There are no standard parallels for this projection.

Remarks

This projection was described by V. V. Kavraisky in 1936. It is also called the Wagner I, for Karlheinz Wagner, who described it in 1932.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('kavrsky6', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Lambert Azimuthal Equal-Area Projection

Classification

Azimuthal

Syntax

`eqaazim`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. The entire Earth can be shown. Spacing decreases away from the central pole.

Pole: The central pole is a point; the other pole is a bounding circle with 1.41 the radius of the Equator.

Symmetry: About any meridian.

Features

This nonperspective projection is equal-area. Only the center point is free of distortion, but distortion is moderate within 90º of this point. Scale is true only at the center point, increasing tangentially and decreasing radially with distance from the center point. This projection is neither conformal nor equidistant.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

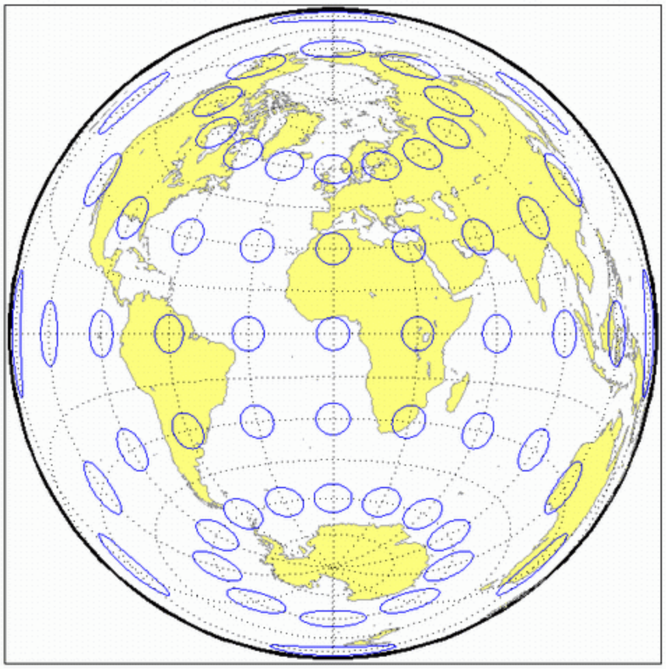
This projection was presented by Johann Heinrich Lambert in 1772. It is also known as the Zenithal Equal-Area and the Zenithal Equivalent projection, and the Lorgna projection in its polar aspect.

Limitations

Data greater than 160º distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqaazim', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Lambert Conformal Conic Projection

Classification

Conic

Syntax

`lambert`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.

Poles: The pole nearest a standard parallel is a point, the other cannot be shown.

Symmetry: About any meridian.

Features

Scale is true along the one or two selected standard parallels. Scale is constant along any parallel and is the same in every direction at any point. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is conformal everywhere but the poles; it is neither equal-area nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Stereographic Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Conformal Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator or two parallels equidistant from the Equator are chosen as the standard parallels, the cone becomes a cylinder, and a Mercator projection results. The default parallels are [15 75].

Remarks

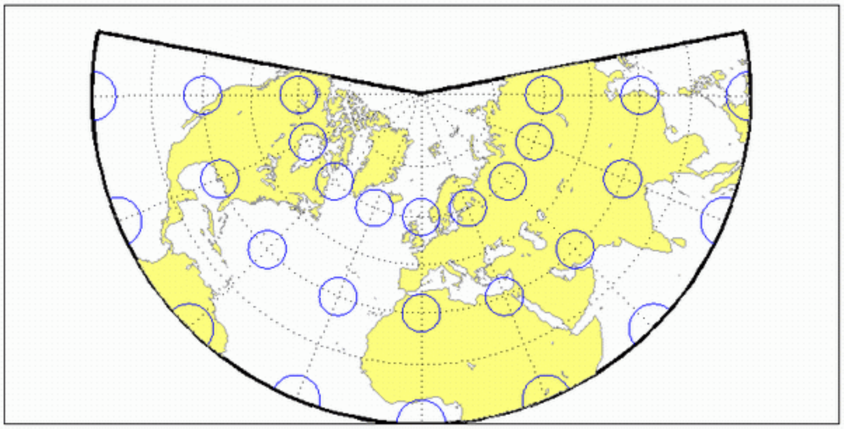
This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.

Limitations

Longitude data greater than 135° east or west of the central meridian is trimmed. The default map limits are [0 90] to avoid extreme area distortion.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('lambert', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`lambertstd`

Classification

Conic

Syntax

`lambertstd`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.

Poles: The pole nearest a standard parallel is a point, the other cannot be shown.

Symmetry: About any meridian.

Features

`lambertstd` implements the Lambert Conformal Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `lambert` for an alternative implementation based on rotating the authalic sphere.

Scale is true along the one or two selected standard parallels. Scale is constant along any parallel and is the same in every direction at any point. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is conformal everywhere but the poles; it is neither equal-area nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Stereographic Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Conformal Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator or two parallels equidistant from the Equator are chosen as the standard parallels, the cone becomes a cylinder, and a Mercator projection results. The default parallels are [15 75].

Lambert Conformal Conic Projection — Standard

Remarks

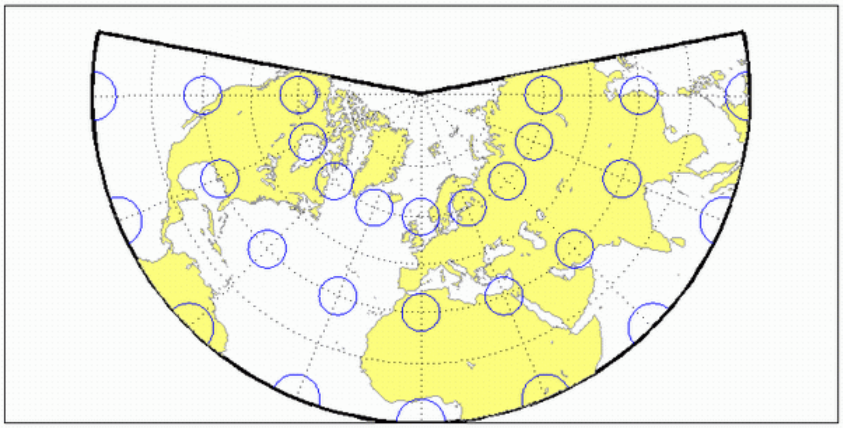
This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.

Limitations

Longitude data greater than 135° east or west of the central meridian is trimmed. The default map limits are [0 90] to avoid extreme area distortion.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('lambertstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`lambert`

Lambert Equal-Area Cylindrical Projection

Classification Cylindrical

Syntax `lambcyln`

Graticule Meridians: Equally spaced straight parallel lines 0.32 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

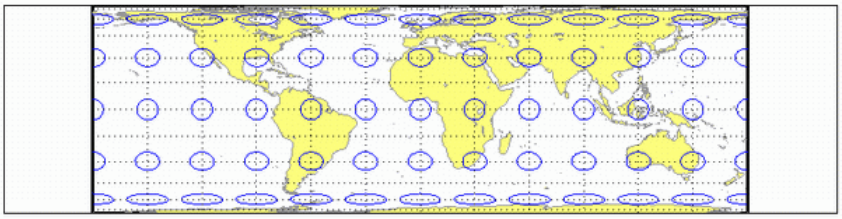
Features This is an orthographic projection onto a cylinder tangent at the Equator. It is equal-area, but distortion of shape increases with distance from the Equator. Scale is true along the Equator and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks This projection is named for Johann Heinrich Lambert and is a special form of the Equal-Area Cylindrical projection tangent at the Equator.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('lambcyn', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Loximuthal Projection

Classification

Pseudocylindrical

Syntax

`loximuth`

Graticule

Central Meridian: Straight line at least half as long as the Equator. Actual length depends on the choice of central latitude. Length is 0.5 when the central latitude is the Equator, for example, and 0.65 for central latitudes of 40º.

Other Meridians: Complex curves intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian. Symmetry about the Equator only when it is the central latitude.

Features

This projection has the special property that from the central point (the intersection of the central latitude with the central meridian), rhumb lines (loxodromes) are shown as straight, true to scale, and correct in azimuth from the center. This differs from the Mercator projection, in that rhumb lines are here shown in true scale and that unlike the Mercator, this projection does not maintain true azimuth for all points along the rhumb lines. Scale is true along the central meridian and is constant along any parallel, but not, generally, between parallels. It is free of distortion only at the central point and can be severely distorted in places. However, this projection is designed for its specific special property, in which distortion is not a concern.

Parallels

For this projection, only one standard parallel is specified: the central latitude described above. Specification of this central latitude defines the center of the Loximuthal projection. The default value is 0º.

Loximuthal Projection

Remarks

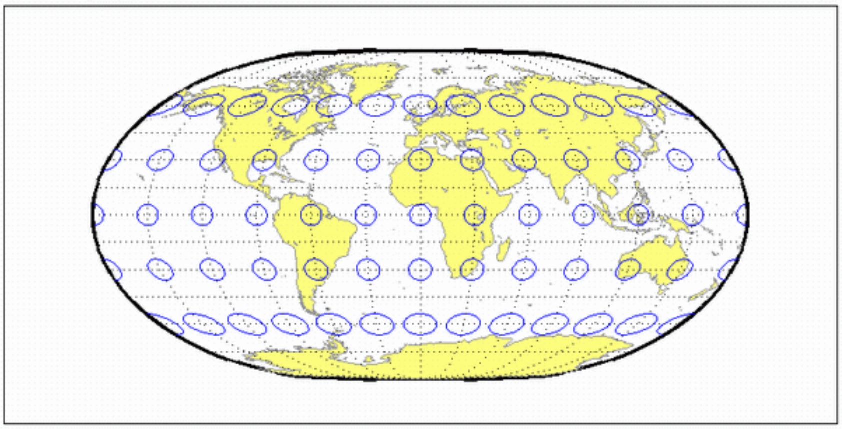
This projection was presented by Karl Siemon in 1935 and independently by Waldo R. Tobler in 1966. The Bordone Oval projection of 1520 was very similar to the Equator-centered Loximuthal.

Limitations

This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('loximuth', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



McBryde-Thomas Flat-Polar Parabolic Projection

Classification

Pseudocylindrical

Syntax

`flatplrp`

Graticule

Central Meridian: Straight line 0.48 as long as the Equator.

Other Meridians: Equally spaced parabolic curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 45º30' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the pointed-polar projections. It is free of distortion only at the two points where the central meridian intersects the 45º30' parallels. This projection is not conformal or equidistant.

Parallels

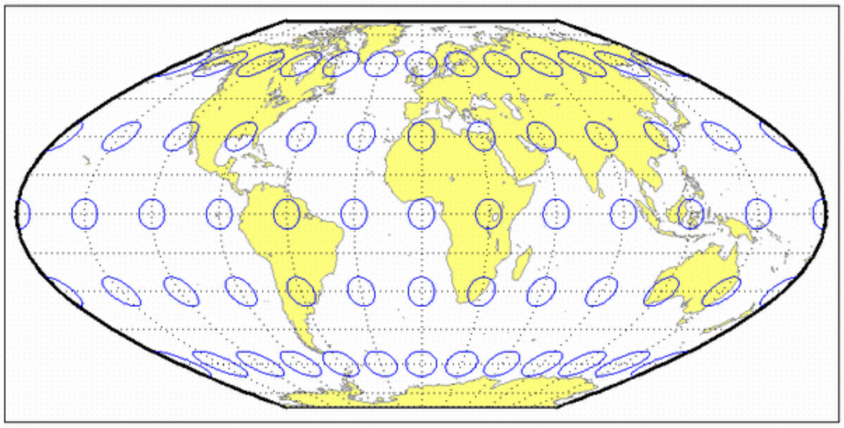
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 45º30'.

Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('flatplrp', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



McBryde-Thomas Flat-Polar Quartic Projection

Classification Pseudocylindrical

Syntax `flatplrq`

Graticule Central Meridian: Straight line 0.45 as long as the Equator.

Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features This is an equal-area projection. Scale is true along the 33º45’ parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the pointed-polar projections. It is free of distortion only at the two points where the central meridian intersects the 33º45’ parallels. This projection is not conformal or equidistant.

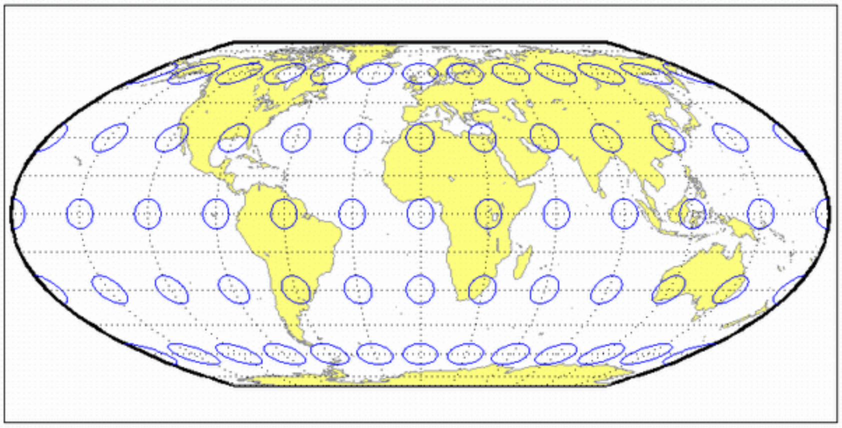
Parallels For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 33º45’.

Remarks This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949, and is also known simply as the Flat-Polar Quartic projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('flatplrq', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```

McBryde-Thomas Flat-Polar Quartic Projection



McBryde-Thomas Flat-Polar Sinusoidal Projection

Classification

Pseudocylindrical

Syntax

`flatplrs`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This projection is equal-area. Scale is true along the 55º51' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the central meridian intersects the 55º51' parallels. This projection is not conformal or equidistant.

Parallels

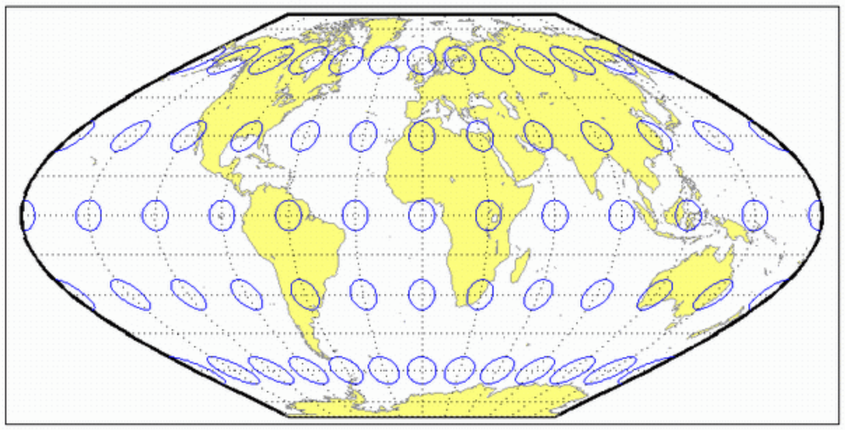
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 55º51'.

Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('flatplrs', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification

Cylindrical

Syntax

```
mercator
```

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Cannot be shown.

Symmetry: About any meridian or the Equator.

Features

This is a projection with parallel spacing calculated to maintain conformality. It is not equal-area, equidistant, or perspective. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. It is also constant in all directions near any given point. Scale becomes infinite at the poles. The appearance of the Mercator projection is unaffected by the selection of standard parallels; they serve only to define the latitude of true scale.

The Mercator, which may be the most famous of all projections, has the special feature that all rhumb lines, or loxodromes (lines that make equal angles with all meridians, i.e., lines of constant heading), are straight lines. This makes it an excellent projection for navigational purposes. However, the extreme area distortion makes it unsuitable for general maps of large areas.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude less than 86º may be chosen; the default is arbitrarily set to 0º.

Remarks

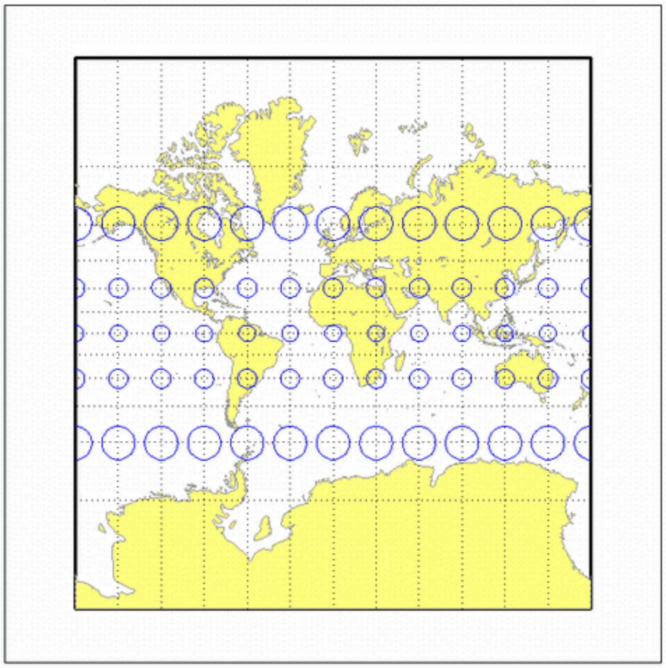
The Mercator projection is named for Gerardus Mercator, who presented it *for navigation* in 1569. It is now known to have been used for the Tunhuang star chart as early as 940 by Ch’ien Lo-Chih. It was first used in Europe by Erhard Etzlaub in 1511. It is also, but rarely, called the Wright projection, after Edward Wright, who developed the mathematics behind the projection in 1599.

Limitations

Data at latitudes greater than 86º is trimmed to prevent large *y*-values from dominating the display.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('mercator', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Miller Cylindrical Projection

Classification

Cylindrical

Syntax

`miller`

Graticule

Meridians: Equally spaced straight parallel lines 0.73 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles, less rapidly than that of the Mercator projection.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection with parallel spacing calculated to maintain a look similar to the Mercator projection while reducing the distortion near the poles and allowing the poles to be displayed. It is not equal-area, equidistant, conformal, or perspective. Scale is true along the Equator and constant between two parallels equidistant from the Equator. There is no distortion near the Equator, and it increases moderately away from the Equator, but it becomes severe at the poles.

The Miller Cylindrical projection is derived from the Mercator projection; parallels are spaced from the Equator by calculating the distance on the Mercator for a parallel at 80% of the true latitude and dividing the result by 0.8. The result is that the two projections are almost identical near the Equator.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks

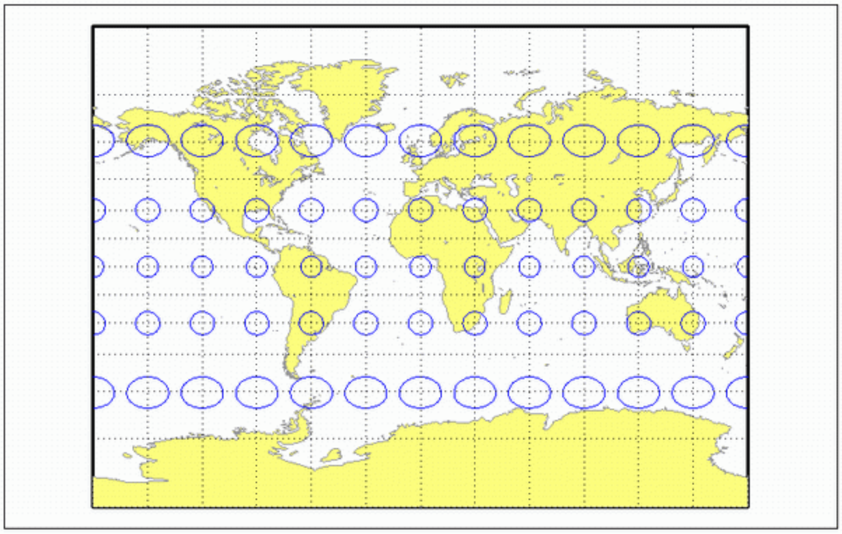
This projection was presented by Osborn Maitland Miller of the American Geographical Society in 1942. It is often used in place of the Mercator projection for atlas maps of the world, for which it is somewhat more appropriate.

Limitations

This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('miller', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification

Pseudocylindrical

Syntax

`mollweid`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Meridians 90º east and west of the central meridian form a circle. The others are equally spaced semiellipses intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator, but the spacing changes gradually.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 40º44' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 40º44' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

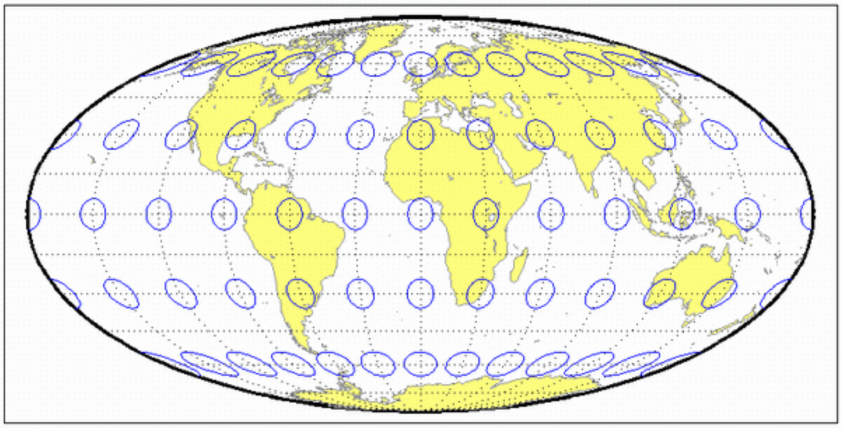
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 40º44'.

Remarks

This projection was presented by Carl B. Mollweide in 1805. Its other names include the Homolographic, the Homalographic, the Babinet, and the Elliptical projections. It is occasionally used for thematic world maps, and it is combined with the Sinusoidal to produce the Goode Homolosine projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('mollweid', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Murdoch I Conic Projection

Classification Conic

Syntax `murdoch1`

Graticule Meridians: Equally spaced straight lines converging at one of the poles.

Parallels: Equally spaced concentric circular arcs.

Poles: Arcs, one of which might become a point in the limit.

Symmetry: About any meridian.

Features This is an equidistant projection that is nearly minimum-error. Scale is true along any meridian and is constant along any parallel. Scale is also true along two standard parallels. These must be calculated, however (see remark on parallels below). The total area of the mapped area is correct, but it is not equal-area everywhere.

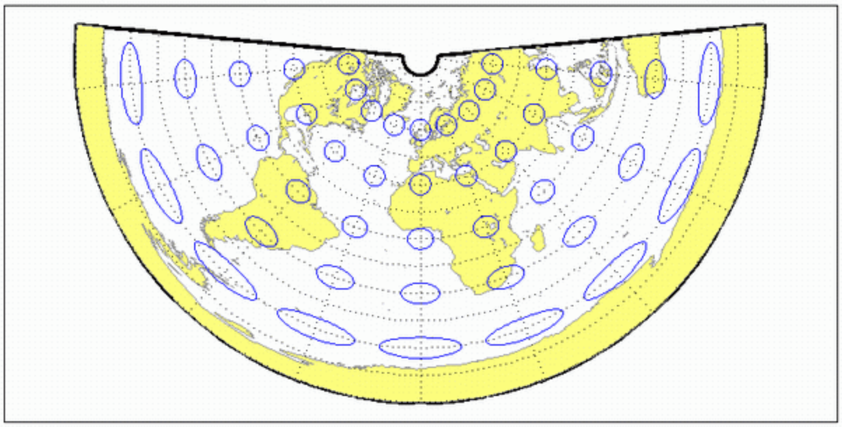
Parallels The parallels for this projection are not standard parallels, but rather limiting parallels. The special feature of this map, correct total area, holds between these parallels. The default parallels are [15 75].

Remarks Described by Patrick Murdoch in 1758.

Limitations This projection is available only for the sphere. Longitude data greater than 135° east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('murdoch1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Murdoch III Minimum Error Conic Projection

Classification

Conic

Syntax

`murdoch3`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles.

Parallels: Equally spaced concentric circular arcs.

Poles: Arcs, one of which might become a point in the limit.

Symmetry: About any meridian.

Features

This is an equidistant projection that is minimum-error. Scale is true along any meridian and is constant along any parallel. Scale is also true along two standard parallels. These must be calculated, however (see remark on parallels below). The total area of the mapped area is correct, but it is not equal-area everywhere.

Parallels

The parallels for this projection are not standard parallels, but rather limiting parallels. The special feature of this map, correct total area, holds between these parallels. The default parallels are [15 75].

Remarks

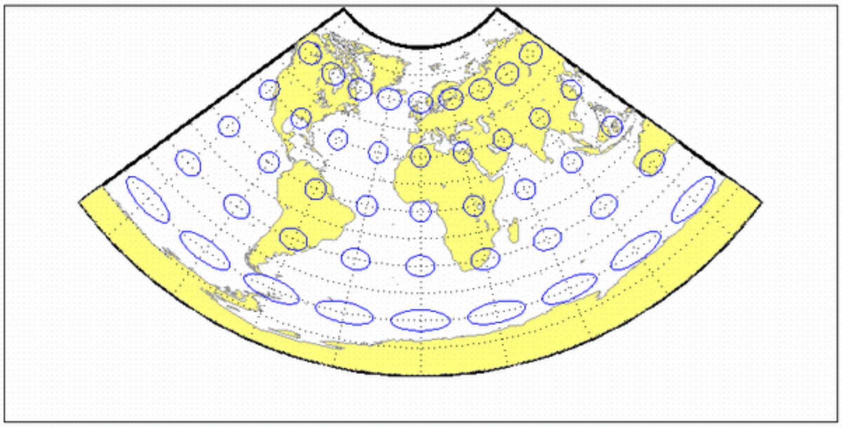
Described by Patrick Murdoch in 1758, with errors corrected by Everett in 1904.

Limitations

This projection is available only for the sphere. Longitude data greater than 135° east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('murdoch3', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Orthographic Projection

Classification

Azimuthal

Syntax

`ortho`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing decreases away from this pole. The opposite hemisphere cannot be shown.

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from an infinite distance (that is, orthogonally). The center point is a pole in the common polar aspect, but can be any point. This projection has two significant properties. It looks like a globe, providing views of the Earth resembling those seen from outer space. Additionally, all great and small circles are either straight lines or elliptical arcs on this projection. Scale is true only at the center point and is constant in the circumferential direction along any circle having the center point as its center. Distortion increases rapidly away from the center point, the only place that is distortion-free. This projection is neither conformal nor equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

This projection appears to have been developed by the Egyptians and Greeks by the second century B.C.

Limitations

This projection is available only for the sphere. Data greater than 89° distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('ortho', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```

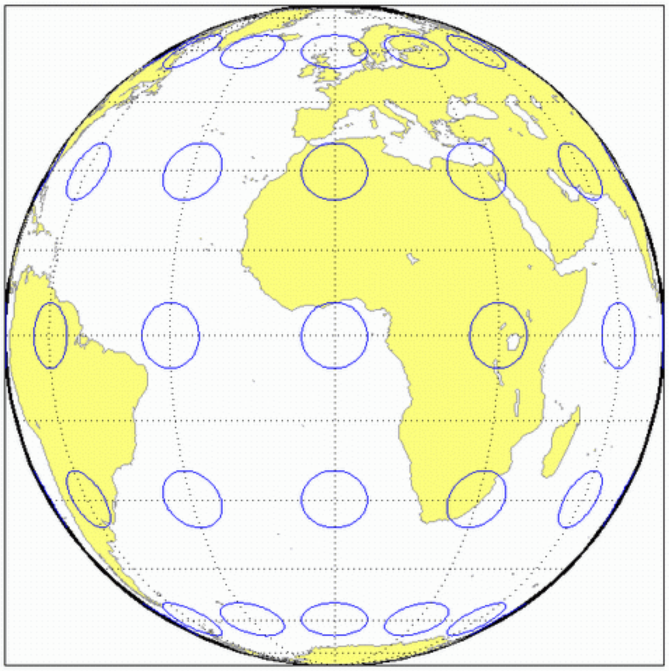


Plate Carrée Projection

Classification

Cylindrical

Syntax

`pcarree`

Graticule

Meridians: Equally spaced straight parallel lines half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having the same spacing as the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the Equator. Distortion of both shape and area increases with distance from the Equator. Scale is true along all meridians (i.e., it is equidistant) and the Equator and is constant along any parallel and along the parallel of opposite sign.

Parallels

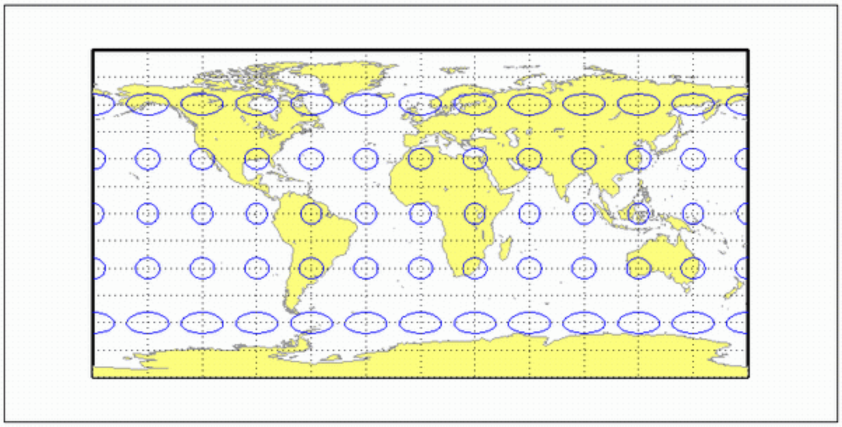
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks

This projection, like the more general Equidistant Cylindrical, is credited to Marinus of Tyre, thought to have invented it about A.D. 100. It may, in fact, have been originated by Erastosthenes, who lived approximately 275–195 B.C. The Plate Carrée has the most simply constructed graticule of any projection. It was used frequently in the 15th and 16th centuries and is quite common today in very simple computer mapping programs. It is the simplest and limiting form of the Equidistant Cylindrical projection. Another name for the Plate Carrée projection is the Simple Cylindrical. Its transverse aspect is the Cassini projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('pcarree', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification Polyconic

Syntax `polycon`

Graticule Central Meridian: A straight line.

Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About the Equator or the central meridian.

Features For this projection, each parallel has a curvature identical to its curvature on a cone tangent at that latitude. Since each parallel has its own cone, this is a “polyconic” projection. Scale is true along the central meridian and along each parallel. This projection is free of distortion only along the central meridian; distortion can be severe at extreme longitudes. This projection is neither conformal nor equal-area.

Parallels By definition, this projection has no standard parallels, since every parallel is a *standard parallel*.

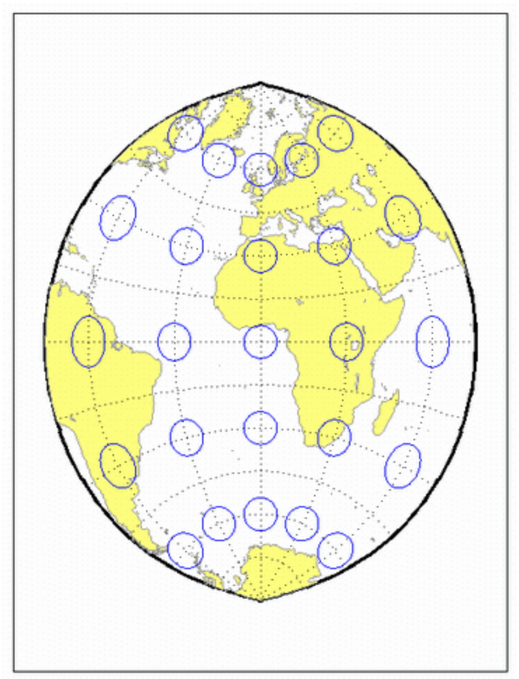
Remarks This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.

Limitations Longitude data greater than 75º east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('polycon', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
```

```
tissot;
```



See Also `polyconstd`

Classification

Polyconic

Syntax

`polyconstd`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About the Equator or the central meridian.

Features

`polyconstd` implements the Polyconic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `polycon` for an alternative implementation based on rotating the rectifying sphere.

For this projection, each parallel has a curvature identical to its curvature on a cone tangent at that latitude. Since each parallel has its own cone, this is a "polyconic" projection. Scale is true along the central meridian and along each parallel. This projection is free of distortion only along the central meridian; distortion can be severe at extreme longitudes. This projection is neither conformal nor equal-area.

Parallels

By definition, this projection has no standard parallels, since every parallel is a *standard parallel*.

Remarks

This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.

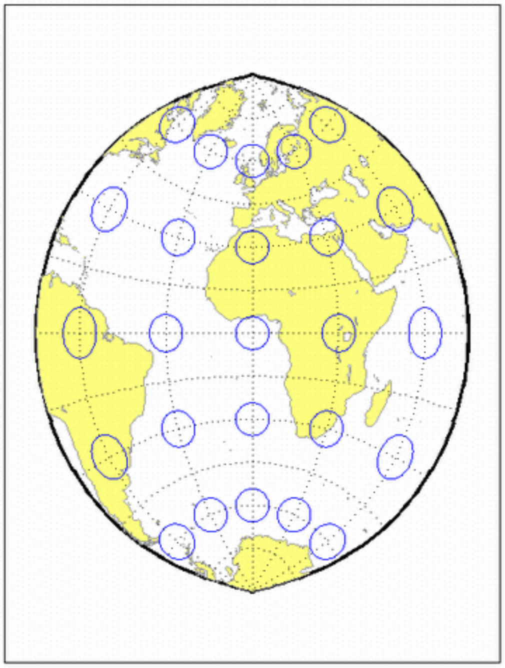
Limitations

Longitude data greater than 75° east or west of the central meridian is trimmed.

Polyconic Projection — Standard

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('polyconstd ', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`polycon`

Putnins P5 Projection

Classification

Pseudocylindrical

Syntax

`putnins5`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced portions of hyperbolas intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 21°14' parallels and is constant along any parallel, between any pair of parallels equidistant from the Equator, and along the central meridian. It is not free of distortion at any point. This projection is not equal-area, conformal, or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 21°14'.

Remarks

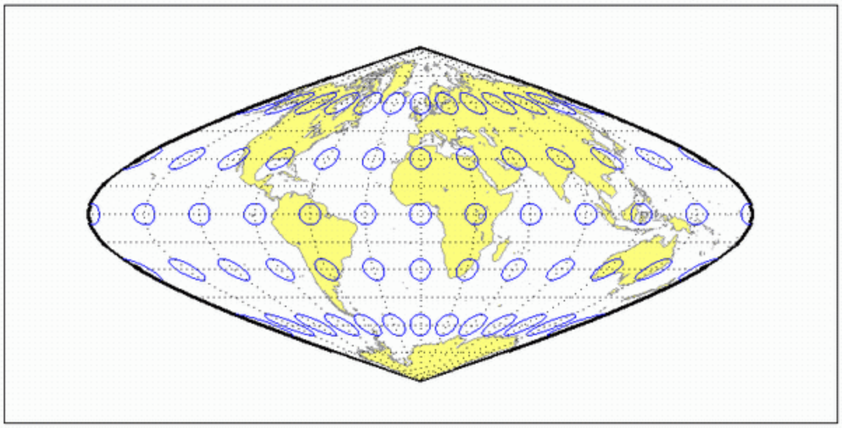
This projection was presented by Reinholds V. Putnins in 1934.

Limitations

This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('putnin5', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Quartic Authalic Projection

Classification

Pseudocylindrical

Syntax

```
quartic
```

Graticule

Central Meridian: Straight line 0.45 as long as the Equator.

Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes gradually and is greatest near the Equator.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the Equator and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the Sinusoidal projection. It is free of distortion along the Equator. This projection is not conformal or equidistant.

Parallels

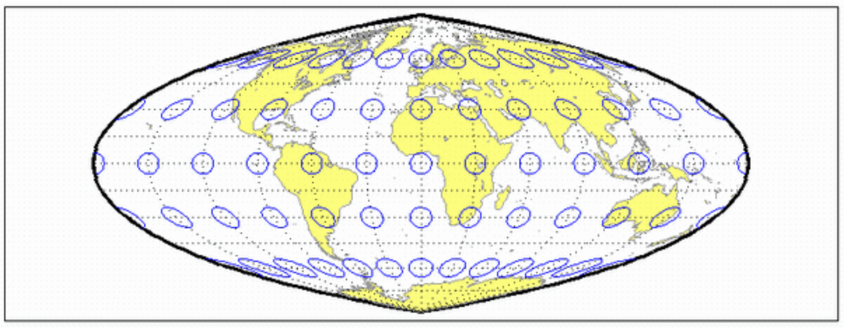
This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

This projection was presented by Karl Siemon in 1937 and independently by Oscar Sherman Adams in 1945.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('quartic', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Classification

Pseudocylindrical

Syntax

`robinson`

Graticule

Central Meridian: Straight line 0.51 as long as the Equator.

Other Meridians: Equally spaced, resemble elliptical arcs and are concave toward the central meridian.

Parallels: Straight parallel lines, perpendicular to the central meridian. Spacing is equal between the 38º parallels, decreasing outside these limits.

Poles: Lines 0.53 as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 38º parallels and is constant along any parallel or between any pair of parallels equidistant from the Equator. It is not free of distortion at any point, but distortion is very low within about 45º of the center and along the Equator. This projection is not equal-area, conformal, or equidistant; however, it is considered to *look right* for world maps, and hence is widely used by Rand McNally, the National Geographic Society, and others. This feature is achieved through the use of tabular coordinates rather than mathematical formulae for the graticules.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 38º.

Remarks

This projection was presented by Arthur H. Robinson in 1963, and is also called the Orthophanic projection, which means *right appearing*.

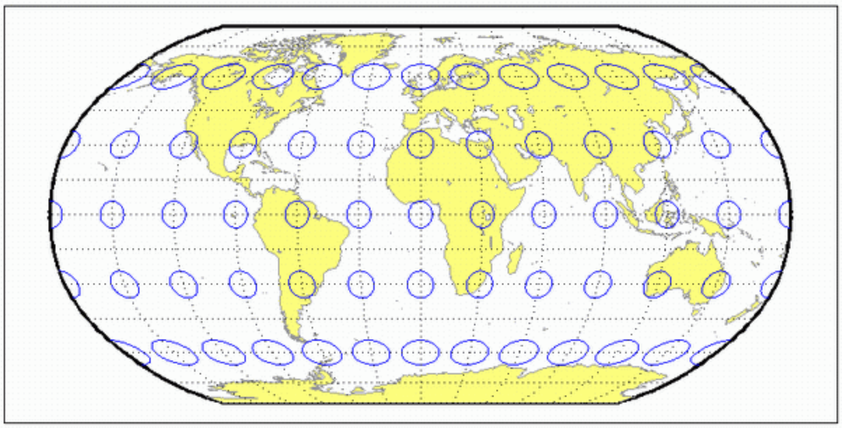
Limitations

This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('robinson', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Sinusoidal Projection

Classification

Pseudocylindrical

Syntax

```
sinusoid
```

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This projection is equal-area. Scale is true along every parallel and along the central meridian. There is no distortion along the Equator or along the central meridian, but it becomes severe near the outer meridians at high latitudes.

Parallels

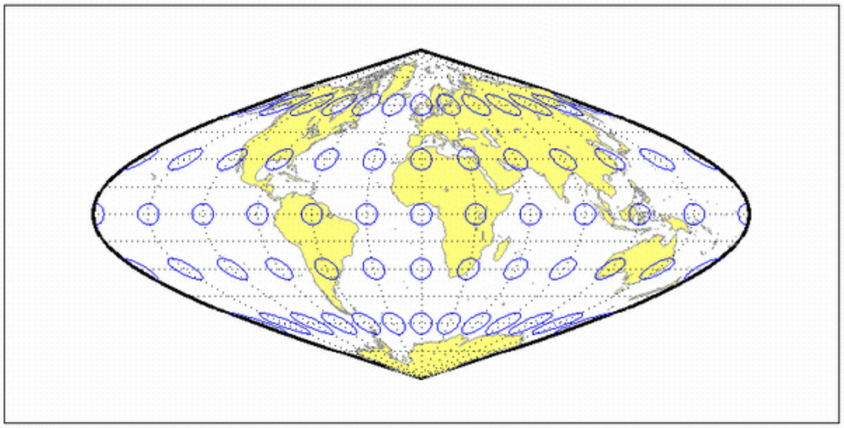
This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

This projection was developed in the 16th century. It was used by Jean Cossin in 1570 and by Jodocus Hondius in Mercator atlases of the early 17th century. It is the oldest pseudocylindrical projection currently in use, and is sometimes called the Sanson-Flamsteed or the Mercator Equal-Area projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('sinusoid', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Stereographic Projection

Classification

Azimuthal

Syntax

```
stereo
```

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing increases gradually away from this pole. The opposite hemisphere cannot be shown

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from the point antipodal to the center point. The center point is a pole in the common polar aspect, but can be any point. This projection has two significant properties. It is conformal, being free from angular distortion. Additionally, all great and small circles are either straight lines or circular arcs on this projection. Scale is true only at the center point and is constant along any circle having the center point as its center. This projection is not equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

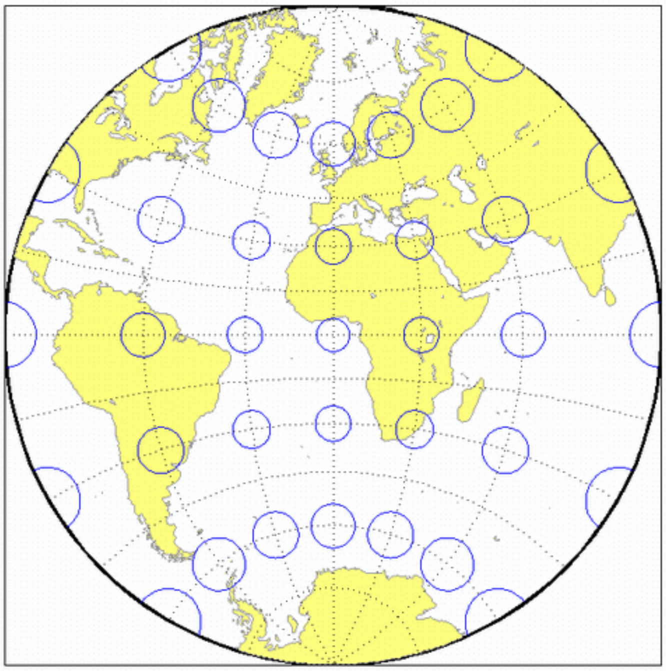
The polar aspect of this projection appears to have been developed by the Egyptians and Greeks by the second century B.C.

Limitations

Data greater than 90º distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('stereo', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Tissot Modified Sinusoidal Projection

Classification Pseudocylindrical

Syntax `modsine`

Graticule Meridians: Sine curves converging at the Poles.

Parallels: Equally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and the central meridian

Features This is an equal-area projection. Scale is constant along any parallel or any pair of equidistant parallels, and along the central meridian. It is not equidistant or conformal.

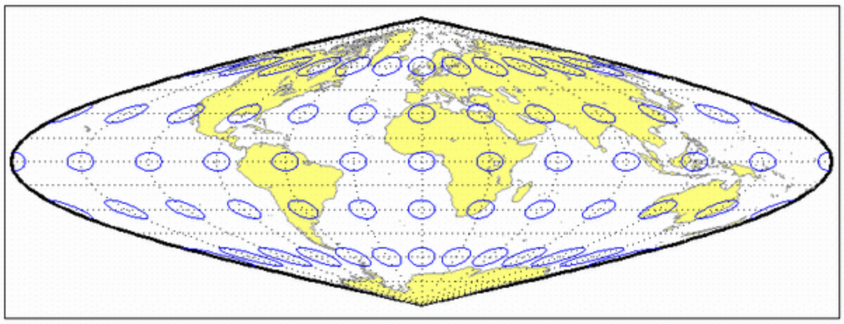
Parallels There are no standard parallels for this projection.

Remarks This projection was first described by N. A. Tissot in 1881

Limitations This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('modsine', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Transverse Mercator Projection

Classification Cylindrical

Syntax `tranmerc`

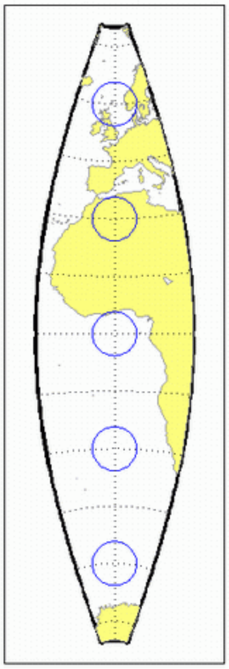
Features This conformal projection is the transverse form of the Mercator projection and is also known as the Gauss-Krueger pojection. It is not equal area, equidistant, or perspective.

The scale is constant along the central meridian, and increases to the east and west. The scale at the entral meridian can be set true to scale, or reduced slightly to render the mean scale of the overall map more nearly correct.

Remarks The uniformity of scale along its centeral meridian makes Transverse Mercator an excellent choice for mapping areas that are elongated north-to-south. Its best known application is the definition of Universal Transverse Mercator (UTM) coordinates. Each UTM zone spans only 6 degrees of longitude, but the northern half extends from the equator all the way to 84 degrees north and the southern half extends from 80 degrees south to the equator. Other map grids based on Transverse Mercator include many of the state plane zones in the U.S., and the U.K. National Grid.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('tranmerc', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Trystan Edwards Cylindrical Projection

Classification Cylindrical

Syntax `trystan`

Graticule Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

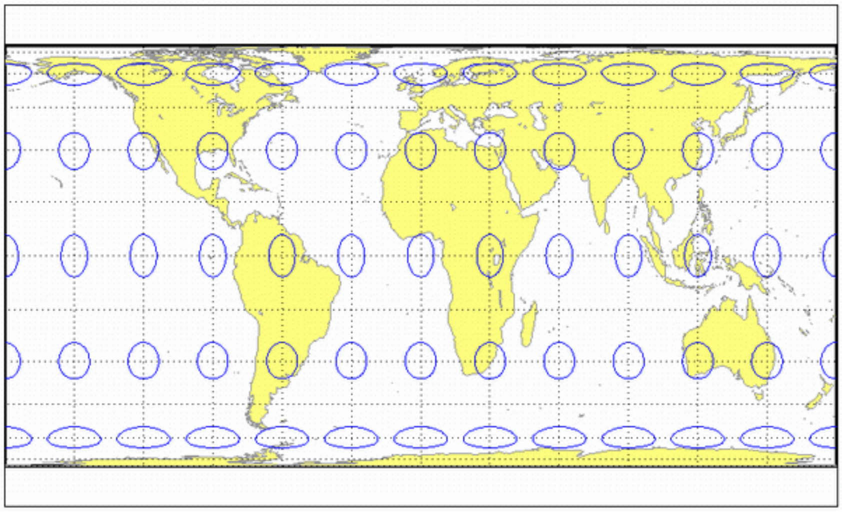
Features This is an orthographic projection onto a cylinder secant at the 37º24' parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 37º24'.

Remarks This projection is named for Trystan Edwards, who presented it in 1953. It is a special form of the Equal-Area Cylindrical projection secant at 37º24'N and S.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('trystan', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Universal Polar Stereographic System

Classification

Azimuthal

Syntax

`ups`

Graticule

The graticule described is for the southern zone.

Meridians: Equally spaced straight lines centered on the South Pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the South Pole. Spacing increases gradually away from the circle of true scale along latitude 87 degrees, 7 minutes N. The opposite pole cannot be shown.

Poles: The South Pole is a point. The North Pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent to either the North or South Pole. It is conformal, being free from angular distortion. Additionally, all great and small circles are either straight lines or circular arcs on this projection. Scale is true along latitudes 87 degrees, 7 minutes N or S, and is constant along any other parallel. This projection is not equal area.

Parallels

The parallels 87 degrees, 7 minutes N and S are lines of true scale by virtue of the scale factor. There are no standard parallels for azimuthal projections.

Remarks

This projection is a special case of the stereographic projection in the polar aspect. It is used as part of the Universal Transverse Mercator (UTM) system to extend coverage to the poles. This projection has two zones: "North" for latitudes 84º N to 90º N, and "South" for latitudes 80º S to 90º S. The defaults for this projection are: scale factor is 0.994, false easting and northing are 2,000,000 meters. The international ellipsoid in units of meters is used as the geoid model.

Universal Transverse Mercator System

Classification

Cylindrical

Syntax

```
utm
```

Graticule

Meridians: Complex curves concave toward the central meridian.

Parallels: Complex curves concave toward the nearest pole.

Poles: Not shown.

Symmetry: About the central meridian or the Equator.

Features

This is a conformal projection with parameters chosen to minimize distortion over a defined set of small areas. It is not equal area, equidistant, or perspective. Scale is true along two straight lines on the map approximately 180 kilometers east and west of the central meridian, and is constant along other straight lines equidistant from the central meridian. Scale is less than true between, and greater than true outside the lines of true scale.

Parallels

There are no standard parallels for this projection. There are two lines of zero distortion by virtue of the scale factor.

Remarks

The UTM system divides the world between 80º S and 84º degrees N into a set of quadrangles called zones. Zones generally cover 6 degrees of longitude and 8 degrees of latitude. Each zone has a set of defined projection parameters, including central meridian, false eastings and northings and the reference ellipsoid. The projection equations are the Gauss-Krüger versions of the Transverse Mercator. The projected coordinates form a grid system, in which a location is specified by the zone, easting and northing.

The UTM system was introduced in the 1940s by the U.S. Army. It is widely used in topographic and military mapping.

Van der Grinten I Projection

Classification Polyconic

Syntax `vgrint1`

Graticule Central Meridian: A straight line.

Meridians: Circular curves spaced equally along the equator and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are circular arcs concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator or the central meridian.

Features In this projection, the world is enclosed in a circle. Scale is true along the Equator and increases rapidly away from the Equator. Area distortion is extreme near the poles. This projection is neither conformal nor equal-area.

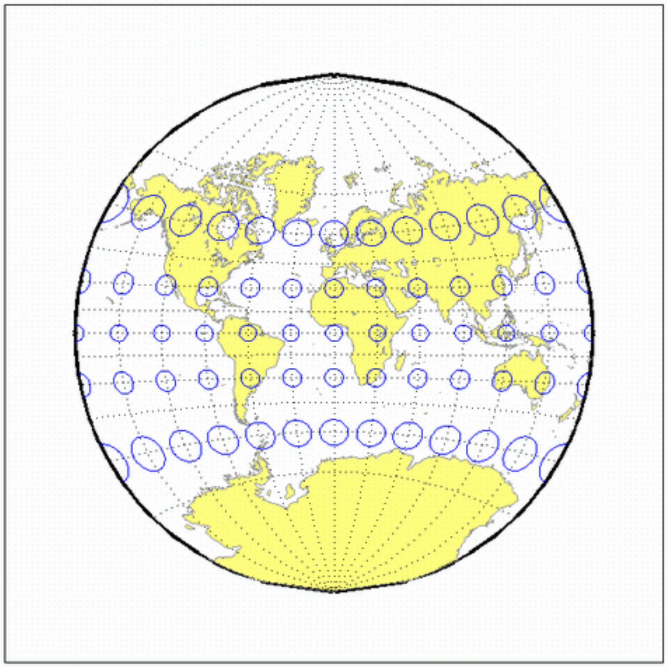
Parallels There are no standard parallels for this projection.

Remarks This projection was presented by Alphons J. Van der Grinten in 1898. He obtained a U.S. patent for it in 1904. It is also known simply as the Van der Grinten projection (without the "I").

Limitations This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('vgrint1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Vertical Perspective Azimuthal Projection

Classification

Azimuthal

Syntax

`vperspec`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing decreases away from this pole. The opposite hemisphere cannot be shown, nor can distant parts of the closer hemisphere. The limit of visibility depends on the observation altitude.

Poles: The central pole is a point. The other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from a finite distance. Scale is true only at the center point, and is constant in the circumferential direction along any circle having the center point as its center. Distortion increases rapidly away from the center point, the only point which is distortion free. This projection is neither conformal nor equal area.

Remarks

This projection provides views of the globe resembling those seen from a spacecraft in orbit. The Orthographic projection is a limiting form with the observer at an infinite distance.

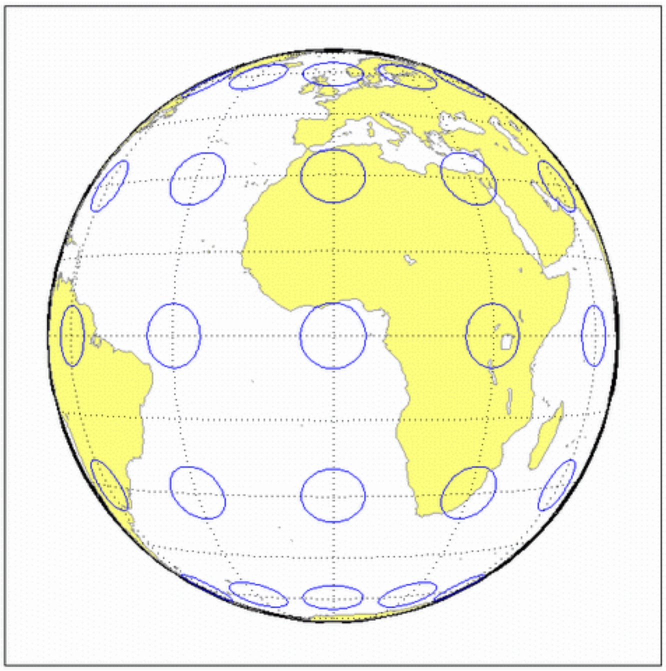
This projection requires a view altitude parameter, which specifies the observer’s altitude above the origin point. Because this parameter is unique to this projection and because the projection does not need any standard parallels, a special workaround is used. Rather than add an extra map axes property just for `vperspec`, the `MapParallels` property is repurposed instead. You should assign the desired view altitude value to the `MapParallels` property. Provide a scalar value for length in the same units as the earth radius or semi-major axis length used in the map axes reference ellipsoid (`'Geoid'`) property.

Limitations

This projection is available only for the sphere. Data more distant than the limit of visibility is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('vperspec', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Wagner IV Projection

Classification

Pseudocylindrical

Syntax

```
wagner4
```

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced portions of ellipses concave toward the central meridian. The meridians 103º55' east and west of the central meridian are circular arcs.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 42º59' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is not as extreme near the outer meridians at high latitudes as for pointed-polar pseudocylindrical projections, but there is considerable distortion throughout the polar regions. It is free of distortion only at the two points where the 42º59' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

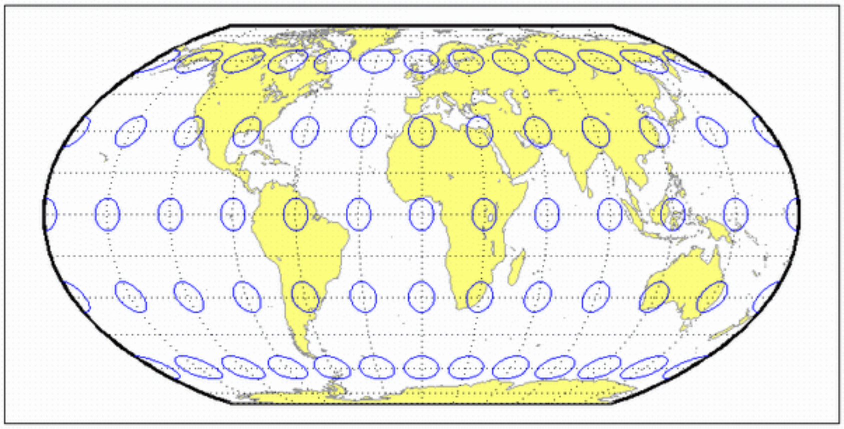
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 42º59'.

Remarks

This projection was presented by Karlheinz Wagner in 1932.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('wagner4', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Werner Projection

Classification Pseudoconic

Syntax `werner`

Graticule Central Meridian: A straight line.

Meridians: Complex curves connecting points equally spaced along each parallel and concave toward the central meridian.

Parallels: Concentric circular arcs spaced at true distances along the central meridian, centered on one of the poles.

Poles: Points.

Symmetry: About the central meridian.

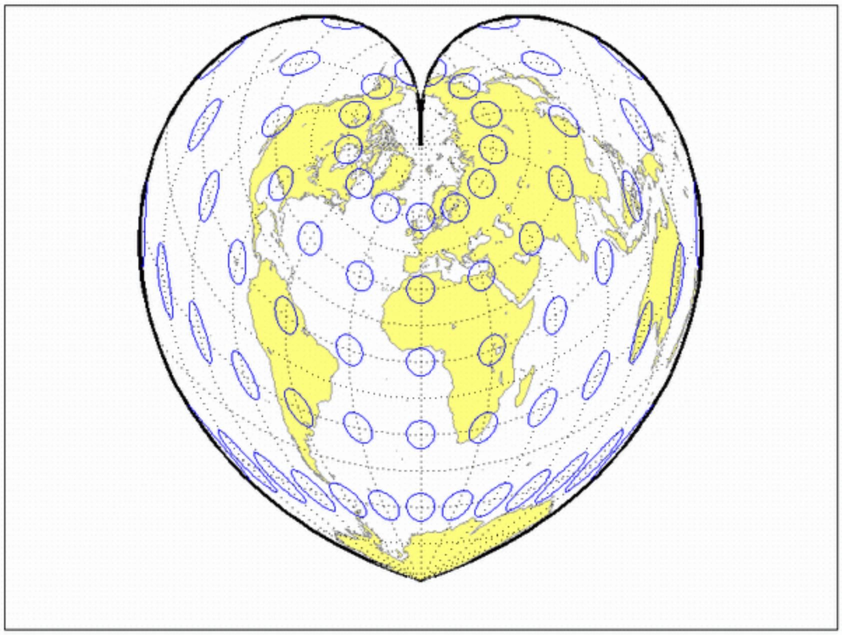
Features This is an equal-area projection. It is a Bonne projection with one of the poles as its standard parallel. The central meridian is free of distortion. This projection is not conformal. Its heart shape gives it the additional descriptor *cordiform*.

Parallels The standard parallel for this projection is set to 90º N.

Remarks This projection was developed by Johannes Stabius (Stab) about 1500 and was promoted by Johannes Werner in 1514. It is also called the Stab-Werner projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('werner', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Wetch Cylindrical Projection

Classification

Cylindrical

Syntax

`wetch`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90° from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a perspective projection from the center of the Earth onto a cylinder tangent to the central meridian. It is not equal-area, equidistant, or conformal. Scale is true along the central meridian and constant between two points equidistant in x and y from the central meridian. There is no distortion along the central meridian, but it increases rapidly away from the central meridian in the y-direction.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, which is the transverse aspect of the Central Cylindrical, the standard parallel *of the base projection* is by definition fixed at 0°.

Remarks

This is the transverse aspect of the Central Cylindrical projection discussed by J. Wetch in the early 19th century.

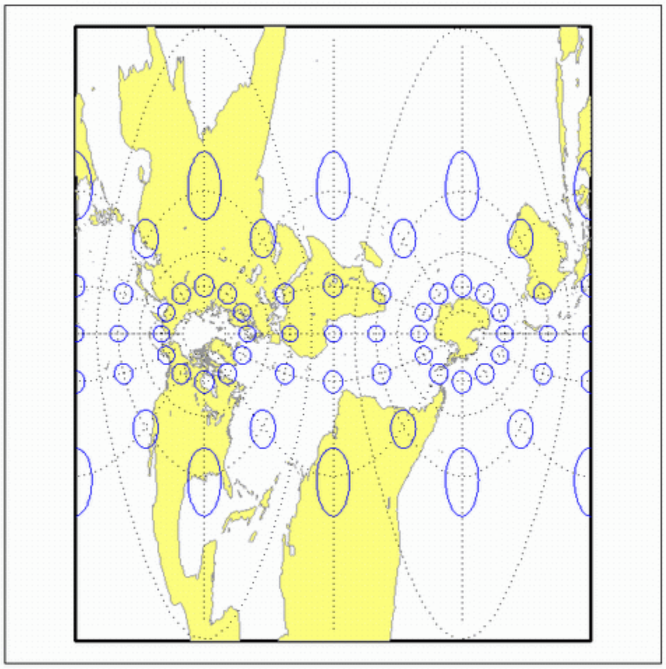
Limitations

This projection is available only for the sphere. To prevent large y-values from dominating the display, data at y-values that would correspond to latitudes of greater than 75° in the normal aspect of the Central Cylindrical projection is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('wetch', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Wiechel Projection

Classification Pseudoazimuthal

Syntax `wiechel`

Graticule The graticule described is for a polar aspect.

Meridians: Equally spaced semicircles from pole to pole, concave toward the west.

Parallels: Concentric circles.

Pole: The central pole is a point; the other pole is a bounding circle.

Symmetry: Radially about the center point.

Features This equal-area projection is a novelty map, usually centered at a pole, showing semicircular meridians in a pinwheel arrangement. Scale is correct along the meridians. This projection is not conformal.

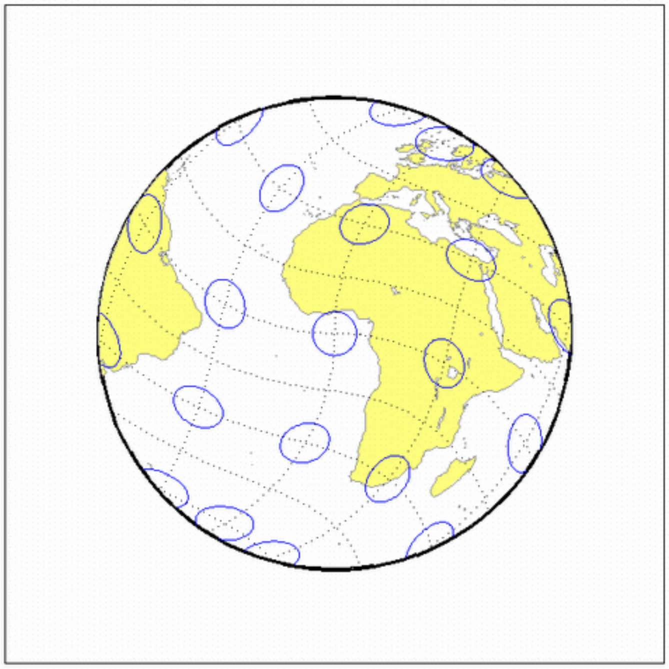
Parallels There are no standard parallels for azimuthal projections.

Remarks This projection was presented by H. Wiechel in 1879.

Limitations Data greater than 65º distant from the center point is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('wiechel', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Winkel I Projection

Classification Pseudocylindrical

Syntax `winkel`

Graticule Central Meridian: Straight line at least half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines at least half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features This projection is an arimethical average of the *x* and *y* coordinates of the Sinusoidal and Equidistant Cylindrical projections. Scale is true along the standard parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. There is no point free of distortion. This projection is not equal-area, conformal, or equidistant.

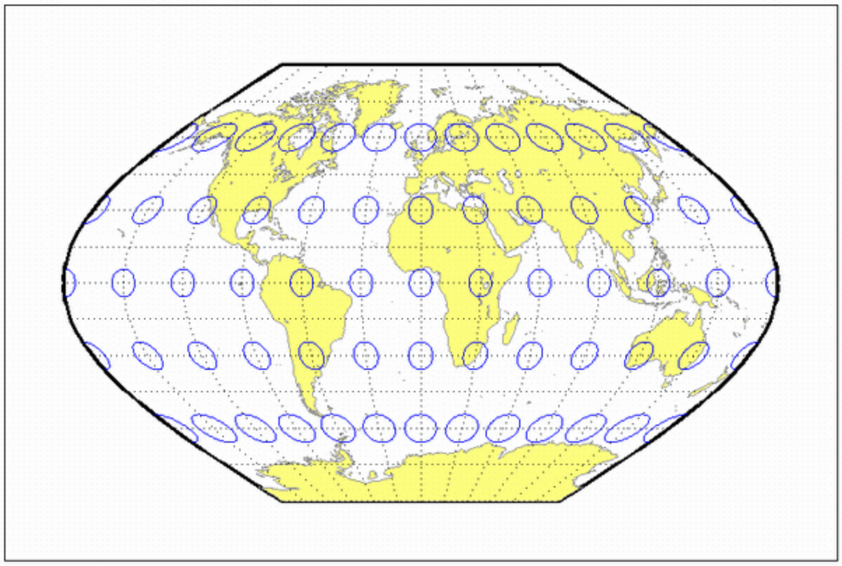
Parallels For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. Any latitude may be chosen; the default is set to 50º28'.

Remarks This projection was developed by Oswald Winkel in 1914. Its limiting form is the Eckert V when a standard parallel of 0º is chosen.

Limitations This projection is available only for the sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('winkel', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Glossary

This glossary of geographical terms is drawn extensively from "An Album of Map Projections", U.S. Geological Survey Professional Paper 1453, by John P. Snyder and Philip M. Voxland.

Because the purpose of this glossary is to assist in understanding and using Mapping Toolbox features, it includes some terms specific to the toolbox, and gives some other terms shades of meaning beyond their general definitions.

Antipodes
Two points on opposite sides of a planet.

Arc-second
1/3600th of a degree (1 second) of latitude or longitude.

Aspect
The conceptual placement of a projection system in relation to the Earth's axis (direct, normal, polar, equatorial, oblique, and so on).

Attribute
In vector geodata, a quantitative or qualitative descriptor of a spatial entity. An attribute can describe a real-world quality (such as population or land area), or a graphic quality (such as patch color or line weight). Attributes are frequently coded as numbers or strings in character-coded or binary tabular data files, with one or more attribute per map feature.

Attribute spec
(Attribute specification) A cell array structure that specifies attributes of geodata to be included in a KML file and defines label strings and format strings for each attribute. Used with `kmlwrite`.

Authalic projection
See Equal-area projection.

Axes
See Map axes.

Azimuth

The angle a line makes with a meridian, taken clockwise from north.

Azimuthal projection

A projection on which the azimuth or direction from a given central point to any other point is shown correctly. When a pole is the central point, all meridians are spaced at their true angles and are straight radii of concentric circles that represent the parallels. Also called a zenithal projection.

Bathymetry

The measurement of water depths of oceans, seas, lakes, and other bodies of water.

Bowditch, Nathaniel

A late 18th/early 19th century mathematician, astronomer, and sailor who “wrote the book” on navigation. John Hamilton Moore’s *The Practical Navigator* was the leading navigational text when Bowditch first went out to sea, and had been for many years. Early in his first voyage, however, Bowditch began noticing errors in Moore’s book, which he recorded and later used in preparing an American edition of Moore’s work. The revisions were to such an extent that Bowditch was named the principal author, and the title was changed to *The New American Practical Navigator*, published in 1802. In 1868, the U.S. Navy bought the copyright to the book, which is still commonly referred to as “Bowditch” and considered the “bible” of navigation.

Buffer zone

The locus of points that lie within a specified distance from a map feature.

Cartography

The art or practice of making charts or maps. *See* Map.

Categorical geodata

Geospatial data in which raster pixel values (or vector data attributes) are categorical indices, usually coded as integers. The meanings of the categories are usually stored in a separate table. Examples are geocodes, land use categories, and indexed color images. *See* Numerical geodata.

Central meridian

The meridian passing through the center of a projection, often a straight line about which the projection is symmetrical.

Central projection

A projection in which the Earth is projected geometrically from the center of the Earth onto a plane or other surface. The Gnomonic and Central Cylindrical projections are examples.

Choropleth

A map portraying regions of homogeneous classified attribute values, changing abruptly at region boundaries, and colored or shaded according to their attribute values. Thematic political maps are usually choropleth maps.

Complex curves

Curves that are not elementary forms such as circles, ellipses, hyperbolas, parabolas, and sine curves, such as rivers, coastlines, and administrative boundaries.

Composite projection

A projection formed by connecting two or more projections along common lines such as parallels of latitude, necessary adjustments being made to achieve fit. The Goode Homolosine projection is an example.

Conformal projection

A projection on which all angles at each point are preserved, except at a finite number of singular points (e.g., the poles in a Mercator projection). Also called an orthomorphic projection.

Conic projection

A projection resulting from the conceptual projection of the Earth onto a tangent or secant cone, which is then cut lengthwise and laid flat. When the axis of the cone coincides with the polar axis of the Earth, all meridians are straight equidistant radii of concentric circular arcs representing the parallels, but the meridians are spaced at less than their true angles. Mathematically, the projection is often only partially geometric.

Constant scale

A linear scale that remains the same along a particular line on a map, although that scale may not be the same as the stated or nominal scale of the map.

Contour

All points that are at the same height above or below a reference datum; generally applied to continuous, single-valued surfaces only, such as elevation, temperature, or magnetic field strength.

Conventional aspect

See Normal aspect.

Correct scale

A linear scale having exactly the same value as the stated or nominal scale of the map, or a scale factor of 1.0. Also called true scale.

Cylindrical projection

A projection resulting from the conceptual projection of the Earth onto a tangent or secant cylinder, which is then cut lengthwise and laid flat. When the axis of the cylinder coincides with the axis of the Earth, the meridians are straight, parallel, and equidistant, while the parallels of latitude are straight, parallel, and perpendicular to the meridians. Mathematically, the projection is often only partially geometric.

Data grid

A raster data set consisting of an array of values posted or sampled at specific geographic points. Mapping Toolbox data grids can be implicit (regular) or explicit (irregular, or geolocated), depending on the uniformity of the grid. *See* Regular data grid, Geolocated data grid.

Datum (vertical)

A base reference level for establishing the vertical dimension of elevation for the earth's surface. A datum defines sea level and incorporates an ellipsoid; thus one can reference a coordinate system to a datum or to a specified ellipsoid, but not both at the same time.

Datum (horizontal)

A base measuring point ("0.0 point") used as the origin of rectangular coordinate systems for mapping or for maintaining excavation provenience. Two examples are the North American Datum of 1927 (NAD27) and the North American Datum of 1983 (NAD83). Earth-centered coordinate systems, such as WGS84, combine horizontal and vertical datums.

Dead reckoning
From "deduced reckoning," the estimation of geographic position based on course, speed, and time.

DEM (Digital Elevation Map/Model)
Elevation data in the form of a data grid, generally a regular (implicit) one. DEM also refers to the five primary types of digital elevation models produced by the U.S. Geological Survey; Mapping Toolbox functions can read 30-meter and 10-meter DEMs as well as 3-second DEMs.

Departure
The arc length distance along a parallel of a point from a given meridian.

Developable surface
A simple geometric form capable of being flattened without stretching. Many map projections can be grouped by a particular developable surface: cylinder, cone, or plane.

Direct aspect
See Normal aspect.

Display Structure
A Mapping Toolbox data structure for mapped objects dating from Version 1 of the product. The structures can contain line, patch, text, regular data grid, geolocated data grid, and light objects; vector data always has coordinates in latitude and longitude. In Version 2 of the toolbox, display structures were superseded by *geostructs* and *mapstructs*. *See* **Geographic data structure** on page Glossary-9. Most vector display structures can be converted to geographic data structures.

Distortion
A variation of the area or linear scale on a map from that indicated by the stated map scale, or the variation of a shape or angle on a map from the corresponding shape or angle on the Earth.

DM

Degrees-minutes angle notation of the form ddd° mm'. There are 60 seconds in a minute, and 60 minutes in a degree. In DM notation, degrees are always integer, but minutes can be fractional. Certain Mapping Toolbox functions represent DM angles as column vectors, `[degrees minutes]`.

DMS

Degrees-minutes-seconds angle notation of the form ddd° mm' ss". There are 60 seconds in a minute, and 60 minutes in a degree. In DMS notation, degrees and minutes are always integer, but seconds can be fractional. Certain Mapping Toolbox functions represent DMS angles as column vectors, `[degrees minutes seconds]`.

DTED

Digital Terrain Elevation Data; a raster data format used by various terrain data products, based on specifications originating in the United States Department of Defense. DTED® files sample terrain elevation in a geographic grid at specific "levels" of spatial resolution; sampling intervals are approximately one kilometer for Level 0, 100 meters for Level 1, 30 meters for Level 2, and so on. High level DTED files are not generally available to the public. Mapping Toolbox software imports all levels of DTED data.

Easting

The distance of a point eastward from the origin in the units of the coordinate system for the defined projection. Paired with Northings.

Ellipsoid

When used to represent the Earth, a solid geometric figure formed by rotating an ellipse about its minor (shorter) axis. Also called spheroid.

Ellipsoid vector

A vector describing a specific ellipsoid model. The ellipsoid vector has the form

```
ellipsvec = [semimajor-axis eccentricity]
```

Ellipsoidal height

Elevation of a point above a reference ellipsoid, as measured along a normal to the ellipsoid.

Equal-area projection

A projection on which the areas of all regions are shown in the same proportion to their true areas. Shapes may be greatly distorted. Also called an equivalent or authalic projection.

Equator

The great circle straddling a planet at a latitude of 0°, perpendicular to its polar axis and midway along it, dividing the northern and southern hemispheres.

Equatorial aspect

An aspect of an azimuthal projection on which the center of projection or origin is some point along the Equator. For cylindrical and pseudocylindrical projections, this aspect is usually called conventional, direct, normal, or regular rather than equatorial.

Equidistant projection

A projection that maintains constant scale along all great circles from one or two points. When the projection is centered on a pole, the parallels are spaced in proportion to their true distances along each meridian.

Equireal projection

See Equal-area projection.

Equivalent projection

See Equal-area projection.

False easting

The value of the easting assigned to the projection origin. Easting values increase to the east.

False northing

The value of the northing assigned to the projection origin. Northing values increase to the north.

Flat-polar projection

A cylindrical projection on which, in normal aspect, the pole is shown as a line rather than as a point. For example, the Miller projection is flat-polar.

Frame

See Map frame.

Free of distortion

Having no distortion of shape, area, or linear scale. On a flat map, this condition can exist only at certain points or along certain lines.

Geodesic

A minimum-distance curve on a curved surface, independent of the choice of a coordinate system. On a sphere a geodesic is equivalent to a great circle arc.

Geolocated data grid

A data grid defined with separate latitude, longitude, and value matrices, allowing irregular sampling, nonrectangular shapes, and noncardinal orientations. Satellite imagery swaths are often represented as geolocated data grids. *See* Data grid, Regular data grid.

Geodata

Geospatial data. *See* Geospatial.

Geoid

The figure of the earth less its topography, defined as an equipotential surface with respect to gravity, more or less corresponding to mean sea level. It is approximately an oblate ellipsoid, but not exactly so because local variations in gravity create minor hills and dales. Empirically determined geoids are used to define *datums* and to compute orbital mechanics.

Geometric projection

See Perspective projection.

Geographic coordinates

Spherical 2-D coordinate tuples (latitudes, longitudes) that specify point locations for unprojected geodata. The analogous term for geodata projected to a rectangular coordinate system is *map coordinates*.

Geographic data structure

A Mapping Toolbox data structure for vector data comprised of a MATLAB structure array with one element per vector geographic feature. It includes a mandatory `Geometry` field, at least two coordinate array fields. The field names are `X` and `Y` (for *mapstructs*), or `Lat` and `Lon` (for *geostructs*), and optional attribute fields.

Georeferencing

Identifying objects and locations by name, identifier, or coordinates to describe where they are located on the Earth's surface.

Geospatial

Spatial data, concepts, and techniques that specifically refer to geographic space or phenomena, and not just to arbitrary coordinate systems or abstract space frames.

Geostruct

A Mapping Toolbox geographic data structure for vector geodata with coordinates in latitude and longitude. *See* **Geographic data structure** on page Glossary-9.

GeoTIFF

An extension of the TIFF image file format with additional tags containing parameters for image georeferencing and projected map coordinate system definition.

GIS (Geographic Information System)

A system, usually computer based, for the input, storage, retrieval, analysis, and display of interpreted geographic data.

Globular projection

Generally, a nonazimuthal projection developed before 1700 on which a hemisphere is enclosed in a circle, and meridians and parallels are simple curves or straight lines.

Graticule

A network of lines representing a subset of the Earth's parallels and meridians (or plane coordinates) used as a reference grid on globes and maps. Generally synonymous with *map grid*, except that many map grids are rulings at regular intervals in projected coordinates. *See* Map grid, National grid (U.S.), and National grid (U.K.). The vertices of the graticule grid are precisely projected, and the map data contained in any grid cell is warped to fit the resulting quadrilateral. A finer graticule grid results in a higher projection fidelity at the expense of greater computational requirements.

Great circle

Any circle on the surface of a sphere, especially when the sphere represents the Earth, formed by the intersection of the surface with a plane passing through the center of the sphere. It is the shortest path between any two points along the circle and therefore important for navigation. All meridians and the Equator are great circles on the Earth taken as a sphere.

Grid

See Map grid, Data grid.

Homalographic/homolographic projection

See Equal-area projection.

Hydrography

The science of measurement, description, and mapping of the surface waters of the Earth, especially with reference to their use in navigation. The term also refers to those parts of a map collectively that represent surface waters and drainage.

Hydrology

The scientific study of the waters of the Earth, especially with relation to the effects of precipitation and evaporation upon the occurrence and character of ground water.

Hypsographic tints

A graphic means of representing terrain or other scalar attributes using a sequence of colors or tints indexed to elevation.

Hypsography

The scientific study of the Earth's topological configuration above sea level, especially the measurement and mapping of land elevation.

Index map

A small-scale map used to help locate a map containing a region or feature of interest in a tiled geospatial database, map series, plat book, or atlas.

Indicatrix

A circle or ellipse useful in illustrating the distortions of a given map projection. Indicatrices are constructed by projecting infinitesimally small circles on the Earth onto a map and giving them visible dimensions. Their axes lie in the directions of and are proportional to the maximum and minimum scales at their point locations. Often called a Tissot indicatrix after the originator of the concept. Mapping Toolbox Tissot indicatrices can be displayed using the `tissot` command, and indicatrices for all supported projections are provided. See Chapter 15, "Map Projections — Alphabetical List" in Mapping Toolbox reference documentation.

Interrupted projection

A projection designed to reduce peripheral distortion by making use of separate sections joined at certain points or along certain lines, usually the Equator in the normal aspect, and split along lines that are usually meridians. There is normally a central meridian for each section. No Mapping Toolbox projections are of this type, but the user can separate data into sections and project these independently to achieve this effect.

Keyhole markup language (KML)

A file format and dialect of XML used to georeference geographic locations and describe their attributes and relations, including hyperlinks, for display in earth browsers.

Large-scale mapping

Mapping at a scale larger than about 1:75,000, although this limit is somewhat flexible. Includes cadastral, utility, and some topographic maps.

Latitude (astronomical)

The complement of the elevation angle of the celestial North Pole, which depends on normal to the Earth's equipotential surface (geoid) at a given point (positive if the point is north of Equator, negative if it is south). It can be thought of as the angle that a plumb line makes with the equatorial plane.

Latitude (auxiliary)

Intermediate forms of latitude that are mathematically constructed (normally by transferring latitudes first from an ellipsoid to a sphere, and then to a plane) in order to achieve desired map projection properties. Types include *conformal* (for constructing conformal maps), *authalic* (for constructing equal-area maps), and *rectifying* (for constructing equidistant maps).

Latitude (geocentric)

The angle at which a line connecting the surface of a sphere or reference ellipsoid to its center intersects the equatorial plane (positive if the point is north of Equator, negative if it is south). One of the two common geographic coordinates of a point on the Earth.

Latitude (geodetic)

The angle made by a perpendicular to a given point on the surface of a sphere or ellipsoid representing the Earth and the plane of the Equator (positive if the point is north of Equator, negative if it is south). Also called *geographic latitude*. One of the two common geographic coordinates of a point on the Earth.

Latitude of opposite sign

See Parallel of opposite sign.

Legs

Line segments connecting waypoints.

Legend

See Map legend.

Limiting forms

The form taken by a system of projection when the parameters of the formulas defining that projection are allowed to reach limits that cause it to be identical with another separately defined projection.

Logical data grid

A binary data grid consisting entirely of 1s and 0s. An example of a logical data grid can be created with the `topo` map by performing a logical test for positive elevations (`topo>0`). Each entry in the data grid contains a 1 if it is above sea level, or a 0 if it is at or below sea level.

Longitude

The angle made by the plane of a meridian passing through a given point on the Earth's surface and the plane of the (prime) meridian passing through Greenwich, England, east or west to 180 (positive if the point is east, negative if it is west). One of the two common geographic coordinates of a point on the Earth. Paired with *Latitude*.

Loxodrome

See Rhumb line.

Map

A diagrammatic or pictorial representation of a planet's surface or part of it, showing the geographical distributions, positions, etc., of natural or artificial features such as roads, towns, relief, land cover, rainfall, populations, etc. Maps represent geospatial data visually.

Map axes

A Handle Graphics axes object augmented with additional properties, including a projection type, projection parameters, map latitude and longitude limits and so forth. Many Mapping Toolbox display functions require that a map axes first be defined. Others create a map axes if necessary (e.g., `worldmap` and `usamap`) or assume that your coordinate data are in a projected map coordinate system (`mapshow` and `mapview`).

Map coordinates

Orthogonal planar 2-D coordinate tuples that specify point locations for projected geodata. The analogous term for unprojected geodata is geographic coordinates. Also called grid coordinates and plane coordinates.

Map frame
A projected rectangle or quadrangle enclosing a geographic data displayed on a Mapping Toolbox map axes.

Map grid
A symbolized network of lines, or graticule, representing parallels and meridians or plane coordinates. Plane coordinate grids are almost always rectangular with uniform spacing. Azimuthal map grids are organized as polar coordinates. *See* Graticule.

Map layer
A vector or raster geographic data set read into the Map Viewer, for example, roads, rivers, municipal boundaries, topographic grids, or orthophoto images. Map layers are “stacked” from top to bottom, and can be reordered and hidden by the user.

Map legend
A key to symbolism used on a map, usually containing swatches of symbols with descriptions, and can include notes on projection, provenance, scale, units of distance, etc.

Mapstruct
A Mapping Toolbox geographic data structure for vector geodata with coordinates in projected (*x,y*) coordinates. *See* **Geographic data structure** on page Glossary-9.

Matrix map
See Data grid.

Meridian
A reference line on the Earth’s surface formed by the intersection of the surface with a plane passing through both poles and some third point on the surface. This line is identified by its longitude. When the Earth is regarded as a sphere, this line is half a great circle; on the Earth regarded as an ellipsoid, it is half an ellipse.

Minimum-error projection
A projection having the least possible total error of any projection in the designated classification, according to a given mathematical criterion. Usually, this criterion calls for the minimum sum of squares

of deviations of linear scale from true scale throughout the map ("least squares").

National grid (U.K.)

A metric grid based on the Transverse Mercator Projection developed by Ordnance Survey in 1936 for use in Great Britain. Sometimes abbreviated "OSGB36," it is the de facto standard projection for display of UK based mapping.

National grid (U.S.)

A metric grid based on the Transverse Mercator Projection, adopted by the Federal Geographic Data Committee (FGDC) in 2001 for use in the United States. It is an evolving standard intended to unify georeferencing across the U.S., but is not yet as widely used as other countries' national grids.

Nominal scale

The stated scale at which a map projection is constructed. Scale is never completely constant across the extent of a map, although in some maps (especially at large scales) it can vary by minuscule amounts.

Normal aspect

A form of a projection that provides the simplest graticule and calculations. It is the polar aspect for azimuthal projections, the aspect having a straight Equator for cylindrical and pseudocylindrical projections, and the aspect showing straight meridians for conic projections. Also called conventional, direct, or regular aspect.

Northing

The distance of a point northward from the origin, in the units of the coordinate system for the defined projection. Paired with Eastings.

Numerical geodata

Geospatial data in which raster pixel values (or vector data attributes) are cardinal, ratio, or ordinal numeric measurements or computed values. For example, the `topo` data set contains numerical geodata. Each value in its data grid is an average elevation in meters for the geographic area covered by that cell. *See* Categorical geodata.

Oblique aspect

An aspect of a projection on which the axis of the Earth is rotated so it is neither aligned with nor perpendicular to the conceptual axis of the map projection.

Orthoapsidal projection

A projection on which the surface of the Earth taken as a sphere is transformed onto a solid other than the sphere and then projected orthographically and obliquely onto a plane for the map.

Orthographic projection

A specific azimuthal projection or a type of projection in which the Earth is projected geometrically onto a surface by means of parallel projection lines.

Orthometric height

Elevation above a datum defined by a geoid representing mean sea level.

Orthomorphic projection

See Conformal projection.

Parallel

A small circle on the surface of the Earth, formed by the intersection of the surface of the reference sphere or ellipsoid with a plane parallel to the plane of the Equator. This line is identified by its latitude, which can be defined in several ways. The Equator (a great circle) is usually also treated as a parallel. *See* entries for Latitude.

Parallel of opposite sign

A parallel that is equally distant from but on the opposite side of the Equator. For example, for lat 30°N (or +30°), the parallel of opposite sign is lat 30° S (or -30°). Also called latitude of opposite sign.

Perspective projection

A projection produced by projecting straight lines radiating from a selected point (or from infinity) through points on the surface of a sphere or ellipsoid and then onto a tangent or secant plane. Other perspective maps are projected onto a tangent or secant cylinder or cone by using straight lines passing through a single axis of the sphere or ellipsoid. Also called geometric projection.

Planar projection

A projection resulting from the conceptual projection of the Earth onto a tangent or secant plane. Usually, a planar projection is the same as an azimuthal projection. Mathematically, the projection is often only partially geometric.

Planimetric map

A map representing only the horizontal positions of features (without their elevations).

Polar aspect

An aspect of a projection, especially an azimuthal one, on which the Earth is viewed from directly above a pole. This aspect is called *transverse* for cylindrical or pseudocylindrical projections.

Pole

An extremity of a planet's axis of rotation. The North Pole is a singular point at 90°N for which longitude is ambiguous. The South Pole has the same characteristics and is located at 90°S.

Polyconic projection

A specific projection or member of a class of projections that are constructed like conic projections but with different cones for each parallel. In the normal aspect, all the parallels of latitude are nonconcentric circular arcs, except for a straight Equator, and the centers of these circles lie along a central axis.

Projected coordinate system

A coordinate system defined for a particular map projection and associated parameters, which normally is planar with well-defined coordinate origin, handedness, nominal scale, and units of distance. While map scale can vary at different coordinate locations, a linear projected coordinate system has constant units of distance.

Projection

A systematic representation of a curved 3-D surface such as the Earth onto a flat 2-D plane. Each map projection has specific properties that make it useful for specific purposes. For a list of Mapping Toolbox map projections, type `maps`.

Projection parameters

The values of constants as applied to a map projection for a specific map; examples are the values of the scale, the latitudes of the standard parallels, and the central meridian. The required parameters vary with the projection.

Pseudoconic projection

A projection that, in the normal aspect, has concentric circular arcs for parallels and on which the meridians are equally spaced along the parallels, like those on a conic projection, but on which meridians are curved.

Pseudocylindric-al projection

A projection that, in the normal aspect, has straight parallel lines for parallels and on which the meridians are (usually) equally spaced along parallels, as they are on a cylindrical projection, but on which the meridians are curved.

Quadrangle

A region bounded by parallels north and south, and meridians east and west.

Raster geodata

A georeferenced array or grid of values corresponding to specific geographic points, usually regularly and rectangularly sampled in either geographic or map space. Values can be continuous or categorical. In the case of georeferenced multiband images, raster geodata can take the form of 3- and higher dimensional arrays.

Reckoning

The determination of geographic position by calculation.

Referencing matrix

A 3-by-2 matrix defining the scaling, orientation, and placement of raster map data on the globe or in planar map coordinates. The matrix specifies an affine transformation that ties (geolocates) 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). *See* Referencing vector.

Referencing vector

A three-component vector defining the geographic placement and unit cell size for raster map data. A referencing vector has the form `[cells/degree north-latitude west-longitude]`, with latitude and longitude limits specified in degrees.

A referencing vector specifies an affine transformation with rows and columns aligned to latitude and longitude, respectively, and the same data spacing in both latitude and longitude. As such, it is more specific than a referencing matrix. Note that a referencing vector can always be transformed to a referencing matrix, but only certain referencing matrices can be transformed to referencing vectors. *See* Referencing matrix.

Regional map

A small-scale map of an area covering at least 5 or 10 degrees of latitude and longitude but less than a hemisphere.

Regular aspect

See Normal aspect.

Regular data grid

A data grid with equally spaced grid points in either latitude-longitude or map coordinates, defined with a referencing matrix or vector, and limited to a rectangular shape and cardinal orientation. *See* Data grid, Geolocated data grid, Referencing matrix.

Retroazimuthal projection

A projection on which the direction or azimuth from every point on the map to a given central point is shown correctly with respect to a vertical line parallel to the central meridian. The reverse of an azimuthal projection.

Rhumb line

A complex curve (a spherical helix) on a planet's surface that crosses every meridian at the same oblique angle; a navigator can proceed between any two points along a rhumb line by maintaining a constant heading. A rhumb line is a straight line on the Mercator projection. Also called a loxodrome.

Scale

The ratio of the distance on a map or globe to the corresponding distance on the Earth; usually stated in the form 1:5,000,000, for example. A given region will appear smaller on a small-scale map than on a large-scale map.

Scale factor

The ratio of the scale at a particular location and direction on a map to the nominal scale of the map. At a standard parallel, or other standard line, the scale factor is 1.0.

Secant cone, cylinder, or plane

A secant cone or cylinder intersects the sphere or ellipsoid along two separate lines; these lines are parallels of latitude if the axes of the geometric figures coincide. A secant plane intersects the sphere or ellipsoid along a line that is a parallel of latitude if the plane is at right angles to the axis.

Selector

A cell array in which the first element is a predicate function and the remaining elements list the names of attributes in a shapefile. Function `shaperead` has an option to screen out any feature in the shapefile for which a predicate returns false when applied to the subset of attributes corresponding to the list in the selector.

Shaded relief

Shading added to a map or image that makes it appear to have three-dimensional aspects. This type of enhancement is commonly done to satellite images and thematic maps utilizing digital topographic data to provide the appearance of terrain relief.

Shapefile

A widely used file format for vector geodata designed by Environmental Systems Research Institute. Shapefiles encode coordinates for points, multipoints, lines (polylines), or polygons along with tabular attributes.

Singular points

Certain points on most but not all conformal projections at which conformality fails, such as the poles on the normal aspect of the Mercator projection.

Skew-oblique aspect

An aspect of a projection on which the axis of the Earth is rotated, so it is neither aligned with nor perpendicular to the conceptual axis of the map projection, and tilted, so the poles are at an angle to the conceptual axis of the map projection.

Small circle

A circle on the surface of a sphere, formed by the intersection with a plane. Parallels of latitude are small circles on the Earth taken as a sphere. Mapping Toolbox great circles, including the Equator and all meridians, are treated as special, limiting cases of small circles. Mapping Toolbox functions generalize the concept of small circle with computations for two other types of curve: the locus of points on an ellipsoid at a given distance (as measured along a geodesic) from a central point, or the locus of points on a sphere or ellipsoid at a given distance from a central point, as measured along a rhumb line.

Small-scale mapping

Mapping at a scale smaller than about 1:1,000,000, although the limiting scale sometimes has been made as large as 1:250,000.

Spatial Data Transfer Standard (SDTS)

A self-documenting geospatial file formatting standard adopted by the U.S. government and others. SDTS can encode locations, attributes, topological relationships, data quality, and other metadata. Note that Mapping Toolbox software can read the SDTS Raster Profile, but does not currently support SDTS vector data.

Spheroid

See Ellipsoid.

Standard parallel

In the normal aspect of a projection, a parallel of latitude along which the scale is as stated for that map. There are one or two standard parallels on most cylindrical and conic map projections and one on many polar stereographic projections.

State Plane

A set of commensurate coordinate systems commonly used for utility and surveying applications in the lower 48 United States. Each state contains one or more zones. Coordinates for zones elongated north-to-south are based on Transverse Mercator projections, while zones elongated east-to-west use Lambert Conformal Conic.

Stereographic projection
A specific azimuthal projection or type of projection in which the Earth is projected geometrically onto a surface from a fixed (or moving) point on the opposite face of the Earth.

Symbolization
In cartography, a mapping between geospatial objects or numerical or categorical values and cartographic symbols. The choice of graphic symbols, their size, density, shape, contrast, color, and pattern are principal aspects of symbolization.

Symbolspec
(Symbol specification) A cell array structure that defines symbolism characteristics for points, lines, and polygons with respect to attributes and their values, or as a default symbolization regardless of attributes.

Tangent cone or cylinder
A cone or cylinder that just touches the sphere or ellipsoid along a single line. This line is a parallel of latitude if the axes of the geometric figures coincide.

Thematic map
A map designed to portray primarily a particular subject, such as population, railroads, or croplands.

Tissot indicatrix
See Indicatrix.

Topographic map
A map that usually represents the vertical positions or elevations of features as well as their horizontal positions.

Transformed latitudes, longitudes, or poles
Graticule of meridians and parallels on a projection after the Earth has been turned with respect to the projection so that the Earth's axis no

longer coincides with the conceptual axis of the projection. Used for oblique and transverse aspects of many projections.

Transverse aspect

An aspect of a map projection on which the axis of the Earth is rotated so that it is at right angles to the conceptual axis of the map projection. For azimuthal projections, this aspect is usually called *equatorial* rather than transverse.

True scale

See Correct scale.

Vector data set

Data representing geospatial objects as sequences of geographic or projected coordinate points that are implicitly connected if they represent linear or areal shapes. In Mapping Toolbox and other software, such geodata is often represented by two vectors, one with latitudes, another with longitudes. Objects can be segmented by inserting NaNs at the same locations in both vectors. Such pairs of coordinate vectors can also be represented as the `Lat` and `Lon` or `X` and `Y` field values in a geographic data structure array.

Viewshed

The portion of a surface that is visible from a given point on or above it; derived from the concept of a watershed.

Waypoints

Points through which a trip, track, or transit passes, usually corresponding to course or speed changes.

WGS 72 (World Geodetic System 1972)

An Earth-centered datum, used as a definition of DMA (now NGA) DEMs. The WGS 72 datum was the result of an extensive effort extending over approximately three years to collect selected satellite, surface gravity, and astrogeodetic data available throughout 1972. This data was combined using a unified WGS solution (a large-scale least squares adjustment).

WGS 84 (World Geodetic System 1984)

The WGS 84 was developed as a replacement for the WGS 72 by the military mapping community as a result of new and more accurate instrumentation and a more comprehensive control network of ground stations. The newly developed satellite radar altimeter was used to deduce geoid heights from oceanic regions between 70° north and south latitude. Geoid heights were also deduced from ground-based Doppler and ground-based laser satellite-tracking data, as well as surface gravity data. The ellipsoid associated with WGS 84 is GRS 80.

World file

A small text file used to georeference different raster image formats, developed to incorporate imagery into ESRI's ArcView software.

Zenithal projection

See Azimuthal projection.

Bibliography

1 Snyder, J.P., *Map Projections — A Working Manual*, U.S. Geological Survey Professional Paper 1395, Washington, D.C., 1987.

2 Maling, D.H., *Coordinate Systems and Map Projections*, 2nd Edition, Pergamon Press, New York, NY, 1992.

3 Snyder, J.P., and Voxland, P.M., *An Album of Map Projections*, U.S. Geological Survey Professional Paper 1453, Washington, D.C., 1994.

4 Snyder, J.P., *Flattening the Earth — 2000 Years of Map Projections*, University of Chicago Press, Chicago, IL, 1993.

5 U.S. National Geospatial Intelligence Agency, “Military Specification: Digital Chart of the World (DCW)”, MIL-D-89009, 13 April 1992.

Examples

Use this list to find examples in the documentation.

Your First Maps

Understanding Vector Geodata

Raster Geodata

Combining Vector and Raster Geodata

Understanding Raster Data

Geolocated Data Grids

Exporting Vector Geodata

Creating and Viewing Maps

Making Three-Dimensional Maps

Using Cartesian MATLAB Display Functions

Using Colormaps and Colorbars

Vector Data Manipulation

Raster Data Manipulation

Projections and Transformations

Web Map Service Maps

Index