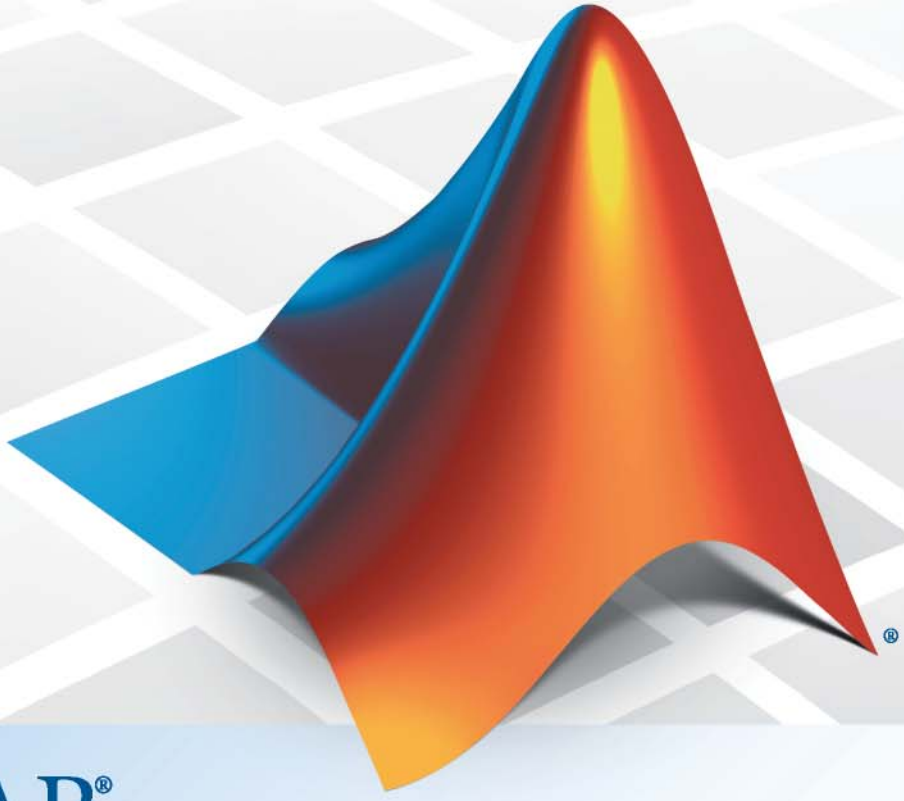
Mapping Toolbox™ 3
User's Guide



MATLAB®

MathWorks®
Accelerating the pace of engineering and science

How to Contact 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

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

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)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.2 (Release 2010b)

Contents

1 Getting Started

2 Understanding Map Data

3 Understanding Geospatial Geometry

4 Creating and Viewing Maps

5 Making Three-Dimensional Maps

6 Customizing and Printing Maps

Manipulating Geospatial Data

7

Using Map Projections and Coordinate Systems

8

9 Creating Web Map Service Maps

10 Mapping Applications

11 Map Projections Reference

1

Getting Started

Product Overview

Mapping Toolbox™ provides tools and utilities for analyzing geographic data and creating map displays. You can import vector and raster data from shapefile, GeoTIFF, SDTS DEM, and other file formats, as well as Web-based data from Web Map Service (WMS) servers. The toolbox lets you customize the imported data by subsetting, trimming, intersecting, adjusting spatial resolution, and applying other methods. Geographic data can be combined with base map layers from multiple sources in a single map display. With function-level access to all key features, you can automate frequent tasks in your geospatial workflow.

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

- Vector and raster data import and export from standard formats and specific data products
- Data retrieval from Web Map Service (WMS) servers for customized geographic datasets and related metadata
- Digital terrain and elevation model analysis functions, including profile, gradient, line-of-sight, and viewshed calculations
- Geometric geodesy, including distance and area calculations, 3D coordinate transformations, and more than 65 map projections
- Utilities for converting units, adjusting spatial resolution, wrapping longitudes, and managing spatially referenced images and raster data
- 2D and 3D map display, customization, and interaction

This chapter provides step-by-step examples of basic Mapping Toolbox capabilities and guides you toward demos and documentation that can help answer your questions. For a complete classified list of Mapping Toolbox functions and features, see “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 Toolbox users. MathWorks 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 map and display geodata. It introduces `worldmap` and other basic functions, and then describes how to use the Map Viewer (`mapview`).

See the World

Spatial data refers to data describing location, shape, and spatial relationships. *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.

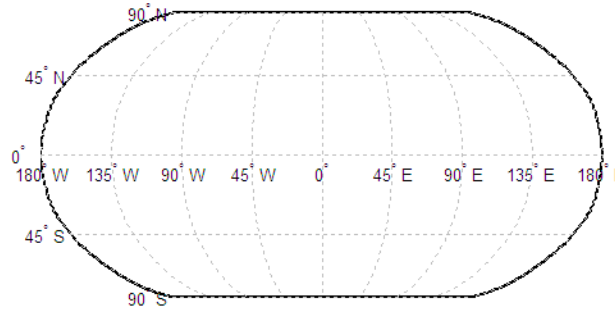
Geospatial data can be voluminous, complex, and difficult to process. Mapping Toolbox functions handle many of the details of loading and displaying data, and built-in data structures facilitate data storage. Nevertheless, the more you understand about your data and the capabilities of the toolbox, the more interesting applications you can pursue, and the more useful your results will be to you and others.

Follow this example to create your first world map.

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 selects a reasonable choice for your map projection and coordinate limits. In this case, it chose a Robinson projection centered on the prime meridian and the equator (0º latitude, 0º longitude).

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—a set of discrete vertices that, when connected in the order given, approximate the coastlines of continents, major islands, and inland seas. The vertex latitudes and longitudes are stored as MATLAB vectors in a MAT-file. First, list the variables in the file:

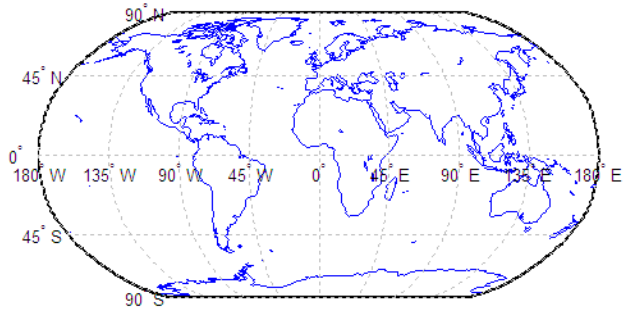
```
whos -file coast.mat
```

The output is shown below:

```
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, transforms them to *x* and *y* via a specified map projection, and displays them in a figure axes. In this example, `worldmap` specifies the projection.

Note Certain Mapping Toolbox functions that end with `m`, such as `plotm` and `textm`, are modeled after familiar MATLAB functions that handle non-geographic coordinate data.

---

4 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 display breaks into separate parts like this because in the vectors `lat` and `long` the vertices of various coastlines are concatenated together but separated by isolated NaN-valued elements. In other words, "NaN separators" implicitly divide each vector into multiple parts. `lat` and `long` include "NaN terminators" as well as separators, showing that the `coast` data set is organized into precisely 241 polygons.

Enter the following code to break out your data into its NaN-separated parts:

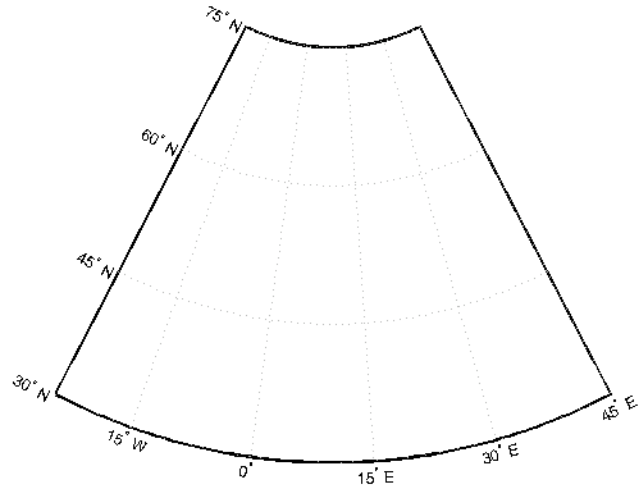
```
[latcells, loncells] = polysplit(lat, long);
numel(latcells)
```

`latcells` and `loncells` are cell vectors, with each cell containing the vertices of just one polygon part. The number of parts appears:

```
ans =
   241
```

5 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 access and modify these properties.

6 To inspect the `MapProjection` property for the map of Europe, type:

```
getm(h, 'MapProjection')
```

The type of projection appears:

```
ans =
eqdconic
```

If you're not familiar with the abbreviation "eqdconic," type:

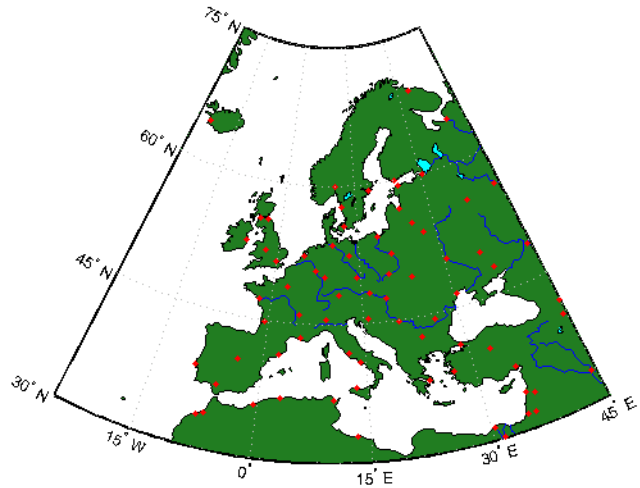
```
help eqdconic
```

To see all the map-specific properties for this map axes, type:

```
getm(h)
```

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

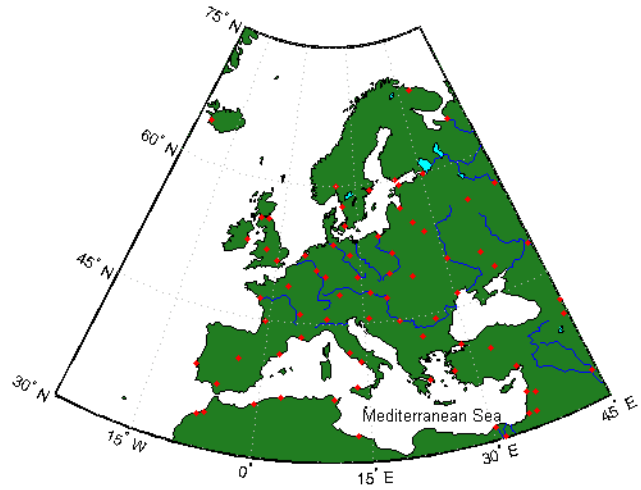
```
geoshow('landareas.shp', 'FaceColor', [0.15 0.5 0.15])
geoshow('worldlakes.shp', 'FaceColor', 'cyan')
geoshow('worldrivers.shp', 'Color', 'blue')
geoshow('worldcities.shp', 'Marker', '.',...
                           'Color', 'red')
```



Note how `geoshow` can plot data directly from files onto a map axes without first importing 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')
```



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.

Tour Boston with the Map Viewer

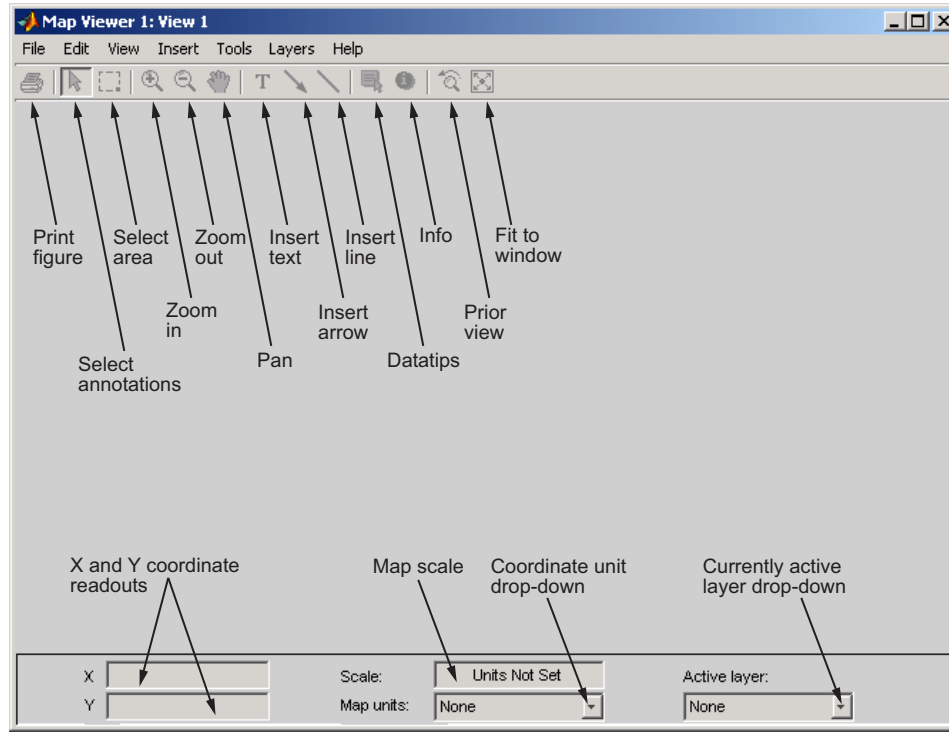
The Map Viewer is an interactive tool for browsing map data. With it you can assemble layers of vector and raster geodata and render them in 2-D. You can import, reorder, symbolize, hide, and delete data layers; identify coordinate locations; and list data attributes. You can display selected data attributes 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 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.



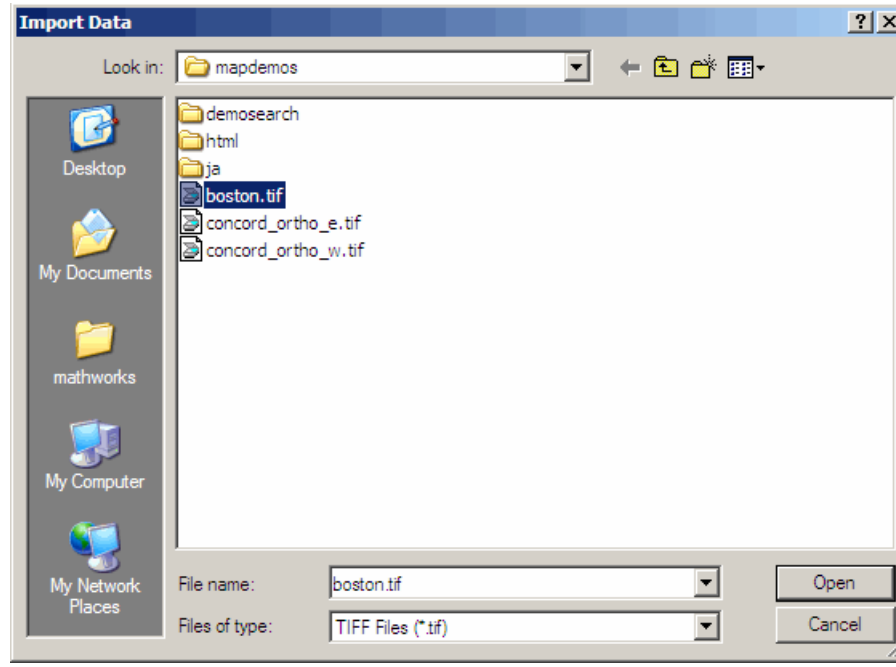
Most of the tool buttons can also be activated from the **Tools** menu.

In the Map Viewer graphic, you can see arrows pointing to the X and Y coordinate readouts. The Map Viewer is designed primarily for working with data sets that refer to a projected map coordinate system (as opposed to a geographic, latitude-longitude system), so the coordinate axes are named X and Y.

2 For ease in importing Mapping Toolbox demo data, set your working directory. Type the following in the Command Window:

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

3 You can also navigate to this directory with the Map Viewer Import Data dialog if you prefer. In the Map Viewer, select the **File** menu and then choose **Import From File**. Open the GeoTIFF file `boston.tif`, as shown below.

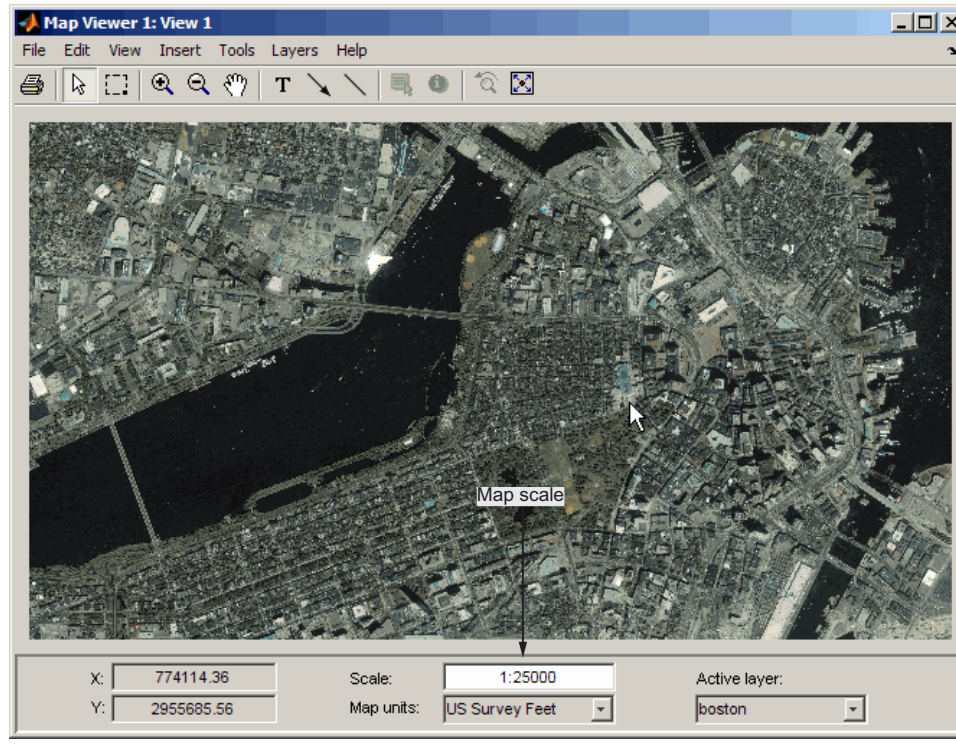


The file opens in the Map Viewer. The image is a visible red, green, and blue composite from a georeferenced IKONOS-2 panchromatic/multispectral product created by GeoEye™. Copyright © GeoEye, all rights reserved. For further information about the image, refer to the text files `boston.txt` and `boston_metadata.txt`.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

```
open 'boston.txt'
```

4 To see the map scale in the Map Viewer, 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 Map Viewer now looks like this.



The cursor in the picture has been placed so that it points 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, enter the following code to import a vector data layer, the streets and highways in the central Boston area:

```
boston_roads = shaperead('boston_roads.shp');
```

The `shaperead` function reads the line shapefile `boston_roads.shp` into the workspace as a *geographic data structure*. As is frequently the case

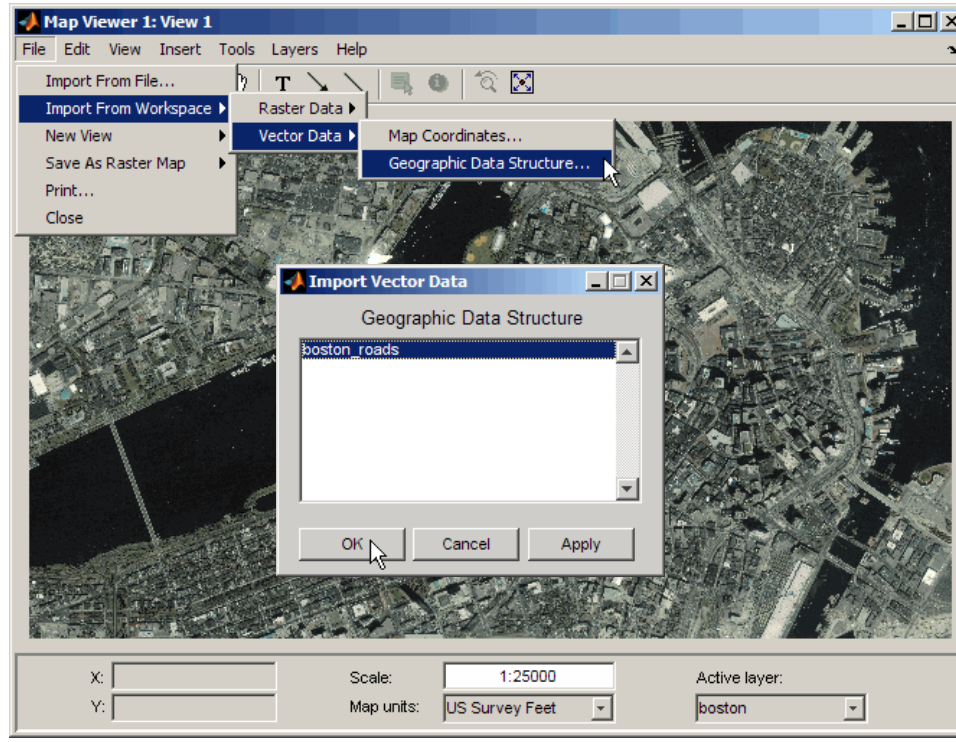
when overlaying geodata, the coordinate system used by `boston_roads.shp` (in units of meters) does not completely agree with the one for the satellite image, `boston.tif` (in units of feet). If you were to ignore this, the two data sets would be out of registration by a large distance.

7 Convert the `X` and `Y` coordinate fields of `boston_roads.shp` from meters to U.S. survey feet:

```
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
```

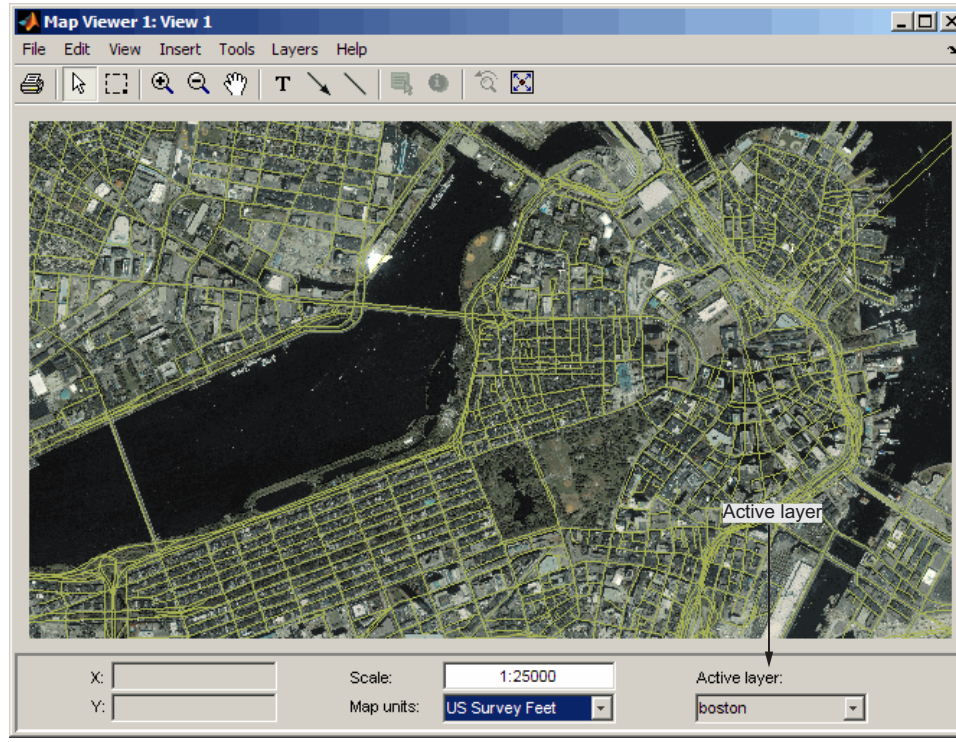
The `unitsratio` function computes conversion factors between a variety of units of length.

8 In the Map Viewer **File** menu, select **Import From Workspace > Vector Data > Geographic Data Structure**. Specify `boston_roads` as the data to import from the workspace, and click **OK**.



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

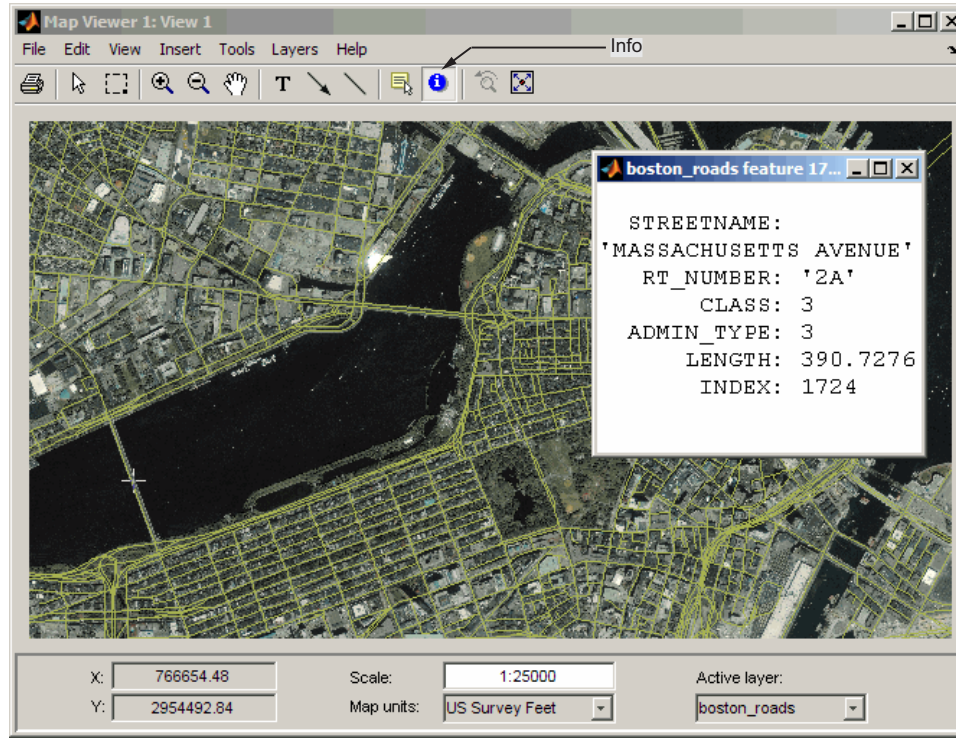
9 After the Map Viewer finishes importing the roads layer, it selects a random color and renders all the shapes with that color as solid lines. The view looks like this.



Being random, the color you see for the road layer may differ.

10 Use the **Active layer** drop-down menu at the bottom right to select `boston_roads`. Changing the active layer has no visual effect. Doing so allows you to query attributes of the layer you select. You can designate any layer to be the *active layer*; it does not need to be the topmost layer. By default, the first layer imported is active.

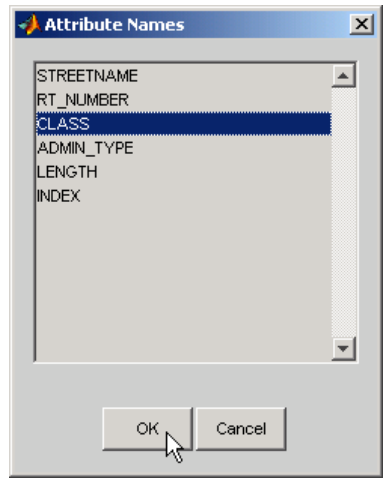
11 One way to see the attributes for a vector layer is to use the **Info** tool, a button near the right end of the toolbar. Select the **Info** tool and click somewhere along the bridge across the Charles River near the lower left of the map. This opens a text window displaying the attributes of the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



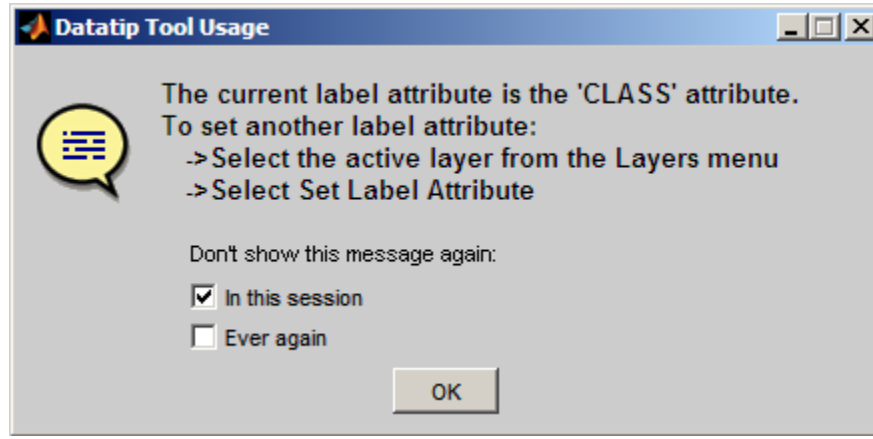
The selected road is Massachusetts Avenue (Route 2A). As the above figure shows, the `boston_roads` vectors have six attributes, including an implicit `INDEX` attribute added by the Map Viewer.

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

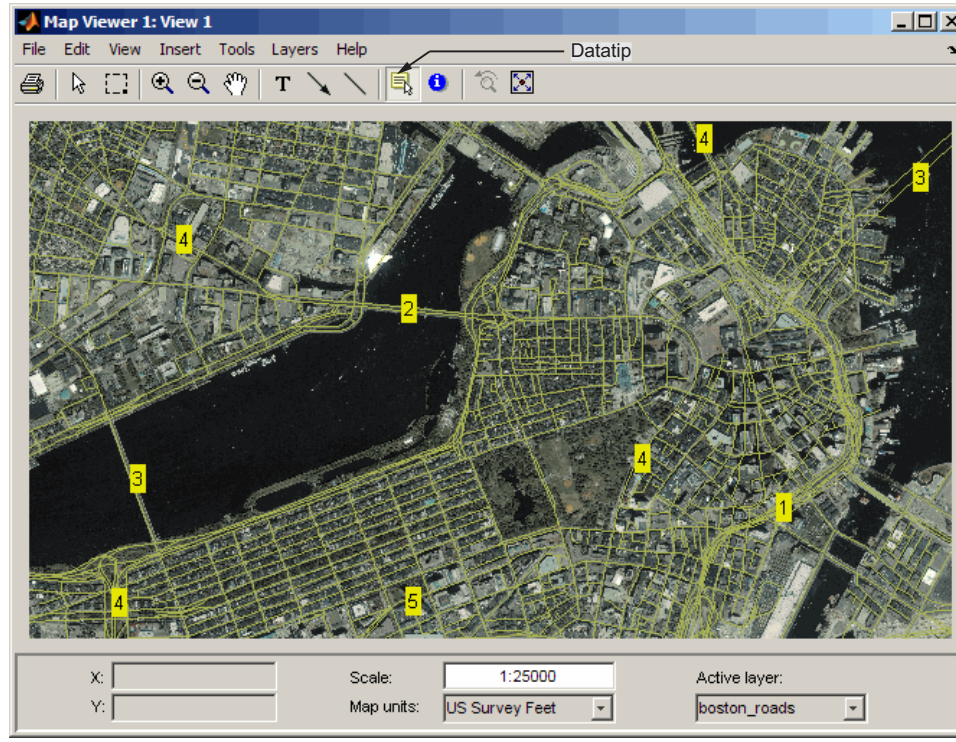
13 Choose an attribute for the **Datatip** tool to inspect. From the **Layers** menu, select **boston_roads > Set Label Attribute**. From the list in the Attribute Names dialog, select `CLASS` and click **OK** to dismiss it.



14 Select the **Datatip** tool. The cursor assumes a crosshairs (+) shape. A dialog box appears to remind you how to change attributes. Click **OK** to dismiss the box.



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



16 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:

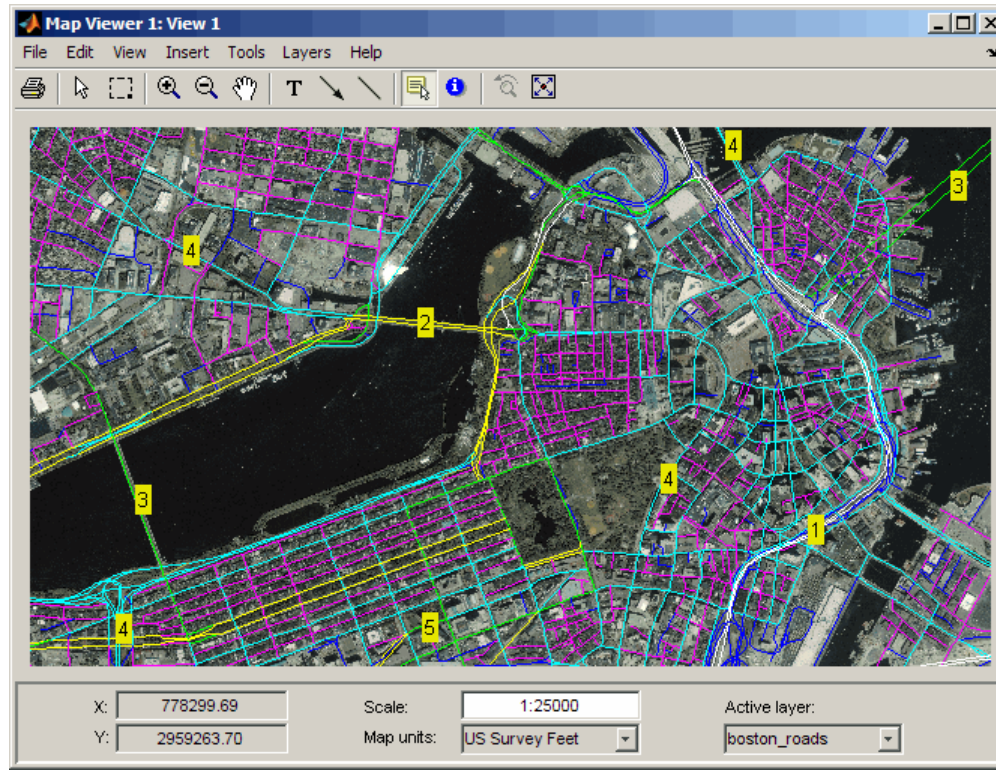
```
roadcolors = makesymbolspec('Line', ...
{'CLASS',1,'Color',[1 1 1]}, {'CLASS',2,'Color',[1 1 0]}, ...
{'CLASS',3,'Color',[0 1 0]}, {'CLASS',4,'Color',[0 1 1]}, ...
{'CLASS',5,'Color',[1 0 1]}, {'CLASS',6,'Color',[0 0 1]})
```

The following output appears:

```
roadcolors =
```

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

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



18 Remove the datatips before going on. To dismiss datatips, right-click one of them and select **Delete all datatips** from the pop-up context menu that appears.

19 Add another layer, a set of points that identifies 13 Boston landmarks. As you did with the `boston_roads` layer, import it from a shapefile:

```
boston_placenames = shaperead('boston_placenames.shp');
```

20 The locations for these landmarks are given in meters, so you must convert their coordinates to units of survey feet before importing them into Map Viewer:

```
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
```

21 From the **File** menu, select **Import From Workspace > Vector Data > Geographic Data Structure**. Choose `boston_placenames` as the data to import from the workspace and click **OK**.

22 The `boston_placenames` markers are symbolized as small `x` markers, but these markers do not show up over the orthophoto. To solve this problem, create a symbolspec for the markers to represent them as red filled circles. At the MATLAB command line, type:

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

The following output appears:

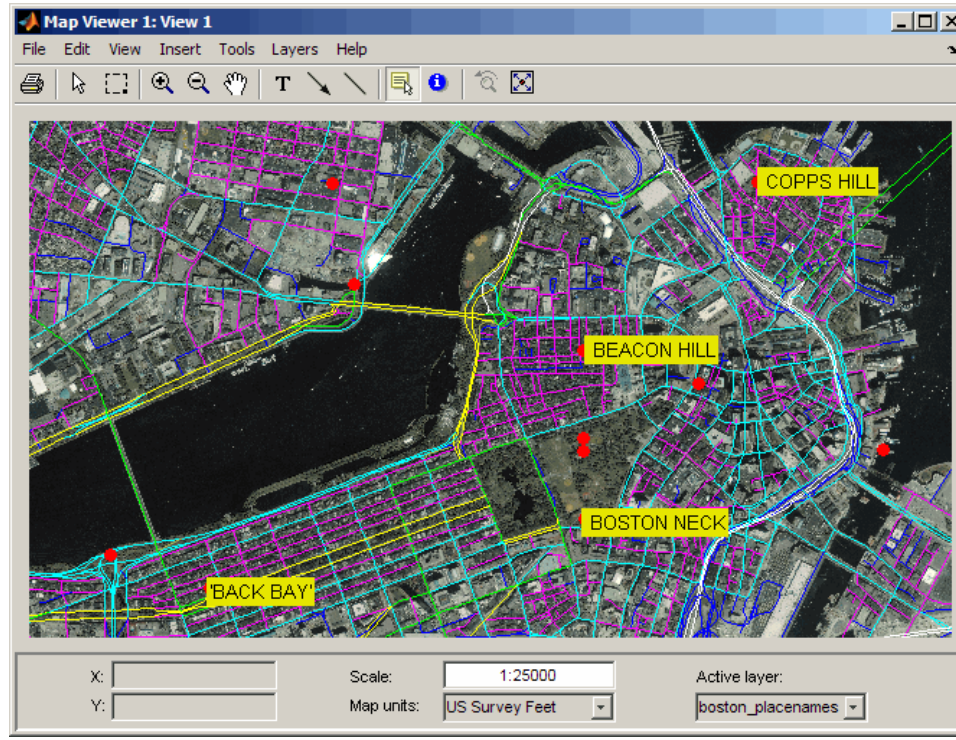
```
places =

          ShapeType: 'Point'
             Marker: {'Default'  ''  'o'}
    MarkerEdgeColor: {'Default'  ''  'r'}
    MarkerFaceColor: {'Default'  ''  '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.

23 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**.

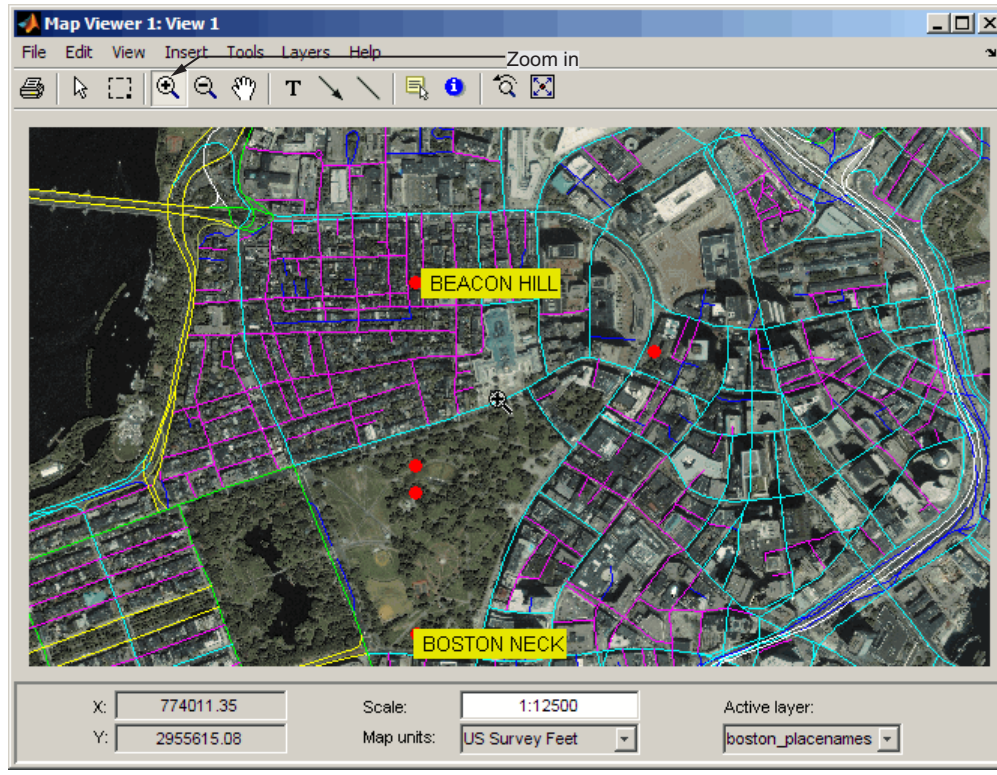
The Map Viewer reads the workspace variable `places`; the cross marks turn into red circles. Note that a layer need not be active in order for you to apply a symbolspec to it.

24 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 datatips you show).



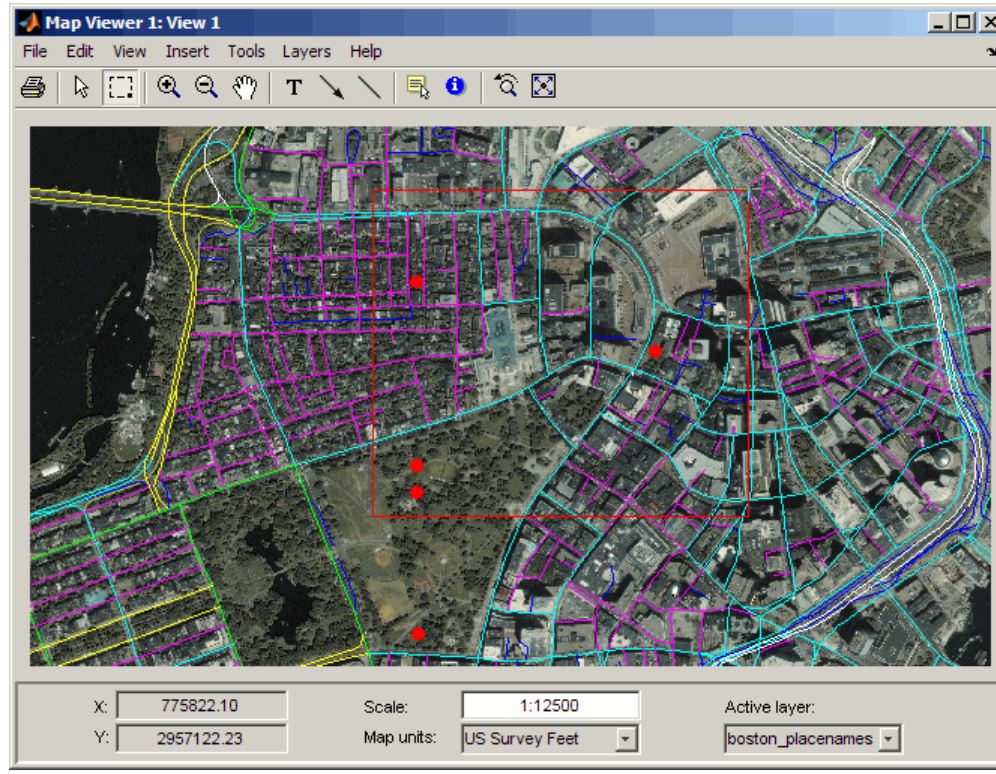
25 Zoom in on Beacon Hill for a closer view of the Massachusetts State House and Boston Common. Select the **Zoom in** tool; move the (magnifier) cursor until the **X** readout is approximately `774,011` and the **Y** readout is roughly

2,955,615; and click once to enlarge the view. The scale changes to about 1:12,500 and the map appears as below.



26 From the **Tools** menu, choose **Select Annotations** to change from the **Datatip** tool back to the original cursor. 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 map.

27 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.



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

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

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 along with the TIF.

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

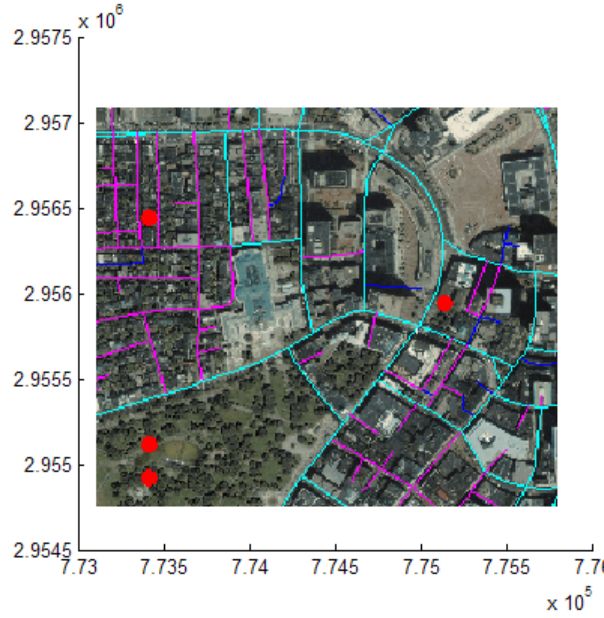
- An image file (*`file`*`.tif`, *`file`*`.png`, or *`file`*`.jpg`)

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

The following steps show you how to read worldfiles and display a georeferenced image outside of mapview.

30 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 the map with mapshow:

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



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

31 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. For more information about the Map Viewer, see the `mapview` reference page.

32 The two examples in the Getting Started chapter provide an introduction to the Mapping Toolbox. To find out what else the toolbox can do, run the demos described in "Mapping Toolbox Demos and Data" on page 1-27. If you have a specific task in mind and want to see a similar problem solved, search the index of examples.

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 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`.

Help is available for individual commands and classes of Mapping Toolbox commands:

- `help` *`functionname`* for help on a specific function, often including examples
- `doc` *`functionname`* to read a function's reference page in the Help browser, including examples and illustrations
- `help map` for a list of functions by category
- `mapdemos` for a list of Mapping Toolbox demos
- `maps` to see a list of all Mapping Toolbox map projections by class, name, and ID string
- `maplist` to return a structure describing all Mapping Toolbox map projections
- `projlist` to list map projections supported by `projfwd` and `projinv`

Consulting Release Notes

To learn how one version of Mapping Toolbox software differs from the next, 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

Try out some demos to see Mapping Toolbox functions in action. Two of the demos are Web demos, and you cannot view their code. To see the code for any of the other demos, open the demo and click **Open <demo>.m in the Editor**. This link is located on the left side of the banner at the top of the page.

Creating Map Displays

- Importing Geographic Data and Creating Map Displays (Web)
- Creating Maps from Geographic (Latitude, Longitude) Data — mapexgeo
- Creating Maps from Data in a Projected *x-y* System — mapexmap
- Creating an Interactive Map for Selecting Point Features — mapexfindcity

Using Geospatial Analysis Tools and Formats

- Converting Coastline Data (GSHHS) to Shapefile Format — mapexgshhs
- Exporting Vector Point Data to KML — mapexkmlexport
- Plotting a 3-D Dome as a Mesh Over a Globe — mapex3ddome
- Un-Projecting a Digital Elevation Model (DEM) — mapexunprojectdem

Working with Georeferenced Images

- Creating a Half-Resolution Georeferenced Image — mapexrefmat
- Georeferencing an Image to an Orthotile Base Layer — mapexreg

Interacting with Web Map Service (WMS) Servers

- Overlaying Web Map Service (WMS) Weather and Satellite Imagery Layers to Calculate Storm Area (Web)
- Compositing and Animating Web Map Service (WMS) Meteorological Layers — mapexwmsanimate

To see this list of links, as well as descriptions of the sample data provided in Mapping Toolbox, type the following:

```
help mapdemos
```

Locating Geospatial Data

Find sample data sets in the Mapping Toolbox `mapdemos` directory (`toolbox\map\mapdemos`). Most of the sample data sets have `.txt` files that provide information or metadata about 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`

2

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 2-21.

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 2-49), 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 2-38.

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', [0 360]);
```

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

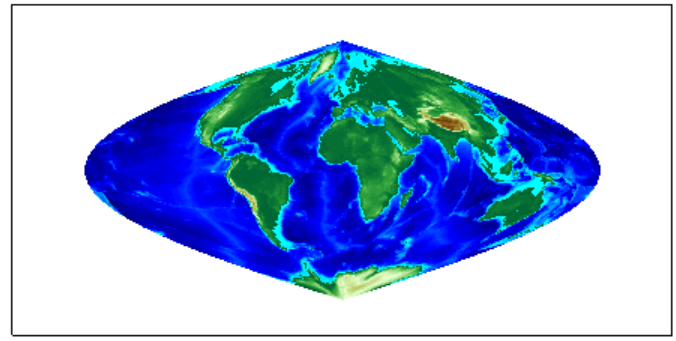
```
axesm sinusoid
```

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)
```

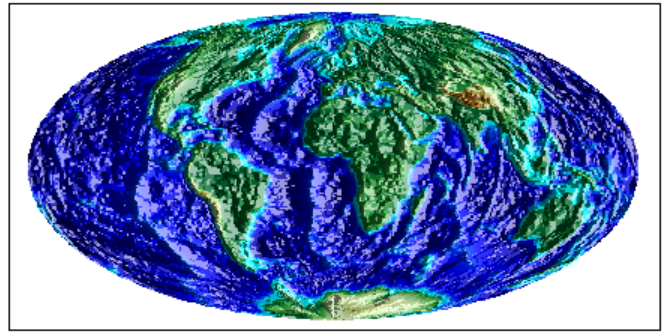
The `geoshow` function displays geodata in geographic (unprojected) coordinates. The `geoshow` output is shown below:



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 5-21.

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
```

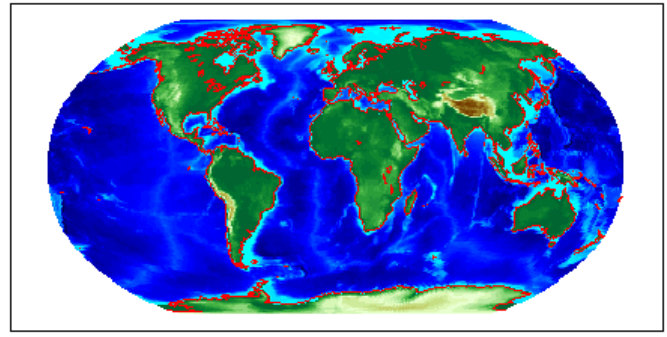
5 Plot the raster topographic data with an appropriate colormap:

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

6 Plot the coastline data in red 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 2-38.

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 2-57

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, and Polygons

In the context of geodata, *vector data* means points, lines, and polygons that represent geographic objects. Vector geospatial data is used to represent point features, such as cities and landmarks; linear features, such as rivers and highways; and areal features, such as bodies of water and voting districts.

Displaying a Point

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, a single point, the location of the Eiffel Tower can be mapped as a vector.

1 Display a map of France.

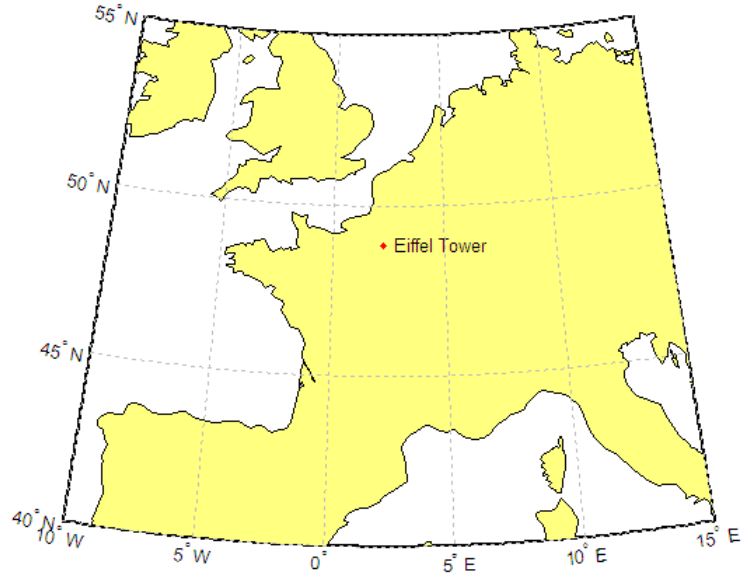
```
h = worldmap('France');
landareas = shaperead('landareas.shp','UseGeoCoords', true);
geoshow (landareas, 'FaceColor', [1 1 .5]);
```

2 Save the location of the Eiffel Tower in vector form.

```
TowerLat = 48.85;
TowerLon = 2.28;
```

3 Place a red dot on the map to indicate the tower and label it.

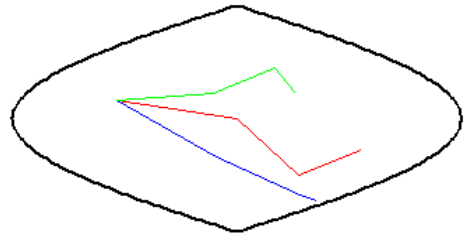
```
geoshow(TowerLat, TowerLon, 'Marker','.','MarkerEdgeColor','red')
textm(TowerLat,TowerLon + 0.5, 'Eiffel Tower');
```



Displaying a Line

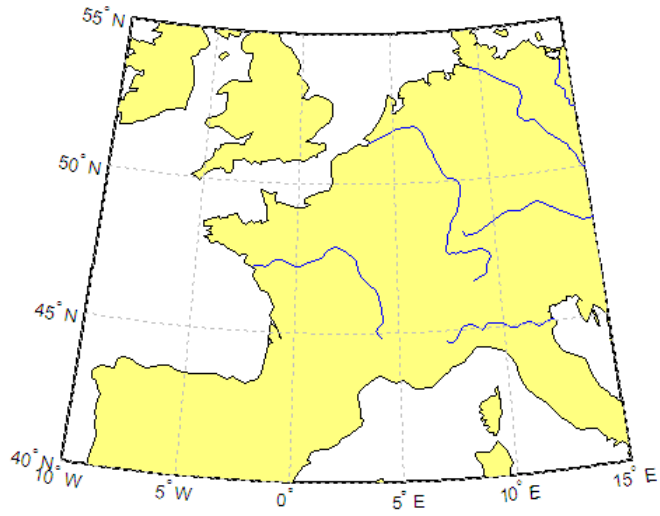
This simple example demonstrates how to use the function `linem` to display vector data for three short lines branching from one common endpoint.

```
axesm sinusoid; framem;
linem([15; 0; -45; -25],[-100; 0; 70; 110],'r-')
linem([15; -30; -60; -65],[-100; -20; 100; 150],'b-')
linem([15; 20; 40; 20],[-100; -20; 40; 50], 'g-')
```



Rivers are examples of lines. Enter the following code to add rivers to a map of France:

```
h = worldmap('France');
landareas = shaperead('landareas.shp','UseGeoCoords', true);
geoshow (landareas, 'FaceColor', [1 1 .5]);
rivers = shaperead('worldrivers', 'UseGeoCoords', true);
geoshow(rivers, 'Color', 'blue')
```



Find out more about the `rivers` structure array:

```
rivers

rivers =

128x1 struct array with fields:
    Geometry
    BoundingBox
    Lon
    Lat
    Name
```

`rivers` contains 128 world rivers. Type the following at the command line to view the structure for the eighth river:

```
rivers(8)

ans =

       Geometry: 'Line'
    BoundingBox: [2x2 double]
            Lon: [129.6929 128.9659 128.7473 NaN]
            Lat: [63.3965 63.4980 63.5220 NaN]
           Name: 'Lena'
```

The rivers are stored as shapes of type `'Line'`. Data for the eighth river, Lena, is stored in `Lat` and `Lon` vectors. Note that each vector ends with a NaN.

Displaying a Polygon

Many common map objects, such as state boundaries, islands, and continents, are polygons. Some polygon objects in the real world 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. A NaN separator serves as a "pen-up" command, a command to stop drawing one polygon and start drawing another. The example below demonstrates how NaN separators divide the data for simple polygons.

1 Copy and paste the following vector variables at the command line:

```
x = [40  55  33  10  0  5  10  40  NaN  10  25  30  25  10...
     10  NaN  90  80  65  80  90  NaN];
y = [50  20  0  0  15  25  55  50  NaN  20  10  10  20  30...
     20  NaN  10  0  20  25  10  NaN];
```

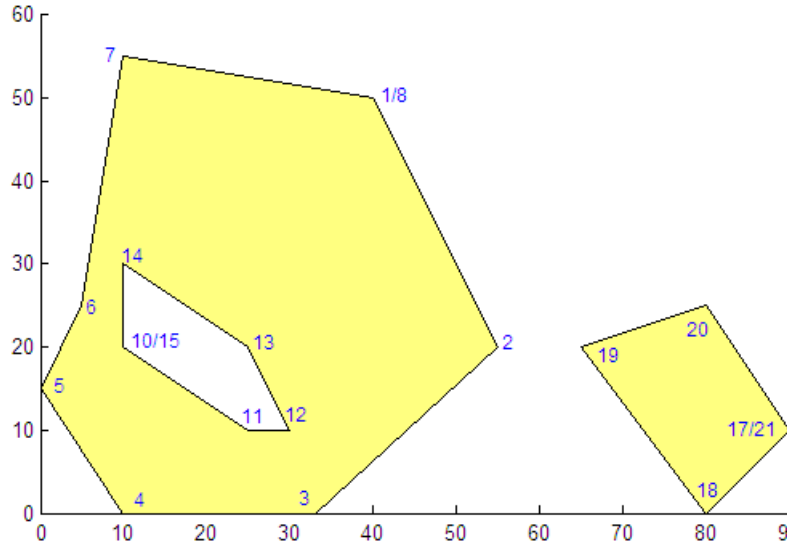
These vector variables appear as rows in the following table, along with a row listing the indices.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
X	40	55	33	10	0	5	10	40	NaN	10	25	30	25	10	10	NaN	90	80	65	80	90	NaN
Y	50	20	0	0	15	25	55	50	NaN	20	10	10	20	30	20	NaN	10	0	20	25	10	NaN

Notice that the NaNs appear in the same locations in both *x* and *y* vectors. Columns 9, 16, and 22 of the table have NaNs. These mark the division between separate polygons. Also, notice that the *x* and *y* data for vertices 1 and 8 are the same. This is the point where segments join together to close the polygon.

2 Use the `mapshow` function to display the polygon.

```
mapshow(x,y,'DisplayType','polygon')
```

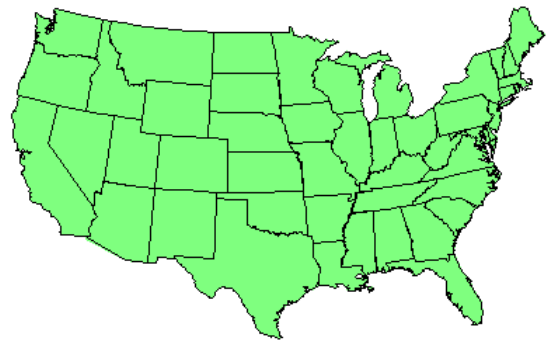


3 In this example, the vector variables contain data that displays as a compact, multipart polygon with a hole. Compare the data from the table to the illustration. Note that the vertices in the illustration have been labelled to correspond to the indices. (Your output will not contain these labels.)

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. You can see that the "hole" has vertices that appear in counter-clockwise order.

Now consider an example of polygons on a map of the United States. Enter the following code to display a map of the U.S. (excluding Alaska and Hawaii):

```
figure; ax = usamap('conus');
set(ax, 'Visible', 'off')
states = shaperead('usastatelo', 'UseGeoCoords', true);
names = {states.Name};
indexConus = 1:numel(states);
stateColor = [0.5 1 0.5];
geoshow(ax, states(indexConus),  'FaceColor', stateColor)
setm(ax, 'Frame', 'off', 'Grid', 'off',...
     'ParallelLabel', 'off', 'MeridianLabel', 'off')
```



Examine the structure for one of the states:

```
states(4)

ans =

       Geometry: 'Polygon'
    BoundingBox: [2x2 double]
            Lon: [1x183 double]
            Lat: [1x183 double]
           Name: 'Arkansas'
       LabelLat: 34.8350
       LabelLon: -91.8861
```

You can see that Arkansas is of shape type `'Polygon'`. View the last entry in the `Lat` vector:

```
states(4).Lat(end)

ans =

   NaN
```

The NaN serves as a separator between polygons.

Compare the first and next-to-last entries:

```
states(4).Lat(1)
states(4).Lat(182)

ans =

   33.0200


ans =

   33.0200
```

The first and next-to-last entries are the same to close the polygon.

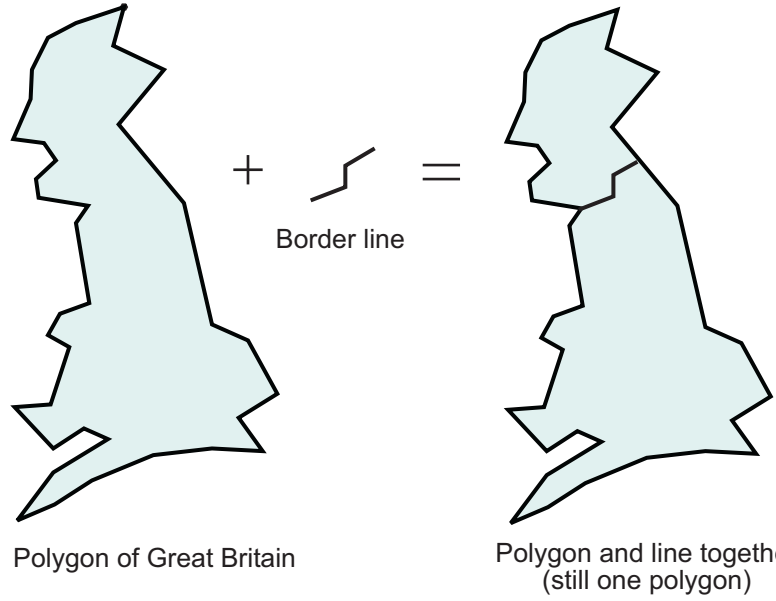
Segments Versus Polygons

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 of vector map features organized in *geographic data structures*.

A geographic data structure is a MATLAB structure array that has one element per geographic feature. Each feature is represented by coordinates and attributes. A geographic data structure that holds geographic coordinates (latitude and longitude) is called a *geostruct*, and one that holds map coordinates (projected *x* and *y*) is called a *mapstruct*. Geographic data structures hold only vector features and cannot be used to hold raster data (regular or geolocated data grids or images).

Shapefiles

Geographic data structures most frequently originate when vector geodata is imported from a shapefile. The Environmental Systems Research Institute designed the shapefile format for vector geodata. Shapefiles encode coordinates for points, multipoints, lines, or polygons, along with non-geometrical attributes.

A shapefile stores attributes and coordinates in separate files; it consists of a main file, an index file, and an xBASE file. All three files have the same base name and are distinguished by the extensions `.shp`, `.shx`, and `.dbf`, respectively. (For example, given the base name `'concord_roads'` the shapefile filenames would be `'concord_roads.shp'`, `'concord_roads.shx'`, and `'concord_roads.dbf'`).

The Contents of Geographic Data Structures

The `shaperead` function reads vector features and attributes from a shapefile and returns a geographic data structure array. The `shaperead` function determines the names of the attribute fields at run-time from the shapefile xBASE table or from optional, user-specified parameters. If a shapefile attribute name cannot be directly used as a field name, `shaperead` assigns the field an appropriately modified name, usually by substituting underscores for spaces.

Fields in a Geographic Data Structure

Field Name	Data Type	Description	Comments
`Geometry`	String	One of the following shape types: `'Point'`, `'MultiPoint'`, `'Line'`, or `'Polygon'`.	For a `'PolyLine'`, the value of the *`Geometry`* field is simply `'Line'`.
`BoundingBox`	2-by-2 numerical array	Specifies the minimum and maximum feature coordinate values in each dimension in the following form: $\begin{bmatrix} \min(X) & \min(Y) \\ \max(X) & \max(Y) \end{bmatrix}$	Omitted for shape type `'Point'`.
`X`, `Y`, `Lon`, or `Lat`	1-by-N array of class `double`	Coordinate vector.	
`Attr`	String or scalar number	Attribute name, type, and value.	Optional. There are usually multiple attributes.

The `shaperead` function does *not* support any 3-D or "measured" shape types: `'PointZ'`, `'PointM'`, `'MultipointZ'`, `'MultipointM'`, `'PolyLineZ'`, `'PolyLineM'`, `'PolygonZ'`, `'PolylineM'`, or `'Multipatch'`. Also, although `'Null Shape'` features can be present in a `'Point'`, `'Multipoint'`, `'PolyLine'`, or `'Polygon'` shapefile, they are ignored.

PolyLine and Polygon Shapes. In geographic data structures with `Line` or `Polygon` geometries, individual features can have multiple parts—disconnected line segments and polygon rings. The parts can include counterclockwise inner rings that outline "holes." For an illustration of this, see "Displaying a Polygon" on page 2-16. 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 multipoint or NaN-separated multipart line or polygon 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 in combination.

Mapstructs and Geostructs. By default, `shaperead` returns a mapstruct containing `X` and `Y` fields. This is appropriate if the data set coordinates are already projected (in a map coordinate system). Otherwise, if the data set coordinates are unprojected (in a geographic coordinate system), use the parameter-value pair `'UseGeoCoords',true` to make `shaperead` return a geostruct having `Lon` and `Lat` fields.

Coordinate Types. If you do not know whether a shapefile uses geographic coordinates or map coordinates, here are some things you can try:

- Ask your data provider.
- Use `shapeinfo` to obtain the `BoundingBox`. By looking at the ranges of coordinates, you may be able to tell what kind of coordinates you have.
- Examine the optional `.prj` file, if one has been provided. The `.prj` file is written in well-known text, a text mark-up language. If your `.prj` file contains the term `PROJCS`, you have map coordinates. If your `.prj` file contains the term `GEOGCS`, but not the term `PROJCS`, you have geographic coordinates.

The `geoshow` function displays geographic features stored in geostructs, and the `mapshow` function displays geographic features stored in mapstructs. If you try to display a mapstruct with `geoshow`, the function issues a warning

and calls mapshow. If you try to display a geostruct with mapshow, the function projects the coordinates with a Plate Carree projection and issues a warning.

Examining a Geographic Data Structure

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

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

The output appears as follows:

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

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.

Look at the 10th element:

```
S(10)
```

The output appears as follows:

```
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
```

This mapstruct contains `'Line'` features. The tenth line has nine vertices. The first two attributes are string-valued. The second happens to be empty. 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.

In this example, `shaperead` returns an unfiltered mapstruct. If you want to filter out some attributes, see "Selecting Data to Read with the shaperead Function" on page 2-32 for more information.

How to Construct Geographic Data Structures

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:

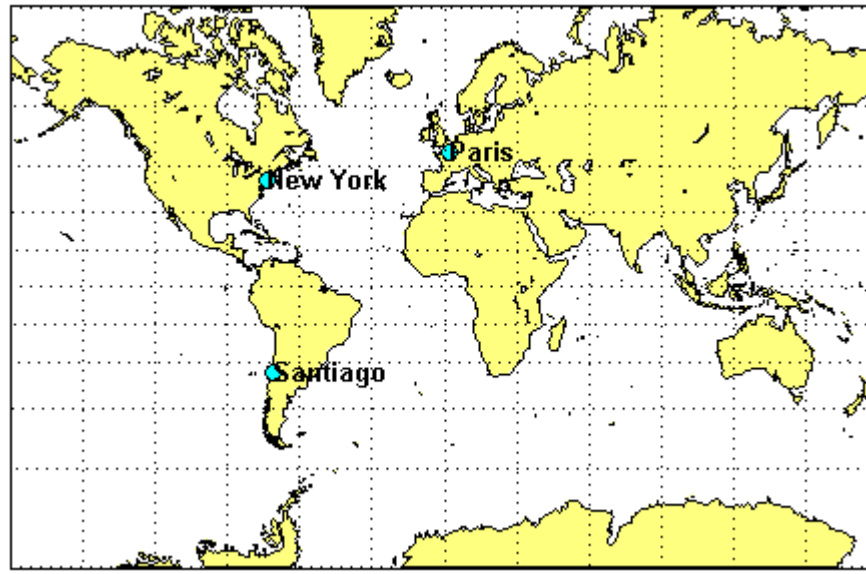
   - `Geometry` (in this case `'Point'`)
   - `Lat` (for points, this is a scalar double)
   - `Lon` (for points, this is a scalar double)

```
% 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);
```

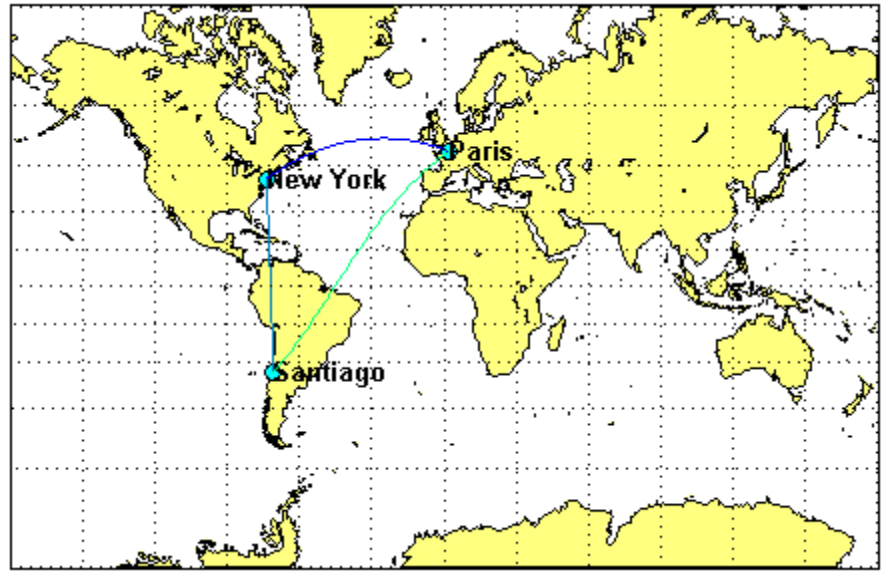
5 Compute lengths of the great circle tracks:

```
% 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" 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 a 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 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 the Mapping Toolbox documentation and function 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 a 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 2-49. 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:

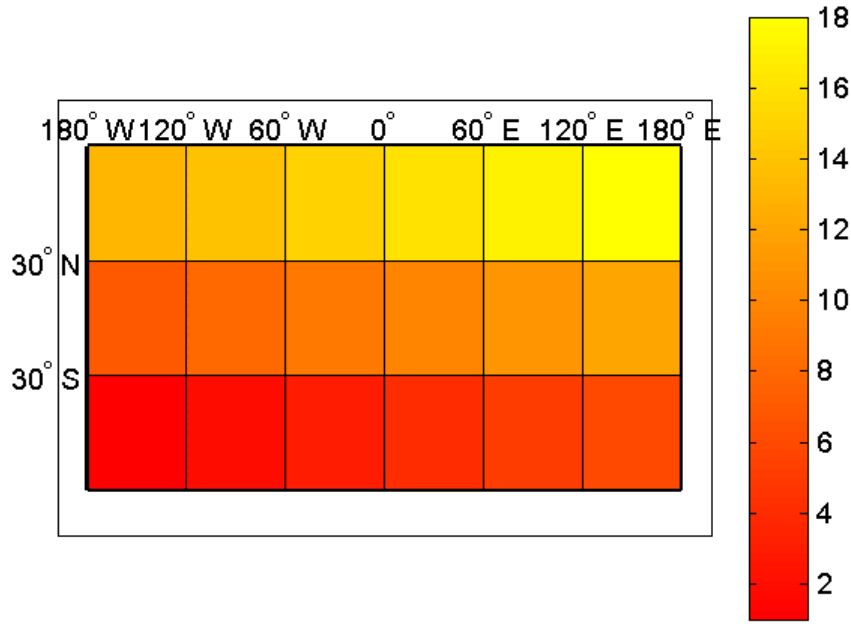
```
axesm('MapProjection', 'eqdcylin')
setm(gca,'GLineStyle','-', 'Grid','on','Frame','on')
```

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:

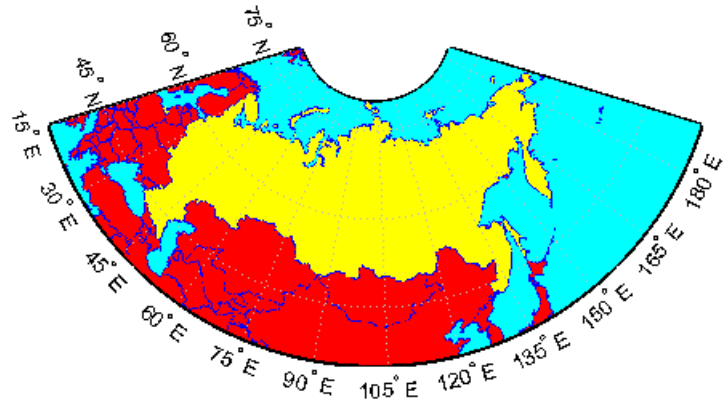
```
clear; load russia
[latlim,longlim] = limitm(map,refvec)
```

The limits are:

```
latlim =
 35    80
longlim =
 15   190
```

Display a map of Russia:

```
worldmap(latlim,longlim);
cmap = jet(4);
geoshow(map,cmap,maplegend)
```



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:

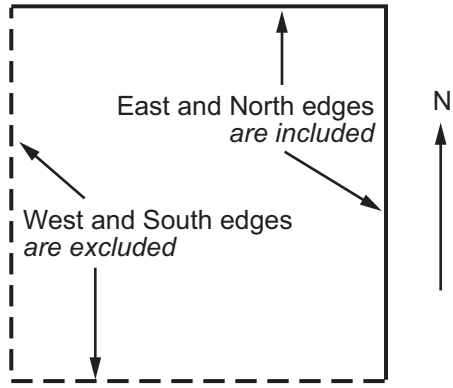
```
row = 23; col = 79;
[lat,long] = setltln(map,refvec,row,col)

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`

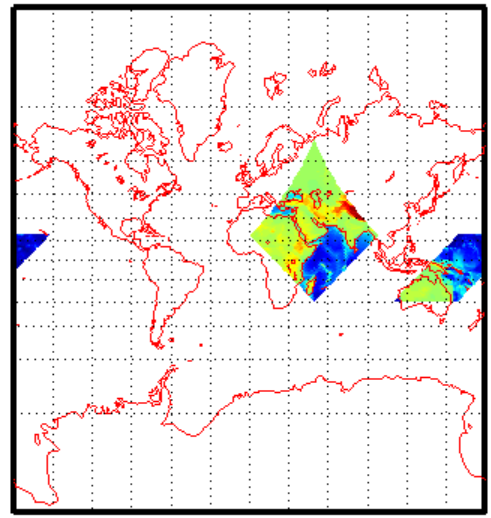
data grid shown in previous examples. Neither of these maps is regular, because their columns do not run north to south.

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')
```

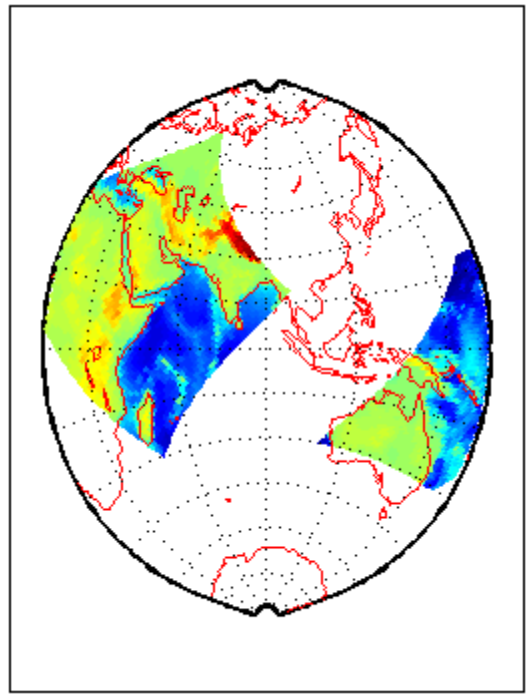


Notice that neither `topo` matrix is a regular rectangle. One looks like a diamond geographically, the other like a trapezoid. The trapezoid is displayed in two pieces because it crosses the edge of the map. These shapes can be thought of as the geographic organization of the data, just as rectangles 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 150º 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:

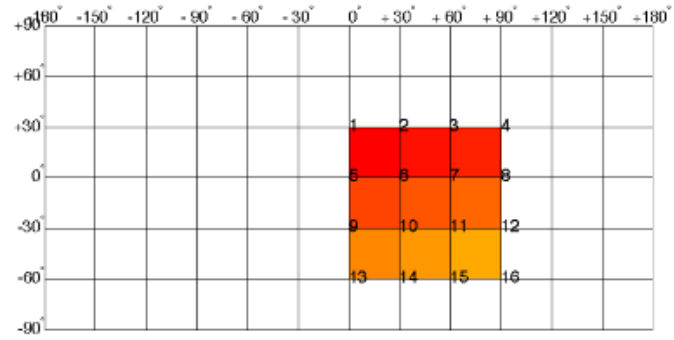
- First, a map matrix having the same number of rows and columns as the latitude and longitude coordinate matrices represents the values of the map data at the corresponding geographic points (centers of data cells).
- Next, a map matrix having one fewer row and one fewer column than the geographic coordinate 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 4-55.

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.

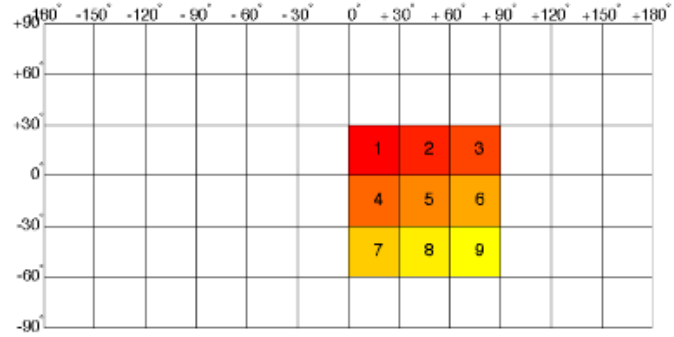


Notice that only 9 of the 16 total cells are displayed. The value displayed for each cell is the value at the upper left corner of that cell, whose coordinates are given by the corresponding `lat` and `long` elements. By convention, the last row and column of the map matrix are not displayed, although 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 ETOPO1c (1-minute), ETOPO2v2c (2-minute), ETOPO2–2001 (2-minute), or ETOPO5 (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

Function	Description	Type of Data	Type of Coordinates
gshhs	Read Global Self-Consistent Hierarchical High-Resolution Shoreline (GSHHS) data	vector	geo
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

Function	Description	Type of Data	Type of Coordinates
`shapewrite`	Write geospatial data and associated attributes in shapefile format	vector	geo map
`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 Geographic Data Structures" on page 2-25 also demonstrates exporting vector data using `shapewrite`. For information about creating KML files, see the following section.

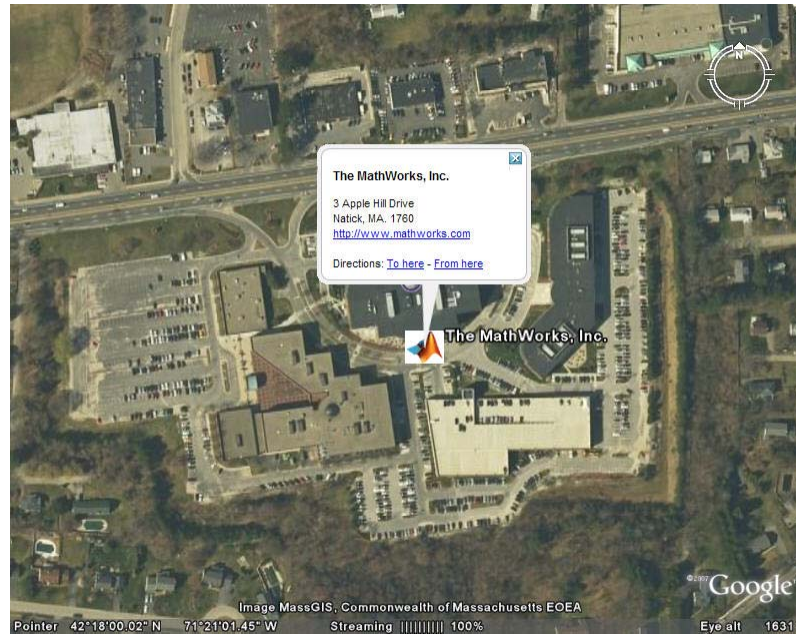
Exporting KML Files for Viewing in Earth Browsers

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 MathWorks, as displayed in the Google Earth application.



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 = '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://www.opengis.net/kml/2.2">
   <Document>
      <name>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://www.opengis.net/kml/2.2">
   <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

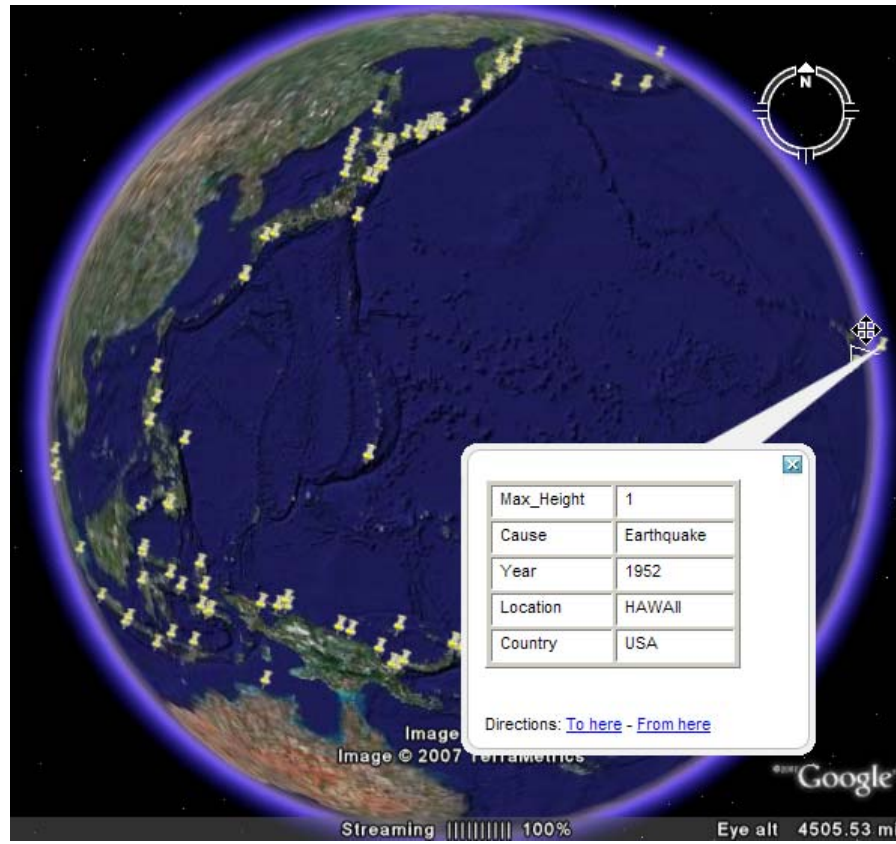
On Windows®, use `winopen` to open Google Earth (which must be installed) to view the KML file.

```
winopen(kmlfilename)
```

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

```
cmd = 'googleearth ';
fullfilename = fullfile(pwd, kmlfilename);
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.



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>';
```

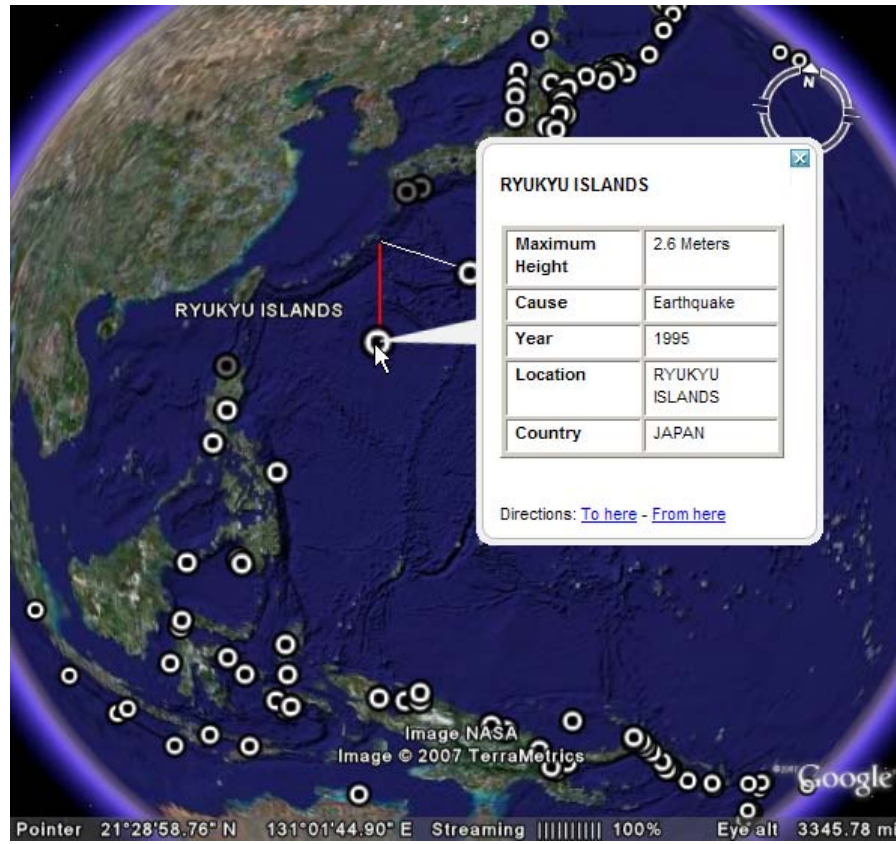
When you use the attribute spec, all the attributes it lists are included in the placemarks generated by `kmlwrite` unless you remove them from the spec manually (e.g., with `rmfield`).

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.



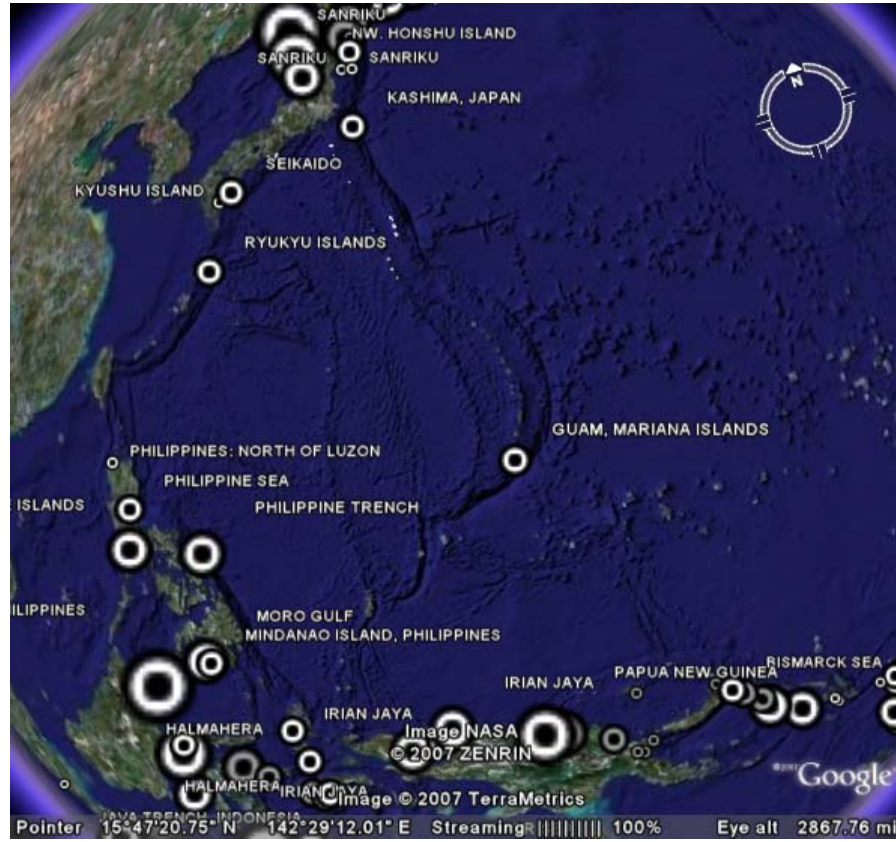
6 Vary placemark size by tsunami height

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.

3

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.

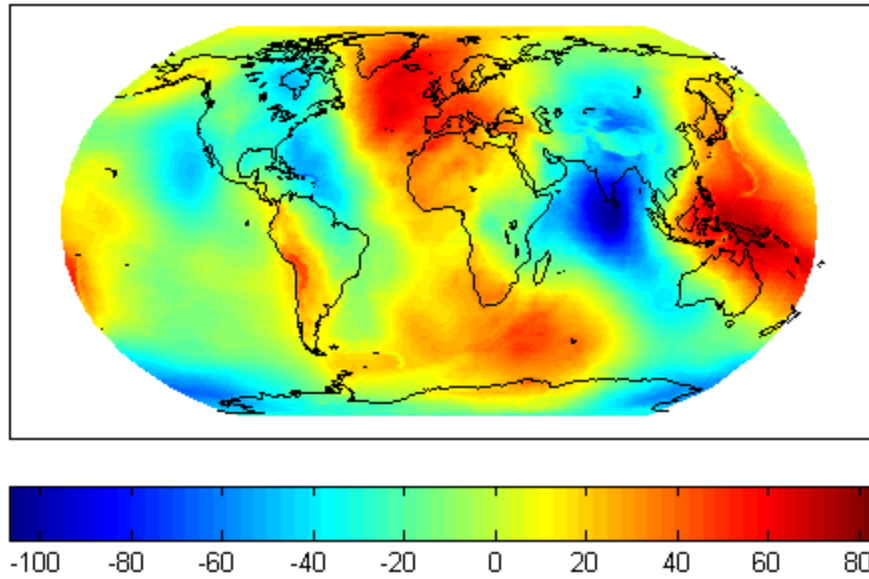
Geoid and Ellipsoid

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

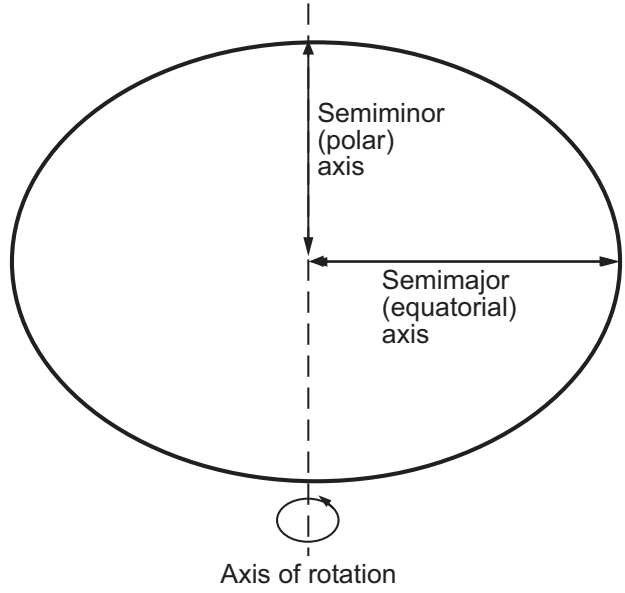
The following figure, made using the `geoid` data set, maps the figure of the Earth. To execute these commands, select them all by dragging over the list in the Help browser, then click the right mouse button and choose `Evaluate Selection`:

```
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 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" 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 `/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:

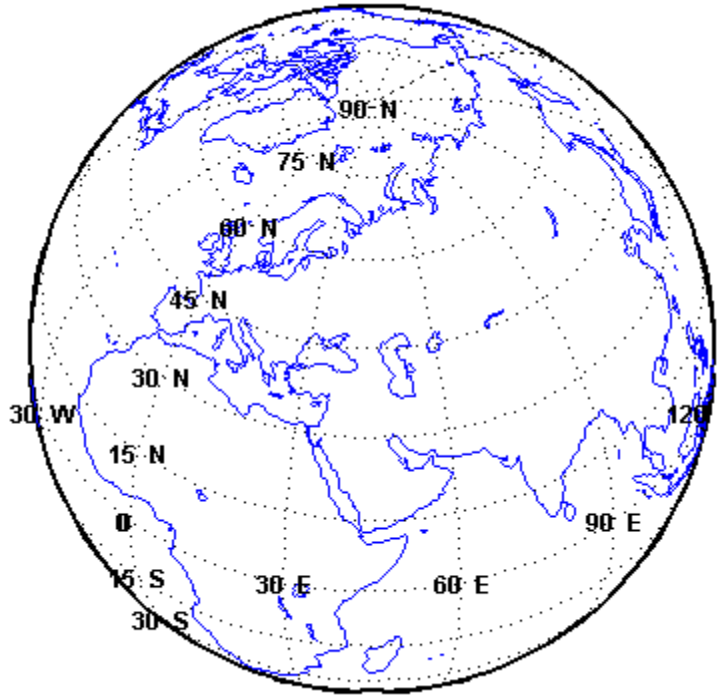
```
ellipsoidvec = [semimajor_axis eccentricity]
```

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* (0º 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 -180º to +180º, but other ranges can be used, such as 0º to +360º. Longitudes can also be specified as east of Greenwich (positive) and west of Greenwich (negative). Adding or subtracting 360º 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 360º-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 =
    10.1600
```

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

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

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 11, "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.

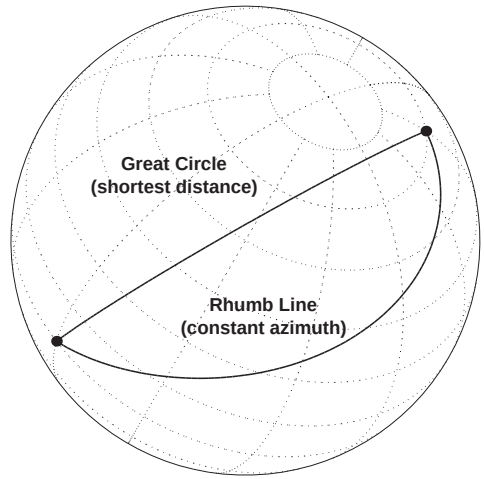
Rhumb Lines

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.

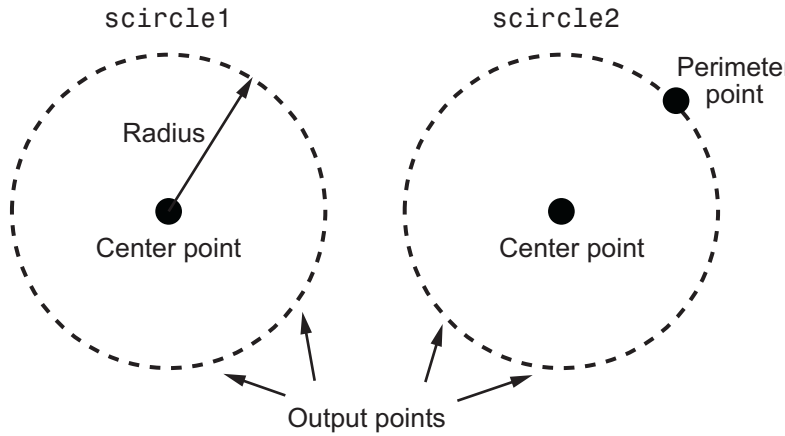
Computing Small Circles

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);
```

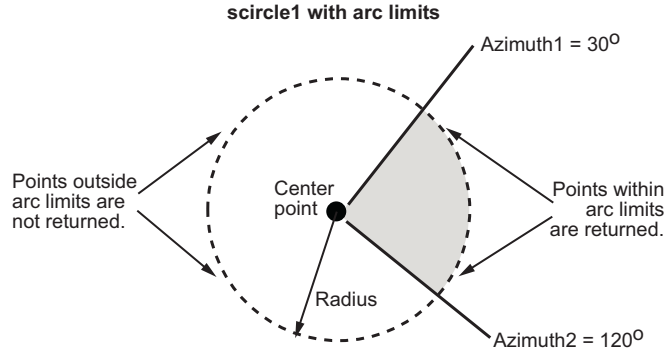
To get the small circle centered at the same point that passes through the point (55ºN,135ºW), use `scircle2`:

```
[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]);
```

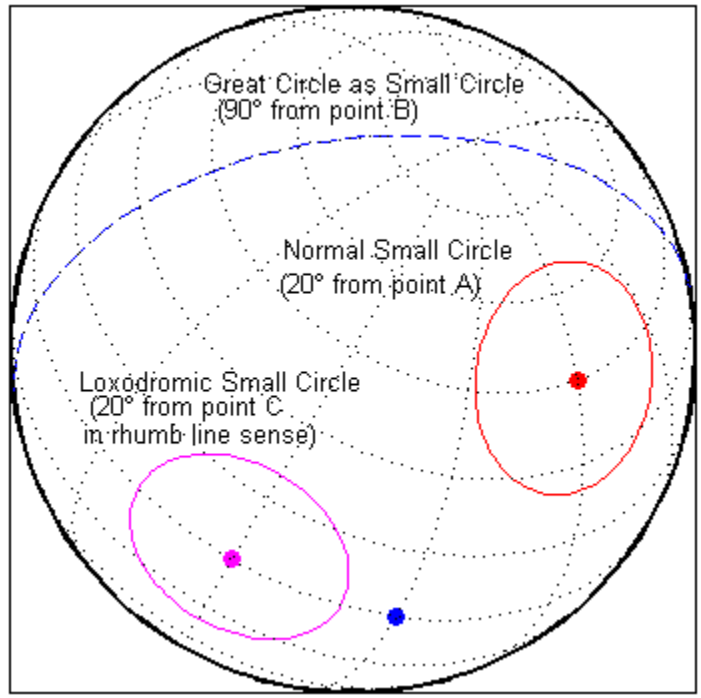


When an entire small circle is calculated, the data is in polygon format. For all calculated small circles, 100 points are returned unless otherwise specified. You can calculate several small circles at once by providing vector inputs. For more information, see the `scircle1` and `scircle2` function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

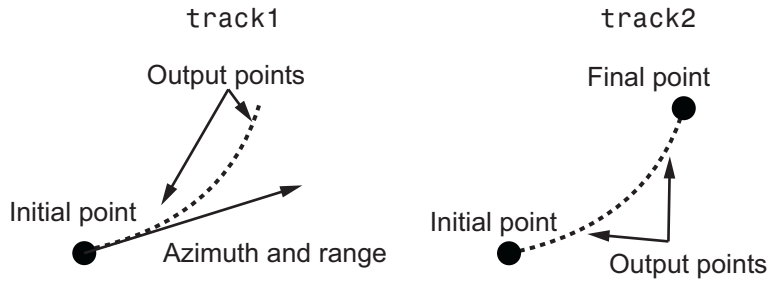
Calculating Tracks — Great Circles and Rhumb Lines

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`:

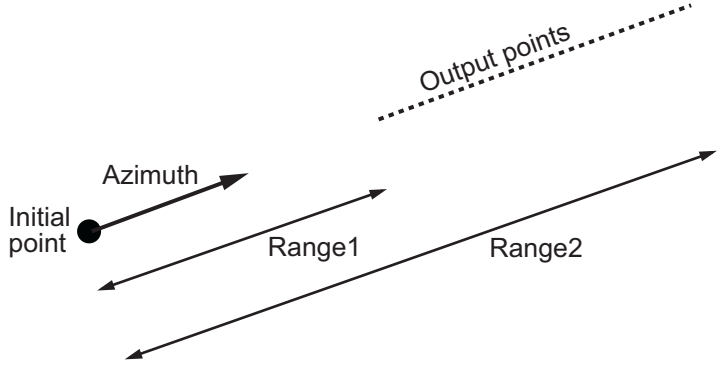
```
[latgc,longc] = track2('gc',-31,90,-23,110);
```



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 180^o 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?

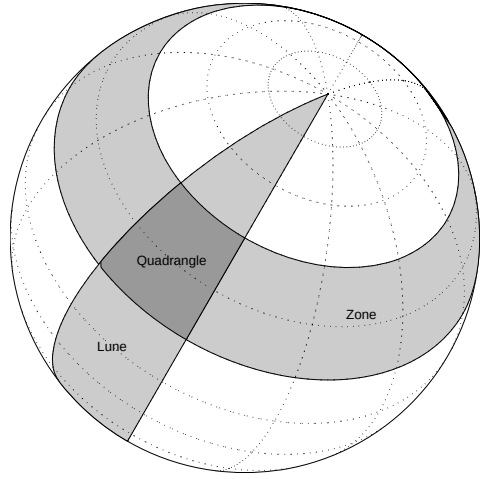
```
[elevang,slantrange,azim] = elevation(0,0,0,0,km2deg(10),10000)

elevang =

   44.9005
```

```
slantrange =

  1.4156e+004


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:

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

ans =

          3443.91846652268        0.0818191910428158

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

ans =

                    3396.9                     0.1105

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

ans =

          1079.94097222222                          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.

4

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.

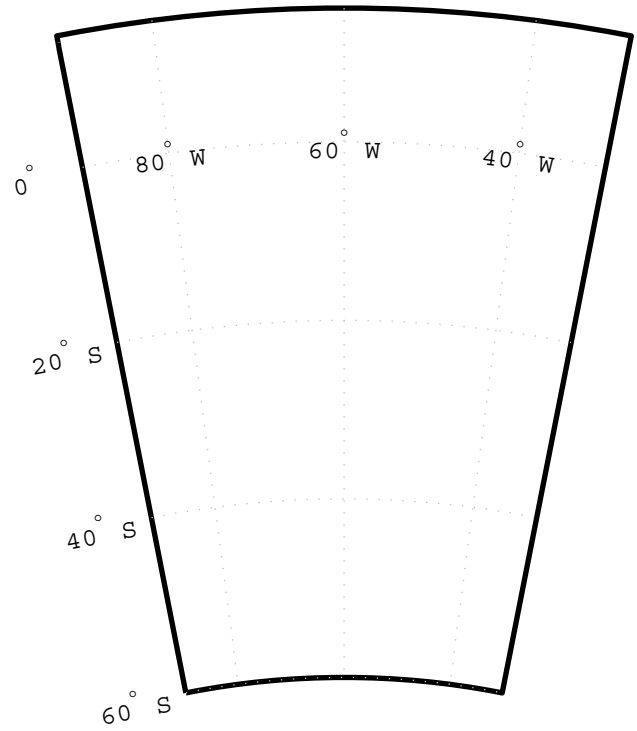
You can also use the Property Editor, part of the MATLAB plotting tools, to modify background colors for figures and axes. See "Plotting Tools — Interactive Plotting" in the MATLAB Graphics documentation for more information.

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:

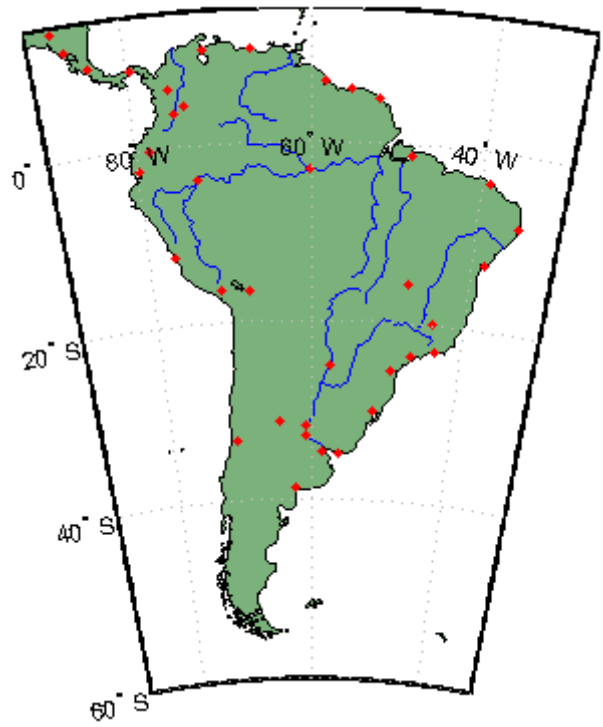
```
getm(gca,'MapProjection')

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

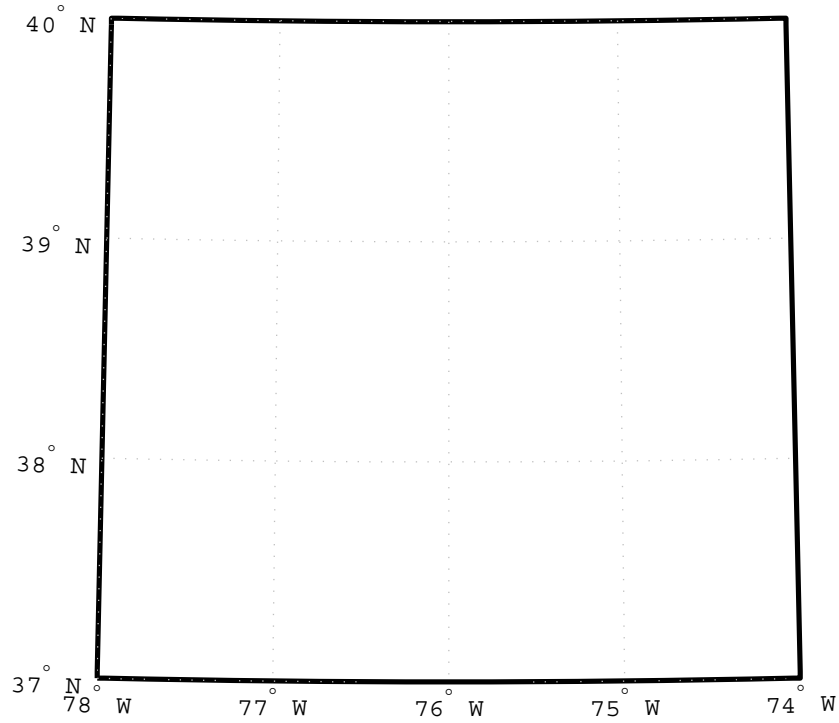
```
usamap
```

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:

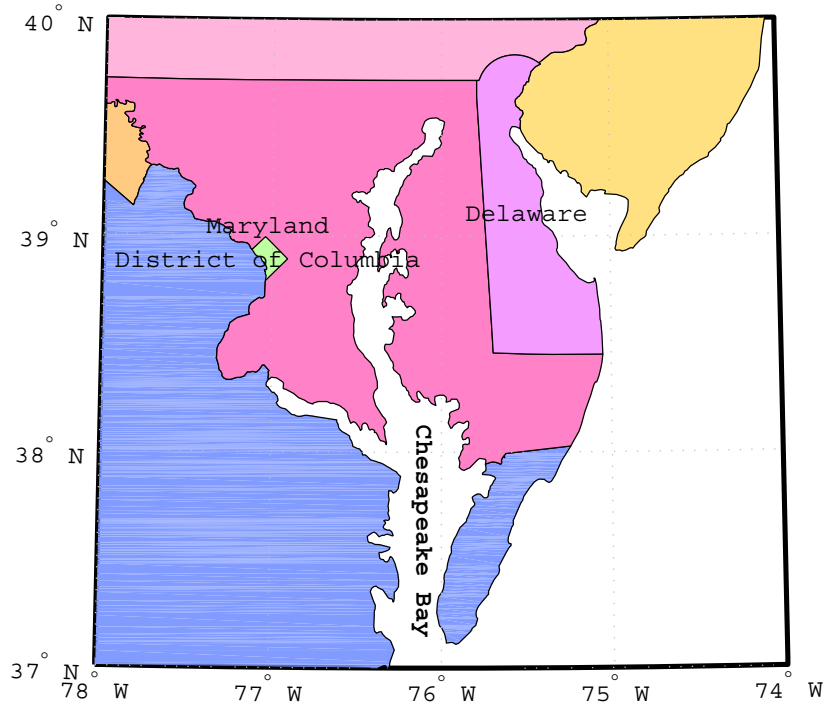
```
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], ...
    'FaceColor', polcmap(numel(states))});
```

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 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`.

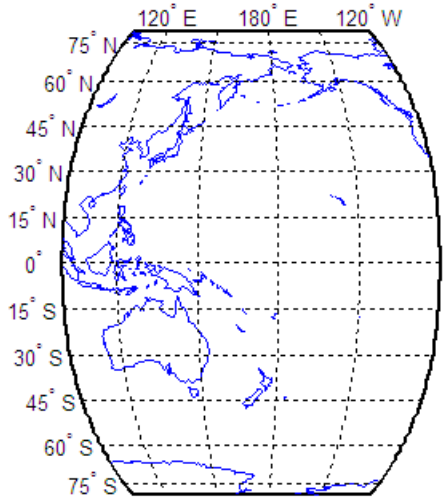
Example 1: Robinson Projection

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
```

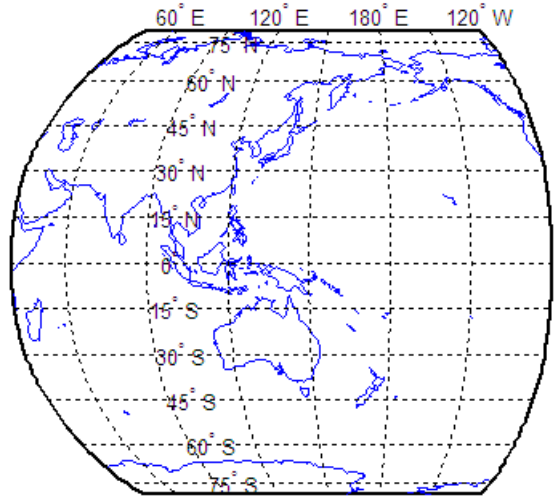
and therefore the latitude limits of the frame equal the value supplied for `MapLatLimit`:

```
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.

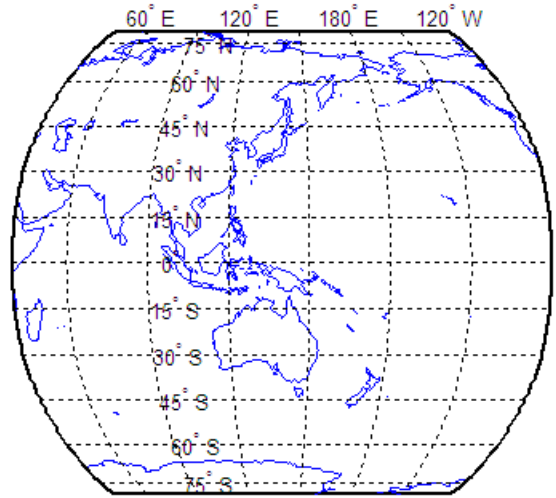
To avoid this asymmetry, you can repeat the operations shown above to figure out that the new central longitude must be at 140 degrees E and add this in the call to `setm` like this:

```
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.

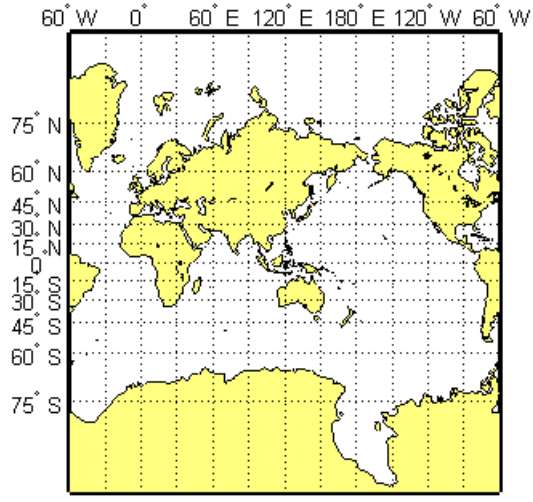
Example 2: Cylindrical Projection

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
```

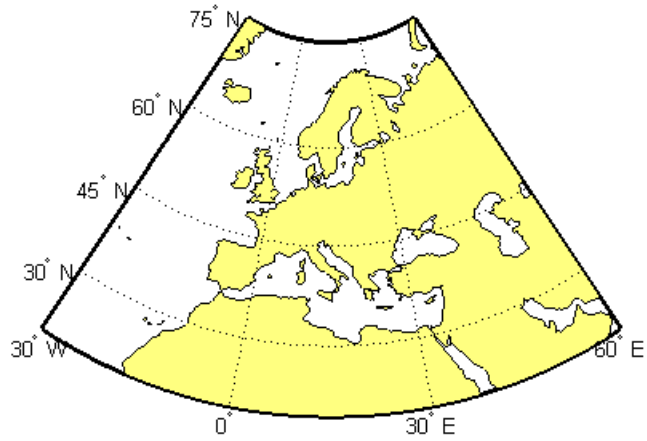
Note that the map and frame limits are clamped to the range of `[-86 86]` imposed by the read-only `TrimLat` property.

```
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')
```



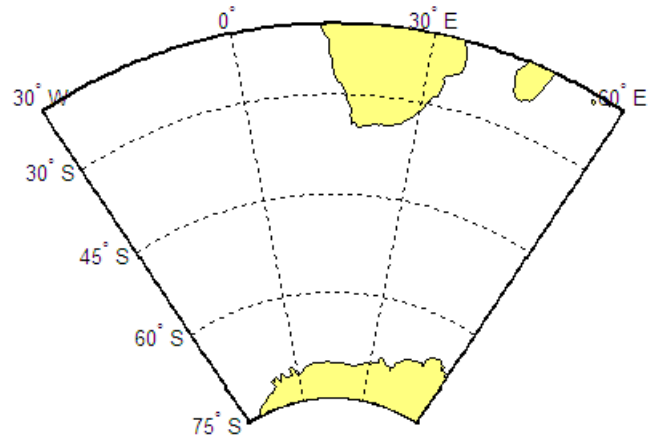
The call to `axesm` above is equivalent to:

```
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`.

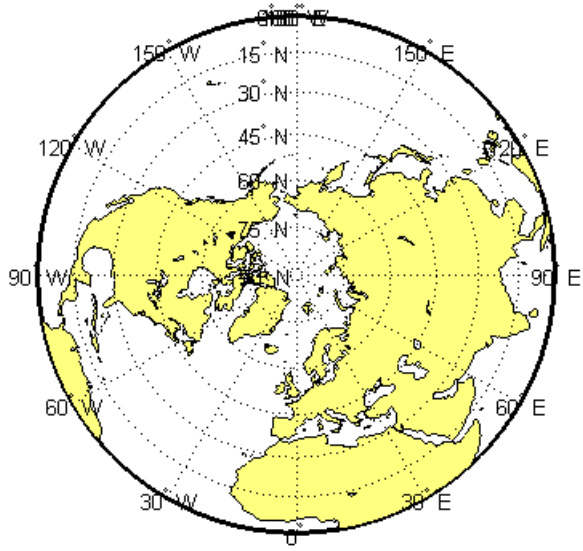
```
coast = load('coast');
figure('Color','w')
axesm('lambertstd','MapParallels',[-75 -15], ...
  'MapLatLimit',[-75 -20],'MapLonLimit',[-30 60])
axis off; framem on; gridm on; mlabel on; plabel on;
geoshow(coast.lat,coast.long,'DisplayType','polygon')
```



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')
```



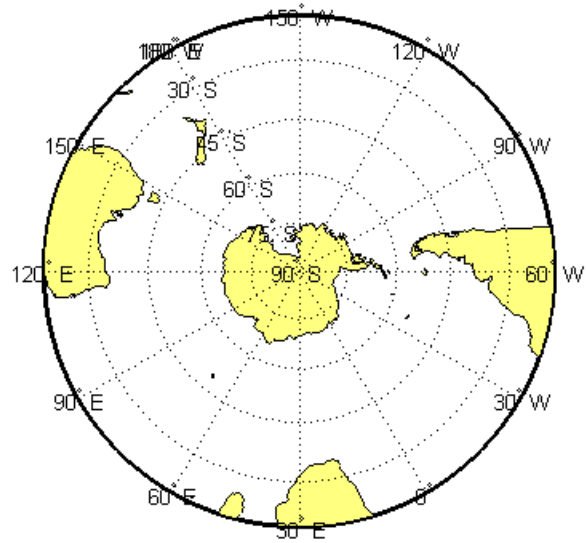
The call to `axesm` above is equivalent to:

```
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')
```



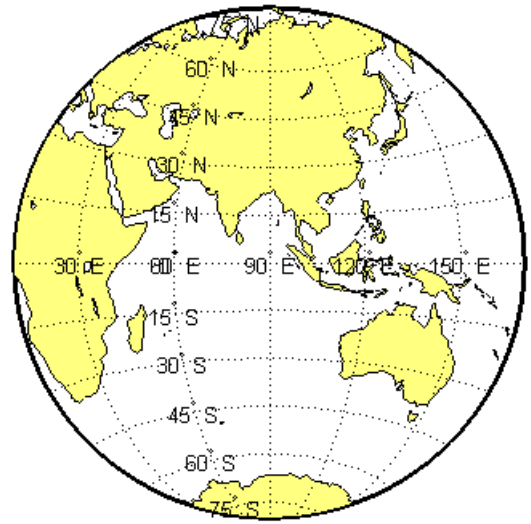
The call to `axesm` above is equivalent to:

```
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','FLatLimit',[],'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')
```



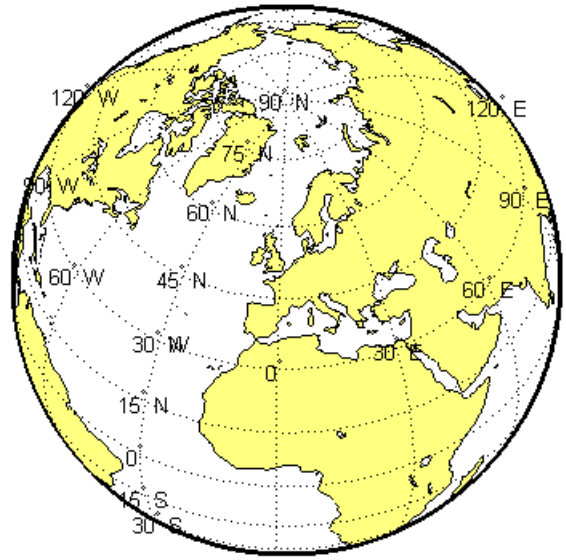
The call to `axesm` above is equivalent to:

```
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')
```

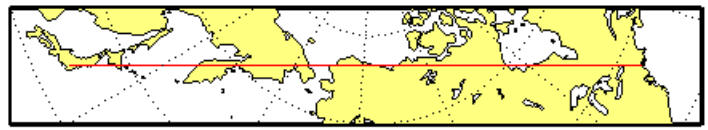


Example 9: Oblique Mercator Projection

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',newMapLonlim)
```

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

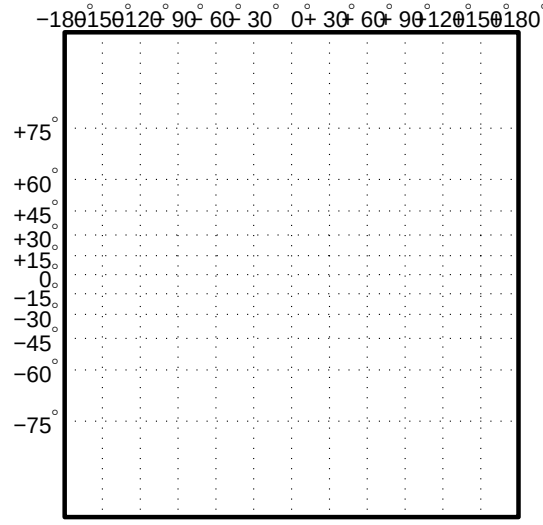
Once a map axes object has been created with `axesm`, whether map data is displayed or not, it is possible to change the current projection as well as many of its parameters. You can use `setm` or the `maptool` UI to reset the projection. The rest of this section describes the considerations and parameters 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 4-42.

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.

Moving 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
```

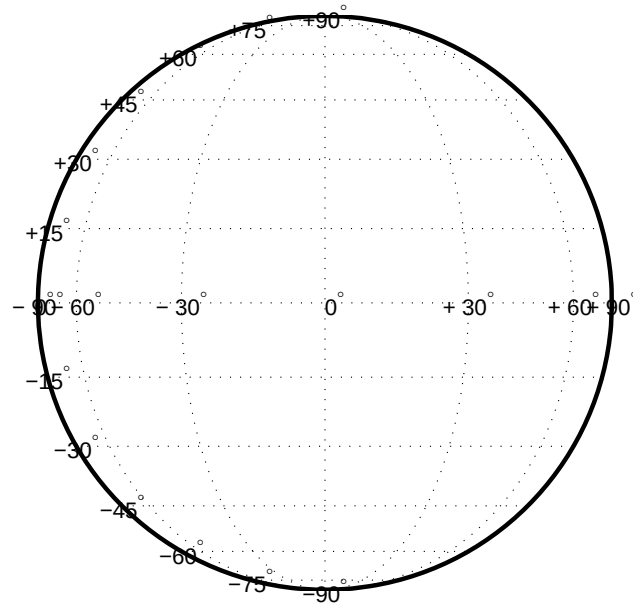
Both the frame and map latitude limits are set to 86º north and south for the Mercator projection to maintain a safe distance from the singularity at the poles.

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 4-58.)

```
setm(gca,'MLabelParallel',0,'PLabelMeridian',-90, ...
   'PLabelMeridian',-30)
```



Resetting Frame Limits

When switching from one projection to another, you may need to reset your origin and frame limits, especially if you are mapping a small portion of the Earth.

1. Construct an empty map axes for a region of the U.S. in the Lambert Conformal Conic projection (the default projection for `usamap`).

   ```
   latlim = [32 42];
   lonlim = [-125 -111];
   h = usamap(latlim, lonlim);
   ```

2. Read in the `'usastatehi'` shapefile and return a subset of the shapefile contents, as defined by the latitude and longitude limits, in a structure called `states`.

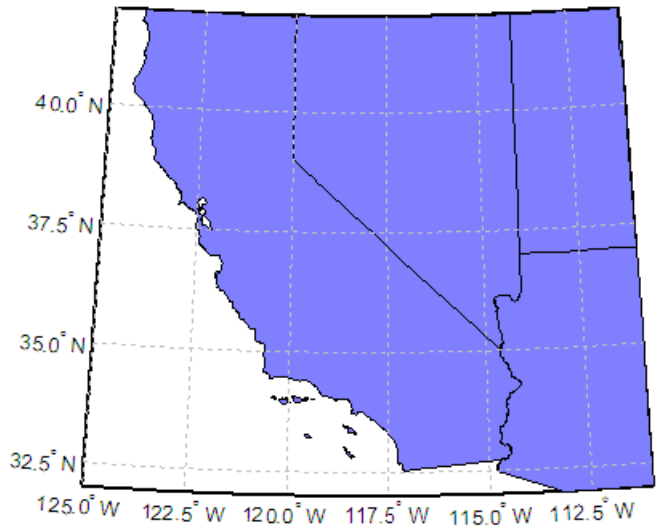
```
states = shaperead('usastatehi', 'UseGeoCoords', true, ...
   'BoundingBox', [lonlim', latlim']);
```

3 Save the latitude and longitude data from the structure in the vectors `lat` and `lon`.

```
lat = [states.Lat];
lon = [states.Lon];
```

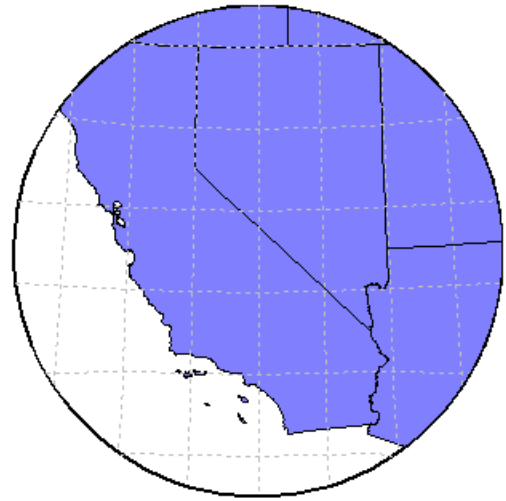
4 Project patch objects on the map axes.

```
patchm(lat, lon, [0.5 0.5 1])
```



5 Change the projection to Lambert Equal Area Azimuthal and reset the origin and frame limits.

```
setm(gca,'MapProjection','eqaazim','Origin',[37 -118], ...
   'FLatLimit',[-Inf 6])
setm(gca,'mlinelocation',2,'plinelocation',2)
tightmap
```



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.

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.

Mapping Use Case	Type of Axes	Reprojection Behavior
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 4-84 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

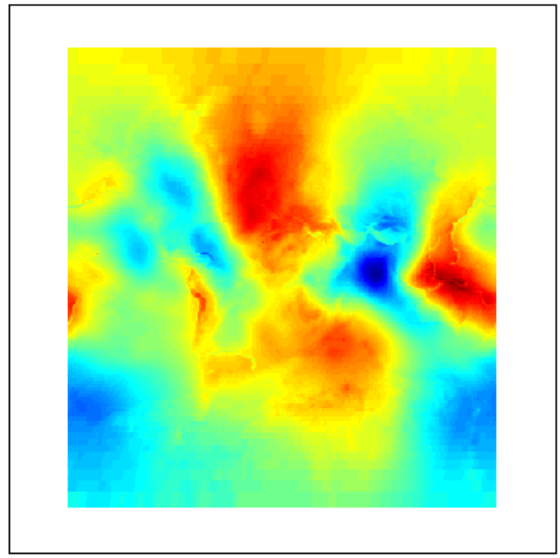
You can display latitude-longitude vector and raster geodata using the `geoshow` function (use `mapshow` to display preprojected coordinates and grids). When you use `geoshow` to display maps on a map axes, the data are projected according to the map projection assigned when `axesm`, `worldmap`, or `usamap` 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*.

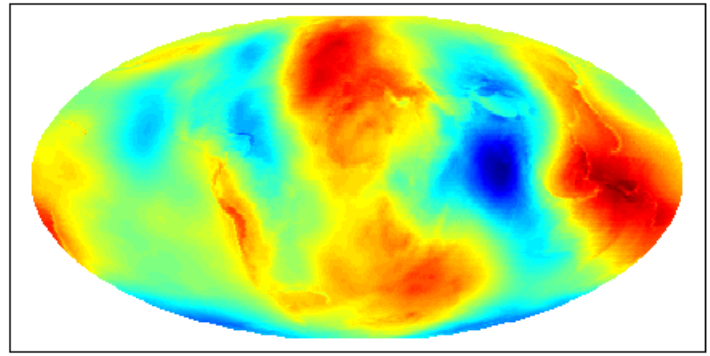
If you are using `geoshow` with a map axes and want to change the map projection after you have displayed data in geographic coordinates, do the following, depending on whether the data are raster or vector:

Raster Data. Change the projection using `setm`. For example,

```
load geoid
figure; axesm mercator
geoshow(geoid,geoidrefvec,'DisplayType','texturemap')
```

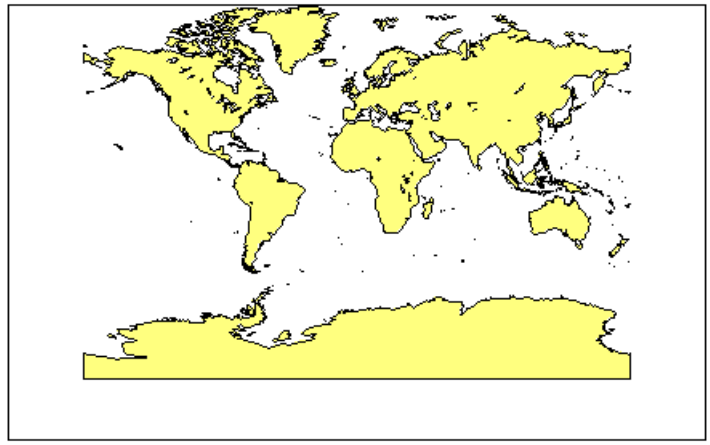


```
setm(gca,'mapprojection','mollweid')
```

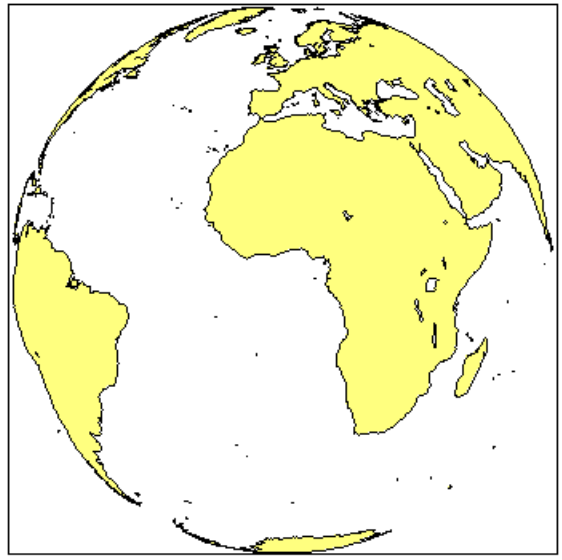


Vector Data. Obtain handles to the line or patch graphic objects, delete the objects from the axes, change the projection using `setm`, and replot the vector data using `geoshow`:

```
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:

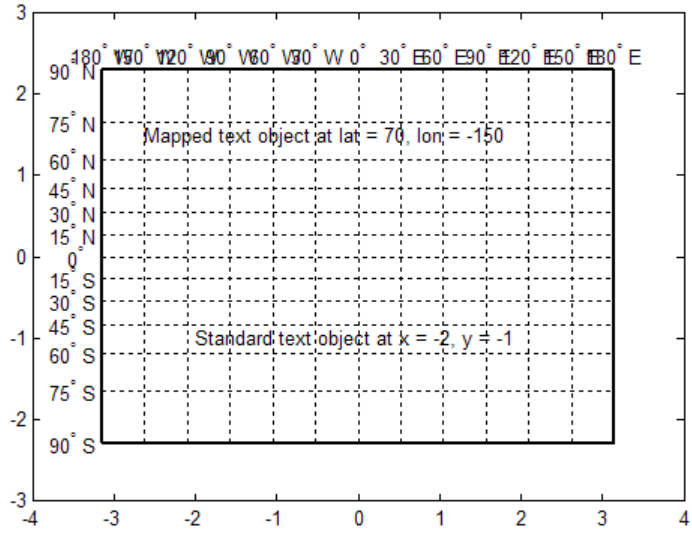
   ```
   axesm miller; framem on; gridm on; mlabel on; plabel on;
   showaxes; grid off;
   ```

   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 4-55.

2. Now place a standard MATLAB text object and a mapped text object, using the two separate coordinate systems:

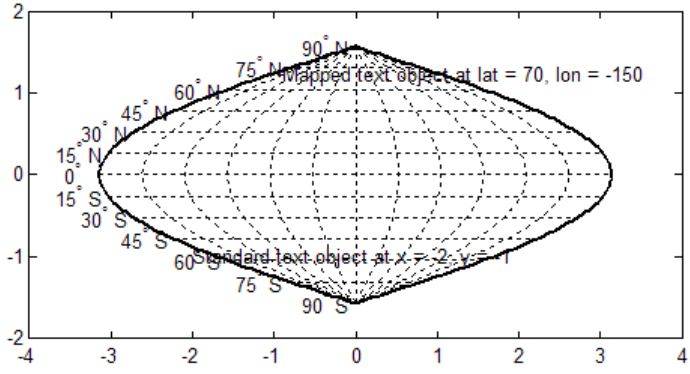
   ```
   text(-2,-1,'Standard text object at x = -2, y = -1')
   textm(70,-150,'Mapped text object at lat = 70, lon = -150')
   ```

   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
```



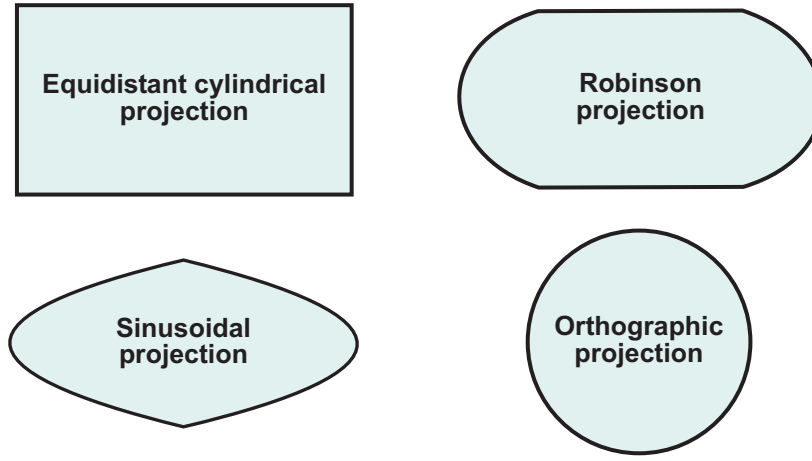
Similarly, vector and matrix data can be displayed using either mapping or standard functions (e.g., `plot`/`plotm`, `surf`/`surfm`). See "Displaying Vector Data with Mapping Toolbox Functions" on page 4-60 for information on plotting vector geodata, and "Displaying Data Grids" on page 4-70 for information 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.

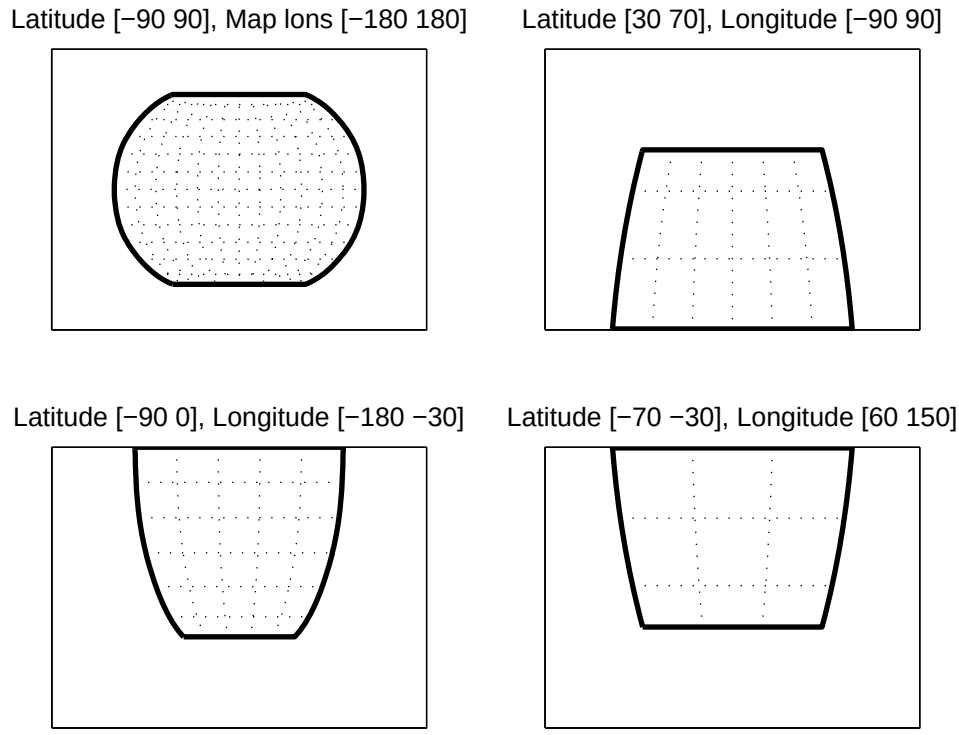


Full-World Map Frames

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)
```



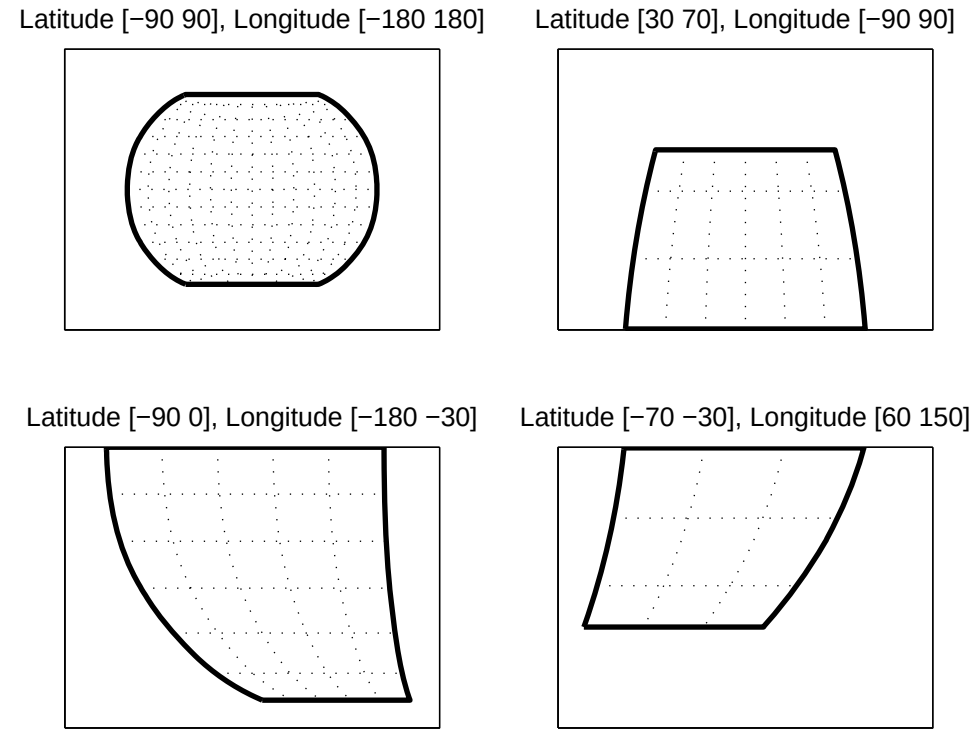
Frame Quadrangles in the Robinson Projection (Symmetric About Prime Meridian)

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]')
```



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.

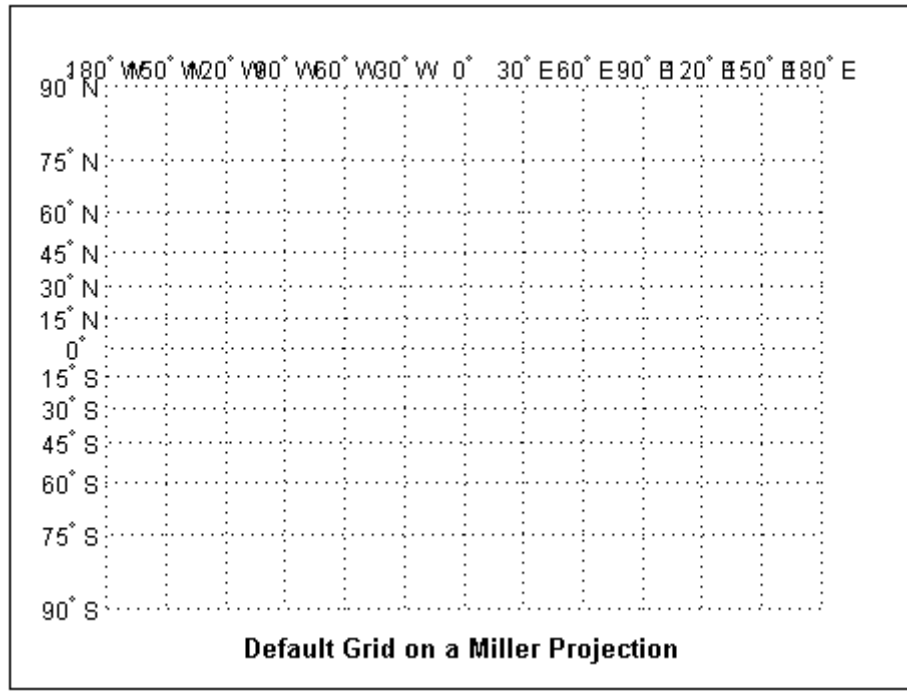
Note For nonazimuthal projections in the normal aspect, the map extent is limited by the minimum of the map limits and the frame limits; hence, the two limits will coincide after evaluation. Therefore, if you manually change one set of limits, you might want to clear the other set to get consistent limits.

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.



Default Grid on a Miller Projection

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.

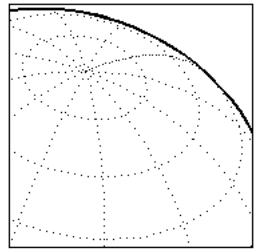
Limiting Grid Lines

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:

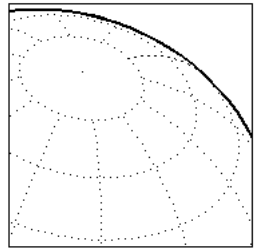
- Use the `MLineLimit` property to specify a pair of latitudes at which to terminate the meridians. For example, setting `MLineLimit` to `[-75 75]` completely clears the region above and below this latitude range of meridian lines.
- If you want some lines to reach the poles but not others, you can specify 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.



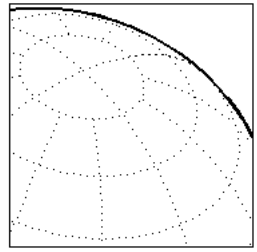
Default grid allows all displayed meridians to extend to the poles:

```
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 MATLAB 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)
```

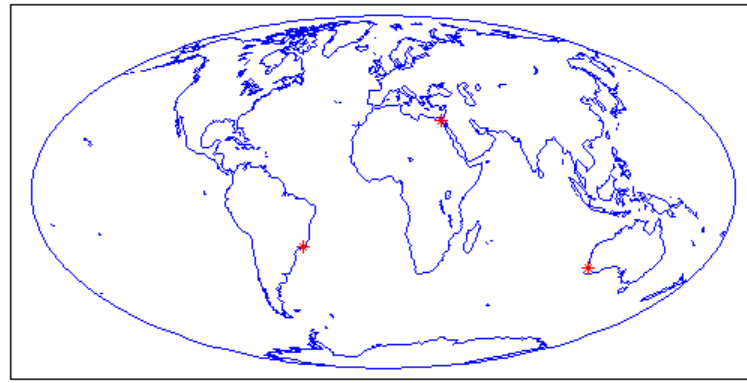
2 Plot the `coast` vector data using `plotm`. Just as with `plot`, you can specify line property names and values in the command.

```
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.

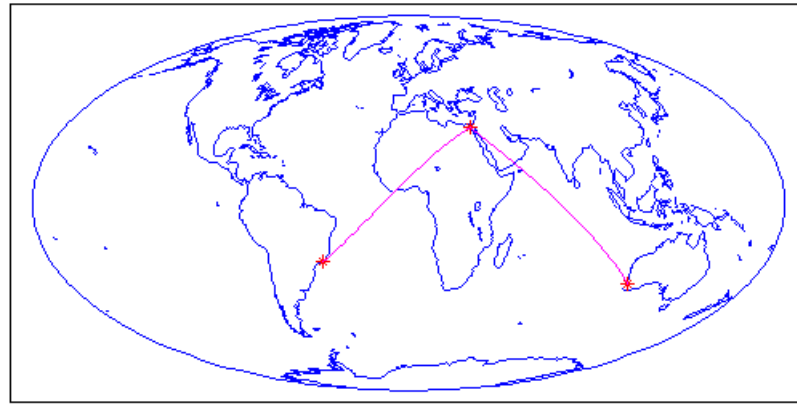
3 Define the three city geographic locations and plot symbols at them:

```
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 2-21, which includes examples of creating geostructs and displaying their contents in "How to Construct Geographic Data Structures" on page 2-25.

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 2-38.

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)
```

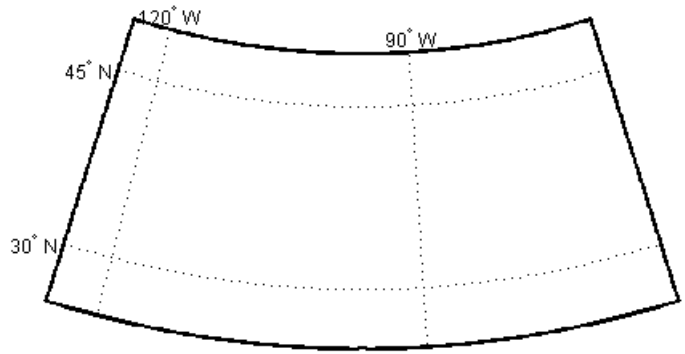
c If you want to add white space around the map, you can do so as follows (here, 2 degrees are added):

```
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:

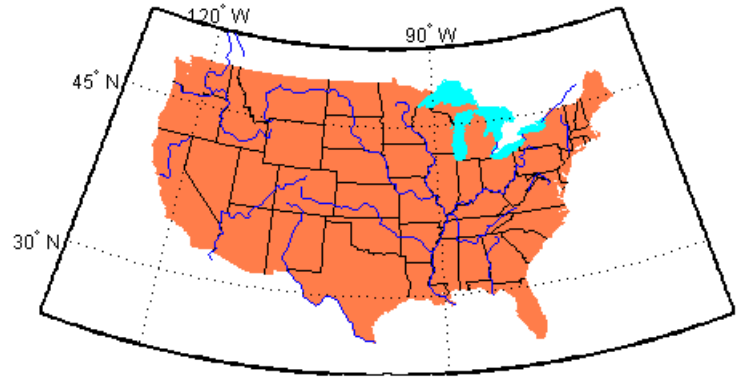
```
geoshow(gtlakelat,gtlakelon, 'DisplayType','polygon',...
    'FaceColor','cyan', 'EdgeColor','none')
```

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

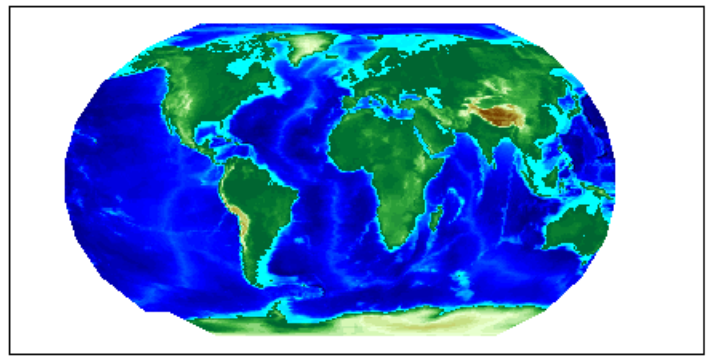
The toolbox projects surface objects in a manner similar to the traditional methods of mapmaking. A cartographer first lays out a grid of meridians and parallels called the *graticule*. Each graticule cell is a geographic quadrangle. The cartographer calculates or interpolates the appropriate *x*-*y* 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)
```

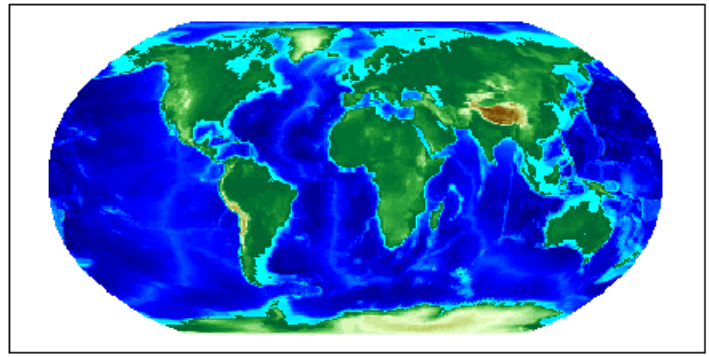


Notice that for this coarse graticule, the edges of the map do not appear as smooth curves. Previous displays used the default [50 100] graticule, for which this effect is negligible.

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.

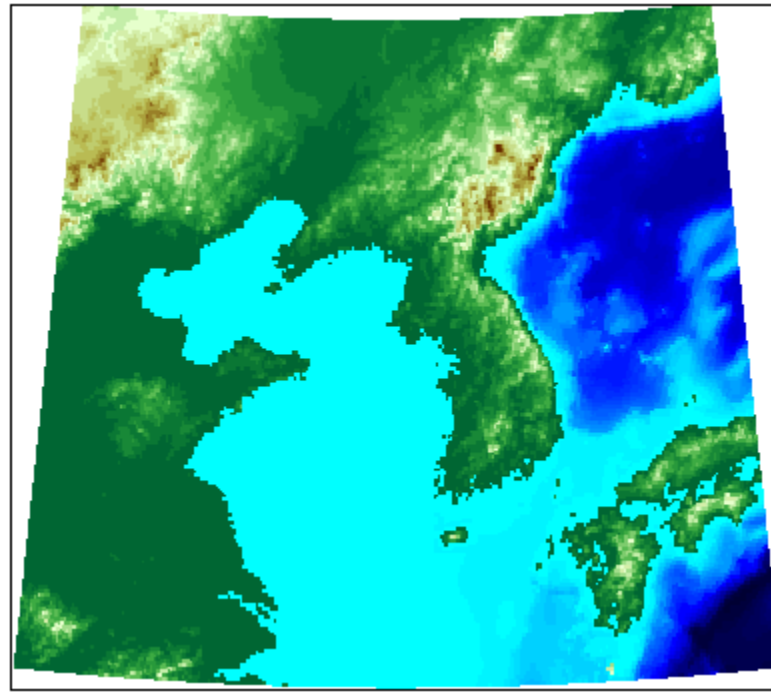
4 Now set up a map axes object with the equal area conic projection:

```
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.

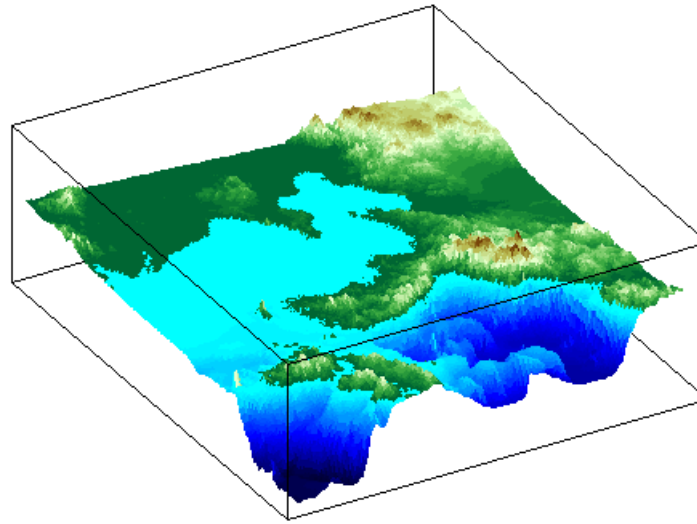


Be aware, however, that this map is really a 3-D view seen from directly overhead (the default perspective). To appreciate that, all you need to do is to change your viewpoint.

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

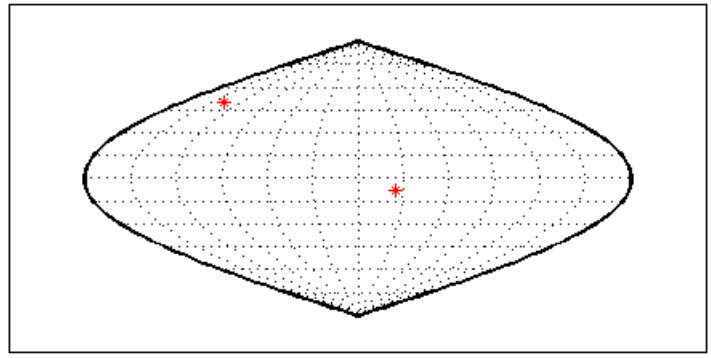
In this section...

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 Creating an Interactive Map for Selecting Point Features `mapexfindcity` 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:

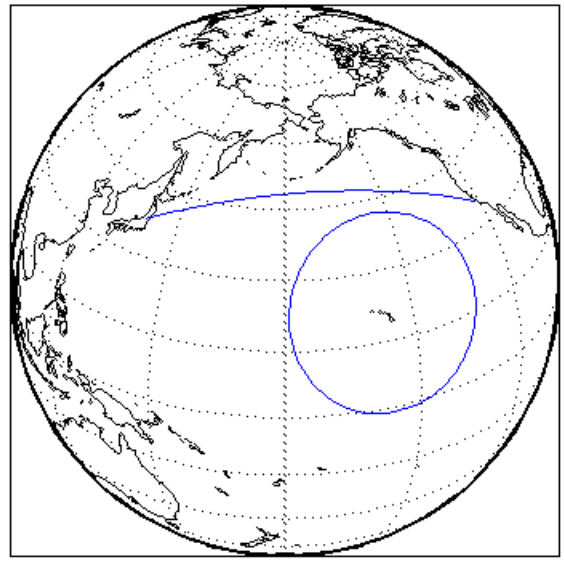
   ```
   trackg
   Track1:  Click on starting and ending points
   ```

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 R**adius** field.

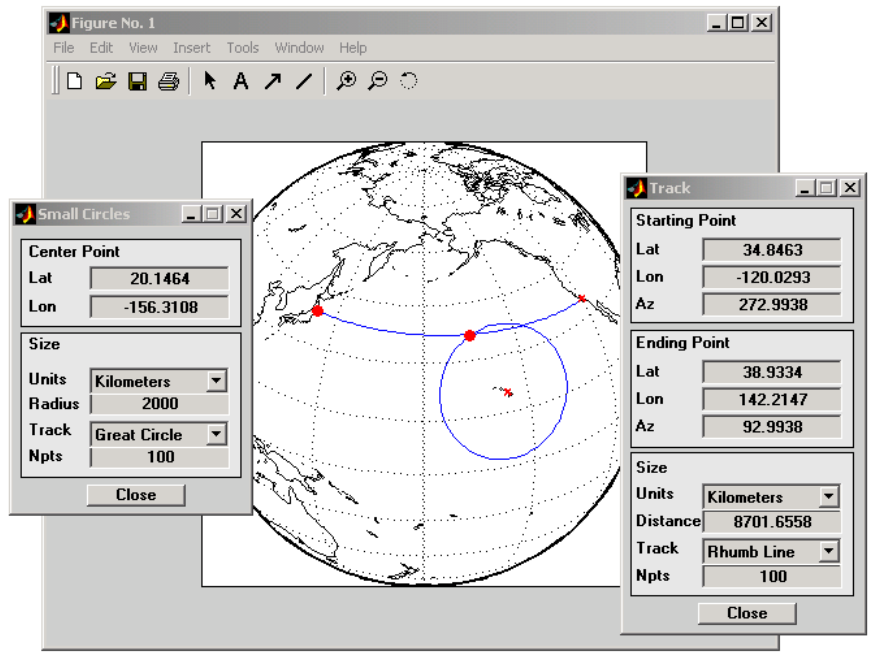
6 Click **Close**. The small circle readjusts to be 2000 km around Hawaii.

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.



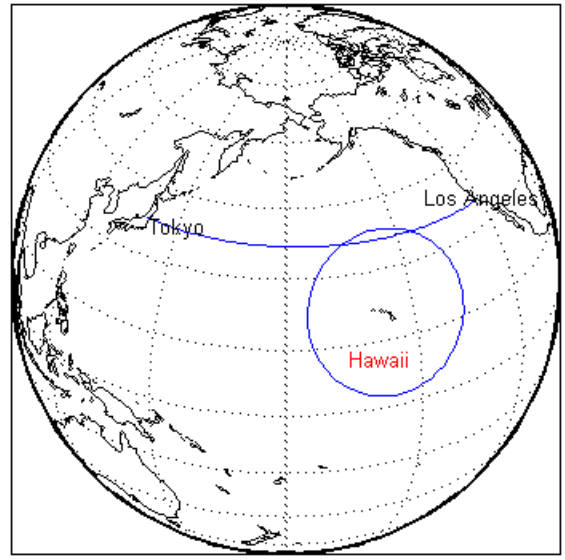
Interactive Text Annotation

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 4-45. 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.

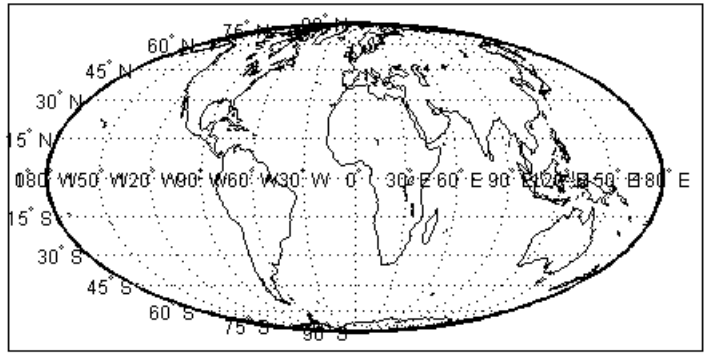
Some mapping display functions like `framem`, `gridm`, and `contourm` assign object tags by default. You can also set the name upon display by assigning a string to the `Tag` property in mapping display functions that use property name/property value pairs. If the `Tag` does not contain a string, the 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.



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 4-70 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
ETOPO1c, ETOPO2v2c, ETOPO2–2001, and ETOPO5	Earth Topography – 1-minute (ETOPO1c), 2-minute (ETOPO2v2c and ETOPO2–2001), and 5-minute (ETOPO5)	`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);
```

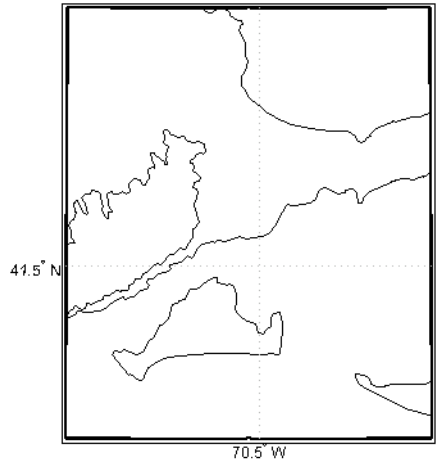
7 Read data for the region defined by the latitude and longitude limits from the `usastatehi` shapefile:

```
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');
```

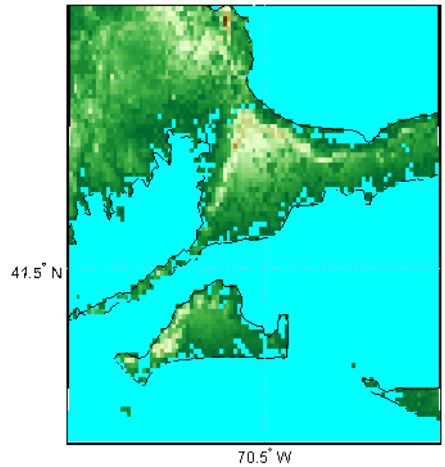
At this point the map looks like this:



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 5-7.

---

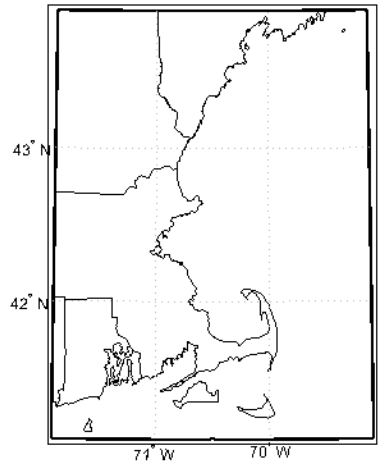
4 As DTEDs contain no bathymetric depths, recode all zero elevations to -1, to enable water areas to be rendered properly:

```
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');
```

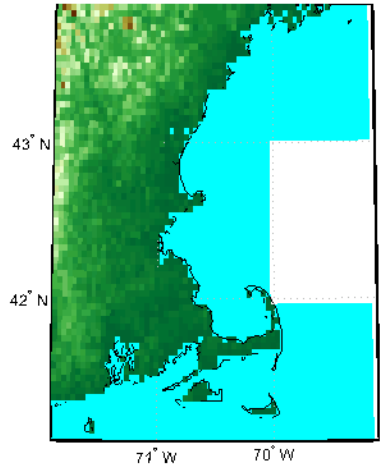
The map now looks like this.



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.

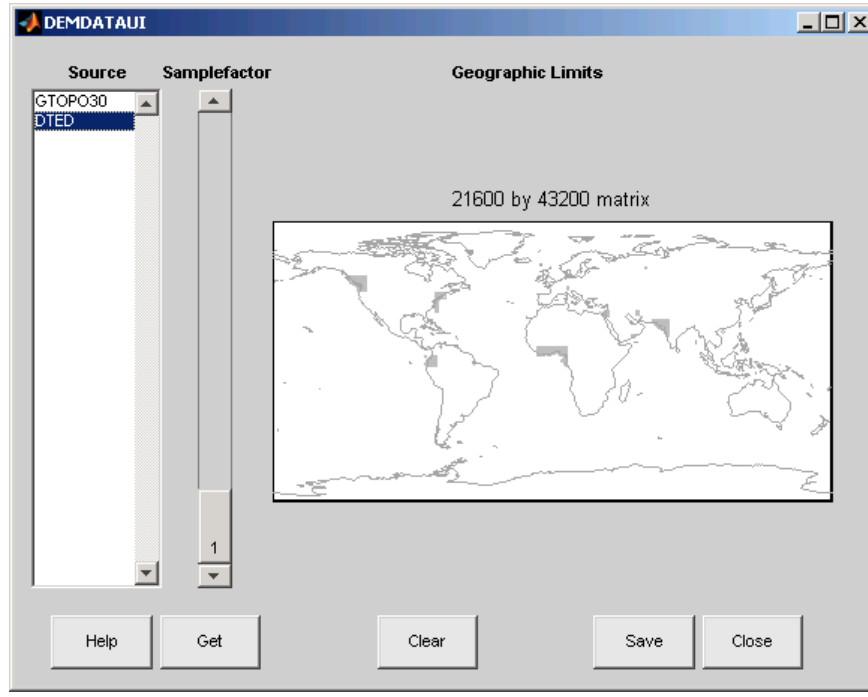
You can choose to read from any of the data sets `demdataui` has located. If `demdataui` does not recognize data you think it should find, check your path and click **Help** to read about how files are identified.

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:

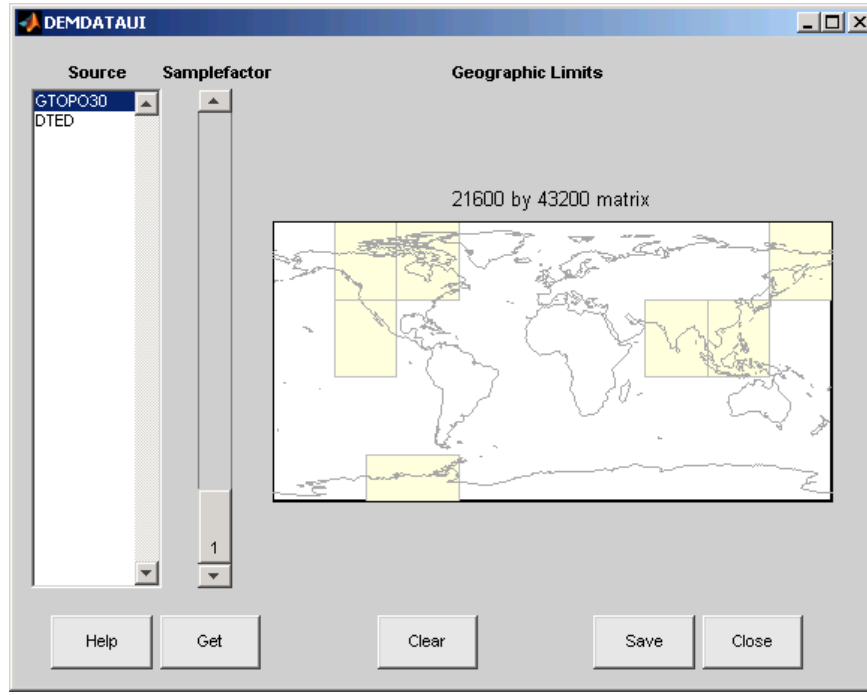
1. Open the `demdataui` UI. It scans the path for data before it is displayed:

```
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.

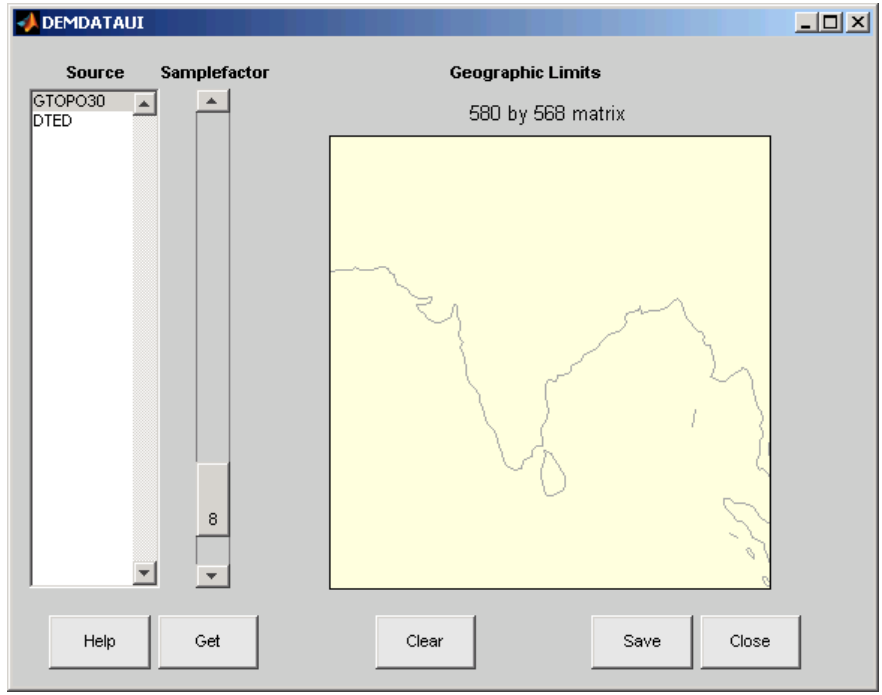


2 Clicking a different source in the left column updates the coverage display. Here is the coverage area for available GTOPO30 tiles.

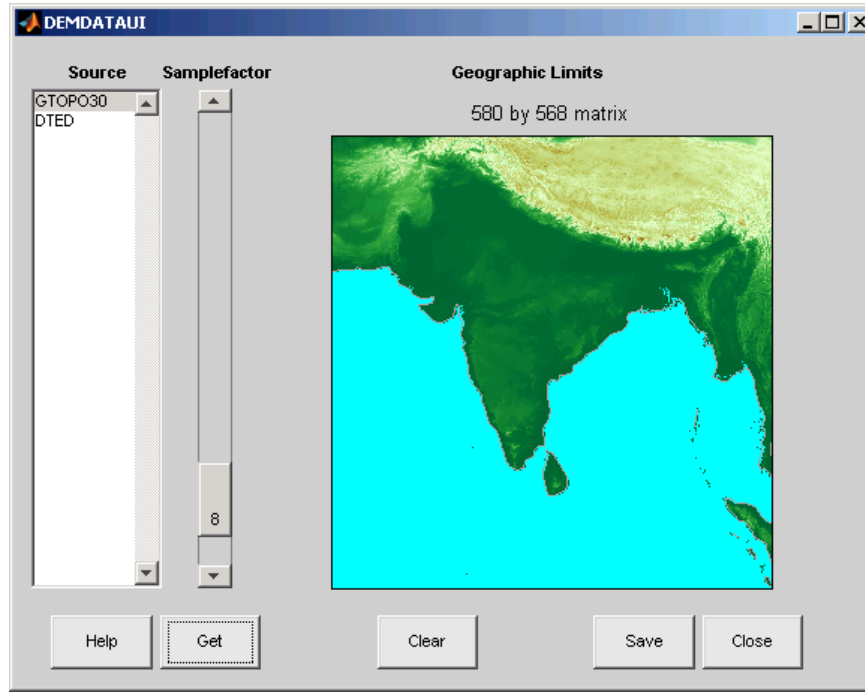


3 Use the map in the UI to specify the location and density of data to extract. To interactively set a region of interest, click in the map to zoom by a factor of two centered on the cursor, or click and drag across the map to define a rectangular region. The size of the matrix of the area currently 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.



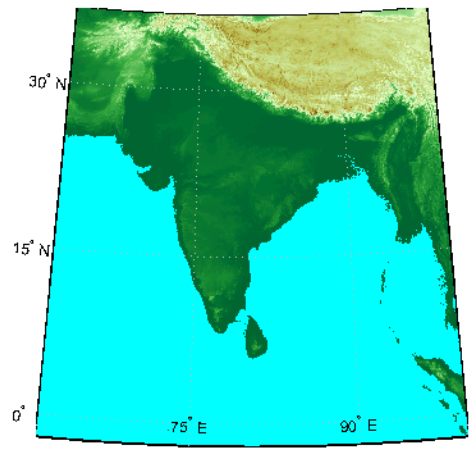
5 If you are not satisfied with the result, click the **Clear** button to remove all data previously read in via **Get** and make new selections. You might need to close and reopen `demdatui` in order to select a new region of interest.

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, a setting which is frequently used in modeling 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 mutually visible—*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):

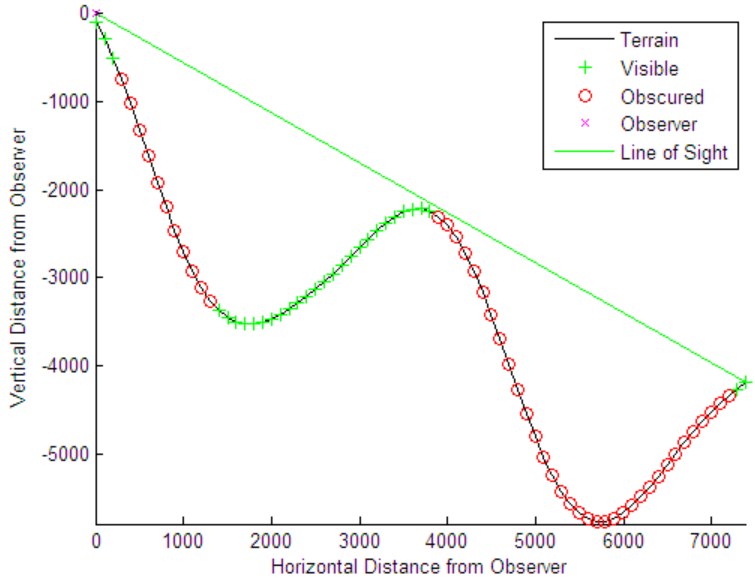
```
map = 500*peaks(100);
maplegend = [ 1000 0 0];
```

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 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 indicates the elements of a terrain elevation grid that are visible from a particular location. The `viewshed` function checks for a line of sight between a fixed observer and each element in the grid. See the `viewshed` function reference page for an example.

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 5-24 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.

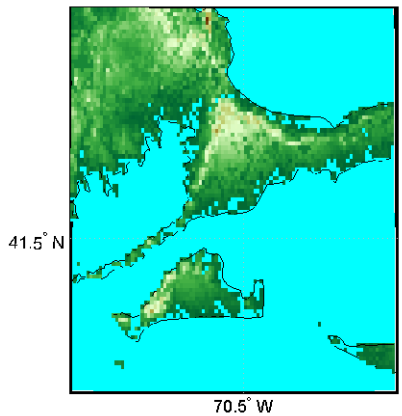
1 To illustrate lighting terrain maps, begin by following the exercise in "Mapping a Single DTED File with the DTED Function" on page 5-7, or execute the steps below:

```
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);
set(gcf,'Renderer','zbuffer')
geoshow(ax,capecoast,'FaceColor','none');
geoshow(ax,capeterrain,caperef,'DisplayType','texturemap');
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`:

```
lightm(42,-71)
```

If you omit arguments, a GUI for setting positional properties for the new light opens.

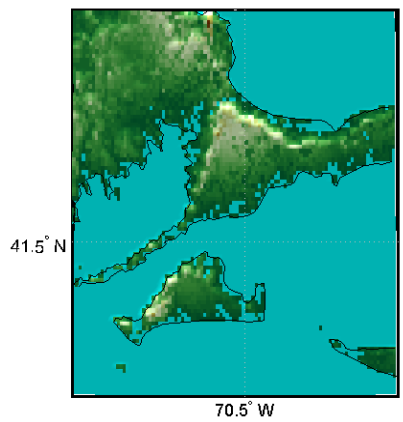
6 The lighting computations caused the map to become quite dark with specular highlights. Now restore its luminance by specifying three surface reflectivity properties in the range of 0 to 1:

```
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.



7 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])
axis off
set(gcf,'Renderer','zbuffer')
```

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:

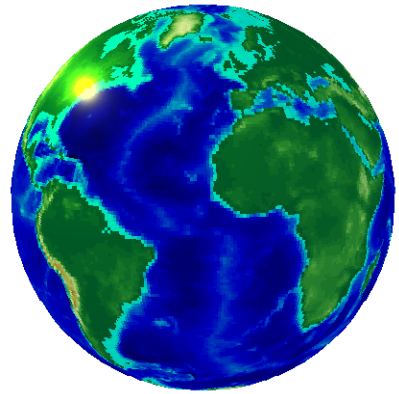
```
lightm(40.75,-73.9,500/earthRadius('km'),...
   'color','yellow','style','local')
```

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.

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:

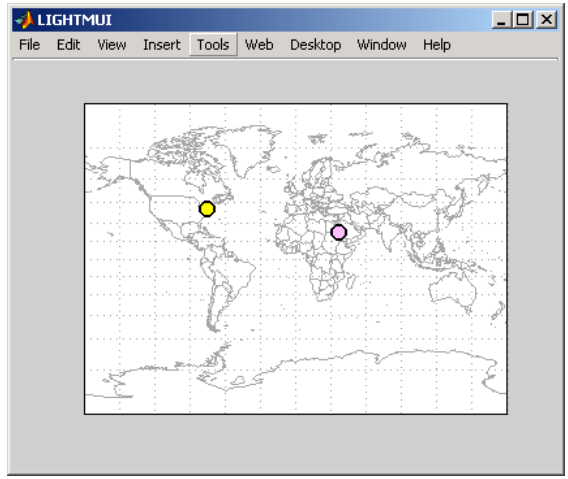
```
lightm(20,40,0.1,'color','magenta','style','local')
```

The second light is magenta, and positioned over the Gulf of Arabia.

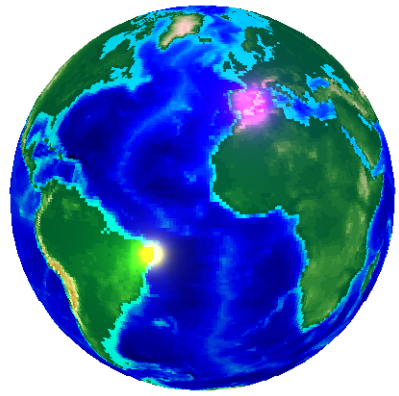
6 To modify the lights, use the `lightmui` GUI, which lets you drag lights across a world map and specify their color and altitudes:

```
lightmui(gca)
```

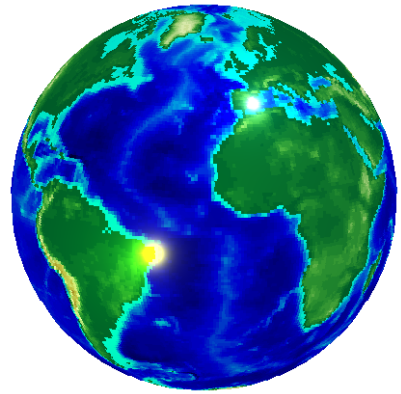
The lights are shown as appropriately colored circles, which you can drag to new positions. You can also **Ctrl**+click a circle to bring up a dialog box for directly specifying that light's position, altitude, and color. The GUI and the map look like this at this point.



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 4-71). The following example illustrates this procedure.

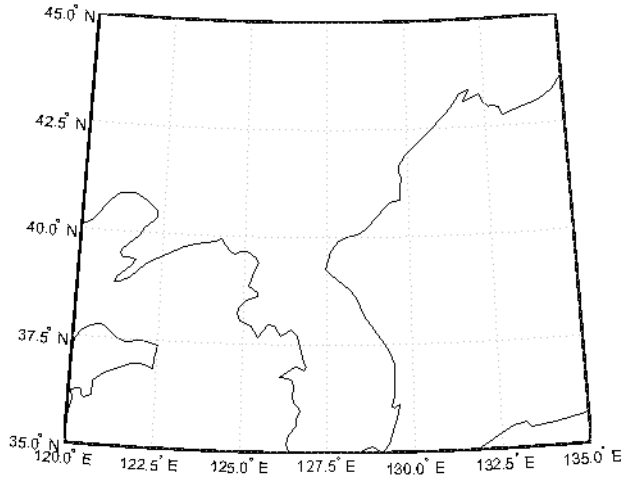
Creating Monochrome Shaded Relief Maps Using surflm

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
```

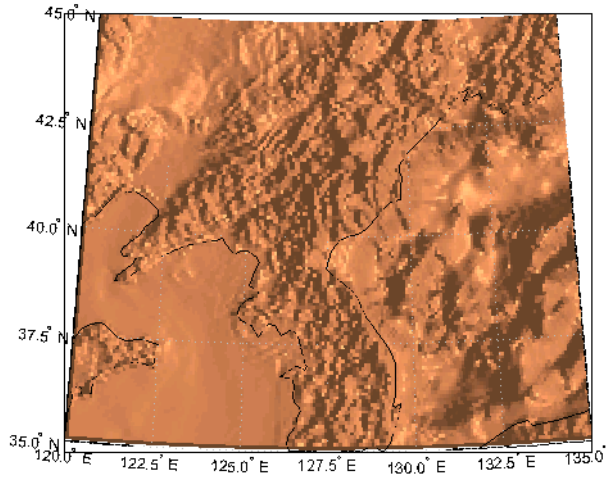
3 Generate the grid of latitudes and longitudes to transform the regular data grid to a geolocated one:

```
[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')
```

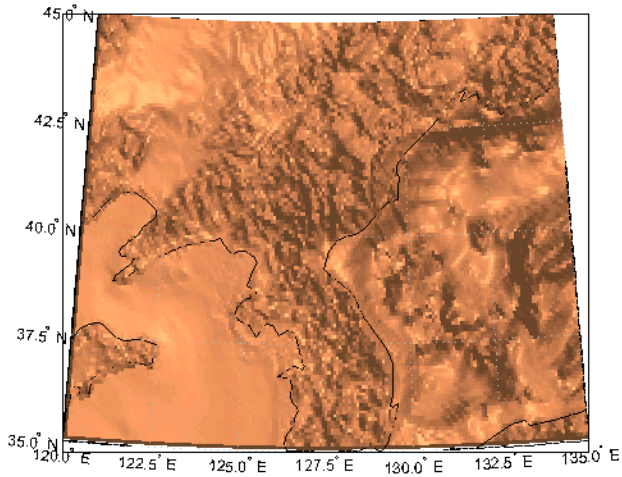
In this default case, the lighting direction is set at 45º counterclockwise from the viewing direction; thus the "sun" is in the southeast. This map is shown below.



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 135º (northeast) and an elevation of 60º above the horizon:

```
clmo(ht); ht=surflm(klat,klon,map,[135,60]);
```

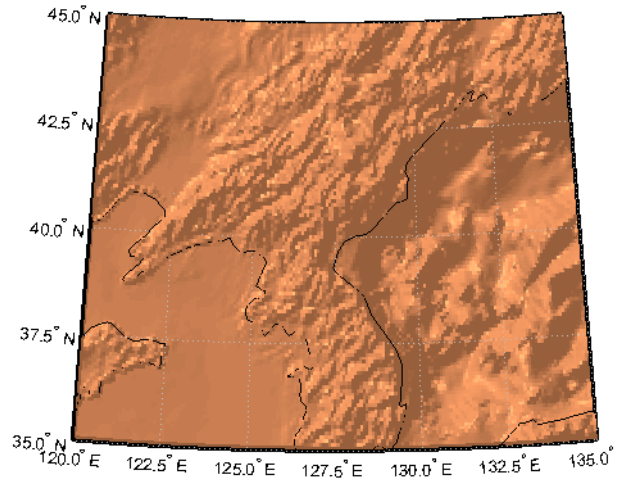
The surface lightens and has a new character because it is lit closer to overhead and from a different direction.



6 Now shift the light to the northwest (-135° azimuth), and lower it to 40° 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 5-34 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 5-27. 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:

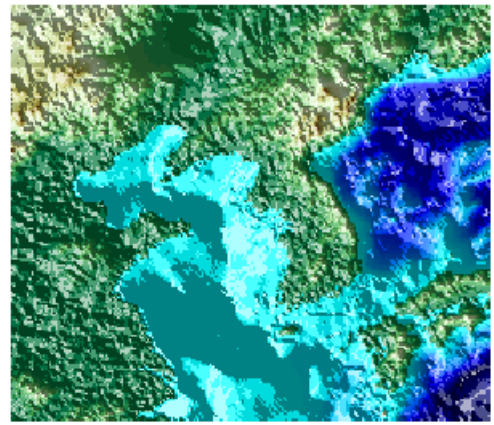
   ```
   surflsrm(klat,klon,map,[-130 50],cmap,clim)
   ```

   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)
   ```

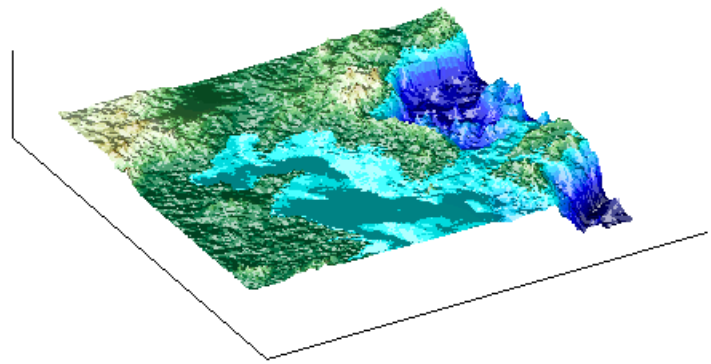
   The map, which has an overhead view, looks like this.



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 -30º (south-southwest) and view altitude to 30º 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 5-24, 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 5-32), 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:

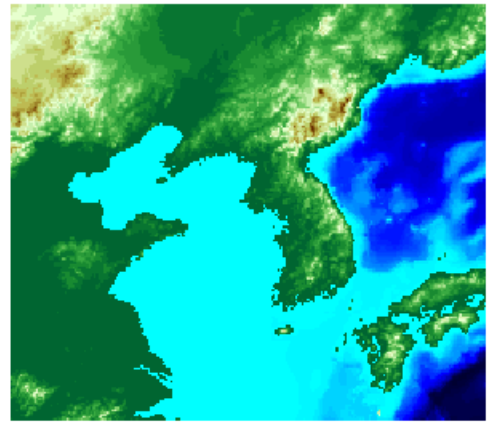
1. If you need to, load the `korea` DEM, and create a map axes using the Miller projection:

```
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.

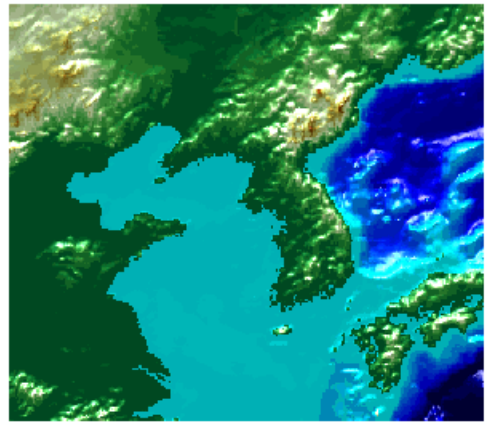
5 Set the ambient (direct), diffuse (skylight), and specular (highlight) surface reflectivity characteristics, respectively:

```
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
```

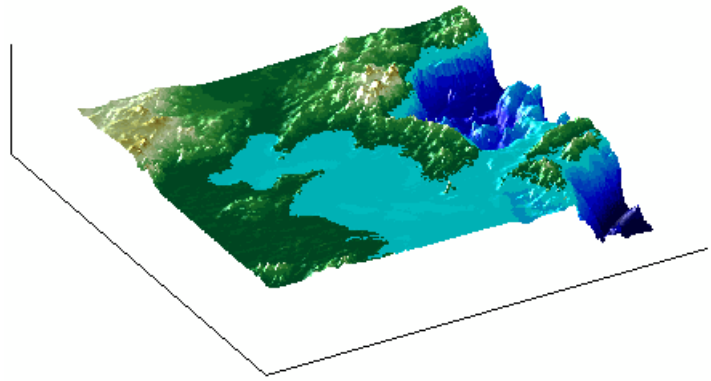
The map now looks like this.



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 5-32.



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:

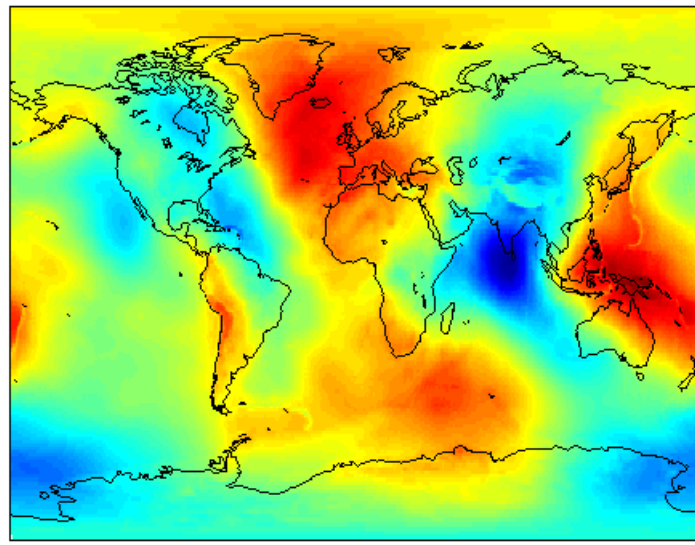
```
meshm(geoid,geoidrefvec,size(geoid),topo)
```

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:

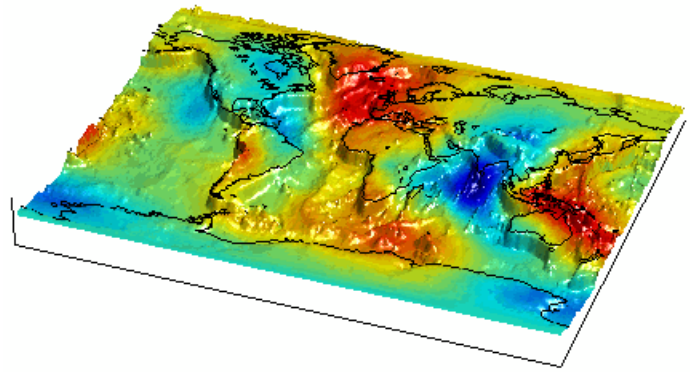
```
set(gca,'Box','off')
camlight;
lighting phong
```

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.

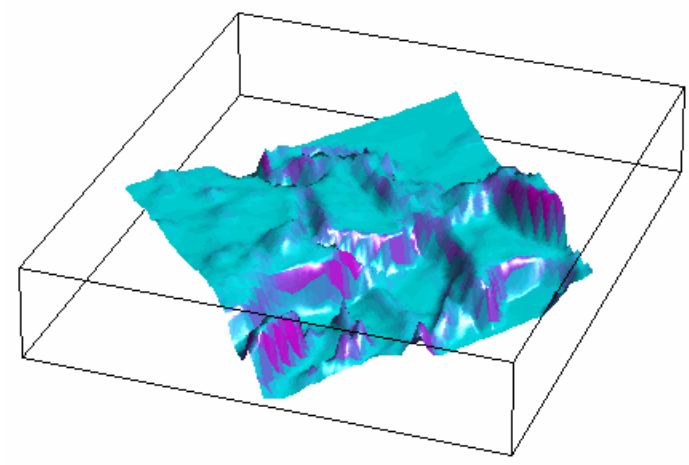
7 The default rendering uses faceted shading (no smooth interpolation). Render the surface again, this time making it shiny with Phong shading and lighting from the east (the default of `camlight` lights surfaces from over the viewer's right shoulder):

```
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);
```

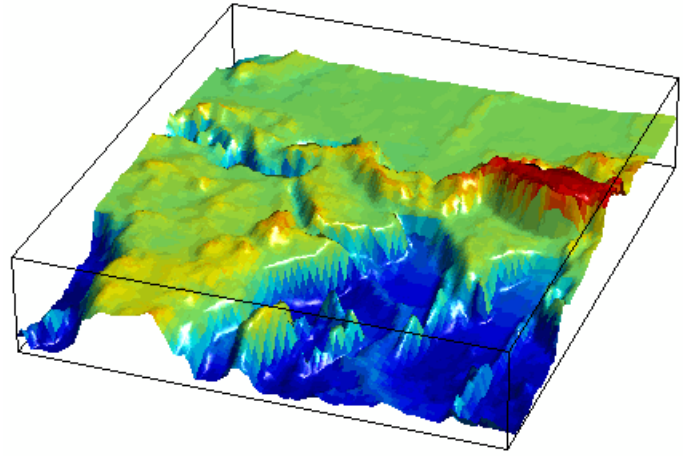
7 Render the figure as a 3-D view from a 20º azimuth and 30º altitude, and exaggerate the vertical dimension by a factor of 200:

```
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')
```

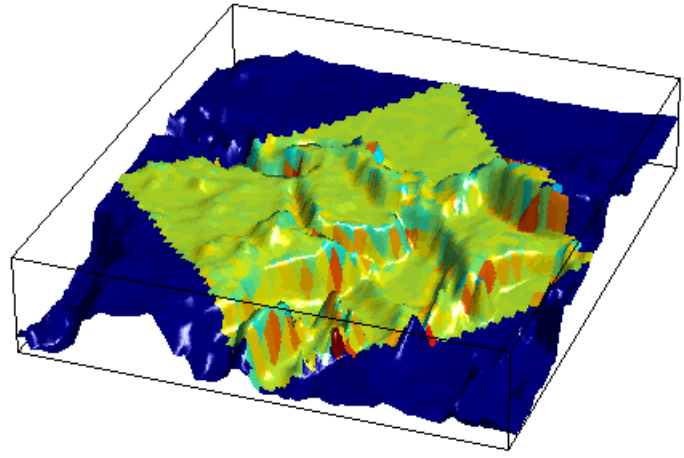
So far, both the surface relief and coloring represent topographic elevation.



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)
   ```

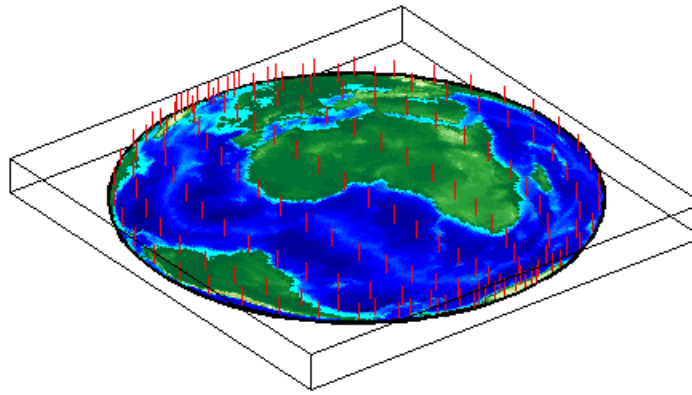
2. View the map obliquely:

   ```
   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',earthRadius)
```

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')
```

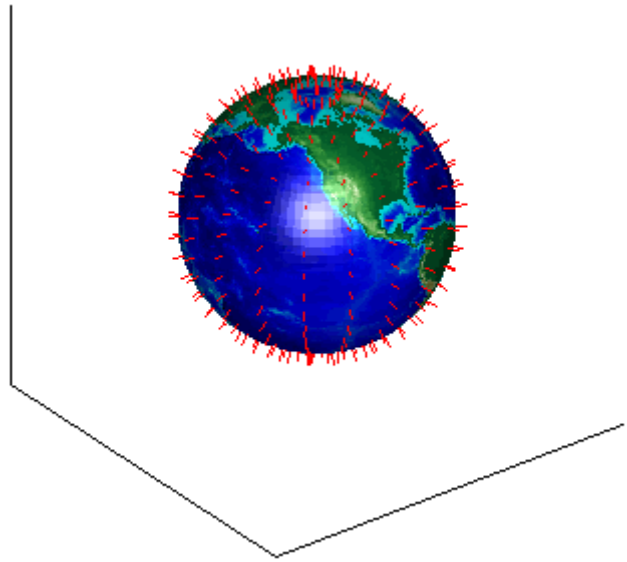
7 You can apply lighting to the display, but its location is fixed, and does not move as the camera position is shifted:

```
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)
axis off; zoom(2)
```

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), ...
   'Latlim',[-90 90],'Lonlim',[0 360]);
```

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)
```

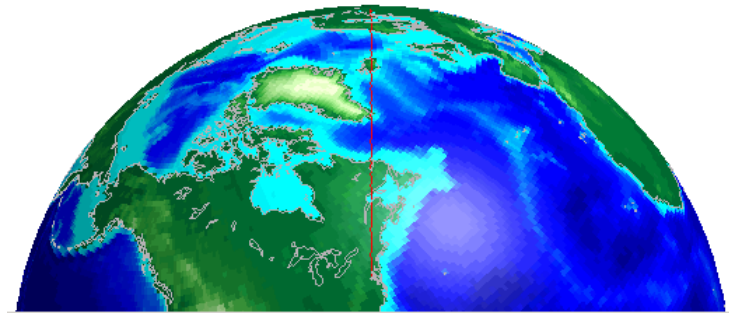
9 Set the field of view for the camera to 20° for the final view:

```
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 a MATLAB program that resets the view to rotate the globe from west to east in one-degree increments.

1. In the Mapping Toolbox editor, create a MATLAB program 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.

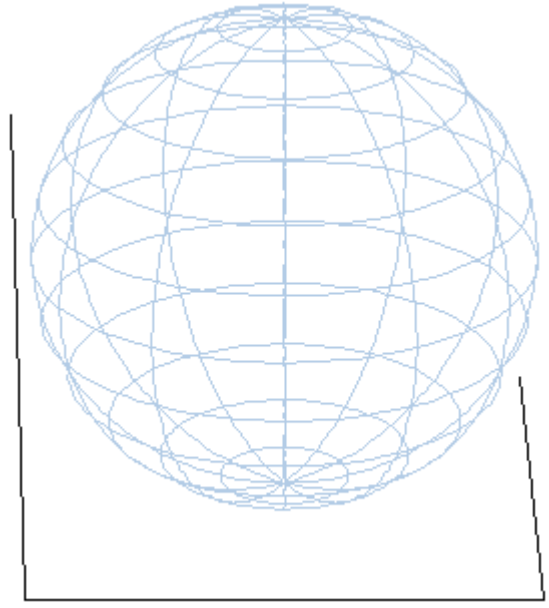
3 Show the axes, but hide the edges of the figure's box, and view it in perspective rather than orthographically (the default perspective):

```
set(gca, 'Box','off', 'Projection','perspective')
```

4 Spin the globe one revolution:

```
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 5-50:

```
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.

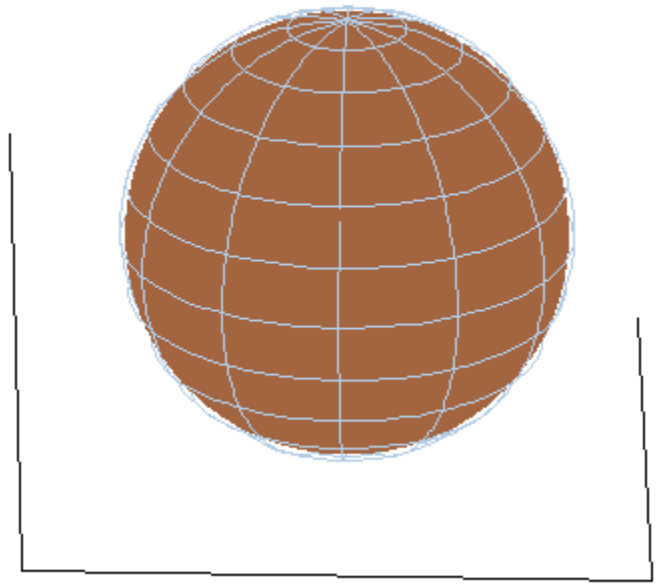
6 Pop up the grid so it appears to float 2.5% above the surface. Prevent the display from stretching to fit the window with the `axis vis3d` command:

```
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
```

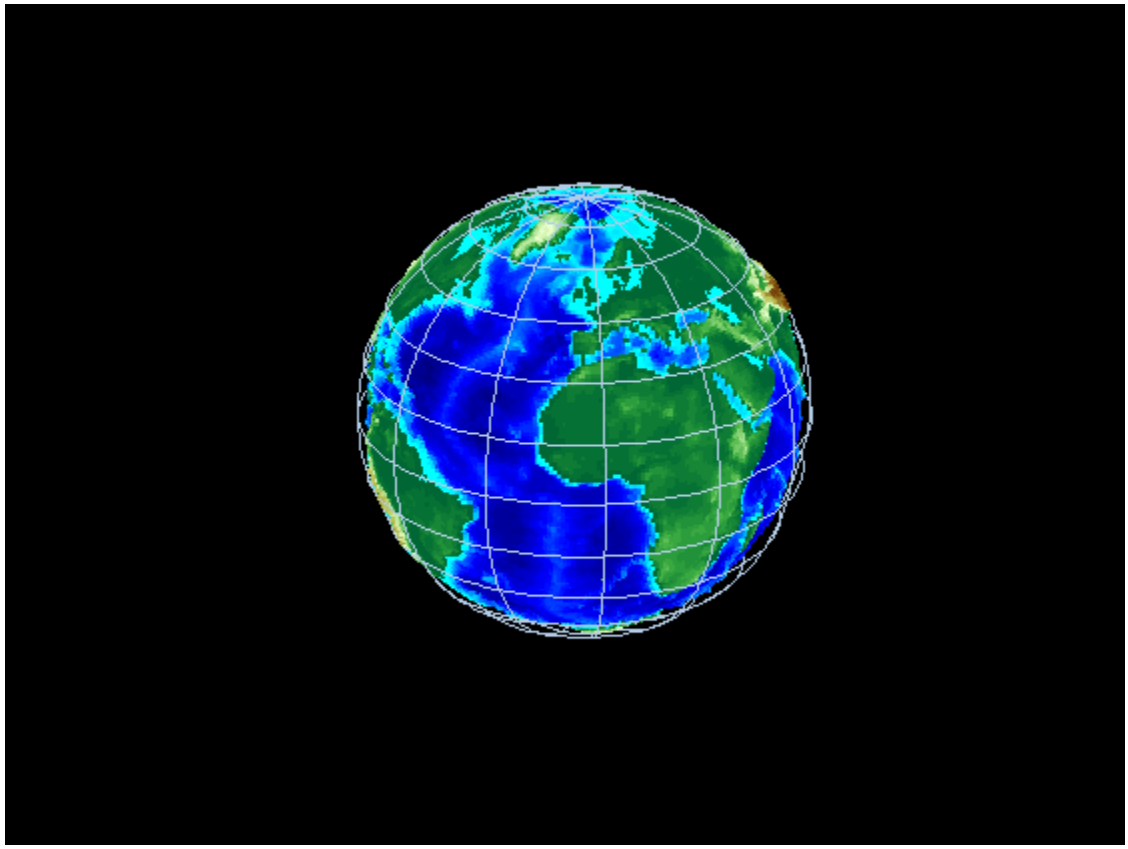
9 Scale the elevations to have an exaggeration of 50 (in units of Earth radii) and plot the surface:

```
topo = topo / (earthRadius('km')* 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
```

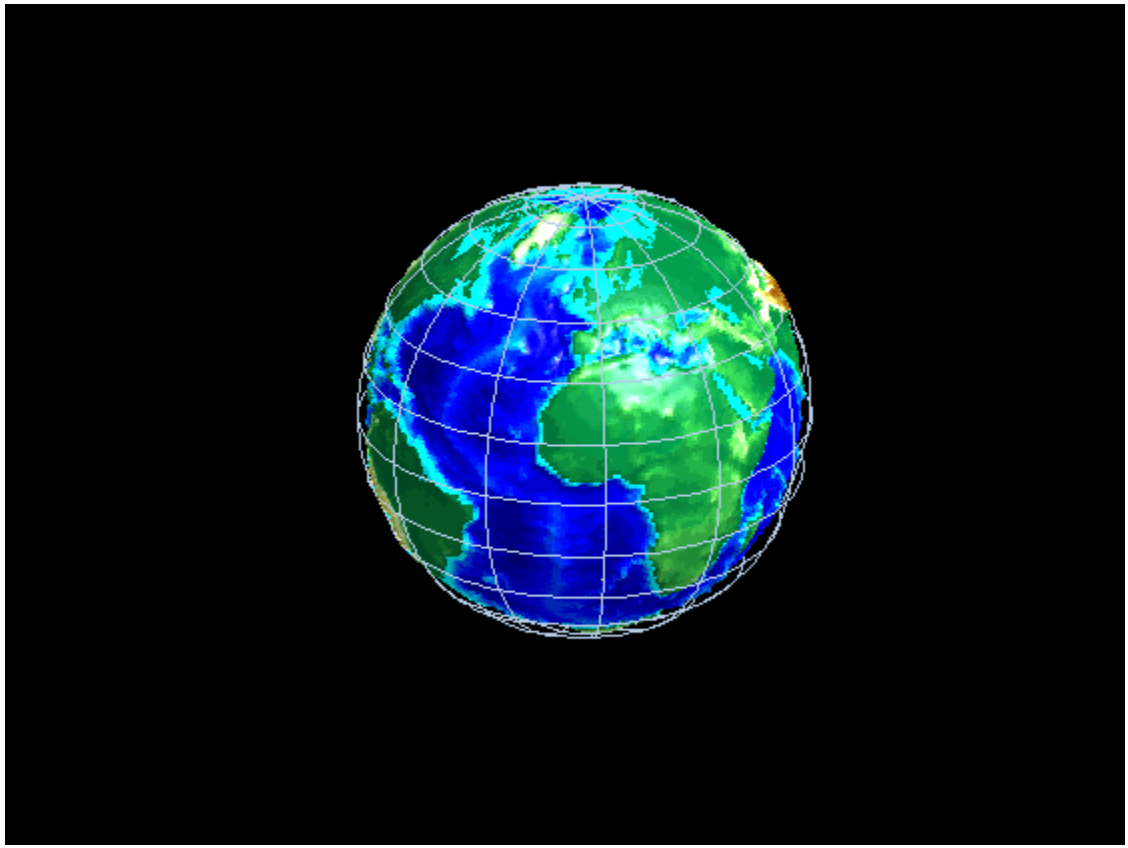
Here is a representative view, showing the Himalayas rising on the Eastern limb of the planet and the Andes on the Western limb.



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.

6

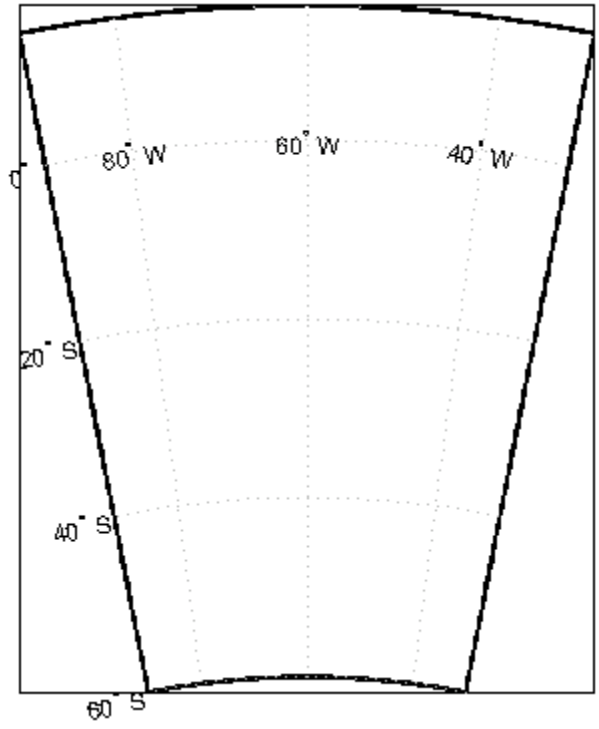
Customizing and Printing Maps

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');
```

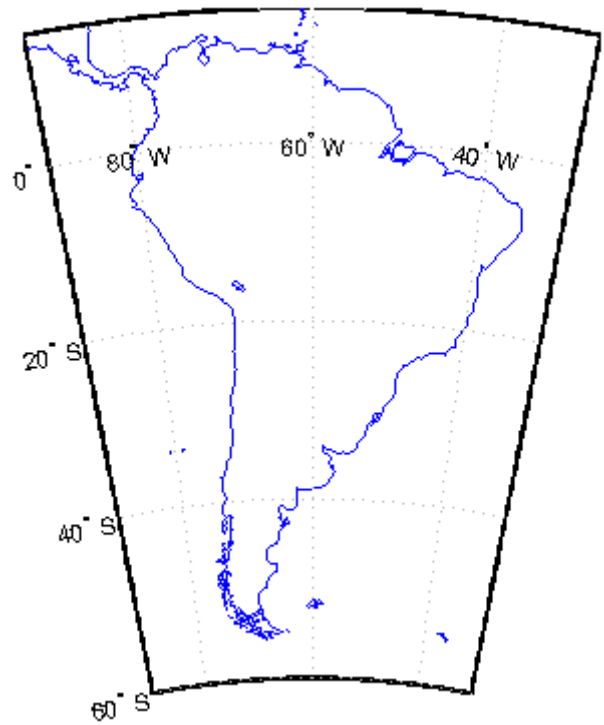


2 Use `shaperead` to read the demo world land areas polygon shapefile:

```
land = shaperead('landareas.shp', 'UseGeoCoords', true);
```

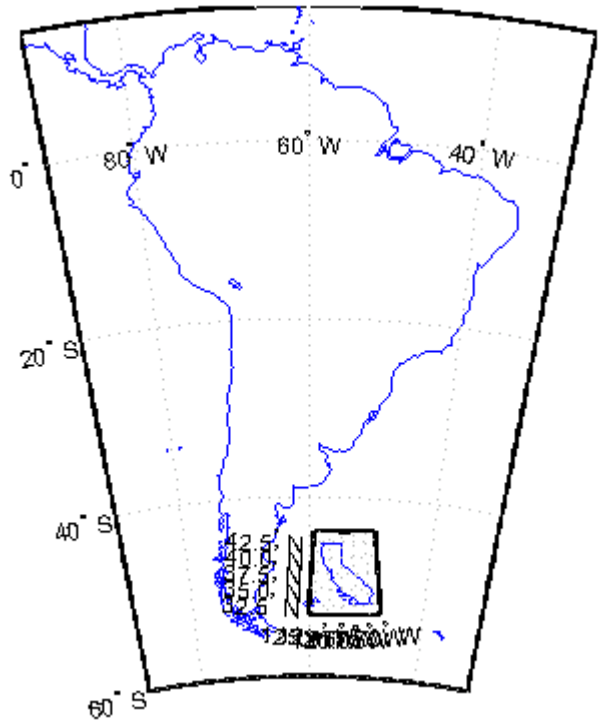
3 Display the data in the map axes:

```
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])
```

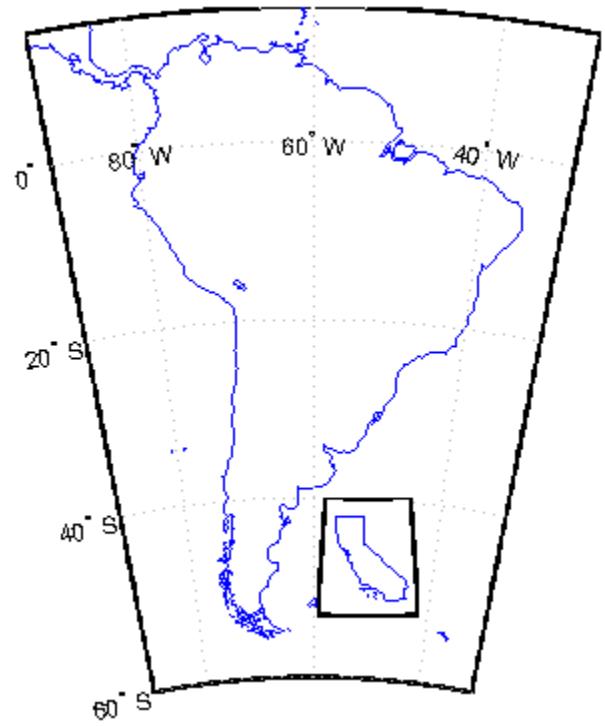


5 Set the frame fill color and set the labels:

```
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:

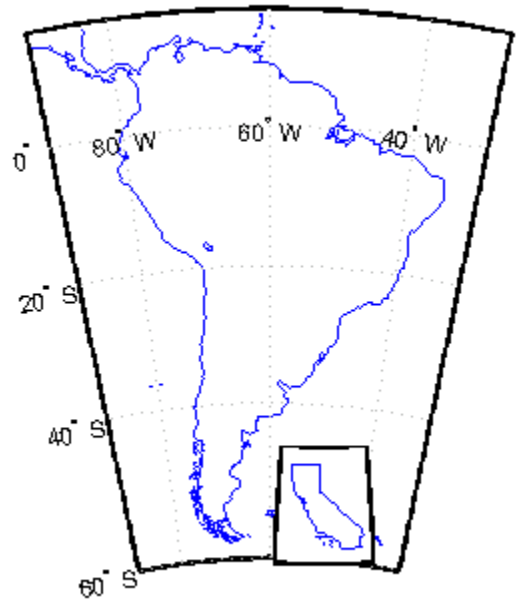
```
axesscale(h1)
set([h1 h2], 'Visible', 'off')
```



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:

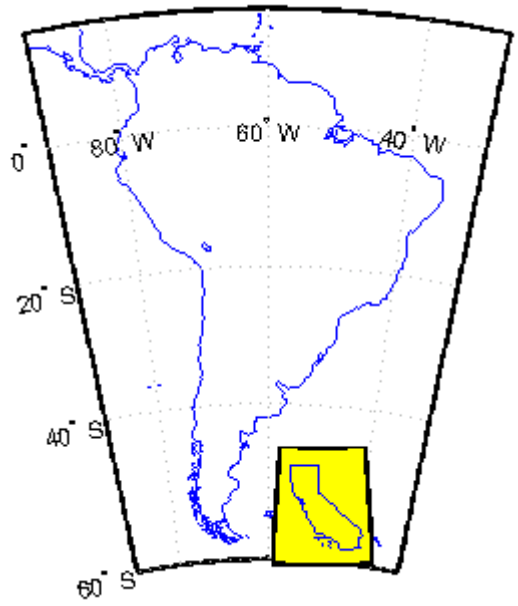
```
getm(h1, 'mapprojection')
ans =
      eqdconic

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')
```


0°
80° W
60° W
40° W
20° S
40° S
60° S

Graphic Scales

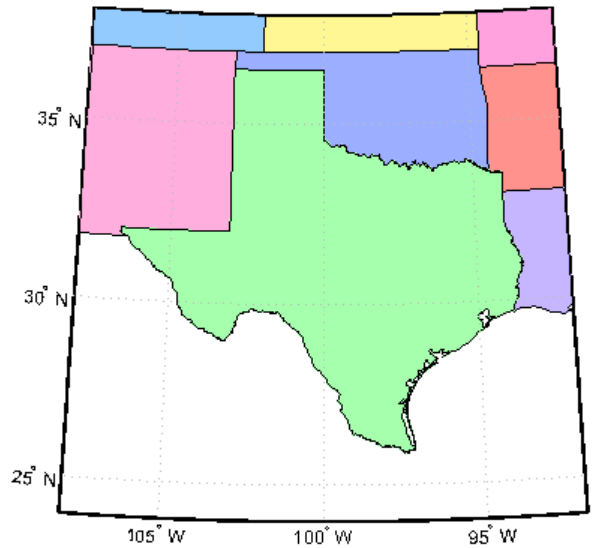
Graphic scale elements are used to provide indications of size even more frequently than insets are. These are ruler-like objects that show distances on the ground at the nominal scale of the projection. You can use the `scaleruler` function to add a graphic scale to the current map. You can check and 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)
```

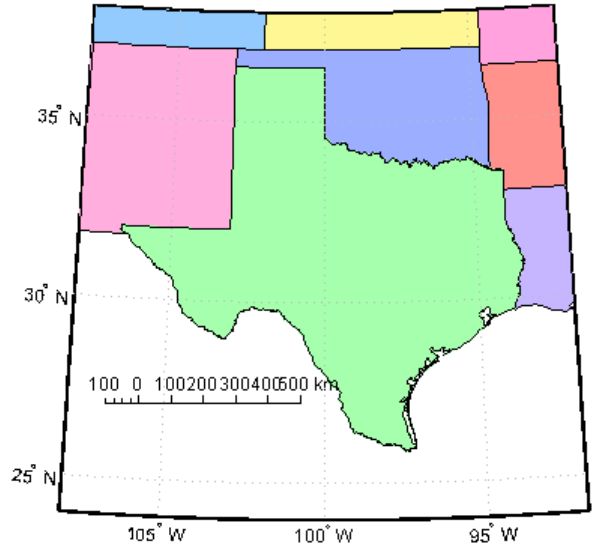
Because `polcmap` randomizes patch colors, your display can look different.



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:

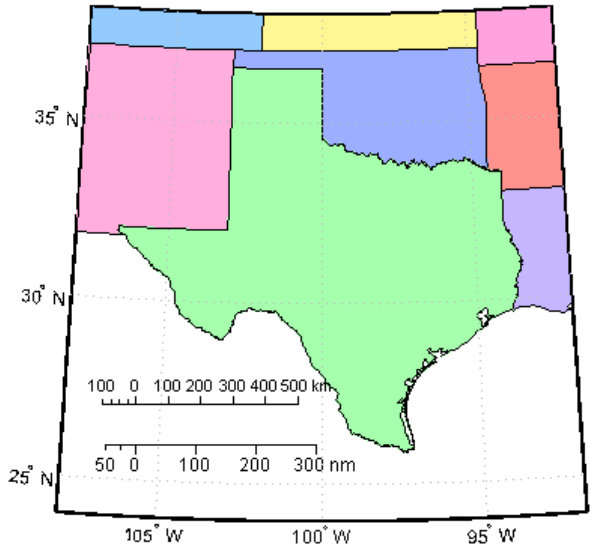
```
setm(s,'fontsize',8)
```

5 Place a second graphic scale, this one in units of nautical miles:

```
scaleruler('units','nm')
```

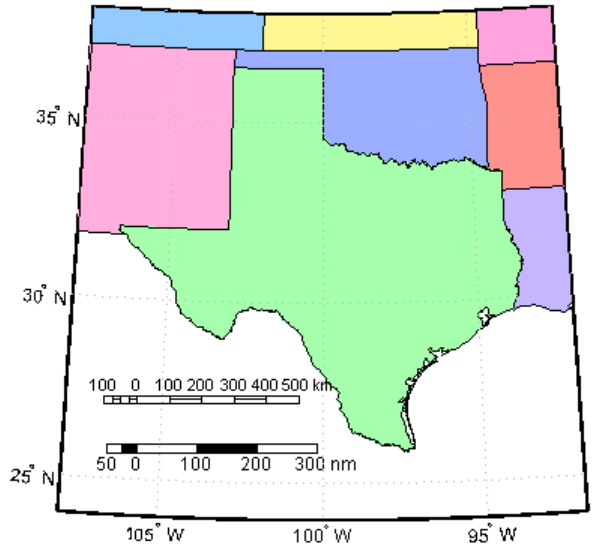
6 Modify its tick properties:

```
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')
```

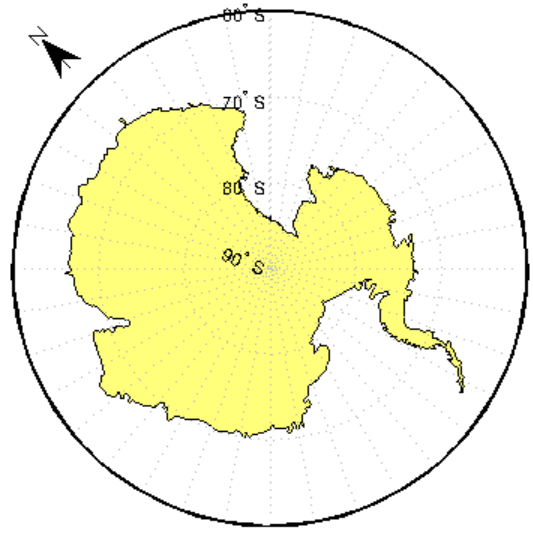

35° N
30° N
25° N
105° W
100° W
95° W
100 0 100 200 300 400 500 km
50 0 100 200 300 nm

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);
```



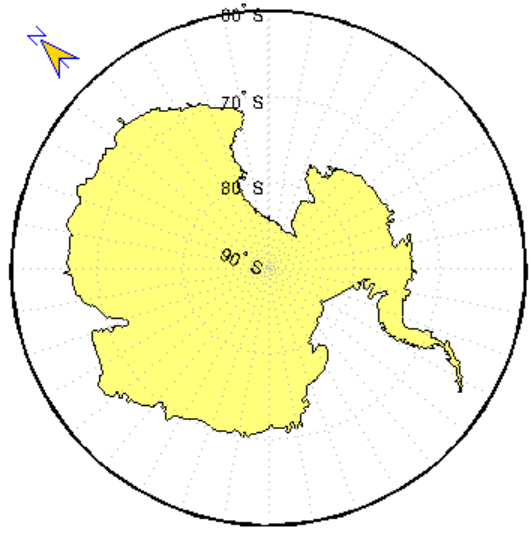
2 Click and drag the north arrow symbol to another corner of the map. Note that it always points to the North Pole.

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**.

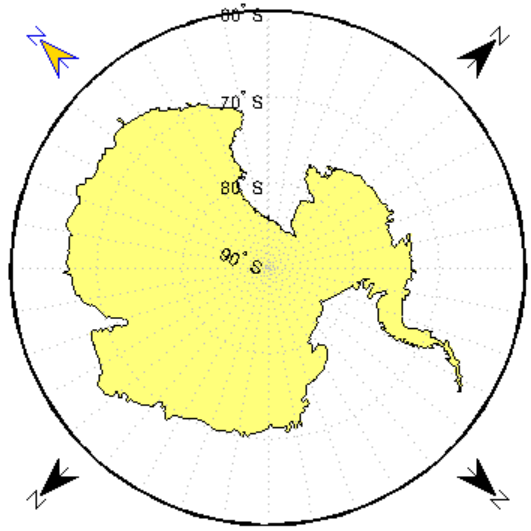
5 Also set some north arrow properties manually, just to get a feel for them:

```
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). Choropleth maps use colors or patterns to represent attributes associated with certain geographic regions. For example, the global distribution of malaria-carrying mosquitoes can be illustrated in a choropleth map, with the habitat of each mosquito represented by a different color. In this example, colors are used to represent nominal data; the categories of mosquitoes have no inherent ranking. If the data is ordinal, rather than nominal, the map may contain a colorbar with shades of colors representing the ranking. For instance, a map of crime rates in different areas could show high crime areas in red, lower crime areas in pink, and lowest crime areas in white.

Creating choropleth maps with the Mapping Toolbox is fairly straightforward. Start with a geographic data structure; create a symbolspec to map attribute values to face colors; and apply either `geoshow` or `mapshow`, depending on whether you are working with latitude-longitude or pre-projected map coordinates. The following example illustrates the process of creating a choropleth map of population density for the six New England states in the year 2000.

1. Set the map limits for the New England region. Import low-resolution U.S. state boundary polygons:

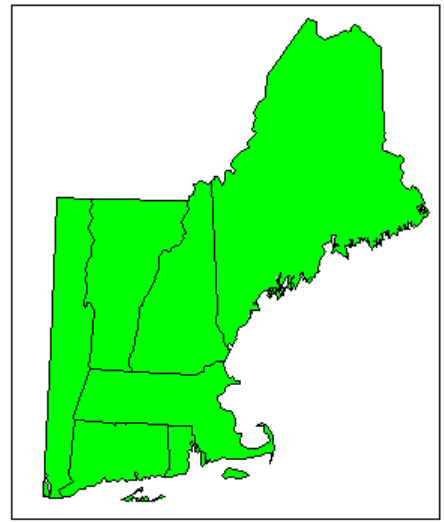
```
MapLatLimit = [41 48];
MapLonLimit = [-74 -66];

NEstates = shaperead('usastatelo', 'UseGeoCoords', true, ...
   'BoundingBox', [MapLonLimit' MapLatLimit']);
```

2. Set up map axes with a projection suitable to display the New England states:

```
axesm('MapProjection', 'eqaconic', 'MapParallels', [],...
  'MapLatLimit', MapLatLimit, 'MapLonLimit', MapLonLimit,...
  'GLineStyle', '-')
```

3. Display the New England states:

```
geoshow(NEstates, 'DisplayType', 'polygon', 'FaceColor','green')
```



4 Identify the maximum population density for New England states:

```
maxdensity = max([NEstates.PopDens2000]);
```

5 Create an `autumn` colormap for the six New England states, and then use the `flipud` command to invert the matrix.

```
fall = flipud(autumn(numel(NEstates)));
```

6 Make a symbol specification structure, a `symbolspec`, that assigns an autumn color to each polygon according to the population density.

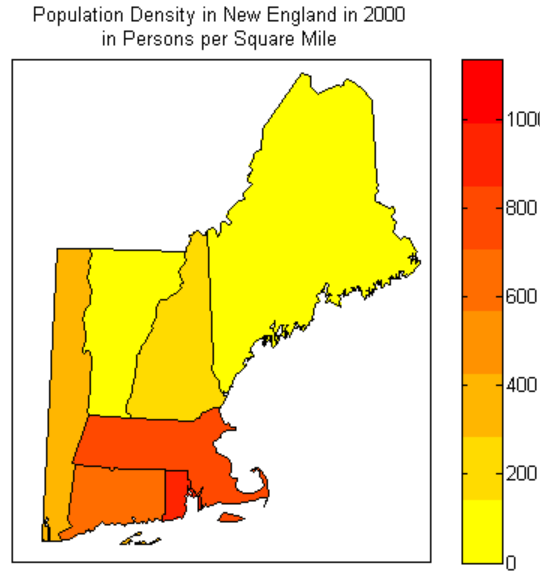
```
densityColors = makesymbolspec('Polygon', {'PopDens2000', ...
   [0 maxdensity], 'FaceColor', fall});
```

7 Display the map.

```
geoshow(NEstates, 'DisplayType', 'polygon', ...
   'SymbolSpec', densityColors)
title ({'Population Density in New England in 2000', ...
   'in Persons per Square Mile'})
```

8 Create a `colorbar`.

```
caxis([0 maxdensity])
colormap(fall)
colorbar
```



9 Experiment with other colormaps. Some names of predefined colormaps are `autumn`, `cool`, `copper`, `gray`, `pink`, and `jet`.

Special Thematic Mapping Functions

In addition to choropleth maps, other Mapping Toolbox display and symbology functions include

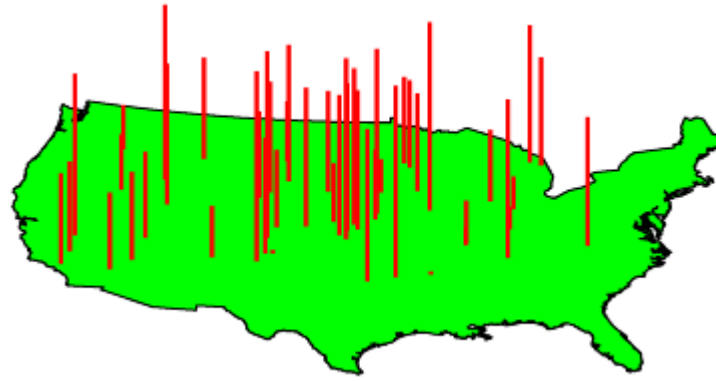
Function	Used For
`cometm`	Traces (animates) vectors slowly from a comet head
`comet3m`	Traces (animates) vectors in 3-D slowly from a comet head

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

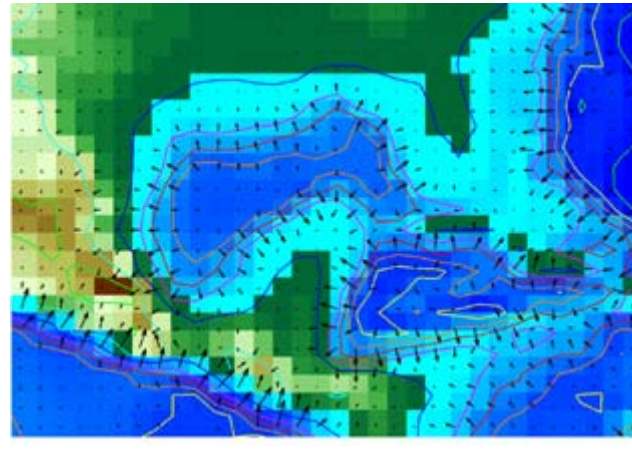
Stem plots are 3-D geographic bar graphs portraying numeric attributes at point locations, usually on vector base maps. Below is an example of a stem plot over a map of the continental United States. The bars could represent anything from selected city populations to the number of units of a product 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*.

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.

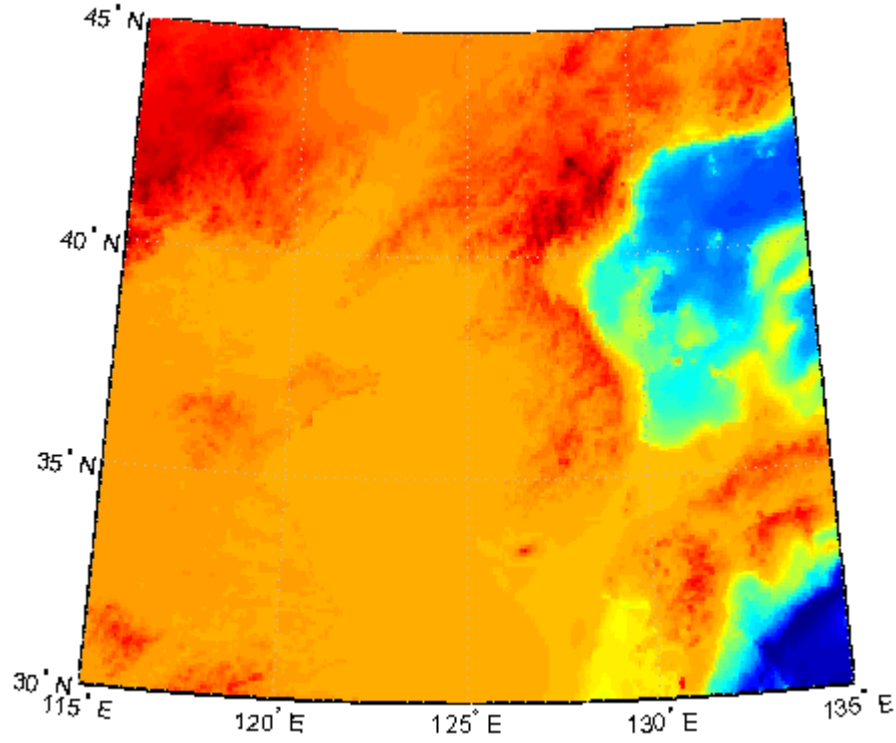
These colormaps, by default, have atlas-like colors varying with elevation or depth that properly preserve the land-sea interface. In cartography, such color schemes are called *hypsometric tints*.

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)
   ```

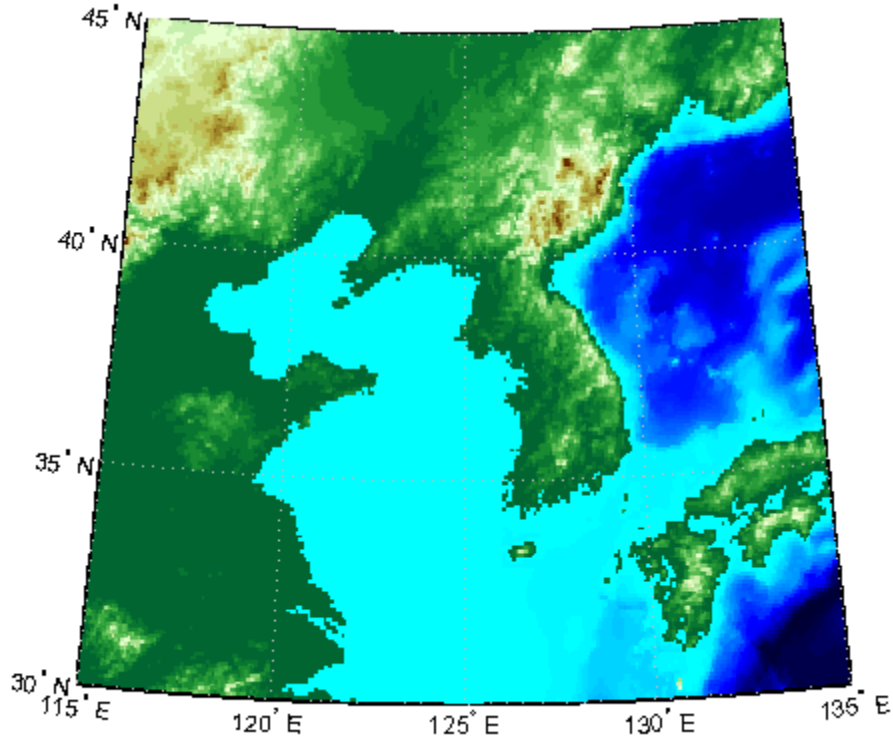
2. Display the data grid with `geoshow`:

   ```
   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:

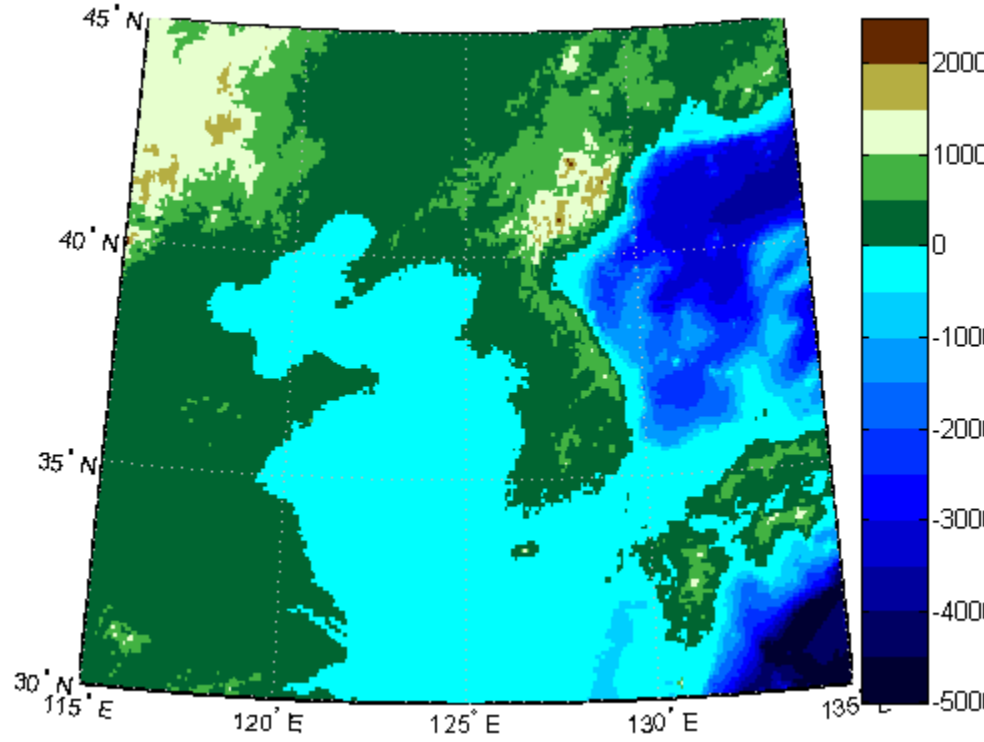
```
demcmap(map)
```



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:

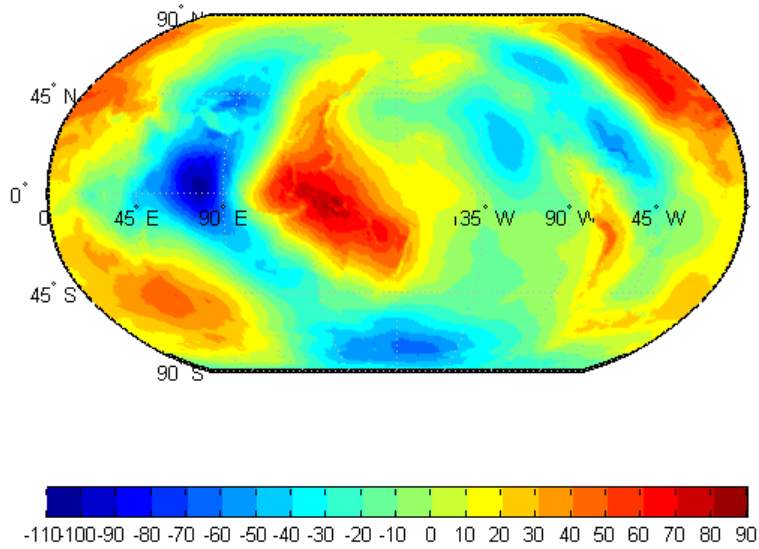
1. Explore `contourcmap` by loading the world `geoid` data set and rendering it with a default colormap:

```
load geoid
figure;
worldmap(geoid,geoidrefvec)
```

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

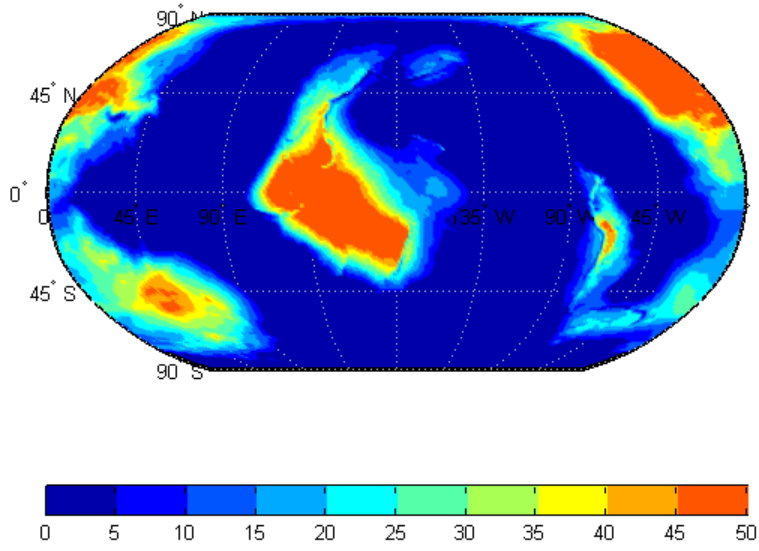
2 Use `contourcmap` to specify a contour interval of 10 (meters), and to place a colorbar beneath the map:

```
contourcmap('jet',10,'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('jet',[0:5:50],...
   '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.

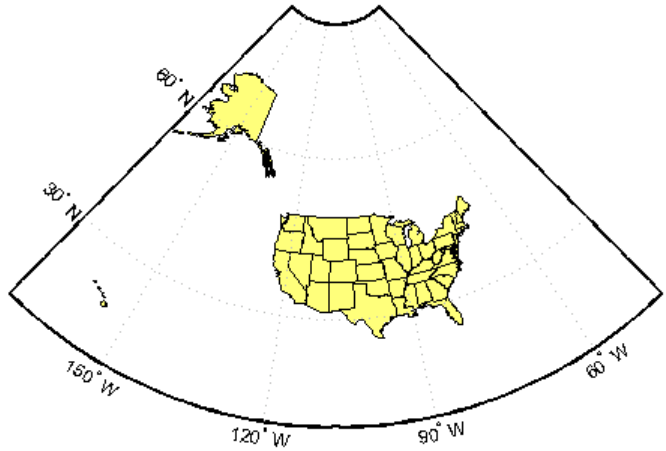
Colormaps for Political Maps

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`:

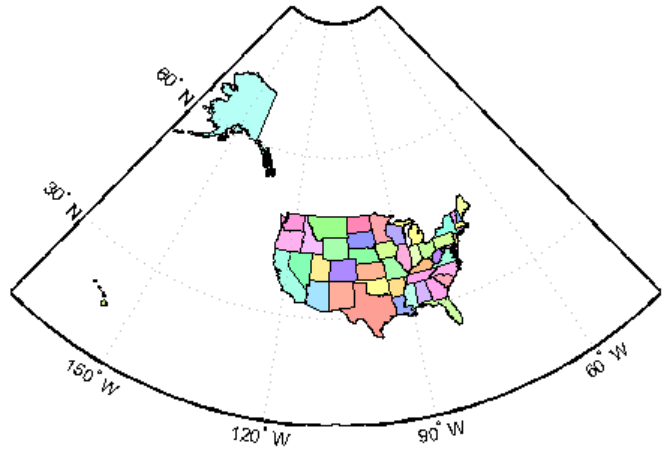
```
figure
worldmap na
states = shaperead('usastatelo', 'UseGeoCoords', true);
geoshow(states)
```

Note that the default face color is black, which is not very interesting.



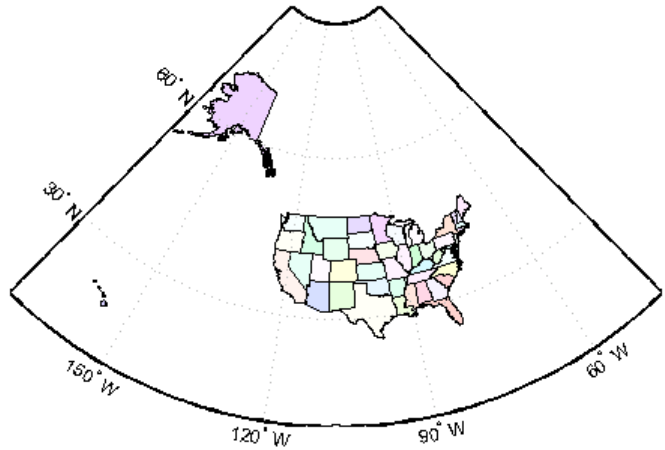
2 Use `polcmap` to populate color definitions to a symbolspec to randomly recolor the patches and expand the map to fill the frame:

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



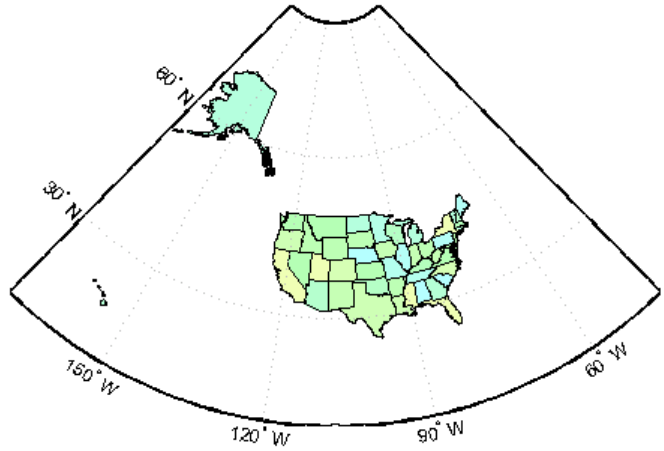
3 The `polcmap` function can also control the number and saturation of colors. Reissue the command specifying 256 colors and a maximum saturation of 0.2. To ensure that the colormap is always the same, reset the seed on the MATLAB random number function using the `'state'` argument with a fixed value 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)
```

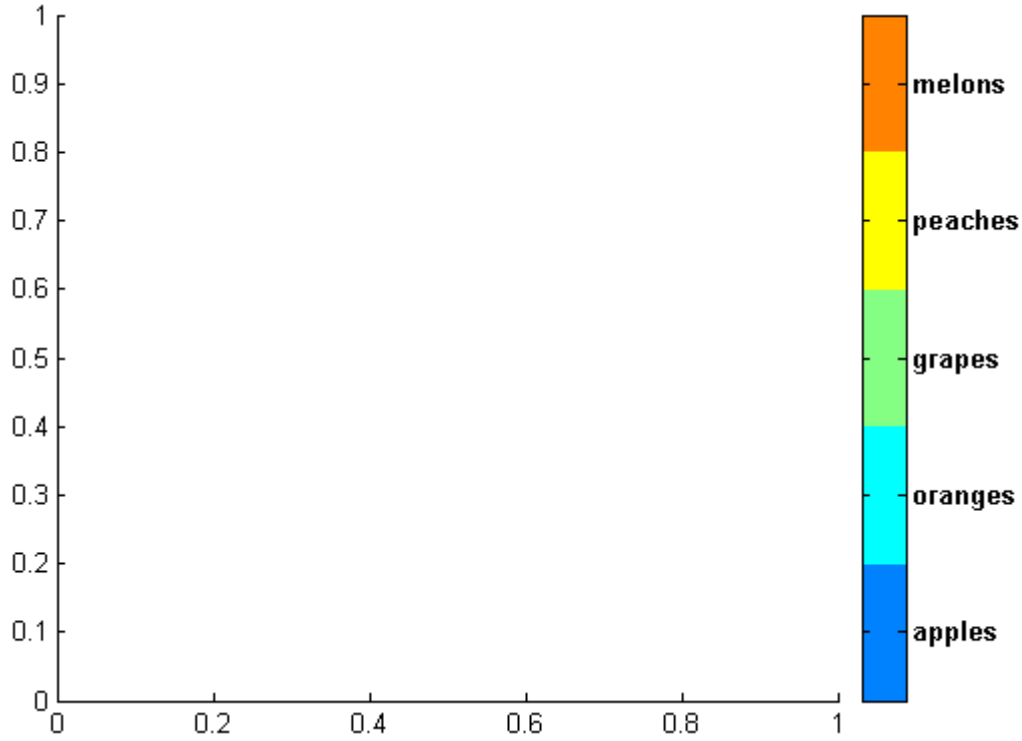


Note The famous Four Color theorem states that any political map can be colored to completely differentiate neighboring patches using only four colors. Experiment to find how many colors it takes to color neighbors differently with `polcmap`.

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.

7

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 `NaN`s, 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
```

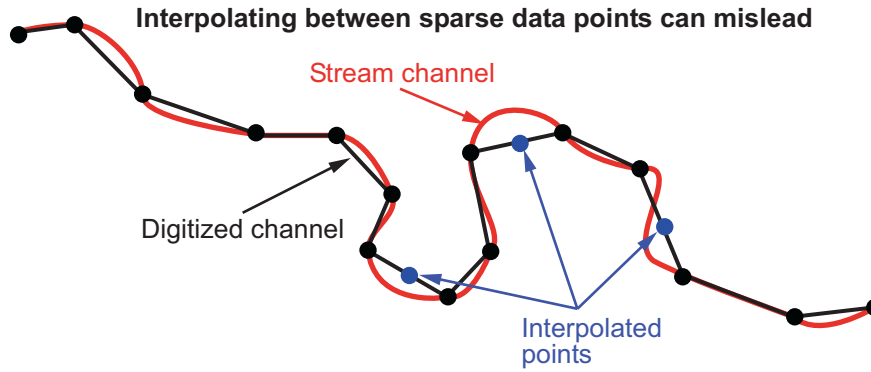
```
    NaN    NaN
```

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 7.3° 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
```

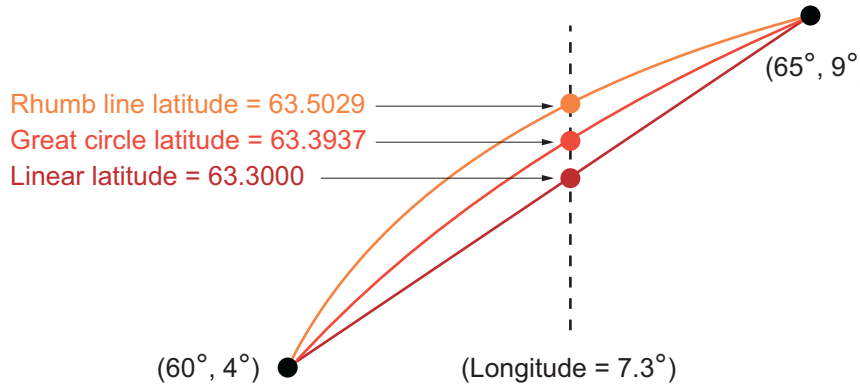
4 Generate the latitude using great circle interpolation:

```
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.



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.

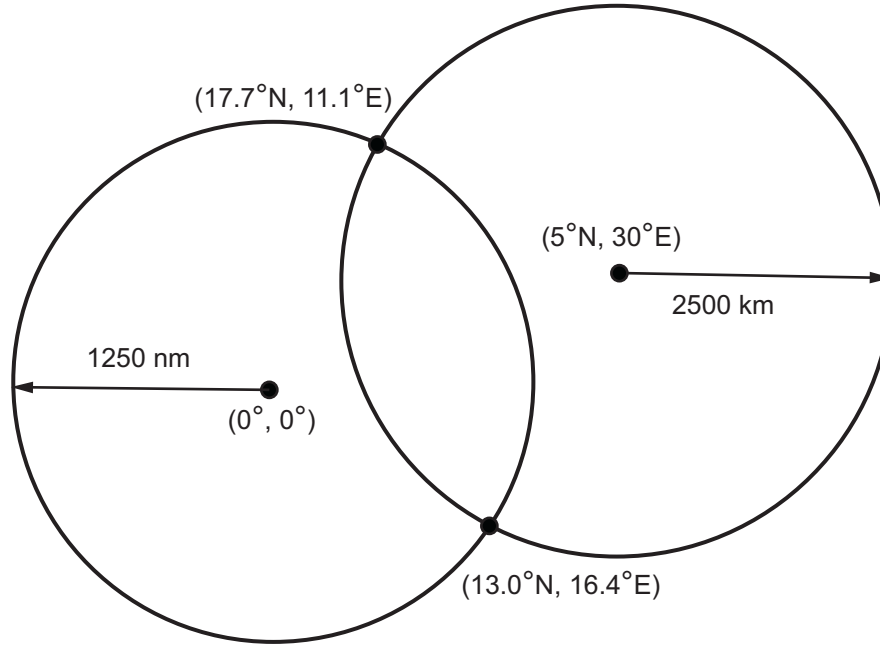
Vector Intersections

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
```



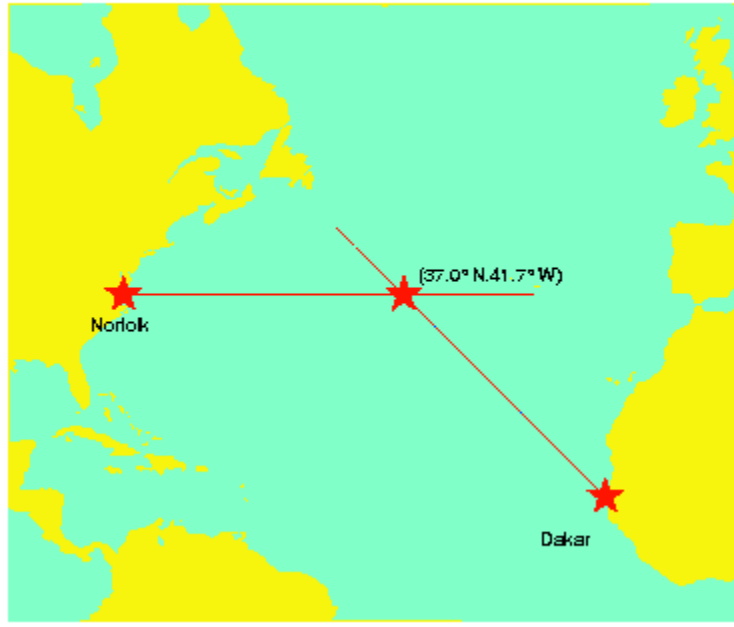
In general, small circles intersect twice or never. For the case of exact tangency, `scxsc` returns two identical intersection points. Other similar commands include `rhxrh` for intersecting rhumb lines, `gcxgc` for intersecting great circles, and `gcxsc` for intersecting a great circle with a small 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
```

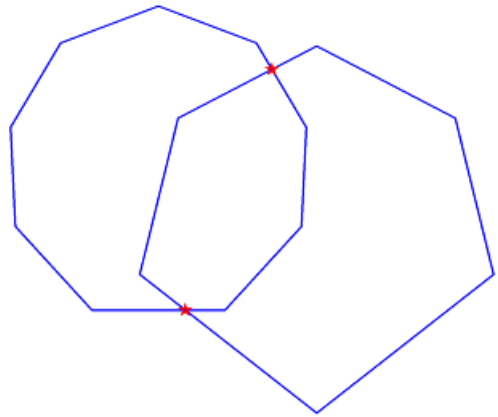
```
long =
 -41.7028
```

The intersection of the tracks is at (37°N,41.7°W), 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 = earthRadius('km');
```

```
area = areaint(uslat,uslon,earthradius)

area =
1.0e+06 *
  7.9256
  0.0035
  0.0004
```

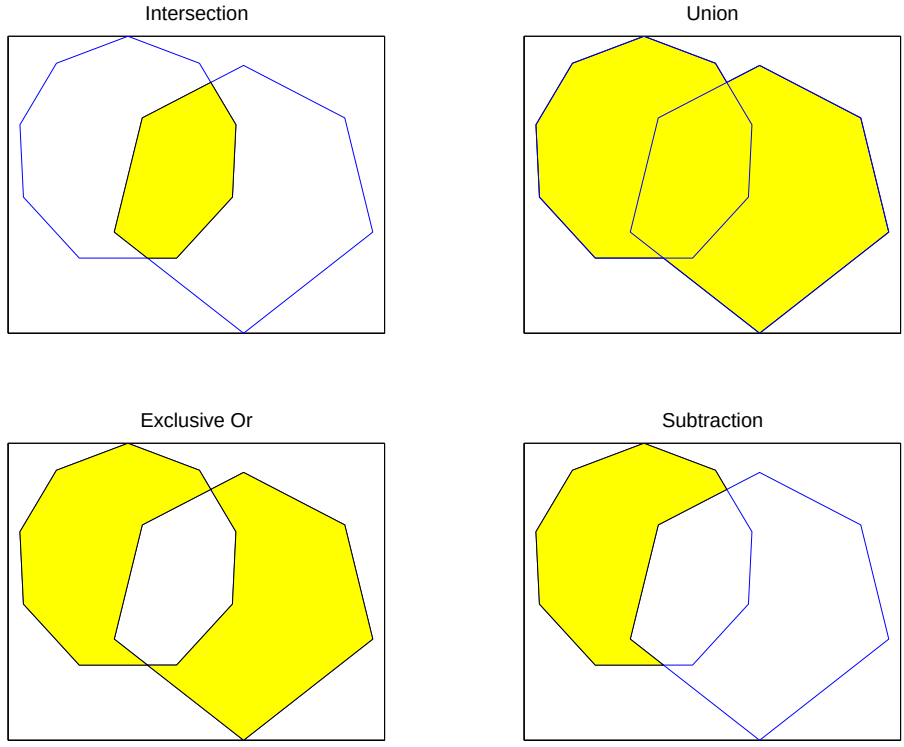
Because the default Earth radius is in kilometers, the area is in square kilometers. From the same variables, the areas of the Great Lakes can be calculated, this time in square miles:

```
load conus
earthradius = earthRadius('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
```

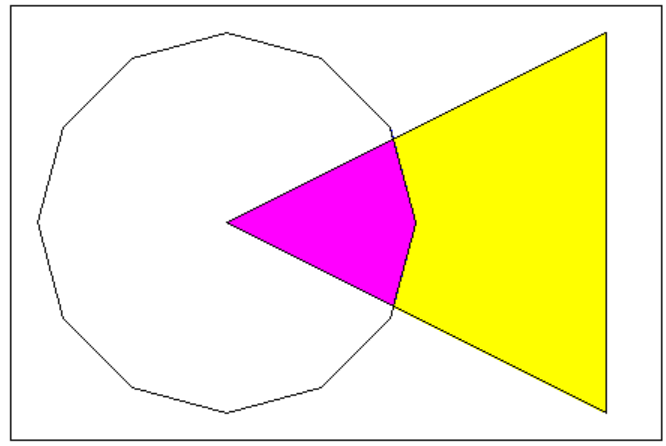
6 Compute the exclusive OR polygon and plot it as a yellow patch:

```
[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 180º connect to points with longitudes near -180º, 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')
```

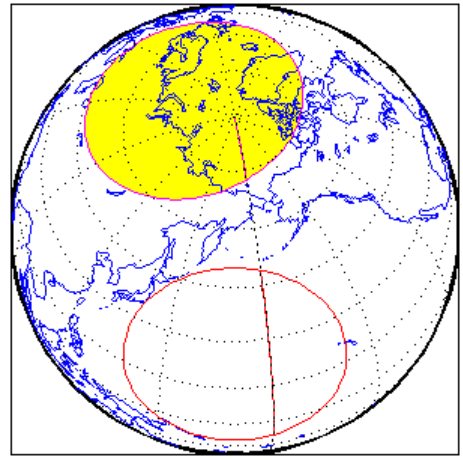
5 Generate a second small circle that does not include a pole:

```
[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.

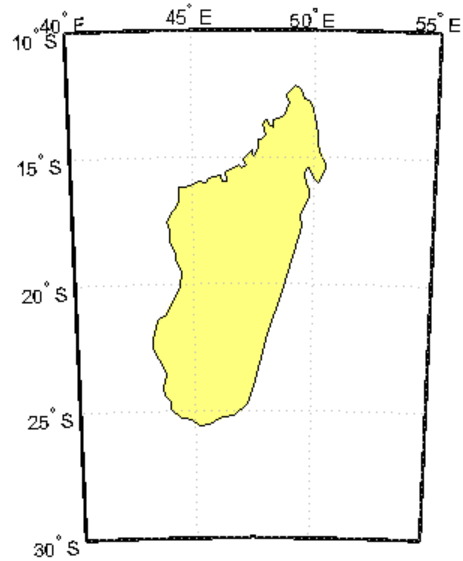
The `bufferm` function computes and returns vectors that represent a set of points that define a buffer zone. It forms the buffer by placing small circles at the vertices of the polygon and rectangles along each of its line segments, and applying a polygon union set operation to these objects.

Generating a Buffer Around a Polygon

Demonstrate `bufferm` using a polygon representing the Island of Madagascar that you extract from the `landareas` data set. The boundary of Madagascar is passed to `bufferm` as latitude and longitude vectors. Using this data, compute a buffer zone at a distance of 0.75 degrees in from the boundaries of Madagascar:

1 Create a base map of the area surrounding Madagascar:

```
ax = worldmap('madagascar');
madagascar = shaperead('landareas',...
    'UseGeoCoords', true,...
    'Selector', {@(name)strcmpi(name,'Madagascar'), 'Name'});
geoshow(ax, madagascar)
```

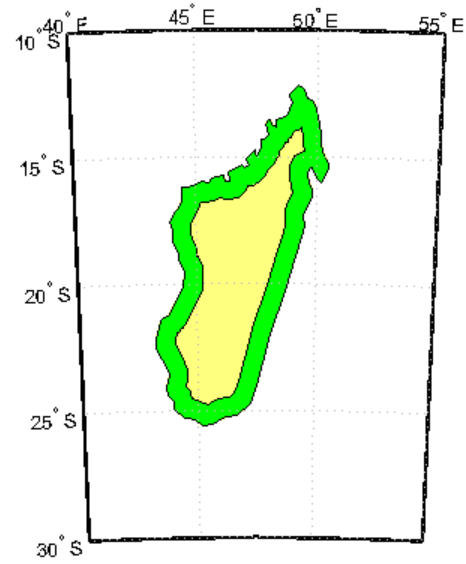


2 Use `bufferm` to process the polygon and output a buffer zone .75 degrees inland:

```
[latb,lonb] = bufferm(madagascar.Lat, madagascar.Lon, .75, 'in');
```

3 Show the buffer zone in green:

```
geoshow(latb, lonb, 'DisplayType', 'polygon', 'FaceColor', 'green')
```



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 7-39.

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:

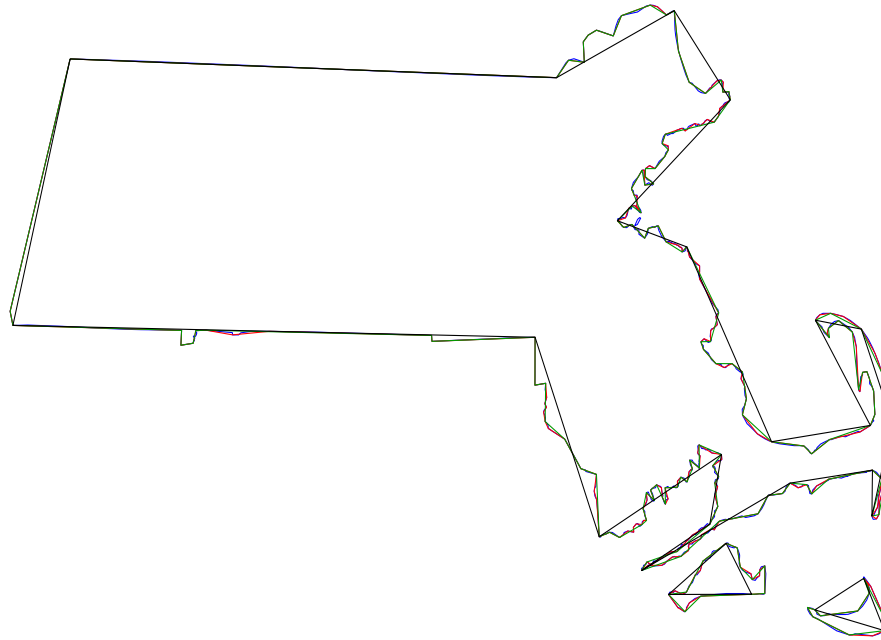
```
numel(maLat2)/numel(maLat)


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:

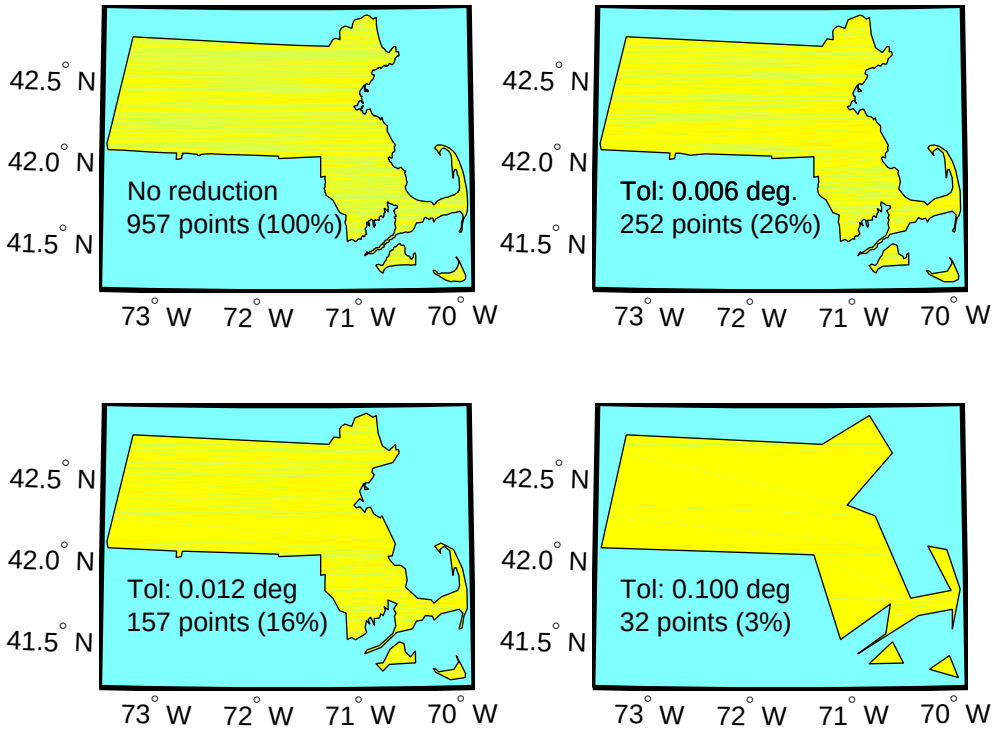
```
[maLat3, maLon3, cerr3, tol3] = reducem(maLat', maLon', 0.1);
geoshow(maLat3, maLon3, 'DisplayType', 'line', 'color', 'black')
```

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 7-39, 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:

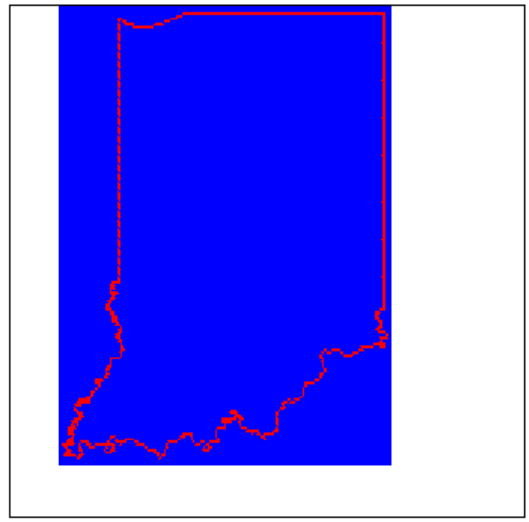
```
gridDensity = 40;
[inGrid, inRefVec] = vec2mtx(inLat, inLon, gridDensity);
whos
```

```
 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.

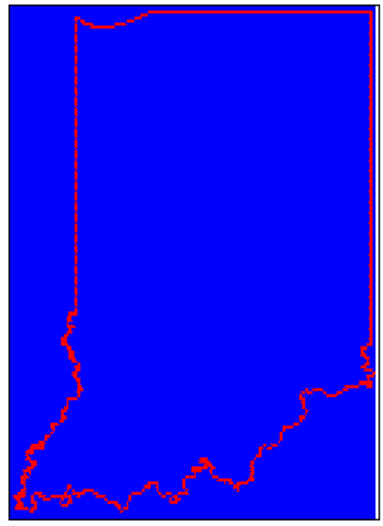
3 Make a map of the data grid in contrasting colors:

```
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
```



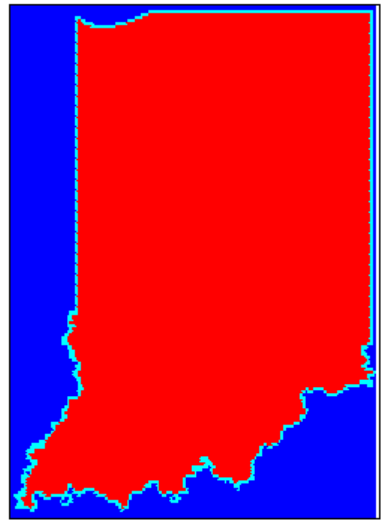
5 To fill (recode) the interior of Indiana, you need a seed point (which must be identified by row and column) and a seed value (to be allocated to all cells within the polygon). Select the middle row and column of the grid and choose an index value of 3 to identify the territory when calling encodem 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:

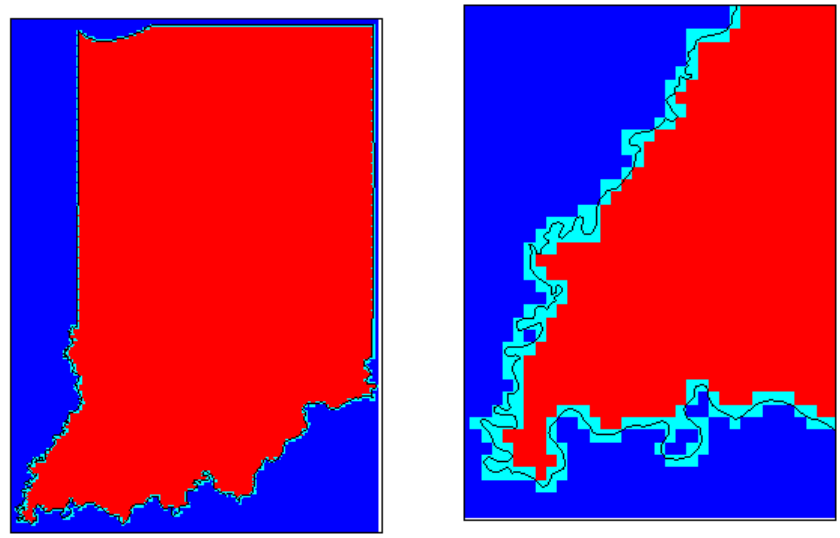
```
meshm(inGrid3, inRefVec)
```



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 7-21 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'});
```

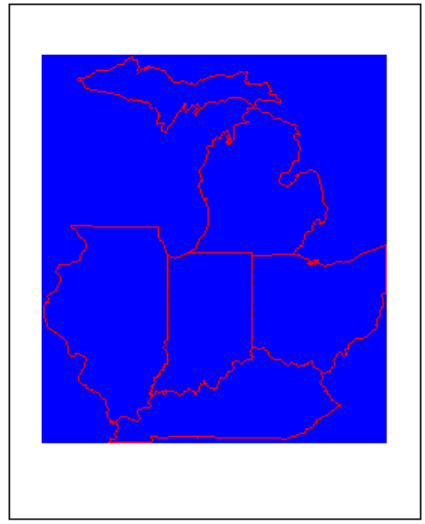
```
meLat = [centralUS.Lat];
meLon = [centralUS.Lon];
```

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.

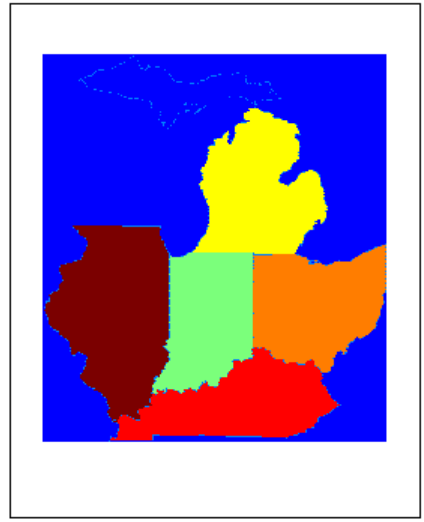
5 Use `encodem` to fill each country with a unique value, producing a new grid:

```
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,earthRadius('km'))


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, earthRadius('km'))


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:

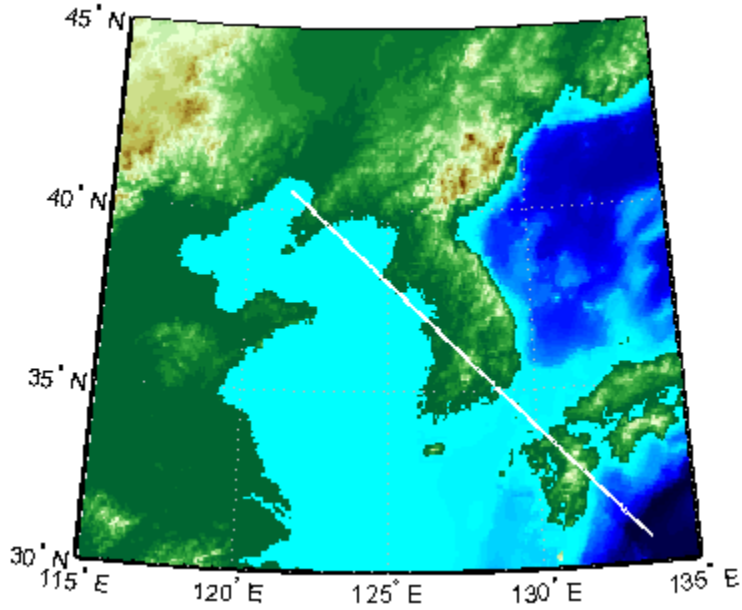
```
plat = [40.5 30.7];
plon = [121.5 133.5];
```

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);
```

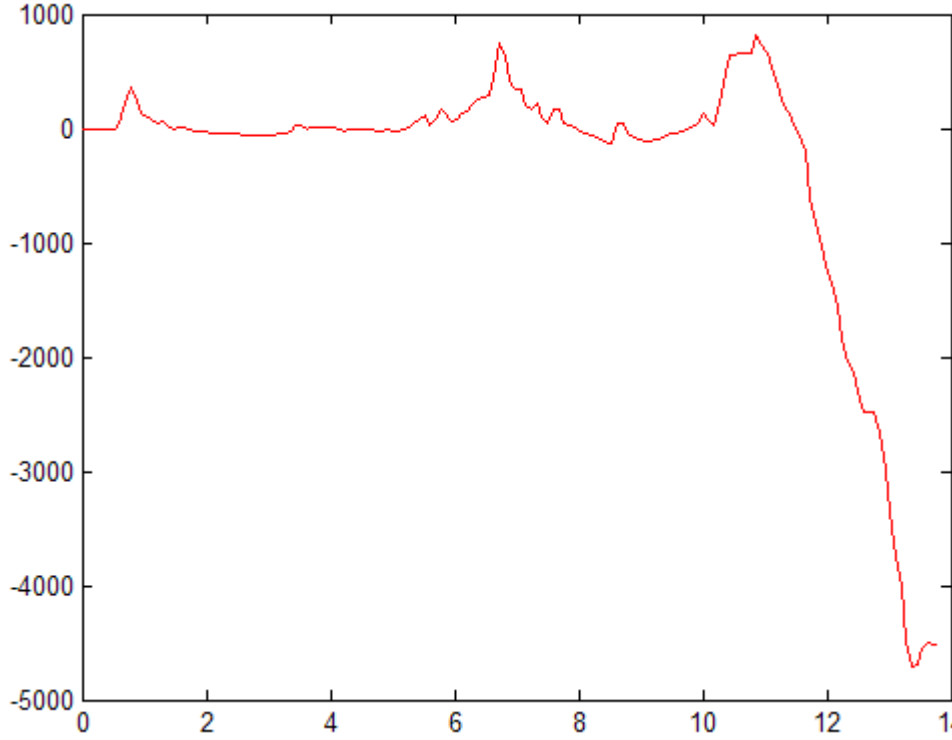
6 Draw the transect in 3-D so it follows the terrain:

```
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')
```

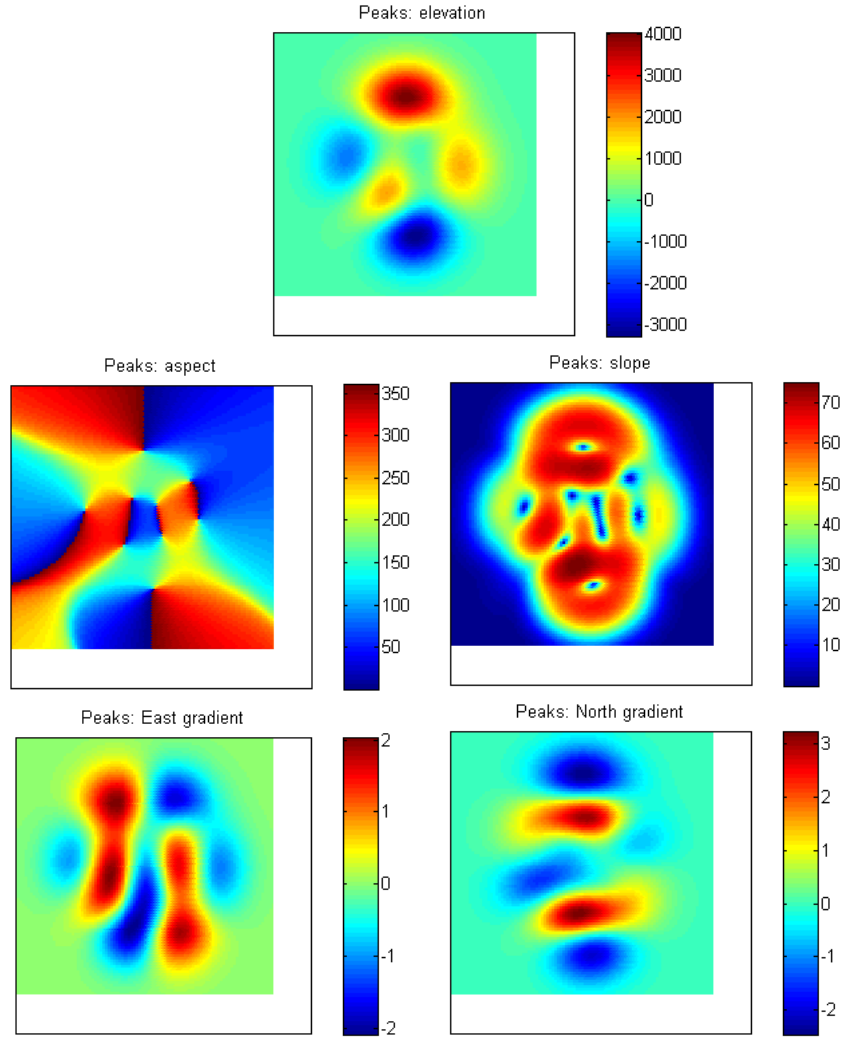
6 Map the gradients to the north:

```
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

8

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 MATLAB 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 11, "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

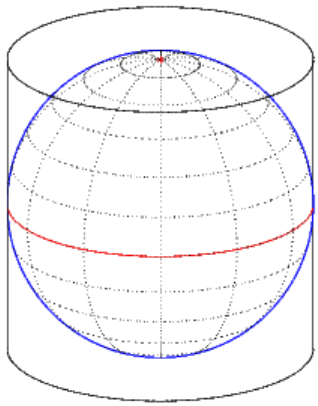
Mapmakers have developed hundreds of map projections, over several thousand years. Three large families of map projection, plus several smaller ones, are generally acknowledged. These are based on the types of geometric shapes that are used to transfer features from a sphere or spheroid to a plane. As 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.

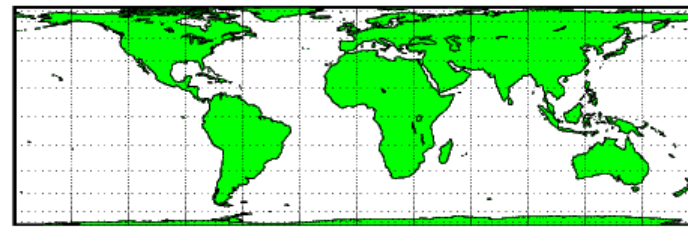
Which developable surface to use for a projection depends on what region is to be mapped, its geographical extent, and the geometric properties that areas, boundaries, and routes need to have, given the purpose of the map. The following sections describe and illustrate how the cylindrical, conic, and 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

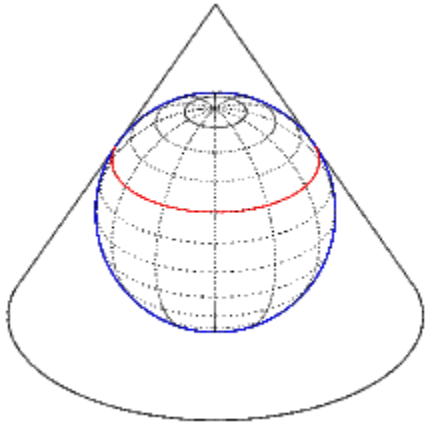
curves, not straight lines. Pseudocylindrical projections can be equal-area, but are not conformal or equidistant.

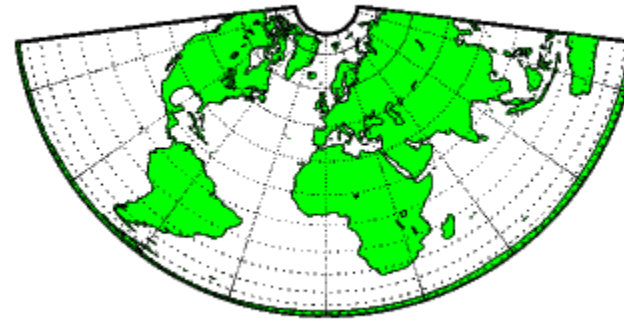
Some widely-used pseudocylindrical map projections are

- 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).



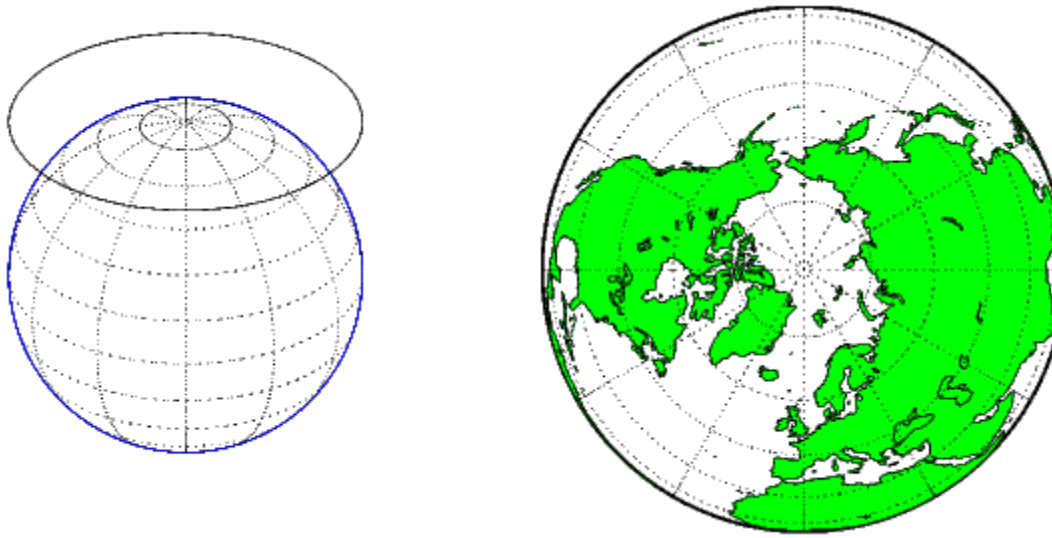


Some widely-used conic projections are

- 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 11, “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.

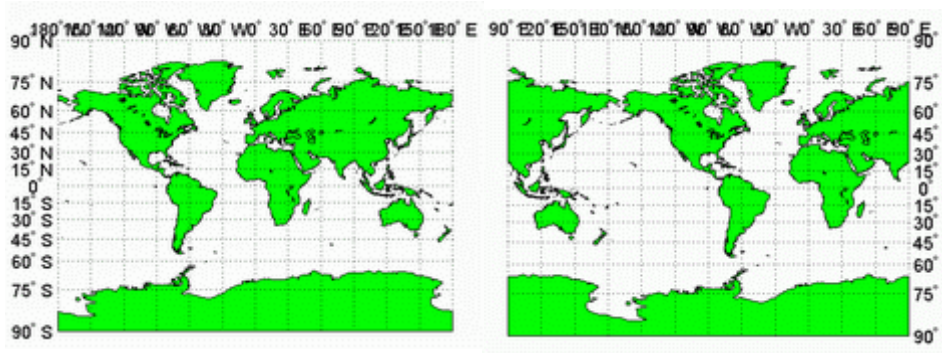
The Orientation Vector

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 (0^o,0^o) 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 0^o.

Both of these Miller projections have normal aspects, despite having different orientation vectors:



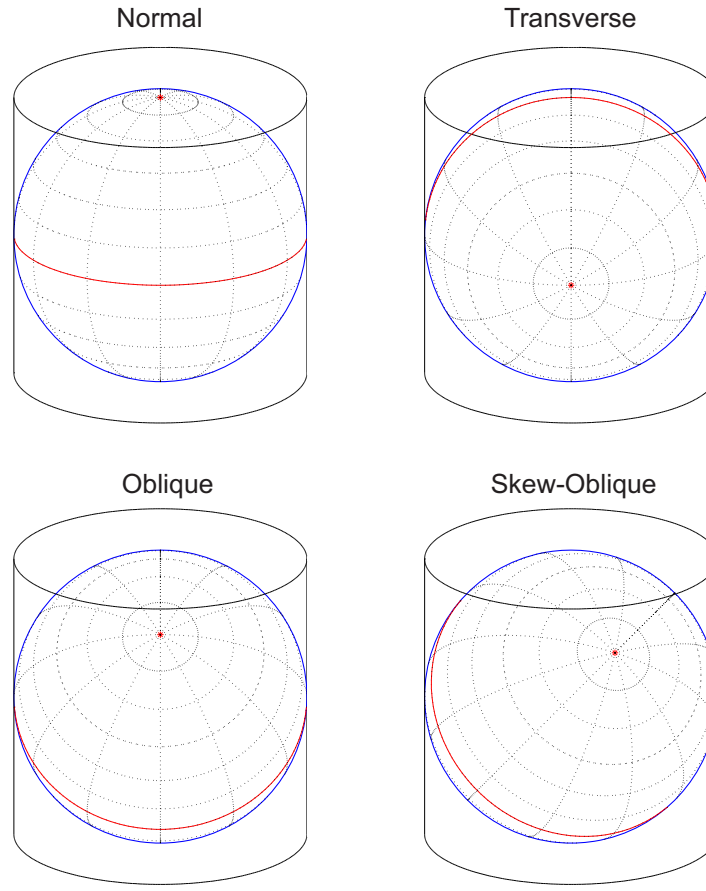
Origin at (0°,0°) with a 0° orientation
(orientation vector = [0 0 0])

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.

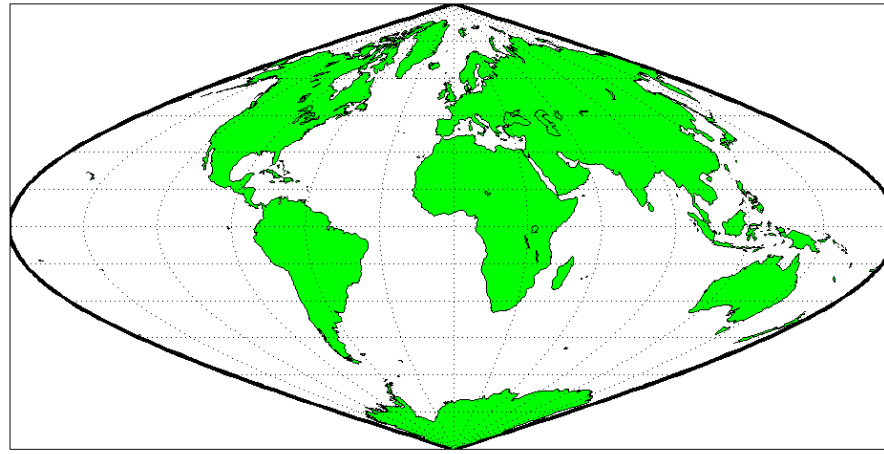
Exploring Projection Aspect

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:

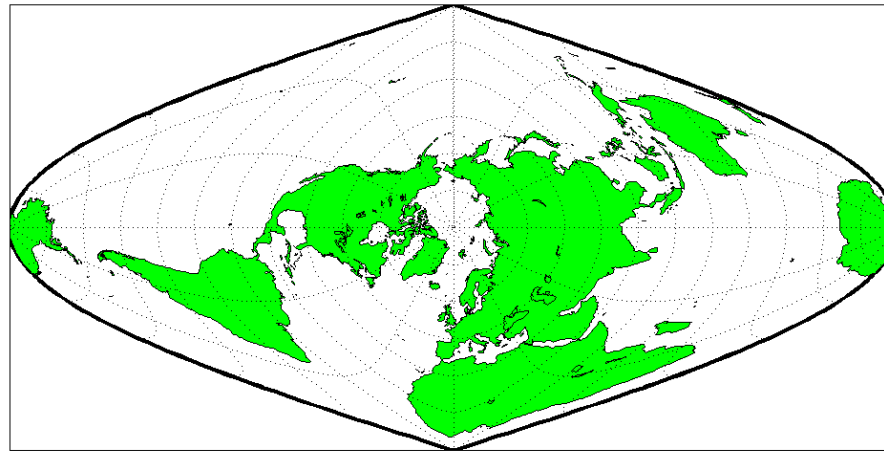
```
getm(gca,'Origin')

ans =
     0     0     0
```

By default, the origin is set at (0ºE, 0ºN), oriented 0º 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 (0º,0º). Do this by setting the first element of `Origin` parameter to a latitude of 90ºN:

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

The shape of the frame is unaffected; this is still a sinusoidal projection.

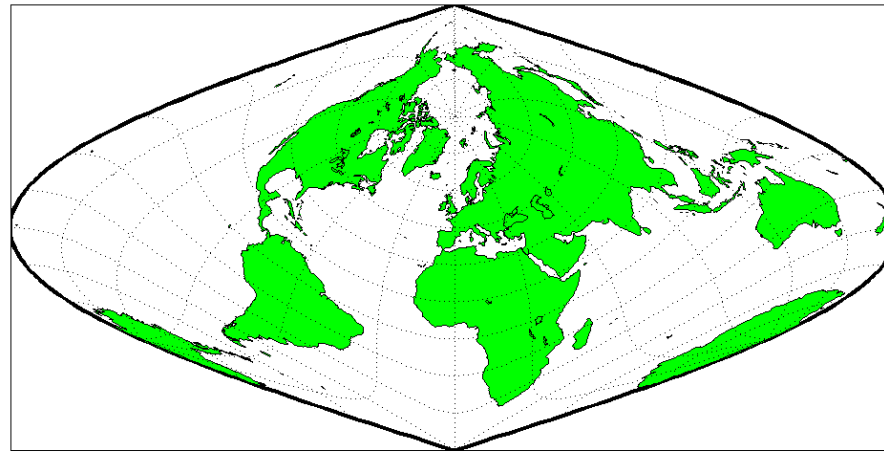


**Transverse aspect: origin at (90°N,0°), orientation 0°
(orientation vector = [90 0 0])**

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.



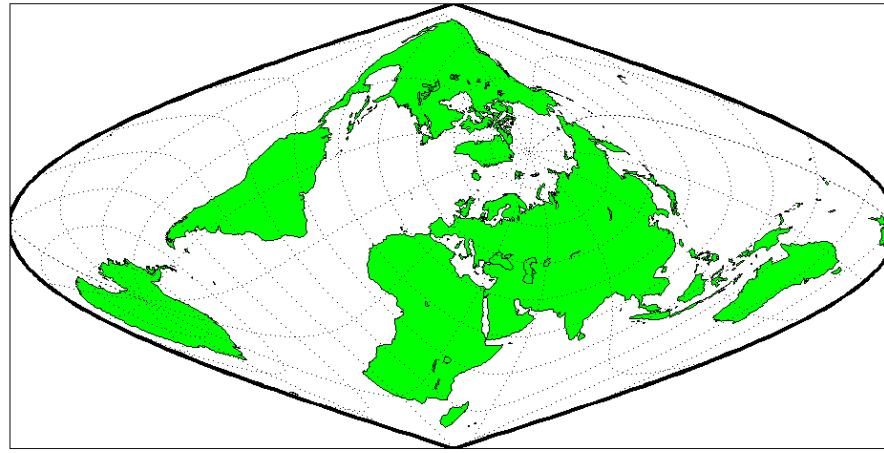
**Oblique aspect: origin at (45°N,0°), orientation 0°
(orientation vector = [45 0 0])**

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.

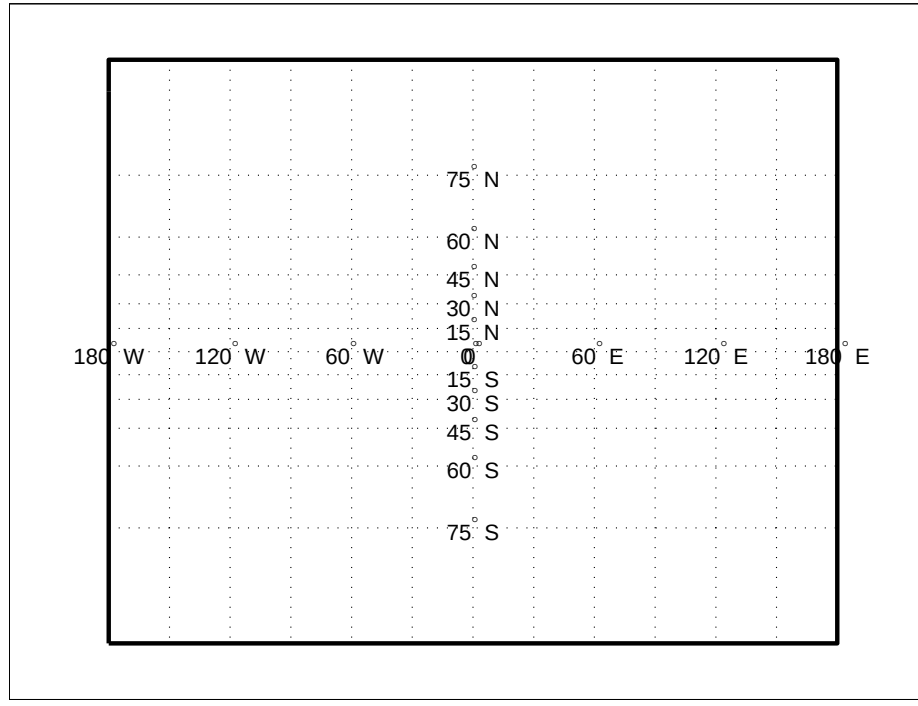
Determining Projection Parameters

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`.

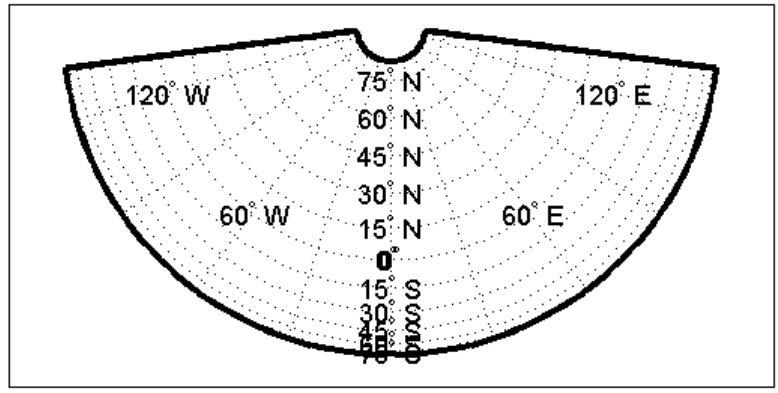
3 Reset the projection type to equal-area conic (`'eqaconic'`). The figure is redrawn to reflect the change. Determine the parameters that the toolbox changes in response:

```
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.



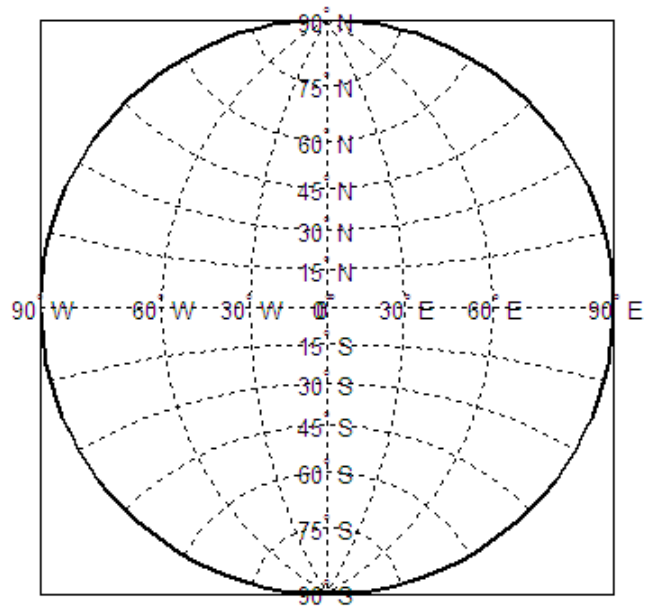
4 Now set the projection type to Stereographic (`'stereo'`) and examine the same properties as you did for the previous projections:

```
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 11, “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:

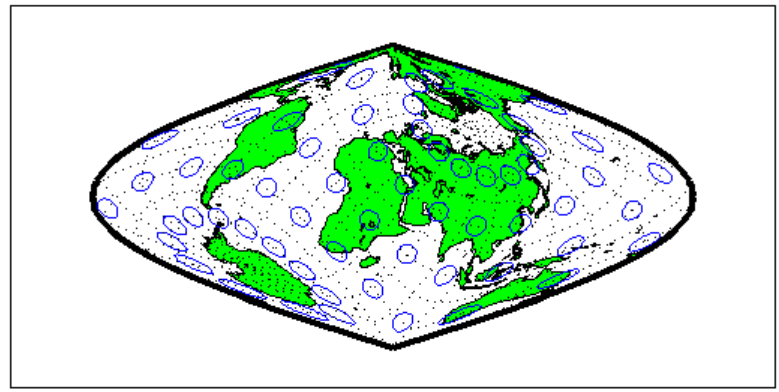
```
figure;
axesm sinusoid
gridm on;framem on;
setm(gca,'Origin', [20 30 45])
```

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
```



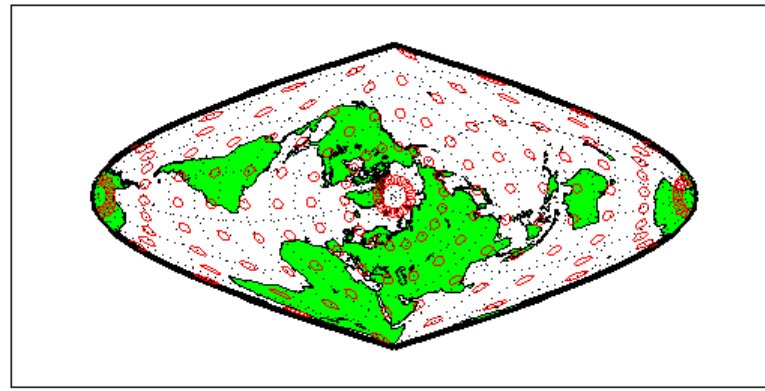
Notice that the circles vary considerably in shape. This indicates that the Sinusoidal projection is not conformal. Despite the distortions, however, the circles all cover equal amounts of area on the map, because the projection has the equal-area property.

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:

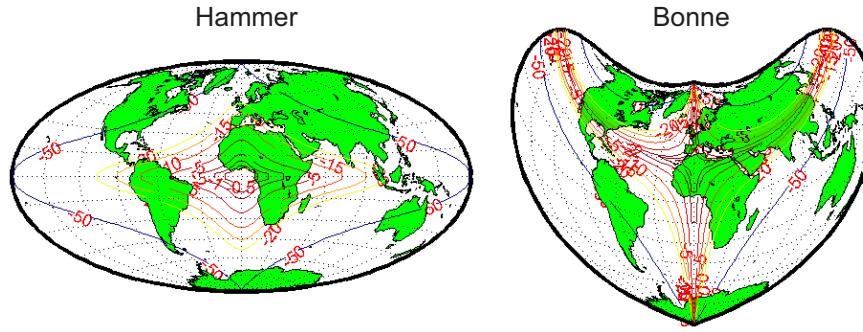
```
mdistort('scaleratio')
```

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'});
```

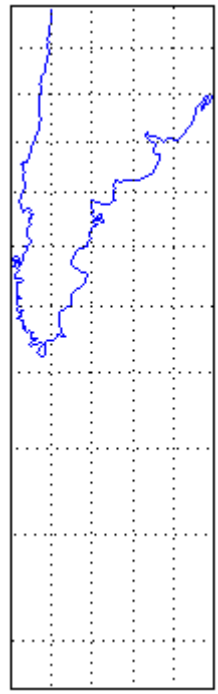
2 Set the spatial extent (map limits) to contain the southern part of South America and also include an area closer to the South Pole:

```
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):

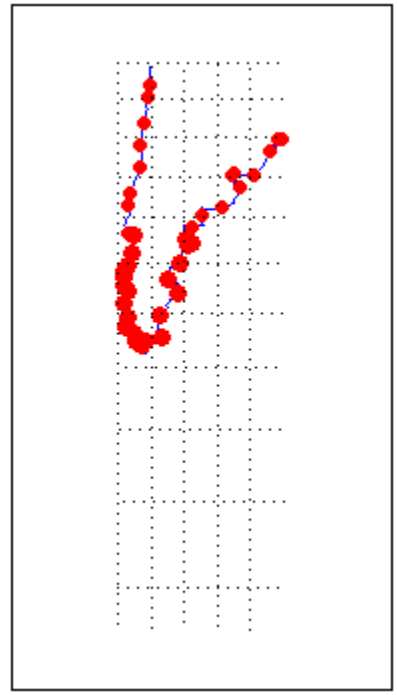
```
adistortmm = [min(adistort) max(adistort)]

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 `NaN`s 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 50ºS, 60ºS, and 70ºS:

```
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
   ```

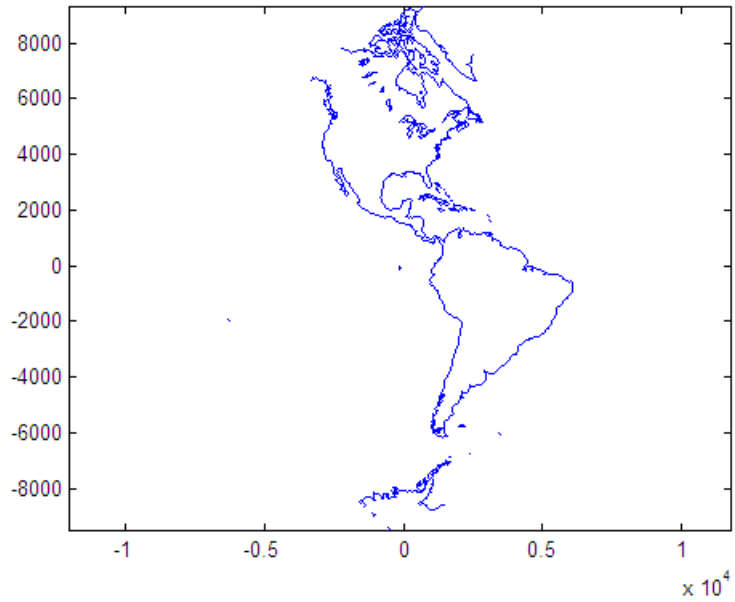
4. Trim the coast to the map limits set above.

   ```
   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
```

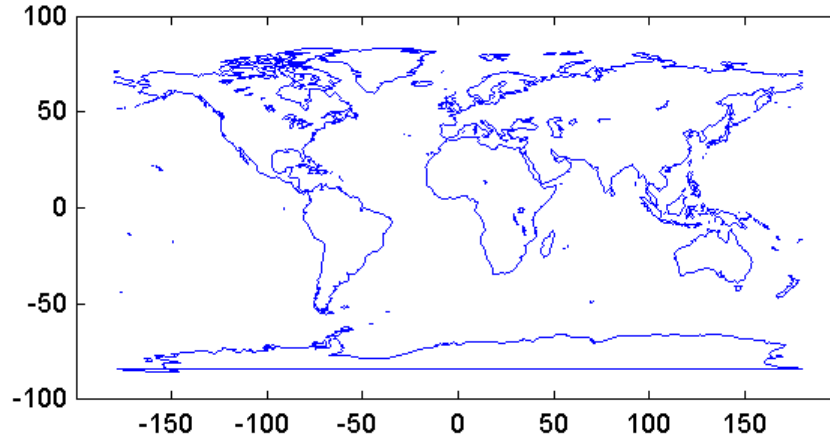
The difference in size between `lat` and `long` and `x` and `y` are due to clipping the *x-y* data to the map frame (`NaN`s are inserted at clip locations).

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 60° from the *x*-axis, to vertical, to 120° from the *x*-axis, quite symmetrically.

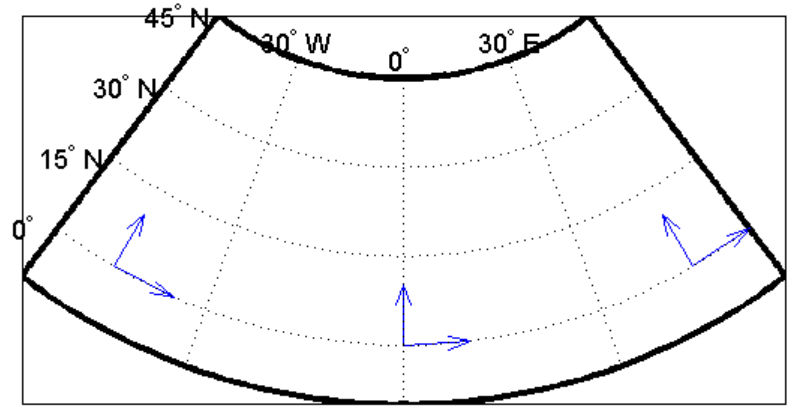
5. Compute projected direction of east for each location:

```
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 49.7258°. 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);
```

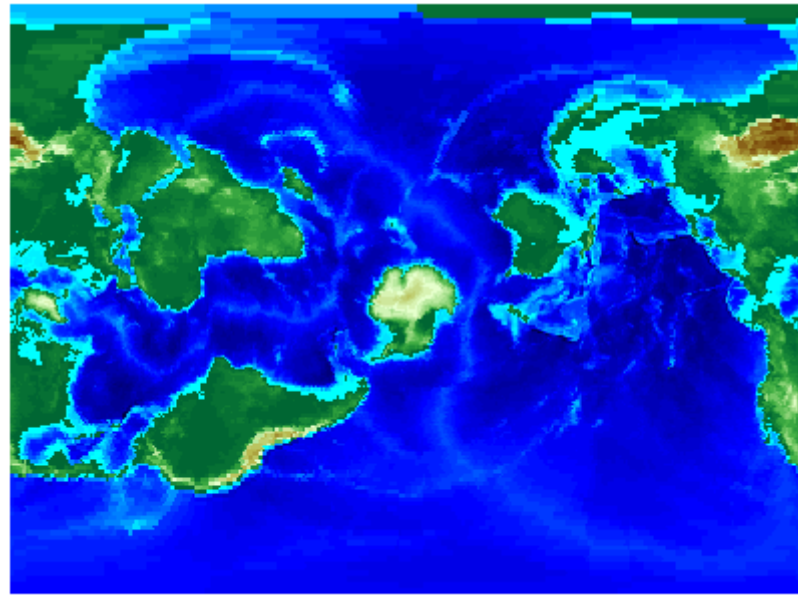
Note that the result, `[Z,lat,lon]`, is a *geolocated data grid*, not a regular data grid like the original `topo` data.

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 3° 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]
                    ...
```

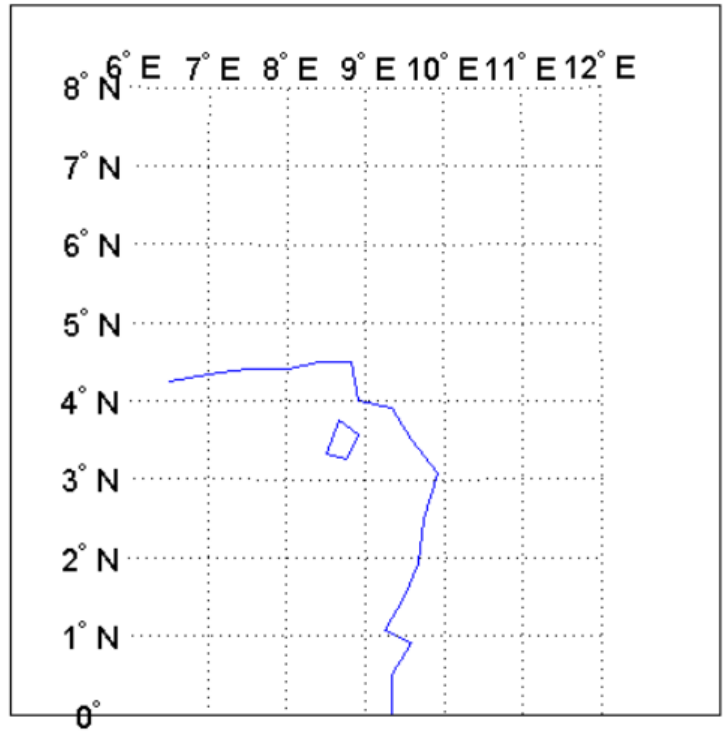
Note that the map origin and limits are adjusted for zone `32N`.

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:

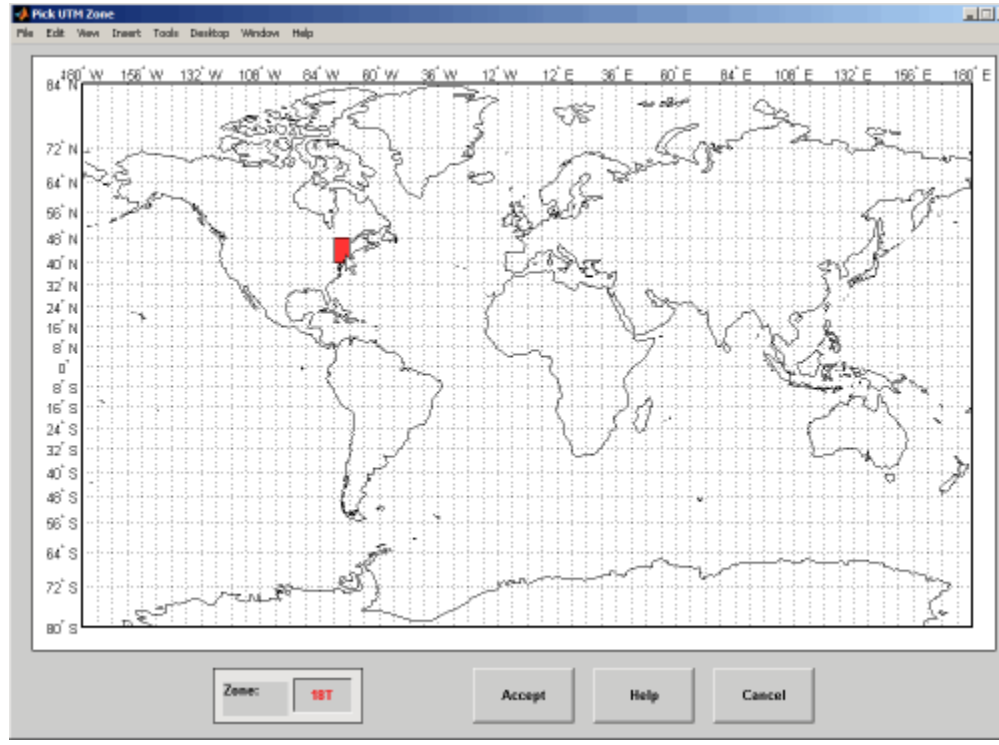
```
figure;
axesm utm
axesmui
```

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:

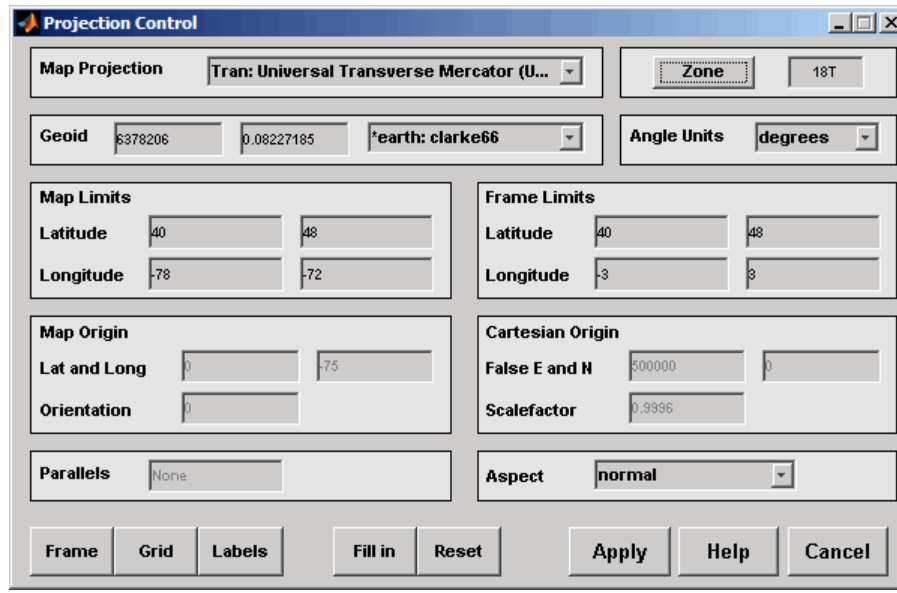


Note that while you can open the `utmzoneui` control panel from the command line, you then have to manually update the figure with the zone name it returns with a `setm` command:

```
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.

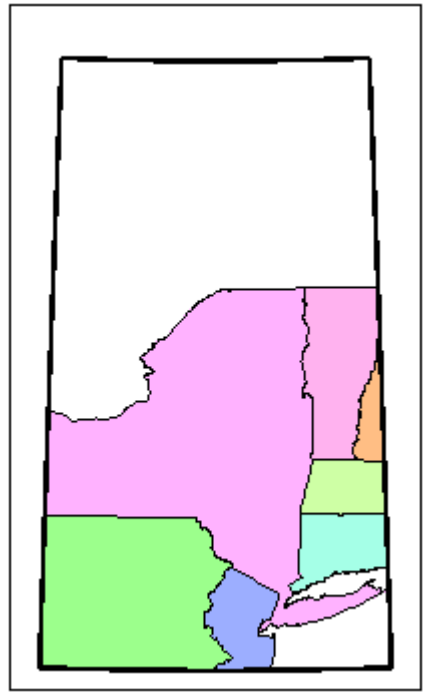


4 Click **Apply** to close the projection control panel.

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

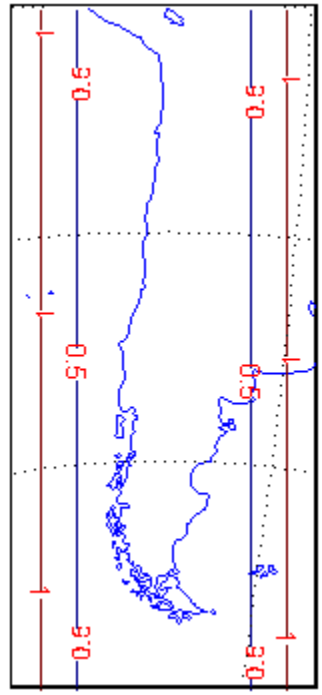
Because UTM is a zone-based coordinate system, it is designed to be used like a map series, selecting from the appropriate sheet. While it is possible to extend one zone's coordinates into a neighboring zone's territory, this is not normally done.

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

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 11, "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	Conformal	Equidistant	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	Conformal	Equidistant	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).

9

Creating Web Map Service Maps

Introduction to Web Map Service

In this section...

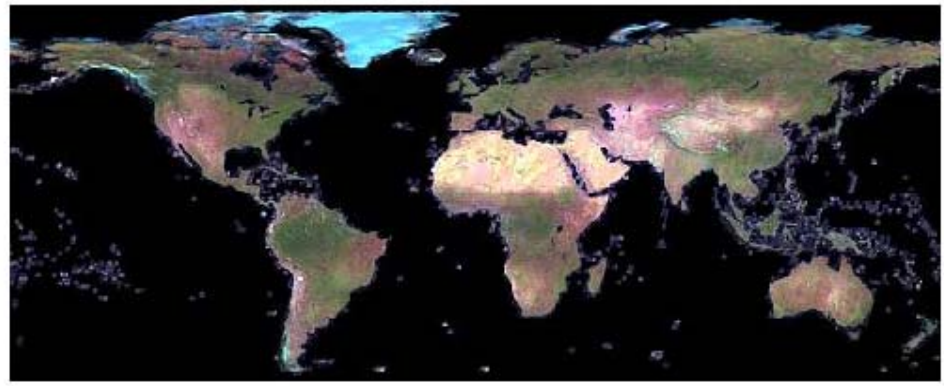
What Web Map Service Servers Provide

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

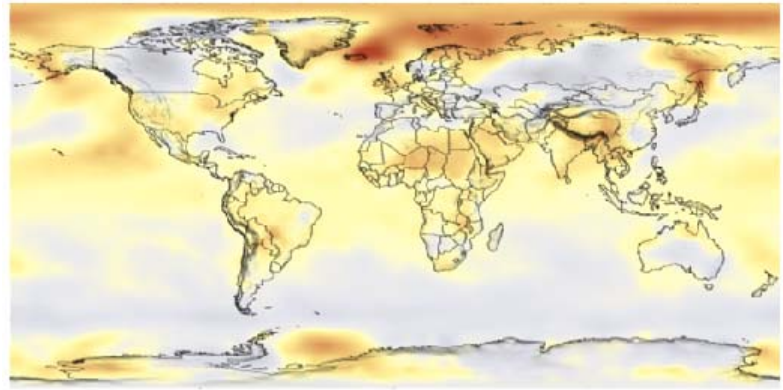
Mapping Toolbox software simplifies the process of WMS map creation by using a stored database of WMS servers. You can search the database for layers and servers that are of interest to you.

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. Search for elevation layers.

   ```
   elevation = wmsfind('elevation');
   ```

   `wmsfind` returns an array of hundreds of `WMSLayer` objects.

2. Refine your search. The MicroImages TNTserver™ hosts a wide variety of layers, including elevation data. Use the `WMSLayer.refine` method to refine your preliminary search results to include only layers on the MicroImages server.

   ```
   microheight = elevation.refine('microimages', ...
      'SearchField', 'serverurl');
   ```

   GTOPO30 is a digital elevation model developed by the United States Geological Survey (USGS). Refine your search to include only the layer with GTOPO30 in the `LayerName` field.

   ```
   gtopolayer = microheight.refine('gtopo30', ...
      '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, modify only the background color. Choose red, green, and blue levels to define an ocean color.

   ```
   oceanColor = [0 170 255];
   ```

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 `gtopolayer` as input for `wmsread`. Set the `wmsread` longitude and latitude limit parameters to the current map axes limits. Set the `BackgroundColor` parameter to `oceanColor`.

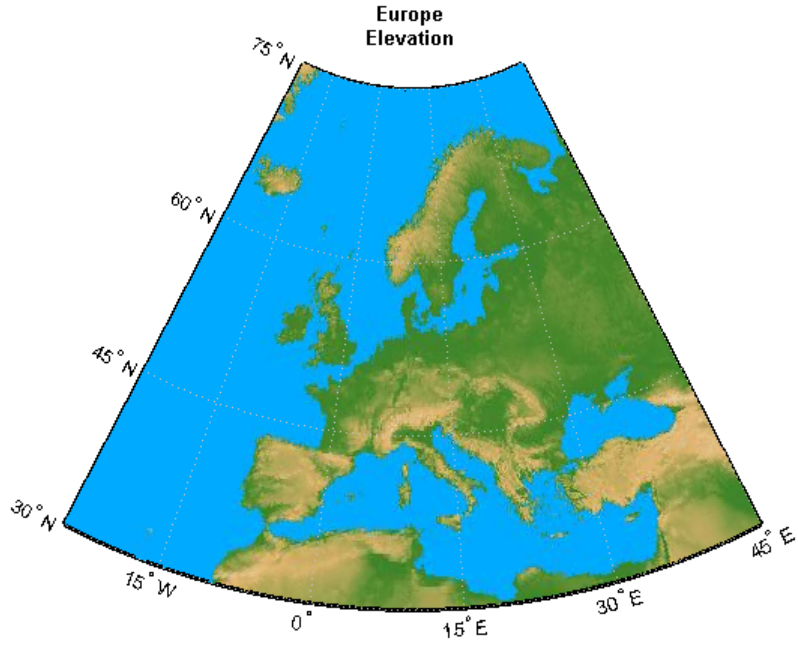
   ```
   [elevationImage, R] = wmsread(gtopolayer, 'Latlim', ...
      mstruct.maplatlimit, 'Lonlim', mstruct.maplonlimit, ...
      'BackgroundColor', oceanColor);
   ```

   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 MicroImages, Inc.

Searching the WMS Database

In this section...

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. MathWorks created the database by conducting a series of Internet searches and qualifying the search results.

Note 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 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 a `WMSLayer class` array, 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 9-78 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.

   ```
   el_nino = global_sst.refine ('El Nino', 'SearchField', ...
      'abstract');
   ```

Type `el_nino(1).Abstract` at the command line to view the abstract of the first layer.

Sample Output:

```
The temperature of the surface of the world's oceans provides
a clear indication of the state of the Earth's climate and
weather....In this visualization of the anomaly covering the
period from June, 2002, to September, 2003, the most obvious
effects are a successive warming and cooling along the equator
to the west of Peru, the signature of an El Nino/La Nina cycle....
```

5. Type `el_nino(1).CoordRefSysCodes` at the command line to view the coordinate reference system codes associated with this layer. In this example, `'EPSG:4326'` is given as the coordinate reference system code. For more information, see "Understanding Coordinate Reference System Codes" on page 9-16 in the *Mapping Toolbox User's Guide*.

6. To view the contents of the `Details` field, type `el_nino(1).Details` at the command line.

Sample Output:

```
ans =

     MetadataURL: 'http://svs.gsfc.nasa.gov/vis/a000000/a002900...
                  /a002906/a002906.fgdc'
      Attributes: [1x1 struct]
     BoundingBox: [1x1 struct]
       Dimension: [1x1 struct]
    ImageFormats: {'image/png'}
     ScaleLimits: [1x1 struct]
           Style: [1x2 struct]
         Version: '1.1.1'
```

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.

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 Next Generation layer shows the Earth's surface for each month of 2004 at high resolution (500 meters/pixel). Read and display the Blue Marble Next Generation 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 `'bluemarbleng'` in the `LayerName` field. This syntax creates an exact search.

```
layer = nasa.refine('bluemarbleng',  'SearchField', 'layername', ...
   'MatchType', 'exact');
```

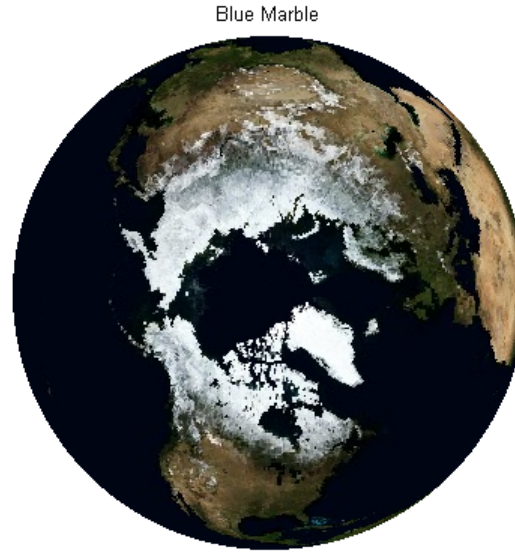
3 Use the `wmsread` function to retrieve the Blue Marble Next Generation layer.

```
[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: Next Generation')
```



Courtesy NASA/JPL-Caltech

Setting Optional Parameters

The `wmsread` function allows you to set many optional parameters, such as 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 MicroImages, Inc. TNTserver™, like NASA, hosts a wide variety of layers. Search the WMS Database for layers that contain the string `'microimages'` in the `ServerURL` field.

   ```
   microLayers = wmsfind('microimages', '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 = microLayers.refine('gtopo30');
   ```

   The refined search returns one layer.

3. Choose red, green, and blue levels to define a background color.

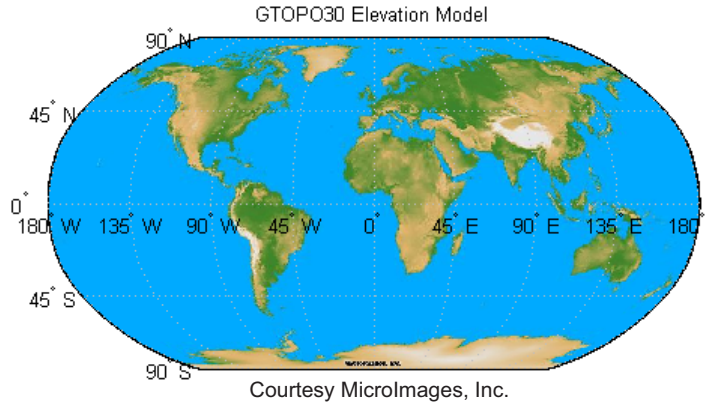
   ```
   oceanColor = [0 170 255];
   ```

4. Use the `BackgroundColor` and `CellSize` parameters of the `wmsread` function to determine the background color and cell size, as you retrieve your map.

   ```
   cellSize = 0.5;
   [A,R] = wmsread(gtopo30Layer, 'BackgroundColor', oceanColor, ...
      'CellSize', cellSize);
   ```

5. Open a figure window and set up a world map axes. Display your map with a title.

   ```
   figure
   worldmap world
   geoshow(A, R)
   title('GTOPO30 Elevation Model')
   ```



Adding a Legend to Your Map

A WMS server renders a layer as an image. Without a corresponding legend, interpreting the pixel colors can be difficult. Some WMS servers provide access to a legend image for a particular layer via a URL that appears in the layer's `Details.Style.LegendURL` field. (See the `WMSLayer.Details` reference page for more information.)

Although a legend provides valuable information to help interpret image pixel colors, only about 45% of the servers in the WMS database contain at least one layer with an available legend. Less than 10% of the layers in the WMS database contain a legend, but nearly 80% of the layers in the database are on the `columbo.nrlssci.navy.mil` server. This server always has empty `LegendURL` fields. You cannot use `wmsfind` to search only for layers with legends because the database does not store this level of detail. You must update a layer from the server before you can access the `LegendURL` field.

This example demonstrates how to create a map of surface temperature, and then obtain and display the associated legend image:

1. Search for layers from the NASA Goddard Space Flight SVS Image Server. This server contains layers that have legend images. You can tell that legend images are available because the layers have content in the `LegendURL` field.

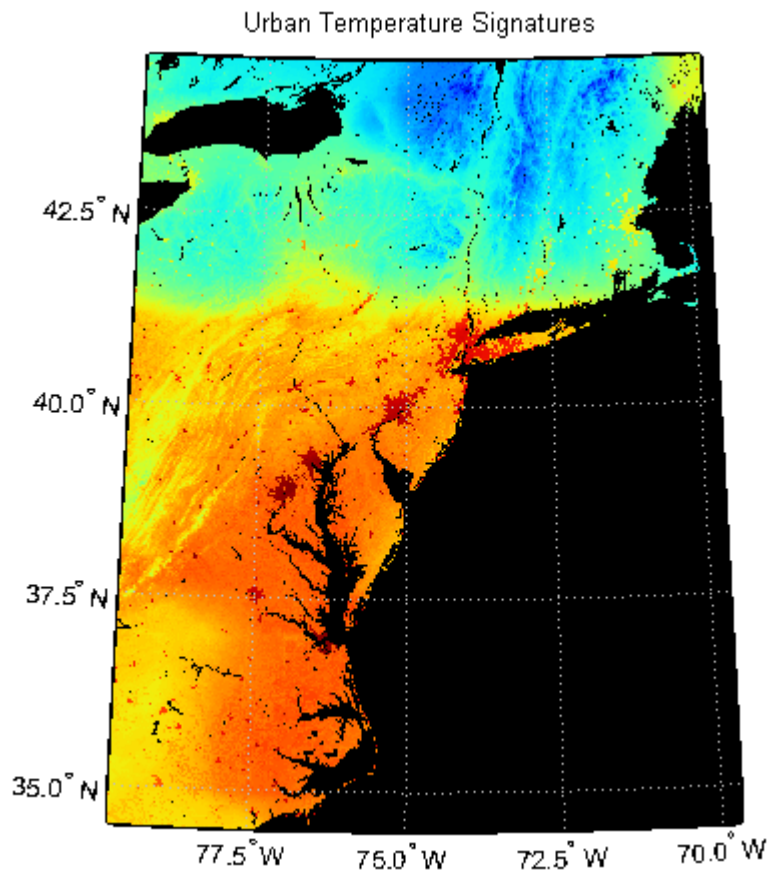
```
layers = wmsfind('gsfc.nasa.gov', 'SearchField', 'serverurl');
serverURL = layers(1).ServerURL;
gsfc = wmsinfo(serverURL);
```

2 Find the layer containing urban temperature signatures and display the abstract:

```
urban_temperature = gsfc.Layer.refine('urban*temperature');
disp(urban_temperature.Abstract)

Big cities influence the environment around them. For example,
urban areas are typically warmer than their surroundings.
Cities are strikingly visible in computer models that simulate
the Earth's land surface. This visualization shows average
surface temperature predicted by the Land Information System (LIS)
for a day in June 2001. Only part of the global computation
is shown, focusing on the highly urbanized northeast corridor
in the United States, including the cities of Boston, New York,
Philadelphia, Baltimore, and Washington.

Additional Credit:
NASA GSFC Land Information System (http://lis.gsfc.nasa.gov/)
```

3 Read and display the layer. The map appears with different colors in different regions, but without a legend it is not clear what these colors represent.

```
[A,R] = wmsread(urban_temperature);
figure
usamap(A,R)
geoshow(A,R)
title('Urban Temperature Signatures')
axis off
```



Courtesy NASA Goddard Space Flight Center

4 Investigate the `Details` field of the `urban_temperature` layer. This layer has only one structure in the `Style` field. The `Style` field determines how the server renders the layer.

```
urban_temperature.Details

ans =

     MetadataURL: [1x65 char]
      Attributes: [1x1 struct]
     BoundingBox: [1x1 struct]
```

```
      Dimension: [1x1 struct]
    ImageFormats: {'image/png'}
     ScaleLimits: [1x1 struct]
           Style: [1x1 struct]
         Version: '1.1.1'
```

Display the `Style` field in the Command Window:

```
urban_temperature.Details.Style

ans =

        Title: 'Opaque'
         Name: 'opaque'
     Abstract: [1x319 char]
    LegendURL: [1x1 struct]
```

Each `Style` element has only one `LegendURL`. Investigate the `LegendURL`:

```
urban_temperature.Details.Style.LegendURL

ans =

    OnlineResource: [1x65 char]
            Format: 'image/png'
            Height: 90
             Width: 320
```

5. Download the legend URL and save it:

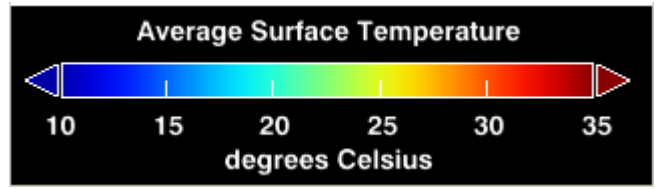
```
url = urban_temperature.Details.Style.LegendURL.OnlineResource
urlwrite(url, 'temp_bar.png');
```

The URL appears in the command window:

```
url =

http://svs.gsfc.nasa.gov/vis/a000000/a003100/a003152/temp_bar.png
```

6. Display the legend image using the `image` command and set properties, such that the image displays with one-to-one, screen-to-pixel resolution.

```
legend = imread('temp_bar.png');
figure('Color','white')
axis off image
set(gca,'units','pixels','position',...
   [0 0 size(legend,2) size(legend,1)]);
pos = get(gcf,'position');
set(gcf,'position',...
   [pos(1) pos(2) size(legend,2) size(legend,1)]);
image(legend)
```



Courtesy NASA Goddard Space Flight Center

Now the map makes more sense. The regions toward the red end of the spectrum are warmer.

Steps 7–10 demonstrate how to capture the output from a map frame and append the legend.

7 By appending the legend in this fashion, you avoid warping text in the legend image. (Legend text warps if you display the image with `geoshow`.)

First set your latitude and longitude limits to match the limits of your map and read in a shapefile with world city data:

```
[latlim,lonlim] = limitm(A,R);
S = shaperead('worldcities', 'UseGeoCoords',true,...
    'BoundingBox',[lonlim(1) latlim(1);lonlim(2) latlim(2)]);
```

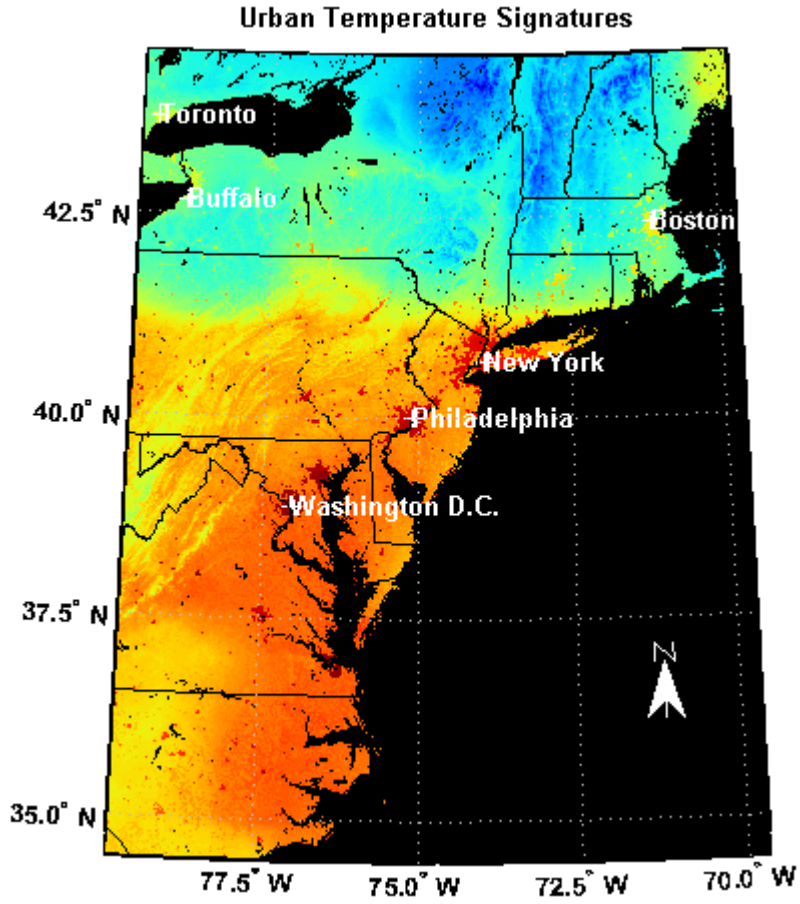
8 Determine the position of the current figure window. Vary the `pos(1)` and `pos(2)` `'Position'` parameters as necessary based on the resolution of your screen.

```
colValue = [1 1 1];
dimension = size(A,1)/2;
figure
```

```
set(gcf,'Color',[1,1,1])
pos = get(gcf, 'Position');
set(gcf, 'Position', [pos(1) pos(2) dimension dimension])
```

9 Display the map and add city markers, state boundaries, meridian and parallel labels, a title, and a North arrow:

```
usamap(A,R)
geoshow(A,R)
geoshow(S, 'MarkerEdgeColor', colValue, 'Color', colValue)

geoshow('usastatehi.shp', 'FaceColor', 'none',...
   'EdgeColor','black')
mlabel('FontWeight','bold')
plabel('FontWeight','bold')
axis off
title('Urban Temperature Signatures', 'FontWeight', 'bold')

for k=1:numel(S)
   textm(S(k).Lat, S(k).Lon, S(k).Name, 'Color', colValue,...
       'FontWeight','bold')
end

lat = 36.249;
lon = -71.173;
northarrow('Facecolor', colValue, 'EdgeColor', colValue,...
    'Latitude', lat, 'Longitude', lon);
```

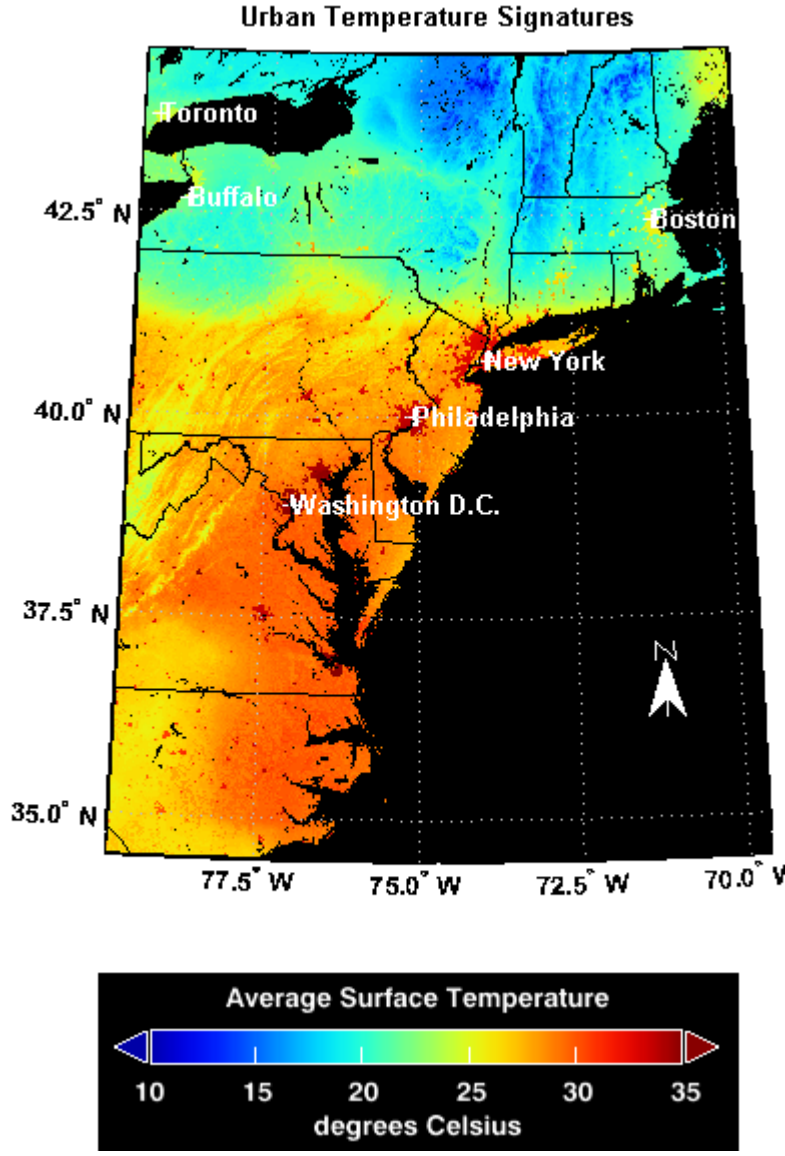


Courtesy NASA Goddard Space Flight Center

10 Display the map and legend as a single, combined image:

```
f = getframe(gcf);
legendImg = uint8(255*ones(size(legend,1), size(f.cdata,2), 3));
offset = dimension/2;
halfSize = size(legend, 2)/2;
legendImg(:,offset-halfSize:offset+halfSize-1,:) = legend;
combined = [f.cdata; legendImg];
figure
```

```
pos = get(gcf,'position');
set(gcf,'position',[10 100 size(combined,2) size(combined,1)])
set(gca,'units','normalized','position', ...
   [0 0 1 1]);
image(combined)
axis off image
```



Courtesy NASA Goddard Space Flight Center

11 Another way to display the map and legend together is to burn the legend into the map at a specified location. To view the image, use the `image` command, setting the position parameters such that there is a one-to-one pixel-to-screen resolution. (Legend text warps if the image is displayed with `geoshow`.)

```
A_legend = A;
A_legend(end-size(legend,1):end-1,...
   end-size(legend,2):end-1,:) = legend;
figure
image(A_legend)
axis off image
set(gca,'Units','normalized','position',...
   [0 0 1 1]);
set(gcf,'Position',[10 100 size(A_legend,2) size(A_legend,1)]);
title('Urban Temperature Signatures', 'FontWeight', 'bold')
```

12 Combine the map and legend in one file, and then publish it to the Web. First write the images to a file:

```
mkdir('html')
imwrite(A_legend, 'html/wms_legend.png')
imwrite(combined, 'html/combined.png')
```

Open the MATLAB Editor, and paste in this code:

```
%%
% <<wms_legend.png>>

%%
% <<combined.png>>
```

Add any other text you want to include in your published document. Then select one of the cells and choose **File > Save File and Publish** from the menu.

Retrieving Your Map with WebMapServer.getMap

The `WebMapServer.getMap` method allows you to retrieve maps in any properly defined EPSG coordinate reference system. If you want to retrieve a map in the EPSG:4326 reference system, you can use `wmsread`. If you want

to retrieve a layer whose coordinates are not in the EPSG:4326 reference system, however, you must use the `WMSMapRequest` class to construct the request URL and the `WebMapServer.getMap` method to retrieve the map. This example demonstrates how to create maps in UTM coordinates using the `WMSMapRequest` and `WebMapServer` classes.

The Microsoft TerraServer provides ortho-imagery and topography maps from various regions of the United States. The server provides the data in both EPSG:4326 and UTM coordinates, as defined by EPSG codes 26905–26920 (representing zones 5–20). For more information about these codes, see the Spatial Reference Web site.

1 Obtain geographic coordinates that are coincidental with the image in the file `boston.tif`.

```
proj = geotiffinfo('boston.tif');
cornerLat = [proj.CornerCoords.Lat];
cornerLon = [proj.CornerCoords.Lon];
latlim = [min(cornerLat) max(cornerLat)];
lonlim = [min(cornerLon) max(cornerLon)];
```

2 Convert the geographic limits to UTM.

```
mstruct = defaultm('utm');
mstruct.zone = '19N';
mstruct.maplatlimit = latlim;
mstruct.maplonlimit = lonlim;
mstruct.geoid = almanac('earth','grs80','m');
mstruct = defaultm(mstruct);
[x y] = mfwdtran(mstruct, latlim, lonlim);
xlimits = [min(x) max(x)];
ylimits = [min(y) max(y)];
```

3 Calculate image height and width values for a sample size of 5 meters.

```
metersPerSample = 5;
imageHeight = round(diff(ylimits)/metersPerSample);
imageWidth  = round(diff(xlimits)/metersPerSample);
```

4 Re-compute the new limits.

```
yLim = [ylimits(1), ylimits(1) + imageHeight*metersPerSample];
xLim = [xlimits(1), xlimits(1) + imageWidth*metersPerSample];
```

5 Find the UTM zone.

```
zone = utmzone(latlim, lonlim)
```

Your output appears as follows:

```
zone =

19T
```

The data lies in zone 19, which corresponds to the EPSG:26919 code.

```
code = 'EPSG:26919';
```

6 Find the TerraServer from the WMS database and select the Digital Ortho-Quadrangle layer.

```
terraserver = wmsfind('terraservice.net','search','serverurl');
doqLayer = terraserver.refine('DOQ');
```

7 Create `WebMapServer` and `WMSMapRequest` objects.

```
server  = WebMapServer(terraserver(1).ServerURL);
request = WMSMapRequest(doqLayer, server);
```

8 Use `WMSMapRequest` properties to modify different aspects of your map request, such as map limits, image size, and coordinate reference system code. Set the map limits to cover the same region as found in the `boston.tif` file.

```
request.CoordRefSysCode = code;
request.ServerURL = 'http://terraservice.net/ogcmap6.ashx?';
request.ImageHeight = imageHeight;
request.ImageWidth  = imageWidth;
request.XLim = xLim;
request.YLim = yLim;
```

9 Request a map of the ortho-imagery in UTM coordinates.

```
A_UTM = server.getMap(request.RequestURL);
R_UTM = request.RasterRef;
```

10 Obtain a map for the same region, but in EPSG:4326 coordinates.

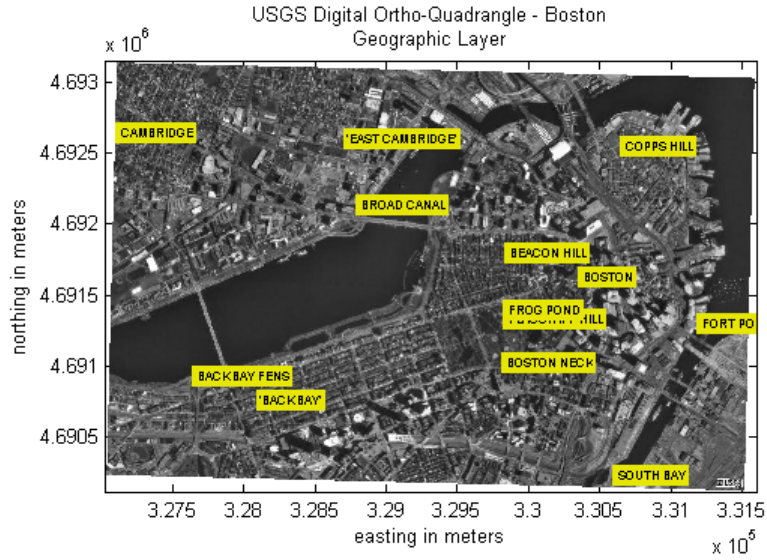
```
[latlim, lonlim] = minvtran(mstruct, xLim, yLim);
request.CoordRefSysCode = 'EPSG:4326';
request.Latlim = latlim;
request.Lonlim = lonlim;
A_Geo = server.getMap(request.RequestURL);
R_Geo = request.RasterRef;
```

11 Read in Boston place names from a shapefile and overlay them on top of the maps. Convert the coordinates of the features to UTM and geographic coordinates. The point coordinates in the shapefile are in meters and Massachusetts State Plane coordinates, but the GeoTIFF projection is defined in survey feet.

```
S = shaperead('boston_placenames');
x = [S.X]*unitsratio('sf','meter');
y = [S.Y]*unitsratio('sf','meter');
names = {S.NAME};
[lat, lon] = projinv(proj, x, y);
[xUTM, yUTM] = mfwdtran(mstruct, lat, lon);
```

12 Project and display the ortho-imagery obtained in EPSG:4326 coordinates into UTM coordinates using geoshow.

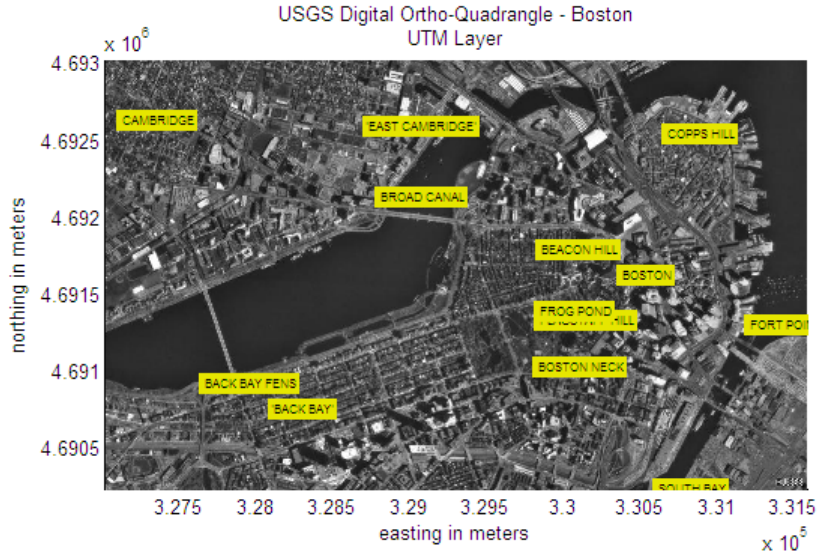
```
figure('Renderer','zbuffer')
axesm(mstruct)
geoshow(A_Geo,R_Geo)
textm(lat, lon, names, 'Color',[0 0 0], ...
    'BackgroundColor',[0.9 0.9 0],'FontSize',6);
tightmap on
showaxes
xlabel('easting in meters')
ylabel('northing in meters')
title({'USGS Digital Ortho-Quadrangle - Boston', ...
   'Geographic Layer'})
```



Courtesy U.S. Geological Survey

13 Display the ortho-imagery obtained in UTM coordinates.

```
figure
mapshow(A_UTM,R_UTM);
text(xUTM, yUTM, names, 'Color',[0 0 0], ...
    'BackgroundColor',[0.9 0.9 0],'FontSize',6,'Clipping','on');
axis tight
xlabel('easting in meters')
ylabel('northing in meters')
title({'USGS Digital Ortho-Quadrangle - Boston', 'UTM Layer'})
```



Courtesy U.S. Geological Survey

Modifying Your Request

In this section...

Setting the Geographic Limits and Time

A `WMSMapRequest` object contains properties to modify the geographic extent and time of the requested map. This example demonstrates how to modify your map request to map sea surface temperature for the ocean surrounding the southern tip of Africa. See the `WMSMapRequest class` reference page for a complete list of properties.

1 Search the WMS Database for all layers on NASA's Earth Observations (NEO) WMS server.

```
neowms = wmsfind('neowms', 'SearchField', 'serverurl');
```

2 Refine your search to include only layers with `'sea surface temperature'` in the layer title or layer name fields of the WMS database.

```
sst = neowms.refine('sea surface temperature');
```

3 Refine your search to include only layers with monthly values from the MODIS sensor on the Aqua satellite.

```
sst = sst.refine('month*modis');
```

4 Construct a `WebMapServer` object from the server URL stored in the `ServerURL` property of the `WMSLayer` object `sst`.

```
server = WebMapServer(sst(1).ServerURL);
```

5 Construct a `WebMapRequest` object from a `WMSLayer` array and a `WebMapServer` object.

```
mapRequest = WMSMapRequest(sst, server);
```

6. Use the `Latlim` and `Lonlim` properties of `WMSMapRequest` to set the latitude and longitude limits.

```
mapRequest.Latlim = [-45 -25];
mapRequest.Lonlim = [15 35];
```

7. Set the time request to March 1, 2009.

```
mapRequest.Time = '2009-03-01';
```

8. Send your request to the server with the `WebMapServer.getMap` method. Pass in a `WMSMapRequest.RequestURL`.

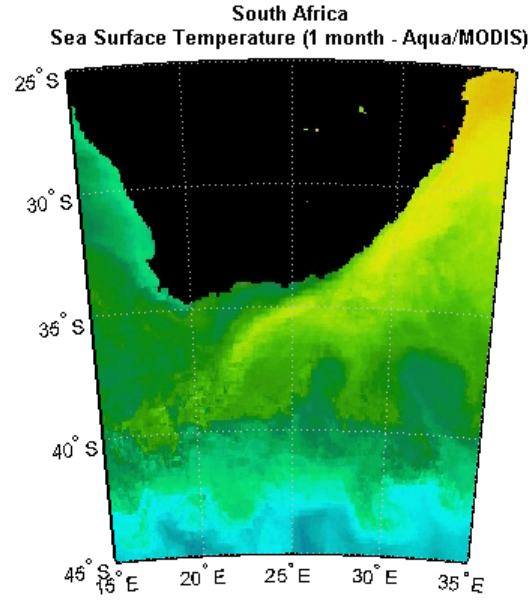
```
sstImage = server.getMap(mapRequest.RequestURL);
```

9. Set up empty map axes with the specified geographic limits.

```
figure
worldmap(mapRequest.Latlim, mapRequest.Lonlim);
setm(gca, 'mlabelparallel', -45)
```

10. 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', sst.LayerTitle}, ...
    'FontWeight', 'bold', 'Interpreter', 'none')
```



Courtesy NASA Earth Observations

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('bluemarbleng', '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://neowms.sci.gsfc.nasa.gov/wms/wms?...
   SERVICE=WMS...
   &LAYERS=BlueMarbleNG...
   &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:

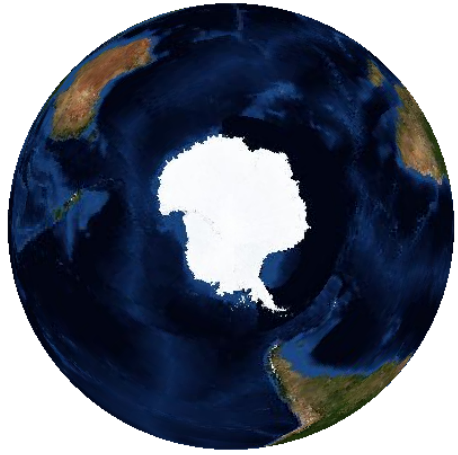
```
&BBOX=-180.0,-90.0,180.0,0.0
```

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')
```

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.

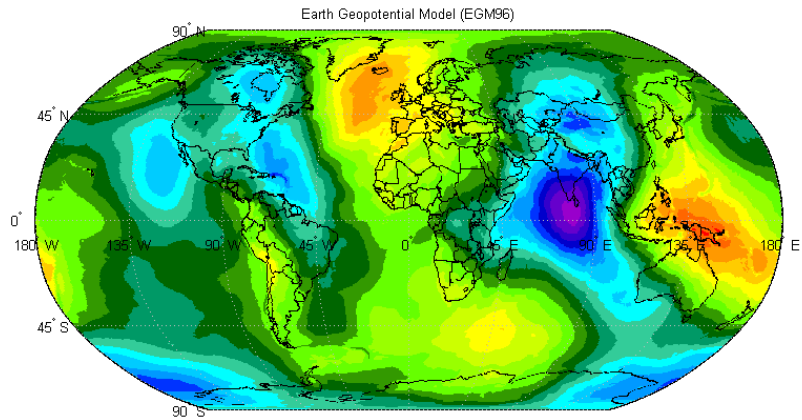
```
request = request.boundImageSize(720);
```

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

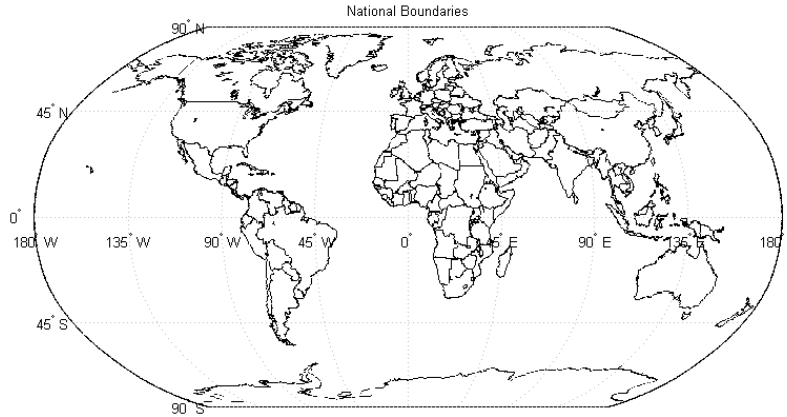
8 Read and display only the coastlines and national boundaries.

```
boundaries = [coastlines; national_boundaries];
```

```
[boundariesImage, R] = wmsread(boundaries, 'CellSize', .5, ...
   'Transparent', true);
```

9 Display the raster map.

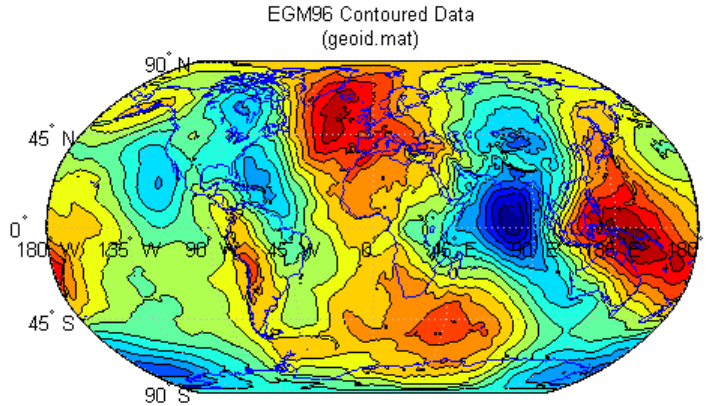
```
figure
worldmap('world')
geoshow(boundariesImage, R);
title(national_boundaries.LayerTitle)
```



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)'})
```



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.

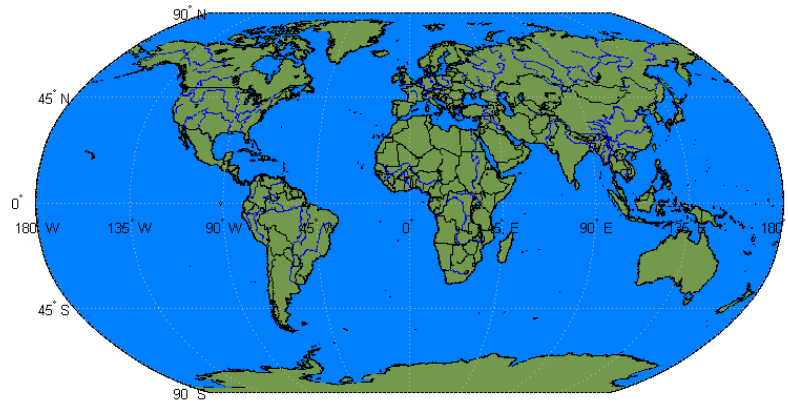
```
mergedImage(riverImage == 1) = 3;
```

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.

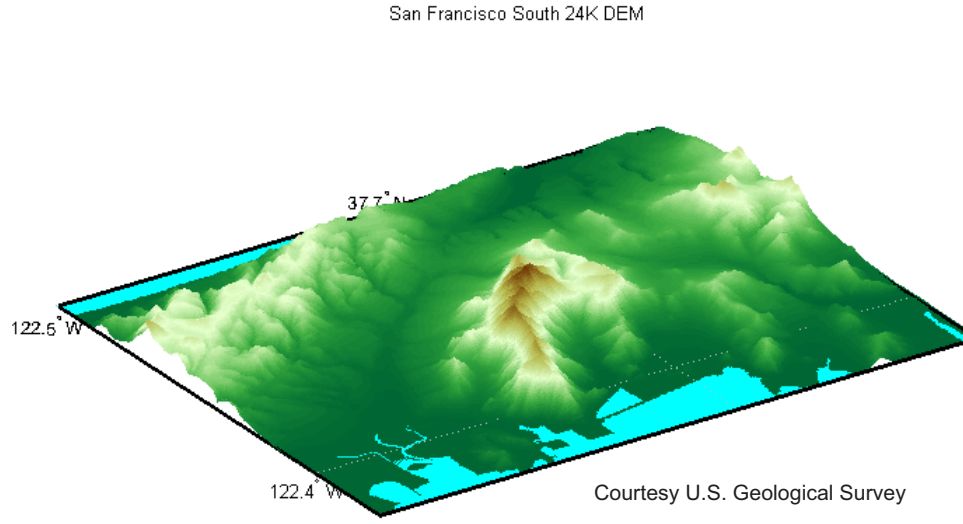
```
filenames = gunzip('sanfranciscos.dem.gz', tempdir);
demFilename = filenames{1};
[lat,lon,Z,header,profile] = usgs24kdem(demFilename, 1);
delete(demFilename);
Z(Z==0) = -1;
latlim = [min(lat(:)) max(lat(:))];
lonlim = [min(lon(:)) max(lon(:))];
```

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.

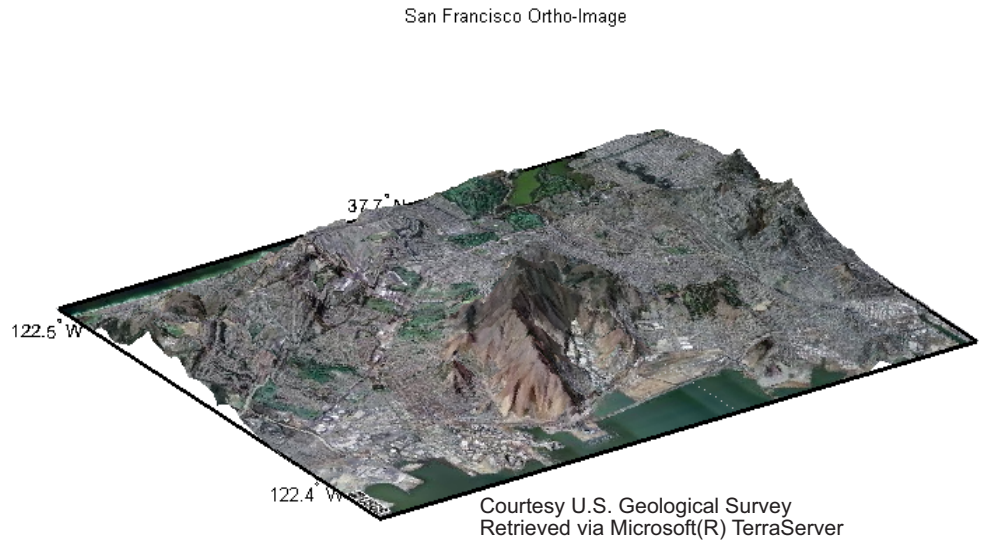
```
orthoImage = server.getMap(mapRequest.RequestURL);
```

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.

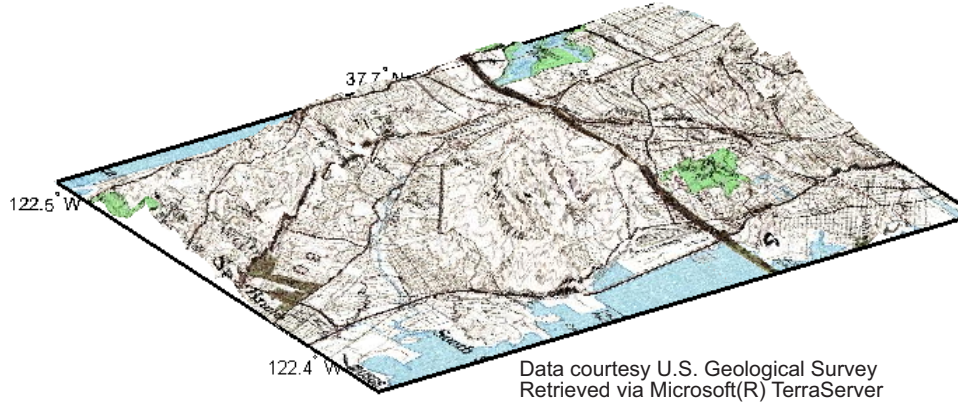
```
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, ...
   'DisplayType', 'surface', 'CData', orthoImage);
daspectm('m',1)
title('San Francisco Ortho-Image');
axis vis3d
set(gca,'CameraPosition', cameraPosition, ...
   'CameraTarget', cameraTarget, ...
   'CameraViewAngle', cameraAngle)
```



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



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.

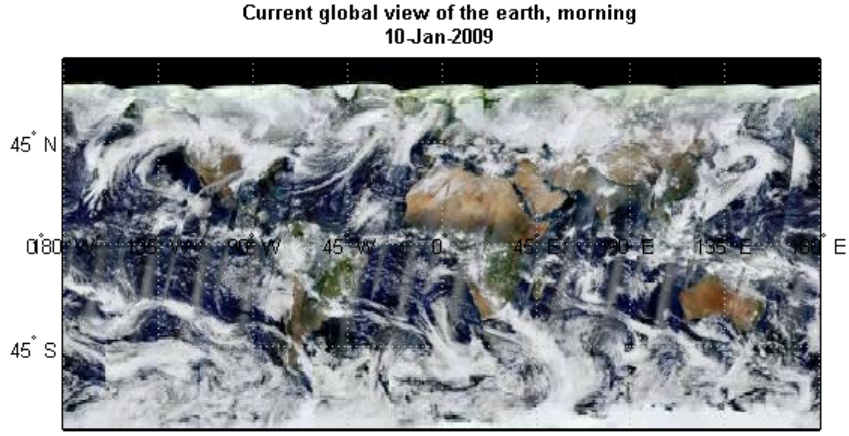
```
numberOfFrames = 31;
```

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
```



Courtesy NASA/JPL-Caltech

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 = strfind(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.

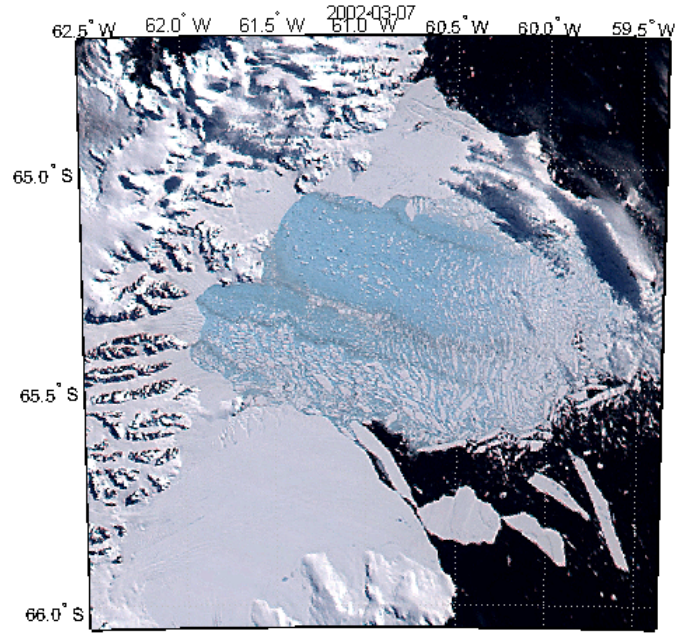
```
animated(1,1,1,numFrames) = 0;
```

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.

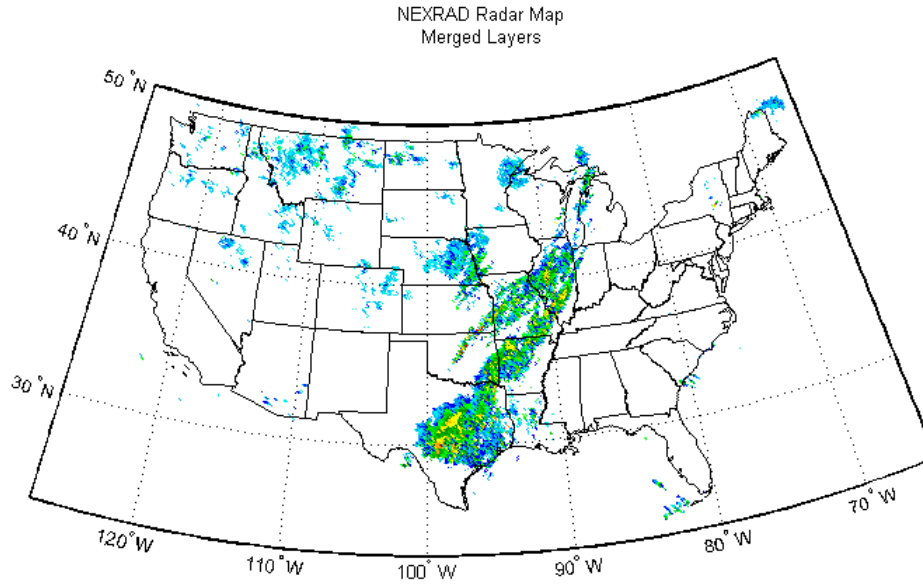
```
index = strcmpi('nexrad-n0r-900913',{nexrad.LayerName});
nexrad(index) = [];
index = strcmpi('nexrad-n0r-wmst',{nexrad.LayerName});
nexrad(index) = [];
```

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;
```

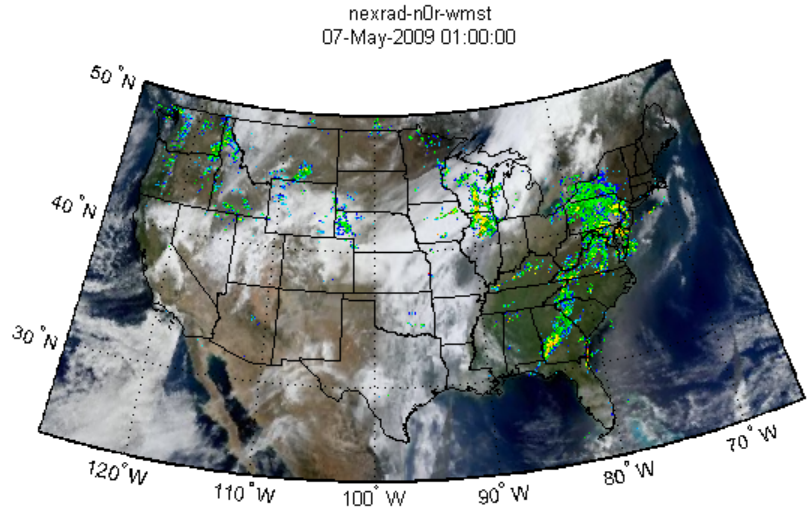
6 For each hour, obtain the hourly NEXRAD map data and combine it with a copy of the backdrop. Because of how this Web server handles PNG format, the resulting map data has an image with class `double`. Thus, you must convert it to `uint8` before merging.

```
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

Retrieving Elevation Data

In this section...

A WMS server typically renders a layer as an RGB image. To have the server return the underlying data, rather than the pictorial representation, often requires you to make a special request. In such cases, you may need to create either a Web Coverage Service (WCS) request, for raster data, or a Web Feature Service (WFS) request, for vector data. In some rare cases, you can request the actual data from a WMS server.

The following sections illustrate how to obtain elevation data from two different NASA WMS servers.

- The NASA JPL Global Imagery WMS server (`http://onearth.jpl.nasa.gov/wms.cgi?`) renders layers in the GeoTIFF (`image/geotiff`) format.
- The NASA WorldWind server (`http://www.nasa.network.com/elev`) renders layers in the band-interleaved (`image/bil`) format.

Display a Merged Elevation and Bathymetry Layer (SRTM30)

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 SRTM30 Plus data set combines GTOPO30, SRTM-derived land elevation and Sandwell bathymetry data from the University of California at San Diego.

Follow this example to read and display the SRTM30 Plus layer for the Gulf of Maine at a 30 arc-second sampling interval using data from the NASA JPL and WorldWind servers.

1. Find and update the `'srtmplus'` layer in the WMS Database. (The `'srtmplus'` layer from JPL is the name of the SRTM30 Plus data set.)

```
jpl = wmsfind('earth.jpl.nasa.gov', 'SearchField', 'serverurl');
srtmplus = jpl.refine('srtmplus');
srtmplus = wmsupdate(srtmplus);
```

2. Set the desired geographic limits.

```
latlim = [40 46];
lonlim = [-71 -65];
```

3. Set the sampling interval to 30 arc-seconds.

```
samplesPerInterval = dms2degrees([0 0 30]);
```

4. Set the `StyleName` property to `'short_int'` for meters.

```
styleName = 'short_int';
```

5. Set the `ImageFormat` to GeoTIFF.

```
imageFormat = 'image/geotiff';
```

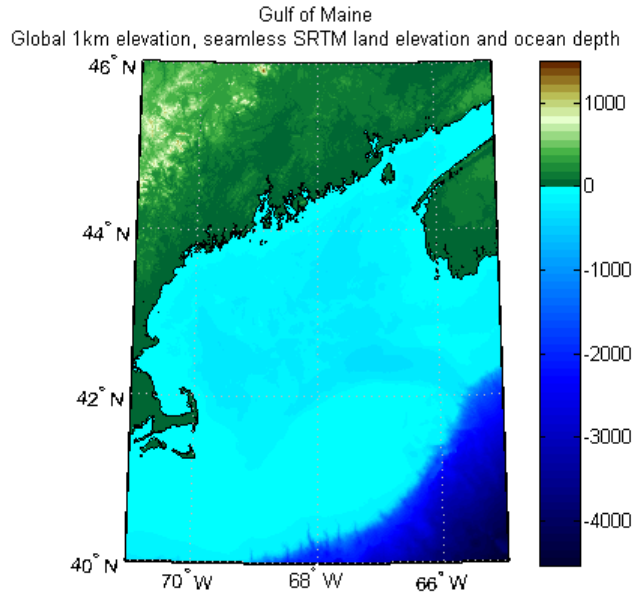
6. Request the map from the JPL server.

```
[Z1, R1] = wmsread(srtmplus, 'Latlim', latlim, ...
   'Lonlim', lonlim, 'ImageFormat', imageFormat, ...
   'StyleName', styleName, 'CellSize', samplesPerInterval);
```

7. Open a figure window and set up a map axes with geographic limits that match the desired limits. The referencing matrix `R1` ties the intrinsic coordinates of the raster map to the EPSG:4326 geographic coordinate system. Create a colormap appropriate for elevation data. Then, display and contour the map at sea level (0 m).

```
figure('Renderer','zbuffer')
worldmap(Z1, R1)
geoshow(Z1, R1, 'DisplayType', 'texturemap')
demcmap(double(Z1))
contourm(double(Z1), R1, [0 0], 'Color', 'black')
```

```
colorbar
title ({'Gulf of Maine', srtmplus.LayerTitle}, 'Interpreter','none')
```



Courtesy NASA/JPL

8 Compare the JPL layer with the WorldWind SRTM layer. The `'srtm30'` layer from WorldWind is the name for the SRTM30 data set.

```
wldwind = wmsfind('nasa.network', 'SearchField', 'serverurl');
srtm30 = wldwind.refine('srtm30', 'SearchField', 'layername');
srtm30 = wmsupdate(srtm30);
```

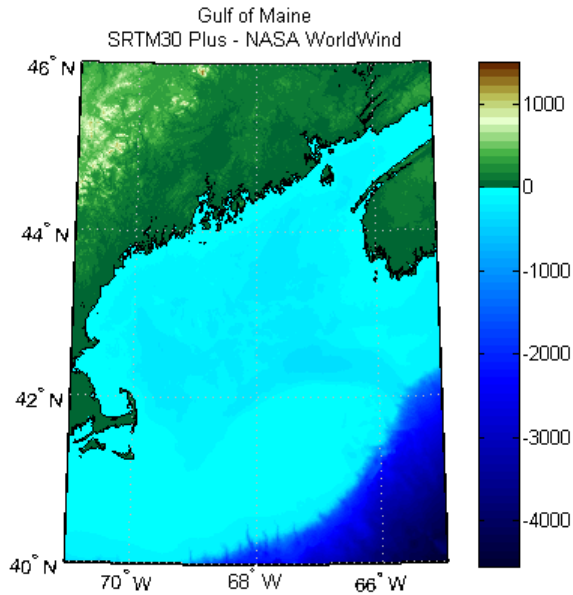
9 Request the map from the WorldWind server.

```
[Z2, R2] = wmsread(srtm30, 'Latlim', latlim, 'Lonlim', lonlim, ...
    'CellSize', samplesPerInterval);
```

10 Display the data.

```
figure('Renderer','zbuffer')
worldmap(Z2, R2)
```

```
geoshow(Z2, R2, 'DisplayType', 'texturemap')
demcmap(double(Z2))
contourm(double(Z2), R2, [0 0], 'Color', 'black')
colorbar
title ({'Gulf of Maine', [srtm30.LayerTitle ' - NASA WorldWind']})
```



11 Compare the results.

```
fprintf(...
   '\nJPL - %s\nMinimun value: %d\nMaximum value: %d\n', ...
   srtmplus.LayerName, min(Z1(:)), max(Z1(:)));
fprintf(...
   '\nWorldWind - %s\nMinimun value: %d\nMaximum value: %d\n', ...
   srtm30.LayerName, min(Z2(:)), max(Z2(:)));
```

The output appears as follows:

```
JPL - srtmplus
Minimun value: -4539
```

```
Maximum value: 1469

WorldWind - srtm30
Minimun value: -4543
Maximum value: 1463
```

Merge Elevation Data with Rasterized Vector Data

The NASA WorldWind WMS server contains a wide selection of layers containing elevation data. Follow this example to merge elevation data with a raster map containing national boundaries.

1 Find the layers from the NASA WorldWind server.

```
layers = wmsfind('nasa.network*elev', 'SearchField', 'serverurl');
layers = wmsupdate(layers);
```

2 Display the name and title of each layer.

```
disp(layers,'Properties',{'LayerTitle','LayerName'})

 11x1 WMSLayer

  Properties:
           Index: 1
      LayerTitle: 'SRTM30 with Bathymetry (900m) merged with
                   global ASTER (30m)'
       LayerName: 'EarthAsterElevations30m'

           Index: 2
      LayerTitle: 'USGS NED 30m'
       LayerName: 'NED'

                .
                .
                .

           Index: 10
      LayerTitle: 'SRTM30 Plus'
       LayerName: 'srtm30'
```

```
        Index: 11
   LayerTitle: 'USGS NED 10m'
    LayerName: 'usgs_ned_10m'
```

3. Select the `'EarthAsterElevations30m'` layer containing SRTM30 data merged with global ASTER data.

```
aster = layers.refine('earthaster', 'SearchField', 'layername');
```

4. Define the region surrounding Afghanistan.

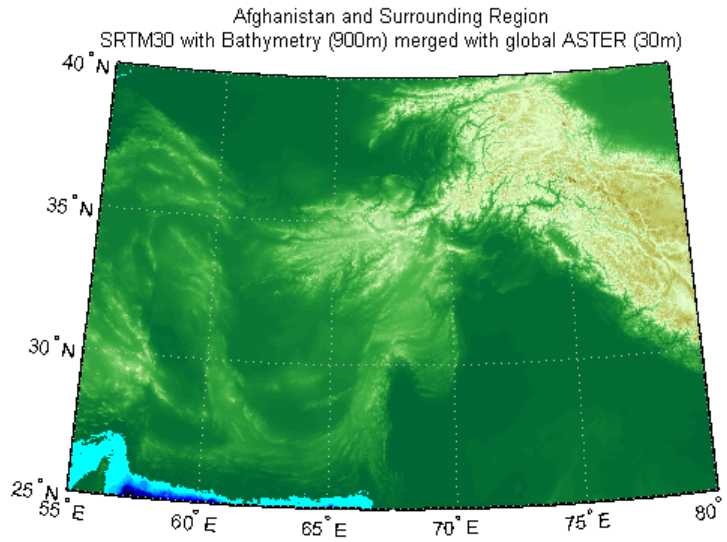
```
latlim = [25 40];
lonlim = [55 80];
```

5. Obtain the data at a 1-minute sampling interval.

```
cellSize = dms2degrees([0,1,0]);
[ZA, RA] = wmsread(aster, 'Latlim', latlim, 'Lonlim', lonlim, ...
    'CellSize', cellSize);
```

6. Display the elevation data as a texture map.

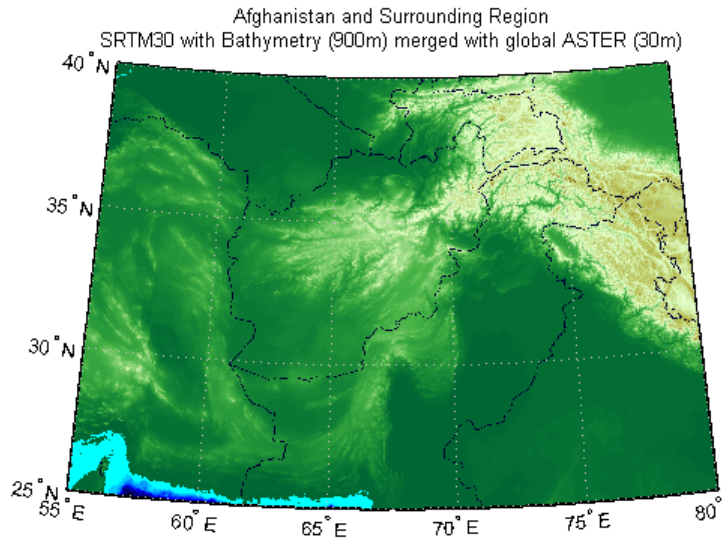
```
figure('Renderer','zbuffer')
worldmap('Afghanistan')
geoshow(ZA, RA, 'DisplayType', 'texturemap')
demcmap(double(ZA))
title({'Afghanistan and Surrounding Region', aster.LayerTitle});
```



Courtesy NASA WorldWind

7 Embed national boundaries from the NASA Globe Visualization Server into the elevation map.

```
vizglobe = wmsfind('viz.globe', 'SearchField', 'serverurl');
boundaries = vizglobe.refine('national*bound');
B = wmsread(boundaries, 'Latlim', latlim, ...
    'Lonlim', lonlim, 'CellSize', cellSize);
ZB = ZA;
ZB(B(:,:,1) == 0) = min(ZA(:));
geoshow(ZB, RA, 'DisplayType', 'texturemap')
```



Drape a Landsat Image onto Elevation Data

For some applications, you may want to merge elevation data with imagery. Follow this example to 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.

1 Obtain the layers of interest.

```
jpl = wmsfind('earth.jpl.nasa.gov', 'SearchFields', 'serverurl');
jpl = wmsupdate(jpl);
global_mosaic = jpl.refine('global_mosaic', 'MatchType', 'exact');
us_ned = jpl.refine('us_ned');
```

2 Assign geographic extent and image size.

```
latlim = [36 36.23];
lonlim = [-113.36 -113.13];
imageHeight = 575;
```

```
imageWidth = 575;
```

3 Read the `global_mosaic` layer.

```
[A, R] = wmsread(global_mosaic, 'StyleName', 'visual', ...
   'Latlim', latlim, 'Lonlim', lonlim, ...
   'ImageHeight', imageHeight, 'ImageWidth', imageWidth);
```

4 Read the USGS NED layer, using the `'image/geotiff'` style in order to obtain actual elevation values.

```
[Z, R] = wmsread(us_ned, 'ImageFormat', 'image/geotiff', ...
    'StyleName', 'real', 'Latlim', latlim, 'Lonlim', lonlim, ...
    'ImageHeight', imageHeight, 'ImageWidth', imageWidth);
```

5 Drape the Landsat image onto the elevation data.

```
figure('Renderer','opengl')
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'}, ...
   'FontSize',8);
axis vis3d
```

Courtesy NASA/JPL-Caltech and U.S. Geological Survey

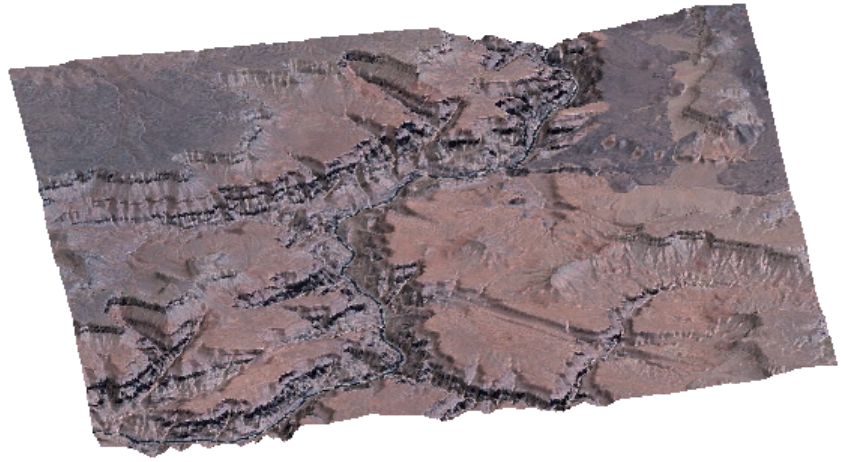
6 Assign camera parameters.

```
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];
```

7 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

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, WHOI, and ESA servers.

1 Find the servers and update all fields.

```
nasa = wmsfind('nasa','SearchField','serverurl');
whoi = wmsfind('whoi','SearchField','serverurl');
esa = wmsfind('esa.int','SearchField','serverurl');
favoriteLayers = [nasa; whoi; esa];
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');
   ```

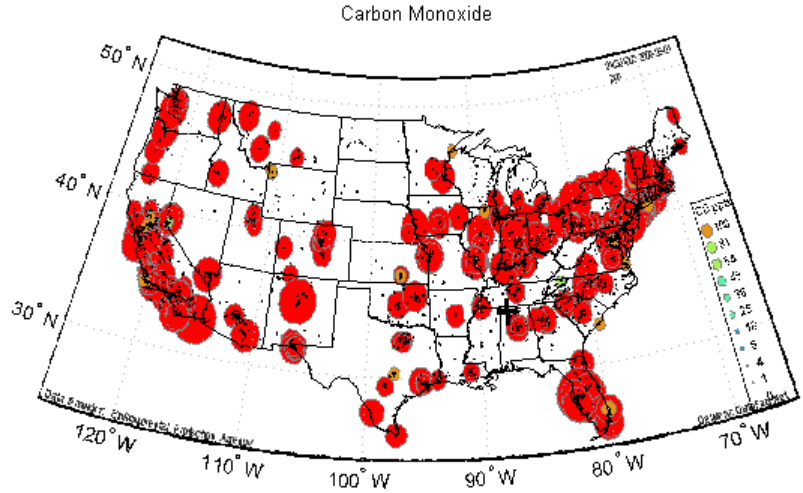
4. Set the longitude and latitude limits to the values specified for the layer.

   ```
   latlim = layer.Latlim;
   lonlim = layer.Lonlim;
   ```

5. Display the map.

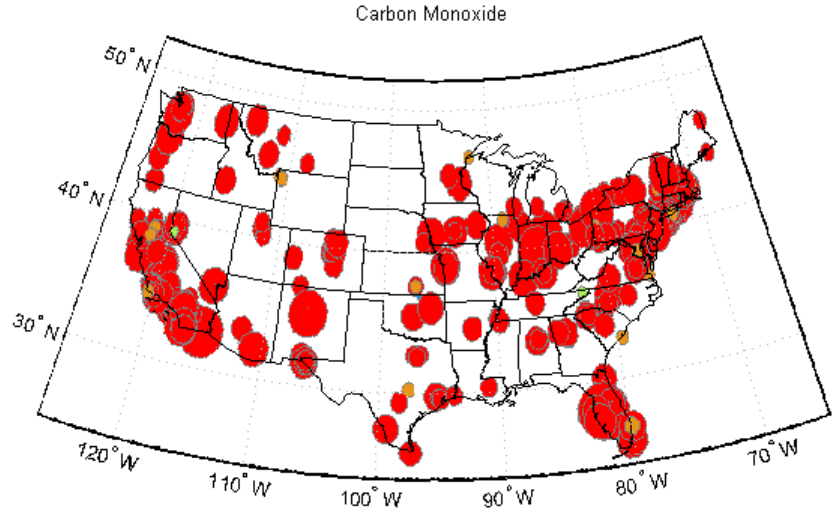
   ```
   figure
   ```

```
usamap(layer.Latlim, layer.Lonlim)
geoshow('usastatehi.shp','FaceColor','none','EdgeColor','black')
geoshow(A,R)
title(layer.LayerTitle)
```



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)
```


Carbon Monoxide
50°N
40°N
30°N
120°W
110°W
100°W
90°W
80°W
70°W
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('earth.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 9-81.

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 are not defined in the EPSG:4326 coordinate reference system. You cannot read these layers with the `wmsread` function.

Workaround: Use the `WMSMapRequest` class to construct a request URL and the `WebMapServer.getMap` method to read the layer. See Understanding Coordinate Reference System Codes and Retrieving Your Map with `WebMapServer.getMap` 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.
```

10

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)
```

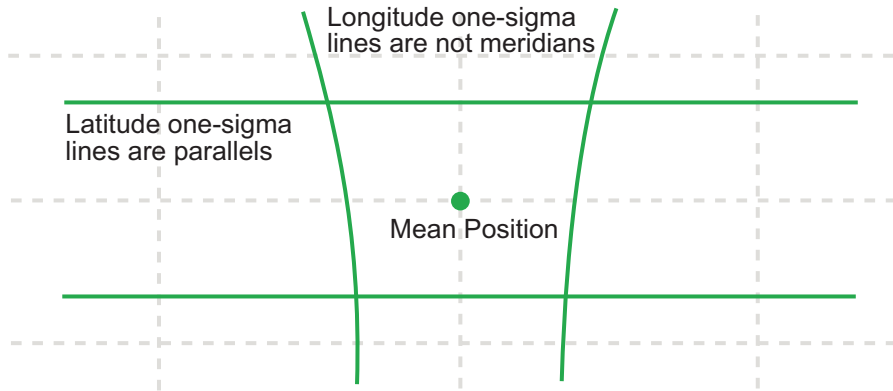
The function returns two deviations, one for latitudes and one for longitudes.

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.



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

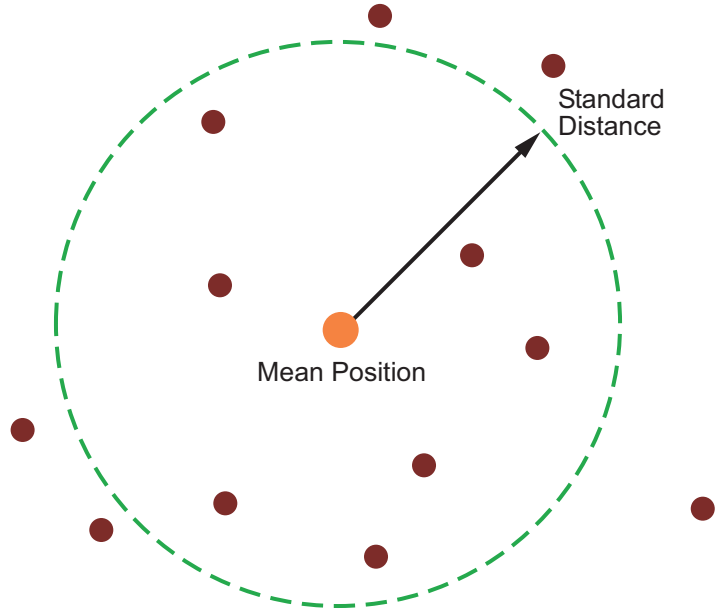
measure of deviation. The `stdm` handling is useful for many applications, especially when the data is not global. For instance, these potential difficulties would not be a danger for data points confined to the country of Mexico.

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

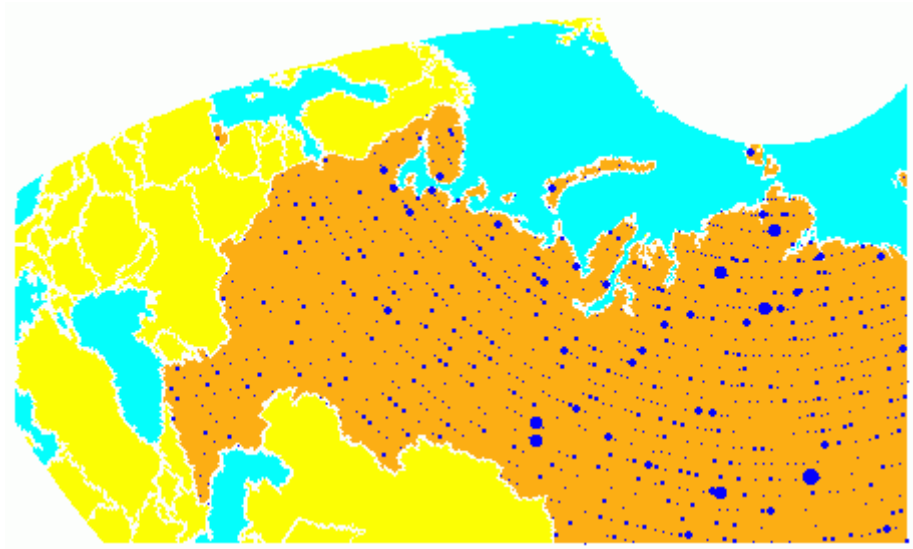
A common error in applying two-dimensional statistics to geographic data lies in ignoring equal-area treatment. It is often necessary to *bin* data to statistically analyze it. In a Cartesian plane, this is easily done by dividing the space into equal *x*-*y* squares. The geographic equivalent of this 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:

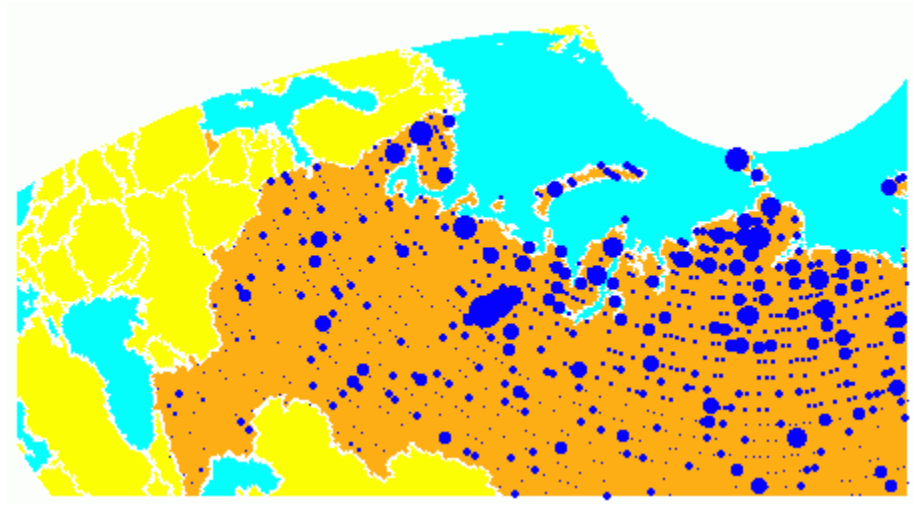


This is a portion of the whole map, displayed in an equal-area Bonne projection. The first step in creating data displays without area bias is to choose an equal-area projection. The proportionally sized symbols are a result of the specialized display function `scatterm`.

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`

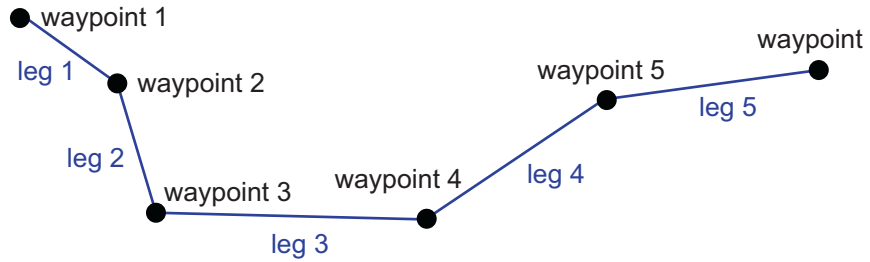
To make these functions easy to use, and to conform to common navigational practice, *for these specific functions only*, certain conventions are used:

- 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.

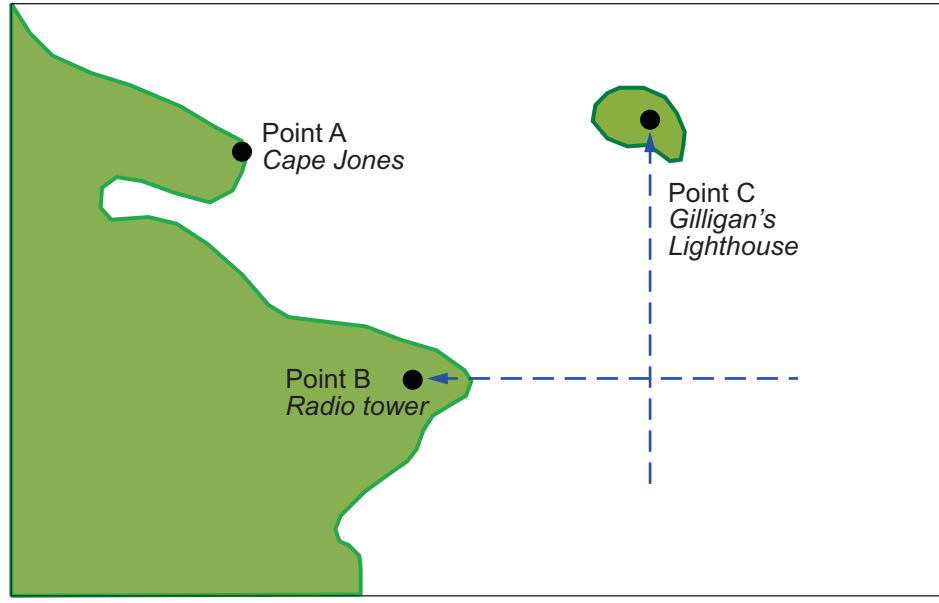
The Mercator was designed exactly for this purpose. Range circles, which might be determined with a radar, are assumed to plot as true circles on a Mercator chart. This allows the navigator to manually draw the range arc with a compass.

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.

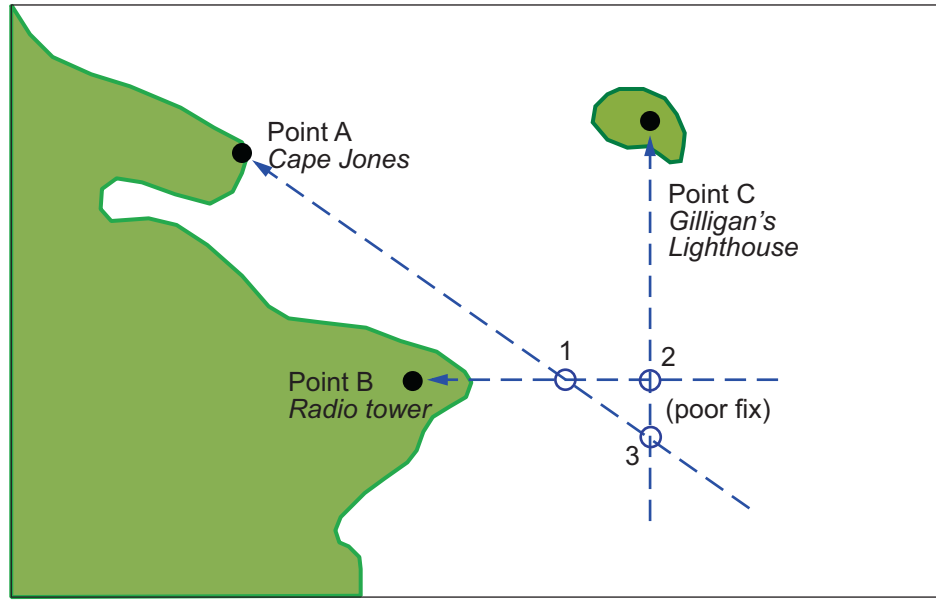
To obtain a good navigational fix, your relationship to at least three known points is considered necessary. A questionable or poor fix can be obtained with two known points.

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.



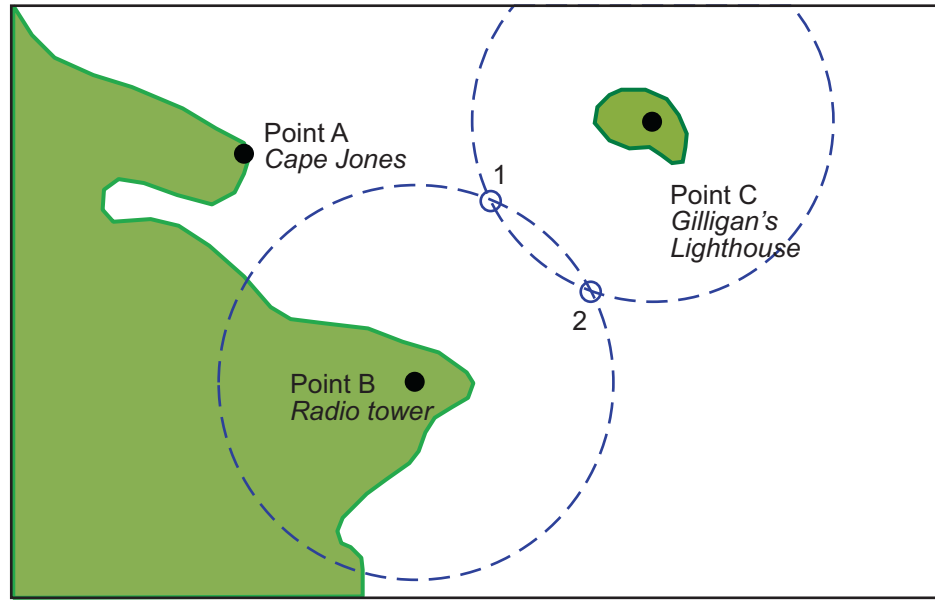
Whoops. What happened? Is your lookout in error? Possibly, but perhaps one or both of your bearings was slightly in error. This happens all the time. Which point, 1, 2, or 3, is correct? As far as you know, they are all equally valid.

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.

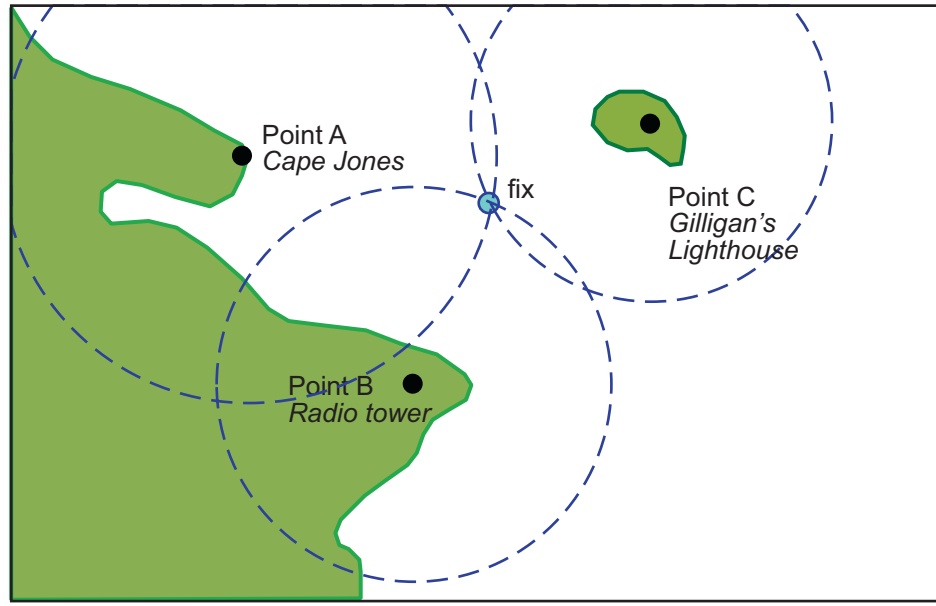
The next time you traverse these straits, it is a very foggy morning. You can't see any landmarks, but luckily, your navigational radar is operating. Each of these landmarks has a good radar signature, so you're not worried.

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.

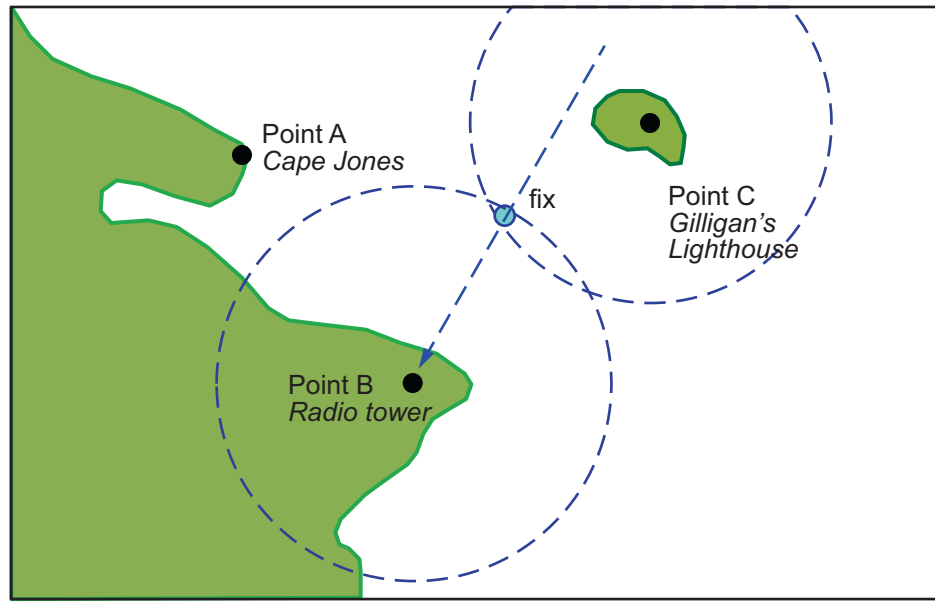
Luckily, your radar watch reports that he has Cape Jones at 18 nautical miles. This should resolve everything.



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:

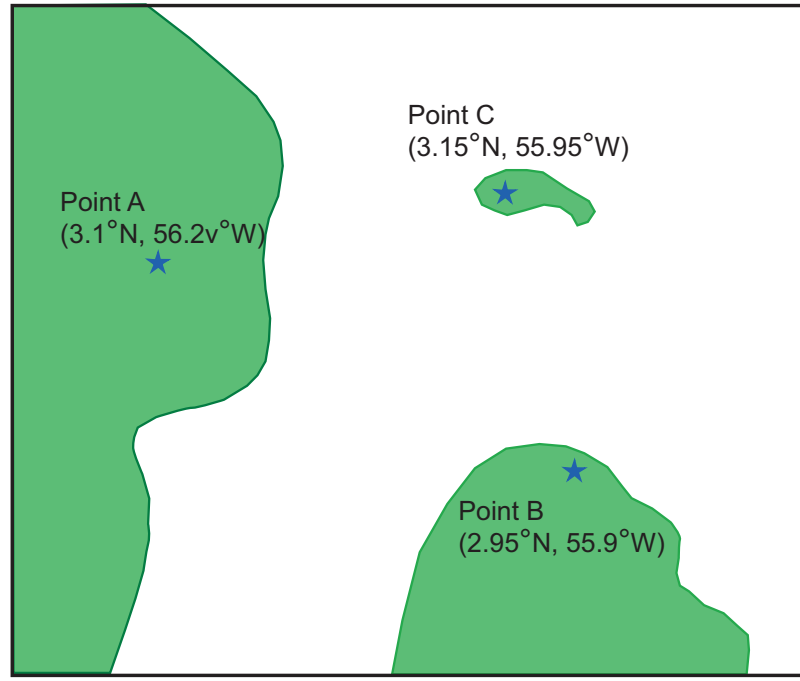
```
lata = 3.1;  lona = -56.2;
latb = 2.95; lonb = -55.9;
```

```
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 (with labeling and imaginary coastlines added after the fact for illustration):



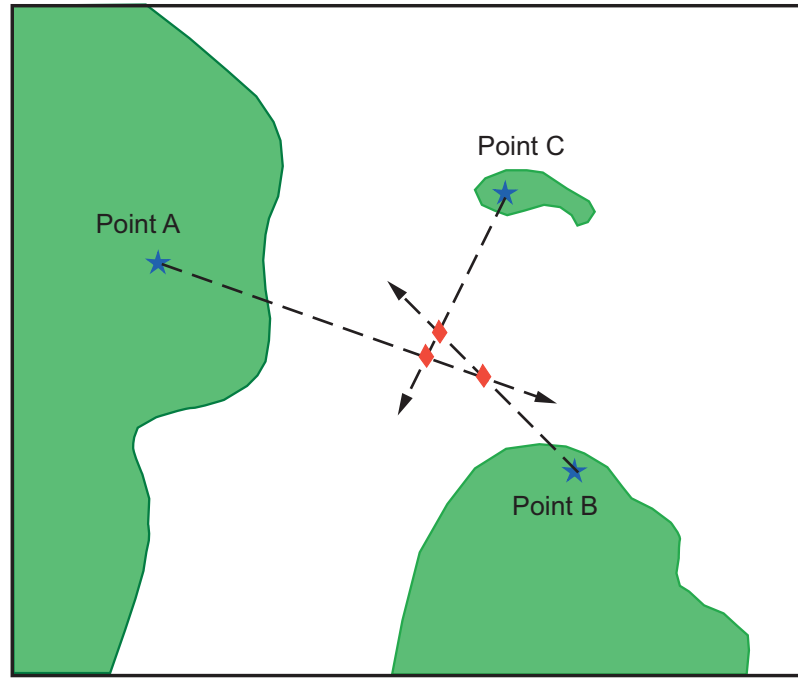
3 Take three visual bearings: Point A bears 289º, Point B bears 135º, and Point C bears 026.5º. Calculate the intersections:

```
[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 intersection points to the map:

```
plotm(newlat,newlong,'LineStyle','none',...
 'Marker','diamond','MarkerEdgeColor','r',...
 'MarkerFaceColor','r','MarkerSize',9)
```

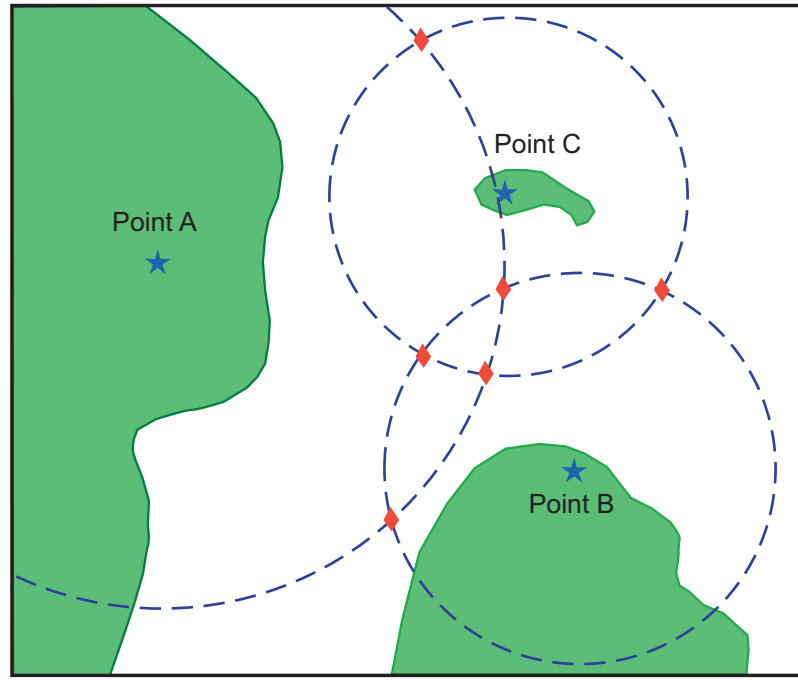


Bearing lines have been added to the map for illustration purposes. Notice that each pair of objects results in only one intersection, since all are lines of bearing.

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
```

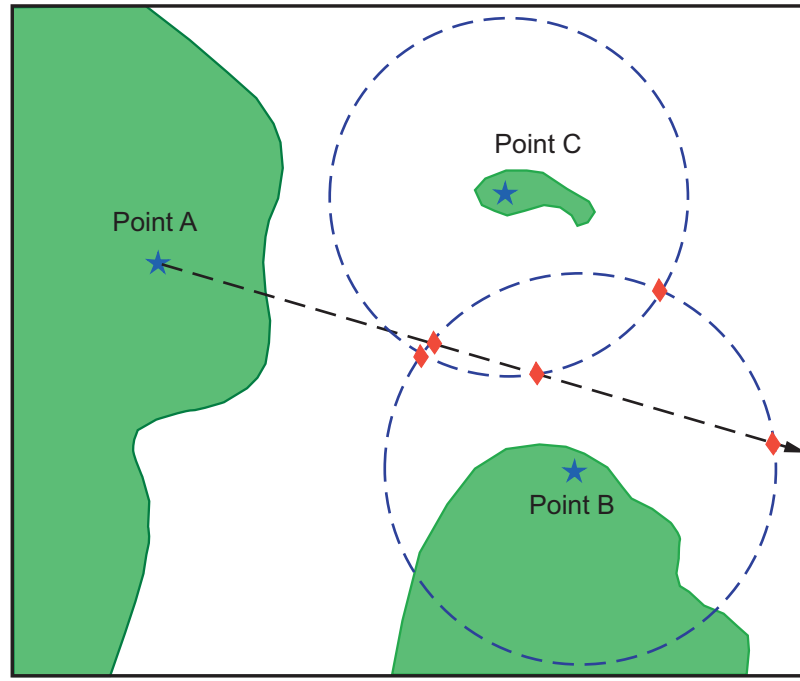
Here's what these points look like:



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 the Shortest Path

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.

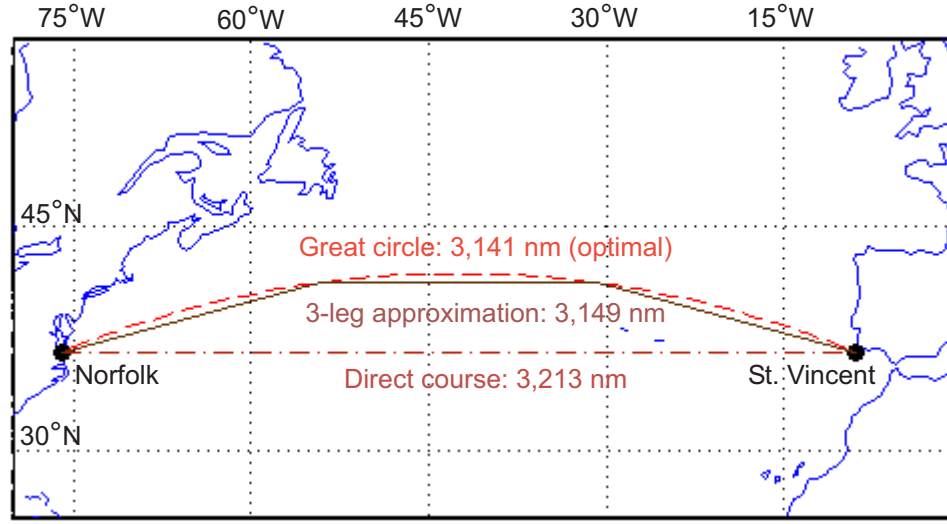
Consider the voyage from Norfolk, Virginia (37ºN,76ºW), to Cape St. Vincent, Portugal (37ºN,9ºW), one of the most heavily trafficked routes in the Atlantic. A due-east rhumb line track is 3,213 nautical miles, while the optimal great circle distance is 3,141 nautical miles.

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.

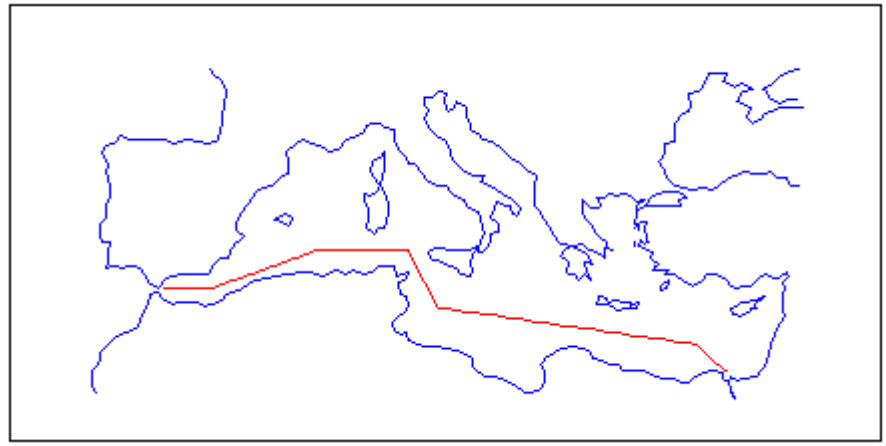
To accomplish this with Mapping Toolbox functions, you can display a map axes with a Mercator projection, select appropriate map latitude and longitude limits to isolate the area of interest, plot coastline data, and interactively mouse-select the waypoints with the `inputm` function. The `track` function 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 4-78, 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
```

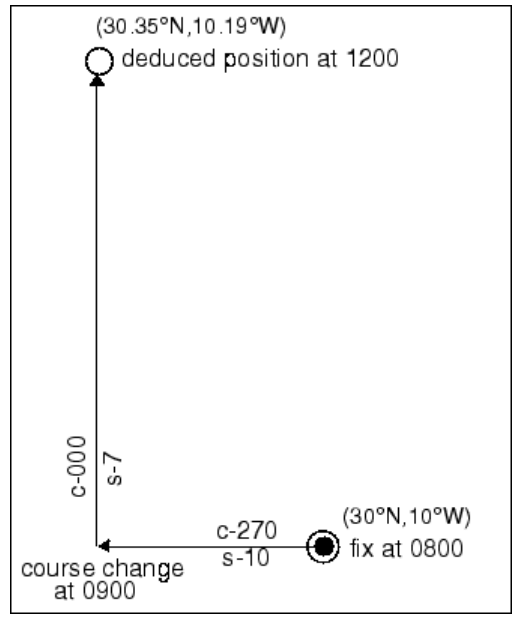
```
356.2117
283.6839
204.2073
854.0092
135.6415
```

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?

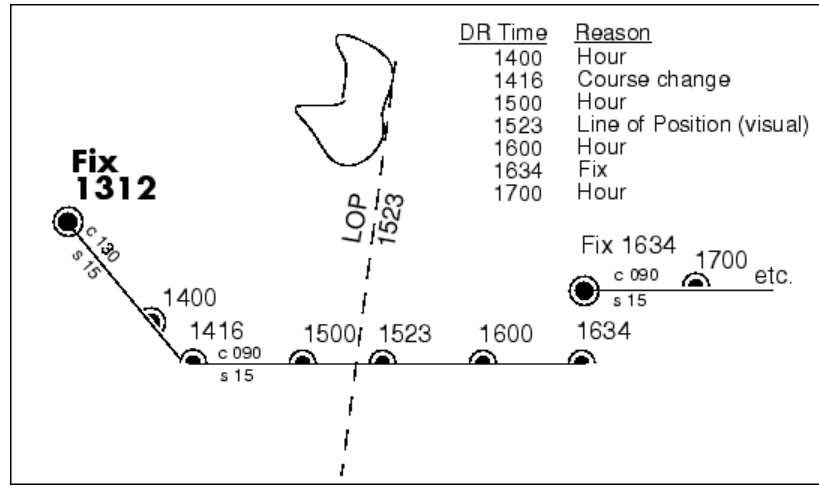
Leaving aside the difficulties in speed determination and the need to tack off course, even modern craft have to contend with winds and currents. However, despite these limitations, dead reckoning is still used for determining position between fixes and for forecasting future positions. This is because 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:



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;
```

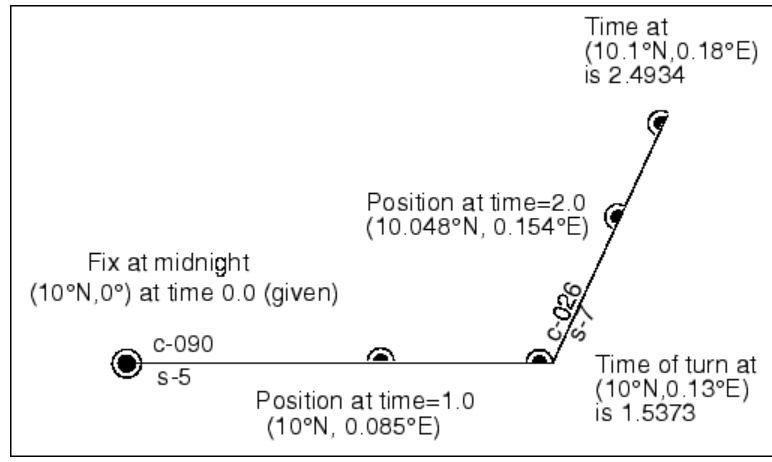
You intend to travel east and alter course at the point (10ºN,0.13ºE) and head for the point (10.1ºN,0.18ºE). On the first leg, you will travel at 5 knots, and on the second leg you will speed up to 7 knots.

```
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:

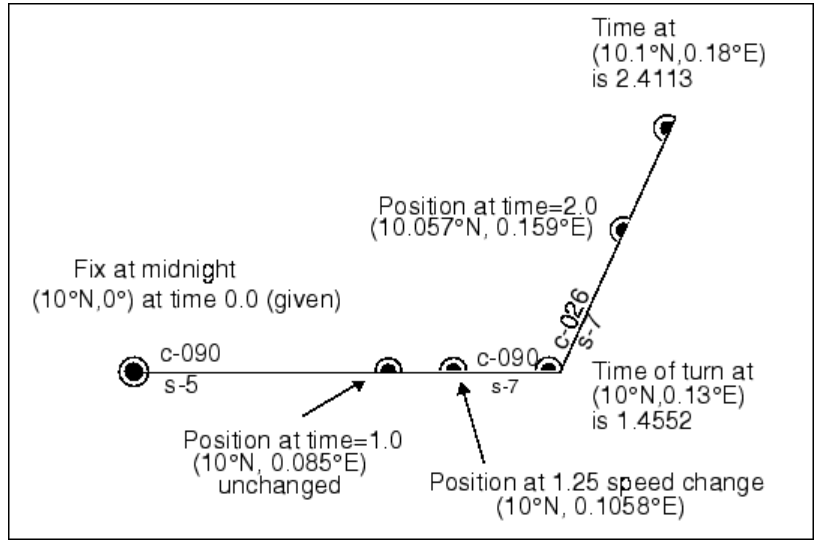


However, you would like to get to the final point a little earlier to make a rendezvous. You decide to recalculate your DRs based on speeding up to 7 knots a little earlier than planned. The first calculation tells you that you were going to increase speed at the turn, which would occur at a time 1.5373 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:



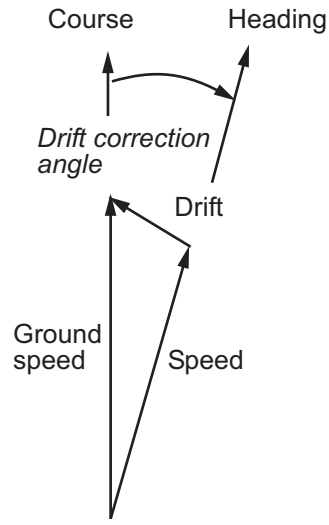
The times at planned positions after the speed change are a little earlier; the position at the known time (2 a.m.) is a little farther along. With this plan, you will arrive at the rendezvous about 4 1/2 minutes earlier, so you may want to consider a greater speed change.

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.

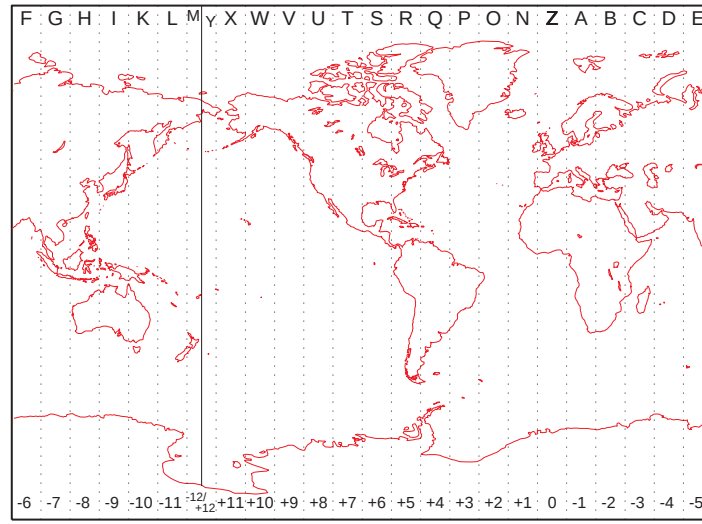
```
[windfrom,windspeed] = ...
driftvel(course,groundspeed,heading,airspeed)

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 15º 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 1.5º 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 123ºE 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.

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

12

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.

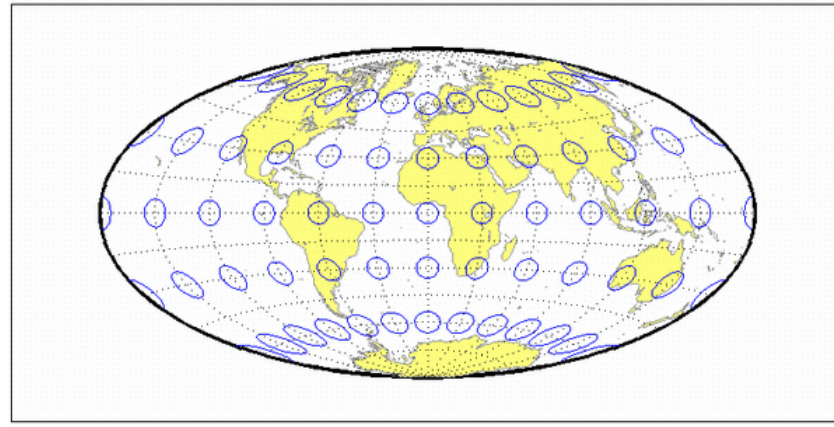
Features This projection is neither conformal nor equal area. The only point free of distortion is the center point. Distortion of shape and area are moderate throughout. This projection has less angular distortion on the outer meridians near the poles than pseudoazimuthal projections

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

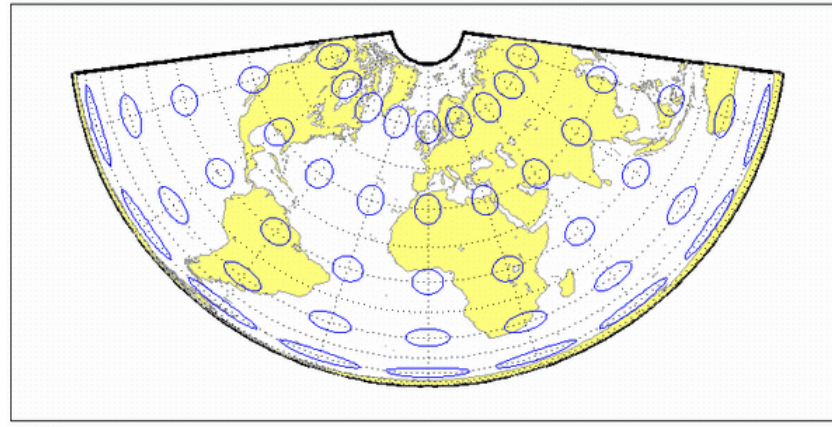
This projection was presented by Heinrich Christian Albers in 1805.

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

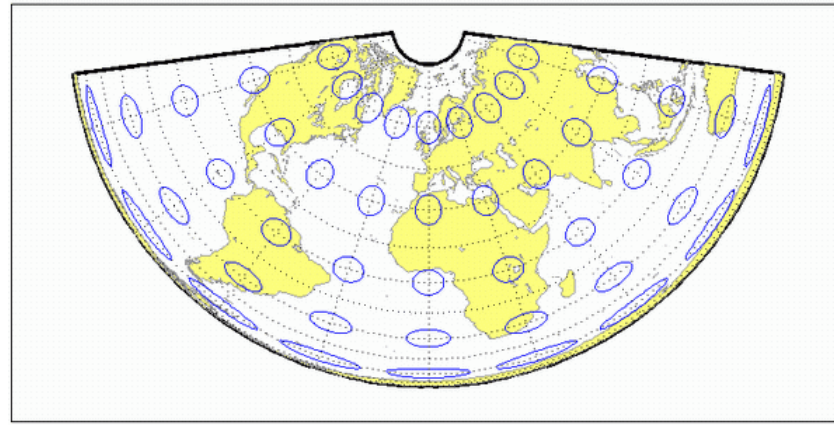
parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

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.

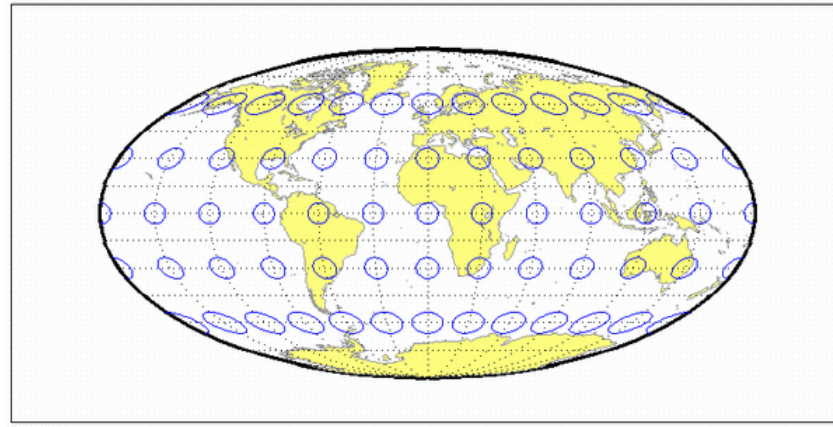
Parallels There is no standard parallel for this projection.

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

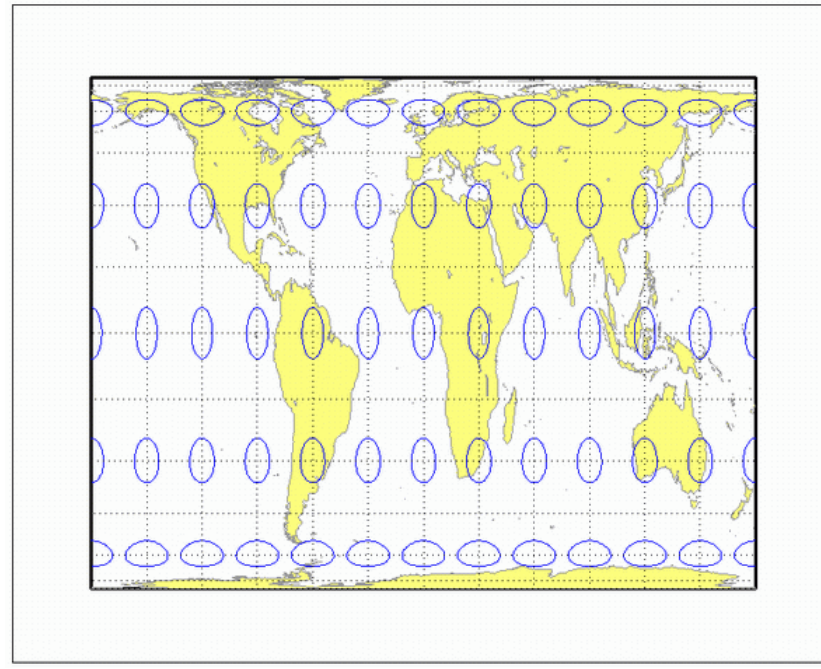
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 50º.

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.

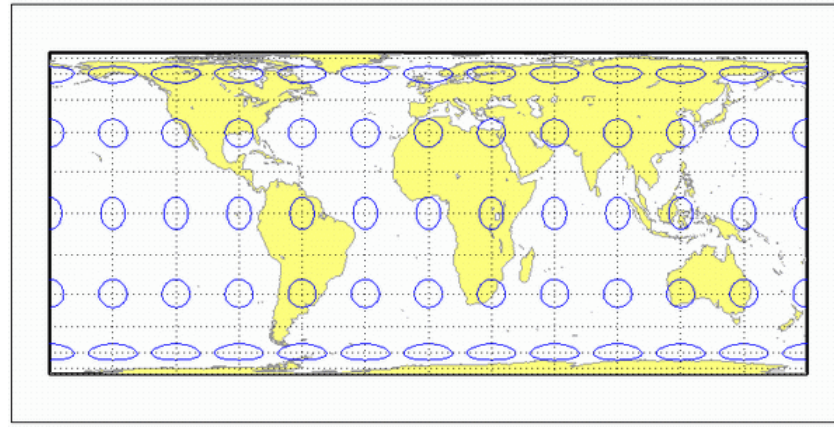
Features This is an orthographic projection onto a cylinder secant at the 30º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection 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

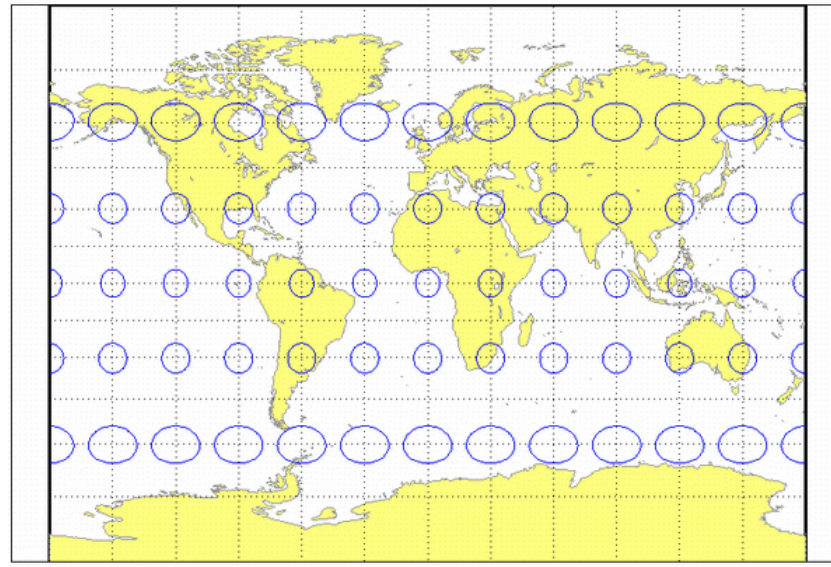
This projection was first described in 1937, when it was used for maps in the *Bolshoi Sovietskii Atlas Mira* (Great Soviet World Atlas). It is commonly abbreviated as the BSAM projection. It is a special form of the Braun Perspective Cylindrical projection secant at 30ºN and S.

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

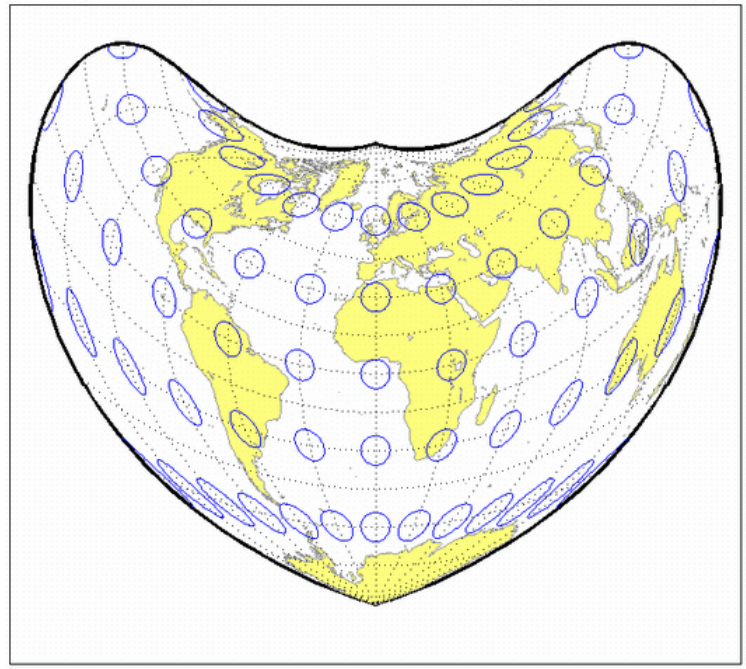
This projection has one standard parallel, which is 30ºN by default. It has two interesting limiting forms. If a pole is employed as the standard parallel, a Werner projection results; if the Equator is used, a Sinusoidal projection results.

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 0^{o}.

Remarks

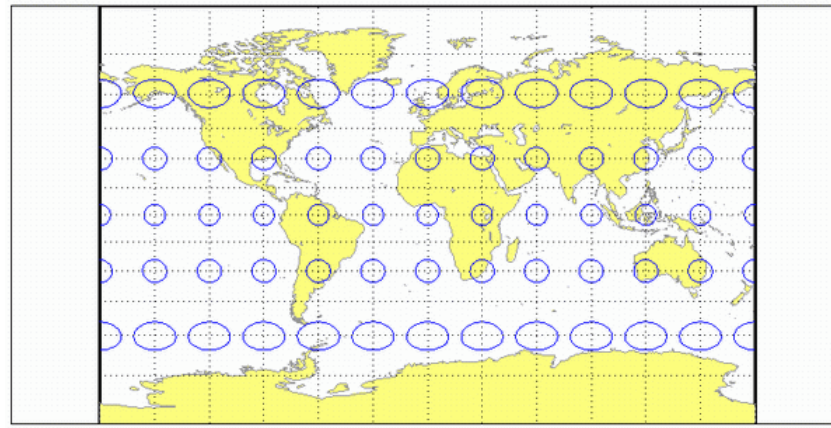
This projection was first described by Braun in 1867. It is less well known than the specific forms of it called the Gall Stereographic and the *Bolshoi Sovietskii Atlas Mira* projections.

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

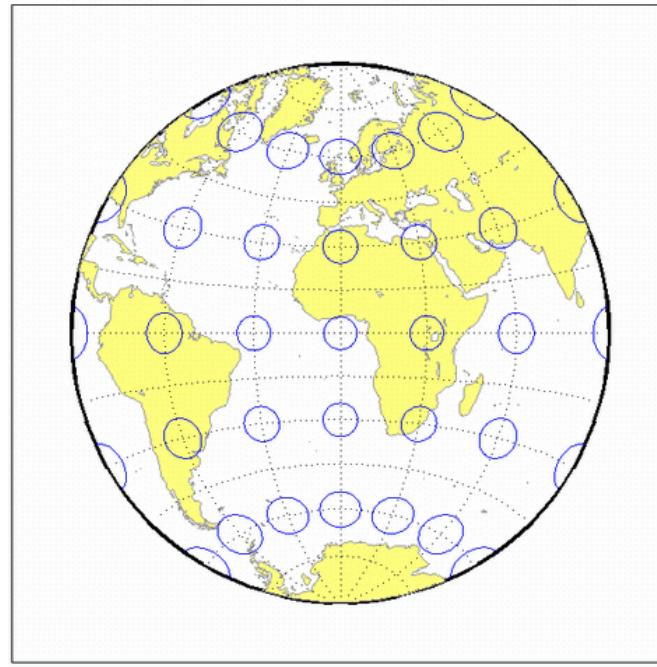
F. A. Arthur Breusing developed a geometric mean version of this projection in 1892. A. E. Young modified this to the harmonic mean version presented here in 1920. This projection is virtually indistinguishable from the Airy Minimum Error Azimuthal projection, presented by George Airy in 1861.

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.

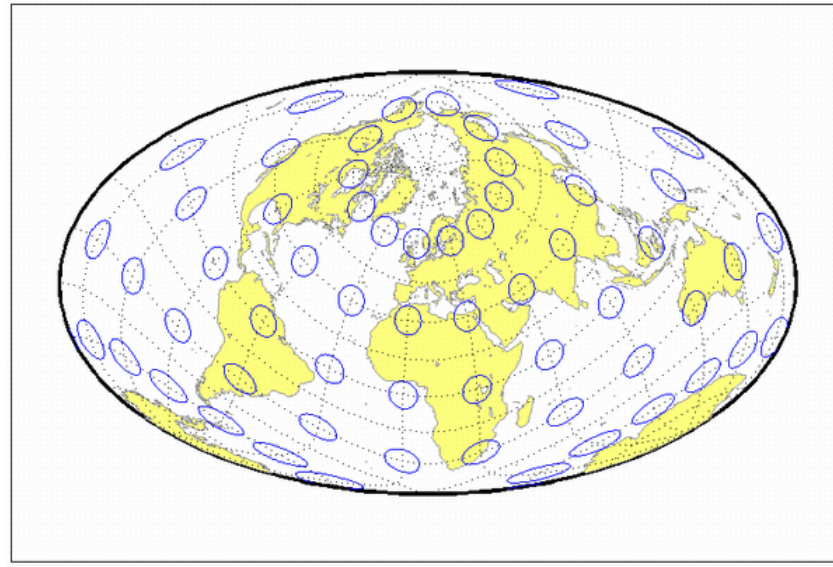
Features This equal-area projection groups the continents about the center of the projection. The only point free of distortion is the center point. Distortion of shape and area are moderate throughout.

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 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

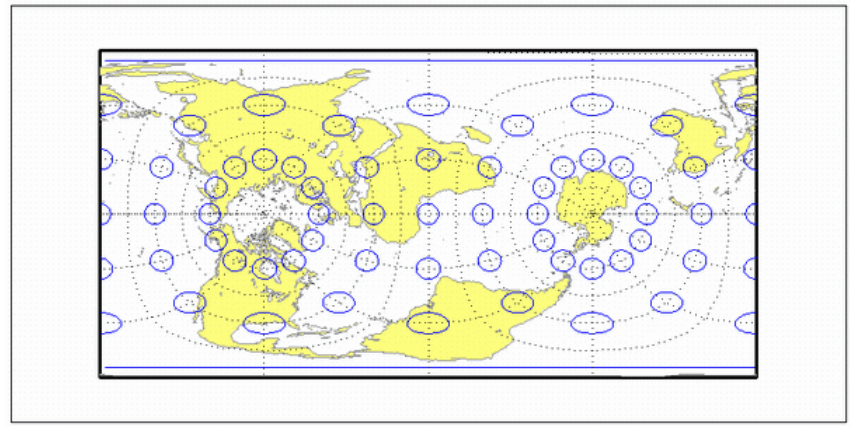
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel *of the base projection* is by definition fixed at 0º.

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.

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

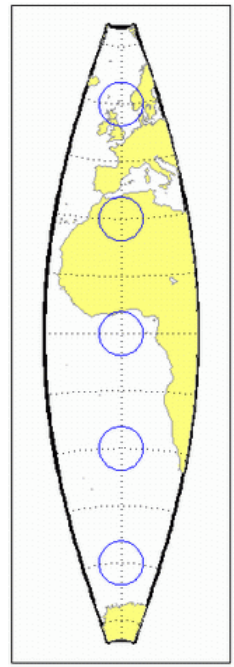
This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.

`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.

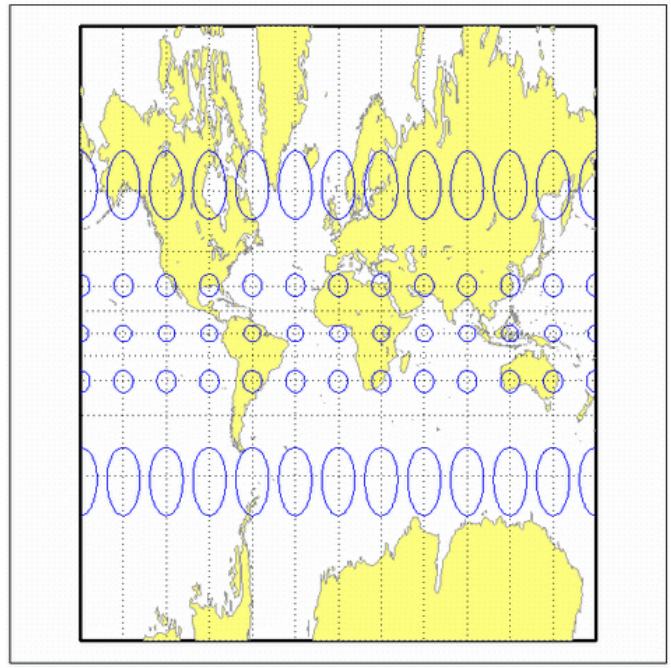
Limitations

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.

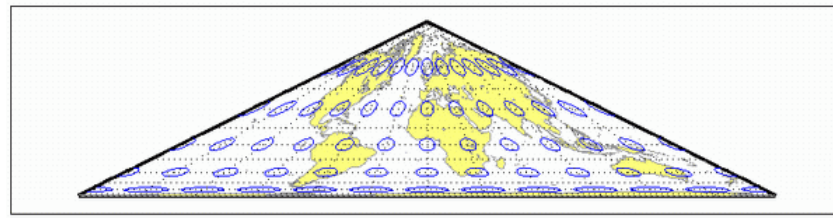
Features This is a novelty projection showing a straight-line, equal-area graticule. Scale is true along the 15°51'N parallel, constant along any parallel, and *different* for any pair of parallels. Distortion is severe in many regions, and is only absent at 15°51'N on the central meridian. This projection 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'.

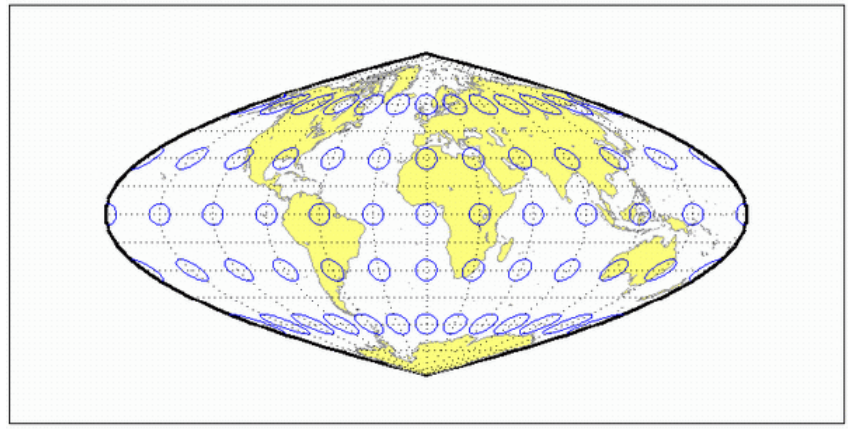
Remarks

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;
```

Craster Parabolic 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.

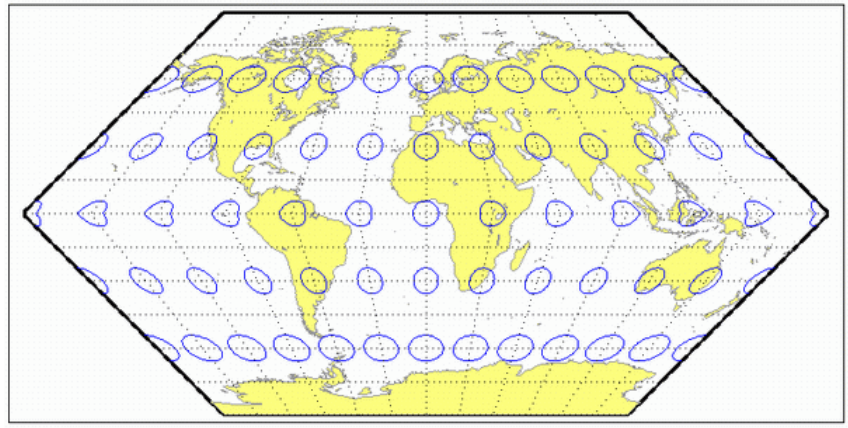
Limitations

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 I 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'.

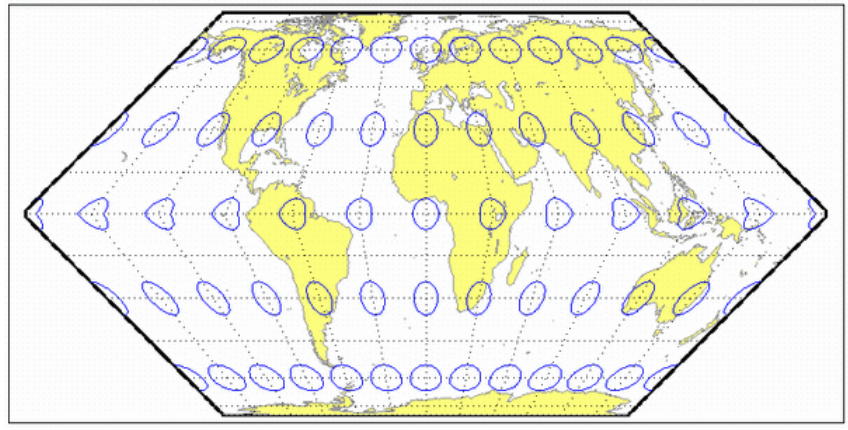
Remarks

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 II 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.

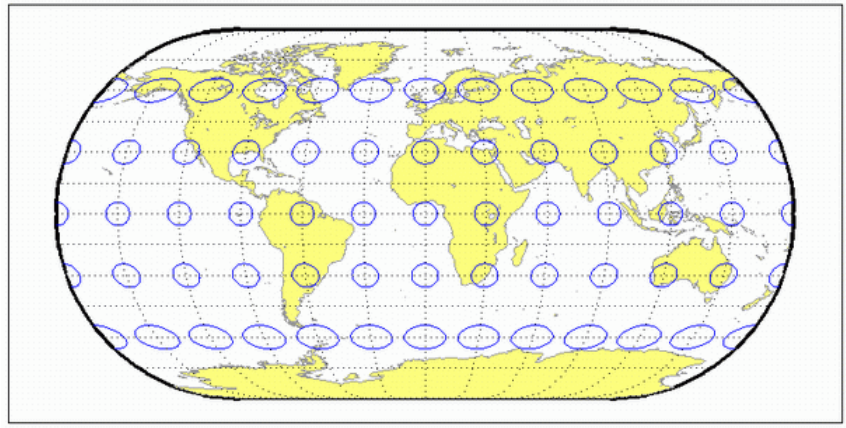
Limitations

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 III 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'.

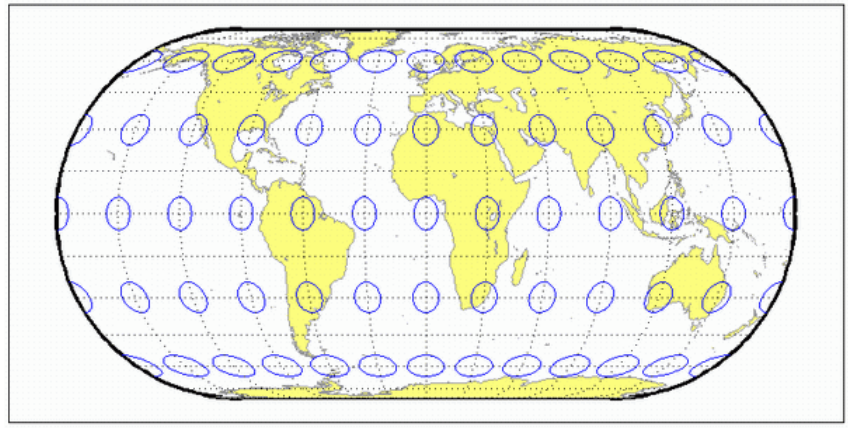
Remarks

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 IV 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 37°55'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 0°.

Remarks

This projection was presented by Max Eckert in 1906.

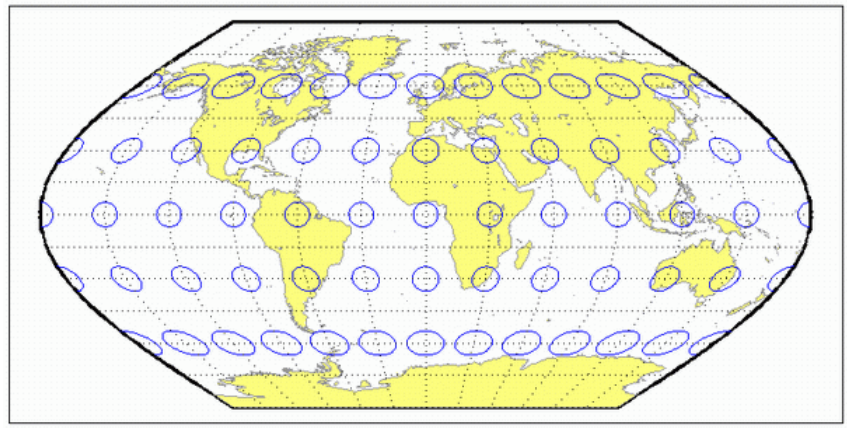
Limitations

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;
```

Eckert V Projection



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'.

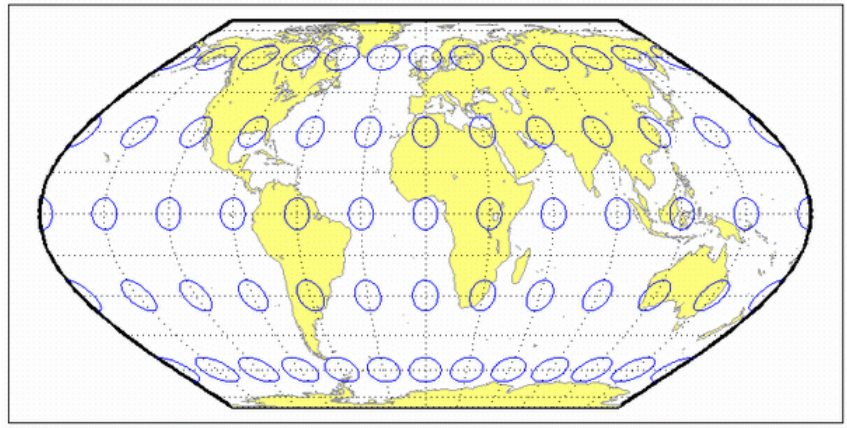
Remarks

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;
```

Eckert VI 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).

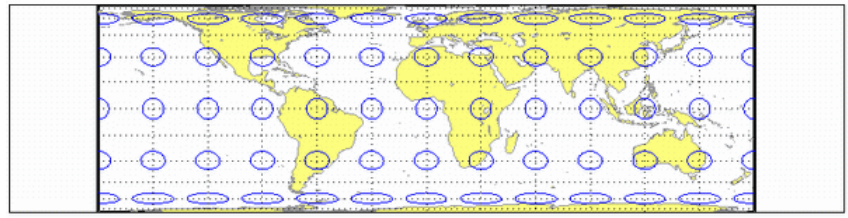
Remarks

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;
```

Equal-Area Cylindrical 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.

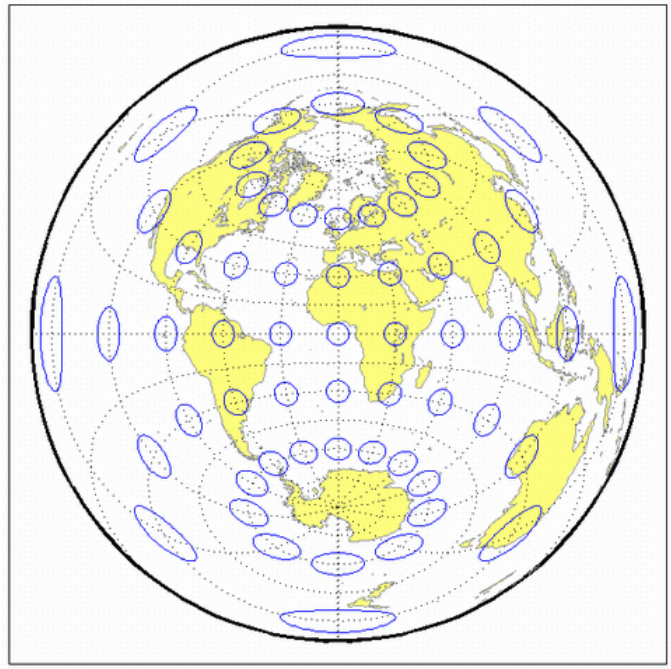
Limitations

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;
```

Equidistant Azimuthal Projection



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.

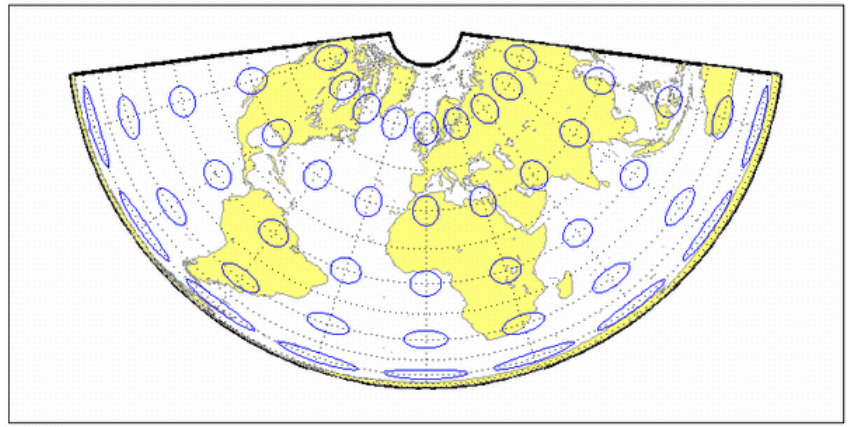
Equidistant Conic Projection

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`

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].

Equidistant Conic Projection — Standard

Remarks

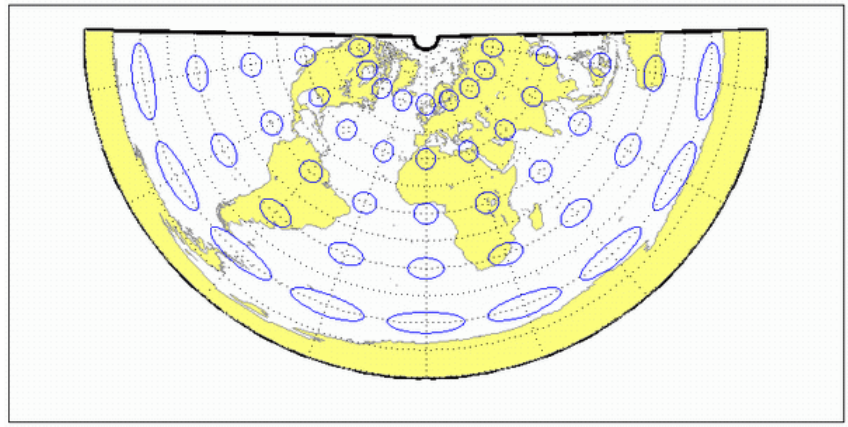
In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.

Limitations

Longitude data greater than 135° east or west of the central meridian is trimmed.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eqdconicstd', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



See Also

`eqdconic`

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.

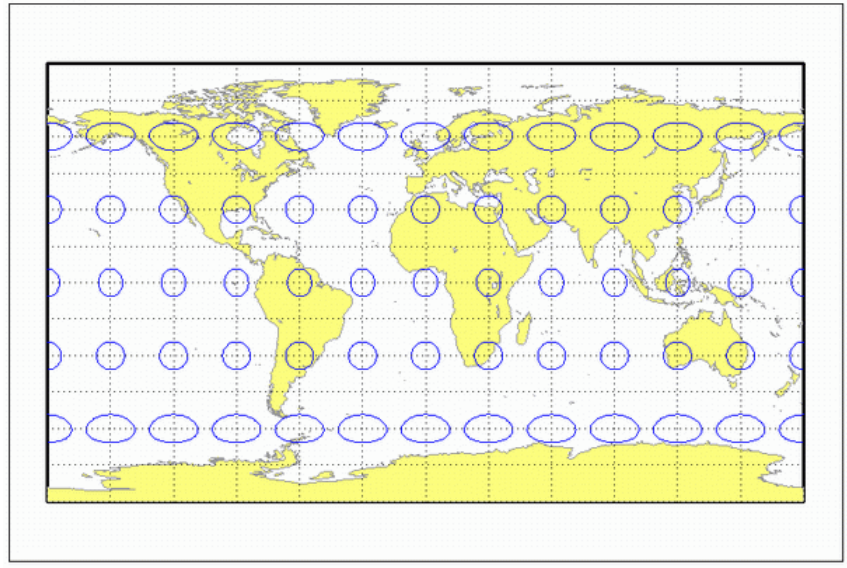
Remarks

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;
```

Equidistant Cylindrical 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.

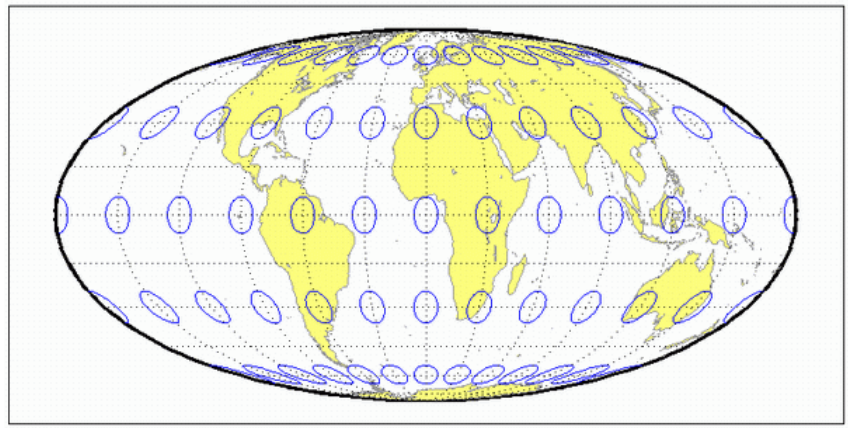
Remarks

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;
```

Fournier 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 45º 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 45º.

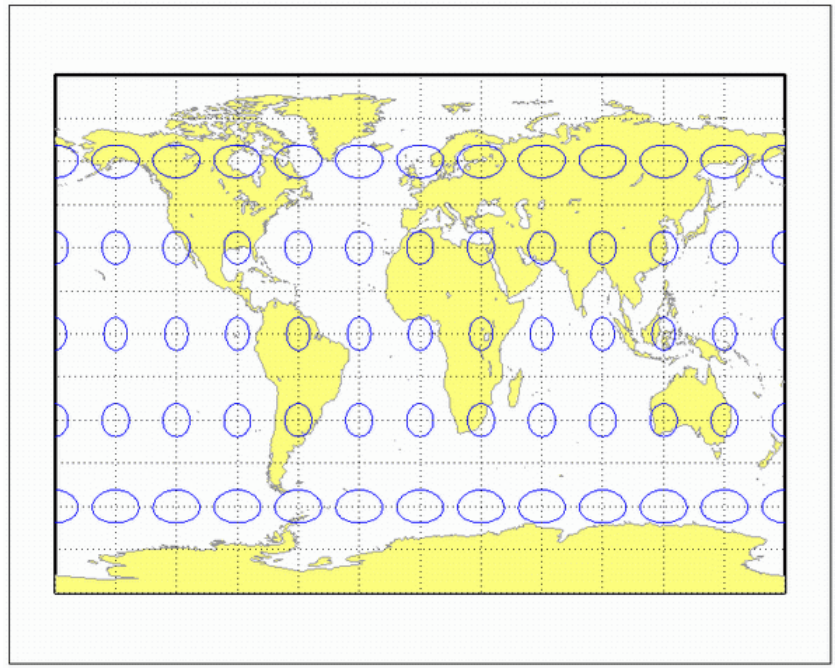
Remarks

This projection is a specific case of the Equidistant Cylindrical projection, with standard parallels at 45ºN 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;
```

Gall Isographic Projection



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.

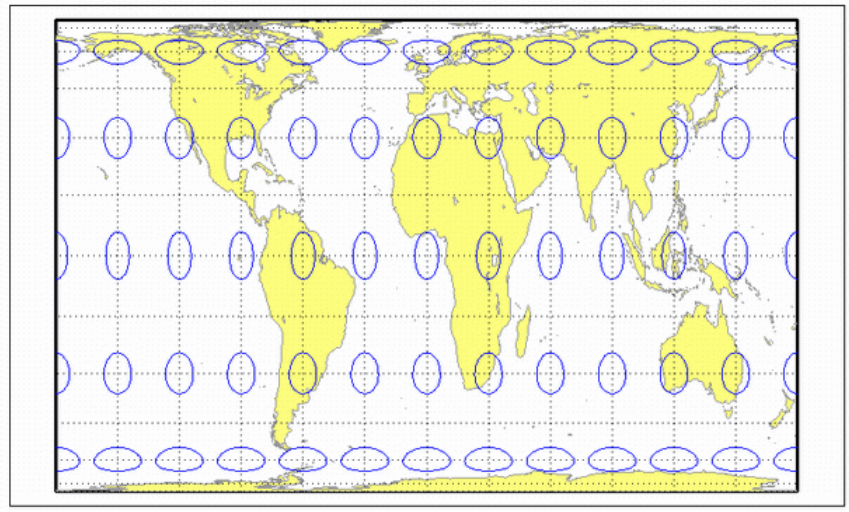
Remarks

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;
```

Gall Orthographic Projection



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.

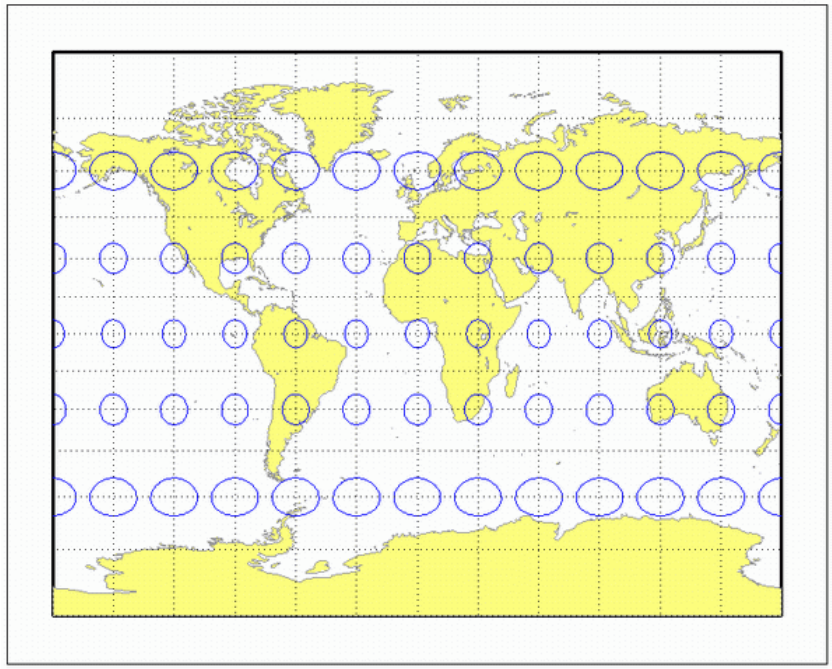
Limitations

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;
```

Gall Stereographic Projection



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

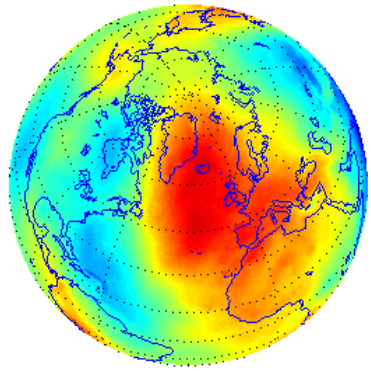
The globe requires no standard parallels.

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.

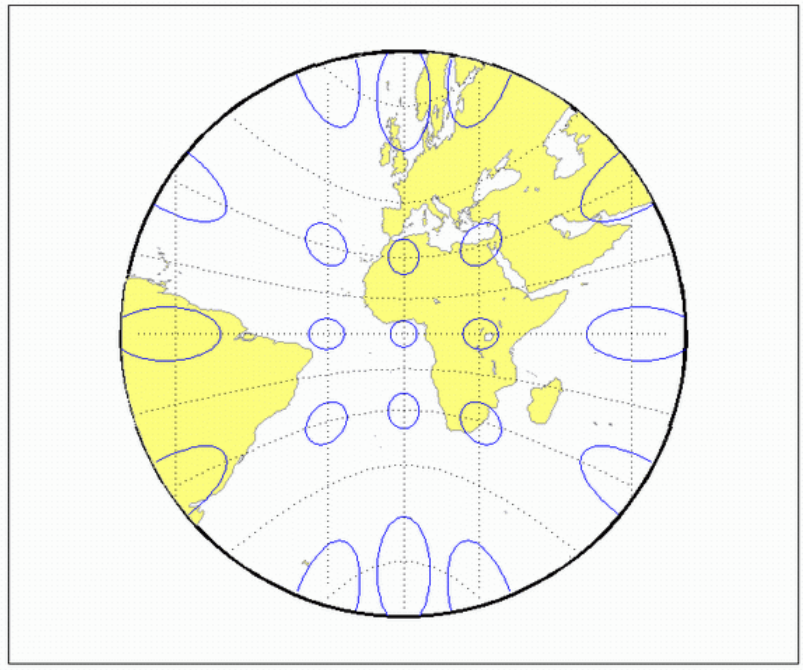
Limitations

This projection is available only on the sphere. Data greater than 65º 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.

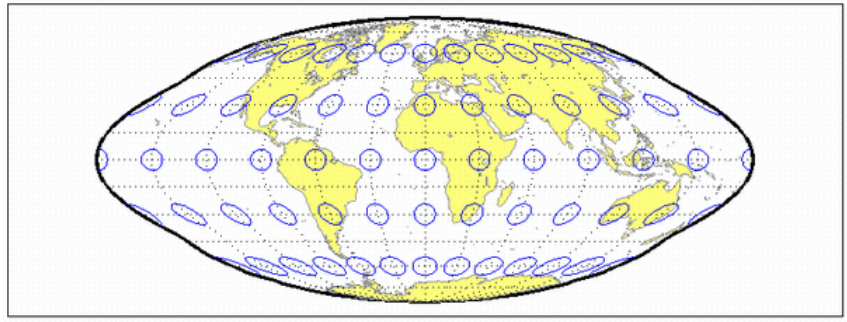
Limitations

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;
```



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.

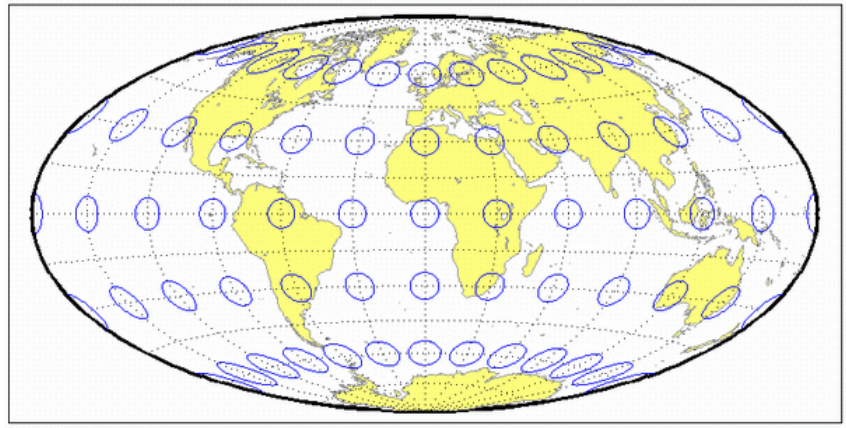
Remarks

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;
```

Hammer Projection



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.

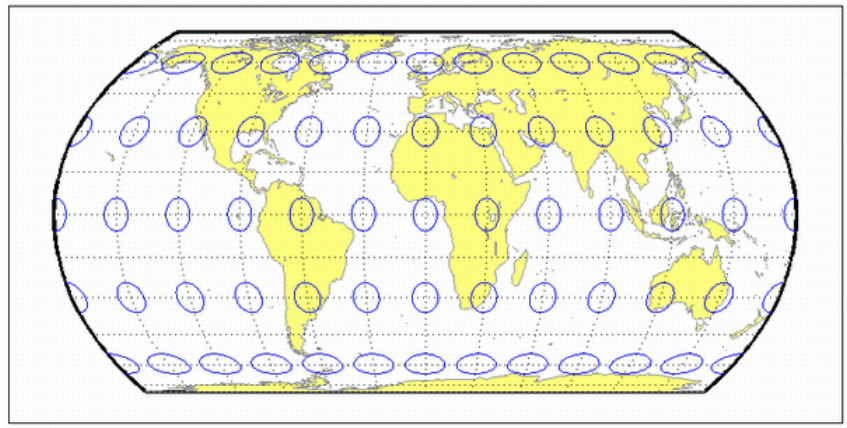
Remarks

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;
```

Hatano Asymmetrical Equal-Area 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.

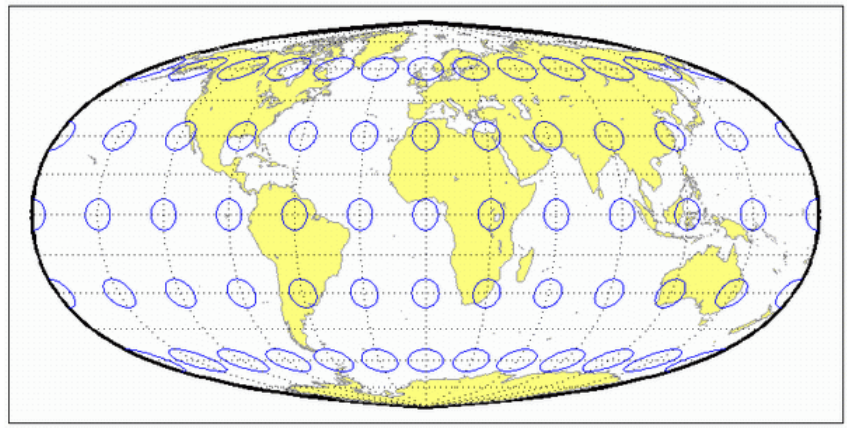
Remarks

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 V Projection



Classification

Pseudocylindrical

Syntax

```
kavrsky6
```

Graticule

Central Meridian: Straight line half the length of the Equator.

Meridians: Sine curves (60º 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.

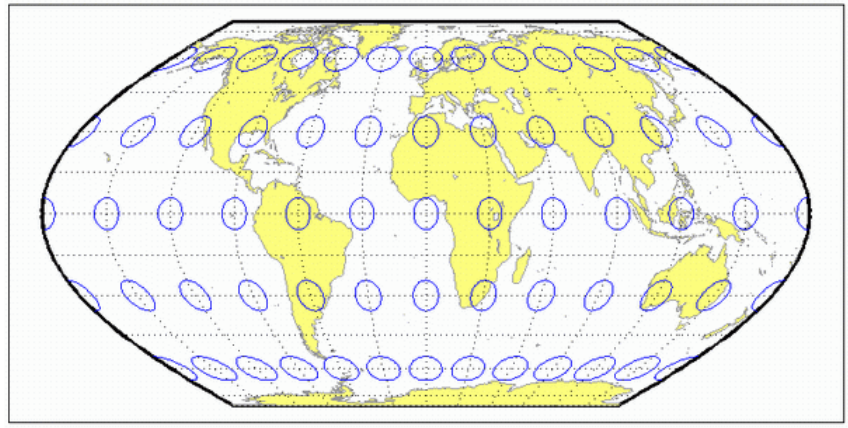
Remarks

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;
```

Kavraisky VI Projection



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.

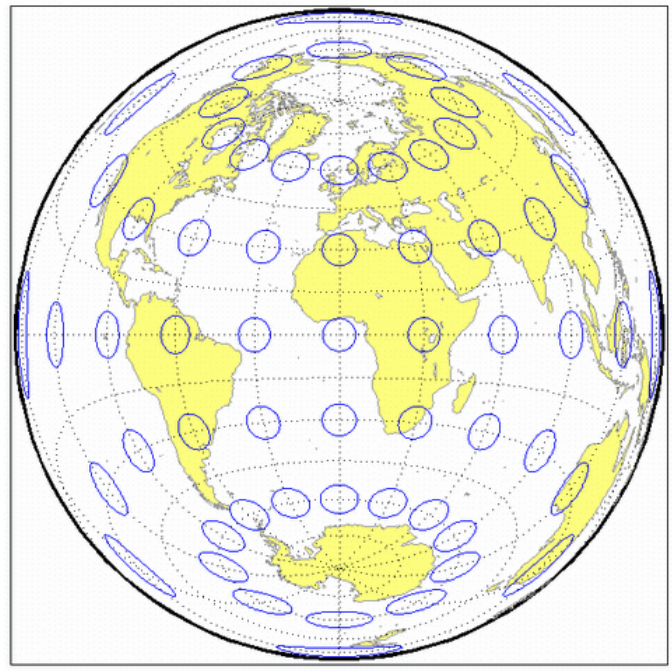
Remarks

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.

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 Azimuthal Equal-Area 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

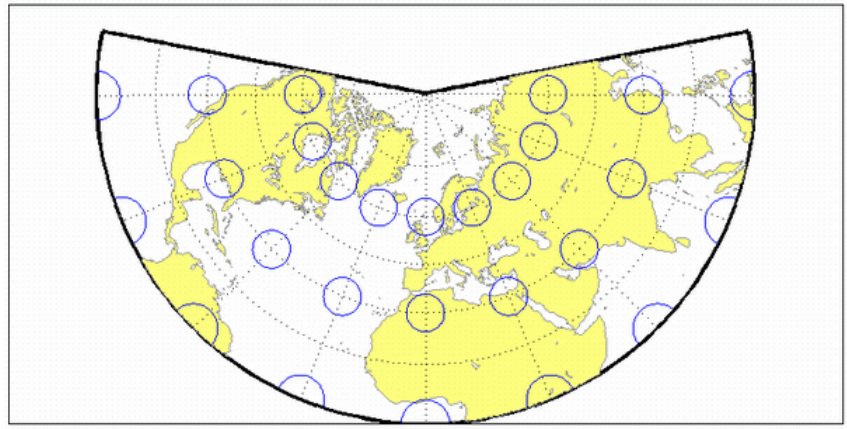
This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.

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

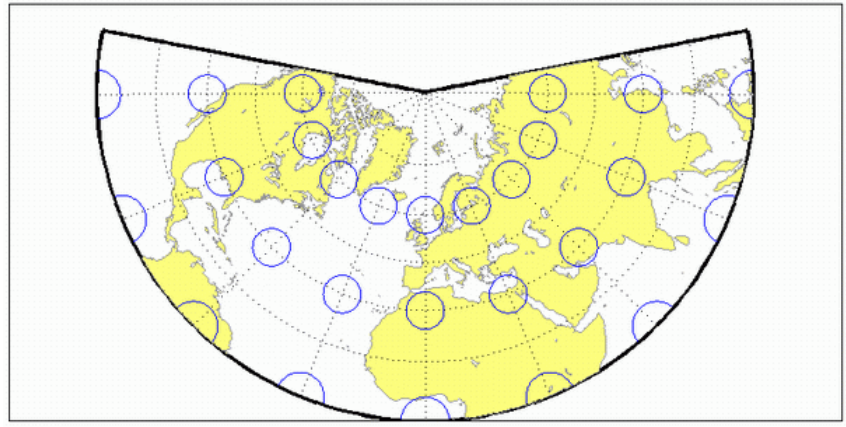
This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.

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.

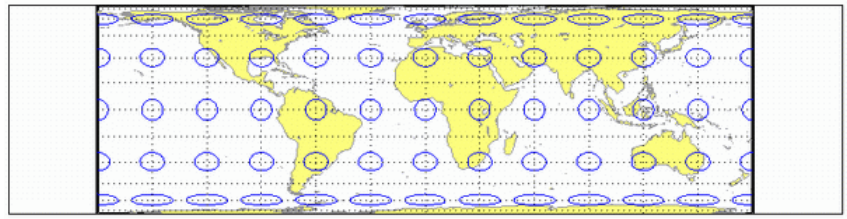
Remarks

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;
```

Lambert Equal-Area Cylindrical 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

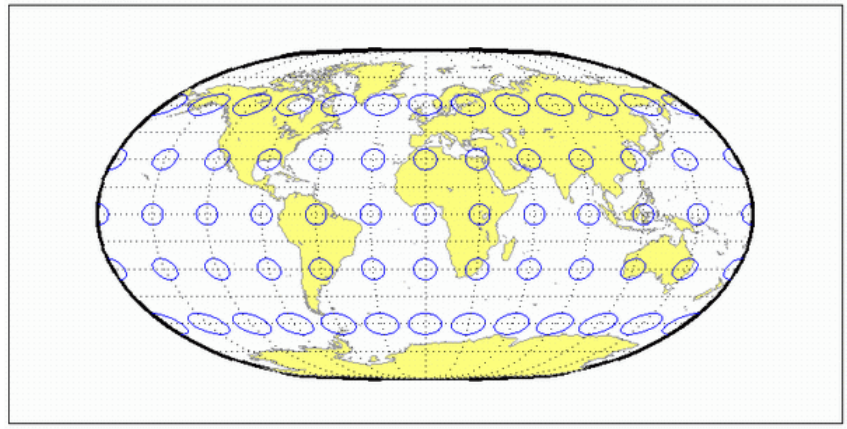
This projection was presented by Karl Siemon in 1935 and independently by Waldo R. Tobler in 1966. The Bordone Oval projection of 1520 was very similar to the Equator-centered Loximuthal.

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'.

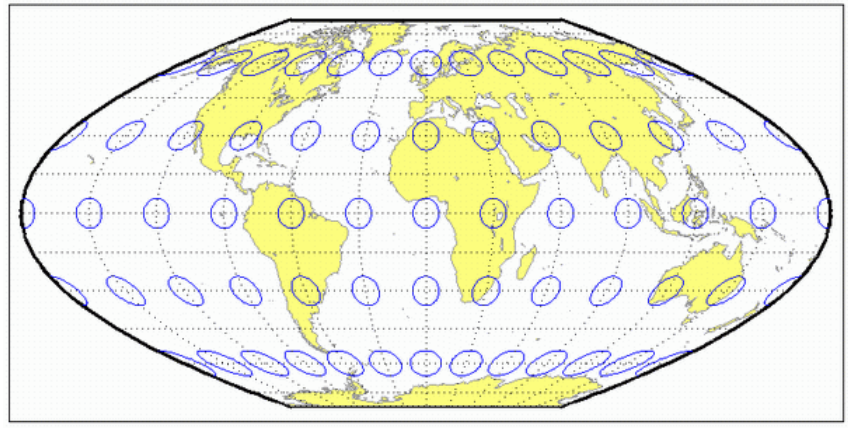
Remarks

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 Parabolic Projection



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'.

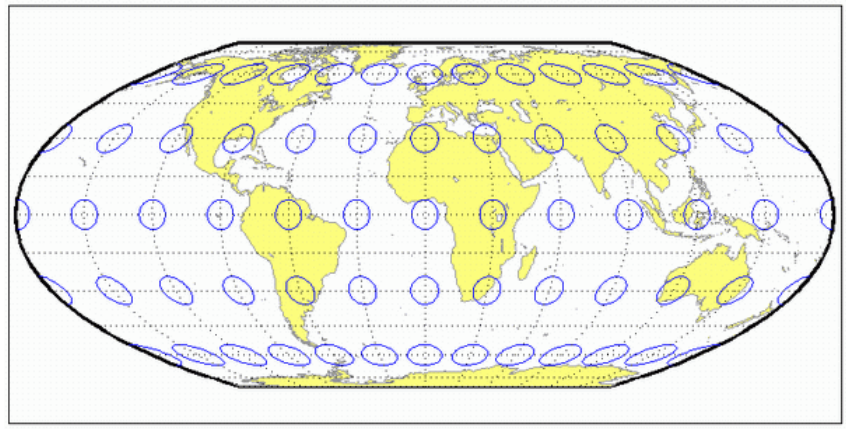
Remarks

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'.

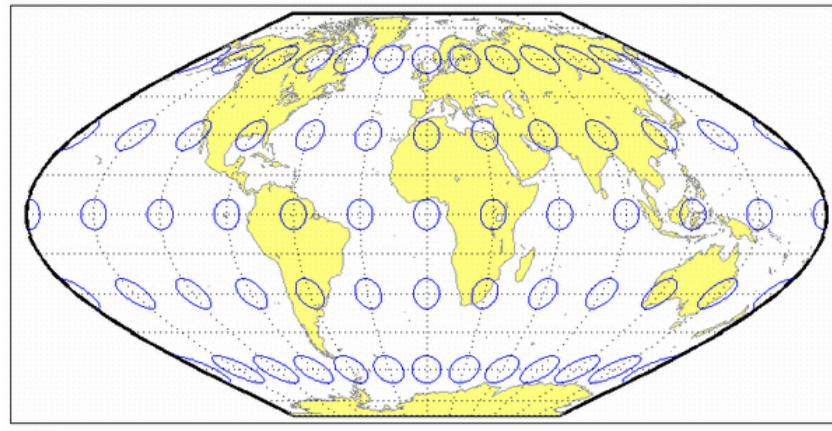
Remarks

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;
```

McBryde-Thomas Flat-Polar Sinusoidal Projection



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,

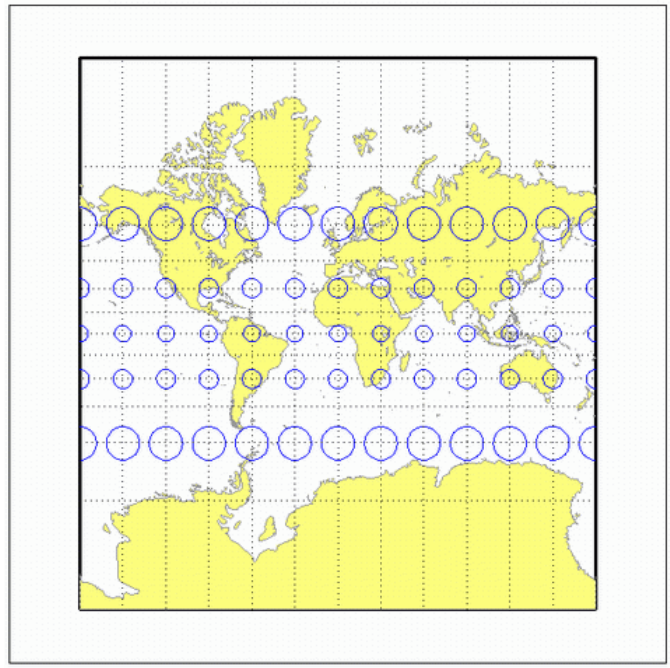
called the Wright projection, after Edward Wright, who developed the mathematics behind the projection in 1599.

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;
```



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

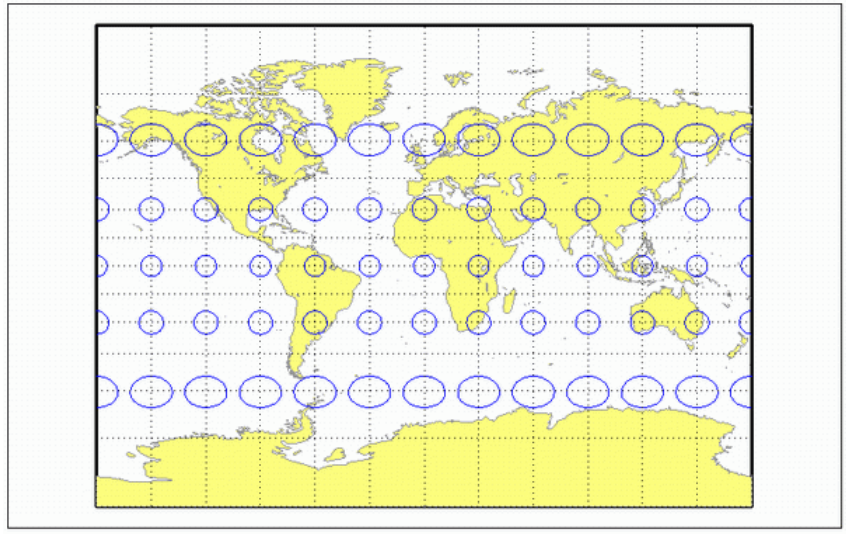
This projection was presented by Osborn Maitland Miller of the American Geographical Society in 1942. It is often used in place of the Mercator projection for atlas maps of the world, for which it is somewhat more appropriate.

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'.

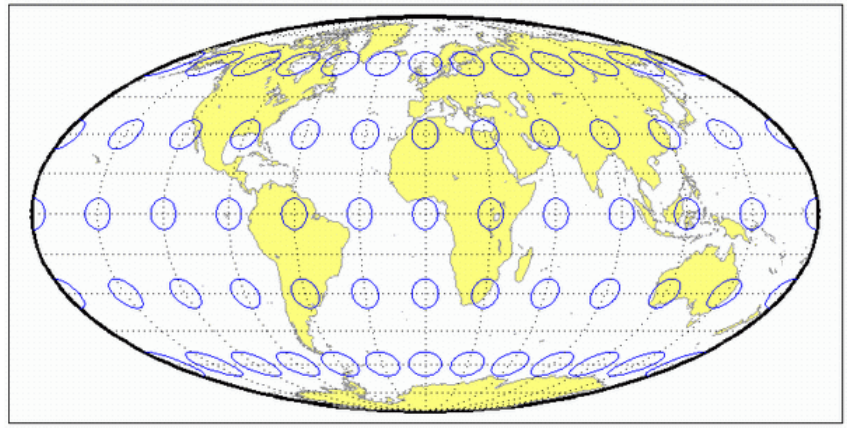
Remarks

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;
```

Mollweide 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.

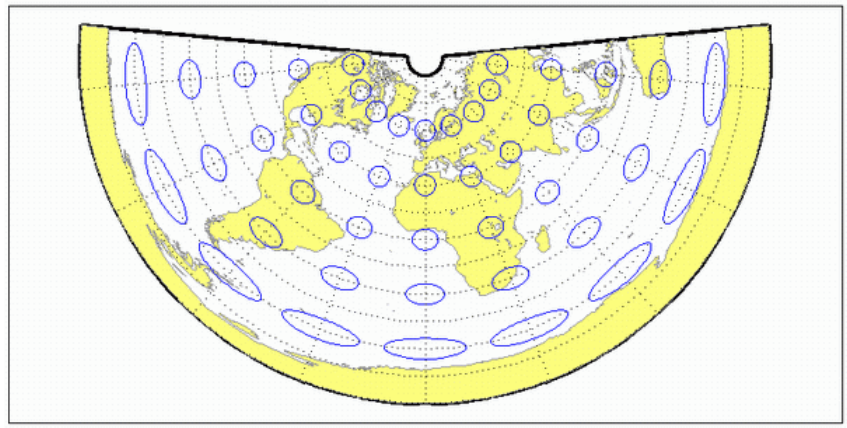
Limitations

This projection is available only for the sphere. Longitude data greater than 135º 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 I Conic Projection



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.

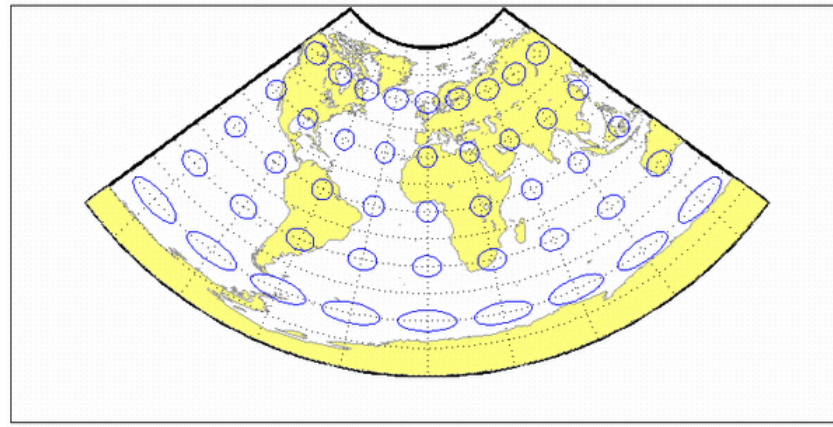
Limitations

This projection is available only for the sphere. Longitude data greater than 135º 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;
```

Murdoch III Minimum Error Conic 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

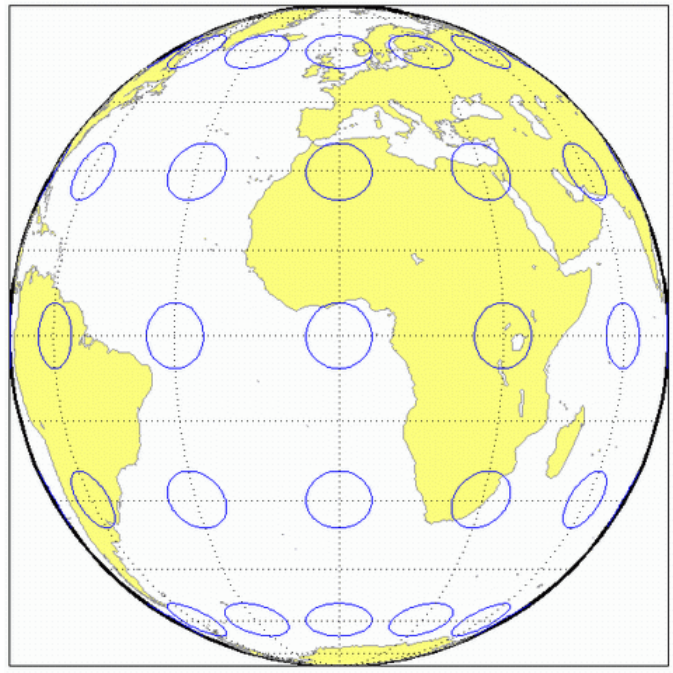
This projection appears to have been developed by the Egyptians and Greeks by the second century B.C.

Limitations

This projection is available only for the sphere. Data greater than 89º 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;
```



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.

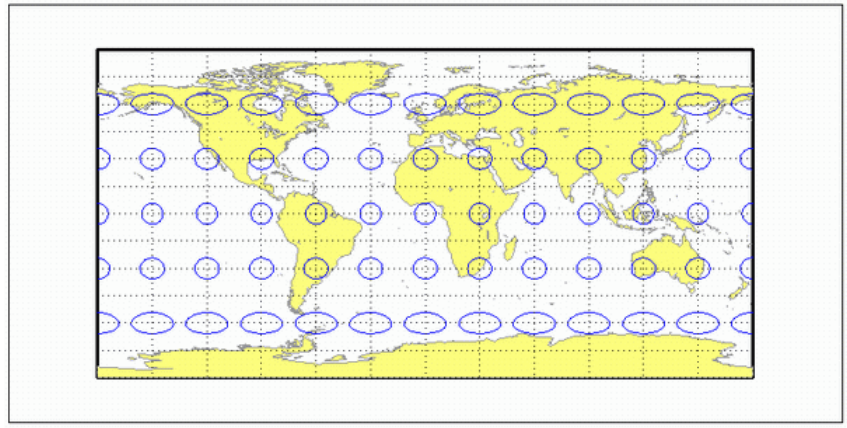
Remarks

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;
```

Plate Carrée Projection



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.

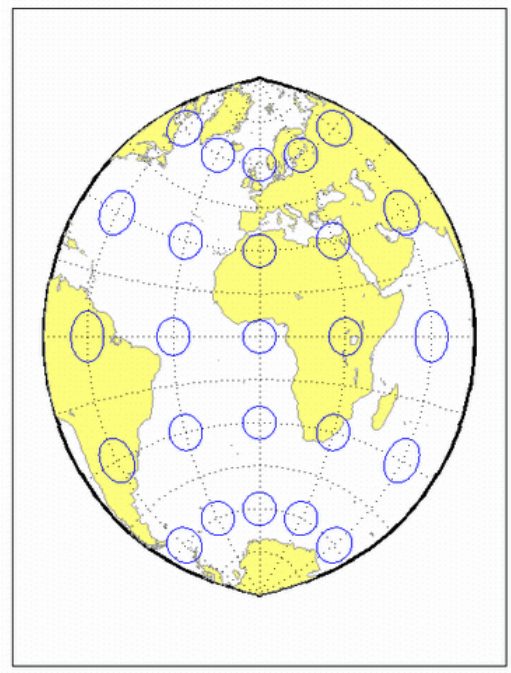
Limitations

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

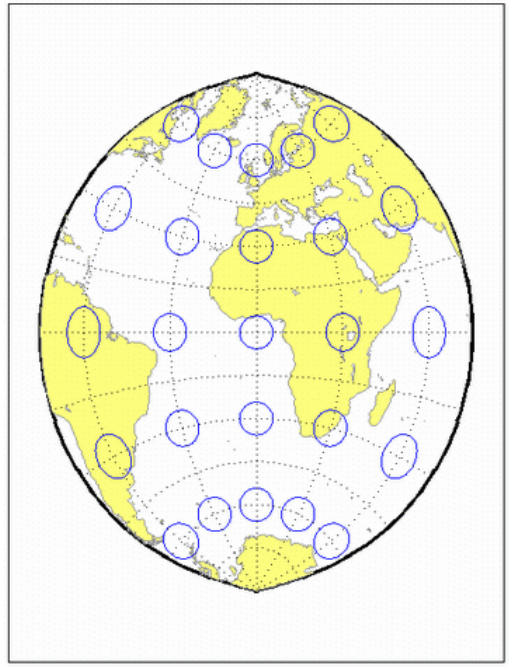
This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.

Limitations

Longitude data greater than 75º east or west of the central meridian is trimmed.

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`

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.

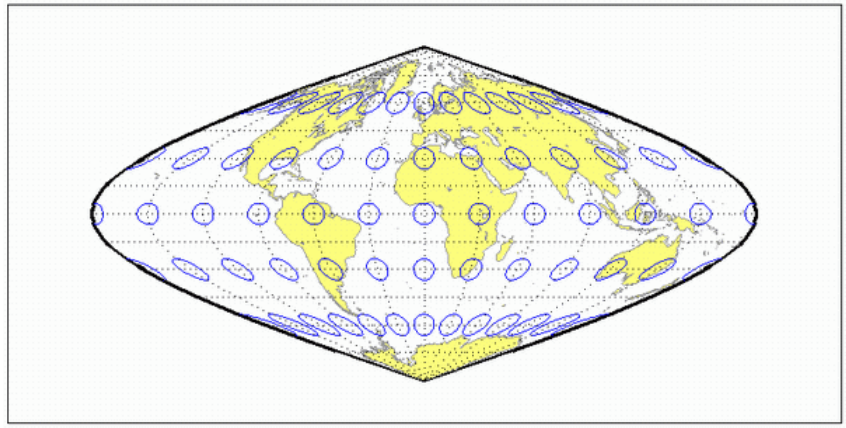
Limitations

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;
```

Putnins P5 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 0^{o}.

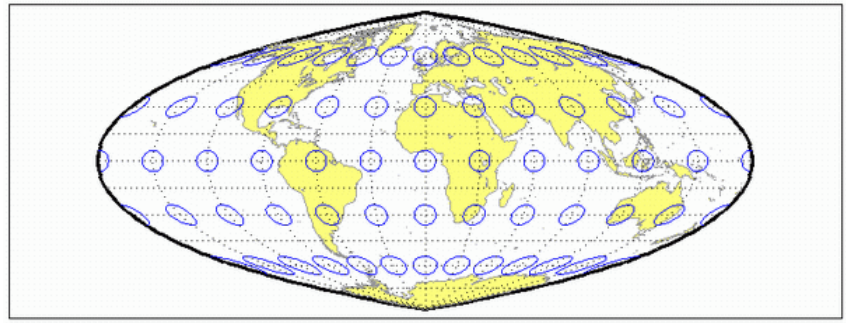
Remarks

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;
```

Quartic Authalic Projection



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*.

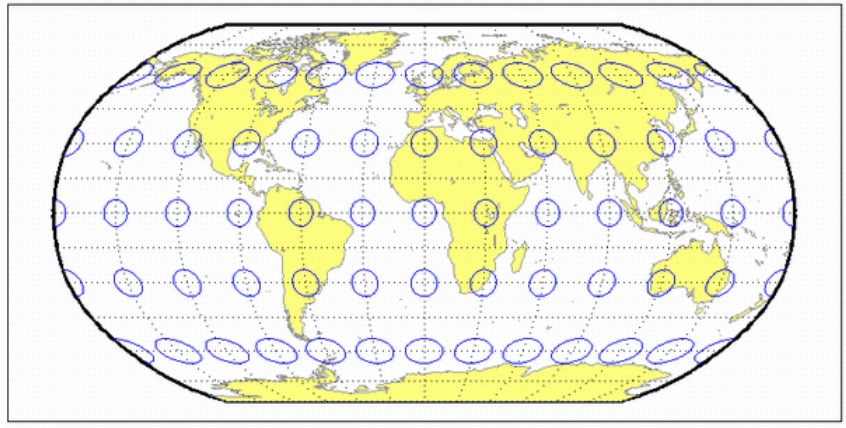
Limitations

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;
```



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.

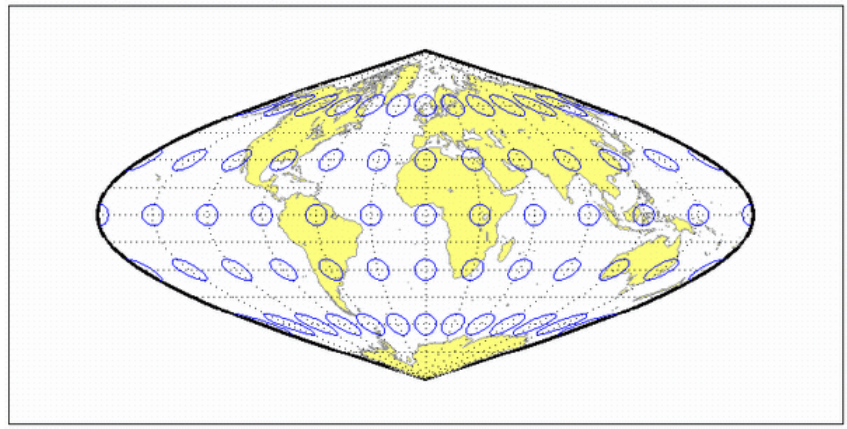
Remarks

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;
```

Sinusoidal 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

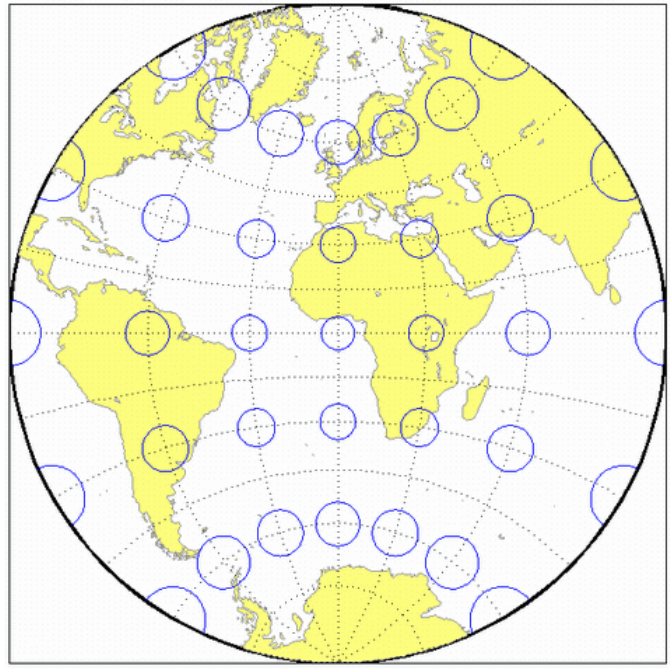
The polar aspect of this projection appears to have been developed by the Egyptians and Greeks by the second century B.C.

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.

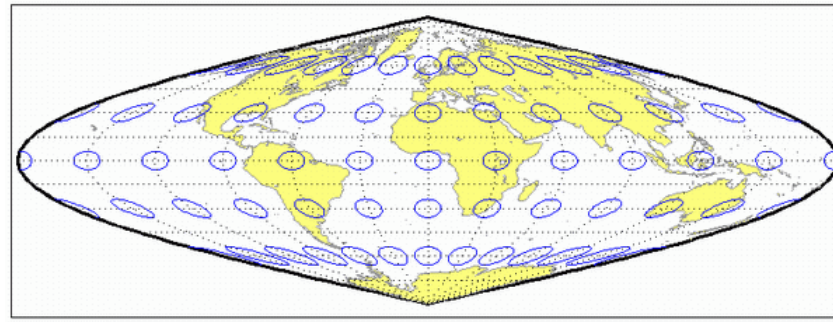
Parallels There are no standard parallels for this projection.

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 projection. It is not equal area, equidistant, or perspective.

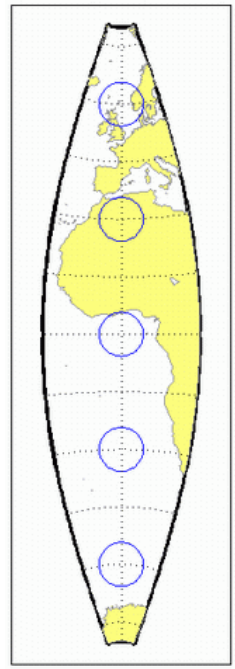
The scale is constant along the central meridian, and increases to the east and west. The scale at the central meridian can be set true to scale, or reduced slightly to render the mean scale of the overall map more correctly.

Remarks

The uniformity of scale along its central 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

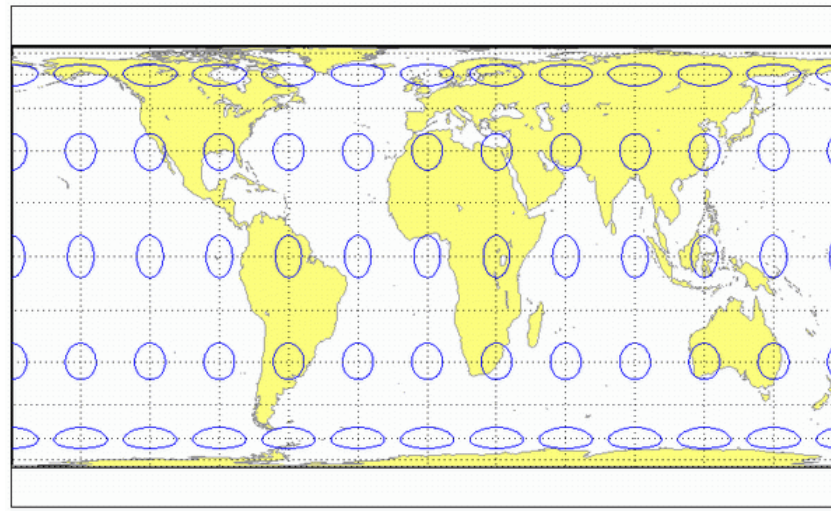
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 37º24'.

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.

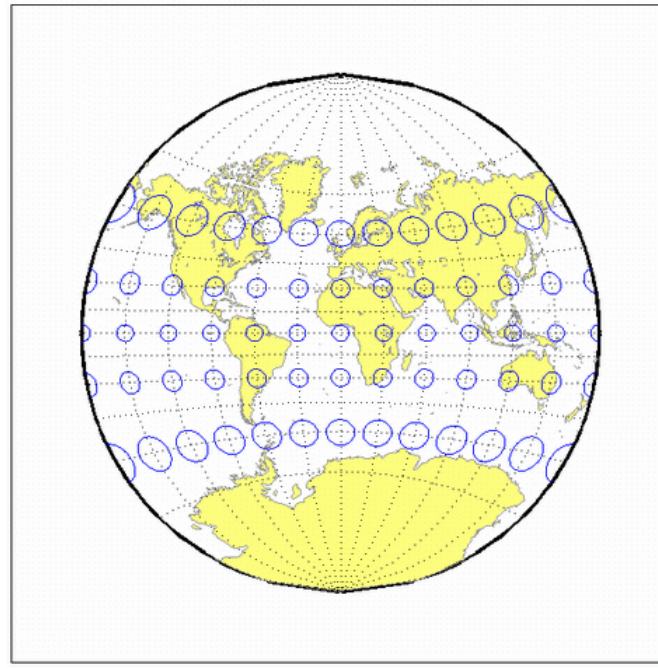
Parallels There are no standard parallels for this projection.

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.

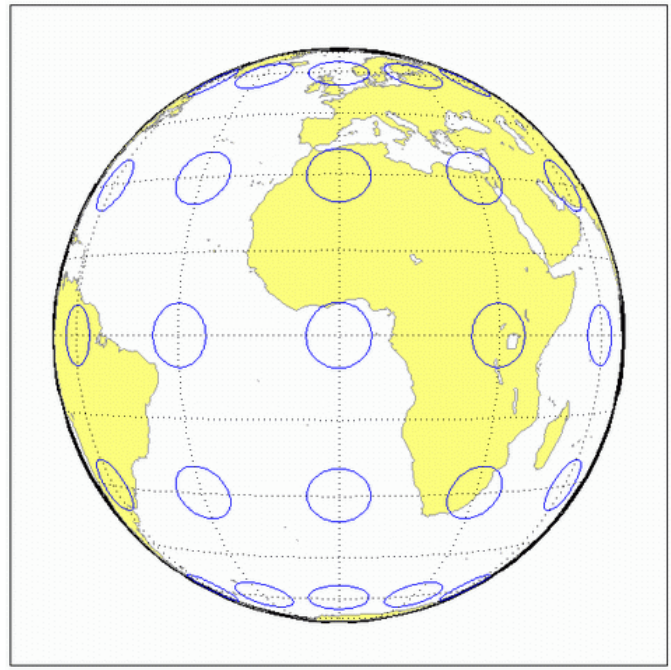
This projection requires a view altitude parameter, which specifies the observer's altitude above the origin point. Because this parameter is unique to this projection and because the projection does not need any standard parallels, a special workaround is used. Rather than add an extra map axes property 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

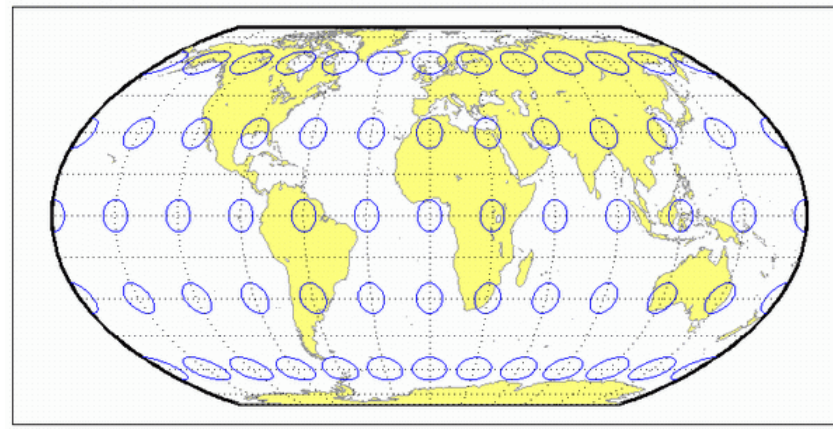
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 42º59'.

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

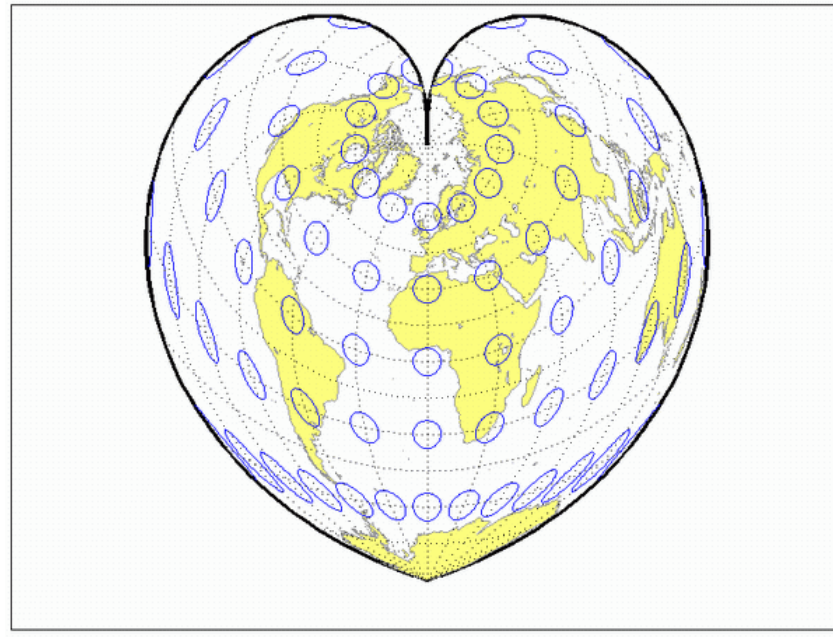
The standard parallel for this projection is set to 90º N.

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 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 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 0º.

Remarks

This is the transverse aspect of the Central Cylindrical projection discussed by J. Wetch in the early 19th century.

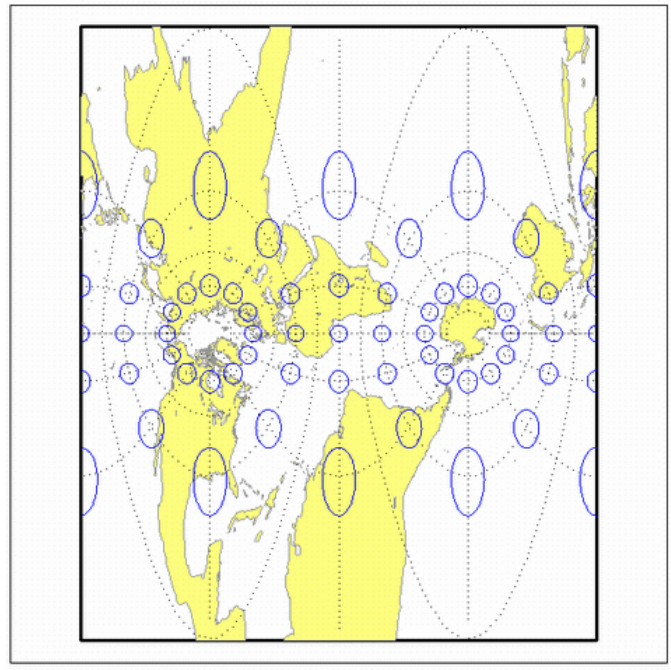
Limitations

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 75º 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.

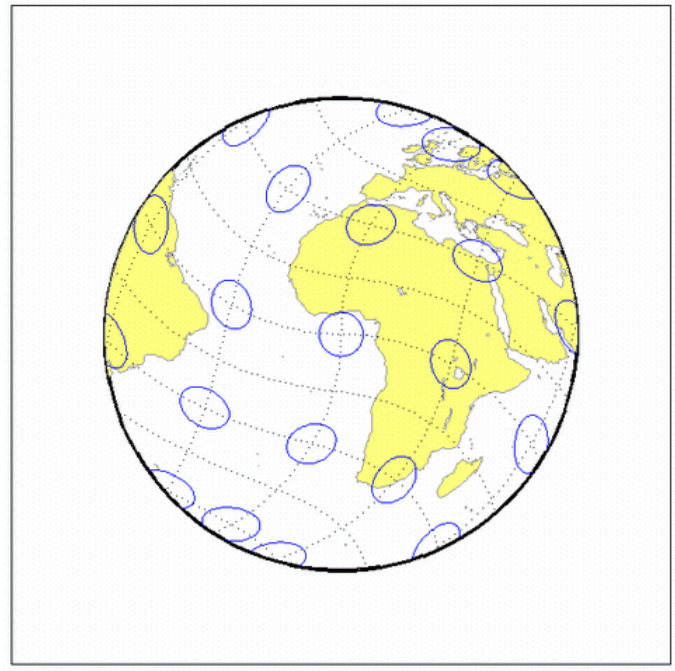
Parallels There are no standard parallels for azimuthal projections.

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

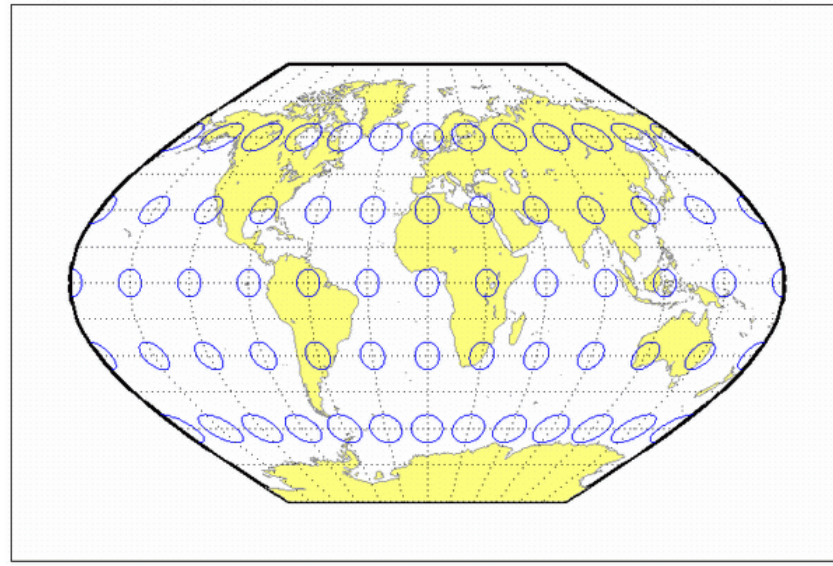
This projection was developed by Oswald Winkel in 1914. Its limiting form is the Eckert V when a standard parallel of 0º is chosen.

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;
```



Winkel I Projection

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 12, "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.

A

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.

B

Examples

Use this list to find examples in the documentation.

Your First Maps

Understanding Vector Geodata

Understanding Raster Geodata

Combining Vector and Raster Geodata

Geolocated Data Grids

Exporting Vector Geodata

Understanding Geospatial Geometry

Creating and Viewing Maps

Making Three-Dimensional Maps

Customizing and Printing Maps

Using Colormaps and Colorbars

Vector Data Manipulation

Raster Data Manipulation

Projections and Transformations

Web Map Service Maps

Navigation

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z