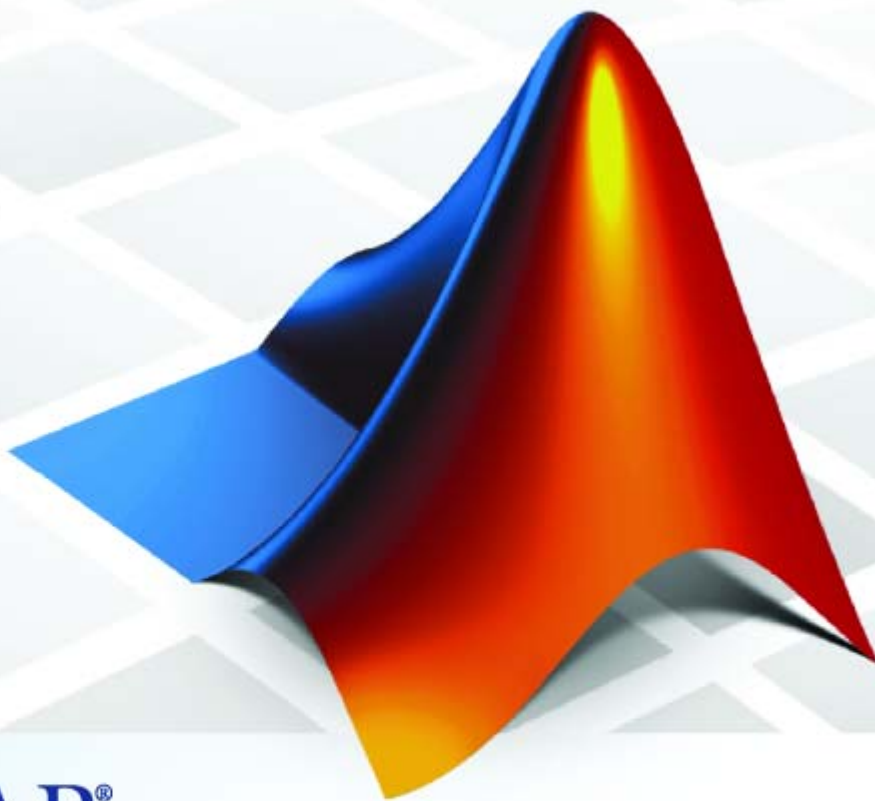


# Mapping Toolbox 2

## User's Guide



MATLAB®

## How to Contact The MathWorks



www.mathworks.com  
comp.soft-sys.matlab  
www.mathworks.com/contact\_TS.html

Web  
Newsgroup  
Technical Support



suggest@mathworks.com  
bugs@mathworks.com  
doc@mathworks.com  
service@mathworks.com  
info@mathworks.com

Product enhancement suggestions  
Bug reports  
Documentation error reports  
Order status, license renewals, passcodes  
Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*Mapping Toolbox User's Guide*

© COPYRIGHT 1997–2007 by The MathWorks, Inc.

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

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

### Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

### Patents

The MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://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)



## Getting Started

### 1

<b>What Is Mapping Toolbox?</b> .....	<b>1-2</b>
<b>Dedication and Acknowledgment</b> .....	<b>1-3</b>
<b>Your First Maps</b> .....	<b>1-4</b>
See the World .....	<b>1-4</b>
Tour Boston with the Map Viewer .....	<b>1-9</b>
<b>Getting More Help</b> .....	<b>1-26</b>
Sources of Help for Mapping Toolbox .....	<b>1-26</b>
Consulting Release Notes .....	<b>1-27</b>
<b>Mapping Toolbox Demos and Data</b> .....	<b>1-28</b>
Available Demos .....	<b>1-28</b>
Locating Map Data .....	<b>1-29</b>

## Understanding Map Data

### 2

<b>Maps and Map Data</b> .....	<b>2-2</b>
What Is a Map? .....	<b>2-2</b>
What Is Geospatial Data? .....	<b>2-2</b>
<b>Types of Map Data Handled by Mapping Toolbox</b> .....	<b>2-4</b>
Vector Geodata .....	<b>2-4</b>
Raster Geodata .....	<b>2-7</b>
Combining Vector and Raster Geodata .....	<b>2-10</b>
<b>Understanding Vector Data</b> .....	<b>2-13</b>

Points, Lines, Polygons .....	2-13
Segments Versus Polygons .....	2-15
Mapping Toolbox Geographic Data Structures .....	2-16
Selecting Data to Read with the shaperead Function .....	2-21
<b>Understanding Raster Data .....</b>	<b>2-27</b>
Georeferencing Raster Data .....	2-27
Regular Data Grids .....	2-29
Geolocated Data Grids .....	2-38
<b>Reading and Writing Geospatial Data .....</b>	<b>2-46</b>
Functions that Read and Write Geospatial Data .....	2-46
Exporting Vector Geodata .....	2-51
Functions That Read and Write Files in Compressed Formats .....	2-61

## Understanding Geospatial Geometry

### 3

<b>Understanding Spherical Coordinates .....</b>	<b>3-2</b>
Spheres, Spheroids, and Geoids .....	3-2
Geoid and Ellipsoid .....	3-2
The Ellipsoid Vector .....	3-4
<b>Understanding Latitude and Longitude .....</b>	<b>3-11</b>
<b>Understanding Angles, Directions, and Distances .....</b>	<b>3-14</b>
Positions, Azimuths, Headings, Distances, Length, and Ranges .....	3-14
Working with Length and Distance Units .....	3-15
Working with Angles: Units and Representations .....	3-18
Working with Distances on the Sphere .....	3-23
Angles as Binary and Formatted Numbers .....	3-27
<b>Understanding Map Projections .....</b>	<b>3-29</b>
What Is a Map Projection? .....	3-29
Forward and Inverse Projection .....	3-30
Projection Distortions .....	3-30

<b>Great Circles, Rhumb Lines, and Small Circles</b> .....	<b>3-32</b>
Great Circles .....	<b>3-32</b>
Rhumb Lines .....	<b>3-32</b>
Small Circles .....	<b>3-33</b>
<b>Directions and Areas on the Sphere and Spheroid</b> ....	<b>3-38</b>
About Azimuths .....	<b>3-38</b>
Reckoning — The Forward Problem .....	<b>3-38</b>
Distance, Azimuth, and Back-Azimuth (the Inverse Problem) .....	<b>3-41</b>
Measuring Area of Spherical Quadrangles .....	<b>3-44</b>
<b>Planetary Almanac Data</b> .....	<b>3-46</b>

## Creating and Viewing Maps

# 4

<b>Introduction to Mapping Graphics</b> .....	<b>4-2</b>
<b>Using worldmap and usamap</b> .....	<b>4-4</b>
Continent, Country, Region, and State Maps Made Easy ..	<b>4-4</b>
Using worldmap .....	<b>4-5</b>
Using usamap .....	<b>4-7</b>
<b>Axes for Drawing Maps</b> .....	<b>4-12</b>
What Is a Map Axes? .....	<b>4-12</b>
Using axesm .....	<b>4-13</b>
Accessing and Manipulating Map Axes Properties .....	<b>4-13</b>
Switching Between Projections .....	<b>4-18</b>
Projected and Unprojected Graphic Objects .....	<b>4-22</b>
<b>Controlling Map Frames and Grids</b> .....	<b>4-31</b>
The Map Frame .....	<b>4-31</b>
The Map Grid .....	<b>4-38</b>
<b>Displaying Vector Data with Mapping Toolbox</b>	
<b>Functions</b> .....	<b>4-43</b>
Programming and Scripting Map Construction .....	<b>4-43</b>

Displaying Vector Maps as Lines .....	4-43
Displaying Vector Maps as Lines or Patches .....	4-46
<b>Displaying Data Grids .....</b>	<b>4-53</b>
Types of Data Grids and Raster Display Functions .....	4-53
Fitting Gridded Data to the Graticule .....	4-54
Using Raster Data to Create 3-D Displays .....	4-57
<b>Interacting with Displayed Maps .....</b>	<b>4-61</b>
Picking Locations Interactively .....	4-61
Defining Small Circles and Tracks Interactively .....	4-63
Working with Objects by Name .....	4-66

## Making Three-Dimensional Maps

# 5

<b>Sources of Terrain Data .....</b>	<b>5-2</b>
Digital Terrain Elevation Data from NGA .....	5-2
Digital Elevation Model Files from USGS .....	5-3
Determining What Elevation Data Exists for a Region ...	5-3
<b>Reading Elevation Data Interactively .....</b>	<b>5-13</b>
Extracting DEM Data with demdataui .....	5-13
<b>Determining and Visualizing Visibility Across Terrain .....</b>	<b>5-19</b>
Computing Line of Sight with los2 .....	5-19
<b>Shading and Lighting Terrain Maps .....</b>	<b>5-22</b>
Lighting a Terrain Map Constructed from a DTED File ..	5-22
Lighting a Global Terrain Map with lightm and lightmui ..	5-25
Surface Relief Shading .....	5-29
Colored Surface Shaded Relief .....	5-33
Relief Mapping with Light Objects .....	5-36
<b>Draping Data on Elevation Maps .....</b>	<b>5-40</b>
Draping Geoid Heights over Topography .....	5-40
Draping Data over Terrain with Different Gridding .....	5-43



<b>Working with the Globe Display</b> .....	<b>5-49</b>
What Is the Globe Display? .....	<b>5-49</b>
The Globe Display Compared with the Orthographic Projection .....	<b>5-50</b>
Using Opacity and Transparency in Globe Displays .....	<b>5-52</b>
Over-the-Horizon 3-D Views Using Camera Positioning Functions .....	<b>5-55</b>
Displaying a Rotating Globe .....	<b>5-57</b>

## Customizing and Printing Maps

# 6

<b>Inset Maps</b> .....	<b>6-2</b>
<b>Graphic Scales</b> .....	<b>6-8</b>
<b>North Arrows</b> .....	<b>6-14</b>
<b>Thematic Maps</b> .....	<b>6-17</b>
What Is a Thematic Map? .....	<b>6-17</b>
Choropleth Maps .....	<b>6-18</b>
Special Thematic Mapping Functions .....	<b>6-23</b>
<b>Using Cartesian MATLAB Display Functions</b> .....	<b>6-28</b>
Adding MATLAB Graphic Objects to Map Axes .....	<b>6-28</b>
Example 1: Triangulating Data Points .....	<b>6-28</b>
Example 2: Constructing Quiver Maps .....	<b>6-30</b>
<b>Using Colormaps and Colorbars</b> .....	<b>6-34</b>
Colormap for Terrain Data .....	<b>6-34</b>
Contour Colormaps .....	<b>6-37</b>
Colormaps for Political Maps .....	<b>6-39</b>
Labeling Colorbars .....	<b>6-43</b>
Editing Colorbars .....	<b>6-44</b>
<b>Printing Maps to Scale</b> .....	<b>6-45</b>

<b>Manipulating Vector Geodata</b> .....	<b>7-2</b>
Repackaging Vector Objects .....	<b>7-2</b>
Matching Line Segments .....	<b>7-4</b>
Geographic Interpolation of Vectors .....	<b>7-5</b>
Vector Intersections .....	<b>7-8</b>
Polygon Area .....	<b>7-11</b>
Overlaying Polygons with Set Logic .....	<b>7-12</b>
Cutting Polygons at the Date Line .....	<b>7-17</b>
Building Buffer Zones .....	<b>7-19</b>
Trimming Vector Data to a Rectangular Region .....	<b>7-21</b>
Trimming Vector Data to an Arbitrary Region .....	<b>7-24</b>
Simplifying Vector Coordinate Data .....	<b>7-25</b>
<b>Manipulating Raster Geodata</b> .....	<b>7-31</b>
Vector-to-Raster Data Conversion .....	<b>7-31</b>
Data Grids as Logical Variables .....	<b>7-39</b>
Data Grid Values Along a Path .....	<b>7-42</b>
Data Grid Gradient, Slope, and Aspect .....	<b>7-45</b>

## Using Map Projections and Coordinate Systems

<b>What Is a Map Projection?</b> .....	<b>8-3</b>
<b>Quantitative Properties of Map Projections</b> .....	<b>8-4</b>
<b>The Three Main Families of Map Projections</b> .....	<b>8-6</b>
Unwrapping the Sphere to a Plane .....	<b>8-6</b>
Cylindrical Projections .....	<b>8-6</b>
Conic Projections .....	<b>8-8</b>
Azimuthal Projections .....	<b>8-9</b>
<b>Projection Aspect</b> .....	<b>8-11</b>
The Orientation Vector .....	<b>8-11</b>

<b>Projection Parameters</b> .....	<b>8-19</b>
Projection Characteristics Maps Can Have .....	<b>8-19</b>
<b>Visualizing and Quantifying Projection Distortions</b> ...	<b>8-28</b>
Displays of Spatial Error in Maps .....	<b>8-28</b>
Quantifying Map Distortions at Point Locations .....	<b>8-32</b>
<b>Accessing, Computing, and Inverting Map Projection</b>	
<b>Data</b> .....	<b>8-38</b>
Accessing Projected Coordinate Data .....	<b>8-38</b>
Projecting Coordinates Without a Map Axes .....	<b>8-40</b>
Inverse Map Projection .....	<b>8-42</b>
Coordinate Transformations .....	<b>8-46</b>
<b>Working with the UTM System</b> .....	<b>8-52</b>
What Is the Universal Transverse Mercator System? ....	<b>8-52</b>
Understanding UTM Parameters .....	<b>8-53</b>
Setting UTM Parameters with a GUI .....	<b>8-55</b>
Working in UTM Without a Map Axes .....	<b>8-60</b>
Mapping Across UTM Zones .....	<b>8-61</b>
<b>Summary and Guide to Projections</b> .....	<b>8-64</b>

## Mapping Applications

# 9

<b>Geographic Statistics</b> .....	<b>9-2</b>
Statistics for Point Locations on a Sphere .....	<b>9-2</b>
Geographic Means .....	<b>9-2</b>
Geographic Standard Deviation .....	<b>9-4</b>
Equal-Areas in Geographic Statistics .....	<b>9-7</b>
<b>Navigation</b> .....	<b>9-11</b>
What Is Navigation? .....	<b>9-11</b>
Conventions for Navigational Functions .....	<b>9-12</b>
Fixing Position .....	<b>9-13</b>
Planning .....	<b>9-25</b>
Track Laydown – Displaying Navigational Tracks .....	<b>9-29</b>

Dead Reckoning .....	9-31
Drift Correction .....	9-36
Time Zones .....	9-38

## Functions — By Category

# 10

<b>Geospatial Data Import and Access</b> .....	<b>10-2</b>
Standard File Formats .....	10-2
Gridded Terrain and Bathymetry Products .....	10-3
Vector Map Products .....	10-4
Miscellaneous Data Sets .....	10-5
GUIs for Data Import .....	10-5
File Reading Utilities .....	10-5
Ellipsoids, Radii, Areas, and Volumes .....	10-5
<b>Vector Map Data and Geographic Data Structures</b> .....	<b>10-6</b>
Geographic Data Structures .....	10-6
Data Manipulation .....	10-6
Utilities for NaN-Separated Polygons and Lines .....	10-7
<b>Georeferenced Images and Data Grids</b> .....	<b>10-8</b>
Spatial Referencing .....	10-8
Terrain Analysis .....	10-9
Other Analysis/Access .....	10-9
Construction and Modification .....	10-10
Initialization .....	10-10
<b>Map Projections and Coordinates</b> .....	<b>10-11</b>
Available Map Projections .....	10-11
Map Projection Transformations .....	10-12
Map Trimming .....	10-12
Angles, Scales, and Distortions .....	10-12
Visualizing Map Distortions .....	10-13
UTM System .....	10-13
Rotating Coordinates on the Sphere .....	10-13
Trimming and Clipping .....	10-13
<b>Map Display and Interaction</b> .....	<b>10-14</b>

Map Creation and High-Level Display .....	10-15
Vector Symbolization .....	10-15
Displaying Lines and Contours .....	10-15
Displaying Patch Data .....	10-16
Displaying Data Grids .....	10-16
Displaying Light Objects and Lighted Surfaces .....	10-16
Displaying Thematic Maps .....	10-17
Annotating Map Displays .....	10-17
Colormaps for Map Displays .....	10-18
Interactive Map Positions .....	10-18
Interactive Track and Circle Definition .....	10-19
Graphical User Interfaces .....	10-19
Map Object and Projection Properties .....	10-20
Controlling Map Appearance .....	10-21
Clearing Map Displays/Managing Visibility .....	10-21
<b>Geographic Calculations .....</b>	<b>10-22</b>
Geometry of Sphere and Ellipsoid .....	10-22
3-D Coordinates .....	10-24
Ellipsoids and Latitudes .....	10-24
Intersections in the Cartesian Plane .....	10-25
Geographic Statistics .....	10-25
Navigation .....	10-26
<b>Utilities .....</b>	<b>10-26</b>
Angle Conversions .....	10-27
Conversion Factors for Angles and Distances .....	10-27
Data Precision .....	10-28
Distance Conversions .....	10-28
Image Conversion .....	10-28
String Formatters .....	10-28
Longitude or Azimuth Wrapping .....	10-29
<b>Graphical User Interface Functions .....</b>	<b>10-29</b>
Map Definition Tools .....	10-30
Mapping Tools .....	10-30
Display Manipulation Tools .....	10-30
Object Property Tools .....	10-31
Track Tools .....	10-31
Map Data Construction Tools .....	10-32

## Functions — Alphabetical List

---

**11**

## Map Projections — By Category

---

**12**

<b>Cylindrical Projections</b> .....	<b>12-2</b>
<b>Pseudocylindrical Projections</b> .....	<b>12-2</b>
<b>Conic Projections</b> .....	<b>12-4</b>
<b>Polyconic and Pseudoconic Projections</b> .....	<b>12-4</b>
<b>Azimuthal, Pseudoazimuthal, and Modified Azimuthal Projections</b> .....	<b>12-4</b>
<b>UTM and UPS Systems</b> .....	<b>12-5</b>
<b>3-D Globe Display</b> .....	<b>12-5</b>

Glossary

Bibliography

A

Examples

B

Your First Maps .....	B-2
Vector Geodata .....	B-2
Raster Geodata .....	B-2
Combining Vector and Raster Geodata .....	B-2
Understanding Vector Data .....	B-2
Understanding Raster Data .....	B-2
Geolocated Data Grids .....	B-3
Exporting Vector Geodata .....	B-3
Creating and Viewing Maps .....	B-3

<b>Making Three-Dimensional Maps</b> .....	<b>B-3</b>
<b>Making Three-dimensional Maps</b> .....	<b>B-4</b>
<b>Customizing and Printing Maps</b> .....	<b>B-4</b>
<b>Using Cartesian MATLAB Display Functions</b> .....	<b>B-4</b>
<b>Using Colormaps and Colorbars</b> .....	<b>B-5</b>
<b>Vector Data Manipulation</b> .....	<b>B-5</b>
<b>Raster Data Manipulation</b> .....	<b>B-5</b>
<b>Projections and Transformations</b> .....	<b>B-6</b>

---

**Index**



# Getting Started

---

What Is Mapping Toolbox? (p. 1-2)	Executive summary
Dedication and Acknowledgment (p. 1-3)	For Mapping Toolbox
Your First Maps (p. 1-4)	Plotting a map with a single command or very few commands
Getting More Help (p. 1-26)	Finding specific types of help
Mapping Toolbox Demos and Data (p. 1-28)	A set of scripts that apply toolbox functions to sample data

---

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

---

## What Is Mapping Toolbox?

Welcome to Mapping Toolbox for MATLAB®. Mapping Toolbox is a collection of MATLAB functions, user interfaces, sample data sets, and demos that read, write, display, and manipulate geospatial data. With it you can make maps of your own geospatial data or use sample data provided with Mapping Toolbox, such as world coastlines, political boundaries, and topography. The following sections get you started using Mapping Toolbox, and then describe what information this documentation covers and where to find it.

Mapping Toolbox provides a comprehensive set of functions and for building map displays and performing geospatial data analysis in MATLAB. You can create map displays that combine data from multiple modalities and display them in their correct spatial relationships. The toolbox supports standard analyses, such as line-of-sight calculations on terrain data or geographic computations that account for the curvature of the Earth's surface. Most of the functions in Mapping Toolbox are written in the open MATLAB language. This means that you can inspect the algorithms, modify the source code, create your own custom functions, and automate frequently performed tasks.

The toolbox supports key mapping and geospatial data analysis, manipulation, and visualization tasks that are useful in applications such as earth and planetary scientific research, oil and gas exploration, environmental monitoring, insurance risk management, aerospace, defense, and security.

## 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.

Mapping Toolbox 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. The MathWorks, Inc. makes no claims that any of this data is free of defects or errors, or that the representations of geographic features or names are up to date or authoritative.

## Your First Maps

In this section...
“See the World” on page 1-4
“Tour Boston with the Map Viewer” on page 1-9

This section helps you exercise high-level functions and (GUIs) to explore mapping and visualizing geodata. It explores `worldmap` and other functions, and then describes how to use the Map Viewer (`mapview`). You can then use the to identify where to find descriptions of the capabilities you want to learn more about.

### See the World

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

Geospatial data can be voluminous, complex, and difficult to work with. Mapping Toolbox handles many of the details of loading and displaying data for you. Nevertheless, the more you understand about your data and the capabilities of the toolbox, the more interesting applications you will be able to pursue, and the more useful their results will be to you and others.

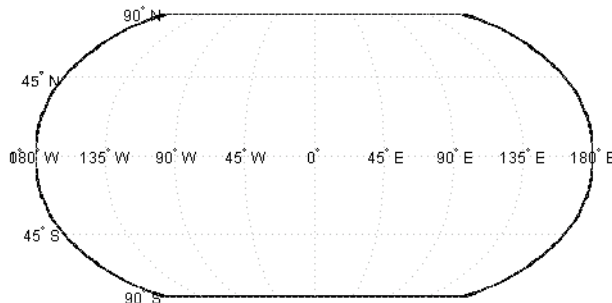
Getting started making world maps with Mapping Toolbox is easy.

**1** In the MATLAB Command Window, type

```
worldmap world
```

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

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



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

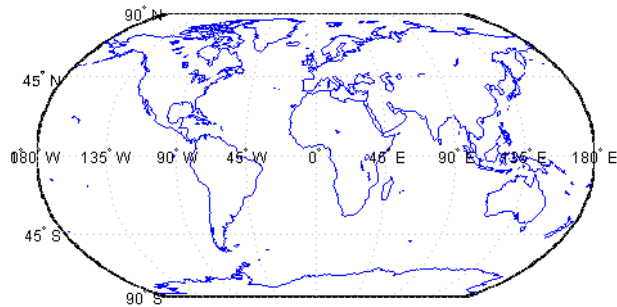
```
whos -file coast.mat
```

Name	Size	Bytes	Class	Attributes
lat	9589x1	76712	double	
long	9589x1	76712	double	

**3** Load and plot the coastlines on the world map:

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

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



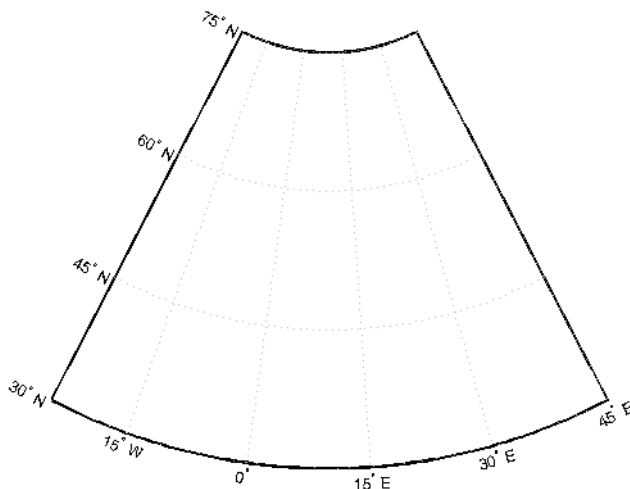
Notice how the world coastlines form distinct polygons, even though only a single vector of latitudes and a corresponding vector of longitudes are provided. The reason is because of NaN separators, which implicitly divide each vector into multiple parts.

```
sum(isnan(lat))  
ans =  
    238
```

lat and long include NaN terminators as well as separators, showing that the coast data set is organized into precisely 238 polygons.

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

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



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

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

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

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

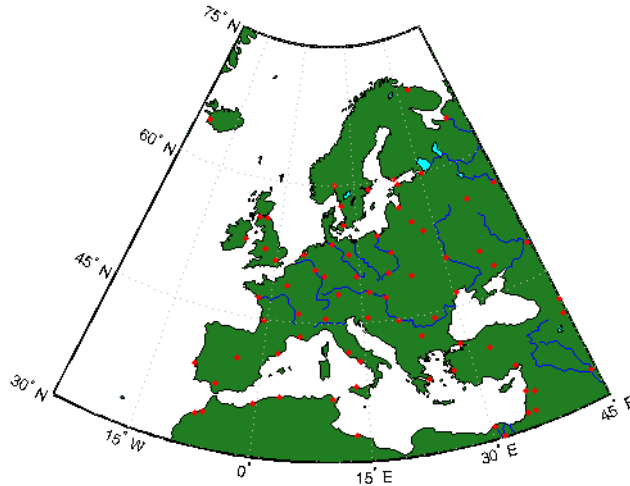
- 6 Now you can inspect the 1-by-1 structure `mstruct` by listing it, using the property editor, or by accessing any field directly. For instance, to see the map projection selected for the map of Europe, type

```
mstruct.mapprojection
ans =
```

```
eqdconic
```

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

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

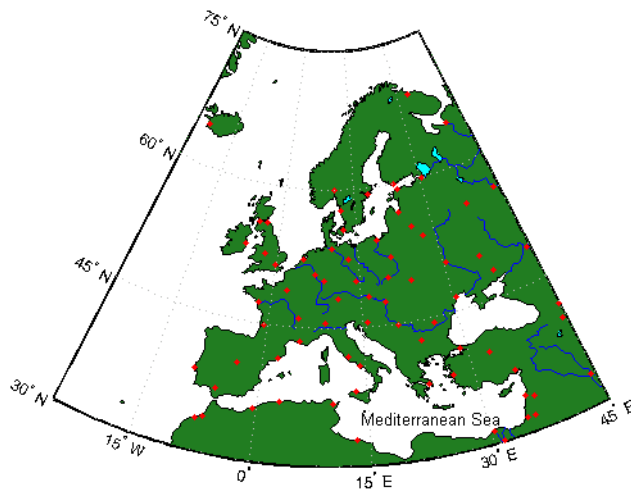


Note how `geoshow` can plot data directly from files onto a map axes without first loading it into the MATLAB workspace.

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

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





Look at the reference documentation for `worldmap` and experiment with its options. To learn more about display properties for map axes and how to control them, see “Accessing and Manipulating Map Axes Properties” on page 4-13. See the reference page for `geoshow` to find out more about its capabilities.

## Tour Boston with the Map Viewer

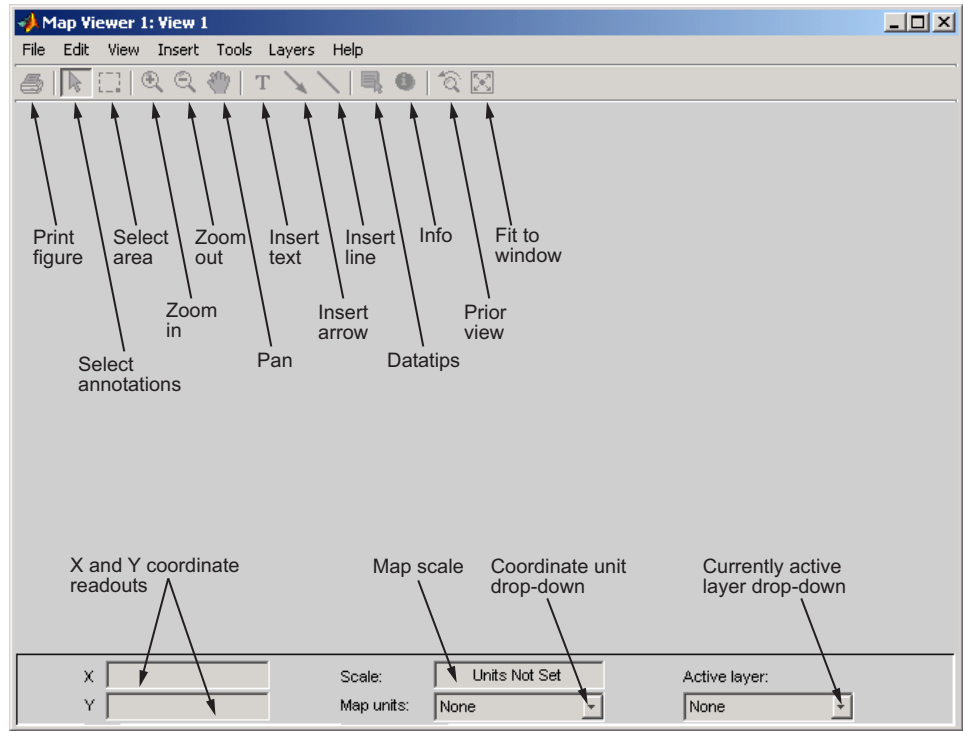
The Map Viewer is an interactive tool for browsing map data. With it you can assemble layers of vector and raster geodata and render them in 2-D. You can import, reorder, symbolize, hide, and delete data layers, identify coordinate locations, list data attributes, and display selected ones as *datatips*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

### A Map Viewer Session

1 You start a Map Viewer session by typing

```
mapview
```

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



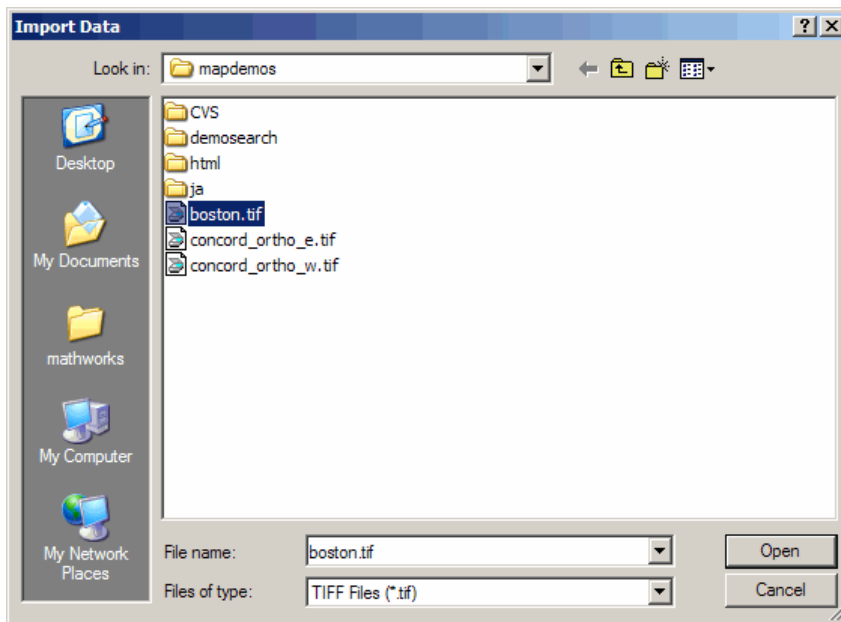
Most of the tool buttons can also be activated from the **Tools** menu.

- 2 For ease in importing data that is furnished with Mapping Toolbox, set your working directory as follows:

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

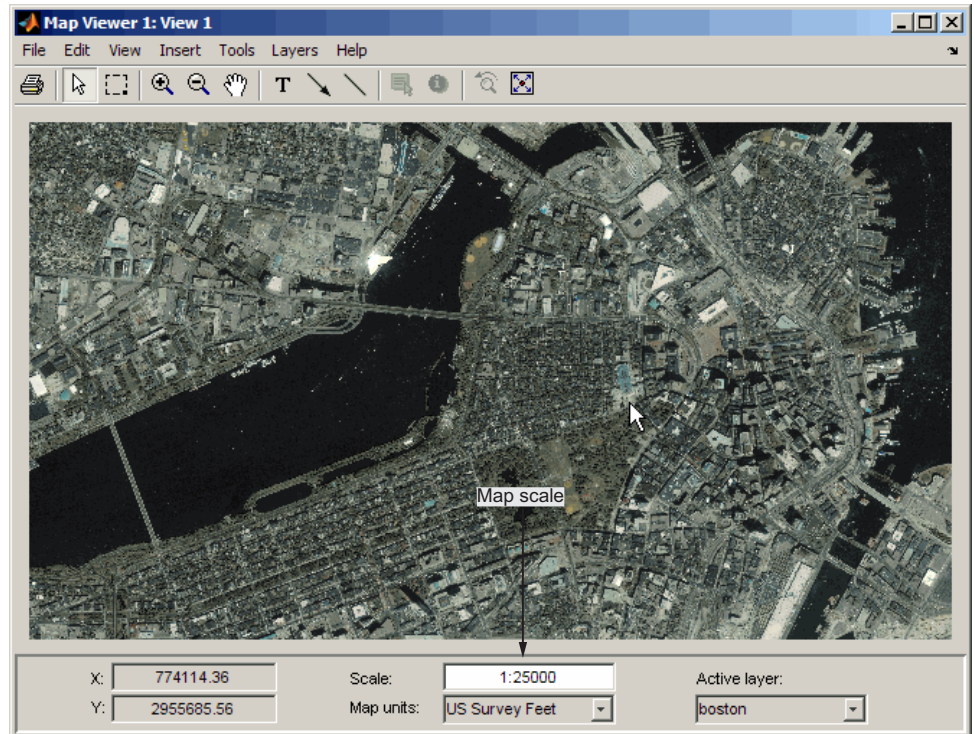
However, you can also navigate to this directory with the Map Viewer Import Data dialog if you prefer.

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



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

- 4 To see the map scale, set the map distance units. Use the drop-down **Map units** menu at the bottom center to select US Survey Feet.
- 5 Now set the scale to 1:25,000 by typing 1:25000 in the **Scale** box, which is above the **Map units** drop-down. The viewer now looks like this.



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

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

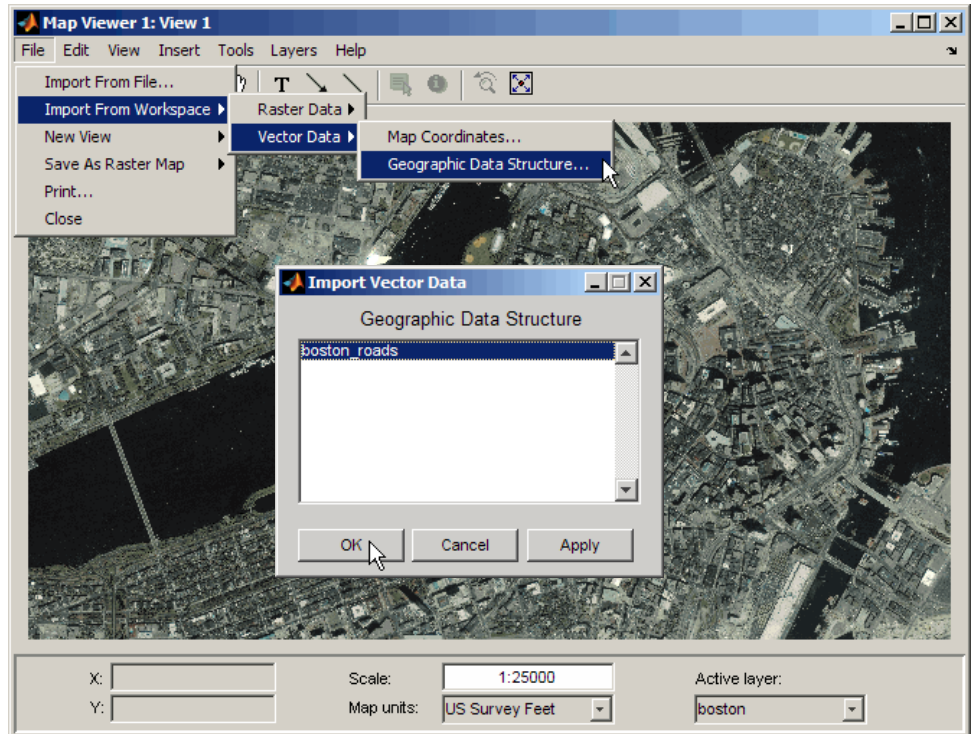
You can use **MATLAB** to convert the units of `boston_roads.shp` to meters. First, read the file into the workspace as a geospatial object using `shaperead`, and

then convert its X and Y coordinate fields from U.S. survey feet to meters using the following code:

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

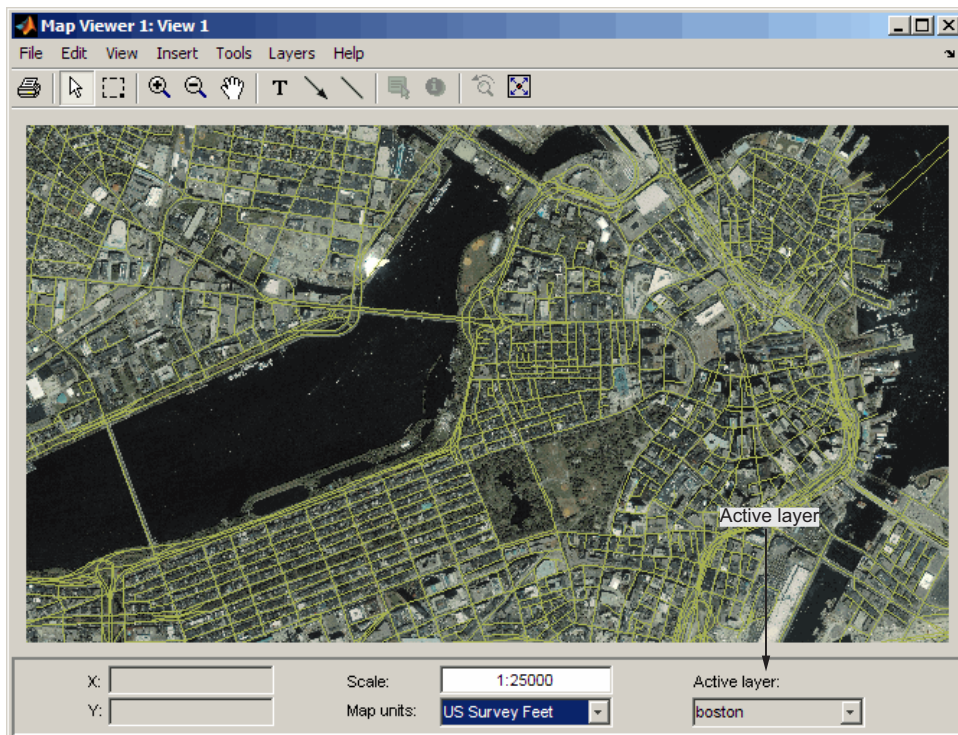
The `unitsratio` function computes conversion factors between a variety of units of length.

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



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

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

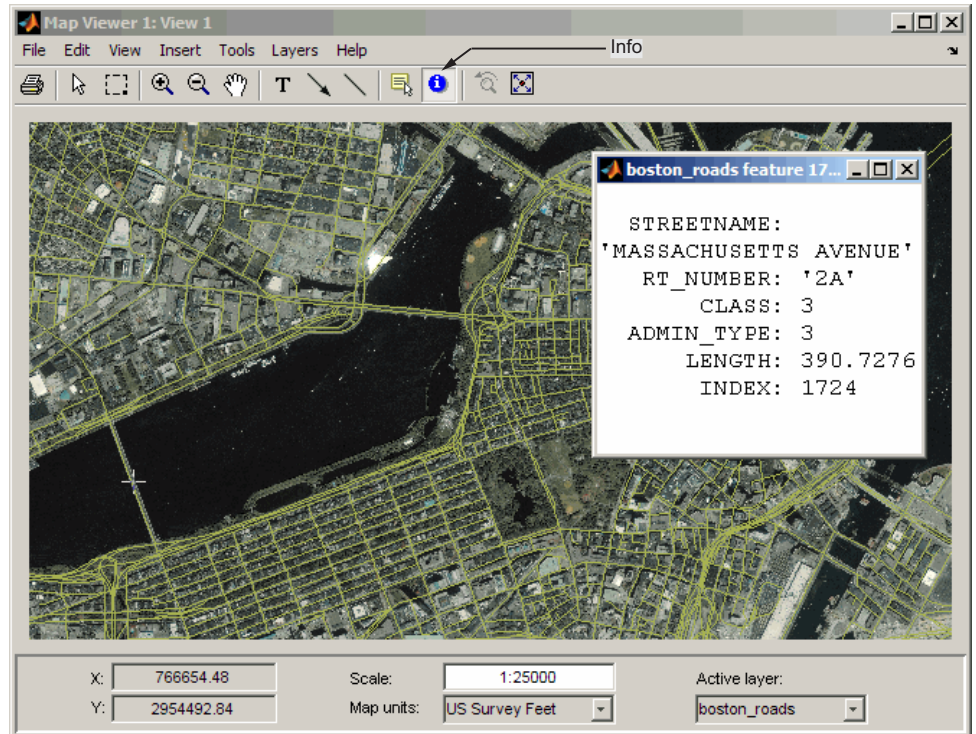


Being random, the color you see for the road layer can differ. How you can specify road colors is discussed below.

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

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

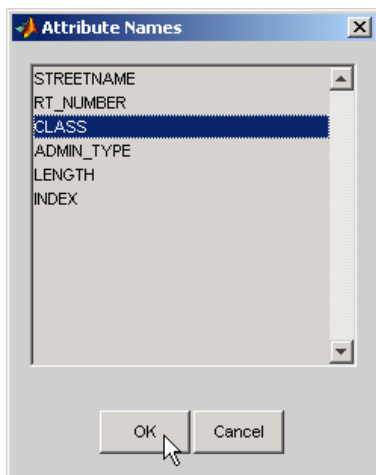
- 10 One way to see the attributes for a vector layer is to use the **Info** tool, a button near the right end of the toolbar. Select the **Info** tool and click somewhere along the bridge across the Charles River near the lower left of the map. This opens a text window displaying the attribute/values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



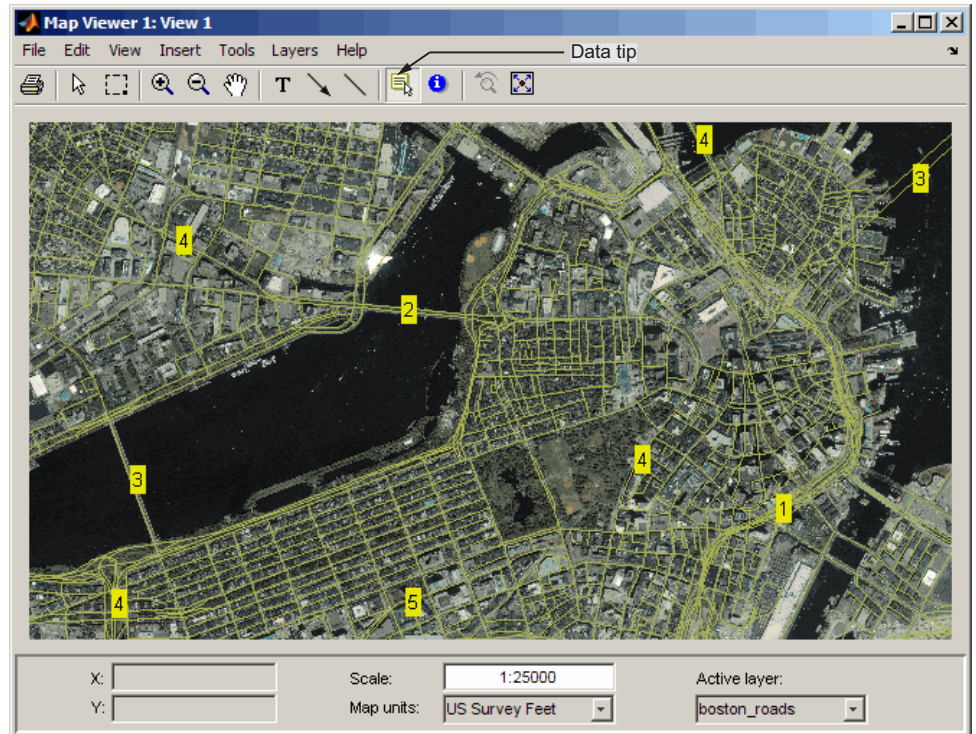
The selected road is Massachusetts Avenue (Route 2A). As the above figure shows, the `boston_roads` vectors have six attributes.

- 11 Get information about some other roads. Dismiss open Info windows by clicking their close boxes.
- 12 Choose an attribute for the **Datatip** tool to inspect. From the **Layers** menu, select **boston\_roads > Set Label Attribute**. From the list in the list box of the Attribute Names dialog, select **CLASS** and click **OK** to dismiss it. The dialog looks like this.





- 13** Select the **Datatip** tool. The cursor assumes a crosshairs (+) shape.
- 14** Use the **Datatip** tool to identify the administrative class of any road displayed. When you click on a road segment, a data tip is left in that place to indicate the CLASS attribute of the active layer, as illustrated below.

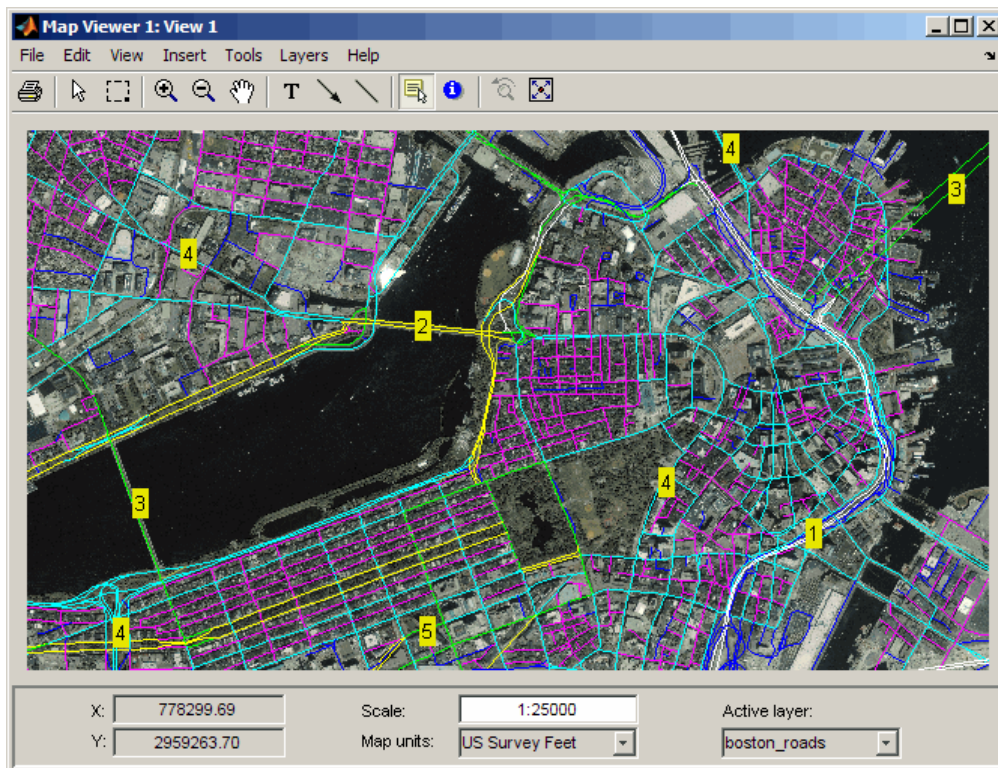


- 15** You can change how the roads are rendered by identifying an attribute to which to key line symbology. Color roads according to their CLASS attribute, which takes on the values 1:6. Do this by creating a *symbolspec* in the workspace. A *symbolspec* is a cell array that associates attribute names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

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

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

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



- 17** Remove the datatips before going on. To dismiss data tips, right-click each of them and select **Delete datatip** or **Delete all datatips** from the pop-up context menu that appears.
- 18** Add another layer, a set of points that identify 13 Boston landmarks. As you did with the **boston\_roads** layer, you import it from a shapefile. The locations for these landmarks are also given in meters, so you must convert their coordinates to units of survey feet before importing them into Map Viewer, as before.

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

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

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

The points of interest are symbolized as small x markers.

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

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

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

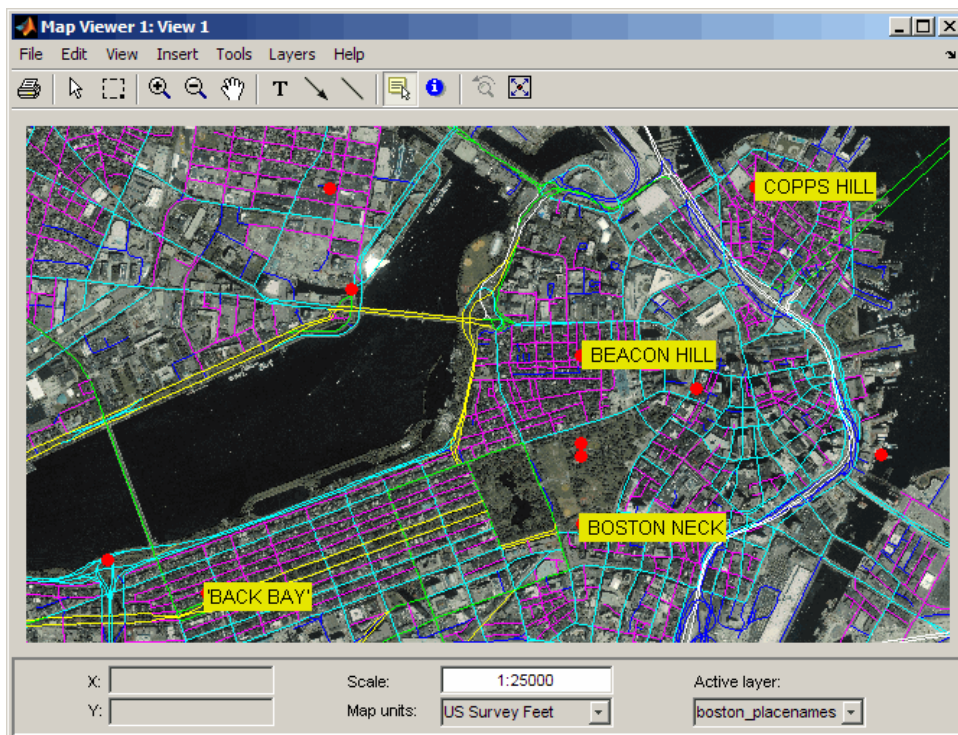
```
places = makesymbolspec('Point',{ 'Default','Marker','o', ...
    'MarkerEdgeColor','r','MarkerFaceColor','r'})
```

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

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

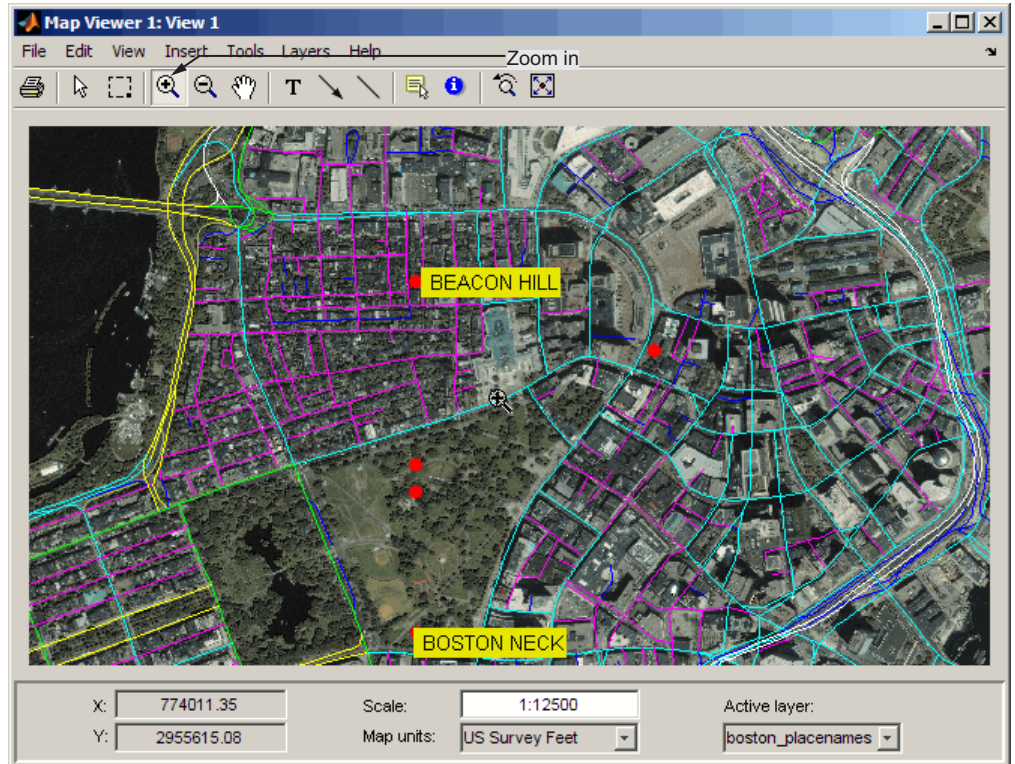
The Map Viewer reads the workspace variable `places`; the cross marks turn into red circles. Note that a layer need not be active in order for you to apply a `symbolspec` to it.

- 22 Now restore the other layers' visibility. In the **Layers** menu, select **boston\_roads**, and then slide right and select **Visible**. Do the same to show the **boston** image layer. The `boston_placenames` marker layer, because it was read in most recently, is on top.
- 23 Use the **Active layer** drop-down menu to make `boston_placenames` the currently active layer, and then select the **Datatip** tool. Click any red circle to see the name of the feature it marks. The map looks like this (depending on which data tips you show).



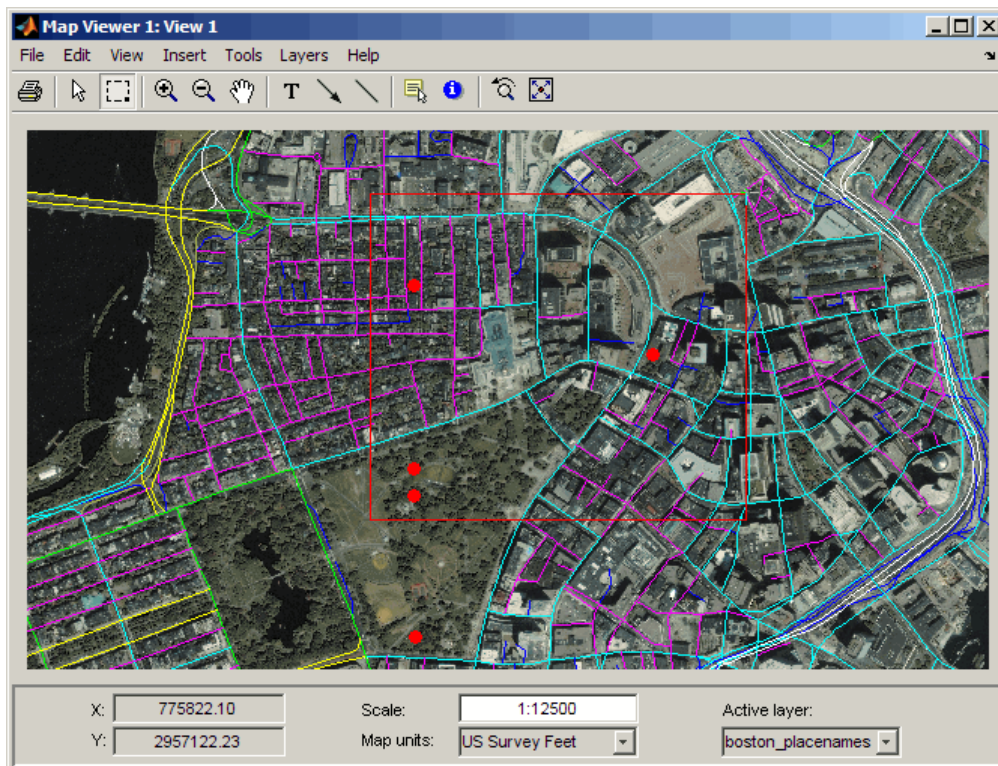
- 24 Zoom in on Beacon Hill for a closer view of the Massachusetts State House and Boston Common. Select the **Zoom in** tool, move the (magnifier) cursor

until the **X** readout is approximately 236,000 M and the **Y** readout is roughly 900,900 M, then click once to enlarge the view. The scale changes to about 1:10,000 and the map appears as below.



- 25 Right-click any of the data tips and select **Delete all datatips** from the pop-up context menu. This clears the place names you added to the maps.
- 26 Select an area of interest to save as an image file. Click the **Select area** tool, and then hold the mouse button down as you draw a selection rectangle. If you do not like the selection, repeat the operation until you are satisfied. If you know what ground coordinates you want, you can use the coordinate readouts to make a precise selection. The selected area appears as a red rectangle.

- 27 In order to be able to save a file in the next step, change your working directory to a writable directory, such as /work.
- 28 Save your selection as an image file. From the **File** menu, select **Save As Raster Map > Selected Area** to open a Save As dialog, as shown below.



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

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

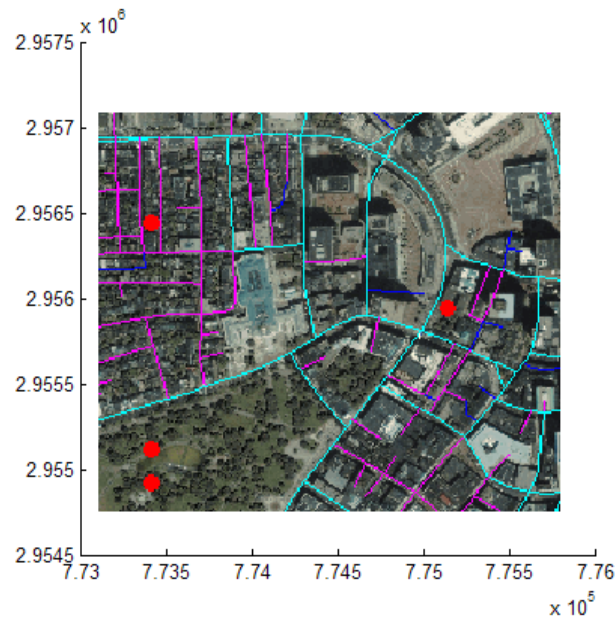
- An image file (`file.tif`, `file.png`, or `file.jpg`)

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

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

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

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



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

- 30** Experiment with other tools and menu items. For example, you can annotate the map with lines, arrows, and text, fit the map to the window,



draw a bounding box for any layer, and print the current view. You can also spawn a new Map Viewer using **New View** from the **File** menu. A new view can duplicate the current view, cover the active layer's extent, cover all layer extents, or include only the selected area, if any. When you are through with a viewing session, close the Map Viewer using the window's close box or select **Close** from the **File** menu.

## Getting More Help

In this section...
“Sources of Help for Mapping Toolbox” on page 1-26
“Consulting Release Notes” on page 1-27

### Sources of Help for Mapping Toolbox

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

You can find a classified list of functions in the “Geospatial Data Import and Access” on page 10-2 (online only). Help is available for individual commands and classes of Mapping Toolbox commands:

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

## **Consulting Release Notes**

To learn how one version of Mapping Toolbox differs from the next, read the Mapping Toolbox Release Notes, which include information on enhancements, bugs, known software and documentation problems, and upgrading issues.

## Mapping Toolbox Demos and Data

In this section...
“Available Demos” on page 1-28
“Locating Map Data” on page 1-29

### Available Demos

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

```
mapdemos
```

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

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

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

You can type

```
help mapdemos
```

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

## **Locating Map Data**

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

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



# Understanding Map Data

---

Maps and Map Data (p. 2-2)	What maps are and what makes digital map data special
Types of Map Data Handled by Mapping Toolbox (p. 2-4)	Representing maps with vector, raster, and mixed data models
Understanding Vector Data (p. 2-13)	Object-oriented data that “connects the dots”
Understanding Raster Data (p. 2-27)	Image- and surface-oriented gridded data
Reading and Writing Geospatial Data (p. 2-46)	Common data formats used for geospatial data that Mapping Toolbox can read or write

## Maps and Map Data

In this section...
“What Is a Map?” on page 2-2
“What Is Geospatial Data?” on page 2-2

### What Is a Map?

Mapping Toolbox manipulates electronic representations of geographic data. It lets you 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 Mapping Toolbox. The conclude by summarizing the functions available for importing and exporting geospatial data formats.

## Types of Map Data Handled by Mapping Toolbox

In this section...
“Vector Geodata” on page 2-4
“Raster Geodata” on page 2-7
“Combining Vector and Raster Geodata” on page 2-10

### 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 Mapping Toolbox, 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 (such as 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 to MATLAB:

```
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  $-80^\circ$  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 handles map projections, see Chapter 8, “Using Map Projections and Coordinate Systems”. For details on data structures that Mapping Toolbox uses to represent vector geodata, see “Vector Geodata” on page 2-4.

## 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. Mapping Toolbox uses the powerful matrix manipulation capabilities of MATLAB to fully exploit 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 2-38), 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, octrees, 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 MATLAB array is used rather than a 2-D (matrix) array. Another distinction is that while data grids are stored as class `double` in Mapping 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 Data” on page 2-27.

## A Look at Raster Data

**1** To view one possible display of the topo data grid, type the following:

```
clear all;
load topo
whos
```

Name	Size	Bytes	Class	Attributes
topo	180x360	518400	double	
topolegend	1x3	24	double	
topomap1	64x3	1536	double	
topomap2	128x3	3072	double	

- 2 The raster elevation data is in the variable `topo`. Inspect it with the MATLAB Array Editor by double-clicking `topo` in the Workspace pane or by typing in the Command Window

```
openvar topo
```

You will see that `topo` is a 2-D array, and that its values near its upper left corner range from 2,500 to 3,000 meters of elevation. The first row represents land elevations near the South Pole. When georeferenced with a three-element referencing vector (the variable `topolegend` in this case), Mapping Toolbox raster data is stored from the bottom up.

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

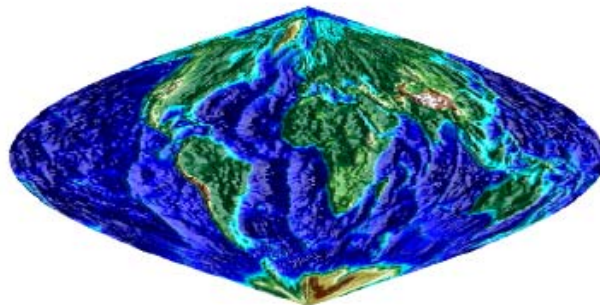
```
axesm sinusoid
```

A MATLAB 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,topolegend,'DisplayType','texturemap')  
demcmap(topo)
```

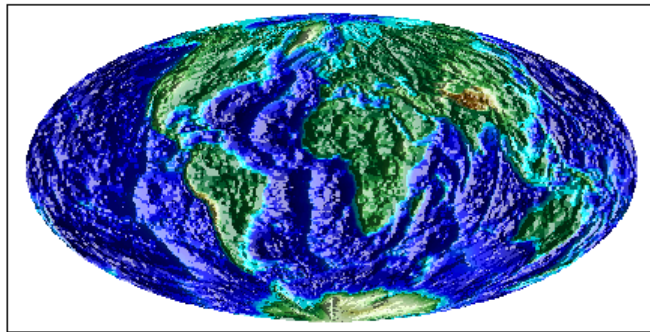
The `geoshow` function displays geodata in geographic (unprojected) coordinates. The `geoshow` output is shown below:



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

```
figure; axesm hammer  
meshlsrcm(topo, topolegend)
```

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



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

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

### Combining Vector and Raster Geodata

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



## Viewing Raster and Vector Data on the Same Map

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

- 1 Clear the current map:

```
clma
```

- 2 Reload the coastline data:

```
load coast
```

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

```
load topo
```

- 4 Set up a Robinson projection:

```
axesm robinson
```

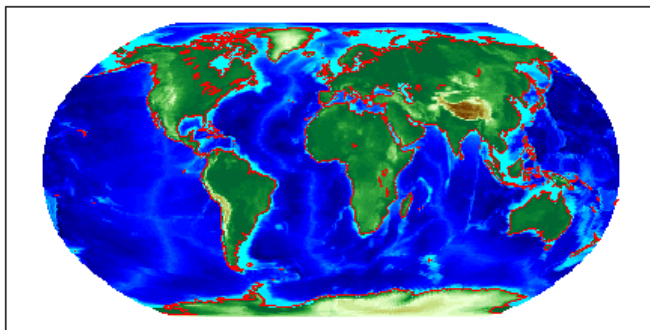
- 5 Plot the raster topographic data with an appropriate colormap:

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

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

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

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



For additional details on how Mapping Toolbox handles raster geodata, see “Understanding Raster Data” on page 2-27.

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 2-46 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 Data

### In this section...

“Points, Lines, Polygons” on page 2-13

“Segments Versus Polygons” on page 2-15

“Mapping Toolbox Geographic Data Structures” on page 2-16

“Selecting Data to Read with the `shaperead` Function” on page 2-21

### Points, Lines, Polygons

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

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

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

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

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

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

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

---

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

---

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

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

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

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

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

---

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

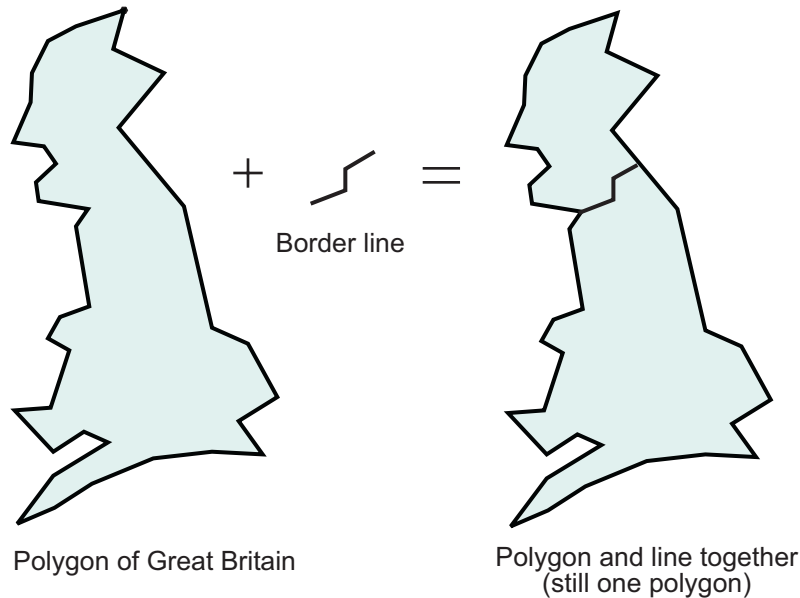
---

## Segments Versus Polygons

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

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

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



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

### Mapping Toolbox Geographic Data Structures

In examples provided in prior chapters, geodata was in the form of individual variables and had to be displayed using mapping functions specific to the type of available data (i.e., line, patch, matrix, text, etc.). Mapping Toolbox also provides an easy means of displaying, extracting, and manipulating collections of all types of map objects that have been organized in a family of specially defined and formatted *geographic data structures* (in general, referred to as a *geostruct*). Note that these structures are different from the *map projection structure* (also called an *mstruct*), which defines a map projection and related display properties. See the documentation for function `defaultm` for information about the contents of *mstructs*.

The following subsections describe two versions of Mapping Toolbox geographic data structures; the current version of the toolbox uses a form of geographic data structure that is more general than the type found in Version 1.x of the toolbox. You can use the older type as well, in appropriate circumstances, and convert it to the newer type when the latter is called for. You should be cognizant of the differences between the two types of structures, because some functions that originate in different versions of the toolbox (for example, `extractm` from Version 1 and `extractfield` from Version 2) can handle only the type of `geostruct` introduced in that version of the toolbox.

## Version 2 Geographic Data Structures

Certain functions introduced in Version 2 of Mapping Toolbox read, create, or manipulate vector geodata using a geographic data structure format that this document notates as *geostruct2*. This data structure has the flexibility to store any kind and number of attributes, and handles either geographic (latitude and longitude) or plane (*x* and *y*) coordinates. In contrast, the Version 1 geographic data structure is limited to a fixed set of fields and can contain geographic coordinates only.

The typical way to create a Version 2 geographic data structure is to input vector geodata to the workspace from a shapefile. The function `shaperead` returns a `geostruct2` that encapsulates some or all of the data stored in a group of shapefiles (which store attributes and coordinates in separate files). To determine what kinds of data a group of shapefiles contain, you can use the `shapeinfo` function to query them. `shapeinfo` returns a structure similar to the one that `shaperead` returns, but it cannot be used as a `geostruct`.

You can also transform a `geostruct1` into a `geostruct2`. Use the function `updategeostruct` for this purpose. See “Version 1 Geographic Data Structures” on page 2-19 for a description of that format.

The fields in a `geostruct2` depend on the type of geometry and the names and types of the attributes that have been read in. There will always be a text field called 'Geometry' that identifies the shape type. If the shape type is not 'Point' there will also be a field called 'BoundingBox' that contains [minX minY; maxX maxY].

Coordinate data is stored in fields called 'X' or 'Lon' and 'Y' or 'Lat', depending on what type of coordinates were read in. The names of these fields

are used by functions to determine if coordinates are projected or unprojected. However, the `geostruct` does not itself identify what map projection can be used or what its parameters are.

When a `geostruct2` contains 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 should polygons contain holes. The convention used encodes the coordinates of outer rings (e.g., continent outlines) in clockwise order, while counterclockwise ordering is used for inner rings (e.g., lakes and inland seas within a continent). Each ring is separated from the one preceding it in coordinate lists by a `NaN`.

When plotted as patches, clockwise rings are filled; counterclockwise rings are transparent, so that any underlying symbology shows through them. 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 the functions check or change the ordering of vertices that define a polygon, and the fourth one splits polygons with holes in a consistent fashion. They are also used in conjunction with the `polybool` function, which performs logical intersection of polygons.

The remainder of the `geostruct2` fields store attribute data. The fields are given appropriately mangled names by `shaperead` if the original attribute name could not be directly used as a field name. Unwanted attributes can be filtered out by `shaperead`.

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

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



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

This indicates that the shapefile contains 609 features. Each one can contain any number of shape points, but will possess the same attribute fields (any of which can be empty). For example, the tenth element has nine coordinates:

```
S(10)

ans =
  Geometry: 'Line'
  BoundingBox: [2x2 double]
             X: [1x9 double]
             Y: [1x9 double]
  STREETNAME: 'WRIGHT FARM'
  RT_NUMBER: ''
  CLASS: 5
  ADMIN_TYPE: 0
  LENGTH: 79.0347
```

For additional information about geographic data structures, see the reference page for `updategeestruct`.

## Version 1 Geographic Data Structures

Mapping Toolbox Version 1 geographic data structures, which are more fixed in their content, contain information required for the display of graphic objects within map axes. This document notates the older format as a *geostruct1*. The objects that a *geostruct1* describes are for the most part MATLAB figure graphic objects. Coordinate data is always given in latitude and longitude. The following table lists the six object types a *geostruct1* can contain, and indicates which fields of information are required for each:

Field	Light	Line	Patch	Regular	Surface	Text
type	•	•	•	•	•	•
tag	•	•	•	•	•	•
lat	•	•	•		•	•
long	•	•	•		•	•
map				•	•	
maplegend				•		
meshgrat				•		
string						•
altitude	•	•	•	•	•	•
otherproperty	•	•	•	•	•	•

Some fields can contain empty entries, but each indicated field must exist for the object to be displayed correctly. For instance, the altitude field can be an empty matrix and the otherproperty field can be an empty cell array.

The type field must be one of the specified map object types: 'line', 'patch', 'regular', 'surface', 'text', or 'light'.

The tag field must be a string different from the type field usually containing the name or kind of map object. Its contents must not be equal to the name of the object type (i.e., line, surface, text, etc.).

The lat, long, and altitude fields can be scalar values, vectors, or matrices, as appropriate for the map object type.

The map field is a data grid. If map is a regular data grid, refvec is its corresponding data grid legend, and meshgrat is a two-element vector specifying the graticule mesh size. If map is a geolocated data grid, lat and long are the matrices of latitude and longitude coordinates.

The otherproperty field is a cell array containing any additional display properties appropriate for the map object. Cell array entries can be a line specification string, such as 'r+', or property name/property value pairs, such as 'color', 'red'. If the otherproperty field is left as an empty cell

array, default colors are used in the display of lines and patches based on the tag field.

You can find additional details about Version 1 geographic data structures in the references pages for `displaym`, `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 features such as function handles, anonymous functions, and nested functions, you need not be familiar with these 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)
    minimumClass = 4;
    minimumLength = 200;
    result = (roadclass >= minimumClass) && ...
            (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 MATLAB 7, putting the selector in a file or subfunction of its own was the only way to work with a selector.

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

### Example 2: Predicate as Function Handle

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

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

### Example 3: Predicate as Anonymous Function

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

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

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

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

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

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

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

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

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

## LENGTH

**Example 5: Parameterizing the Selector; Predicate as Nested Function**

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

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

### In this section...

“Georeferencing Raster Data” on page 2-27

“Regular Data Grids” on page 2-29

“Geolocated Data Grids” on page 2-38

### Georeferencing Raster Data

Raster geodata consists of georeferenced data grids and images that MATLAB stores internally 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 Mapping 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):

$$\begin{aligned} [x \ y] &= [\text{row} \ \text{col} \ 1] * R, \text{ or} \\ [\text{long} \ \text{lat}] &= [\text{row} \ \text{col} \ 1] * R \end{aligned}$$

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 prior to 2.0 did not use referencing matrices, and called referencing vectors *map legend vectors* or sometimes just *map legends*. 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 90°N (the North Pole).
- The western edge of the map is at 0° (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 2-38. 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, not sparse, matrices that contain double values. MATLAB stores them in column order, with their southern edge as the first row and their northern edge as their last row.

## Constructing a Global Data Grid

Imagine an extremely coarse map of the world in which each cell represents 60°. Such a map matrix would be 3-by-6, and its referencing vector would be defined as

```
refvec = [1/60 90 -180] = [0.0167 90 -180]
```

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

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

- 2 Now make a referencing vector, as described above:

```
minivec= [1/60 90 -180]
```

```
minivec =
    0.0167    90.0000 -180.0000
```

As is often the case for global grids, the western edge is the international date line, at 180°W:

- 3 Set up an equidistant cylindrical map projection:

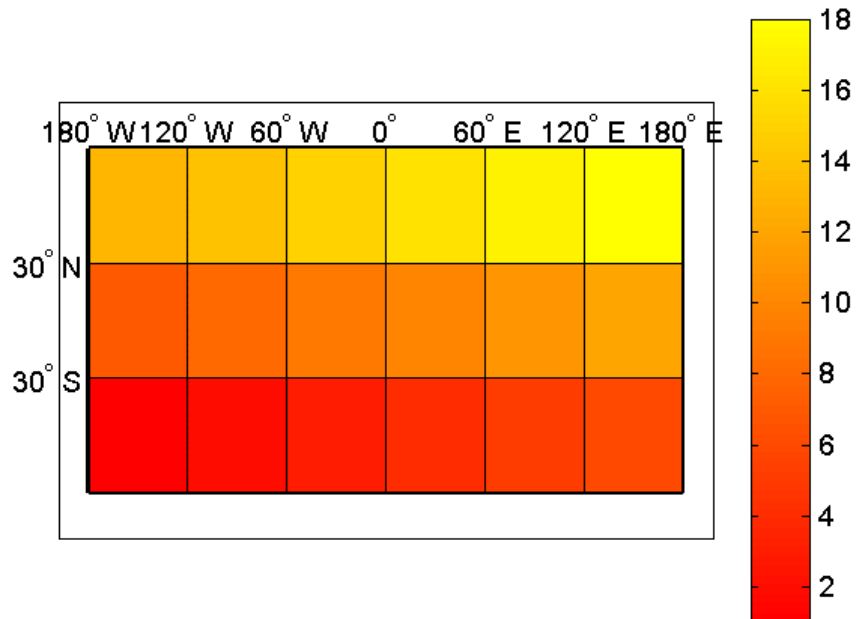
```
axesm('MapProjection', 'eqdcylin')
setm(gca, 'MapLatLimit', [-90 90], 'MapLonLimit', [-180 180], ...
'GLineStyle', '-', 'Grid', 'on', 'Frame', 'on')
```

- 4 Draw a graticule with parallel and meridian labels at 60° 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(minigrid, minivec); 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. All regular data grids in Mapping Toolbox, as well as regular surfaces in MATLAB, are displayed in this fashion.

### Computing Map Limits from Reference Vectors

Given a regular data grid and its reference 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 30°N to 45°N and from 115°W to 135°W:

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

latlimits =
    30    45
longlimits =
    115   135
```

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

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

rows =
    180
cols =
    240

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

southlat =
    30

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

eastlon =
```

135

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

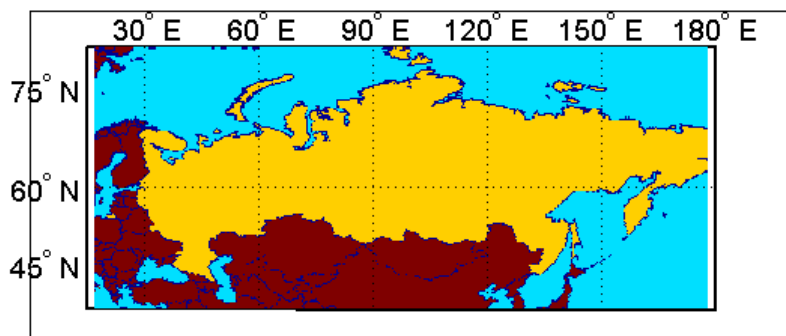
### Geographic Interpretation of Matrix Elements

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

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

latlim =
    35    80
longlim =
    15   190
```

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



The function `set1t1n` 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] = set1t1n(map,refvec,row,col)
```

```
lat =
 39.5
long =
 30.7
```

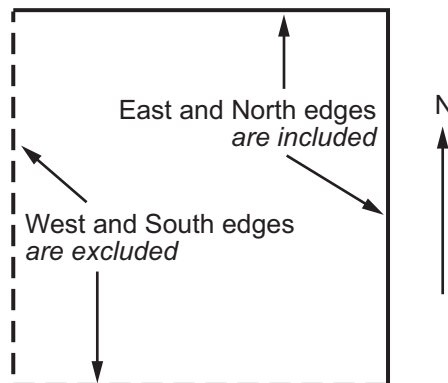
setpostn does the reverse of this, determining the row and column of the data grid element containing a given geographic point location:

```
[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 360° 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 `setlatln` 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] = setlatln(map,refvec,row,col)

lat =
    35.5
long =
    18.3
```

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

### Accessing Data Grid Elements

The actual values contained within the map matrix entries are important as well. Mapping Toolbox provides several functions for accessing and altering 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 MATLAB `find` function:

```
[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 MATLAB 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 25°N to 50°N and longitudes from 60°W to 130°W:

```

cellspdeg = 10;
[r,c,maplegend] = sizeM([25 50],[-130 -60],cellspdeg)

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:

```

cellspdeg2 = 5;
[r,c,maplegend] = sizeM([25 50],[-130 -60],cellspdeg2)

```

```
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, Mapping 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	

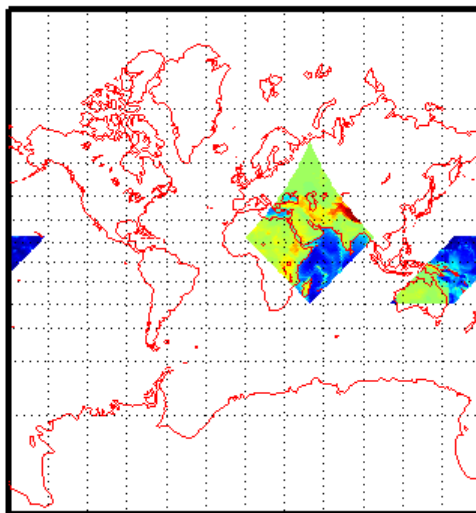
Two geolocated data grids are in this data set, each requiring three variables. The values contained in `map1` correspond to the latitude and longitude coordinates, respectively, in `lt1` and `lg1`. Notice that all three matrices are the same size. Similarly, `map2`, `lt2`, and `lg2` together form a 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:

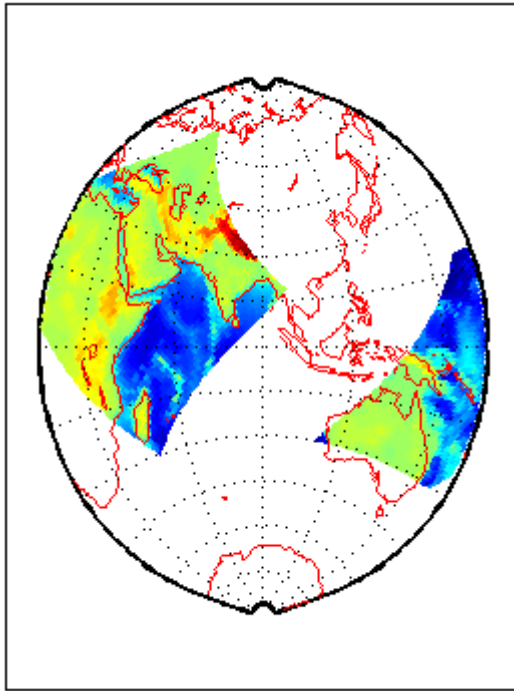
```
load coast
plotm(lat,long,'r')
```



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 0°N, 90°E:

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



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

### **Geographic Interpretations of Geolocated Grids**

Mapping Toolbox 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 .

**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:

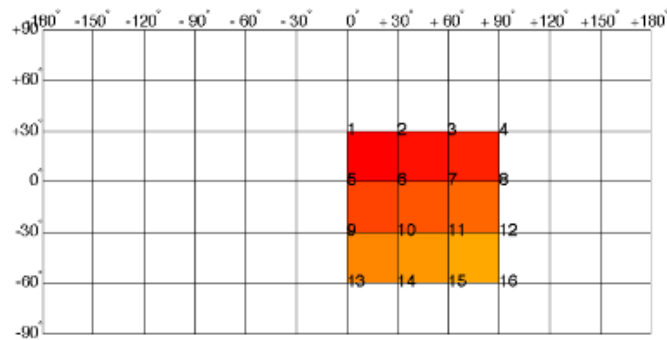
```
map = [ 1  2  3  4; ...
       5  6  7  8; ...
       9 10 11 12; ...
       3 14 15 16];

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.



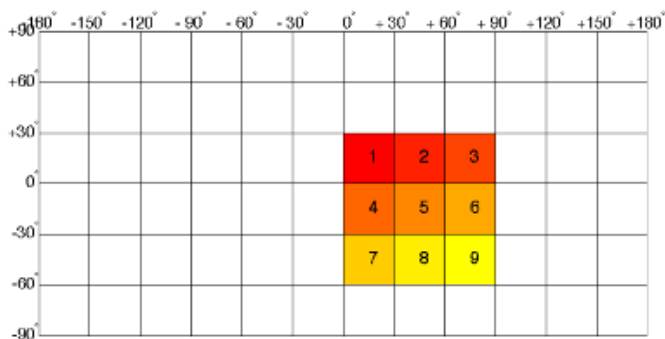


Notice that only 9 of the 16 total cells are displayed. The value displayed for each cell is the value at the upper left corner of that cell, whose coordinates are given by the corresponding `lat` and `long` elements. By MATLAB convention, the last row and column of the map matrix are not displayed, although they exist in the `CData` a 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” on page 2-46
“Exporting Vector Geodata” on page 2-51
“Functions That Read and Write Files in Compressed Formats” on page 2-61

### 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 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, Mapping 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 MathWorks Tech Note 2101, “Accessing Geospatial Data on the Internet for Mapping Toolbox,” 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 geostruct it returns should have X and Y or Lon and Lat fields).

Function	Description	Type of Data	Type of Coordinates
<code>arcgridread</code>	Read a gridded data set in Arc ASCII Grid Format	raster	map
<code>avhrrgoode</code>	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
<code>avhrrlambert</code>	Read AVHRR GLCC and NDVI data products stored in the Lambert Conformal Conic projection	raster	geolocated
<code>dcwdata</code>	Read selected data from the Digital Chart of the World (DCW)	vector	geo
<code>dcwgaz</code>	Search for entries in the DCW gazette	vector	geo
<code>dcwread</code>	Read a DCW file	vector	geo
<code>dcwrhead</code>	Read a DCW file header	properties	geo

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

<b>Function</b>	<b>Description</b>	<b>Type of Data</b>	<b>Type of Coordinates</b>
globedems	List GLOBE data filenames for a specified latitude-longitude quadrangle	filenames	geo
gshhs	Read Global Self-Consistent Hierarchical High-Resolution Shoreline (GSHHS) data	vector	geo
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
sdtsemread	Read U.S. Geological Survey (USGS) digital elevation model (DEM) stored in SDTS (Spatial Data Transfer Standard) format (Raster Profile)	raster	geo map
sdtinfo	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

Function	Description	Type of Data	Type of Coordinates
shaperead	Read geographic feature coordinates and associated attributes from a shapefile	vector	geo map
shapewrite	Write geospatial data and associated attributes in shapefile format	vector	geo map
tbase	Read data from the 5-minute TerrainBase global digital terrain model	raster	geo
tgrline	Read data from TIGER/Line files	vector	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 (VMAPO) CD-ROMs	vector	geo
vmap0read	Read a VMAPO file	vector	geo
vmap0rhead	Read VMAPO file headers	properties	geo
vmap0ui	Activate GUI for interactively selecting VMAPO 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

MATLAB 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. 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 and Using Referencing Matrices” 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 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. Mapping Toolbox provides the functions `shapewrite` and `kmlwrite` for exporting 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`. For information about creating KML files, see the following section.

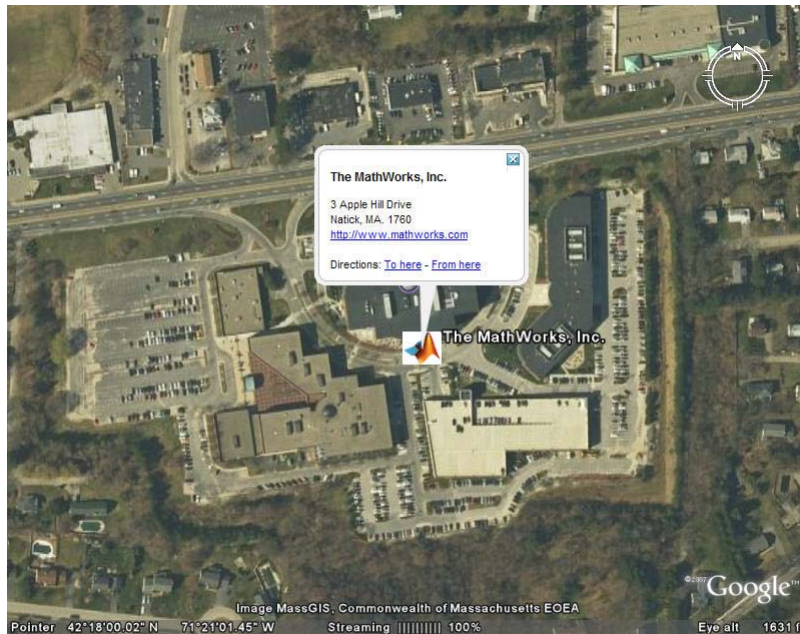
## Exporting KML Files for Viewing in Earth Browsers

Keyhole Markup Language (KML) is a XML dialect for formatting 2-D and 3-D geodata for display in “Earth browsers,” such as Google Earth™, Google Maps™, Google Mobile™, 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. Mapping Toolbox lets you 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 from Mapping Toolbox using `kmlwrite` that locates the headquarters of The 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 = 'The_MathWorks.kml';
kmlwrite(filename, lat, lon, ...
    'Description', description, 'Name', name, ...
    'Icon', iconFilename, 'IconScale', iconScale);

```

The file produced by `kmlwrite` looks like this:

```

<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://earth.google.com/kml/2.1">
  <Document>
    <name>The_MathWorks</name>
    <Placemark>
      <Snippet maxLine="0"> </Snippet>
      <description>3 Apple Hill Drive&lt;br&gt;Natick, MA. 01760&lt;br&gt;
        &lt;br&gt;http://www.mathworks.com&lt;br&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. Google Earth 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 Google Maps does not support address-based placemarks.) If the viewing application is capable of looking up addresses, such placemarks can be displayed in appropriate, although possibly imprecise, locations. When you use addresses, `kmlwrite` creates an `<address>` element for each placemark rather than `<point>` elements containing `<coordinates>` elements. For example, here is code for `kmlwrite` that generates address-based placemarks for three cities in Australia from a cell array:

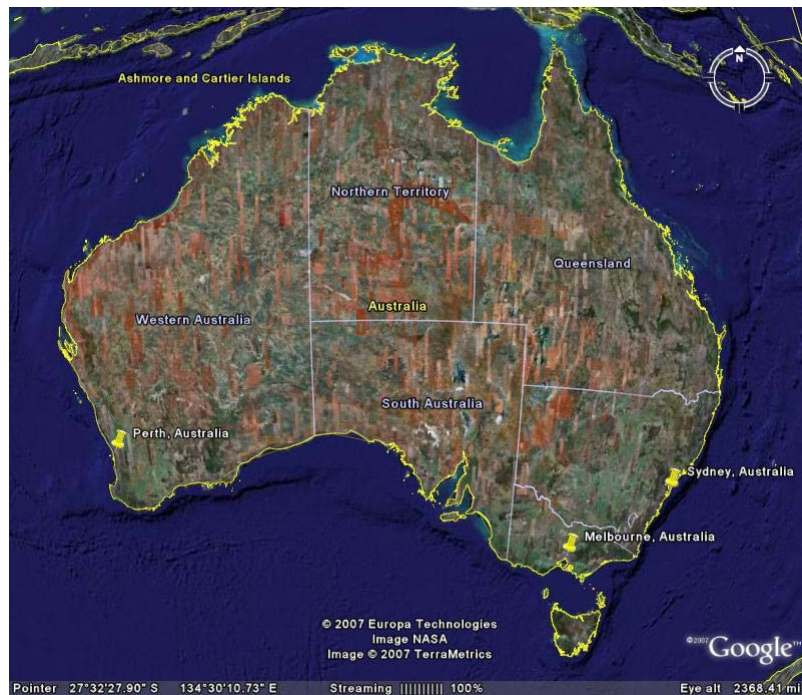
```
address = {'Perth, Australia', ...
           'Melbourne, Australia', ...
           'Sydney, Australia'};
filename = 'Australian_Cities.kml';
kmlwrite(filename, address, 'Name', address);
```

The generated KML file has the following structure and content:

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

```
<Placemark>
  <Snippet maxLine="0"> </Snippet>
  <description> </description>
  <name>Sydney, Australia</name>
  <address>Sydney, Australia</address>
</Placemark>
</Document>
</kml>
```

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



**Exporting Road Names to Placemarks.** The `concord_roads.shp` sample data contains line geometry and attributes for the road network of Concord, Massachusetts. The road geometry and attribute data is read in from a shapefile using `shaperead`. This example shows how to unproject the coordinates (which are stored in meters) to latitude and longitude, extract just one instance of each road, changing the geometry from line to point, and export these locations as KML placemarks:

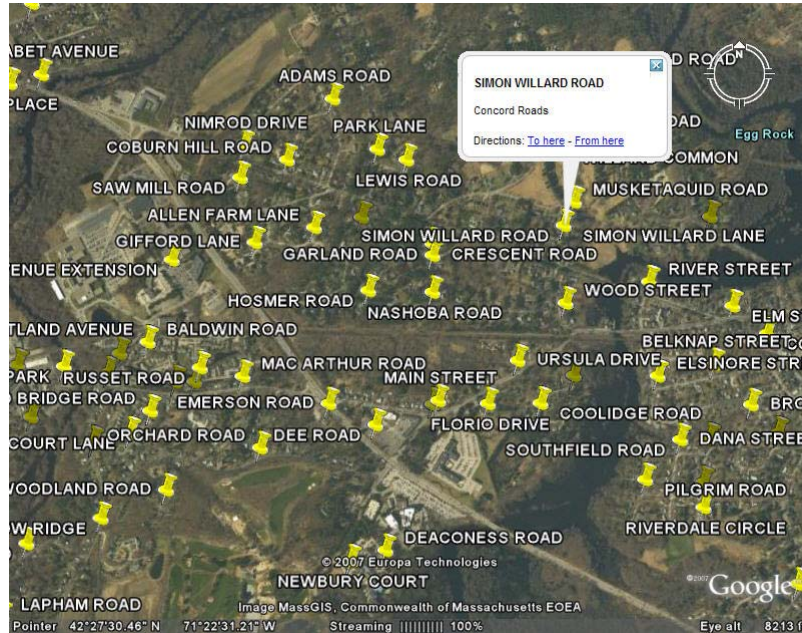
```
% Import the shapefile and change its Geometry type to 'Point'
roads = shaperead('concord_roads');
[roads.Geometry] = deal('Point');
%
% Eliminate all but one mention of each street in the geostruct
[U, i] = unique({roads.STREETNAME});
roads = roads(i);
%
% The coordinates of the roads are in projected map coordinates,
% and must be unprojected from meters to latitude-longitude.
% Obtain projection parameters from a related GeoTIFF file,
% which enables a mapping between latitude-longitude and
% map coordinates in U.S. survey feet.
%
proj = geotiffinfo('boston.tif');

% Loop through roads, assigning Lat and Lon fields.
feetPerMeter = unitsratio('survey foot','meter');
for k=1:numel(roads)
    % Take the first point on each road and convert its coordinates
    % from meters to survey feet, to make them compatible with proj.
    x = feetPerMeter * roads(k).X(1);
    y = feetPerMeter * roads(k).Y(1);
    [roads(k).Lat, roads(k).Lon] = projinv(proj, x, y);
end
%
% Having unprojected, the X and Y fields are no longer useful
roads = rmfield(roads,{'X','Y'});

% Write contents of roads to a KML file. Give all the roads the
% same KML description tag, 'Concord Roads', and derive the KML
% name tag for each road from the value of its STREETNAME field.
```

```
filename = 'concord_roads.kml';
kmlwrite(filename, roads, 'Description', 'Concord Roads', ...
         'Name', {roads.STREETNAME})
```

In Google Earth, a portion of the placemarks appears like this.



**Setting Up Tables for the Road Placemarks.** While placemarks in Google Earth can have elaborate content and formatting, `kmlwrite` generates simple two-column tables only, having the form

Attribute1 Name	Attribute1 Value
Attribute2 Name	Attribute2 Value
Attribute3 Name	Attribute3 Value
...	...

Placemark tables can have many rows; if they exceed a certain length, the Earth browser generally formats them with scrollbars. Typically, you create

tables from attribute data contained in the geostruct from which you are extracting coordinates for placemarks. Each table is encoded inside a KML `<description>` tag. Continuing with the preceding example, the attributes of road segments for a portion of Concord, Massachusetts are formatted as a table using a struct output from function `makeattribspec`:

```
% Create an attribute spec that describes all fields in the geostruct
attribspec = makeattribspec(roads)

attribspec =
    STREETNAME: [1x1 struct]
    RT_NUMBER: [1x1 struct]
    CLASS: [1x1 struct]
    ADMIN_TYPE: [1x1 struct]
    LENGTH: [1x1 struct]
```

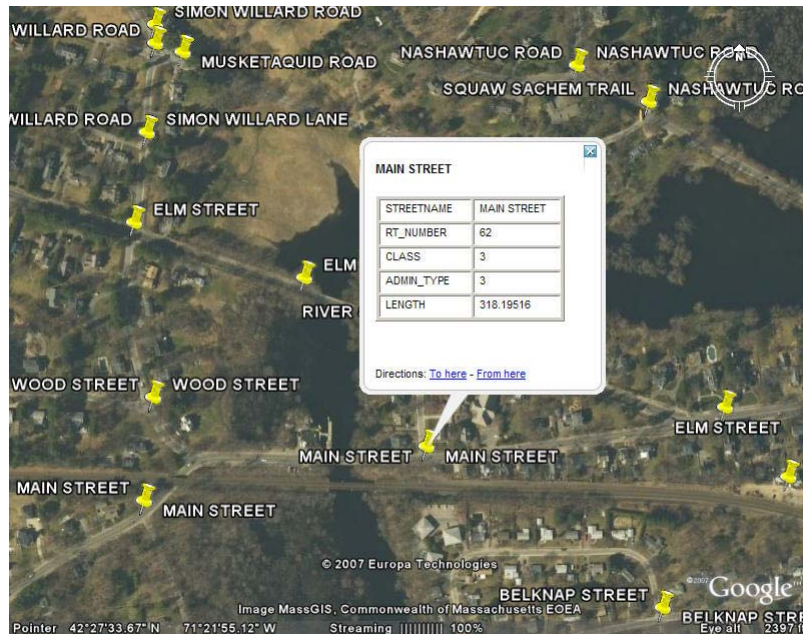
Each of these fields has two subfields that control table entry appearance:

- `AttributeLabel` — The label to be printed for the attribute in placemark tables
- `Format` — a `sprintf`-type format string to write numeric and string values

Because attribute labels are copied from geostruct field names, if you use the attribute spec in its initial form to format the data, all fields (except coordinates) are included in the KML file and they are presented in upper case with underscores instead of spaces, as the following shows in Google Earth:

```
filename = 'concord_roads0.kml';
kmlwrite(filename, roads, 'Description', ...
        attribspec, 'Name', {roads.STREETNAME})
```





You can customize the display and make it more readable, by modifying the attribute spec. For example, you can

- Eliminate the ADMIN\_TYPE attribute
- Rename the STREETNAME field to 'Street Name'
- Rename the RT\_NUMBER field to 'Route Number'
- Rename all other fields with first letter upper case only
- Highlight each attribute label with a bold font
- Reduce the number of decimal places used to display road lengths

As the proj struct indicates that length units are meters, add that information to the format specifier for LENGTH.

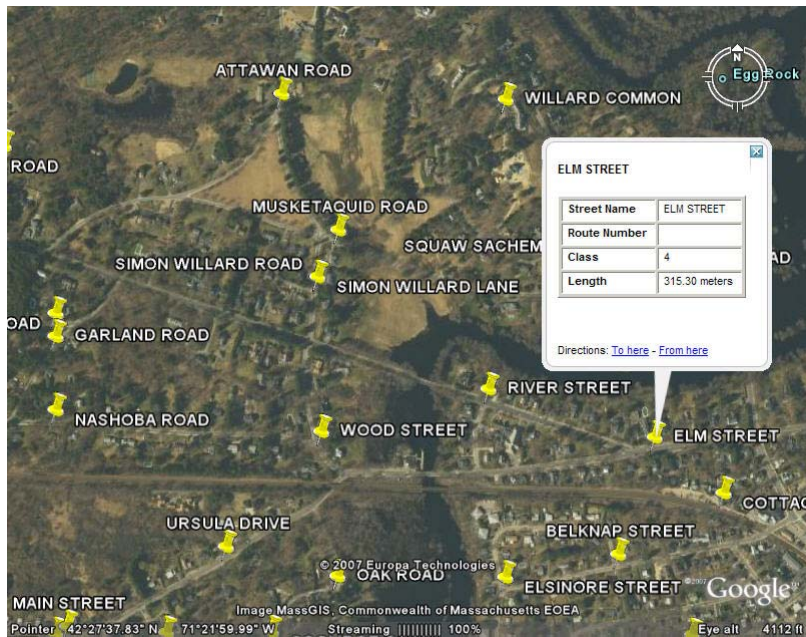
```
attribspec = rmfield(attribspec, 'ADMIN_TYPE');
attribspec.STREETNAME.AttributeLabel = '<b>Street Name</b>';
attribspec.RT_NUMBER.AttributeLabel = '<b>Route Number</b>';
attribspec.CLASS.AttributeLabel = '<b>Class</b>';
```

```
attribspec.LENGTH.AttributeLabel = '<b>Length</b>';  
attribspec.LENGTH.Format = '%.2f meters';
```

Now again pass the attribute spec as a Description to kmlwrite along with the geostruct, also identifying the geostruct element to use to name each feature:

```
filename = 'concord_roads.kml';  
kmlwrite(filename, roads, 'Description', ...  
         attribspec, 'Name', {roads.STREETNAME})
```

A Google Earth view of part of this cluster of street features is shown below.



---

**Note** You can only use attribute specs when you pass a geostruct to `kmlwrite`. When specifying placemarks via latitude-longitude points or vectors or via cell arrays of addresses, you need to construct a `Description` manually if you want to display one. Such descriptions can include attribute values and HTML tables, but you must format values and insert HTML markups yourself using `sprintf` or by other means.

---

## Functions That Read and Write Files in Compressed Formats

Geospatial data, like other files, are frequently stored and transmitted in compressed archive formats, such as zip or tar, or compressed formats such as GNU zip. MATLAB can read and write such files, and can uncompress the archives it reads, to create files in a directory for which you 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 describes MATLAB functions that you can use to read, uncompress, compress, and write archived data files, geospatial or otherwise:

Function	Purpose
<code>gunzip</code>	Uncompress files in the GNU zip format
<code>untar</code>	Extract the contents of a tar file
<code>unzip</code>	Extract the contents of a zip file
<code>gzip</code>	Compress files into the GNU zip format
<code>tar</code>	Compress files into a tar file
<code>zip</code>	Compress files into a zip file

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



# Understanding Geospatial Geometry

---

Understanding Spherical Coordinates (p. 3-2)

Geodetic approaches to modeling the shapes of planets

Understanding Latitude and Longitude (p. 3-11)

Locating positions on spheres and spheroids

Understanding Angles, Directions, and Distances (p. 3-14)

How length, angles, and their representations relate on the sphere

Understanding Map Projections (p. 3-29)

Flattening the Earth for display and analysis

Great Circles, Rhumb Lines, and Small Circles (p. 3-32)

Three important types of curves on the surface of the sphere or spheroid

Directions and Areas on the Sphere and Spheroid (p. 3-38)

What an azimuth is, and how its meaning can vary

Planetary Almanac Data (p. 3-46)

Using the almanac function to set up spherical parameters for mapping calculations

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” on page 3-2
“Geoid and Ellipsoid” on page 3-2
“The Ellipsoid Vector” on page 3-4

### 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 accurately models the shape, or figure, of the Earth and other planets.

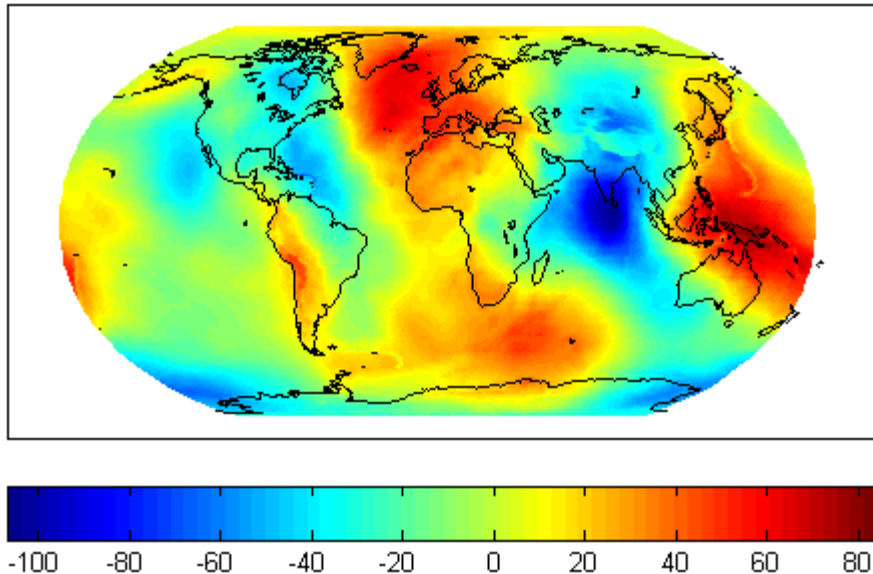
### Geoid and Ellipsoid

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

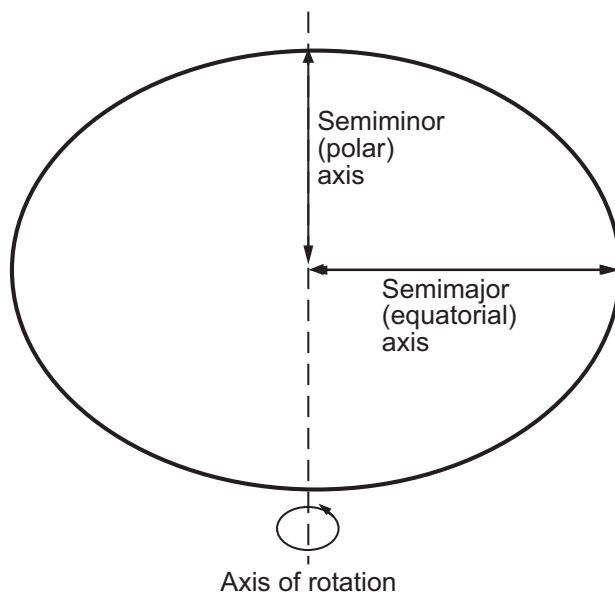
The following figure, made using the geoid data set, maps the figure of the Earth. To execute these commands, select them all by dragging over the list in the Help browser, then click the right mouse button and choose Evaluate Selection:

```
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.



Mapping 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

- “How Mapping Toolbox Manages Ellipsoids” on page 3-7
- “Functions that Define Ellipsoid Vectors” on page 3-9
- “What Is the “Correct” Ellipsoid Vector?” on page 3-9



Ellipsoids in Mapping Toolbox are most often represented as two-element vectors, called *ellipsoid vectors* in this guide. 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 `almanac` function in Mapping Toolbox (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 MATLAB to list 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 functions in Mapping Toolbox 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.

## How Mapping Toolbox Manages Ellipsoids

Most maps you make with the toolbox are displayed in a map axes, which is a MATLAB axes that contains a key data structure called a “map projection structure,” or *mstruct*. A reference ellipsoid is fundamental to defining a map axes, and is stored in the `geoid` field of the *mstruct*. (The geographic term “geoid” actually refers to a model of the shape of the earth that is much more detailed. See “Geoid and Ellipsoid” on page 3-2 for more information.) Other *mstruct* fields specify parameters that define the map axes’ current projection and for controlling the appearance of the map frame, grid, and grid labels. You define an *mstruct* with the `axesm` or `defaultm` functions. See “Map Axes Object Properties” on page 11-32 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] = mwdtran(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] = mwdtran(0,90)

x =
  1.5708
y =
     0
```

This value for `x` turns out to equal  $\pi/2$ , 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
```

Mapping 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 Mapping Toolbox. If you cannot find the ellipsoid vector you need, you can create it in the following form:

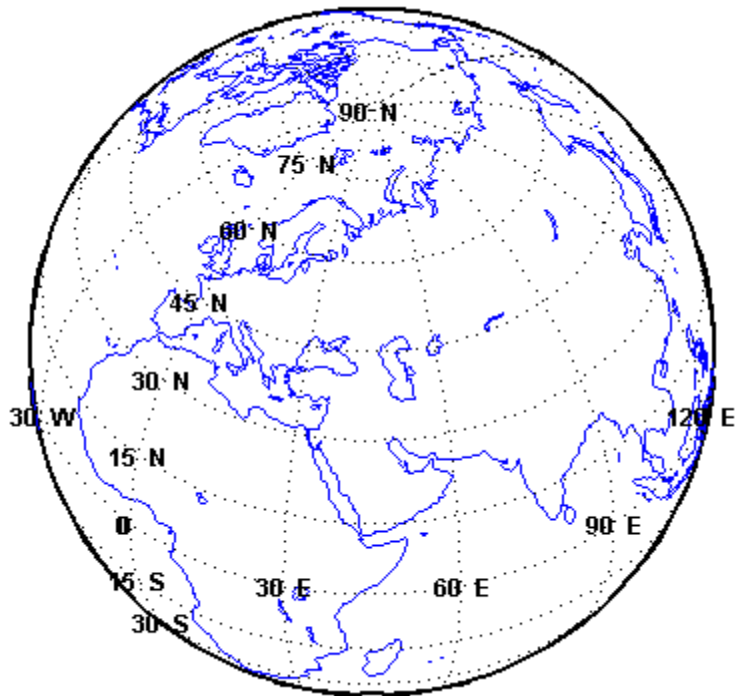
```
ellipsoidvec = [semimajor_axis eccentricity]
```

## 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  $+90^\circ$  at the north pole, and negative in the southern hemisphere, reaching a limit of  $-90^\circ$  at the south pole. Lines of constant latitude are called *parallels*. This system is depicted in the following figure, commands for which are

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



*Longitude* is the angle at the center of the planet between two planes that align with and intersect along the axis of rotation, perpendicular to the plane of the equator. One plane passes through the surface point in question, and the other plane is the *prime meridian* ( $0^\circ$  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 ( $90^\circ\text{N}$  and  $-90^\circ\text{S}$ ), and consequently longitude is under-specified in those two places.

Longitudes typically range from  $-180^\circ$  to  $+180^\circ$ , but other ranges can be used, such as  $0^\circ$  to  $+360^\circ$ . Longitudes can also be specified as east of Greenwich (positive) and west of Greenwich (negative). Adding or subtracting  $360^\circ$  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 artificial



discontinuities that result from forcing all values to lie within some 360°-wide interval.

## Understanding Angles, Directions, and Distances

### In this section...

“Positions, Azimuths, Headings, Distances, Length, and Ranges” on page 3-14

“Working with Length and Distance Units” on page 3-15

“Working with Angles: Units and Representations” on page 3-18

“Working with Distances on the Sphere” on page 3-23

“Angles as Binary and Formatted Numbers” on page 3-27

### 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 contains functions 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 9-11 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

- “Choosing Units of Length” on page 3-16
- “Converting Units of Length” on page 3-16
- “Computing Conversion Factors” on page 3-17

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 Mapping 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

Mapping Toolbox provides the following functions for converting between different units of length:

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

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

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

### Computing Conversion Factors

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

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

The syntax for `unitsratio` is

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

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

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

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

```
cmPerInch =  
    2.5400
```

```
cm =  
    10.16
```

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

```
inch = unitsratio('in','centimeter') * cmPerInch  
  
inch =  
    1
```

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

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

## Working with Angles: Units and Representations

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

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

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

Absolute positions are expressed in *geodetic coordinates*, which are actually angles between lines or planes on a reference sphere or ellipsoid. Relative positions use units of angle to express the direction between one place on the reference body from another one. Spherical distances quantify how far two places are from one another in terms of the angle subtended along a

great-circle arc. On nonspherical reference bodies, distances are usually given in linear units such as kilometers (because on them, arc lengths are no longer proportional to subtended angle).

## Radians and Degrees

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

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

As shown above, you can scale degrees to radians by multiplying by `pi/180`. However, you should consider using the Mapping Toolbox function `deg2rad` for this purpose:

```
thetaInRadians = rad2deg(thetaInDegrees)
```

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

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

or by using `rad2deg`,

```
thetaInRadians = rad2deg(thetaInDegrees)
```

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 `rad2deg` and `deg2rad` are very simple and efficient, and operate on vector and higher-dimensioned input as well as scalars.

#### **Default and Variable Angle Units**

Unlike the trigonometric functions in MATLAB, 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.,  $35^{\circ} 15'$ , equal to  $35.25^{\circ}$
- Degrees-minutes-seconds notation (DMS), e.g.,  $35^{\circ} 15' 45''$ , equal to  $35.2625^{\circ}$

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.)

Mapping 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(rad2deg(-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(rad2deg(-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.

Mapping 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 `deg2rad` and `rad2deg` 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? Mapping 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 M-function in which the units of both input angles and output angles remain unknown until run time, you can still accomplish the conversion by using the `unitsratio` function. For example,

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

piInDegrees =
    180
```

## Working with Distances on the Sphere

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

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

could also call the same number a “distance in earth radii.” When you work with transformations of geodata, keep this in mind.

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

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

distInMeters =
    1.0571e+007
```

Then convert the result to an angle in radians:

```
distInRadians = distInMeters / earthRadiusInMeters

distInRadians =
    1.6593
```

Finally, convert to an angle in degrees:

```
distInDegrees = rad2deg(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 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 statute 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 unit-to-unit distance and arc conversion functions provided in Mapping Toolbox. 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
<b>Degrees</b>	toDegrees fromDegrees	deg2rad toRadians fromDegrees	deg2km	deg2nm	deg2sm
<b>Radians</b>	rad2deg toDegrees fromRadians	toRadians fromRadians	rad2km	rad2nm	rad2sm
<b>Kilometers</b>	km2deg	km2rad		km2nm	km2sm
<b>Nautical Miles</b>	nm2deg	nm2rad	nm2km		nm2sm
<b>Statute Miles</b>	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(rad2deg(1))
```

```
ans =  
57.2958
```

gives a value in decimal degrees. In casual communication it is common to refer to a quantity such as rad2deg(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(rad2deg(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?” on page 3-29

“Forward and Inverse Projection” on page 3-30

“Projection Distortions” on page 3-30

## 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 8-6 for discussions and illustrations of how these transformations work.

Mapping Toolbox includes 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. Mapping 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 12, “Map Projections — By Category”. “Summary and Guide to

Projections” on page 8-64 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. Mapping 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 provides tools to help quantify and visualize projection distortions.

## Great Circles, Rhumb Lines, and Small Circles

In this section...
“Great Circles” on page 3-32
“Rhumb Lines” on page 3-32
“Small Circles” on page 3-33

### 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.

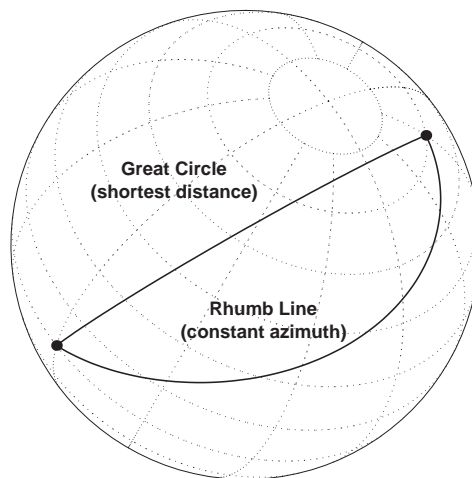
### Rhumb Lines

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  $90^\circ$ . 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 and Mapping Toolbox: 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. In Mapping Toolbox, this definition

includes 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 (45°N,60°E)* 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 90° 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 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.

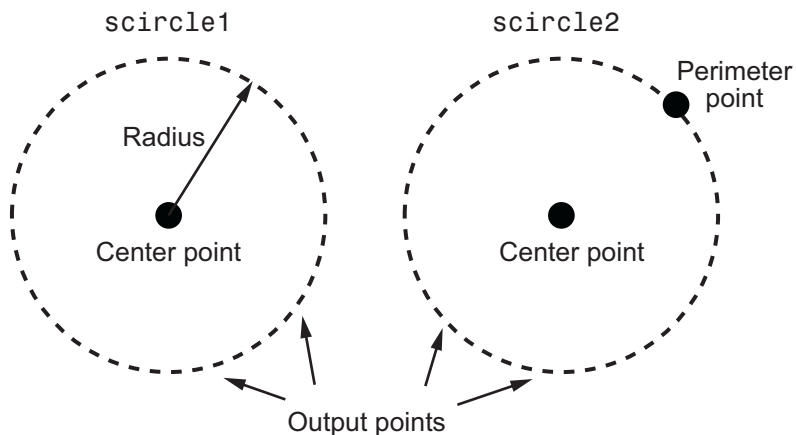
### Computing Small Circles

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 10° distance from (67°N, 135°W), use the following:

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

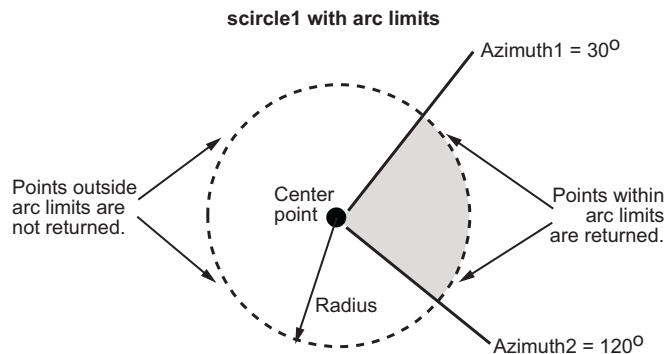
To get the small circle centered at the same point that passes through the point (55°N,135°W), use `scircle2`:

```
[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  $10^\circ$  in distance and between  $30^\circ$  and  $120^\circ$  in azimuth from  $(67^\circ\text{N}, 135^\circ\text{W})$ , simply provide arc limits:

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



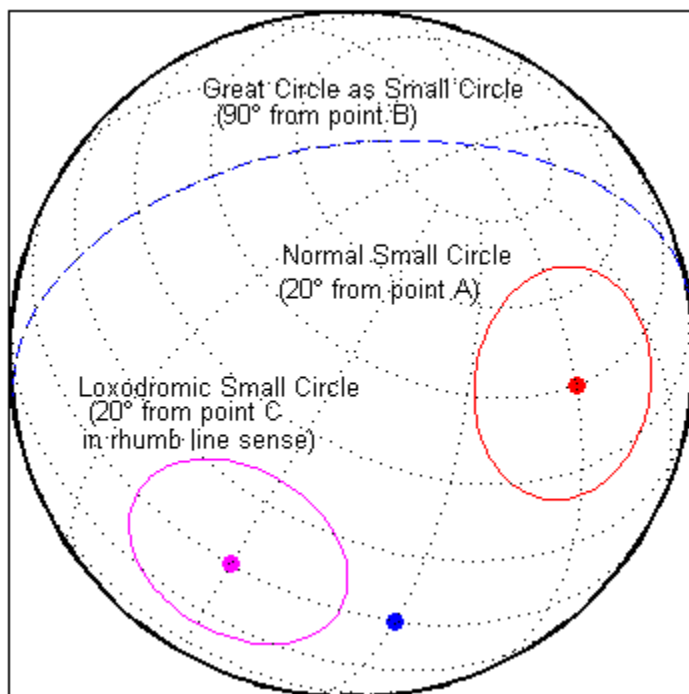
When an entire small circle is calculated, the data is in polygon format. For all calculated small circles, 100 points are returned unless otherwise specified. You can calculate several small circles at once by providing vector inputs. For more information, see the `scircle1` and `scircle2` function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





## Directions and Areas on the Sphere and Spheroid

### In this section...

“About Azimuths” on page 3-38

“Reckoning — The Forward Problem” on page 3-38

“Distance, Azimuth, and Back-Azimuth (the Inverse Problem)” on page 3-41

“Measuring Area of Spherical Quadrangles” on page 3-44

### About Azimuths

*Azimuth* is the angle a line makes with a meridian, measured clockwise from north. Thus the azimuth of due north is  $0^\circ$ , due east is  $90^\circ$ , due south is  $180^\circ$ , and due west is  $270^\circ$ . You can instruct Mapping Toolbox 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  $0^\circ$ ,  $90^\circ$ ,  $180^\circ$ , and  $270^\circ$  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  $((Az + 180^\circ) \bmod 360^\circ)$ . 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 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 Mapping 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 (40.75°N, 73.9°W) 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.

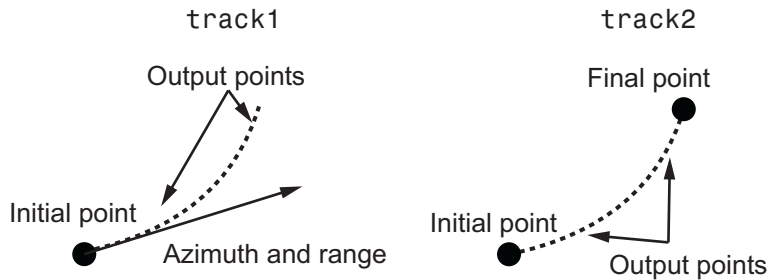
### Calculating Tracks – Great Circles and Rhumb Lines

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 (31°S, 90°E) with an azimuth of 45° with a length of 12°, use `track1`:

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

For the great circle from (31°S, 90°E) to (23°S, 110°E), use `track2`:

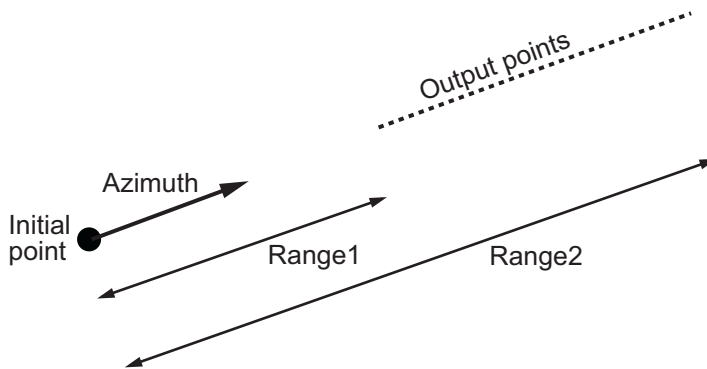
```
[latgc,longc] = track2('gc',-31,90,-23,110);
```



The track1 function also allows you to specify range endpoints. For example, if you want points along a rhumb line starting 5° away from the initial point and ending 13° away, at an azimuth of 55°, 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 360°, encircling the planet and returning to the initial point. For rhumb lines, the complete track terminates at the poles, unless the azimuth is 90° or 270°, 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 9-11.

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

When you calculate the distance between two points with Mapping Toolbox, the result depends upon whether you want a great circle or a 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 (15°S, 0°) and (60°N, 150°E). 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 Mapping 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° 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	0° or 360°
Northeast	45°
East	90°
Southeast	135°
South	180°
Southwest	225°
West	270°
Northwest	315°

*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?

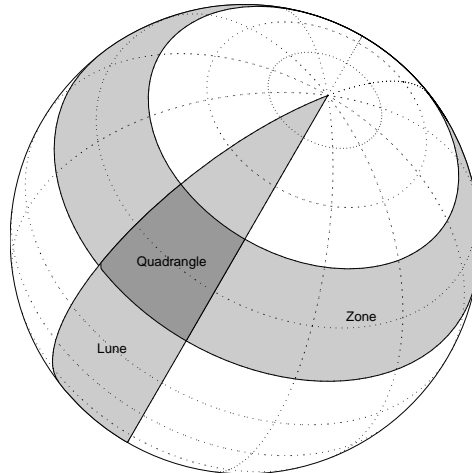
```
[elevang,slanrange,azim] = elevation(0,0,0,0,km2deg(10),10000)
```

```
elevang =
    44.901
slanrange =
    14156
azim =
```

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 15°N and 45°N latitudes, and a lune, which is the region bounded by 0° and 30°E longitude. Under the spherical planet assumption, the fraction of the entire spherical surface area inscribed in the quadrangle can be calculated:

$$\text{area} = \text{areaquad}(15, 0, 45, 30)$$

$$\text{area} = 0.0187$$



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 contains a function that provides almanac data on the major bodies of our solar system. Basic geometric parameters, such as ellipsoid vectors, radii, surface areas, and volumes, can be accessed for the Sun, the Earth's moon, and all of the planets, in any of the supported units of distance measurement.

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

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

```
ans =  
    3443.92      0.08
```

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

```
ans =  
    3396.90      0.11
```

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

```
ans =  
    1079.97      0
```

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

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

```
ans =  
    2439
```

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

```
ans =  
    1
```

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

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

```
ans =  
    1.4441e+08
```

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

```
ans =  
    1.0833e+12
```

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

```
ans =  
    1.0832e+12
```

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



# Creating and Viewing Maps

---

Mapping Toolbox provides many ways to control displays of geospatial data. This chapter provides an overview of the most important functions and associated interfaces for displaying and interacting with vector and raster geodata.

Introduction to Mapping Graphics (p. 4-2)	Understanding Mapping Toolbox functions as extensions of MATLAB graphics
Using worldmap and usamap (p. 4-4)	Generating maps with worldmap and usamap
Axes for Drawing Maps (p. 4-12)	Creating and handling map axes objects with axesm, setm, and getm
Controlling Map Frames and Grids (p. 4-31)	Controlling your window on the world and its appearance
Displaying Vector Data with Mapping Toolbox Functions (p. 4-43)	Creating maps of line and patch data with Mapping Toolbox functions
Displaying Data Grids (p. 4-53)	Creating maps of raster geodata with Mapping Toolbox functions
Interacting with Displayed Maps (p. 4-61)	Using functions and interfaces to place text, tracks, and circles, and manipulating mapped objects

## 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.

With Mapping Toolbox, you can display geographic information almost as easily as you can plot tabular or time-series data in MATLAB. 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 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, Mapping Toolbox redraws the map with the new settings, undoing any cuts or trims if necessary. See “Accessing, Computing, and Inverting Map Projection Data” on page 8-38 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, Mapping 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” on page 4-4
“Using worldmap” on page 4-5
“Using usamap” on page 4-7

### 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.

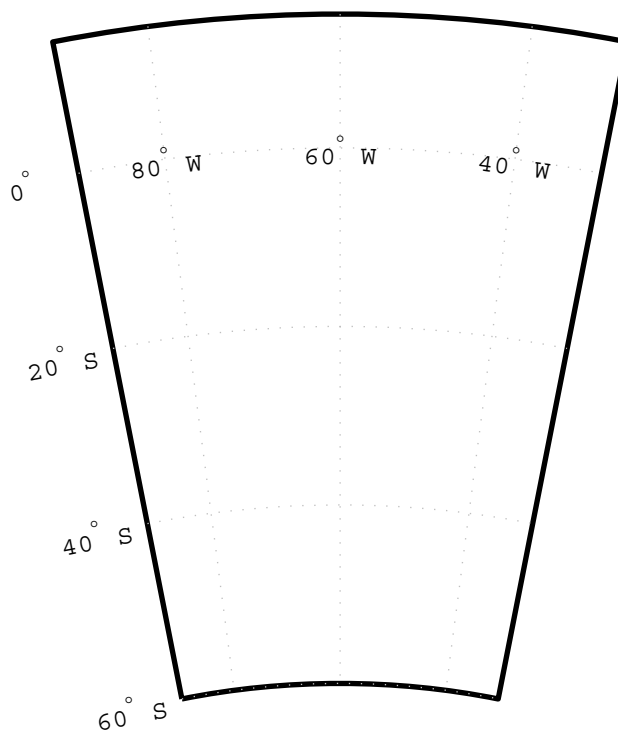
You can also use the Property Editor, part of the MATLAB plotting tools, to modify background colors for figures and axes. See “Plotting Tools — Interactive Plotting” in the MATLAB Graphics documentation for more information.

## 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')
```

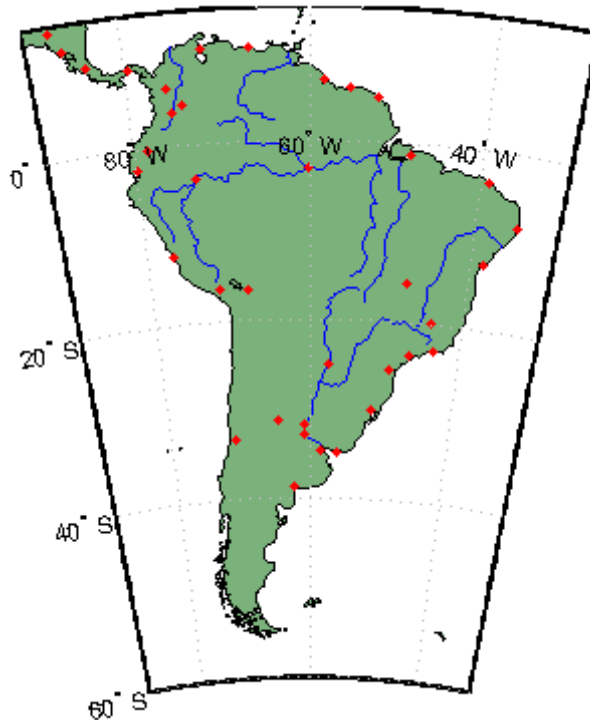
```
ans =  
eqdconic
```

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

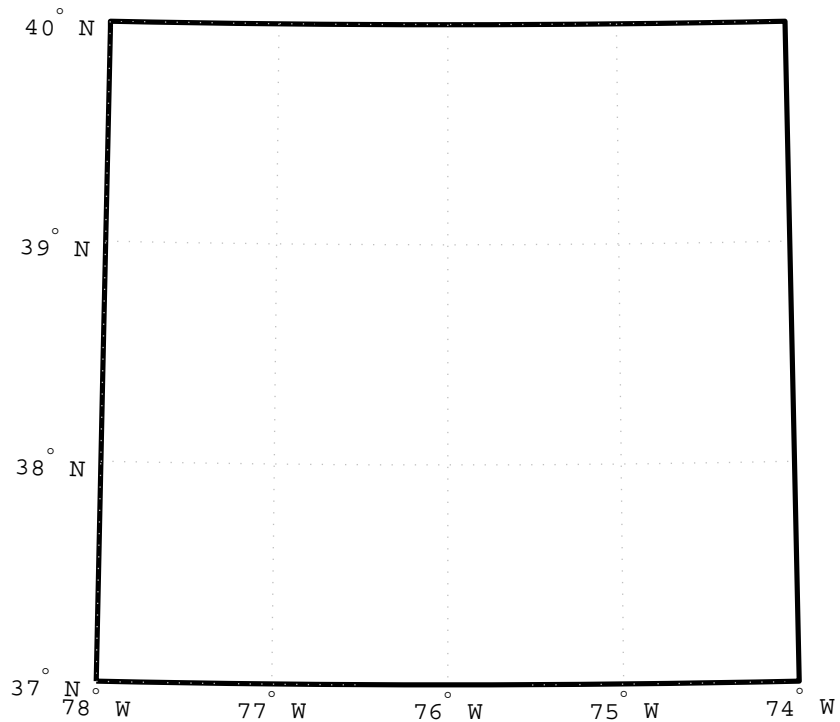
```
usamap
```

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:

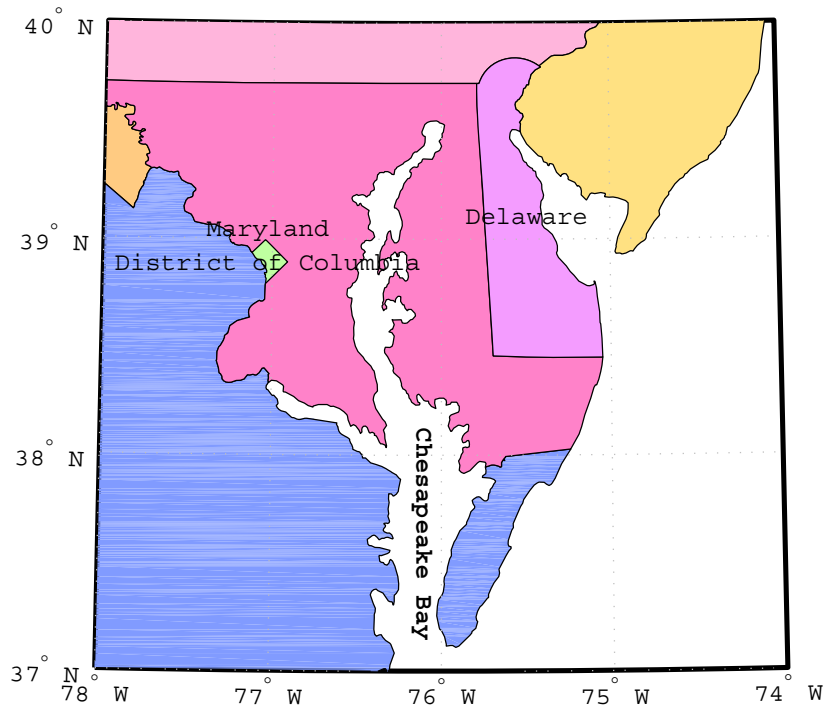
```
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], ...
    'FaceColor', polcmap(numel(states))});
```

- 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?” on page 4-12

“Using axesm” on page 4-13

“Accessing and Manipulating Map Axes Properties” on page 4-13

“Switching Between Projections” on page 4-18

“Projected and Unprojected Graphic Objects” on page 4-22

### What Is a Map Axes?

When you create a map, you can use one of Mapping Toolbox’s 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 provides 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 create 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 and 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)
```

## 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 for figures 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'
falseeastng: []
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.

### 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 redefine the projection. The rest of this section describes the considerations and parameters involved in switching projections in a map axes. Additional

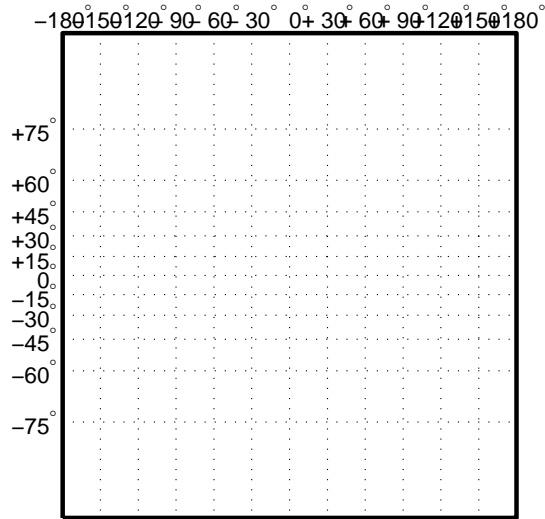
details are given for doing this with the `geoshow` function in “Changing Map Projections when Using `geoshow`” on page 4-25.

When you switch from one projection to another, you might need to change some of the map axes properties to achieve proper appearance. Settings that are suitable for one projection might not be appropriate for another. Some projections have default properties that define *that* particular projection and cannot be altered; for example, the Balthasart cylindrical projection is defined to have standard parallels (`MapParallels`) at 50°. Other projections have default properties that are initially set for proper world display; for example, the Mercator projection limits the latitude range to  $\pm 86^\circ$  to avoid “blowing up” at the poles.

Although similar projections can share the same set of properties (Miller cylindrical and Plate Carrée cylindrical), others can be drastically different (polyconic and stereographic azimuthal). The classification of map projections provided in “Summary and Guide to Projections” on page 8-64 is a rough guide to whether map projection parameters might need to be modified. For instance, switching from a cylindrical to an azimuthal projection requires a few modifications, as the following examples indicate:

**1** Create a Mercator projection with meridian and parallel labels:

```
axesm mercator
framem on; gridm on; mlabel on; plabel on
setm(gca, 'LabelFormat', 'signed')
```



- 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 86° 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')
```

What happened to the map frame and labels? If you recall, the frame latitude limits have not been changed and still correspond to the default values for a Mercator projection, as do all the other properties.



- 4** Only those properties that are required to have values are updated for the current projection. Among those that need not be are the latitude and longitude limits. Use `getm` to see their settings:

```
getm(gca, 'FLatLimit')
ans =
-86    86
```

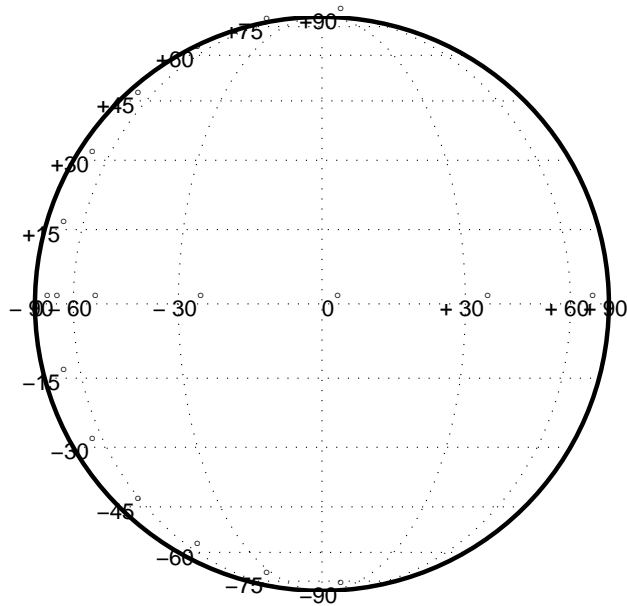
- 5** You must manually reset the frame and map limits to appropriate values for an orthographic projection so that the circular frame is displayed. If you don't know the default or appropriate numeric values, provide an empty matrix for any of the property values:

```
setm(gca, 'FLatLimit', [], 'MapLatLimit', [])
[getm(gca, 'MapLatLimit'); getm(gca, 'FLatLimit')]
ans =
-90    90
-Inf   89
```

- 6** You also need to manually specify the locations of the meridian and parallel labels (see “Labeling Grids” on page 4-41):

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

Now the map is displayed correctly, with the frame:



You can reset default property values to new values by specifying empty matrices, as shown in the last example. You can reset the entire set of properties to default values by using the **Reset** button on the axesmui GUI.

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

### Projected and Unprojected Graphic Objects

Many graphic functions in Mapping Toolbox 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 one of those built into Mapping Toolbox, you can use its parameters to project geodata in geographic coordinates to display it in the same axes. For additional information, see “Using Cartesian MATLAB Display Functions” on page 6-28.

There are four common use cases for changing a map projection in a map axes with `setm` or for reprojecting map data plotted on a regular MATLAB axes:

Mapping Use Case	Type of Axes	Reprojection Behavior
Plot geographic ( <i>latitude-longitude</i> ) vector coordinate data or data grid using a Mapping Toolbox function from releases prior to Version 2 (e.g., <code>plotm</code> )	Map axes	Automatic reprojection
Plot geographic vector data with <code>geoshow</code>	Map axes	No automatic reprojection; delete graphics objects prior to changing the projection and redraw them afterwards.
Plot data grids, images, and contours with geographic coordinates with <code>geoshow</code>	Map axes	Automatic reprojection; this behavior could change in a future release
Plot projected ( <i>x-y</i> ) vector or raster map data with <code>mapshow</code> or with a MATLAB graphics function (e.g., <code>line</code> , <code>contour</code> , or <code>surf</code> )	Regular axes	Manual reprojection (reproject coordinates with <code>minvtran</code> / <code>mfwdtran</code> or <code>projinv</code> / <code>projfwd</code> ); 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 4-67 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')`).

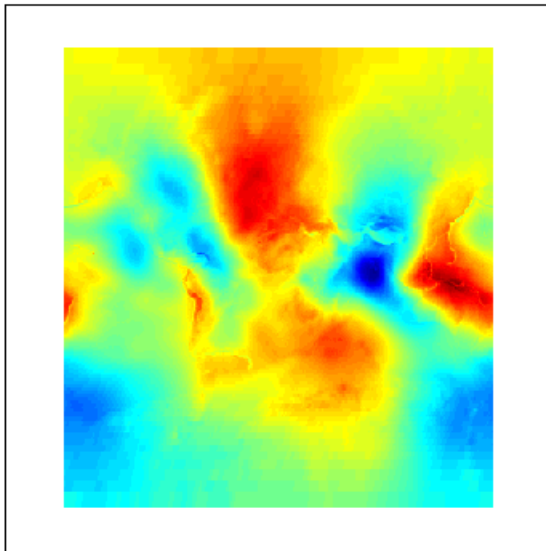
You can also use `geoshow` to display latitude-longitude data on a regular axes (created by the `axes` function, for example). When you do this, the

latitude-longitude data are displayed using a Plate Carrée Projection, which linearly maps longitude to  $x$  and latitude to  $y$ .

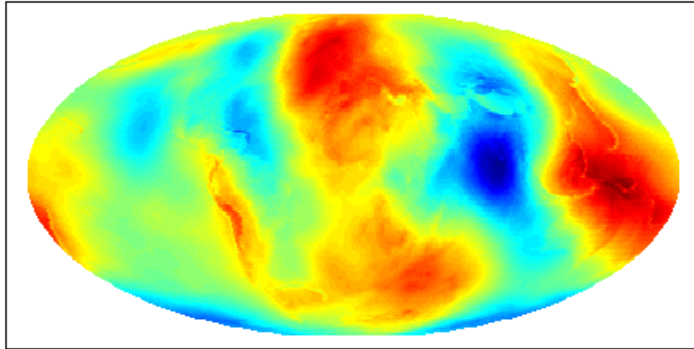
If you are using `geoshow` with a map axes and want to change the map projection after you have displayed data in geographic coordinates, do the following, depending on whether the data are raster or vector:

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

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

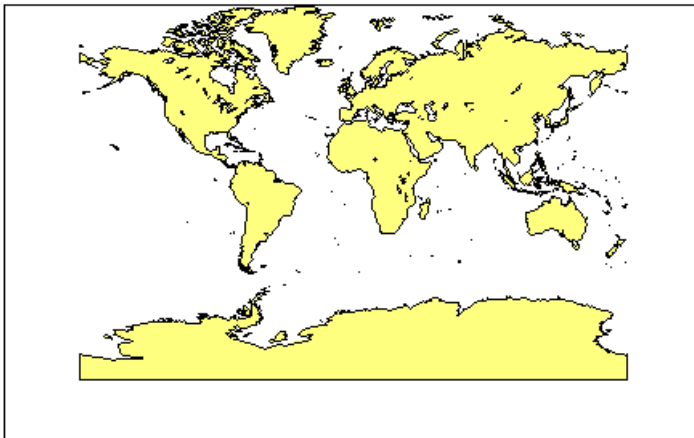


```
setm(gca,'mapprojection','mollweid')
```

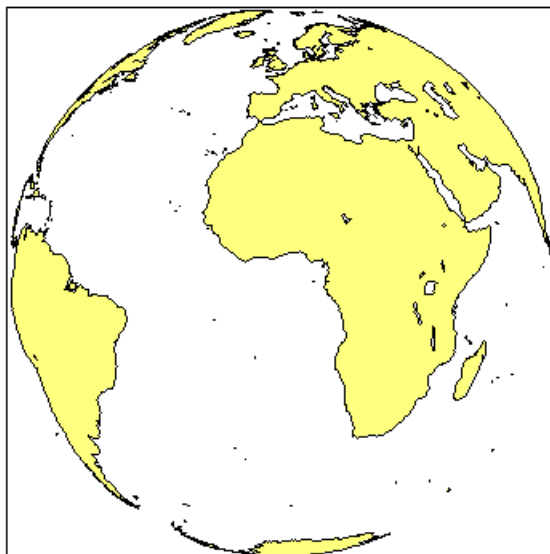


**Vector Data.** Obtain handles to the line or patch graphic objects, delete the objects from the axes, change the projection using `setm`, and replot the vector data using `geoshow`:

```
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;
```

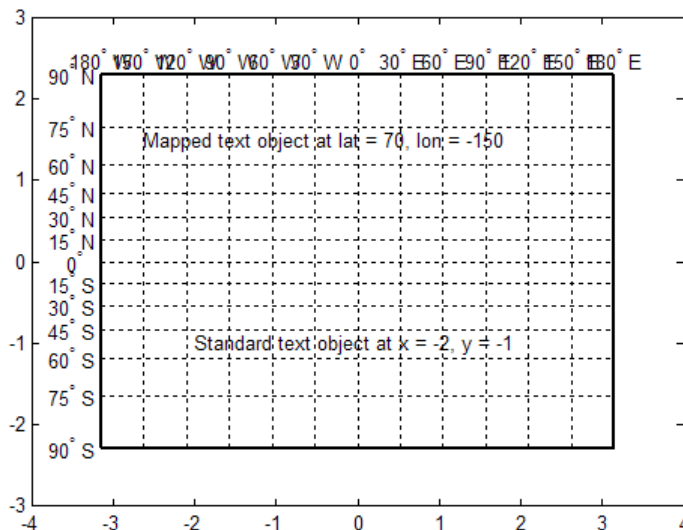


These function calls create a map axes object, a map frame enclosing the region of interest, and geographic grid lines. The  $x$ - $y$  axes, which are normally hidden, are displayed, and the axes  $x$ - $y$  grid is turned off. The Mapping Toolbox function `gridm` constructs lines to illustrate the 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 latitude-longitude grid (or *graticule*) can contain curves while a MATLAB grid never does. For more information about graticules, see “The Map Grid” on page 4-38.

- 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')
```

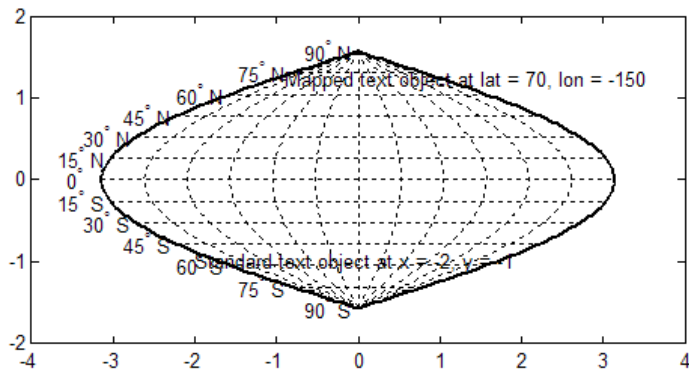
In the figure, shown below, a standard text object is placed at  $x=-2$  and  $y=-1$ , while a mapped text object is placed at  $(70^\circ\text{N}, 150^\circ\text{W})$  in the Miller projection.



- 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 4-43 for information on plotting vector geodata, and “Displaying Data Grids” on page 4-53 for information on plotting raster geodata.

## Controlling Map Frames and Grids

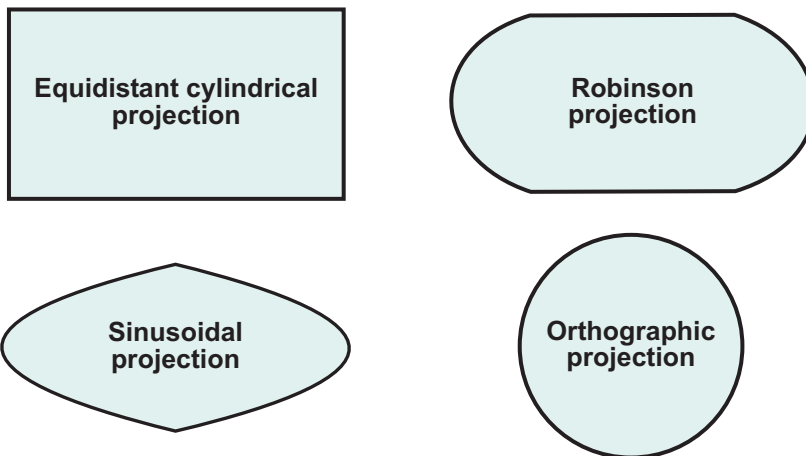
### In this section...

“The Map Frame” on page 4-31

“The Map Grid” on page 4-38

### The Map Frame

In Mapping Toolbox, the *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 360° 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.



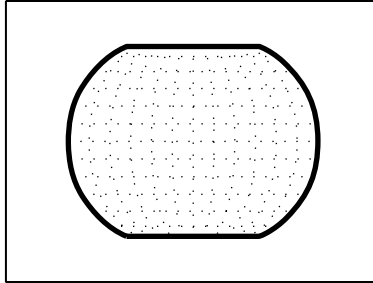
### Full-World Map Frames

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, `FAtLimit` is `[-90 90]`, and

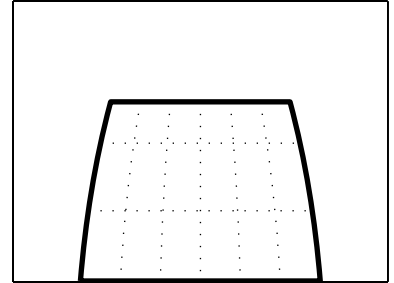
FLonLimit is [-180 180] for each case. The frame quadrangle can encompass smaller regions, as well, in which case the shape is a section of a full-world outline or simply a quadrilateral with straight or curving sides. Execute this code to produce the figure that follows:

```
% Plot four regions of Robinson frame and grid using map limits
%
figure('color','white')
% Default map frame
subplot(2,2,1);
axesm('MapProjection','robinson',...
      'Frame','on','Grid','on')
title('Latitude [-90 90], Map lons [-180 180]','FontSize',10)
%
subplot(2,2,2);
axesm('MapProjection','robinson',...
      'MapLatLimit',[30 70],'MapLonLimit',[-90 90],...
      'Frame','on','Grid','on')
title('Latitude [30 70], Longitude [-90 90]','FontSize',10)
%
subplot(2,2,3);
axesm('MapProjection','robinson',...
      'MapLatLimit',[-90 0],'MapLonLimit',[-180 -30],...
      'Frame','on','Grid','on')
title('Latitude [-90 0], Longitude [-180 -30]','FontSize',10)
%
subplot(2,2,4);
axesm('MapProjection','robinson',...
      'MapLatLimit',[-70 -30],'MapLonLimit',[60 150],...
      'Frame','on','Grid','on')
title('Latitude [-70 -30], Longitude [60 150]','FontSize',10)
```

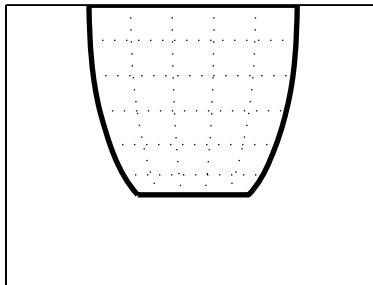
Latitude [-90 90], Map lons [-180 180]



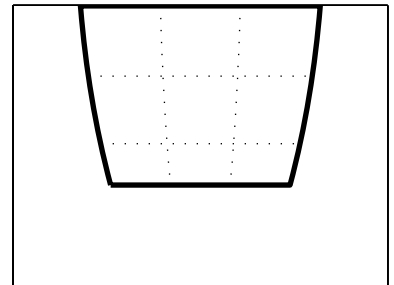
Latitude [30 70], Longitude [-90 90]



Latitude [-90 0], Longitude [-180 -30]



Latitude [-70 -30], Longitude [60 150]



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

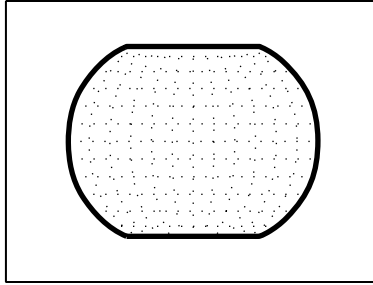
For the frames shown above, the projection is centered on the prime meridian, or 0 longitude. Such a frame would be the result of creating a map axes with the defaults for the Robinson projection and then resetting the frame limits to cover just part of the world.

When you want your frame to be symmetric about the region of interest, let axesm determine the proper settings for you. If you specify the map limits without specifying the map origin and frame limits, axesm will automatically set the appropriate values for a proper symmetric frame.

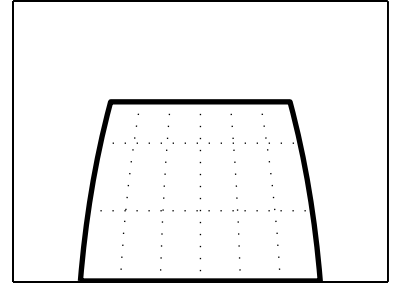
In the following example, the axes limits are set using `setm` after the Robinson map axes is created. Note that map axes properties that concern frames begin with “F”:

```
% Same regions as above, but with frame limits
%   altered after projecting
%
figure('color','white')
% Default frame limits
h11 = subplot(2,2,1);
axesm('MapProjection','robinson',...
      'Frame','on','Grid','on')
title('Latitude [-90 90], Longitude [-180 180]')
%
h12 = subplot(2,2,2);
axesm('MapProjection','robinson',...
      'Frame','on','Grid','on')
setm(h12,'FLatLimit',[30 70],'FLonLimit',[-90 90])
title('Latitude [30 70], Longitude [-90 90]')
%
h21 = subplot(2,2,3);
axesm('MapProjection','robinson',...
      'Frame','on','Grid','on')
setm(h21,'FLatLimit',[-90 0],'FLonLimit',[-180 -30])
title('Latitude [-90 0], Longitude [-180 -30]')
%
h22 = subplot(2,2,4);
axesm('MapProjection','robinson',...
      'Frame','on','Grid','on')
setm(h22,'FLatLimit',[-70 -30],'FLonLimit',[60 150])
title('Latitude [-70 -30], Longitude [60 150]')
```

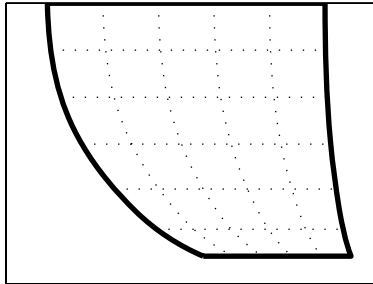
Latitude [-90 90], Longitude [-180 180]



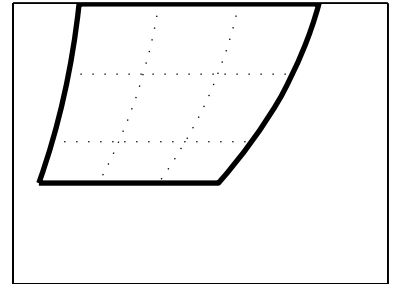
Latitude [30 70], Longitude [-90 90]



Latitude [-90 0], Longitude [-180 -30]



Latitude [-70 -30], Longitude [60 150]



**Frame Quadrangles in the Robinson Projection (Symmetric About Map Limits)**

The differences between the two examples are obvious when projections are not centered on the prime meridian. If you wanted to create a symmetric frame in the lower right subplot of the above figure, set the map limits with `axesm` instead of using `setm`:

```
axesm('MapProjection','robinson',...
      'MapLatLimit',[-70 -30],...
      'MapLonLimit',[60 150],...
      'Frame','on')
```

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

In Mapping Toolbox, the 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.

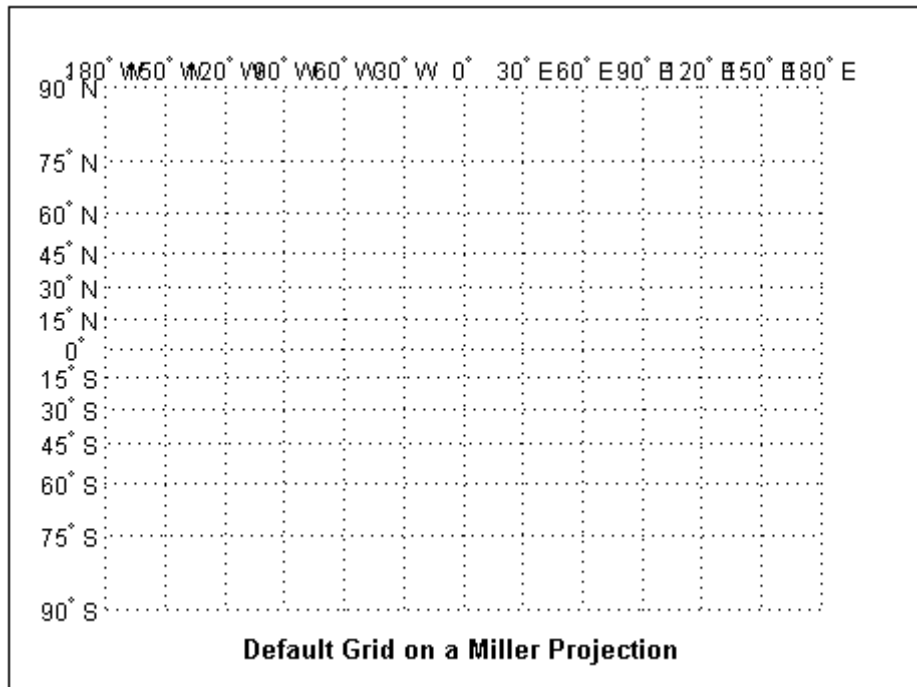
---

### 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  $30^\circ$  for meridians and every  $15^\circ$  for parallels.



### Grid Layering

By default, the grid is placed as the top layer of any display. You can alter this by changing the `GAltitude` property, so that other map objects can be placed “above” the grid. The new grid is drawn at its new altitude. The units used for `GAltitude` are specified with the `daspectm` function.

To reposition the grid back to the top of the display, use the command `gridm reset`. You can also control the appearance of grid lines with the `GLineStyle` and `GLineWidth` properties, which are `' : '` and `0.5`, respectively, by default.

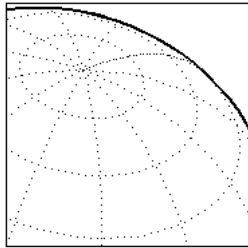
### Limiting Grid Lines

The Miller projection is an example in which all the meridians can extend to the poles without appearing to be cluttered. In other projections, such as the orthographic (below), the map grid can obscure the surface where they

converge. Two map axes properties, `MLineLimit` and `MLineException`, enable you to control such clutter:

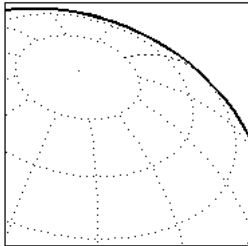
- Use the `MLineLimit` property to specify a pair of latitudes at which to terminate the meridians. For example, setting `MLineLimit` to `[-75 75]` completely clears the region above and below this latitude range of meridian lines.
- If you want some lines to reach the poles but not others, you can specify them with the `MLineException` property. For example, if `MLineException` is set to `[-90 0 90 180]`, then the meridians corresponding to the four cardinal longitudes will continue past the limit on to the pole.

The use of these properties is illustrated in the figure below. Note that there are two corresponding map axes properties, `PLineLimit` and `PLineException`, for controlling the extent of displayed parallels.



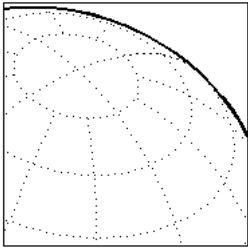
Default grid allows all displayed meridians to extend to the poles:

```
axesm('MapProjection','ortho',...
      'Origin',[40,40,14],...
      'Grid','on','Frame','on');
```



The property `MLineLimit` truncates meridians at given latitudes:

```
axesm('MapProjection','ortho',...
      'Origin',[40,40,14],...
      'Grid','on','Frame','on',...
      'MLineLimit', [-75 75]);
```



The property `MLineLineException` permits certain meridians to extend to the poles, regardless of `MLineLimit`:

```
axesm('MapProjection','ortho',...
      'Origin',[40,40,14],...
      'Grid','on','Frame','on',...
      'MLineLimit', [-75 75],...
      'MLineException', [-90 0 90 180]);
```

## Labeling Grids

You can label displayed parallels and meridians. `MeridianLabel` and `ParallelLabel` are on-off properties for displaying labels on the meridians and parallels, respectively. They are both 'off' by default. Initially, the label locations coincide with the default displayed grid lines, but you can alter this by using the `PlabelLocation` and `MlabelLocation` properties. These grid lines are labeled across the north edge of the map for meridians and along the west edge of the map for parallels. However, the property `MlabelParallel` allows you to specify 'north', 'south', 'equator', or a specific latitude at which to display the meridian labels, and `PlabelMeridian` allows the choice of 'west', 'east', 'prime', or a specific longitude for the parallel labels. By default, parallel labels are displayed in the range of 0° to 90° north and south of the equator while meridian labels are displayed in the range of 0° to

180° east and west of the prime meridian. You can use the `mlabelzero22pi` function to redisplay the meridian labels in the range of 0° to 360° east of the prime meridian.

Properties affecting grid labeling are listed below.

<b>Property</b>	<b>Effect</b>
<code>MeridianLabel</code>	Toggle display of meridian labels
<code>ParallelLabel</code>	Toggle display of parallel labels
<code>MlabelLocation</code>	Alternate interval for labeling meridians
<code>PlabelLocation</code>	Alternate interval for labeling parallels
<code>MlabelParallel</code>	Keyword or latitude for placing meridian labels
<code>PlabelMeridian</code>	Keyword or longitude for placing parallel labels
<code>mlabelzero22pi(function)</code>	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” on page 4-43

“Displaying Vector Maps as Lines” on page 4-43

“Displaying Vector Maps as Lines or Patches” on page 4-46

### 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 the functions available in Mapping Toolbox and MATLAB, you can create and customize more elaborate maps interactively by entering commands in the Command Window or by writing M-code in functions and scripts. This section describes how to use the principal mapping functions for displaying vector geospatial data. The following section describes displaying raster map data.

### Displaying Vector Maps as Lines

Mapping Toolbox lets you display vector map data as line objects much like the line display functions in MATLAB. 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
<code>contourm</code>	Contour plot of map data
<code>contour3m</code>	Contour plot of map data in 3-D space
<code>geoshow</code>	High-level function to plot points, lines, patches, grids, and georeferenced images in geocoordinates

Function	Used For
<code>linem</code>	Draws line objects projected on map axes
<code>mapshow</code>	High-level function to plot points, lines, patches, grids, and georeferenced images in plane coordinates
<code>plotm</code>	Clears figure and draws line objects projected on map axes
<code>plot3m</code>	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.

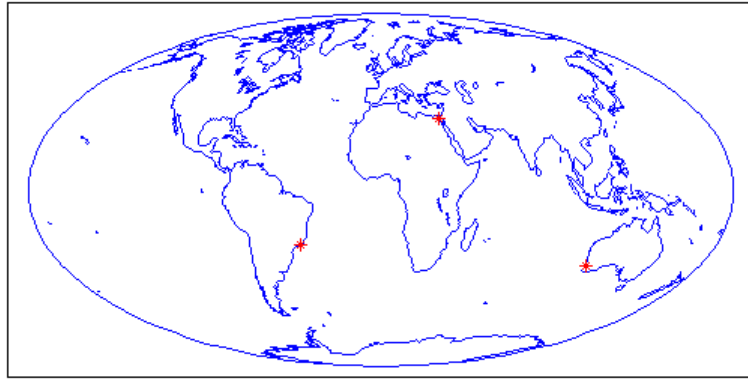
```
plotm(lat,long,'LineWidth',1,'Color','blue')
```

Sometimes vector data represents specific points. Suppose you have variables representing the locations of Cairo (30°N,32°E), Rio de Janeiro (23°S,43°W), and Perth (32°S,116°E), 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:

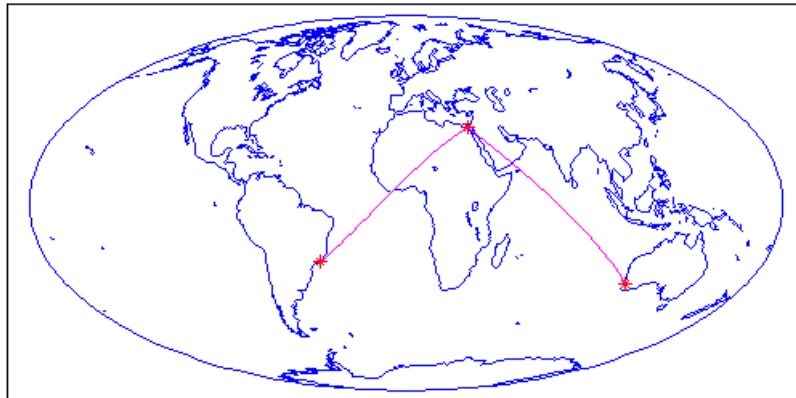
```
citylats = [30 -23 -32]; citylongs = [32 -43 116];
plotm(citylats,citylongs,'r*')
```





- 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 Version 2 geostructs as input data.

---

### 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 or geostructs. 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 NaNs), 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 NaNs) for the Great Lakes. The variables `statelat` and `statelon` contain line-segment data

(separated by NaNs) for the borders between states, which is not formatted for patch display.

- 2** Verify that line and polygon data contains NaNs (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 Version 2 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 how Mapping Toolbox structures geographic data, see “Understanding Vector Data” on page 2-13 and “Understanding Raster Data” on page 2-27.

- 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`:

- 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)
```

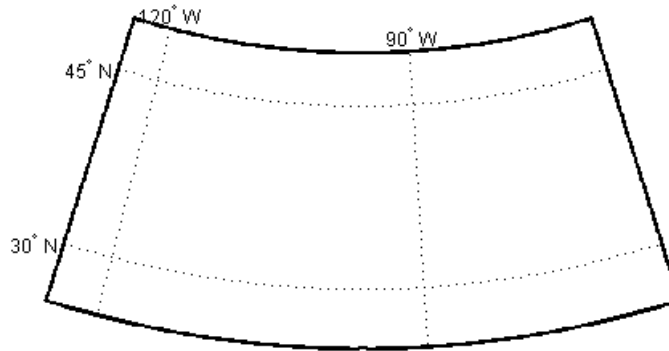
- c** If you want to add white space around the map, you can do so as follows (here, 2 degrees are added):

```
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:

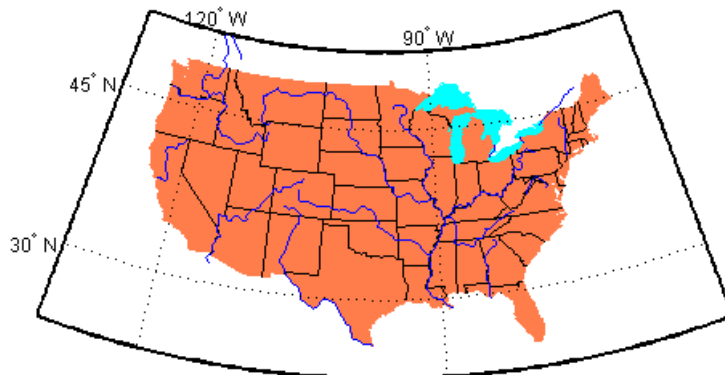
```
geoshow(gtlakelat,gtlakelon, 'DisplayType','polygon',...  
        'FaceColor','cyan', 'EdgeColor','none')
```

- 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
<code>fillm</code>	Filled 2-D map polygons
<code>fill3m</code>	Filled 3-D map polygons in 3-D space
<code>geoshow</code>	Display map latitude and longitude data in 2-D
<code>mapshow</code>	Display map data without projection in 2-D
<code>patchm</code>	Patch objects projected on map axes
<code>patchesm</code>	Patches projected as individual objects on map axes

The `fillm` function makes use of the low-level function `patchm`. Mapping 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 uses NaNs to separate unconnected patch faces, unlike MATLAB, which does not handle NaN clipped 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 `geostruct2` geographic data structures 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” on page 4-53

“Fitting Gridded Data to the Graticule” on page 4-54

“Using Raster Data to Create 3-D Displays” on page 4-57

### Types of Data Grids and Raster Display Functions

Mapping Toolbox provides functions for the display and enhancement of 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, Mapping Toolbox grid function names are mostly identical to their MATLAB counterparts, with an `m` appended.

---

**Note** In 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
surf1m	3-D shaded surface with lighting projected on map axes
mesh1m	3-D lighted shaded relief of regular data grid
surf1m	3-D lighted shaded relief of geolocated data grid

## Fitting Gridded Data to the Graticule

Mapping 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, Mapping 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.

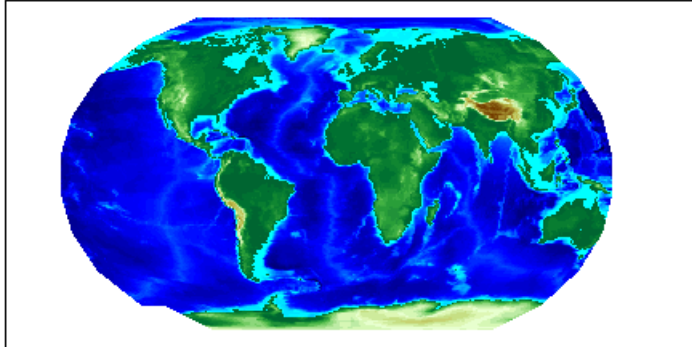
In mapping data grids using the toolbox, as in traditional cartography, the finer the mesh (analogous to using a graticule with more meridians and parallels), the greater precision the projected map display will have, at the cost of greater effort and time. The graticule in a printed map is analogous to the spacing of grid elements in a regular data grid, which Mapping Toolbox represents as 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, where

*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 equiangular mesh.

In other words, while the structure of cells for regular data grids is restricted to equal-angle quadrangles (i.e., length of cell in latitude must equal length of cell in longitude), geolocated data grids have no such constraints. Their cells can be of any size.

The topo regular data grid can be displayed quickly using a coarse graticule, at a cost in precision of representation. Observe the map that results from the following commands:

```
load topo                                %Get data grid and ref vec
figure; axesm robinson                    %Set up Robinson proj
spacing = [10 20];                        %Spec a 10x20 cell grid
h = meshm(topo,topolegend,spacing);       %Draw data into grid
demcmap(topo)                             %Set DEM color map
```

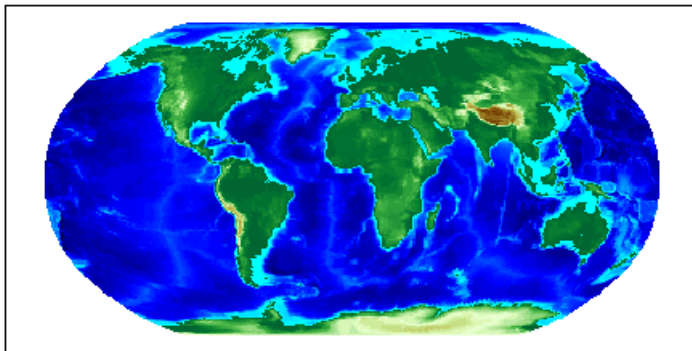


Notice that for this coarse graticule, the edges of the map do not appear as smooth curves. What might not be as obvious is that the easternmost column of graticule cells and the southwesternmost cell are sometimes invisible on displayed data grids. This is necessary for the proper projection of the surface object and is not a concern except with the coarsest graticules. 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 projection fidelity, the resolution of its value data is 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 mapping `setm` function allows the `MeshGrat` graticule property to be manipulated for regular matrix surfaces. 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 tradeoff between resolution and time, doing this takes longer to display the map:

```
setm(h,'MeshGrat',[200 400])
```



Another way you can do this is with the `meshgrat` function:

```
[latgrat,longrat] = meshgrat(topo,topolegend,[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 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,maplegend);
```

- 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 array 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.

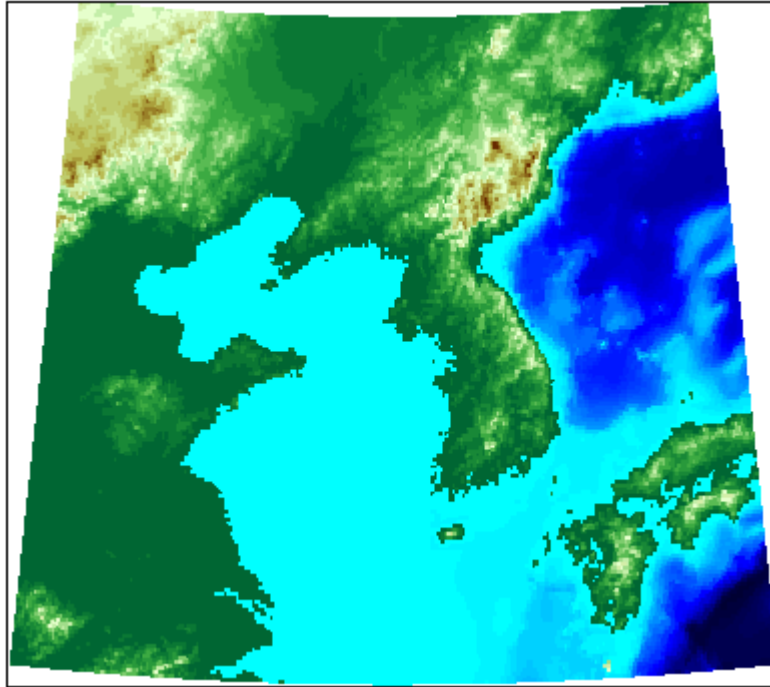
**4** Now set up a map axes object with the equal area conic projection:

```
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.

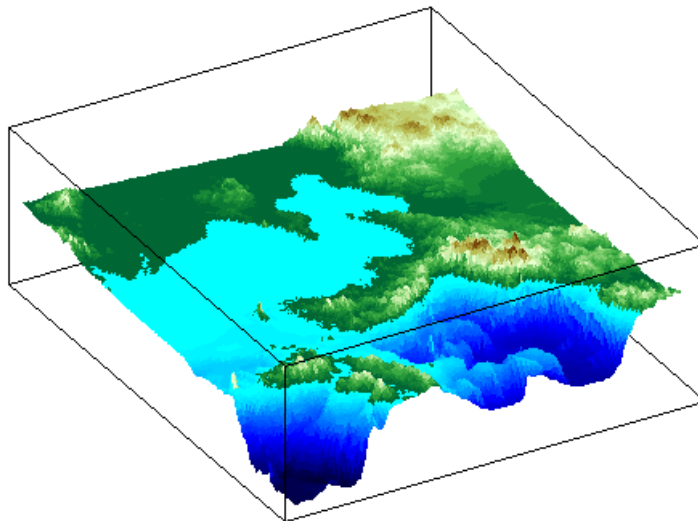


Be aware, however, that this map is really a 3-D view seen from directly overhead (the default perspective). To appreciate that, all you need to do is to change your viewpoint.

- 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:



Mapping Toolbox provides many other controls over perspective map representations. See Chapter 5, “Making Three-Dimensional Maps” for additional help on constructing 3-D views.



## Interacting with Displayed Maps

### In this section...

“Picking Locations Interactively” on page 4-61

“Defining Small Circles and Tracks Interactively” on page 4-63

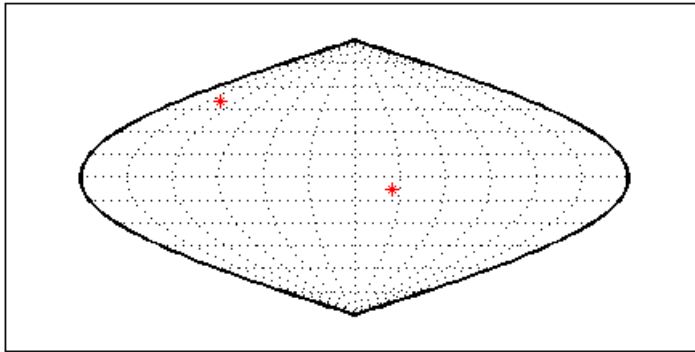
“Working with Objects by Name” on page 4-66

### Picking Locations Interactively

You can use Mapping Toolbox to interact with maps, both in `mapview` and in figures created with `axesm`. This section describes two useful graphic input functions, `inputm` and `gcpmap`. The `inputm` function (analogous to the MATLAB `ginput` function) allows you to get the latitude-longitude position of a mouse click. The `gcpmap` function (analogous to the MATLAB function `get(gca, 'CurrentPoint')`) returns the current mouse position, also in latitude and longitude.

Explore `inputm` with the following commands, which display a map axes with its grid and then request three mouse clicks, the locations of which are stored as geographic coordinates in the variable `points`. Then the `plotm` function plots the points you clicked as red markers. The display you see depends on the points you select:

```
axesm sinusoid
framem on; gridm on
points=inputm(3)
points =
    -41.7177  -145.0293
     7.9211   -0.5332
    38.5492  149.2237
plotm(points, 'r*')
```



---

**Note** If you click outside the map frame, `inputm` returns a valid but incorrect latitude and longitude, even though the point you indicated is off the map.

---

One reason you might want to manually identify points on a map is to interactively explore how much distortion a map projection has at given locations. For example, you can feed the data acquired with `inputm` to the `distortcalc` function, which computes area and angular distortions at any location on a displayed map axes. If you do so using the `points` variable, the results of the previous three mouse clicks are as follows:

```
[areascale,angledef] = distortcalc(points(1,1),points(1,2))
areascale =
    1.0000
angledef =
    85.9284
>> [areascale,angledef] = distortcalc(points(2,1),points(2,2))
areascale =
    1.0000
angledef =
    3.1143
[areascale,angledef] = distortcalc(points(3,1),points(3,2))
areascale =
    1.0000
angledef =
    76.0623
```

This indicates that the current projection (sinusoidal) has the equal-area property, but exhibits variable angular distortion across the map, less near the equator and more near the poles.

To see a working application that uses the `inputm` function, view and run the `mapexfindcity` Interactive Global City Finder demo.

## Defining Small Circles and Tracks Interactively

Geographic line annotations such as navigational tracks and small circles can be generated interactively. Great circle tracks are the shortest distance between points, and when closed partition the Earth into equal halves; a small circle is the locus of points at a constant distance from a reference point. Use `trackg` and `scircleg` to create them by clicking on the map. Double-click the tracks or circles to modify the lines. **Shift**+click to type specific parameters into a control panel. The control panels also allow you to retrieve or set properties of tracks and circles (for instance, great circle distances and small circle radii).

The following example illustrates how to interactively create a great circle track from Los Angeles, California, to Tokyo, Japan, and a 1000 km radius small circle centered on the Hawaiian Islands. The track is made via the `trackg` function, which prompts you to select endpoints for a track with the mouse. The `scircleg` function prompts for two points also, a center and any point on the circumference of the small circle. The specifics of the track and the circle are then adjusted more precisely with dialog controls:

- 1 Set up an orthographic view centered over the Pacific Ocean. Use the coast MAT-file:

```
axesm('ortho','origin',[30 180])
framem;gridm
load coast
plotm(lat,long,'k')
```

- 2 Create a track with the `trackg` function, which prompts for two endpoints. The default track type is a great circle:

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

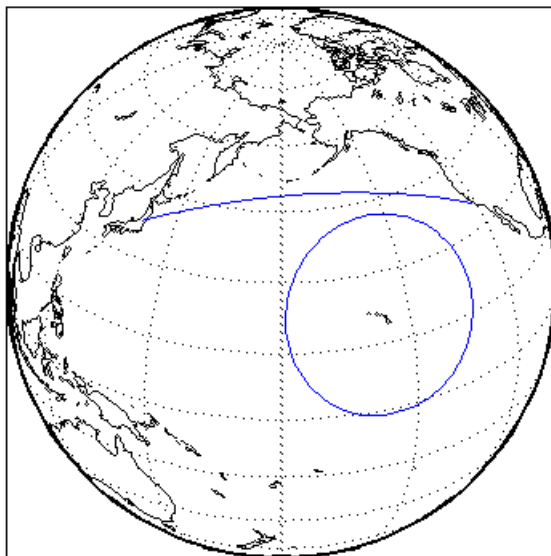
Click near Los Angeles and Tokyo, and the track is drawn.

- 3 Now create a small circle around Hawaii with the `scircleg` function, which prompts for a center point and a point on the perimeter. Make the circle's radius about 2000 km, but don't worry about getting the size exact:

```
scircleg
```

```
Circle 1: Click on center and perimeter
```

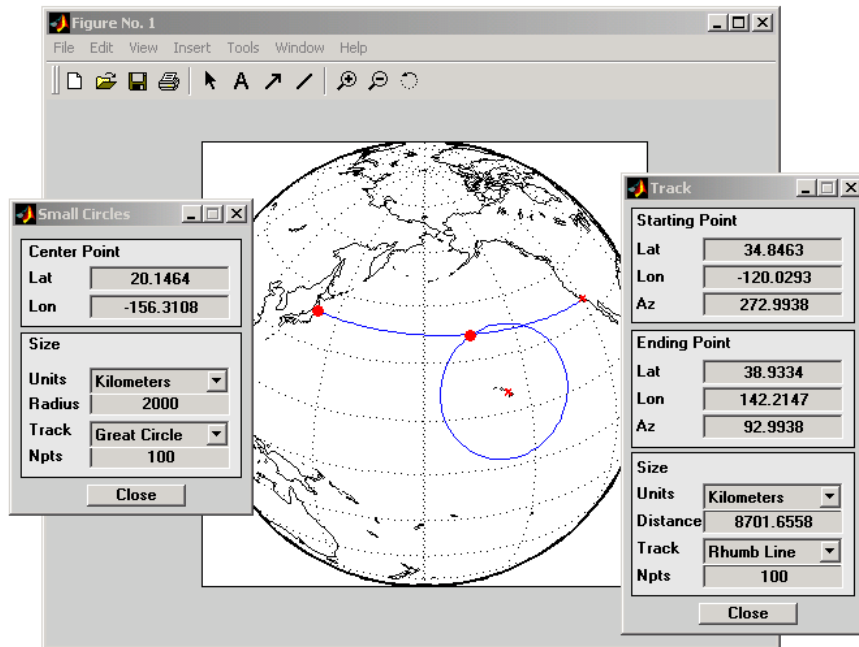
The map should look approximately like this.



- 4 Adjust the size of the small circle to be 2000 km by **Shift**+clicking anywhere on its perimeter. The Small Circles dialog box appears.
- 5 Type 2000 into the **Radius** field.
- 6 Click **Close**. The small circle readjusts to be 2000 km around Hawaii.
- 7 To adjust the track between Los Angeles and Tokyo, **Shift**+click on it. This brings up the Track dialog, with which you specify a position and initial azimuth for either endpoint, as well as the length and type of the track.

- 8 Change the track type from Great Circle to Rhumb Line with the Track pop-up menu. The track immediately changes shape.
- 9 Experiment with the other Track dialog controls. Also note that you can move the endpoints of the track with the mouse by dragging the red circles, and obtain the arc's length in various units of distance.

The following figure shows the Small Circles and Track dialog boxes.



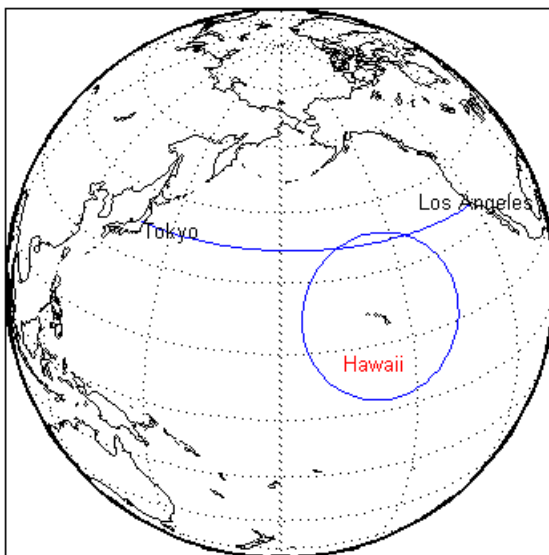
### Interactive Text Annotation

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 4-28. 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

Mapping Toolbox allows you to manipulate displayed objects by name. Many mapping 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.

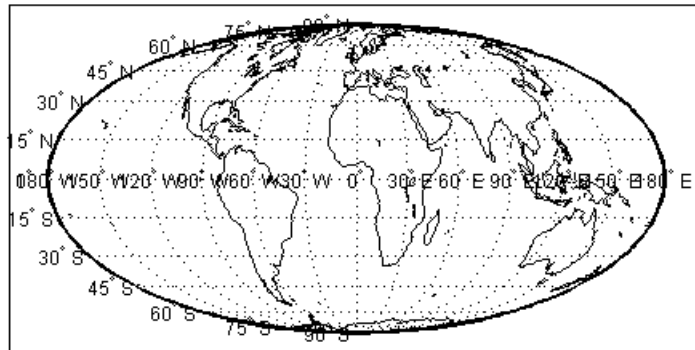
Some mapping display functions like `framem`, `gridm`, and `contourm` assign object tags by default. You can also set the name upon display by assigning a string to the `Tag` property in mapping display functions that use property name/property value pairs. If the `Tag` does not contain a string, the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



# Making Three-Dimensional Maps

---

Mapping Toolbox constructs three-dimensional as well as two-dimensional map displays. Any map can be constructed and viewed in three dimensions. Some thematic mapping functions plot 3-D symbolism. The most common 3-D application is terrain visualization, for which terrain data grids supply the altitude data. This chapter describes how to obtain and work with terrain data, techniques for making 3-D surface representations, and continues on to describe ways to drape other data over terrain, and how to shade, light, and view both planimetric and spherical 3-D relief displays.

Sources of Terrain Data (p. 5-2)

Notes on terrain data available from U.S. mapping agencies

Reading Elevation Data Interactively (p. 5-13)

Using the `dteds`, `dted`, and `demdataui` functions

Determining and Visualizing Visibility Across Terrain (p. 5-19)

Computing line-of-sight and viewsheds with `los2` and `viewshed`

Shading and Lighting Terrain Maps (p. 5-22)

Different approaches to illuminating terrain: using the `lightm`, `surf1m`, `surf1srm`, and `mesh1srm` functions

Draping Data on Elevation Maps (p. 5-40)

Using shading and color to combine surface relief with other surface characteristics to make bivariate maps

Working with the Globe Display (p. 5-49)

Visualizing around and around a round world

## Sources of Terrain Data

In this section...
“Digital Terrain Elevation Data from NGA” on page 5-2
“Digital Elevation Model Files from USGS” on page 5-3
“Determining What Elevation Data Exists for a Region” on page 5-3

### Digital Terrain Elevation Data from NGA

Nearly all published terrain elevation data is in the form of data grids. “Displaying Data Grids” on page 4-53 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

Mapping Toolbox provides several functions and a GUI to assist you in deriving file names for and managing 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 MATLAB path and indicate the names of data sets that you can download or order on magnetic media or CD-ROM.

Mapping Toolbox has utility functions for describing and importing 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*.

<b>File Type</b>	<b>Description</b>	<b>Function to Read Files</b>	<b>Function to Identify or Summarize Files</b>
DTED	U.S. Department of Defense Digital Terrain Elevation Data	dted	dteds
DEM	USGS 1-degree (3-arc-second resolution) digital elevation models	usgsdem	usgsdems
DEM24K	USGS 1:24K (30-meter resolution) digital elevation models	usgs24kdem	N/A
ETOPO5 ETOPO2	Earth Topography – 5-minute (ETOPO5) and 2-minute (ETOPO2)	etopo	N/A
GTOPO30	Tiles of 30-arc-second global elevation models	gtopo30	gtopo30s
SATBATH	Global 2-minute (4 km) satellite topography and bathymetry data	satbath	N/A
SDTS DEM	Digital elevation models in U.S. SDTS format	sdtsemread	sdtinfo (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 41.1°N to 43.9°N latitude and from 71.9°W to 69.1°W 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);
```

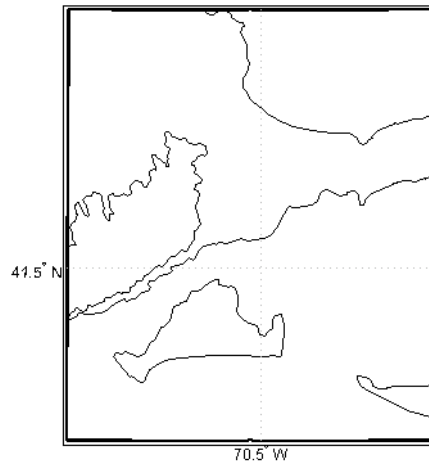
- 7** Read data for the region defined by the latitude and longitude limits from the `usastatehi` shapefile:

```
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');
```

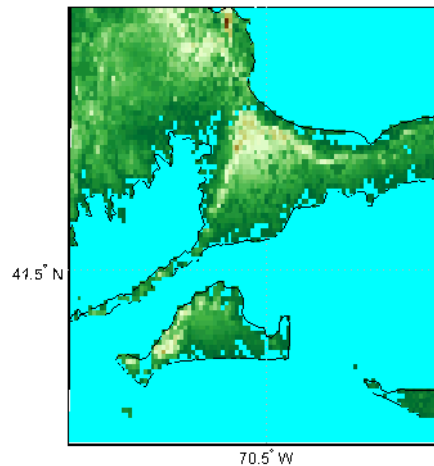
At this point the map looks like this:



- 9** Render the elevations, and set the colormap accordingly:

```
meshm(capeterrain, caperef, size(capeterrain), capeterrain);  
demcmmap(capeterrain)
```

The resulting map, shown below, is a window on Cape Cod, and illustrates the relative coarseness of DTED level 0 data.



### Mapping Multiple DTED Files with the DTED Function

When your region of interest extends across more than one DTED tile, the `dted` function concatenates the tiles into a single matrix, which can be at full resolution or a sample of every  $n$ th row and column. You can specify a single DTED file, a directory containing several files (for different latitudes along a constant longitude), or a higher level directory containing subdirectories with files for several longitude bands.

- 1 To follow this exercise, you need to acquire the necessary DTED files from the Internet as described in the following tech note

<http://www.mathworks.com/support/tech-notes/2100/2101.html>

or from a CD-ROM. This yields a set of directories that contain the following files:

```
/dted
  /w070
    n41.avg
    n41.dt0
    n41.max
    n41.min
    n43.avg
```

```
        n43.dt0
        n43.max
        n43.min
/w071
        n41.avg
        n41.dt0
        n41.max
        n41.min
        n42.avg
        n42.dt0
        n42.max
        n42.min
        n43.avg
        n43.dt0
        n43.max
        n43.min
/w072
        n41.avg
        n41.dt0
        n41.max
        n41.min
        n42.avg
        n42.dt0
        n42.max
        n42.min
        n43.avg
        n43.dt0
        n43.max
        n43.min
```

- 2** Change your working directory to the directory that includes the top-level DTED directory (which is always named dted).
- 3** Use the dted function, specifying that directory as the first argument:

```
latlim = [ 41.1  43.9];
lonlim = [-71.9 -69.1];
samplefactor = 5;
[capetopo,caperef] = dted(pwd, samplefactor, latlim, lonlim);
```

The sample factor value of 5 specifies that only every fifth data cell, in both latitude and longitude, will be read from the original DTED file. You can choose a larger value to save memory and speed processing and display, at the expense of resolution and accuracy. The size of your elevation array (`capetopo`) will be inversely proportional to the square of the sample factor.

---

**Note** You can specify a DTED filename rather than a directory name if you are accessing only one DTED file. If the file cannot be found, a file dialog is presented for you to navigate to the file you want. See the example “Mapping a Single DTED File with the DTED Function” on page 5-6.

---

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

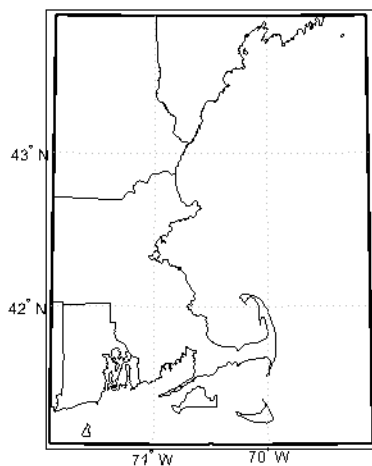
```
capetopo(capetopo==0)=-1;
```

- 5** Obtain the elevation grid’s latitude and longitude limits; use them to draw an outline map of the area to orient the viewer:

```
[latlim,lonlim] = limitm(capetopo, caperef);

figure;
ax = usamap(latlim,lonlim);
capecoast = shaperead('usastatehi',...
    'UseGeoCoords', true,...
    'BoundingBox', [lonlim' latlim']);
geoshow(ax,capecoast,'FaceColor','None');
```

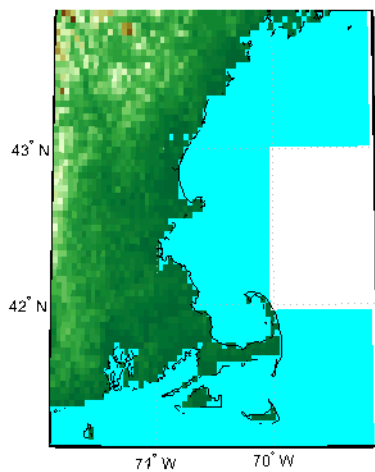
The map now looks like this.



- 6** Render the elevation grid with `meshm`, and then recolor the map with `demcmap` to display hypsometric colors (elevation tints):

```
meshm(capetopo, caperef, size(capetopo), capetopo);  
demcmap(capetopo)
```

Here is the map; note the missing tile to the right where no DTED data exists.



# Reading Elevation Data Interactively

## Extracting DEM Data with demdataui

You can browse many formats of digital elevation map data using the demdataui graphical user interface. The demdataui GUI determines and graphically depicts coverage of ETOPO5, TerrainBase, the satellite bathymetry model (SATBATH), GTOPO30, GLOBE, and DTED data sets on local and network file systems, and can import these files into the workspace.

---

**Note** When it opens, demdataui scans your MATLAB 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.

---

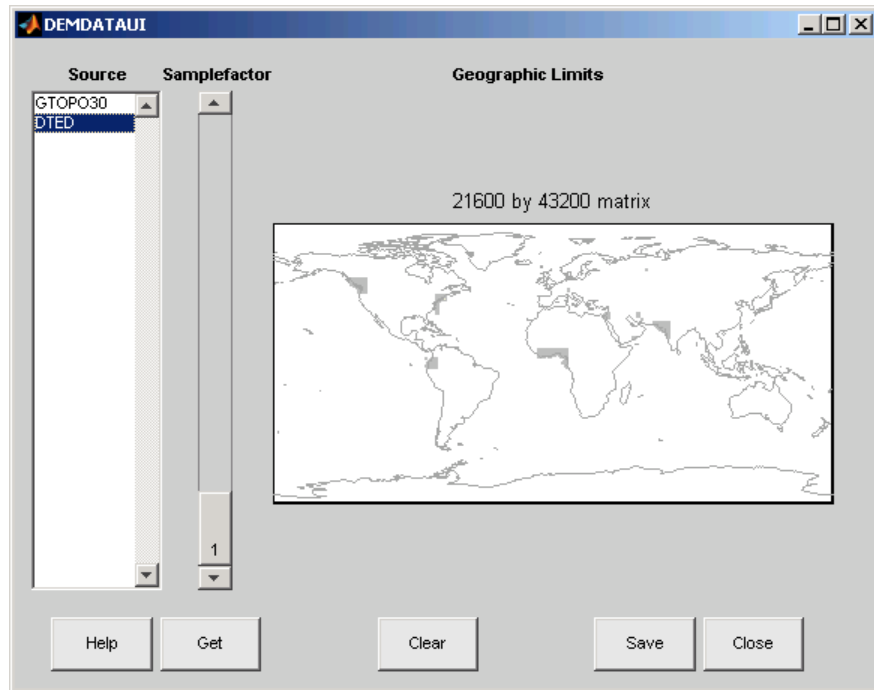
You can choose to read from any of the data sets demdataui has located. If demdataui does not recognize data you think it should find, check your path and click **Help** to read about how files are identified.

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:

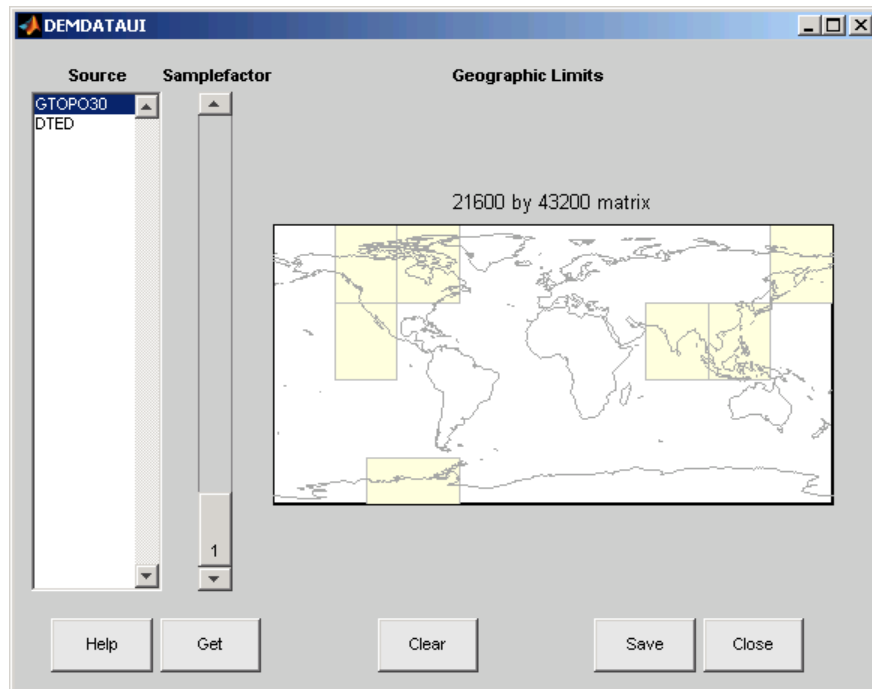
```
demdataui
```

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.



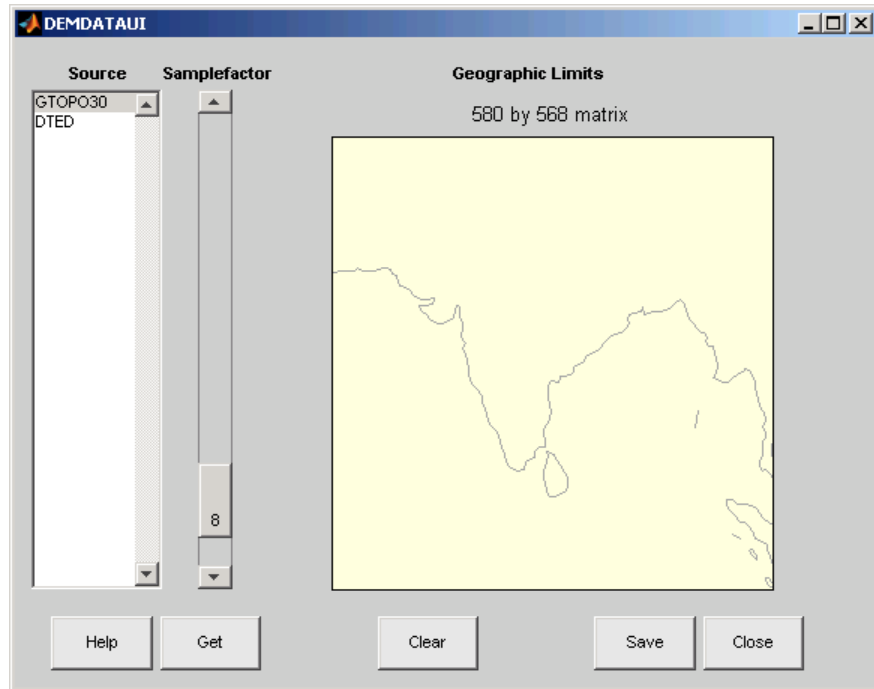
- 2 Clicking a different source in the left column updates the coverage display. Here is the coverage area for available GTOPO30 tiles.



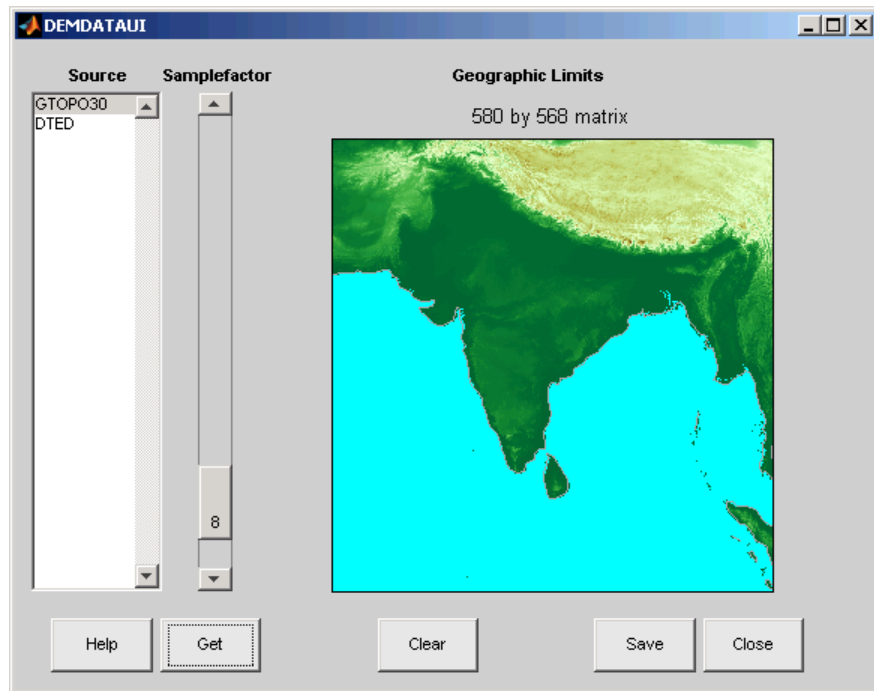


- 3** Use the map in the UI to specify the location and density of data to extract. To interactively set a region of interest, click in the map to zoom by a factor of two centered on the cursor, or click and drag across the map to define a rectangular region. The size of the matrix of the area currently displayed is printed above the map. To reduce the amount of data, you can continue to zoom in, or 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.

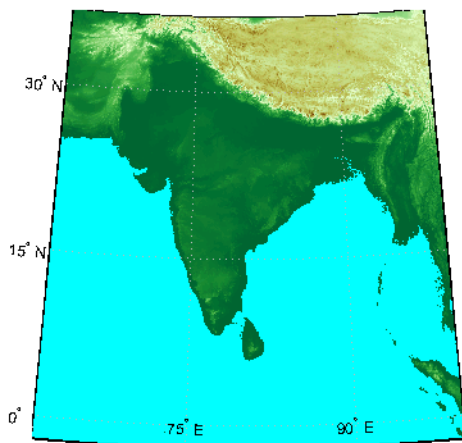


- 5 If you are not satisfied with the result, click the **Clear** button to remove all data previously read in via **Get** and make new selections. You might need to close and reopen demdatui in order to select a new region of interest.
- 6 When you are ready to import DEM data to the workspace or save 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 demdatui function returns one or more matrices as an array of geographic data 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 demdatui contains geostruct1 (Version 1 style) data structures. You cannot update these to geostruct2 geographic data structures using the updategeostruct function, because they are of type surface, which the 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 geographic data structure, use its field names. Here `map` and `maplegend` are copied from the structure and used to create a lighted three-dimensional elevation map display using `worldmap`. (`demdata` is the default name for the structure, which you can override when you save it.)

```
Z = demdata.map;  
refvec = demdata.maplegend;  
figure  
ax = worldmap(Z, refvec);  
geoshow(ax, Z, refvec, 'DisplayType', 'texturemap');  
axis off  
demcmap(Z);
```



## Determining and Visualizing Visibility Across Terrain

### Computing Line of Sight with `los2`

You can use regular data grids of elevation data to answer questions about the mutual visibility of locations on a surface (intervisibility). For example,

- Is the line of sight from one point to another obscured by terrain?
- What area can be seen from a location?
- What area can see a given location?

The first question can be answered with the `los2` function. In its simplest form, `los2` determines the visibility between two points on the surface of a digital elevation map. You can also specify the altitudes of the observer and target points, as well as the datum with respect to which the altitudes are measured. For specialized applications, you can even control the actual and effective radius of the Earth. This allows you to assume, for example, that the Earth has a radius 1/3 larger than its actual value, which is a model frequently used in predicting radio wave propagation.

The following example shows a line-of-sight calculation between two points on a regular data grid generated by the `peaks` function. The calculation is performed by the `los2` function, which returns a logical result: 1 (points are intervisible), or 0 (points are not intervisible).

- 1 Create an elevation grid using `peaks` with a maximum elevation of 500, and set its origin at (0°N, 0°W), with a spacing of 1000 cells per degree):

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

- 2 Define two locations on this grid to test intervisibility:

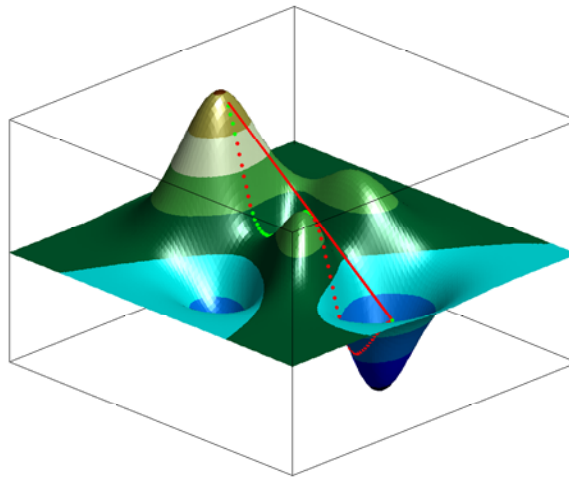
```
lat1 = -0.027; lon1 = 0.05; lat2 = -0.093; lon2 = 0.042;
```

- 3 Calculate intervisibility. The final argument specifies the altitude (in meters) above the surface of the first location (`lat1`, `lon1`) where the observer is located (the viewpoint):

```
los2(map,maplegend,lat1,lon1,lat2,lon2,100)
```

```
ans =  
1
```

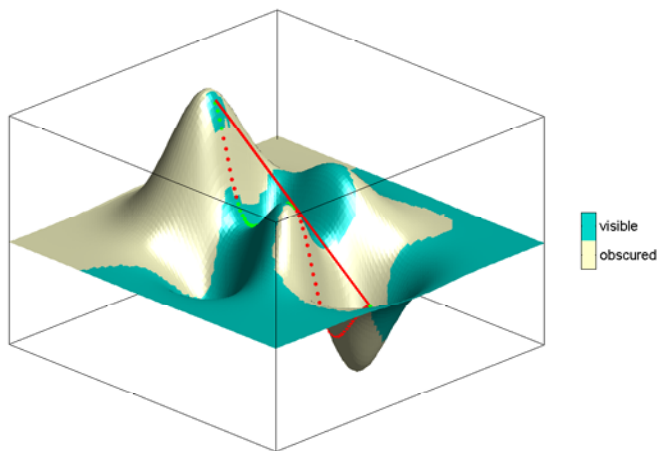
The `los2` function also produces a profile diagram in a figure window showing visibility at each grid cell along the line of sight that can be used to interpret the Boolean result. In this example, the diagram shows that the line between the two locations just barely clears an intervening peak.



You can also compute the *viewshed*, a name derived from watershed, which is all of the areas that are visible from a particular location. The `viewshed` function can be thought of as performing the `los2` line-of-sight calculation from one point on a digital elevation map to every other entry in the matrix. The `viewshed` function supports the same options as `los2`.

The following shows which parts of the peaks elevation map in the previous example are visible from the first point:

```
[vismap,vismapleg] = viewshed(map,maplegend,lat1,lon1,100);
```



## Shading and Lighting Terrain Maps

### In this section...

“Lighting a Terrain Map Constructed from a DTED File” on page 5-22  
 “Lighting a Global Terrain Map with `lightm` and `lightmui`” on page 5-25  
 “Surface Relief Shading” on page 5-29  
 “Colored Surface Shaded Relief” on page 5-33  
 “Relief Mapping with Light Objects” on page 5-36

### Lighting a Terrain Map Constructed from a DTED File

The `lightm` function creates light objects in the current map. To modify the positions and colors of lights created on world maps or large regions you can use the interactive `lightmui` GUI. For finer control over light position (for example, in small areas lit by several lights), you have to specify light positions using projected coordinates. This is because lights are children of axes and share their coordinate space. See “Lighting a Global Terrain Map with `lightm` and `lightmui`” on page 5-25 for an example of using `lightmui`.

In this exercise, you manually specify the position of a single light in the northwest corner of a DTED DEM for Cape Cod.

- 1 To illustrate lighting terrain maps, begin by following the exercise in “Mapping a Single DTED File with the DTED Function” on page 5-6, or execute the steps below:

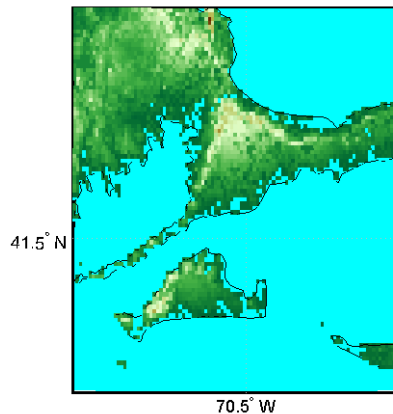
```
latlim = [ 41.20  41.95];
lonlim = [-70.95 -70.10];
cd dted\w071 %Note: Your absolute path may vary
samplefactor = 1;
[capeterrain, caperef] = dted('n41.dto', samplefactor,...
    latlim, lonlim);
capeterrain(capeterrain == 0) = -1;
capecoast = shaperead('usastatehi',...
    'UseGeoCoords', true,...
    'BoundingBox', [lonlim' latlim]);
```



- 2** Construct a map of the region within the specified latitude and longitude limits:

```
figure
ax = usamap(latlim,lonlim);
geoshow(ax, capecoast, 'FaceColor', 'none');
geoshow(ax, capeterrain, caperef, 'DisplayType', 'mesh');
demcmap(capeterrain)
```

The map looks like this.



- 3** Set the vertical exaggeration. Use `daspectm` to specify that elevations are in meters and should be multiplied by 20:

```
daspectm('m',20)
```

- 4** Make sure that the line data is visible. To ensure that it is not obscured by terrain, use `zdatam` to set it to the highest elevation of the `cape1` terrain data:

```
zdatam('allline',max(capeterrain(:)))
```

- 5** Specify a location for a light source with `lightm`:

```
h = lightm(42, -71);
```

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

- 6** To see the properties of light objects, inspect the handle returned by `lightm`:

```
get(h)
  Position = [-0.00616097 0.796039 1]
  Color = [1 1 1]
  Style = infinite

  BeingDeleted = off
  ButtonDownFcn =
  Children = []
  Clipping = on
  CreateFcn =
  DeleteFcn =
  BusyAction = queue
  HandleVisibility = on
  HitTest = on
  Interruptible = on
  Parent = [138.001]
  Selected = off
  SelectionHighlight = on
  Tag =
  Type = light
  UIContextMenu = []
  UserData = [ (1 by 1) struct array]
  Visible = on
```

Had you used the MATLAB `light` function in place of `lightm`, you would have needed to specify the position in Cartesian 3-space.

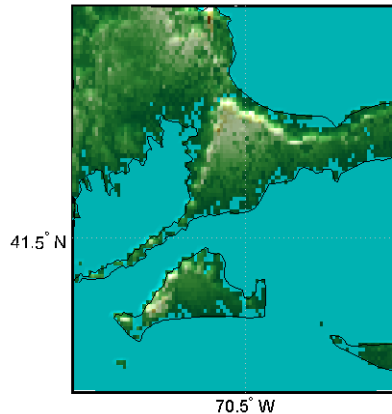
- 7** The lighting computations caused the map to become quite dark with specular highlights. Now restore its luminance by specifying three surface reflectivity properties in the range of 0 to 1:

```
ambient = 0.7; diffuse = 1; specular = 0.6;
material([ambient diffuse specular])
```

The surface looks blotchy because there is no interpolation of the lighting component (flat facets are being modeled). Correct this by specifying Phong shading:

```
lighting phong
```

The map now looks like this.



**8** If you want to compare the lit map with the unlit version, you can toggle the lighting off:

```
lighting none
```

For additional information, see the reference pages for `daspectm`, `lightm`, `light`, `lighting`, and `material`.

## Lighting a Global Terrain Map with `lightm` and `lightmui`

In this example, you create a global topographic map and add a local light at a distance of 250 km above New York City, (40.75 °N, 73.9 °W). You then change the material and lighting properties, add a second light source, and activate the `lightmui` tool to change light position, altitude, and colors.

The `lightmui` display plots lights as circular markers whose facecolor indicates the light color. To change the position of a light, click and drag the

circular marker. Alternatively, right-clicking the circular marker summons a dialog box for changing the position or color of the light object. Clicking the color bar in that dialog box invokes the `uisetcolor` dialog box that can be used to specify or pick a color for the light.

- 1** Load the topo DTM files, and set up an orthographic projection:

```
load topo
axesm ('mapprojection','ortho', 'origin',[10 -20 0])
```

- 2** Plot the topography and assign a topographic colormap:

```
meshm(topo,topolegend);
demcmap(topo)
```

- 3** Set up a yellow light source over New York City:

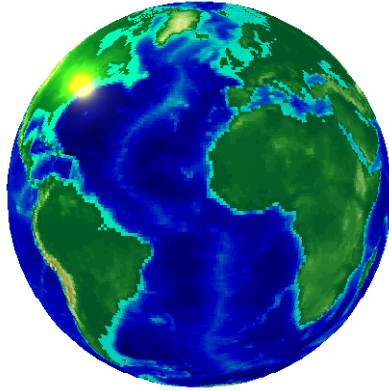
```
hl = lightm(40.75,-73.9, 500/almanac('earth','radius'),...
'color','yellow', 'style', 'local');
```

The first two arguments to `lightm` are the latitude and longitude of the light source. The third argument is its altitude, in units of Earth radii (in this case they are in kilometers, the default units of `almanac`).

- 4** The surface is quite dark, so give it more reflectivity by specifying

```
material([0.7270 1.5353 1.9860 4.0000 0.9925]);
lighting phong; hidem(gca)
```

The lighted orthographic map looks like this.



- 5 If you want, add more lights, as follows:

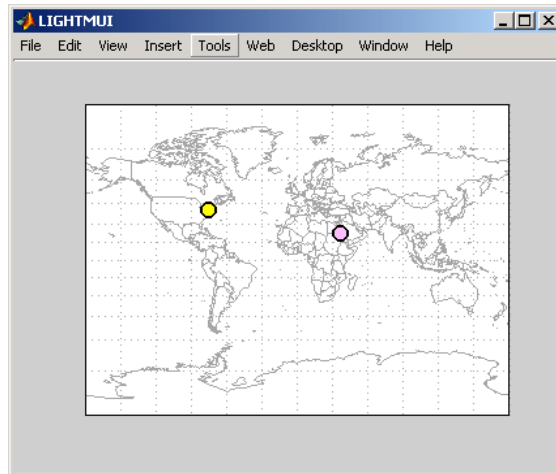
```
h2 = lightm(20,40, .1,'color','magenta', 'style', 'local')
```

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

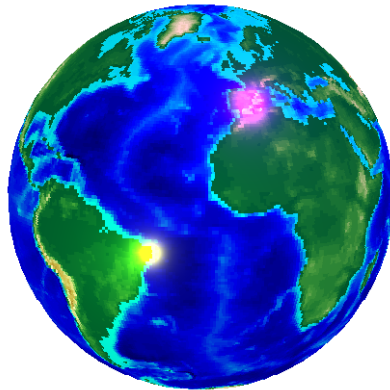
- 6 To modify the lights, use the `lightmui` GUI, which lets you drag lights across a world map and specify their color and altitudes:

```
lightmui(gca)
```

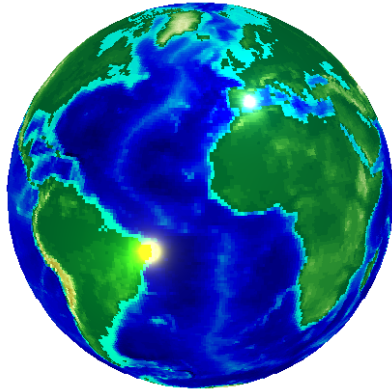
The lights are shown as appropriately colored circles, which you can drag to new positions. You can also **Ctrl**+click a circle to bring up a dialog box for directly specifying that light's position, altitude, and color. The GUI and the map look like this at this point.



- 7 In the `lightmui` window, drag the yellow light to the eastern tip of Brazil, and drag the magenta light to the Straits of Gibraltar.



- 8 **Ctrl**+click or **Shift**+click the magenta circle in the `lightmui` window. A second UI, for setting light position and color, opens. Set the **altitude** to 0.04 (Earth radii). Set the light **color** components to 1.0 (red), 0.75 (green), and 1.0 (blue). Press **Return** after each action. The colorbar on the UI changes to indicate the color you set. If you prefer to pick a color, click on the colorbar to bring up a color-choosing UI. The map now looks like this.



For additional information, see the reference pages for `lightm` and `lightmui`.

## Surface Relief Shading

You can make dimensional monochrome shaded-relief maps with the function `surf1m`, which is analogous to the MATLAB `surf1` function. The effect of `surf1m` 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 `surf1m` function works with any projection except globe.

The `surf1m` 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 4-54). The following example illustrates this procedure.

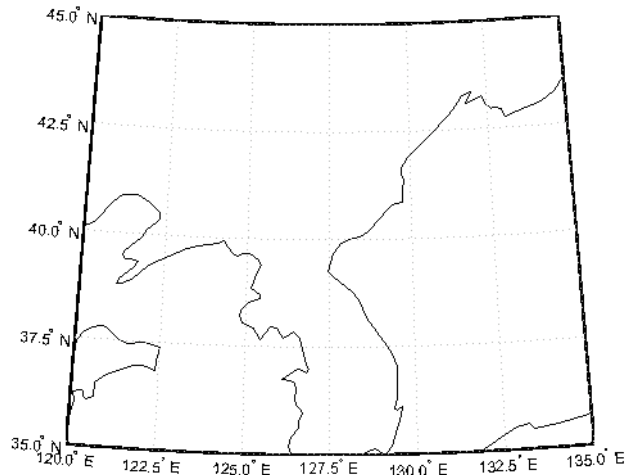
### Creating Monochrome Shaded Relief Maps Using `surf1m`

As stated above, `surf1m` 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 `surf1m` 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:

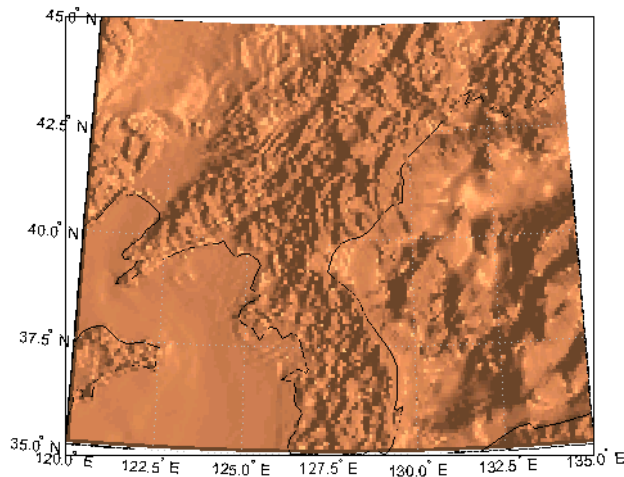


```
[klat,klon] = meshgrat(map,refvec);
```

- 4** Use `surf1m` to generate a default shaded relief map, and change the colormap to a monochromatic scale, such as gray, bone, or copper.

```
ht = surf1m(klat,klon,map);
colormap('copper')
```

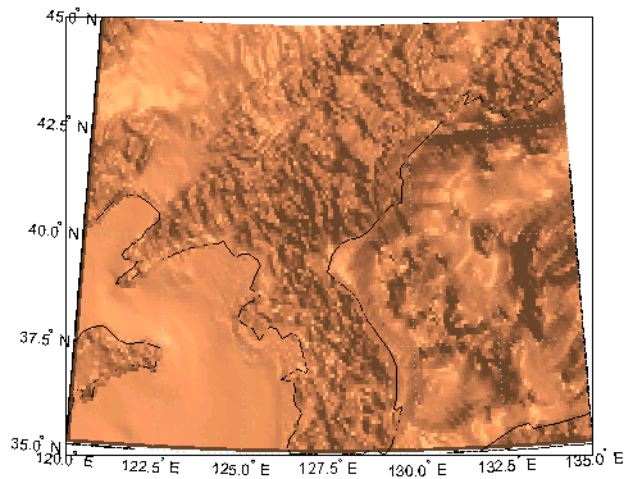
In this default case, the lighting direction is set at  $45^\circ$  counterclockwise from the viewing direction; thus the “sun” is in the southeast. This map is shown below.



- 5** To make the light come from some other direction, specify the light source’s azimuth and elevation as the fourth argument to `surf1m`. Clear the terrain map and redraw it, specifying an azimuth of  $135^\circ$  (northeast) and an elevation of  $60^\circ$  above the horizon:

```
clmo(ht); ht=surf1m(klat,klon,map,[135,60]);
```

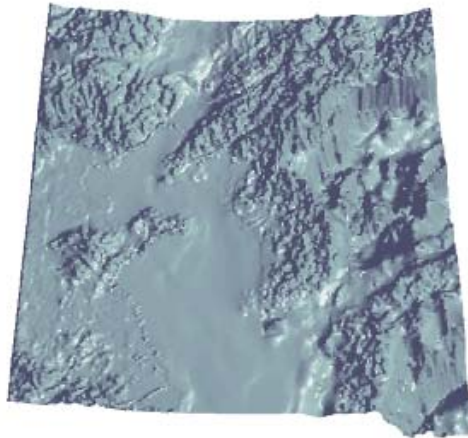
The surface lightens and has a new character because it is lit closer to overhead and from a different direction.



- 6 Now shift the light to the northwest ( $-135^\circ$  azimuth), and lower it to  $40^\circ$  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 `surf1m`. 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=surf1m(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 `surf1m` and `surf1`.

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 `mesh1srm` and `surf1srm` display maps as shaded relief with surface coloring as well as light source shading. You can think of them as extensions to `surf1m` that combine surface coloring and surface light shading. Use `mesh1srm` to display regular data grids and `surf1srm` to render geolocated data grids.

These two functions construct a new colormap and associated `CData` matrix that uses grayscales to lighten or darken a matrix component based on its calculated surface normal to a light source. While there are no analogous MATLAB display functions that work like this, you can obtain similar results using MATLAB light objects, as “Relief Mapping with Light Objects” on page 5-36 explains.

### Coloring Shaded Relief Maps and Viewing Them in 3-D

In this exercise, you use `surf1srm` in a way similar to how you used `surf1m` in the preceding exercise, “Creating Monochrome Shaded Relief Maps Using `surf1m`” on page 5-29. In addition, you set a vertical scale and view the map from various perspectives.

- 1 Start with a new map axes and the korea data, and then georeference the regular data grid:

```
load korea
[klat,klon] = meshgrat(map,refvec);
axesm miller
```

- 2 Create a colormap for DEM data; it is transformed by `surf1srm` 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  $-135^\circ$  and an altitude of  $50^\circ$  for the light source:

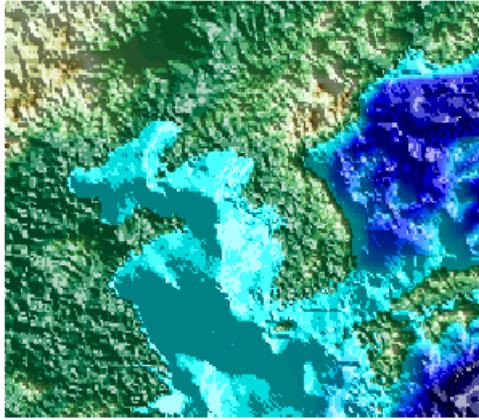
```
surf1srm(klat,klon,map,[-130 50],cmap,clim)
```

You can also achieve the same effect with `mesh1srm`, which operates on regular data grids (it first calls `meshgrat`, just as you just did), e.g., `mesh1srm(map,maplegend)`.

- 4 The surface has more contrast than if it were not shaded, and it might help to lighten it uniformly by 25% or so:

```
brighten(.25)
```

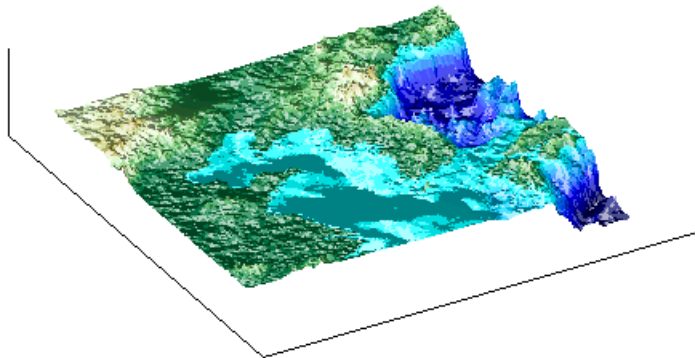
The map, which has an overhead view, looks like this.



- 5 Plot an oblique view of the surface by hiding its bounding box, exaggerating terrain relief by a factor of 50, and setting the view azimuth to  $-30^\circ$  (south-southwest) and view altitude to  $30^\circ$  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 `surf1srm`.

For further information, see the reference pages for `surf1srm`, `mesh1srm`, `daspectm`, and `view`.

## Relief Mapping with Light Objects

In the exercise “Lighting a Global Terrain Map with `lightm` and `lightmui`” on page 5-25, you created light objects to illuminate a Globe display. In the following one, you create a light object to mimic the map produced in the previous exercise (“Coloring Shaded Relief Maps and Viewing Them in 3-D” on page 5-34), which uses shaded relief computations rather than light objects.

The `mesh1srm` and `surf1srm` 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. Mapping 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:

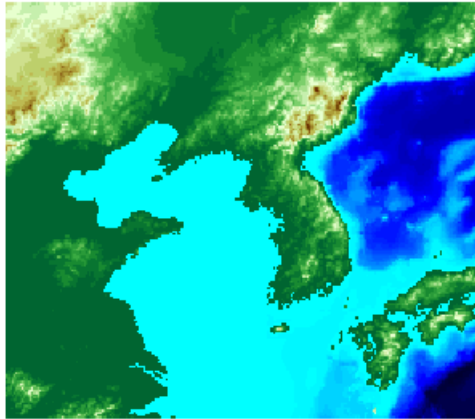
- 1 If you need to, load the korea DEM, and create a map axes using the Miller projection:

```
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.

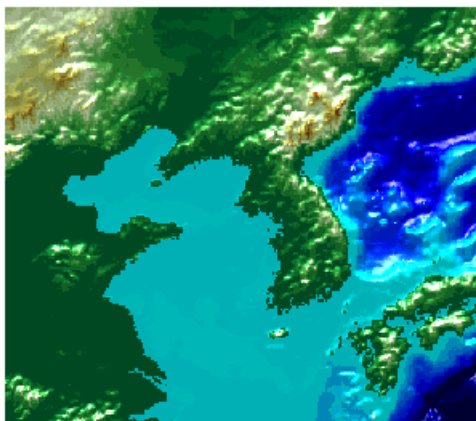
- 5** Set the ambient (direct), diffuse (skylight), and specular (highlight) surface reflectivity characteristics, respectively:

```
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
```

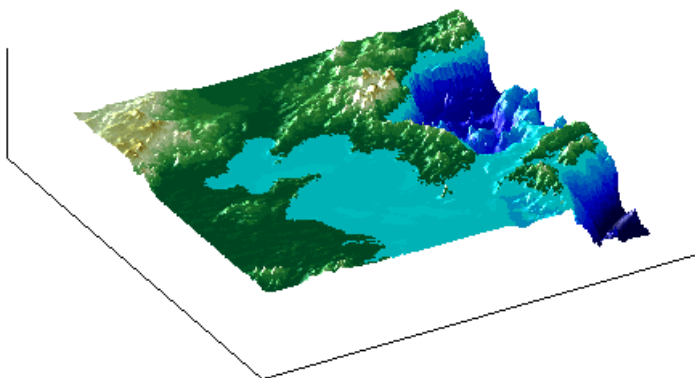
The map now looks like this.



- 7 Finally, remove the edges of the bounding box and set a viewpoint of  $-30^\circ$  azimuth,  $30^\circ$  altitude:

```
set(gca, 'Box', 'off')  
view(-30,30)
```

The view from  $(-30,30)$  with one light at  $(45,115,1)$  and Phong shading is shown below. Compare it to the final map in the previous exercise, “Coloring Shaded Relief Maps and Viewing Them in 3-D” on page 5-34.



To remove a light (when there is only one) from the current figure, type



```
clmo(handlem('light'))
```

For more information, consult the reference pages for `lightm`, `daspectm`, `material`, `lighting`, and `view`, along with the section on “Lighting as a Visualization Tool” in the 3-D Visualization documentation.

## Draping Data on Elevation Maps

In this section...
“Draping Geoid Heights over Topography” on page 5-40
“Draping Data over Terrain with Different Gridding” on page 5-43

### 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 can generate such displays using 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:

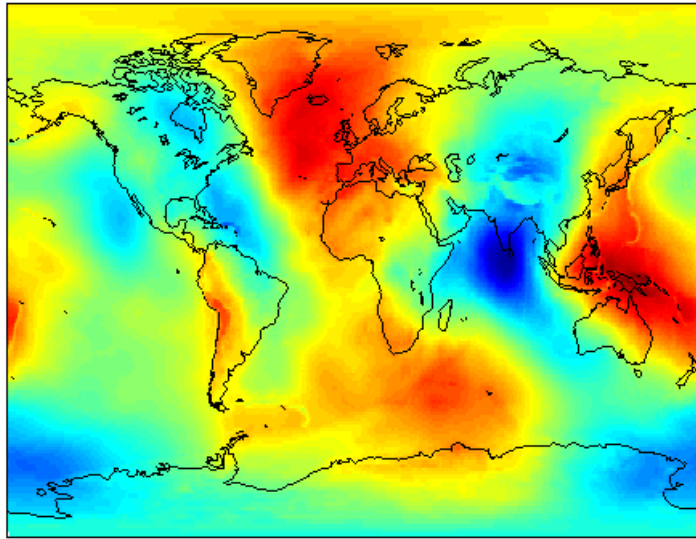
```
meshm(geoid,geoidrefvec,size(geoid),topo)
```

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:

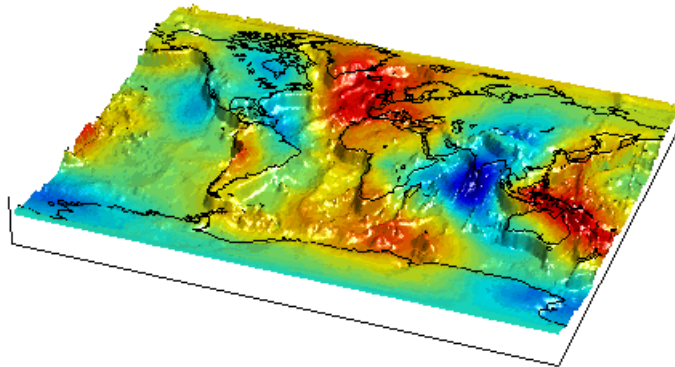
```
set(gca,'Box','off')  
camlight;  
lighting phong
```

- 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 `1t1n2val` 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.

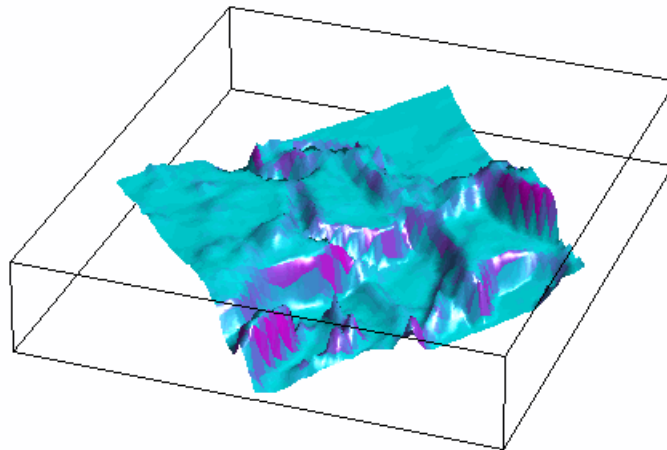
- 7 The default rendering uses faceted shading (no smooth interpolation). Render the surface again, this time making it shiny with Phong shading and lighting from the east (the default of camlight lights surfaces from over the viewer's right shoulder):

```
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 NaNs 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);
```

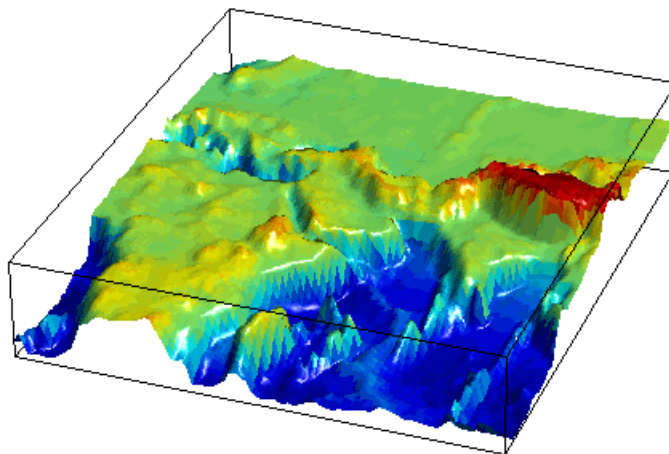
- 7** Render the figure as a 3-D view from a 20° azimuth and 30° altitude, and exaggerate the vertical dimension by a factor of 200:

```
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')
```

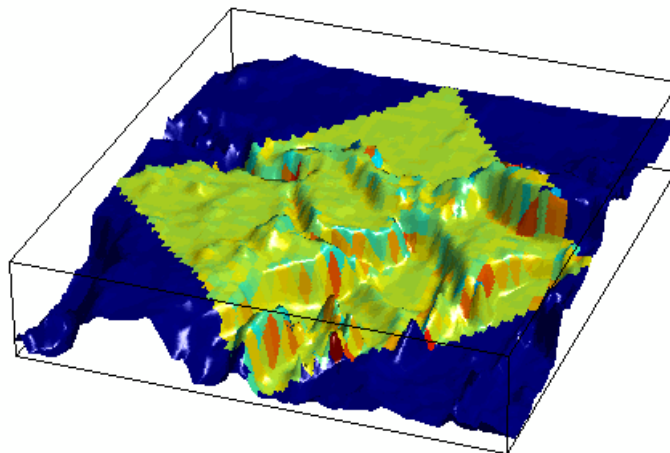
So far, both the surface relief and coloring represent topographic elevation.



- 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?” on page 5-49

“The Globe Display Compared with the Orthographic Projection” on page 5-50

“Using Opacity and Transparency in Globe Displays” on page 5-52

“Over-the-Horizon 3-D Views Using Camera Positioning Functions” on page 5-55

“Displaying a Rotating Globe” on page 5-57

### 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 projections in Mapping Toolbox transform latitudes and longitudes to map  $x$ - and  $y$ -coordinates. The `globe` function is special because it can render relative relief of elevations above, below, or on a sphere. In Earth-centered Cartesian  $(x,y,z)$  coordinates,  $z$  is not an optional elevation; rather, it is an axis in Cartesian three-space. `globe` is useful for geospatial applications that require three-dimensional relationships between objects to be maintained, such as when one simulates flybys, and/or views planets as they rotate.

The Globe display is based on a *coordinate transformation*, and is not a map projection. While it has none of the distortions inherent in planar projections, it is a three-dimensional model of a planet that cannot be displayed without distortion or in its entirety. That is, in order to render the globe in a figure window, either a perspective or orthographic transformation must be applied, both of which necessarily involve setting a viewpoint, hiding the back side and distortions of shape, scale, and angles.

## The Globe Display Compared with the Orthographic Projection

The following example illustrates the differences between the two-dimensional orthographic projection, which looks spherical but is really flat, and the three-dimensional Globe display. Use the **Rotate 3D** tool to manipulate the display.

- 1 Load the topo data set and render it with an orthographic map projection:

```
load topo
axesm ortho; framem
meshm(topo,topolegend);demcmap(topo)
```

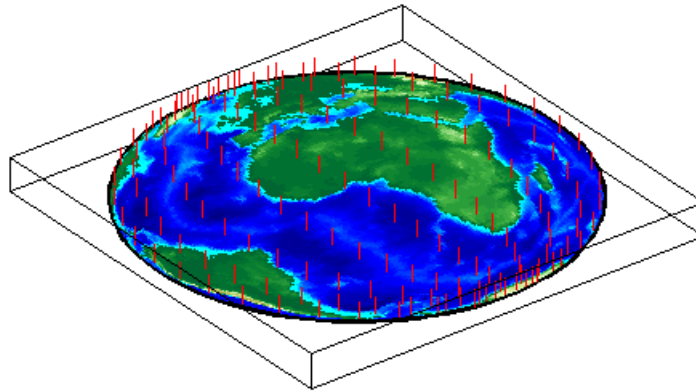
- 2 View the map obliquely:

```
view(3); daspectm('m',1)
```

- 3 You can view it in 3-D from any perspective, even from underneath. To visualize this, define a geolocated data grid with `meshgrat`, populate it with a constant  $z$ -value, and render it as a stem plot with `stem3m`:

```
[latgrat,longrat] = meshgrat(topo,topolegend,[20 20]);
stem3m(latgrat,longrat,500000*ones(size(latgrat)),'r')
```

Use the **Rotate 3D** tool on the figure window toolbar to change your viewpoint. No matter how you position the view, you are looking at a disc with stems protruding perpendicularly.



- 4** Create another figure using the Globe transform rather than orthographic projection:

```
figure
axesm('globe','Geoid',almanac('earth','radius','m'))
```

- 5** Display the topo surface in this figure and view it in 3-D:

```
meshm(topo,topolegend); demcmap(topo)
view(3)
```

- 6** Include the stem plot to visualize the difference in surface normals on a sphere:

```
stem3m(latgrat,longrat,500000*ones(size(latgrat)),'r')
```

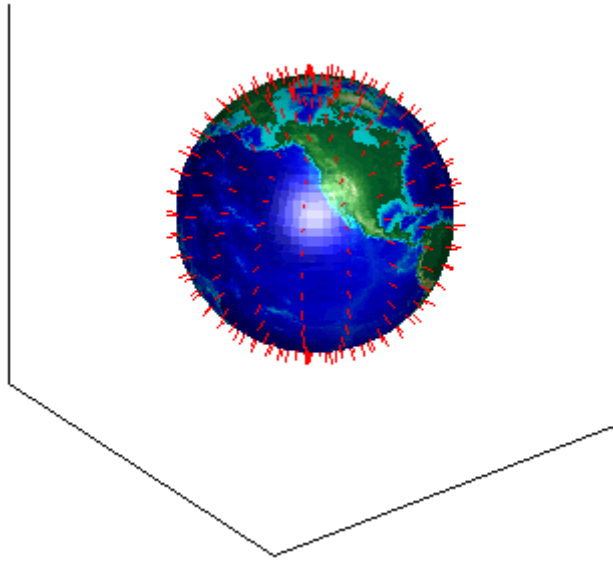
- 7** You can apply lighting to the display, but its location is fixed, and does not move as the camera position is shifted:

```
camlight('headlight','infinite')
```

- 8** If you prefer a more unobstructed view, hide the 3-D axes:

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

Here is a representative view using the Globe display with the headlight.



You can use the `LabelRotation` property when you use the Orthographic or any other Mapping Toolbox projection to align meridian and parallel labels with the graticule. Because the Globe display is not a true map projection and is handled differently internally, `LabelRotation` does not work with it.

For additional information on functions used in this example, see the reference pages for `view`, `camlight`, `meshgrat`, and `stem3m`.

### Using Opacity and Transparency in Globe Displays

Because Globe displays depict 3-D objects, you can see into and through them as long as no opaque surfaces (e.g., patches or surfaces) obscure your view. This can be particularly disorienting for point and line data, because features on the back side of the world are reversed and can overlay features on the front side.

Here is one way to create an opaque surface over which you can display line and point data:

- 1 Create a figure and set up a Globe display:

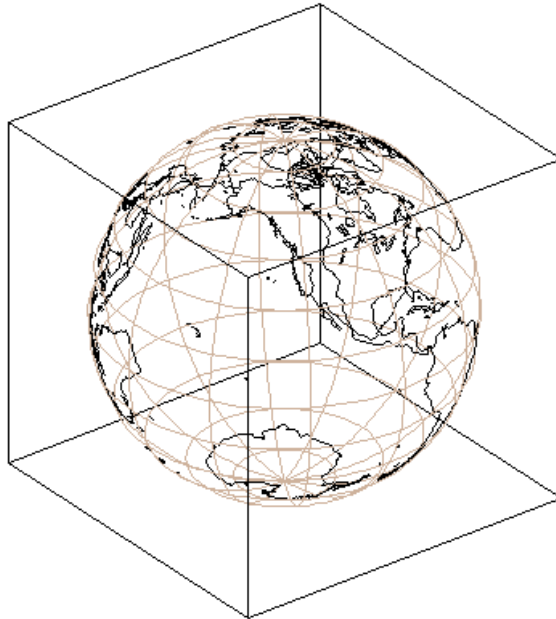
```
figure; axesm('globe')
```

- 2** Draw a graticule in a light color, slightly raised from the surface:

```
gridm('LineStyle','-','Gcolor',[.8 .7 .6],'Galtitude',.02)
```

- 3** Load and plot the coast data in black, and set up a 3-D perspective:

```
load coast
plot3m(lat,long,.01,'k')
view(3)
```



- 4** Use the **Rotate 3D** tool on the figure's toolbar to rotate the view. Note how confusing the display is because of its transparency.

- 5** Make a uniform 1-by-1-degree grid and a referencing vector for it:

```
base = zeros(180,360); baseref = [1 90 0];
```

- 6** Render the grid onto the globe, color it copper, light it from camera right, and make the surface reflect more light:

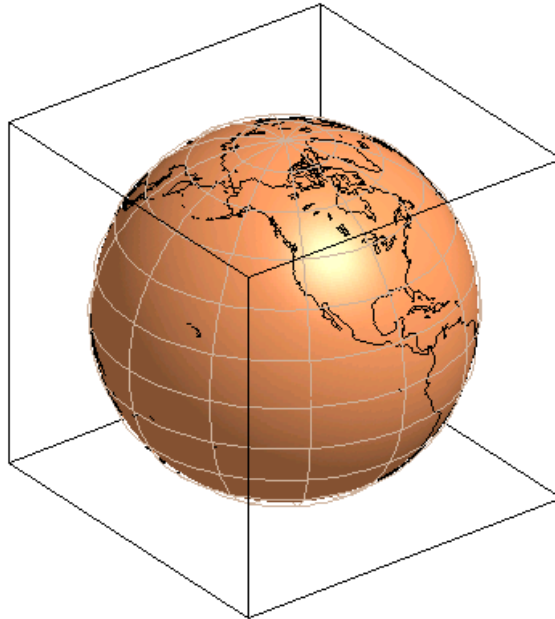
```
hs = meshm(base,baseref,size(base));  
colormap copper  
camlight right  
material([.8 .9 .4])
```

---

**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 versions of `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)); demcmmap(topo);
```

The default view is from above the North Pole with the central meridian running parallel to the  $x$ -axis.

- 3 Add world coastlines from the global `landareas` shapefile and plot them in light gray:

```
coastlines = shaperead('landareas',...
    'UseGeoCoords', true, 'Attributes', {});
plotm([coastlines.Lat], [coastlines.Lon], 'Color', [.7 .7 .7])
```

- 4** Read the coordinate locations for Moscow and Washington from the worldcities shapefile:

```
moscow = shaperead('worldcities',...
    'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'Moscow'),'Name'});
washington = shaperead('worldcities',...
    'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'Washington D.C.'),...
    'Name'});
```

- 5** Create a great circle track to connect Washington with Moscow and plot it in red:

```
[latc,lonc] = track2('gc',...
    moscow.Lat, moscow.Lon, washington.Lat, washington.Lon);
plotm(latc,lonc,'r')
```

- 6** Point the camera at Moscow. Wherever the camera is subsequently moved, it always looks toward [moscow.Lat moscow.Lon]:

```
camtargm(moscow.Lat, moscow.Lon, 0)
```

- 7** Station the camera above Washington. The third argument is an altitude in Earth radii:

```
camposm(washington.Lat, washington.Lon, 3)
```

- 8** Establish the camera up vector with the camera target's coordinates. The great circle joining Washington and Moscow now runs vertically:

```
camupm(moscow.Lat, moscow.Lon)
```

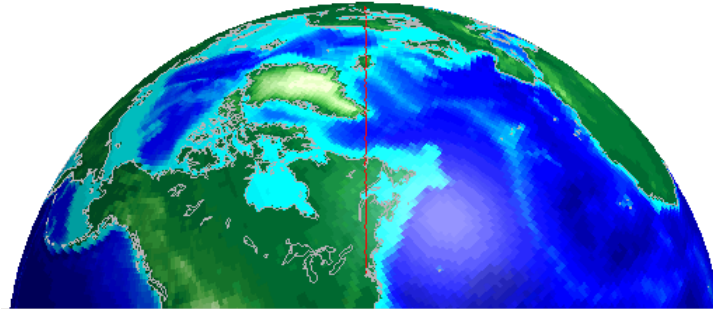
- 9** Set the field of view for the camera to 20° for the final view:

```
camva(20)
```

- 10** Add a light, specify a relatively nonreflective surface material, and hide the map background:

```
camlight; material(0.6*[ 1 1 1])
hidem(gca)
```

Here is the final view.



For additional information, see the reference pages for `extractm`, `camtargm`, `camposm`, `camupm`, `globe`, and `camlight`.

## Displaying a Rotating Globe

Because the Globe display can be viewed from any angle without the need to recompute a projection, you can easily animate it to produce a rotating globe. If the displayed data is simple enough, such animations can be redrawn at relatively fast rates. In this exercise, you progressively add or replace features on a Globe display and rotate it under the control of an M-file that resets the view to rotate the globe from west to east in one-degree increments.

1 In the MATLAB editor, create an M-file containing the following code:

```
% spin.m: Rotates a view around the equator one revolution
% in 5-degree steps. Negative step makes it rotate normally
% (west-to-east).
for i=360:-5:0
    view(i,23.5);    % Earth's axis tilts by 23.5 degrees
    drawnow
end
```

Save this as `spin.m` in your current directory or on the MATLAB 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],'LineStyle','-')
```

The view is from above the North Pole.

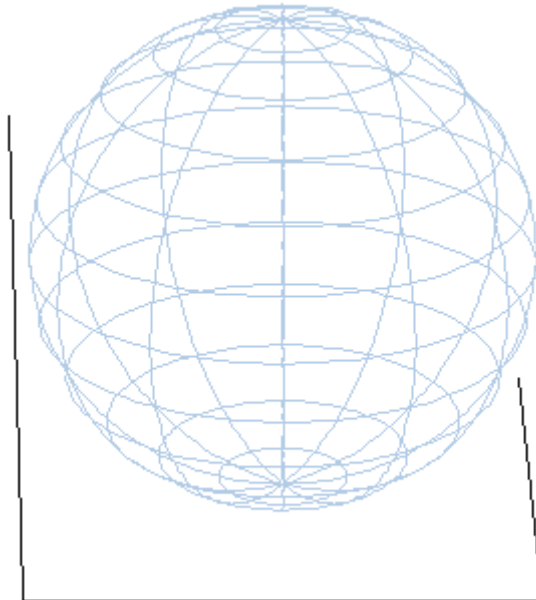
- 3** Show the axes, but hide the edges of the figure's box, and view it in perspective rather than orthographically (the default perspective):

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

- 4** Spin the globe one revolution with your M-file:

```
spin
```

The globe spins rapidly. The last position looks like this.



- 5** To make the globe opaque, create a sea-level data grid as you did for the previous exercise, “Using Opacity and Transparency in Globe Displays” on page 5-52:

```
base = zeros(180,360); baseref = [1 90 0];  
hs = meshm(base,baseref,size(base));  
colormap copper
```

The globe now is a uniform dark copper color with the grid overlaid.

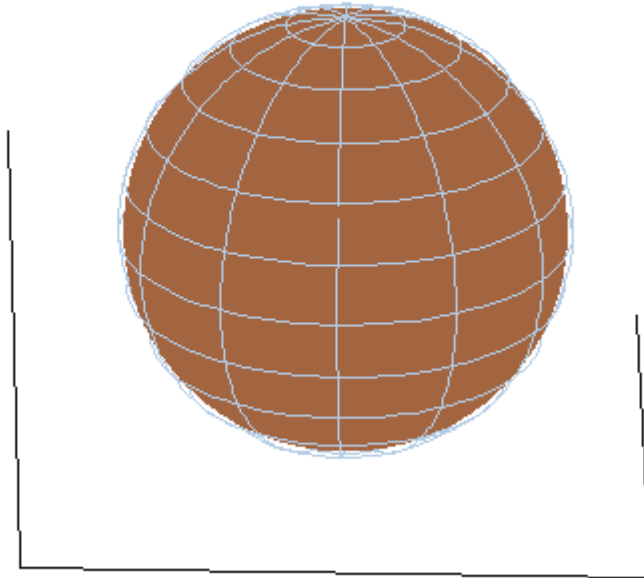
- 6 Pop up the grid so it appears to float 2.5% above the surface. Prevent the display from stretching to fit the window with the axis `vis3d` command:

```
setm(gca, 'Galtitude',0.025);  
axis vis3d
```

- 7 Spin the globe again:

```
spin
```

The motion is slower, due to the need to rerender the 180-by-360 mesh:  
The last frame looks like this.



- 8** Get ready to replace the uniform sphere with topographic relief by deleting the copper mesh:

```
clm(hs)  
load topo
```

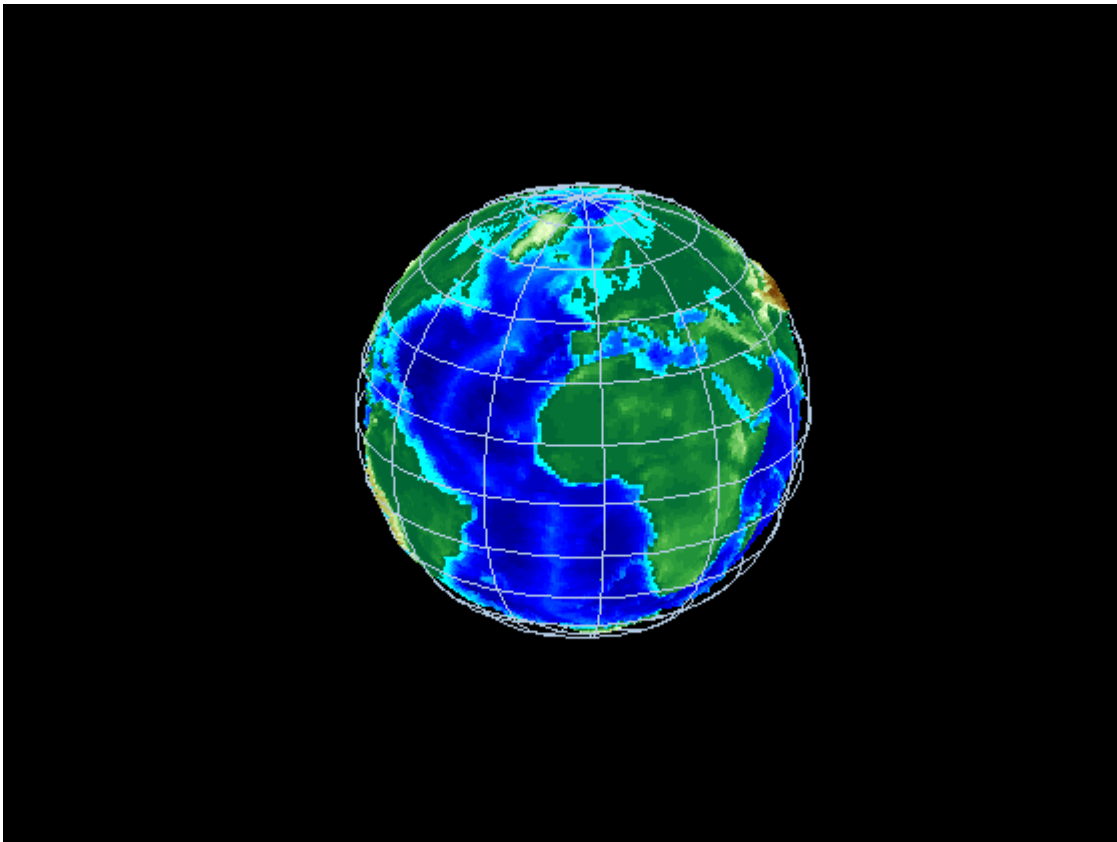
- 9** Scale the elevations to have an exaggeration of 50 (in units of Earth radii) and plot the surface:

```
topo = topo / (almanac('earth','radius') * 20);  
hs = meshm(topo,topolegend,size(topo),topo);  
demcmmap(topo)
```

- 10** Show the Earth in space; blacken the figure background, turn off the three axes, and spin again:

```
set(gcf,'color','black');  
axis off;  
spin
```

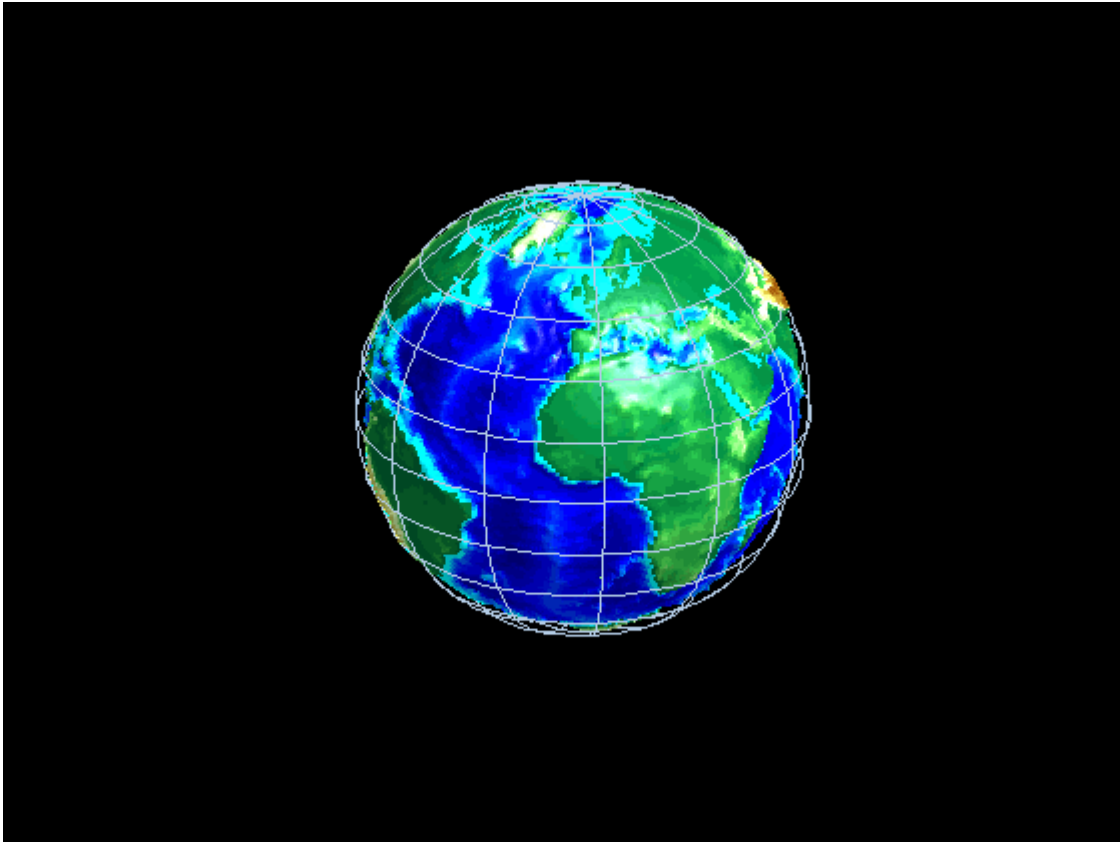
Here is a representative view, showing the Himalayas rising on the Eastern limb of the planet and the Andes on the Western limb.



- 11** You can apply lighting as well, which shifts as the planet rotates. Try the following settings, or experiment with others:

```
camlight right  
lighting phong;  
material ( [.7, .9, .8] )
```

Here is the illuminated version of the preceding view:



For additional information, see the globe, camlight, and view reference pages.



# Customizing and Printing Maps

---

Using Mapping Toolbox you can place several types of map annotations in addition to those previously described (tracks, circles, grids, meridian and parallel labels, and other text objects). The following sections describe some of this additional functionality for defining annotation elements and for making a variety of thematic maps.

Inset Maps (p. 6-2)

Placing small overview maps in a map frame

Graphic Scales (p. 6-8)

Placing scale bars in a map frame and controlling their appearance

North Arrows (p. 6-14)

Placing arrows in map frames that point to true north

Thematic Maps (p. 6-17)

Symbolizing vector and raster data and attributes in 2-D and 3-D

Using Cartesian MATLAB Display Functions (p. 6-28)

Exploiting nonmapping MATLAB functions and integrating their outputs into map axes

Using Colormaps and Colorbars (p. 6-34)

Creating colormaps and colorbar legends

Printing Maps to Scale (p. 6-45)

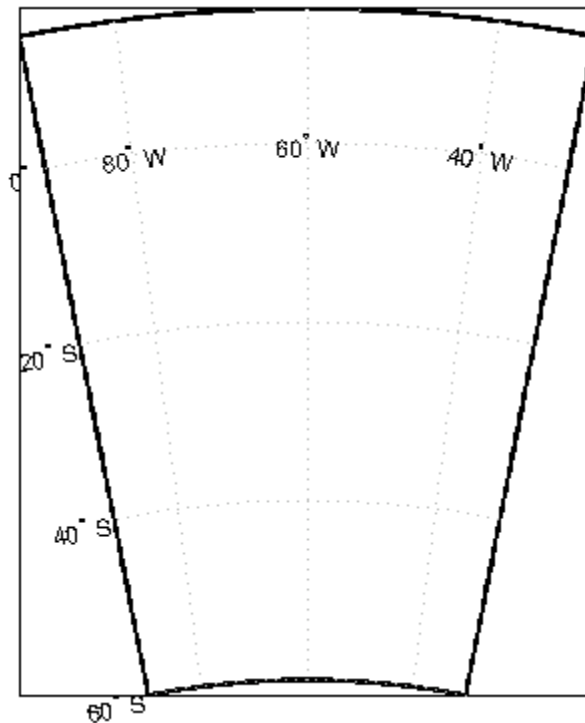
How to determine the size a map will be when a figure window is printed

## 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');
```

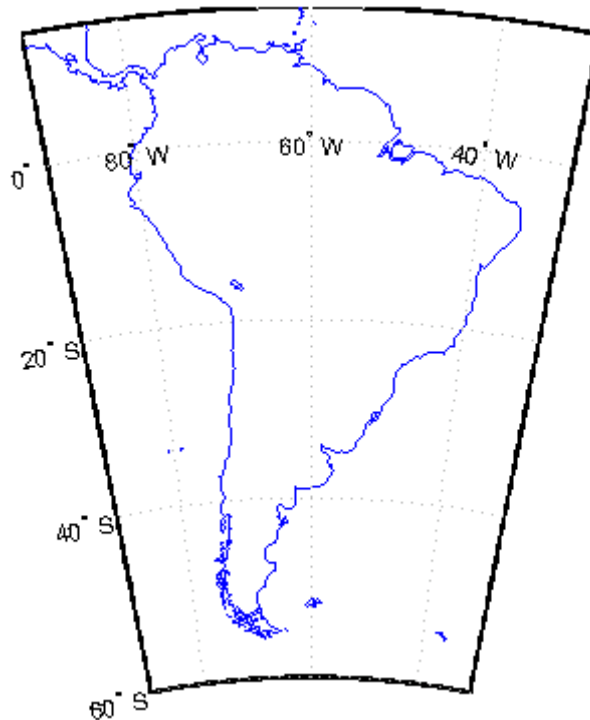


- 2 Use `shaperead` to read the demo world land areas polygon shapefile:

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

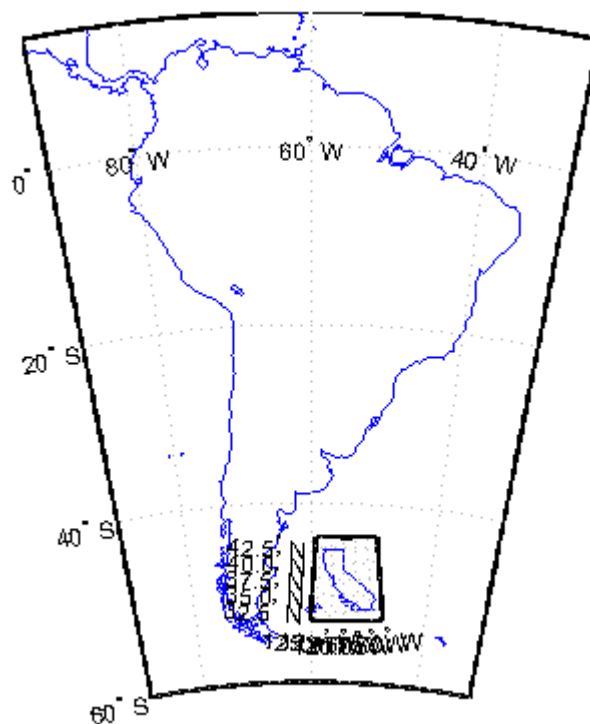
**3** Display the data in the map axes:

```
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])
```

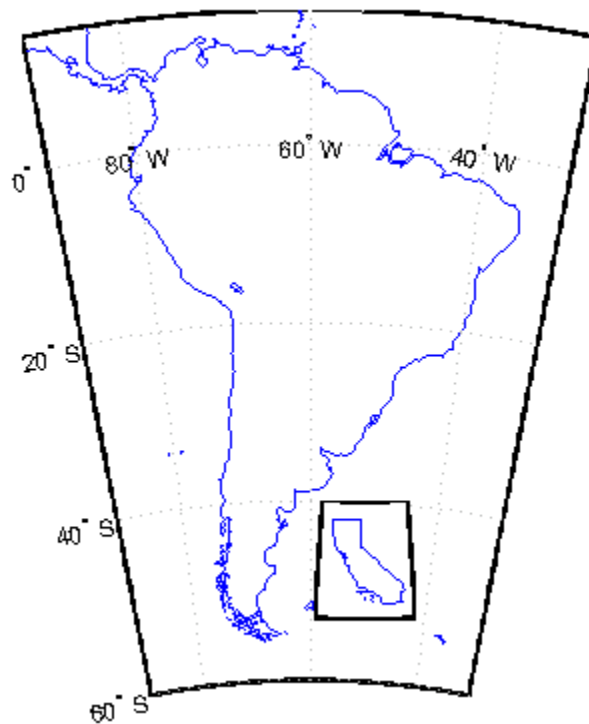


- 5** Set the frame fill color and set the labels:

```
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')
```



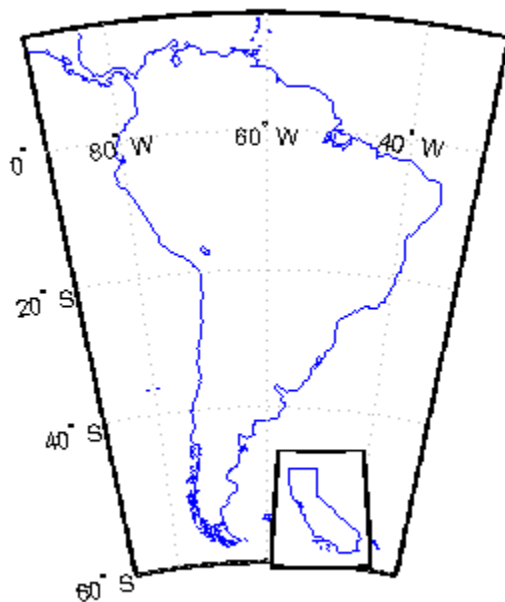
Note that Mapping Toolbox chose a different projection and appropriate parameters for each region based on its location and shape. You can override these choices to make the two projections the same.

- 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
```

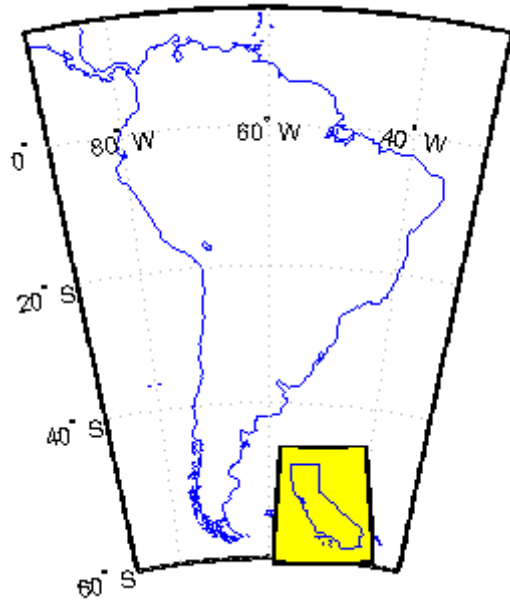
```
setm(h1, 'mapprojection', getm(h2, 'mapprojection'))
```



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')
```



## 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.

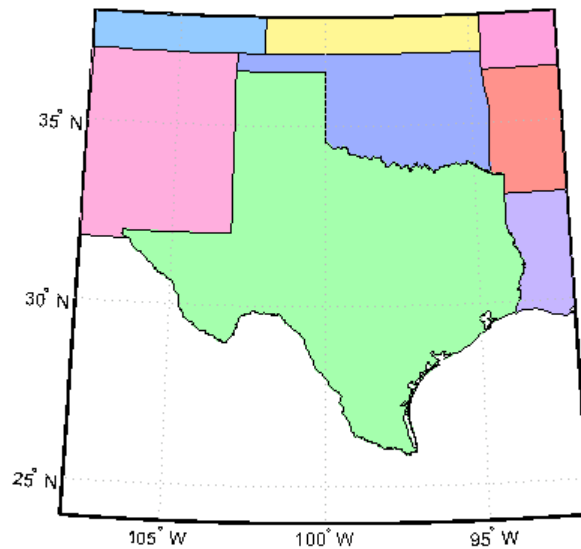
Try this by creating a map, adding a graphic scale with the default settings, and shifting its location. Then add a second scale in nautical miles, and change the tick mark style and direction:

- 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.

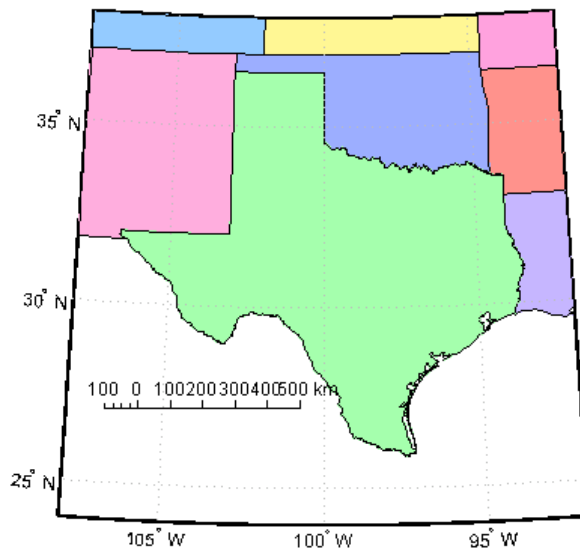




**2** Add a default graphic scale and then move it to a new location:

```
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 hgroup 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: 0.500000000000000
    MajorTick: [0 100 200 300 400 500]
    MajorTickLabel: {6x1 cell}
    MajorTickLength: 20
```

```
        MinorTick: [0 25 50 75 100]
MinorTickLabel: '100'
MinorTickLength: 12.500000000000000
        Radius: 'earth'
RulerStyle: 'ruler'
        TickDir: 'up'
TickMode: 'auto'
        Units: 'km'
        XLoc: 0.15000000000000000
        YLoc: 0.50000000000000000
        ZLoc: []
```

- 4** Change the scaleruler's font size to 8 points:

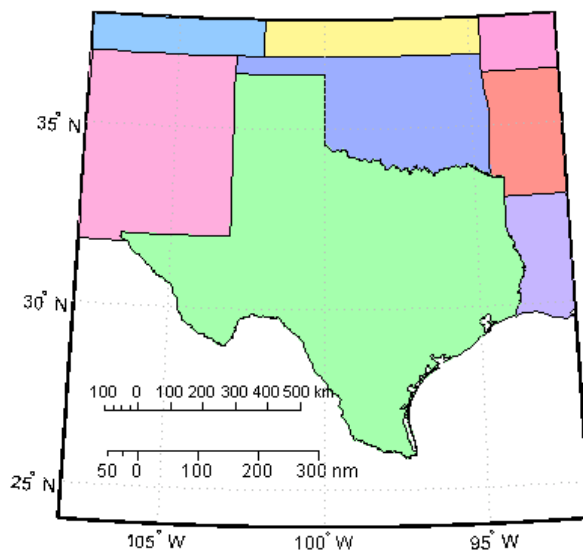
```
setm(s, 'fontsize', 8)
```

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

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

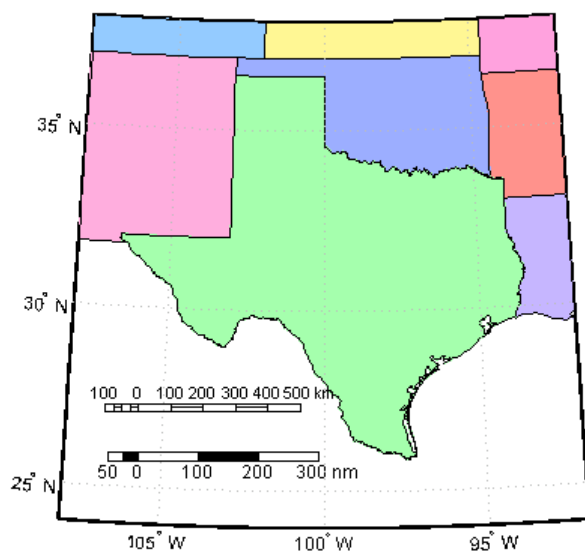
- 6** Modify its tick properties:

```
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')
```

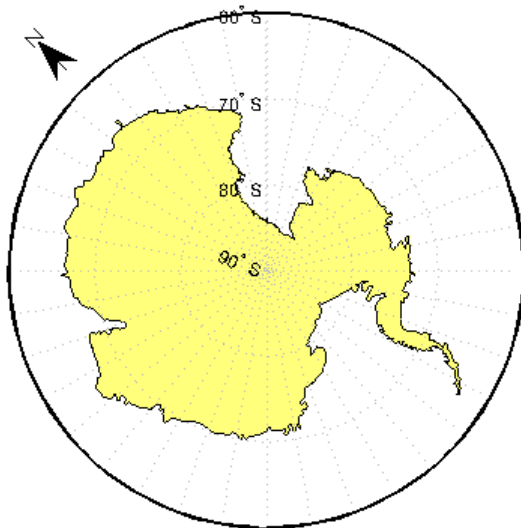


## North Arrows

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:

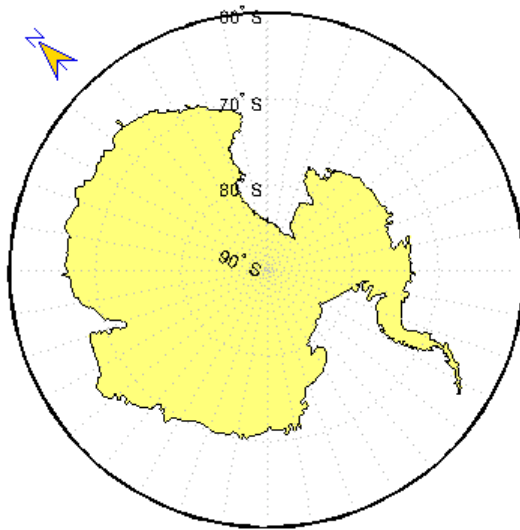
- 1 To illustrate the use of north arrows, create a map centered at the South Pole and add a north arrow symbol at a specified geographic position:

```
Antarctica = shaperead('landareas', 'UseGeoCoords', true, ...  
    'Selector',{@(name) strcmpi(name,{'Antarctica'})}, 'Name');  
figure;  
worldmap('south pole')  
geoshow(Antarctica)  
northarrow('latitude', -57, 'longitude', 135);
```



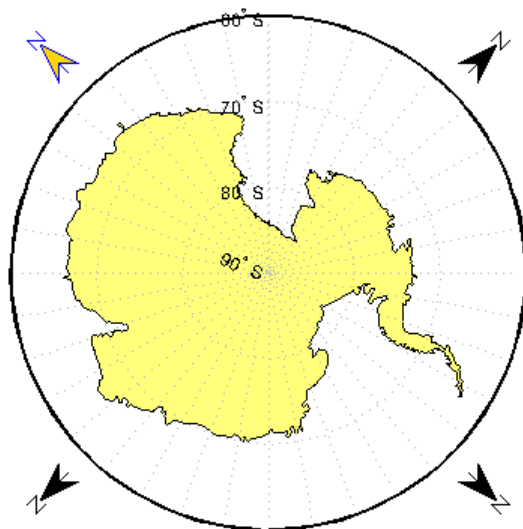
- 2** Click and drag the north arrow symbol to another corner of the map. Note that it always points to the North Pole.
- 3** Drag the north arrow back to the top left corner.
- 4** Right-click or **Ctrl**+click the north arrow. The Inputs for North Arrow dialog opens, which lets you specify the line 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 = handle('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?” on page 6-17
“Choropleth Maps” on page 6-18
“Special Thematic Mapping Functions” on page 6-23

### 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 provides functions to produce most of these types of map symbology.

## Choropleth Maps

The most familiar form of thematic map is probably the choropleth map (from the Greek *choros*, for place, and *plethos*, for magnitude). Often used to present data in newspapers, magazines, and reports, choropleth maps fill geographic zones (such as countries or states, but also matrices) with colors and/or patterns to represent nominal, ordinal, or cardinal data values. As there are usually more possible data values than unique symbols or colors capable of differentiating them, choropleth maps usually classify their data into value ranges.

Mapping Toolbox uses patch objects to construct choropleth maps. It assigns a color to each patch face to represent a specified variable, one value per patch. When the variable is scalar (as opposed to nominal) it generally represents a density (such as population per unit area), intensity (such as income per family), or incidence rate (such as fatalities per thousand persons). It can also convey extensive measurements or counts (such as electoral votes per state) if used carefully.

To make a choropleth map you need to input or compute a vector of values, one for each patch in a vector data set. Symbolizing such data values with Mapping Toolbox is straightforward. It involves assigning the data values to the `CData` property of a set of patches, and then setting up a colormap with an appropriate color scheme and range. Colormaps usually map  $N$  or fewer values (for  $N$  patches) to  $M$  colors.  $M$  can be any number between 2 and  $N$ , but typically ranges between 5 and 10.

In the following example, patches representing the 50 states of the U.S. (and the District of Columbia) are displayed and colored according to the surface areas calculated by the `areaaint` function. An equal-area projection is appropriate for this and other choropleth maps. This is because data is often computed or normalized over the patches being displayed, and thus area distortion should be minimized, even at the expense of shape distortion.

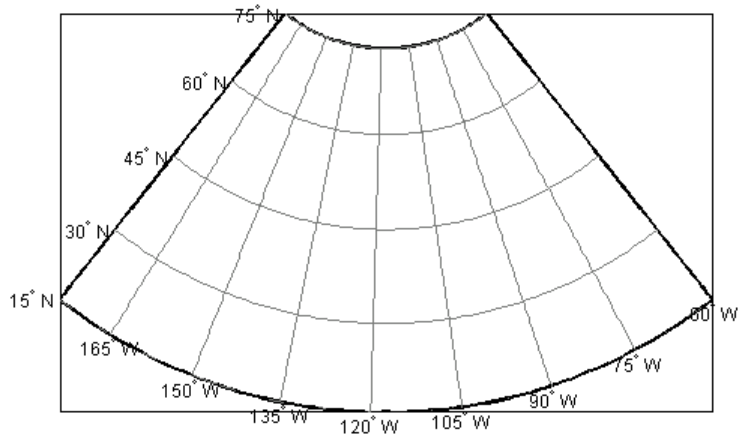
**1** Import low-resolution U.S. state boundary polygons:

```
states = shaperead('usastate10', 'UseGeoCoords', true);
```

This data set includes patch data for individual states, the United States, and its Great Lakes.

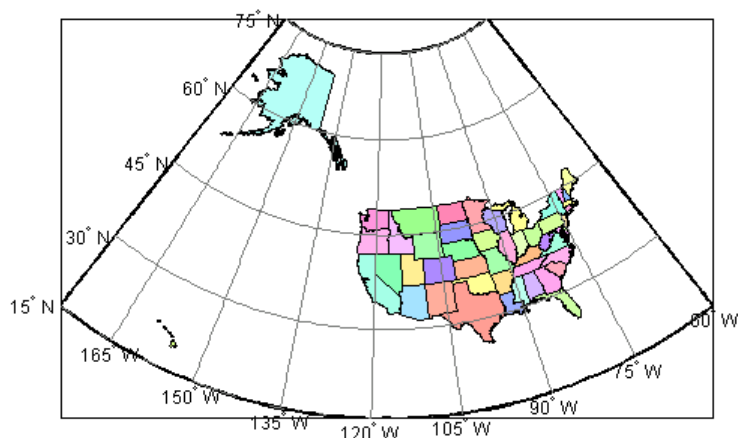
- 2** Set up map axes with a projection suitable to display all 50 states with equal areas, a graticule, and grid labels:

```
axesm('MapProjection', 'eqaonic', 'MapParallels', [],...
      'MapLatLimit', [15 75], 'MapLonLimit', [-175 -60],...
      'MLineLocation', 15, 'MLabelParallel', 'south',...
      'MeridianLabel', 'on', 'ParallelLabel', 'on',...
      'LineStyle', '-', 'GColor', 0.5*[1 1 1],...
      'Grid', 'on', 'Frame', 'on')
```



- 3** Draw the polygon map in the state structure using face colors randomly selected by polcmap:

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



**4** Choose an ellipsoid for computing spherical area:

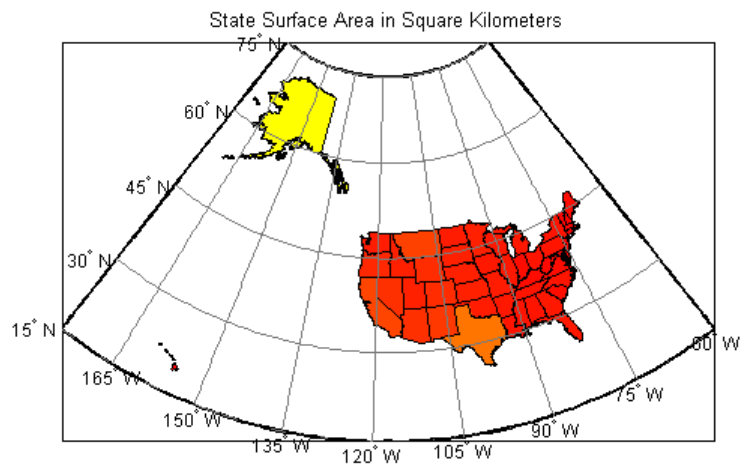
```
wgs84 = almanac('earth', 'geoid', 'kilometers', 'grs80');
```

**5** Add a 'SurfaceArea' field to the states geostruct, and assign surface areas in square kilometers for each U.S. state plus D.C. with a for loop:

```
for k = 1:numel(states)
    states(k).SurfaceArea = sum(areaint(states(k).Lat, ...
    states(k).Lon, wgs84));
end
maxarea = max([states.SurfaceArea]);
```

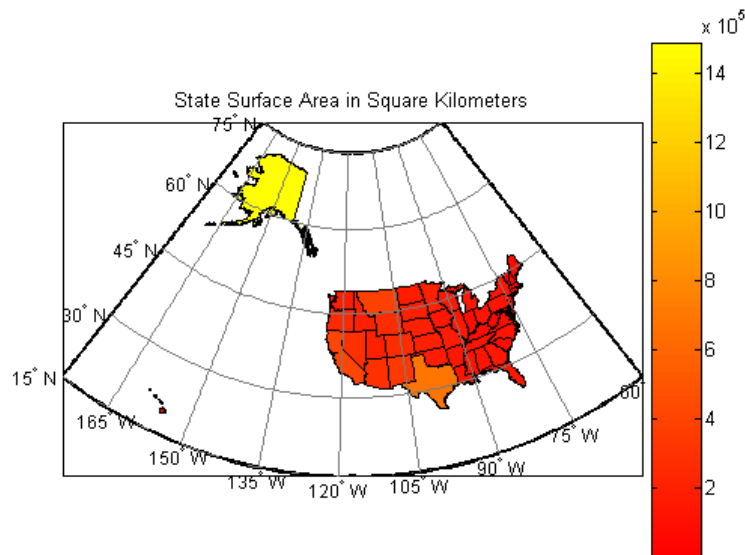
**6** Redisplay the states based on the surface area. Use a monotonic colormap from red to yellow.

```
surfaceColors = makesymbolspec('Polygon', ...
    {'SurfaceArea', [0 maxarea], ...
    'FaceColor', autumn(numel(states))});
geoshow(states, 'DisplayType', 'polygon', ...
'SymbolSpec', surfaceColors)
title('State Surface Area in Square Kilometers')
```



- 7** Show a colorbar as a key to the symbology, in its default location. This legend relates patch color to area in square km:

```
caxis([0 maxarea])  
colormap('autumn')  
colorbar
```



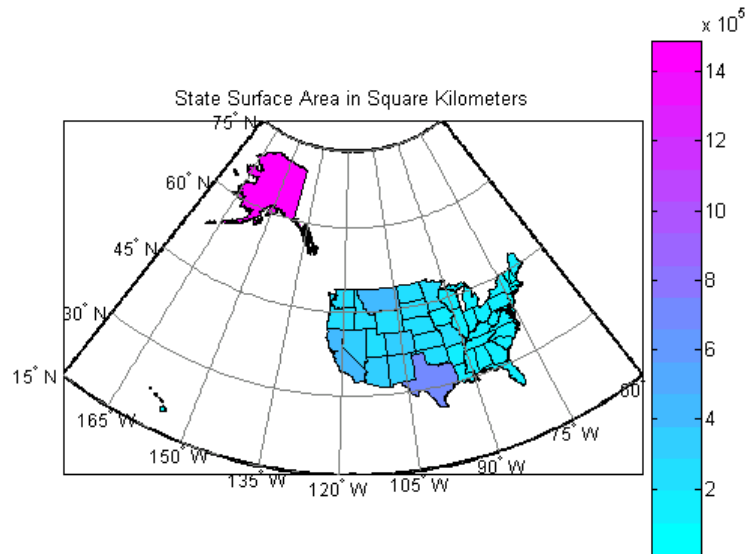
- 8 The map is mostly red, as the above figure shows. Experiment with other colormaps. Some names of predefined colormaps are autumn, cool, copper, gray, pink, and jet.

Note that while the color scale varies continuously, many states appear to be the same color. This is because of the skewed distribution of state areas. One way to differentiate the symbology is to clamp the lower end (because the smallest patches, such as District of Columbia and Rhode Island, are much smaller than average) and the upper end (because Alaska's area is so much larger than that of any other state).

- 9 Change the colormap to one that has more hues and a smaller number of steps, and redraw the colorbar to display the new value range:

```
minarea = 10000;
surfaceColors = makesymbolspec('Polygon',...
    {'Default','FaceColor','red'}, ...
    {'SurfaceArea', [ minarea maxarea], 'FaceColor', cool(16)});
geoshow(states,'DisplayType', 'polygon', ...
    'SymbolSpec', surfaceColors)
```

```
caxis([minarea maxarea])
colormap(cool(16))
colorbar
```



Note how you can specify the size of a colormap with the colormap syntax used above. Be aware that, because you clamped the value range, the numeric limits of the colorbar overstate the minimum area and understate the maximum area. However, the map gives much more information overall because more states have distinct symbology, as the resulting map depicts.

## Special Thematic Mapping Functions

In addition to choropleth maps, Mapping Toolbox provides other display and symbology functions. These include the following:

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

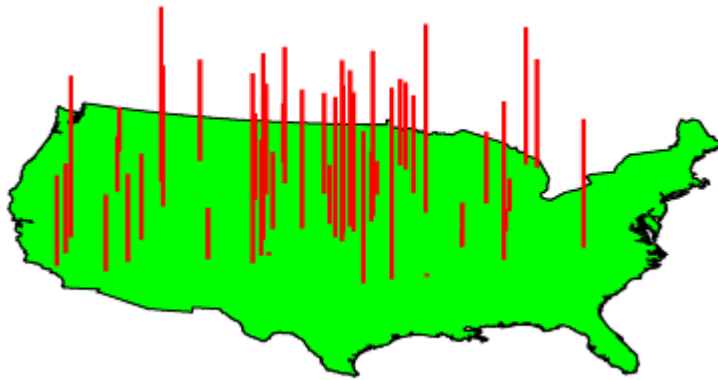
<b>Function</b>	<b>Used For</b>
<code>quiverm</code>	Plots directed vectors in 2-D from specified latitudes and longitudes with lengths also specified as latitudes and longitudes
<code>quiver3m</code>	Plots directed vectors in 3-D from specified latitudes, longitudes, and altitudes with lengths also specified as latitudes and longitudes and altitudes
<code>scatterm</code>	Draws fixed or proportional symbol maps for each point in a vector with specified marker symbol. Similar maps can be generated using <code>geoshow</code> and <code>mapshow</code> using appropriate symbol specifications (“ <code>symbolspecs</code> ”).
<code>stem3m</code>	Projects a 3-D stem plot map on the current map axes

The `cometm` and `quiverm` functions operate like their MATLAB counterparts `comet` and `quiver`. The `stem3m` function allows you to display geographic bar graphs. Like the MATLAB `scatter` function, the `scatterm` function allows you to display a thematic map with proportionally sized symbols. The `tissot` function calculates and displays Tissot Indicatrices, which graphically portray the shape distortions of any map projection. For more information on these capabilities, consult the descriptions of these functions in the reference pages.

### **Stem Maps**

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

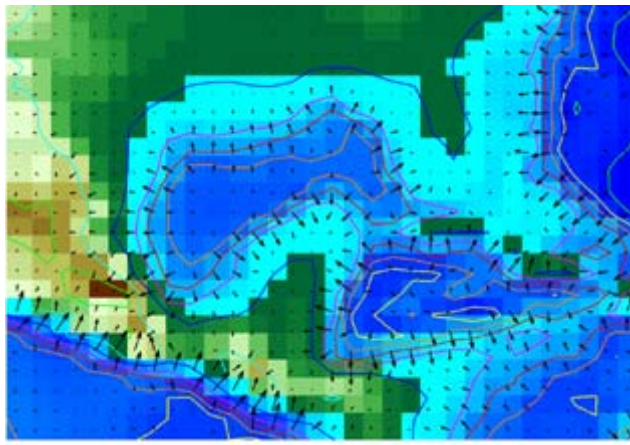




### Contour Maps

Contour and quiver plots can be useful in analyzing matrix data. In the following example, contour elevation lines have been drawn over a topographical map. The region displayed is the Gulf of Mexico, obtained from the topo matrix. Quiver plots have been added to visualize the gradient of the topographical matrix.

Here is the displayed map:

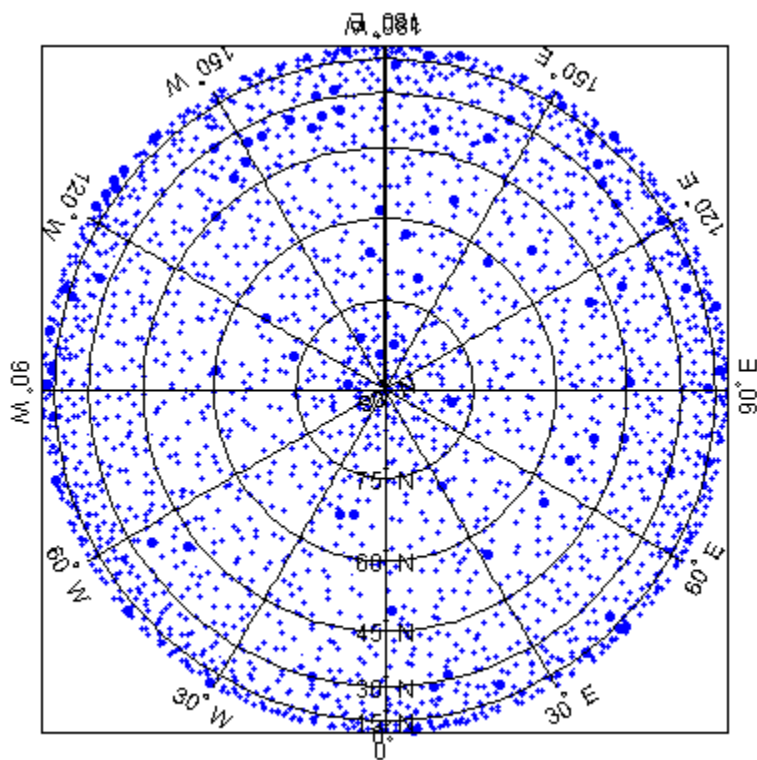


## Scatter Maps

The `scatterm` function plots symbols at specified point locations, like the MATLAB scatter function. If the symbols are small and inconspicuous and do not vary in size, the result is a *dot-distribution map*. If the symbols vary in size and/or shape according to a vector of attribute values, the result is a *proportional symbol map*.

Below is an example of using `scatterm` to create a star chart of the northern sky. The stars are represented by filled circles whose size is proportional to visual magnitude. To execute the following commands, select them all by dragging over the list in the Help browser, then right-click and choose Evaluate Selection:

```
close all; clear all
load stars
% Set all visual magnitude zero values to eps
index = find(vmag <= 0);
vmag(index) = eps;
% View the sky orthographically
axesm('MapProjection','ortho','Origin',[90 0])
setm(gca,'FLatLimit',[90 0],'MapLatLimit',[90 0])
gridm on
setm(gca,'LabelFormat','compass','LabelRotation','on')
setm(gca,'MLabelParallel',0,'PLabelMeridian',0)
setm(gca,'MeridianLabel','on','ParallelLabel','on')
setm(gca,'GLineStyle','-')
% Make scatterplot of vmag data with blue filled circles
scatterm(lat, long, vmag, 'b', 'filled')
```



## Using Cartesian MATLAB Display Functions

### In this section...

“Adding MATLAB Graphic Objects to Map Axes” on page 6-28

“Example 1: Triangulating Data Points” on page 6-28

“Example 2: Constructing Quiver Maps” on page 6-30

### Adding MATLAB Graphic Objects to Map Axes

If you cannot find a Mapping Toolbox display function that makes the kind of display that you need, you might be able to use MATLAB functions. When placing graphic objects on a map axes, you can use the MATLAB function to add the graphic objects to the display, using latitude and longitude as  $x$  and  $y$ , and then project the data afterwards.

---

**Note** Before applying MATLAB functions to geodata, you should take into consideration that performing Cartesian geometric operations on geographic coordinates can yield inaccurate results when the data covers large regions of a planet or lies near one of its poles.

---

### Example 1: Triangulating Data Points

Mapping Toolbox does not have a function that displays a triangulated surface from random data points, a structure generally known as a *triangulated irregular network* (TIN). However, MATLAB does have a function to create *Delaunay triangles*, a method that is often used to form TINs from projected point coordinate data. Explore triangulating some point data and bringing the result into Mapping Toolbox:

- 1 Use the seamount data provided with MATLAB:

```
load seamount
```

- 2 Determine the bounds of the coordinates and add a degree of white space:

```
latlim = [min(y) - .5 max(y) + .5];  
lonlim = [min(x) - .5 max(x) + .5];
```

- 3** Create map axes to contain the seamount region (worldmap selects a projection for you):

```
worldmap(latlim,lonlim)
```

- 4** Create a Delaunay triangulation of  $x$  and  $y$  (longitude and latitude):

```
tri = delaunay(y,x);
```

- 5** Generate a 3-D surface that combines the triangulation and  $z$ -values:

```
h = trisurf(tri,y,x,z);
```

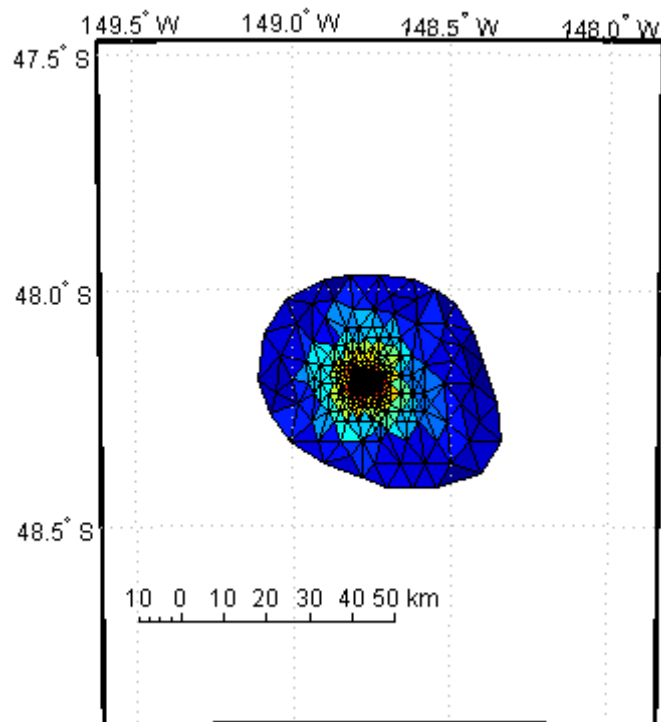
- 6** Map the surface onto the axes by projecting to the  $x$ - $y$  plane (project is a Mapping Toolbox function especially for this purpose):

```
project(h,'yx')
```

Note that even though the triangulated surface appears to be part of the map, it does not have a geostruct at this point (see “Mapping Toolbox Geographic Data Structures” on page 2-16).

- 7** Add a default graphic scale to the display:

```
scaleruler on
```



If, as in this example, the displayed objects are already in the right place and do not need to be projected, you can trim them to the map frame and convert them to mapped objects (having geostructs) using `trimcart` and `makemapped`. They can then be manipulated as if they had been created with map display functions.

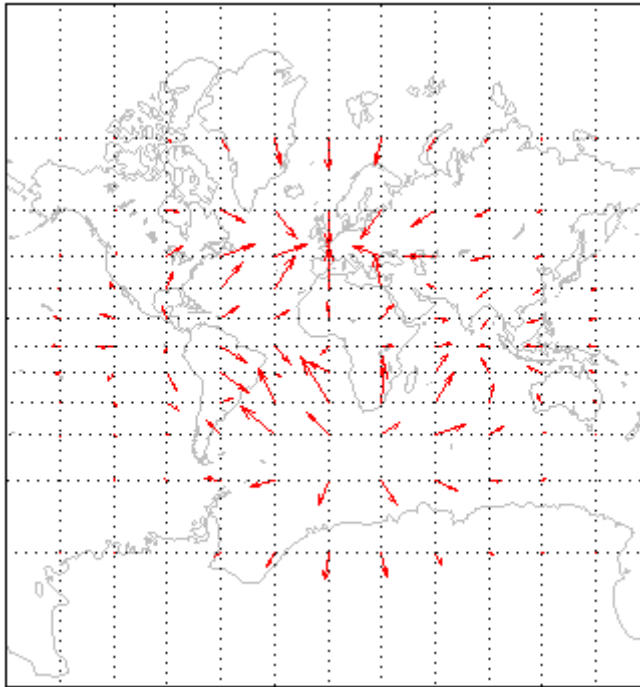
### Example 2: Constructing Quiver Maps

As was briefly described for text objects in “Projected and Unprojected Graphic Objects” on page 4-22, you can also combine Mapping Toolbox and MATLAB functions to mix spherical and Cartesian coordinates. An example would be a quiver plot (sometimes known as a *vector field*) in which the locations of the vectors are geographic, but the lengths, being specified by attributes, are not. In that case, you can use Mapping Toolbox projection calculations and MATLAB graphics functions. Cylindrical projections are the

simplest to use because north is up, south is down, and east and west are on an orthogonal axis.

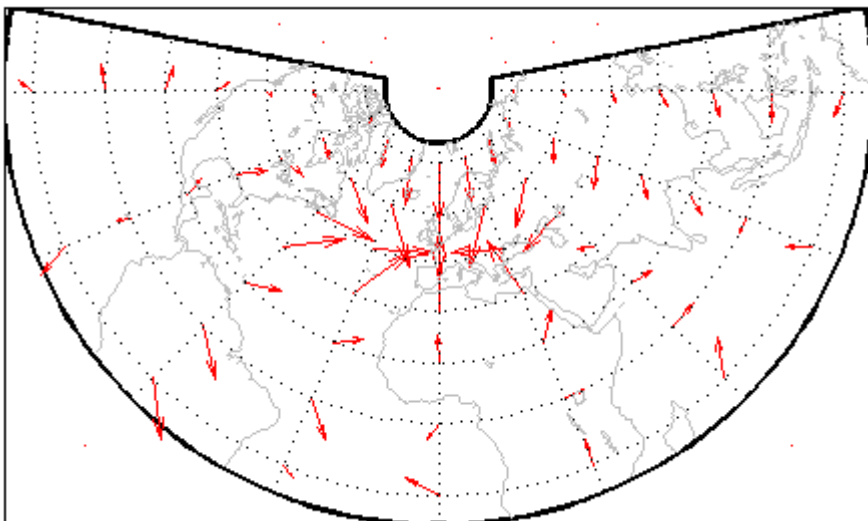
In this example, you will impose a quiver map of the slope of a surface on a world map. The surface is a Gaussian field generated by the MATLAB `peaks` function.

```
figure; axesm mercator; framem; gridm
load coast
plotm(lat,long,'color',[.75 .75 .75])
[u,v] = gradient(peaks(13)/10);
[mlat,m lon] = meshgrat(-90:15:90,-180:30:180);
[x,y] = mfw dtran(mlat,m lon);
h = quiver(x,y,u,v,.2,'r');
trimcart(h)
tightmap
```



An extra step might be required for noncylindrical projections. In these projections, compass directions vary with location. To make the directions agree with the map grid, vectors should be rotated to bring them into alignment. This can be done with the vector transformation function `vfwdtran`. Consider the same data displayed on a conic projection.

```
load coast; figure
axesm('lambert','MapLatLimit',[-20 80])
framem; gridm
plotm(lat,long,'color',[.75 .75 .75])
[u,v] = gradient(peaks(13)/10);
[mlat,m lon] = meshgrat(-90:15:90,-180:30:180);
[x,y] = mfwdtran(mlat,m lon);
thproj = deg2rad(vfwdtran(mlat,m lon,90*ones(size(mlat))));
[th,r] = cart2pol(u,v);
[uproj,vproj] = pol2cart(th+thproj,r);
h = quiver(x,y,uproj,vproj,0,'r');
trimcart(h)
tightmap
```





Conformal projections, such as this Lambert conformal conic, are often the best choice for quiver displays. They preserve angles, ensuring that the difference between north and east will always be 90 degrees in projected coordinates.

## Using Colormaps and Colorbars

### In this section...

“Colormap for Terrain Data” on page 6-34

“Contour Colormaps” on page 6-37

“Colormaps for Political Maps” on page 6-39

“Labeling Colorbars” on page 6-43

“Editing Colorbars” on page 6-44

### Colormap for Terrain Data

Colors and colorscales (ordered progressions of colors) are invaluable for representing geographic variables on maps, particularly when you create terrain and thematic maps. The following sections describe techniques and provide examples for applying colormaps and colorbars to maps.

In previous examples, the function `demcmap` was used to color several digital elevation model (DEM) topographic displays. This function creates colormaps appropriate to rendering DEMs, although it is certainly not limited to DEMs.

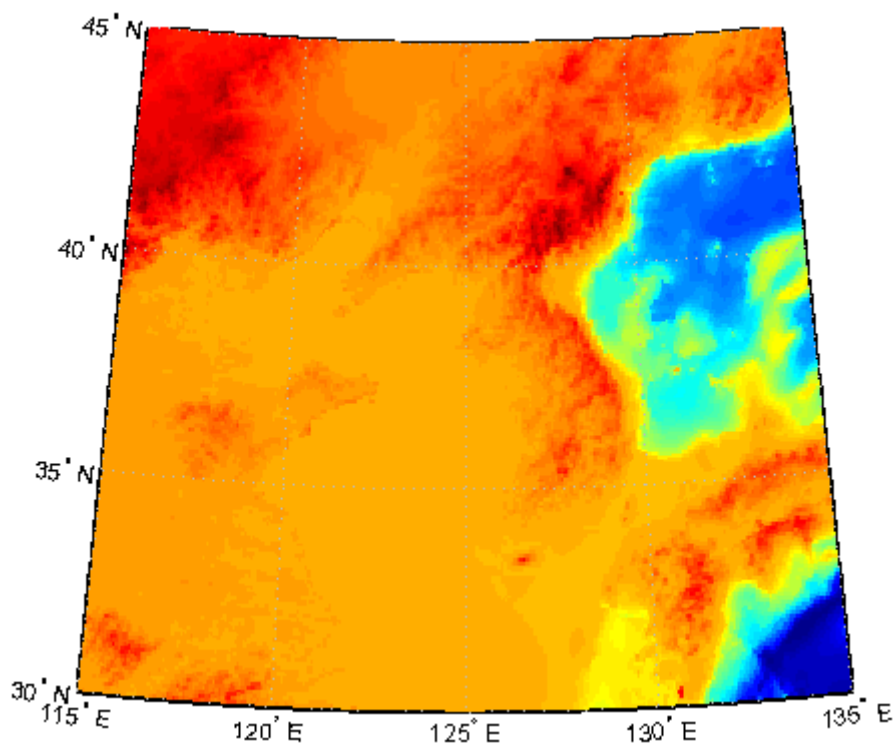
These colormaps, by default, have atlas-like colors varying with elevation or depth that properly preserve the land-sea interface. In cartography, such color schemes are called *hypsometric tints*.

- 1 Here you explore `demcmap` using the topographic data for the Korean peninsula provided in the `korea` data set. To set up an appropriate map projection, pass the `korea` data grid and referencing vector to `worldmap`:

```
load korea
figure
worldmap(map,refvec)
```

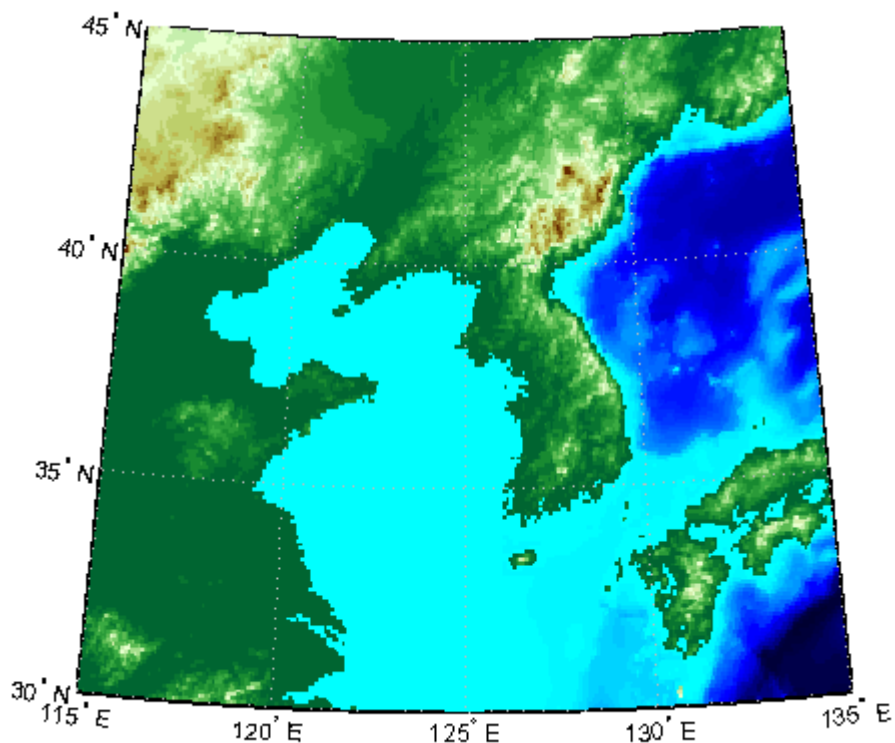
- 2 Display the data grid with `geoshow`:

```
geoshow(map, refvec, 'DisplayType', 'mesh')
```



- 3** The Korea DEM is displayed using the default colormap, which is inappropriate and causes the surface to be unrecognizable. Now apply the default DEM colormap:

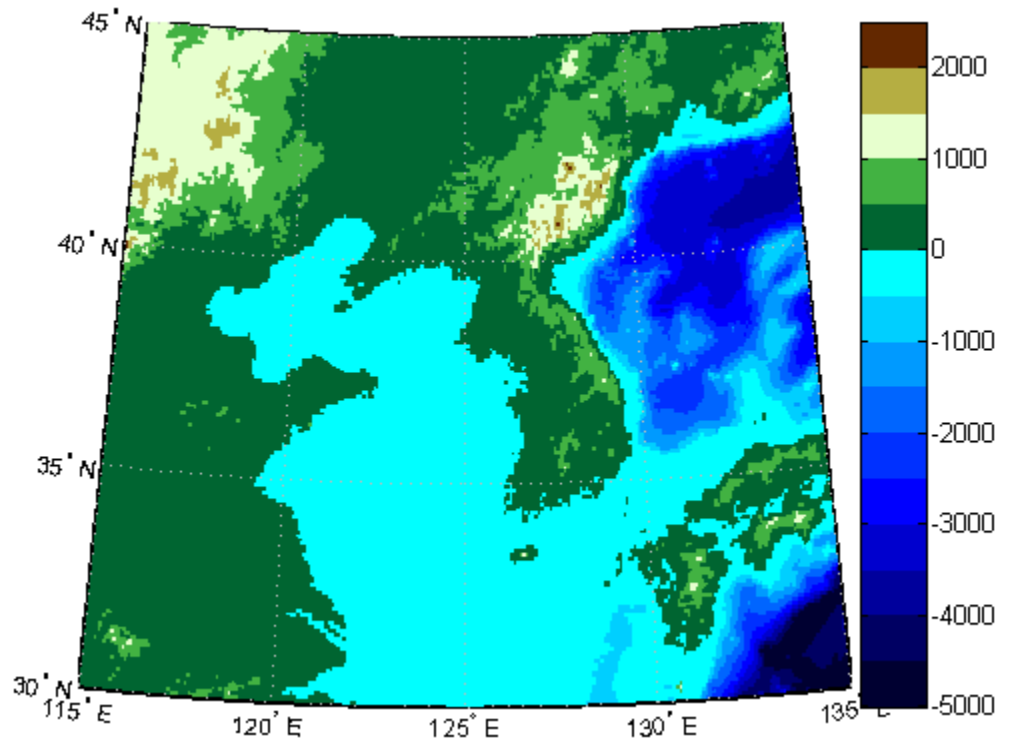
```
demcmap(map)
```



- 4 You can also make `demcmap` assign all altitudes within a particular range to the same color. This results in a quasi-contour map with breaks at a constant interval. Now color this map using the same color scheme coarsened to display 500 meter bands:

```
demcmap('inc',map,500)
colorbar
```

Note that the first argument to `demcmap`, `'inc'`, indicates that the third argument should be interpreted as a value range. If you prefer, you can specify the desired number of colors with the third argument by setting the first argument to `'size'`.



## Contour Colormaps

You can create colormaps that make surfaces look like contour maps for other types of data besides terrain. The `contourcmap` function creates a colormap that has color changes at a fixed value increment. Its required arguments are the increment value and the name of a colormap function. Optionally, you can also use `contourcmap` to add and label a colorbar similarly to the MATLAB `colorbar` function:

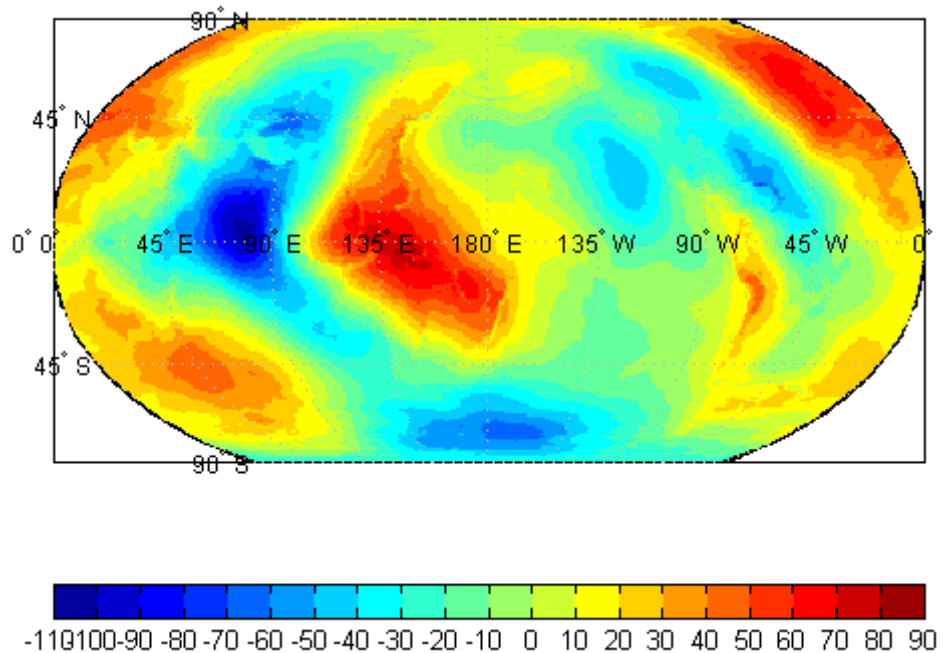
- 1 Explore `contourcmap` by loading the world geoid data set and rendering it with a default colormap:

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

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

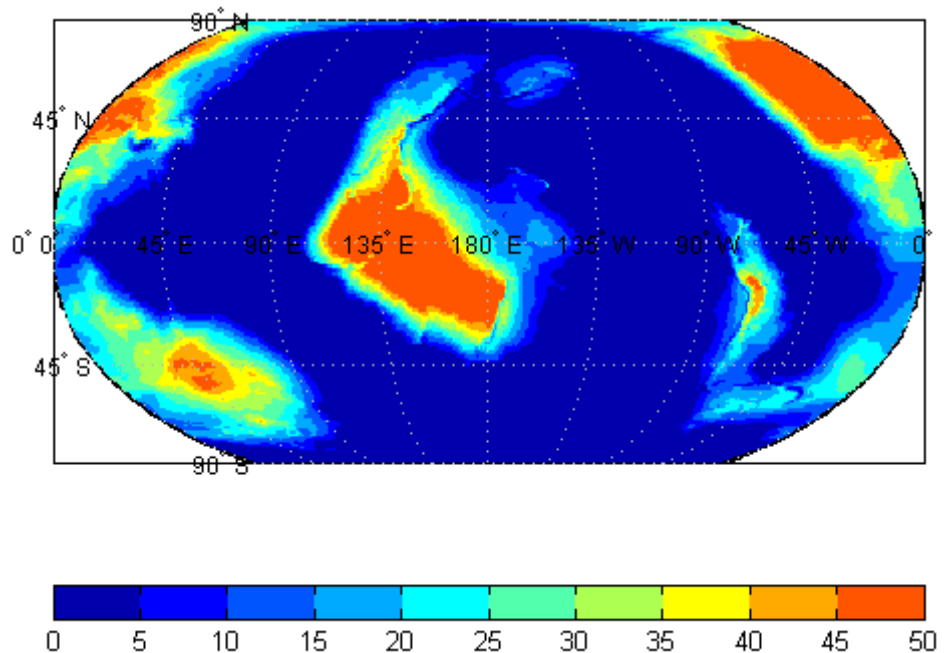
- 2 Use `contourcmap` to specify a contour interval of 10 (meters), and to place a colorbar beneath the map:

```
contourcmap(10, 'jet', 'colorbar', 'on', 'location', 'horizontal')
```



- 3 If you want to render a restricted value range, you can enter a vector of evenly spaced values for the first argument. Here you specify a 5-meter interval and truncate symbology at 0 meters on the low end and 50 meters at the high end:

```
contourcmap([0:5:50],...  
'jet', 'colorbar', 'on', 'location', 'horizontal')
```



Should you need to write a custom colormap function, for example, one that has irregular contour intervals, you can easily do so, but it should work like those provided with MATLAB.

## Colormaps for Political Maps

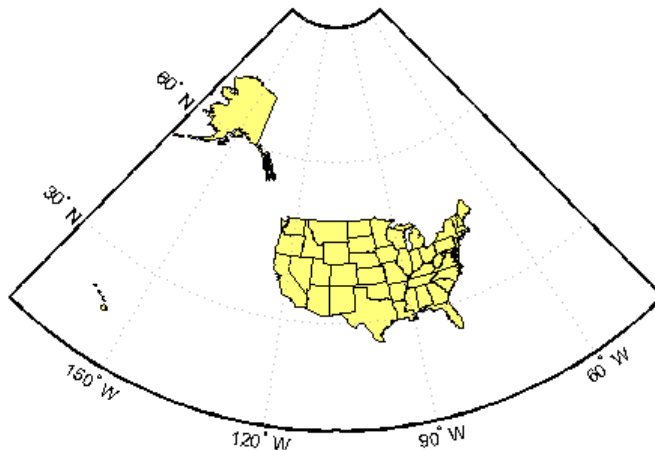
Political maps typically use muted, contrasting colors that make it easy to distinguish one country from its neighbors. You can create colormaps of this kind using the `polcmap` function. The `polcmap` function creates a colormap with randomly selected colors of all hues. Since the colors are random, if you don't like the result, execute `polcmap` again to generate a different colormap:

- 1 To explore political colormaps, display the `usastate10` data set as patches, setting up the map with `worldmap` and plotting it with `geoshow`:

```
figure
worldmap na
```

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

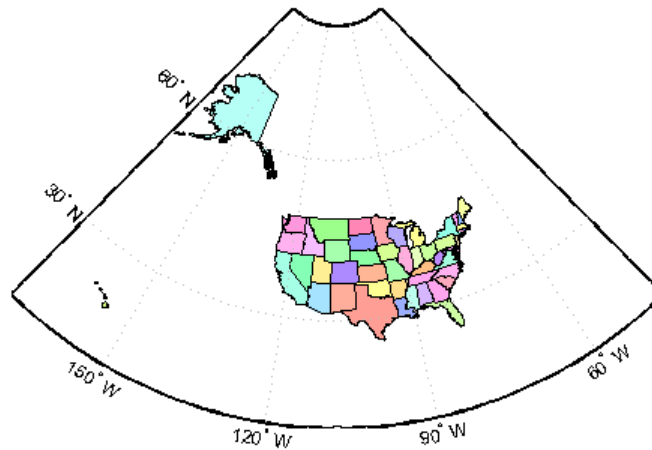
Note that the default face color is black, which is not very interesting.



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

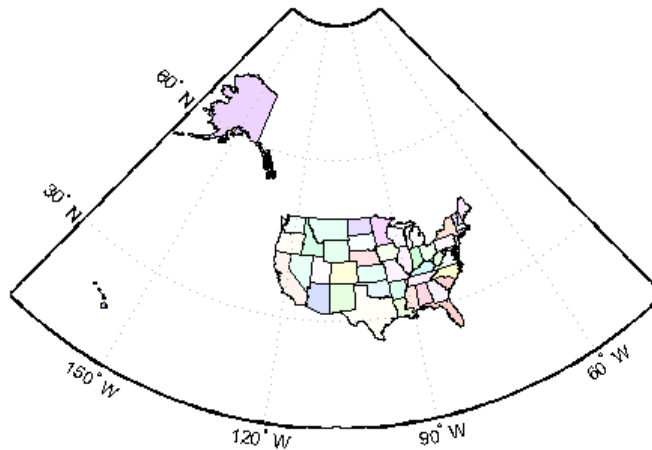
```
faceColors = makesymbolspec('Polygon',...  
    {'INDEX', [1 numel(states)], 'FaceColor',...  
    polcmap(numel(states))});  
geoshow(states, 'SymbolSpec', faceColors)
```





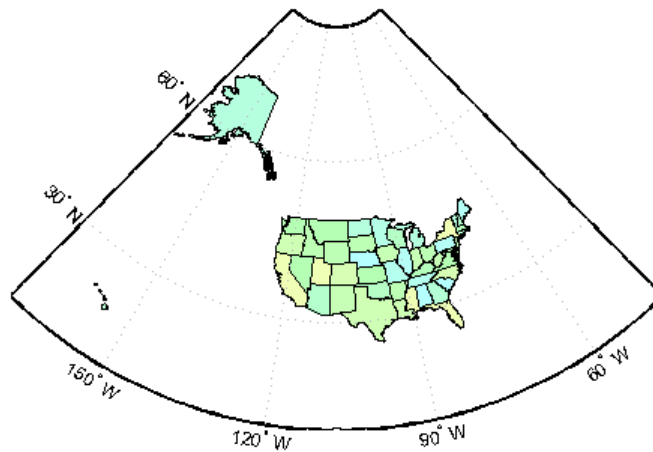
- 3** The `polcmap` function can also control the number and saturation of colors. Reissue the command specifying 256 colors and a maximum saturation of 0.2. To ensure that the colormap is always the same, reset the seed on the MATLAB random number function using the 'state' argument with a fixed value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

```
figure
worldmap na
rand('state',0)
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', polcmap(256,.2)});
geoshow(states, 'SymbolSpec', faceColors)
```



- 4** For maximum control over the colors, specify the ranges of hues, saturations, and values. Use the same set of random color indices as before.

```
figure
worldmap na
rand('state',0)
faceColors = makesymbolspec('Polygon', ...
    {'INDEX', [1 numel(states)], ...
    'FaceColor', polcmap(256,[.2 .5],[.3 .3],[1 1]) });
geoshow(states, 'SymbolSpec', faceColors)
```




---

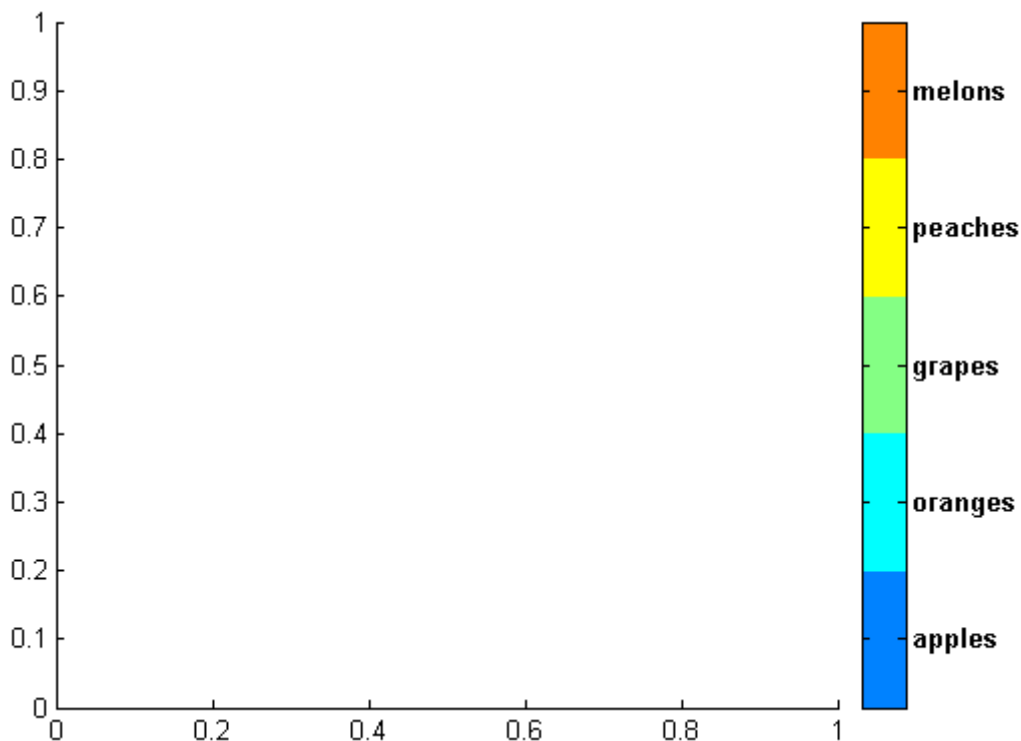
**Note** The famous Four Color theorem states that any political map can be colored to completely differentiate neighboring patches using only four colors. Experiment to find how many colors it takes to color neighbors differently with `polcmap`.

---

## Labeling Colorbars

Political maps are an example of nominal data display. Many nominal data sets have names associated with a set of integer values, or consist of codes that identify values that are ordinal in nature (such as low, medium, and high). The function `lcolorbar` creates a colorbar having a text label aligned with each color. Nominal colorbars are customarily used only with small colormaps (where 10 categories or fewer are being displayed). `lcolorbar` has options for orienting the colorbar and aligning text in addition to the graphic properties it shares with axes objects.

```
figure; colormap(jet(5))
labels = {'apples', 'oranges', 'grapes', 'peaches', 'melons'};
lcolorbar(labels, 'fontweight', 'bold');
```



### Editing Colorbars

Maps of nominal data often require colormaps with special colors for each index value. To avoid building such colormaps by hand, use the MATLAB GUI for colormaps, `colormapeditor`, described in the MATLAB Function Reference pages. Also see the MATLAB `colormap` function documentation.

## Printing Maps to Scale

Maps are often printed at a size that makes objects on paper a particular fraction of their real size. The linear ratio of the mapped to real object sizes is called *map scale*, and it is usually notated with a colon as “1:1,000,000” or “1:24,000.” Another way of specifying scale is to call out the printed and real lengths, for example “1 inch = 1 mile.”

You can specify the printed scale using the `paperscale` function. It modifies the size of the printed area on the page to match the scale. If the resulting dimensions are larger than your paper, you can reduce the amount of empty space around the map using `tightmap`, `zoom`, or `panzoom`, and by changing the axes position to fill the figure. This also reduces the amount of memory needed to print with the `zbuffer` (raster image) renderer. Be sure to set the paper scale last. For example,

```
set(gca,'Units','Normalized','Position',[0 0 1 1])
tightmap
paperscale(1,'in', 5,'miles')
```

The `paperscale` function also can take a scale denominator as its first and only argument. If you want the map to be printed at 1:20,000,000, type

```
paperscale(2e7)
```

To check the size and extent of text and the relative position of axes, use `previewmap`, which resizes the figure to the printed size.

```
previewmap
```

For more information on printing, see the “Printing and Exporting” section of the MATLAB Graphics documentation.



# Manipulating Geospatial Data

---

For some purposes, geospatial data is fine to use as is. Sooner or later, though, you need to extract, combine, massage, and transform geodata. This chapter discusses some of the tools and techniques that Mapping Toolbox provides for such purposes.

Manipulating Vector Geodata (p. 7-2)	Ways to extract, compare, densify, and reduce data
Manipulating Raster Geodata (p. 7-31)	Encoding, extracting, and transforming gridded data values

## Manipulating Vector Geodata

### In this section...

“Repackaging Vector Objects” on page 7-2  
“Matching Line Segments” on page 7-4  
“Geographic Interpolation of Vectors” on page 7-5  
“Vector Intersections” on page 7-8  
“Polygon Area” on page 7-11  
“Overlaying Polygons with Set Logic” on page 7-12  
“Cutting Polygons at the Date Line” on page 7-17  
“Building Buffer Zones” on page 7-19  
“Trimming Vector Data to a Rectangular Region” on page 7-21  
“Trimming Vector Data to an Arbitrary Region” on page 7-24  
“Simplifying Vector Coordinate Data” on page 7-25

### 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 lon] == [lat2 lon2]

ans =
     1     1
     1     1
     1     1
     0     0
     1     1
     1     1
     1     1
     1     1
    
```

The logical comparison is false for the NaN delimiters, by definition.

**6** You can test for global equality, including NaNs, as follows:

```
isequalwithequalnans(lat,lat2) & isequalwithequalnans(long,lon2)

ans =
     1
```

See the `polysplit` and `polyjoin` reference pages for further information.

## Matching Line Segments

A common operation on sets of line segments is the concatenation of segments that have matching endpoints. The `polymerge` command compares endpoints of segments within latitude and longitude vectors to identify endpoints that match exactly or lie within a specified distance. The matching segments are then concatenated, and the process continues until no more coincidental endpoints can be found. The two required arguments are a latitude ( $x$ ) vector and a longitude ( $y$ ) vector. The following exercise shows this process at work.

## Linking Line Segments into Polygons

**1** Construct column vectors representing coordinate values:

```
lat = [3 2 NaN 1 2 NaN 5 6 NaN 3 4]';
lon = [13 12 NaN 11 12 NaN 15 16 NaN 13 14]';
```

**2** Concatenate the segments that match exactly:

```
[latm,lonm] = polymerge(lat,lon);
[latm lonm]

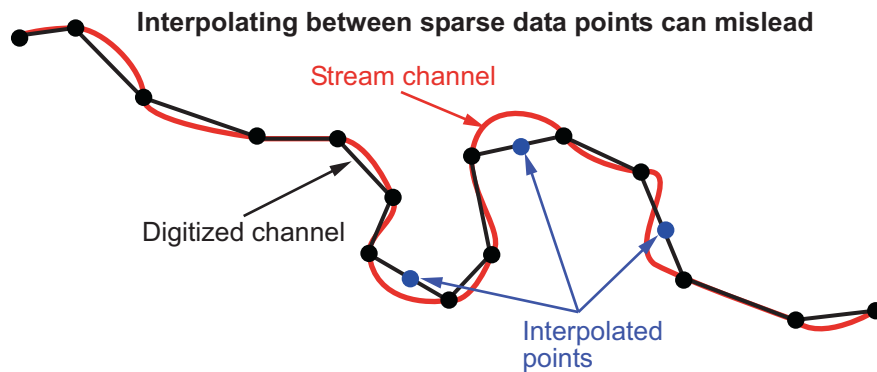
ans =
     5     15
     6     16
    NaN    NaN
     4     14
     3     13
     3     13
     2     12
     2     12
```

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 `interp` 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 `interp` to fill in any gaps greater than 1° in either direction:

```
[newlats,newlongs] = interp(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 `interp` reference page for further information.

## Interpolating Coordinates at Specific Locations

`interp` 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 `interp` 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  $7.3^\circ$  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')
```

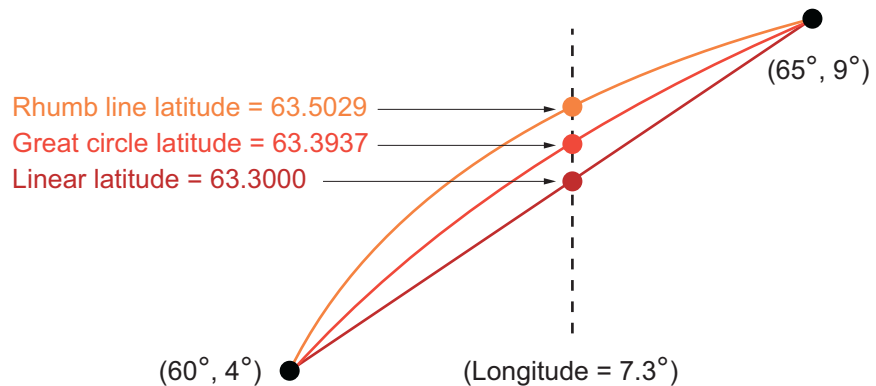
```
newlat =  
63.5029
```

- 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

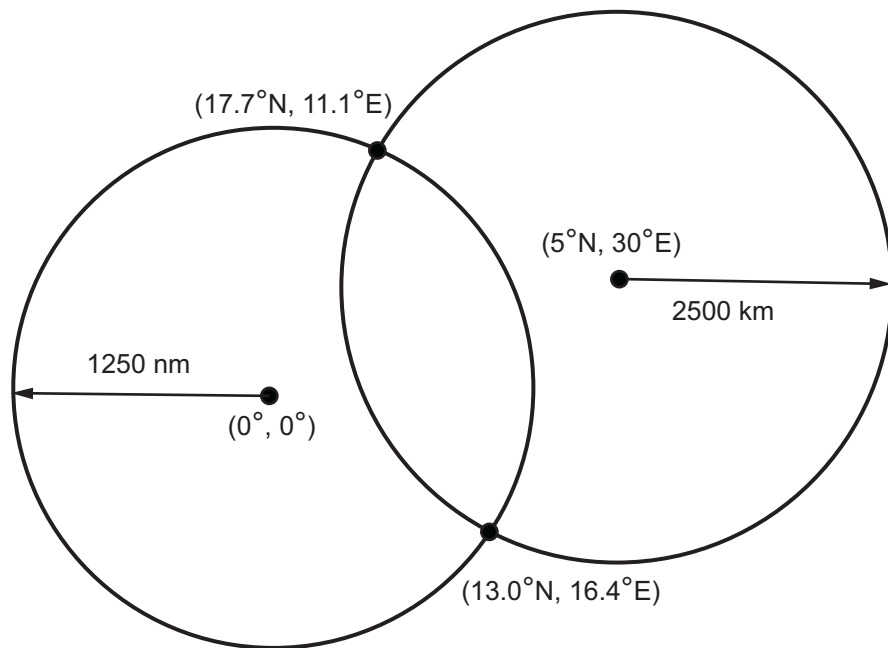
Mapping Toolbox provides a set of functions to perform intersection calculations on vector data computed by the toolbox, which include great and small circles as well as rhumb line tracks. The functions also determine intersections of arbitrary vector data.

Compute the intersection of a small circle centered at  $(0^\circ, 0^\circ)$  with a radius of 1250 nautical miles and a small circle centered at  $(5^\circ\text{N}, 30^\circ\text{E})$  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 ( $37^\circ\text{N}, 76^\circ\text{W}$ ), maintaining a steady due-east course ( $90^\circ$ ), and another ship setting sail from Dakar, Senegal ( $15^\circ\text{N}, 17^\circ\text{W}$ ), with a steady northwest course ( $315^\circ$ ). Where would the tracks of the two vessels cross?

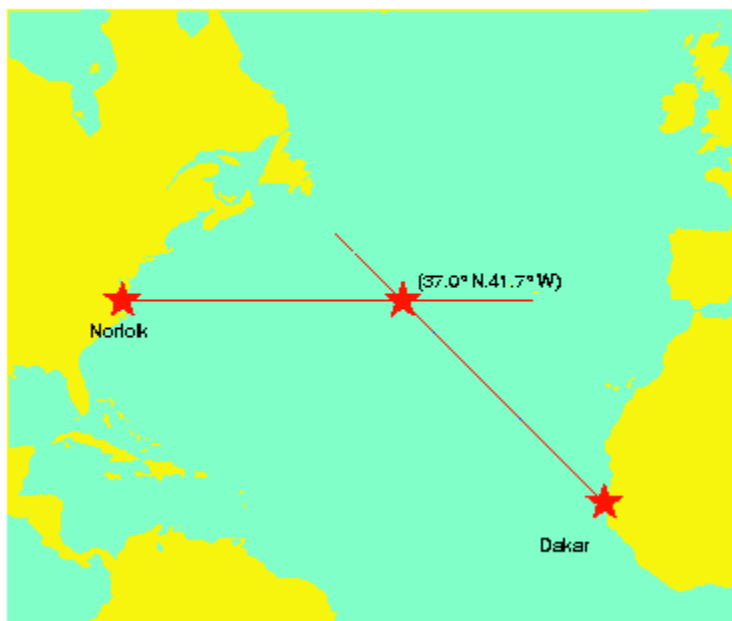
```
[lat, long] = rhxrh(37, -76, 90, 15, -17, 315)
```

```

lat =
  37
    
```

```
long =  
-41.7028
```

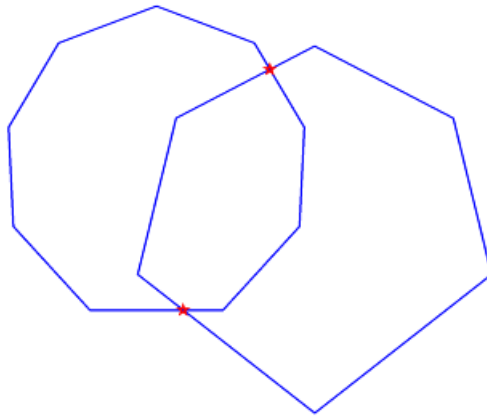
The intersection of the tracks is at (37°N,41.7°W), which is roughly 600 nautical miles west of the Azores in the Atlantic Ocean.



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 = almanac('earth','radius');
```

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

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

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

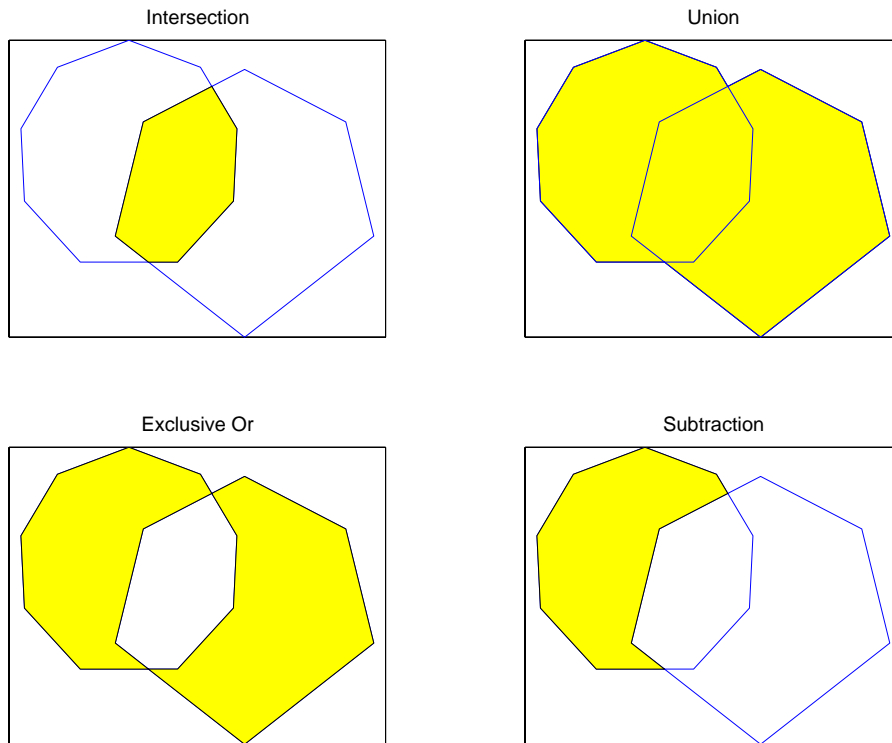
```
load conus  
earthradius = almanac('earth','radius','miles');  
area = areaint(gtlakelat,gtlakelon,earthradius)
```

```
area =  
1.0e+004 *  
 8.0119  
 1.0381  
 0.7634
```

Again, three areas are returned, the largest for the polygon representing Superior, Michigan, and Huron together, the other two for Erie and Ontario.

### Overlaying Polygons with Set Logic

Polygon set operations are used to answer a variety of questions about logical relationships of vector data polygon objects. Standard set operations include intersection, union, subtraction, and an exclusive OR operation. The `polybool` function performs these operations on two sets of vectors, which can represent  $x$ - $y$  or latitude-longitude coordinate pairs. In computing points where boundaries intersect, interpolations are carried out on the coordinates as if they were planar. Here is an example that shows all the available operations.



The result is returned as NaN-clipped vectors by default. In cases where it is important to distinguish outer contours of polygons from interior holes, `polybool` can also accept inputs and return outputs as cell arrays. In the cell array format, a cell array entry starts with the list of points making up the outer contour. Subsequent NaN-clipped faces within the cell entry are interpreted as interior holes.

### Overlaying Polygons with the `polybool` Function

The following exercise demonstrates how you can use `polybool`:

- 1 Construct a twelve-sided polygon:

```
theta = -(0:pi/6:2*pi)';  
lat1 = sin(theta);  
lon1 = cos(theta);
```

**2** Construct a triangle that overlaps it:

```
lat2 = [0 1 -1 0]';  
lon2 = [0 2 2 0]';
```

**3** Plot the two shapes together with blue and red lines:

```
axesm miller  
plotm(lat1,lon1,'b')  
plotm(lat2,lon2,'r')
```

**4** Compute the intersection polygon and plot it as a green patch:

```
[loni,lati] = polybool('intersection',lon1,lat1,lon2,lat2);  
[lati loni]  
geoshow(lati,loni,'DisplayType','polygon','FaceColor','g')
```

```
ans =  
      0      1.0000  
-0.4409      0.8819  
      0      0  
 0.4409      0.8819  
      0      1.0000
```

**5** Compute the union polygon and plot it as a magenta patch:

```
[lonu,latu] = polybool('union',lon1,lat1,lon2,lat2);  
[latu lonu]  
geoshow(latu,lonu,'DisplayType','polygon','FaceColor','m')
```

```
ans =  
-1.0000      2.0000  
-0.4409      0.8819  
-0.5000      0.8660  
-0.8660      0.5000  
-1.0000      0.0000  
-0.8660     -0.5000
```

```

-0.5000 -0.8660
      0 -1.0000
 0.5000 -0.8660
 0.8660 -0.5000
 1.0000 -0.0000
 0.8660  0.5000
 0.5000  0.8660
 0.4409  0.8819
 1.0000  2.0000
-1.0000  2.0000

```

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

```

[lonx,latx] = polybool('xor',lon1,lat1,lon2,lat2);
[latx lonx]
geoshow(latx,lonx,'DisplayType','polygon','FaceColor','y')

```

```

ans =
-1.0000  2.0000
-0.4409  0.8819
-0.5000  0.8660
-0.8660  0.5000
-1.0000  0.0000
-0.8660 -0.5000
-0.5000 -0.8660
      0 -1.0000
 0.5000 -0.8660
 0.8660 -0.5000
 1.0000 -0.0000
 0.8660  0.5000
 0.5000  0.8660
 0.4409  0.8819
 1.0000  2.0000
-1.0000  2.0000
      NaN      NaN
 0.4409  0.8819
      0      0
-0.4409  0.8819
      0  1.0000
 0.4409  0.8819

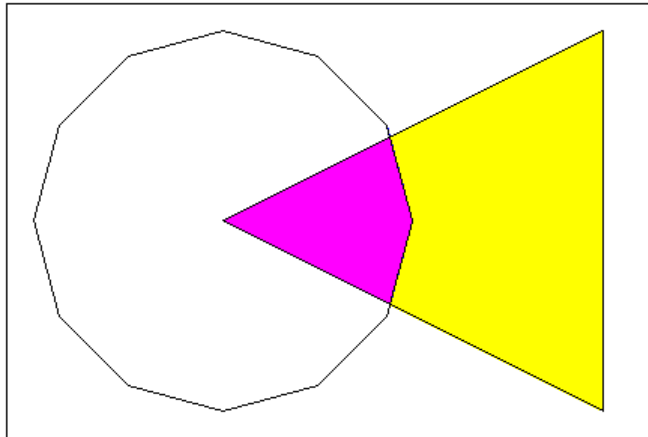
```

- 7** Subtract the triangle from the 12-sided polygon and plot the resulting concave polygon as a white patch:

```
[lonm,latm] = polybool('minus',lon1,lat1,lon2,lat2);  
[latm lonm]  
geoshow(latm,lonm,'DisplayType','polygon','FaceColor','w')
```

```
ans =  
    0.8660    0.5000  
    0.5000    0.8660  
    0.4409    0.8819  
         0         0  
   -0.4409    0.8819  
   -0.5000    0.8660  
   -0.8660    0.5000  
   -1.0000    0.0000  
   -0.8660   -0.5000  
   -0.5000   -0.8660  
         0        -1.0000  
    0.5000   -0.8660  
    0.8660   -0.5000  
    1.0000   -0.0000  
    0.8660    0.5000
```

The final set of colored shapes is shown below.



See the `polybool` reference page for further information.

## Cutting Polygons at the Date Line

Polygon set operations treat input vectors as plane coordinates. The `polyxpoly` function can be confused by geographic data that has discontinuities in longitude coordinates at date-line crossings. This can happen when points with longitudes near  $180^\circ$  connect to points with longitudes near  $-180^\circ$ , as might be the case for eastern Siberia and Antarctica, and also for small circles and other patch objects generated by toolbox functions.

You can prepare such geographic data for use with `polybool` or for patch rendering by cutting the polygons at the date line with the `flatearthpoly` function. The result of `flatearthpoly` is a polygon with points inserted to follow the date line up to the pole, traverse the longitudes at the pole, and return to the date line crossing along the other edge of the date line.

## Removing Discontinuities from a Small Circle

1 Create an orthographic view of the Earth and plot coast on it:

```
axesm ortho
setm(gca,'Origin',[60 170]); framem on; gridm on
```

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

- 2** Generate a small circle that encompasses the North Pole and color it yellow:

```
[latc,lonc] = scircle1(75,45,30);
patchm(latc,lonc,'y')
```

- 3** Flatten the small circle with `flatearthpoly`:

```
[latf,lonf] = flatearthpoly(latc,lonc);
```

- 4** Plot the cut circle that you just generated as a magenta line:

```
plotm(latf,lonf,'m')
```

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

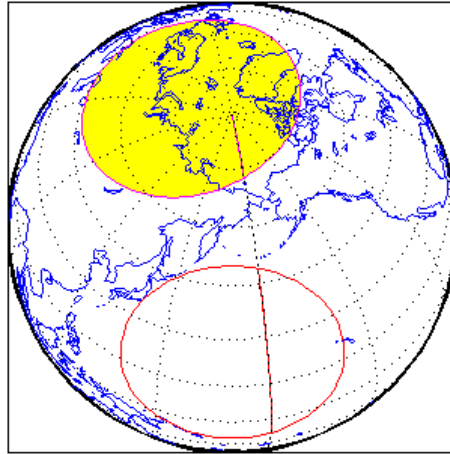
```
[latc1 lonc1] = scircle1(20, 170, 30);
```

- 6** Flatten it and plot it as a red line:

```
[latf1,lonf1] = flatearthpoly(latc1,lonc1);
plotm(latf1,lonf1,'r')
```

The following figure shows the result of these operations. Note that the second small circle, which does not cover a pole, has been clipped into two pieces along the date line. On the right, the polygon for the first small circle is plotted in plane coordinates to illustrate its flattened shape.





The `flatearthpoly` function assumes that the interior of the polygon being flattened is in the hemisphere that contains most of its edge points. Thus a polygon produced by `flatearthpoly` does not cover more than a hemisphere.

---

**Note** As this figure illustrates, you do not need to use `flatearthpoly` to prepare data for a map display. The Mapping Toolbox display functions automatically cut and trim geographic data if required by the map projection. Use this function only when conducting set operations on polygons.

---

See the `flatearthpoly` reference page for further information.

## Building Buffer Zones

A *buffer zone* is the area within a specified distance of a map feature. For vector geodata, buffer zones are constructed as polygons. For raster geodata, buffer zones are collections of contiguous, identically coded grid cells. When the feature is a polygon, a buffer zone can be defined as the locus of points within a certain distance of its boundary, either inside or outside the polygon. A buffer zone for a linear object is the locus of points a certain distance away from it. Buffer zones form equidistant contour lines around objects.

The `bufferm` function computes and returns vectors that represent a set of points that define a buffer zone. It forms the buffer by placing small circles at

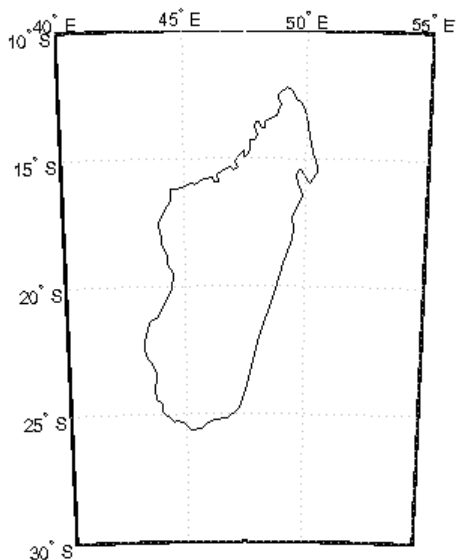
the vertices of the polygon and rectangles along each of its line segments, and applying the union set operation to these objects.

### Generating a Buffer Around a Compound Polygon

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

- 1 Create a base map of the area surrounding Madagascar, and hide the border:

```
ax = worldmap('madagascar');
madagascar = shaperead('landareas',...
    'UseGeoCoords', true,...
    'Selector', {@(name)strcmpi(name,'Madagascar'), 'Name'});
geoshow(ax, madagascar, 'FaceColor', 'none');
madaLat = madagascar.Lat;
madaLon = madagascar.Lon;
```



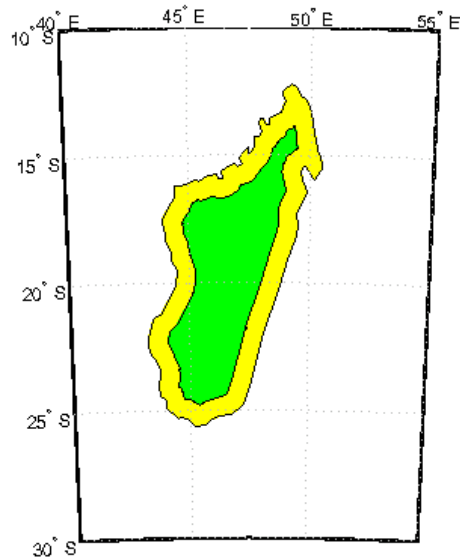
- 2** Use `bufferm` to process the polygon and output a buffer zone .75 degrees inland:

```
[latb,lonb] = bufferm(madaLat, madaLon, .75, 'in');
```

This can take several minutes, because of the great number of geometric computations that `bufferm` is performing.

- 3** Show the buffer zone in yellow, and the rest of the region in green. This is achieved by drawing Madagascar in yellow and the buffer zone in green:

```
patchesm(madaLat, madaLon, 'y')
patchesm(latb, lonb, 'g')
```



## Trimming Vector Data to a Rectangular Region

It is not unusual for vector data to extend beyond the geographic region currently of interest. For example, you might have coastline data for the entire world (such as the coast data set), but are interested in mapping Australia only. In this and other situations, you might want to eliminate unnecessary data from the workspace and from calculations in order to save memory or to speed up processing and display.

Line data and patch data need to be trimmed differently. You can trim line data by simply removing points outside the region of interest by clipping lines at the map frame or to some other defined region. Patch data requires a more complicated method to ensure that the patch objects are correctly formed.

For the vector data, two functions are available to achieve this. If the vectors are to be handled as line data, the `maptriml` function returns variables containing only those points that lie within the defined region. If, instead, you want to maintain polygon format, use the `maptrimp` function. Be aware, however, that patch-trimmed data is usually larger and more expensive to compute.

---

**Note** When drawing maps, Mapping Toolbox automatically trims 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

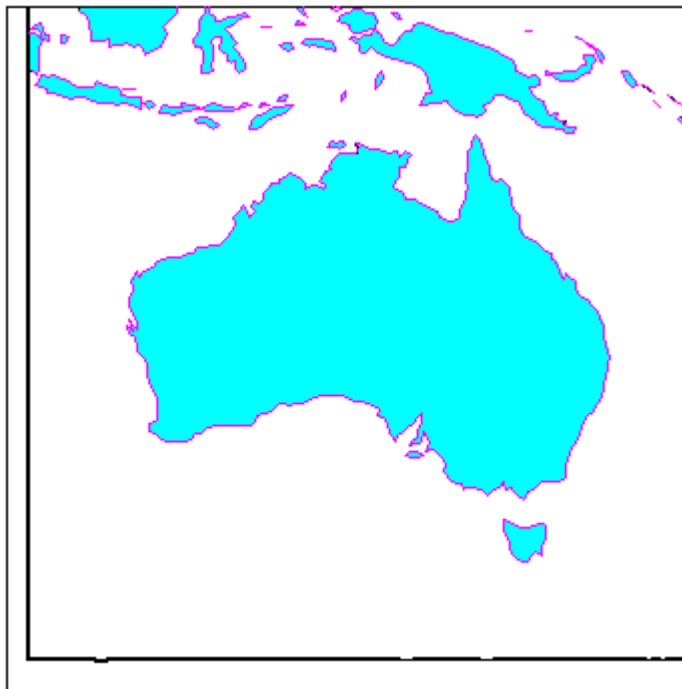
Note that the clipped data is only 10% as large as the original data set.

**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 `maptrim1` and `maptrim` 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. Mapping 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
```

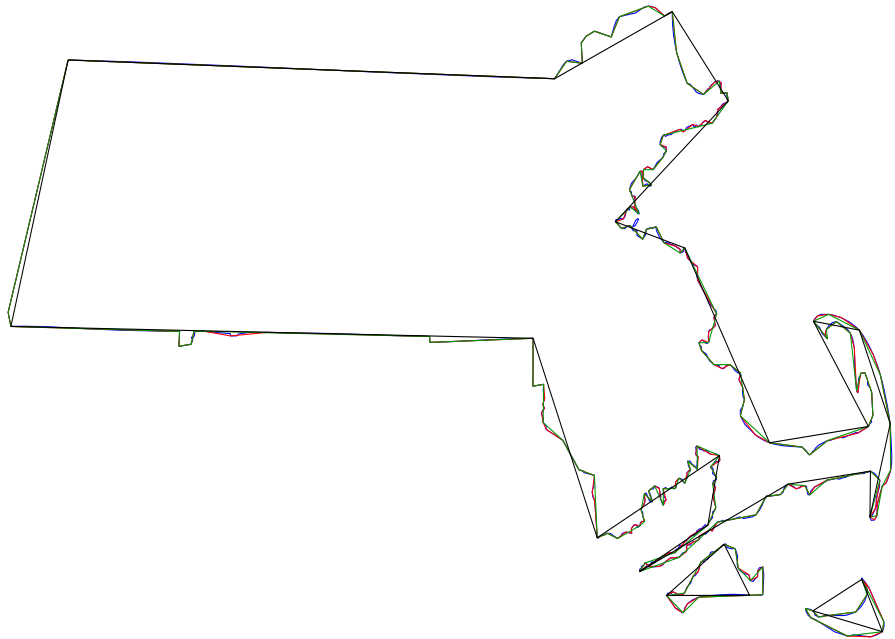
```
geoshow(maLat2, maLon2, 'DisplayType', 'line', 'color', [0 .6 0])
```

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.

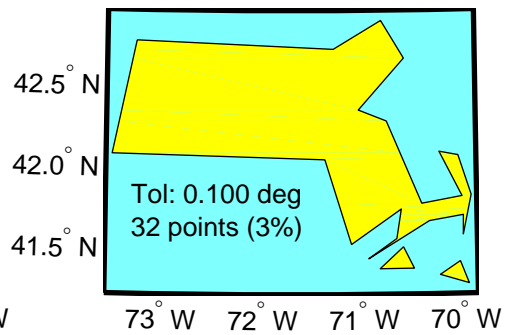
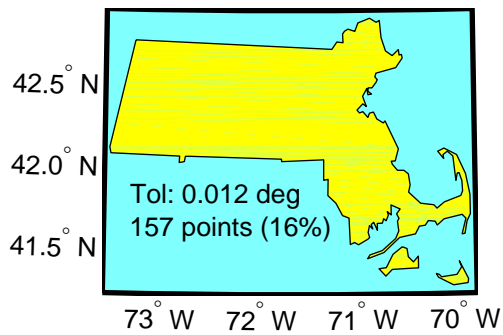
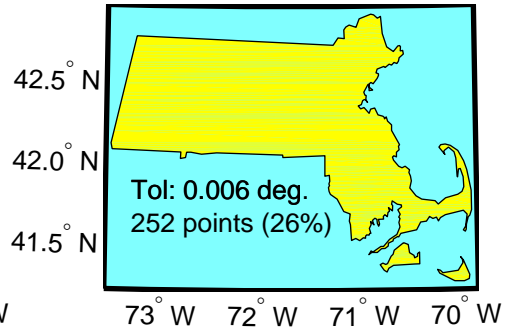
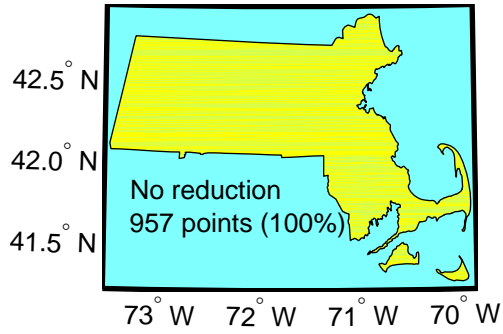


As this example and the composite map below demonstrate, the visual effects of point reduction are subtle, up to a point. Most of the vertices can be eliminated before the effects of line simplification are very noticeable. Experimentation is usually required, because the visual effects a given value 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 `reduce` can result in inconsistencies in boundary representation and gaps at points where states meet. For best results, `reduce` 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” on page 7-31

“Data Grids as Logical Variables” on page 7-39

“Data Grid Values Along a Path” on page 7-42

“Data Grid Gradient, Slope, and Aspect” on page 7-45

### 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. Mapping Toolbox provides GUI tools to help you do 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:

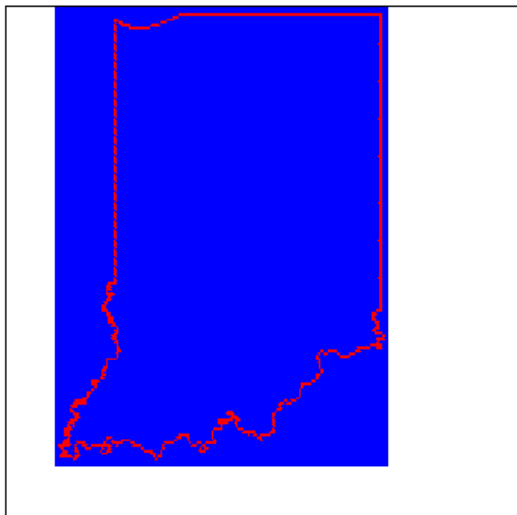
```
gridDensity = 40;
[inGrid, inRefVec] = vec2mtx(inLat, inLon, gridDensity);
whos
```

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.

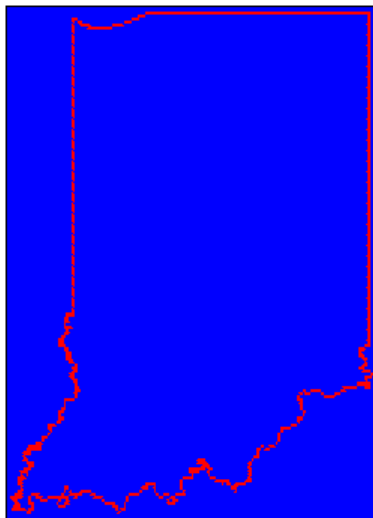
- 3** Make a map of the data grid in contrasting colors:

```
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:

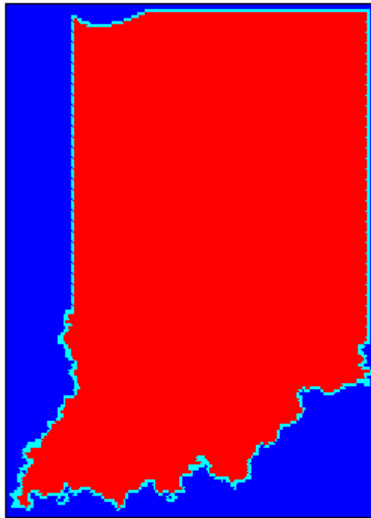
```
inPt = round([size(inGrid)/2, 3]);  
inGrid3 = encodem(inGrid, inPt,1);
```

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)
```

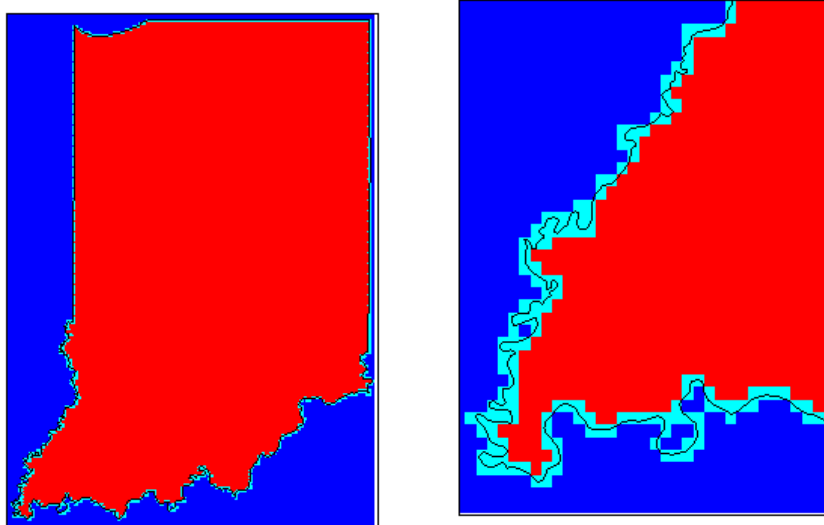




7 Plot the original vectors on the grid to see how well data was rasterized:

```
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 `maptrim` (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'});
```

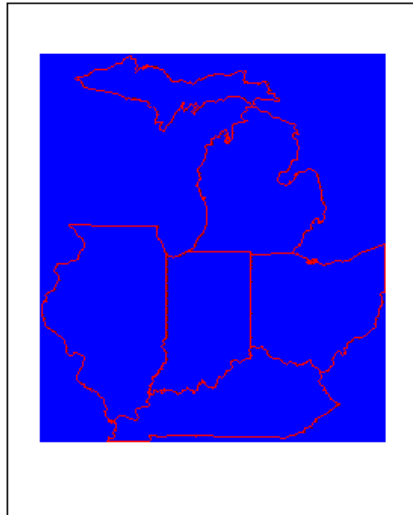
```
meLat = [centralUS.Lat];
meLon = [centralUS.Lon];
```

- 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.

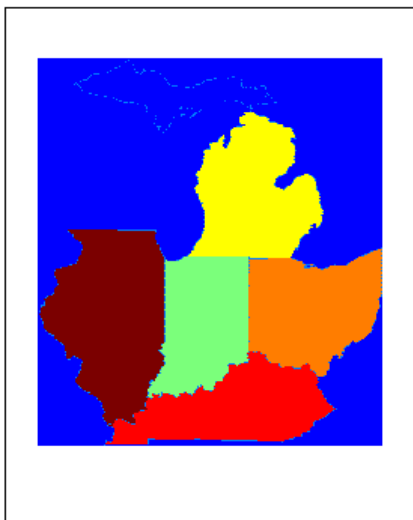
- 5** Use `encodem` to fill each country with a unique value, producing a new grid:

```
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. `Themaptrim` 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 in registration), 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)
```

Mapping Toolbox provides functions enabling 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), and might need to be cast to `double` in order to be used in certain functions. The following example shows how you can generate grids of all 1s and all 0s.

## Generating “Blank” Logical Grids

Construct a pair of five-cell-per-degree grids. They will contain doubles.

- 1 Cover the conterminous United States with a grid of 1s; define the country’s bounding latitudes and longitudes and the grid resolution:

```
latlims = [25 55]; longlims = [-130 -60]; scale = 5;
```

- 2 Generate a grid of all 1s over this region at 1/5-degree resolution:

```
onesgrid = onem(latlims,lonlims,scale);
```

**3** Generate a grid of all 0s over this region at 1/5-degree resolution:

```
zerosgrid = zerom(latlims,lonlims,scale);
```

Turn the grids into logical-valued grids and note the difference in size:

```
lonesgrid = logical(onesgrid);  
lzerosgrid = logical(zerosgrid);  
whos
```

Name	Size	Bytes	Class
latlims	1x2	16	double
lonesgrid	150x350	52500	logical
lonlims	1x2	16	double
lzerosgrid	150x350	52500	logical
onesgrid	150x350	420000	double
scale	1x1	8	double
zerosgrid	150x350	420000	double

**4** Create a referencing vector for mapping the grids:

```
gridref = [5 latlims(2) lonlims(1)]
```

```
gridref =  
    5    55   -130
```

Remember that referencing vectors take the form

```
[cells-per-degree northern-latitude western-longitude]
```

See the reference pages for `onem` and `zerom` for more details. You can create grids of all NaNs and sparse grids of all 0s in a similar fashion with the commands `nanm` and `spzerom`, respectively.

## 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
a = areamat((topo>0),topolegend)
```

```
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),topolegend,almanac('earth','radius'))
```

```
a =
1.4739e+08
```

- 3 Use the `usamtx` data grid codes to find the area of a specific state within the U.S.A. As an example, determine the area of the state of Texas, which is coded as 46 in the `usamtx` grid:

```
load usamtx
a = areamat((map==46), maplegend, almanac('earth', 'radius'))
```

```
a =
6.2528e+005
```

The grid codes 625,277 square kilometers of land area as belonging to the U.S.

- 4 You can construct more complex queries. For instance, using the last example, compute what portion of the land area of the conterminous U.S. that Texas occupies (water and bordering countries are coded with 2 and 3, respectively):

```
usaland = areamat((map>3|map==1), maplegend);
texasland = areamat((map==46), maplegend);
texasratio = texasland/usaland
```

```
texasratio =
    0.0735
```

This indicates that Texas occupies roughly 7.35% of the land area of the U.S.

For further information, see the `areamat` reference page.

## Data Grid Values Along a Path

A common application for gridded geodata is to calculate data values along a path, for example, the computation of terrain height along a transect, a road, or a flight path. The `mapprofile` function does this, based on numerical data defining a set of waypoints, or by defining them interactively via graphic input from a map display. Values computed for the resulting profile can be displayed in a new plot or returned as output arguments for further analysis or display.

### Using the `mapprofile` Function

The following example computes the elevation profile along a straight line:

- 1 Load the Korean elevation data:

```
figure;
load korea
```

- 2 Get its latitude and longitude limits using `limitm` and use them to set up a map frame via `worldmap`:

```
[latlim, lonlim] = limitm(map, maplegend);
```



```
worldmap(latlim, lonlim)
```

worldmap plots only the map frame.

- 3** Render the map and apply a digital elevation model (DEM) colormap to it:

```
meshm(map,maplegend,size(map),map)  
demcmap(map)
```

- 4** Define endpoints for a straight-line transect through the region:

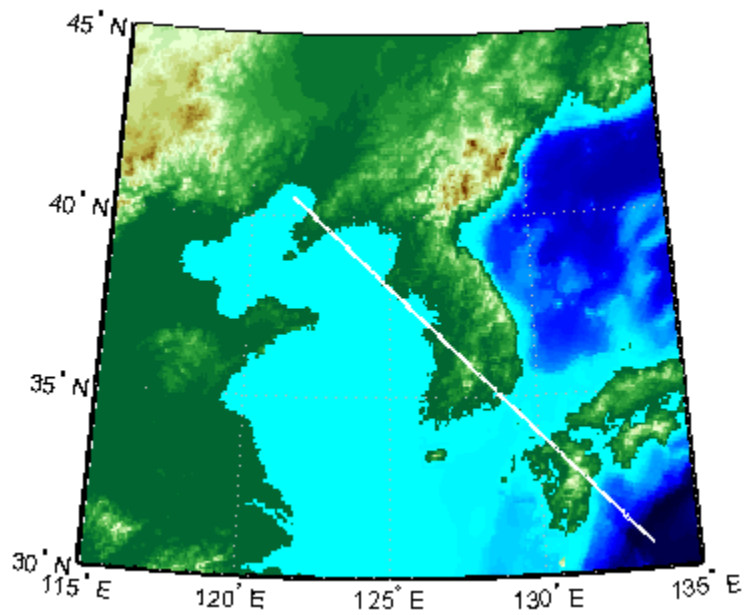
```
plat = [40.5 30.7];  
plon = [121.5 133.5];
```

- 5** Compute the elevation profile, defaulting the track type to great circle and the interpolation type to bilinear:

```
[z,rng,lat,lon] = mapprofile(map,maplegend,plat,plon);
```

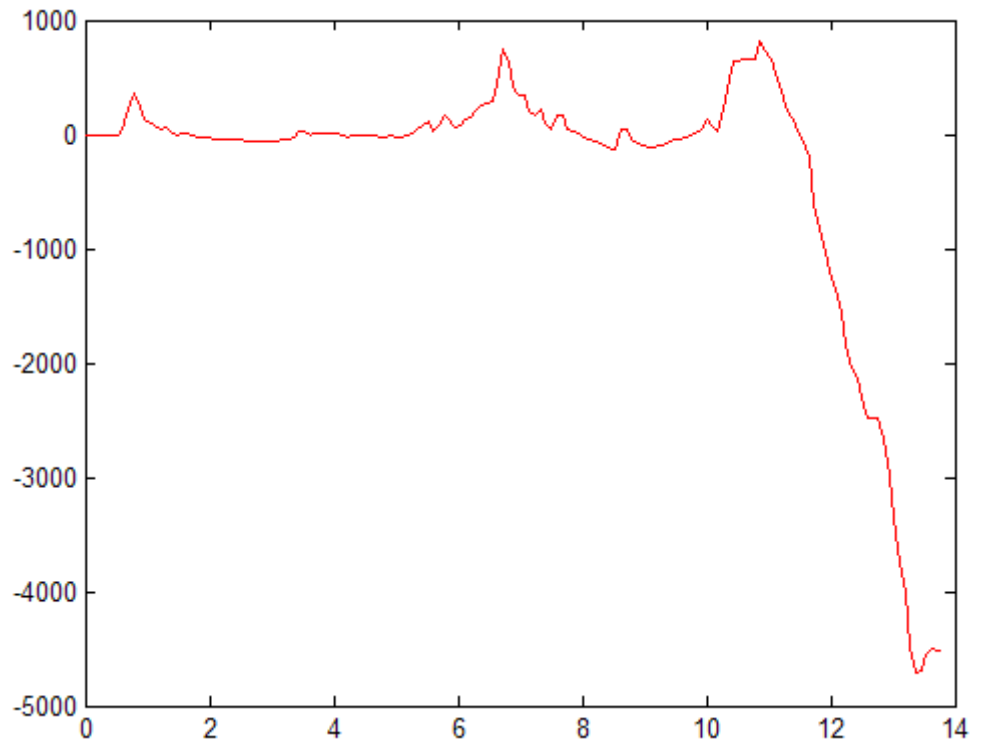
- 6** Draw the transect in 3-D so it follows the terrain:

```
plot3m(lat,lon,z,'w','LineWidth',2)
```



**7** Construct a plot of transect elevation and range:

```
figure; plot(rng,z,'r')
```



The `mapprofile` function has other useful options, including the ability to interactively define tracks and specify units of distance for them. For further information, see the `mapprofile` reference page.

## Data Grid Gradient, Slope, and Aspect

A map profile is often used to determine slopes along a path. A related application is the calculation of slope at all points on a matrix. The `gradientm` function uses a finite-difference approach to compute gradients for either a regular or a georeferenced data grid. The function returns the components of the gradient in the north and east directions (i.e., north-to-south, east-to-west), as well as slope and aspect. The *gradient* components are the change in the grid variable per meter of distance in the north and east directions. If the grid contains elevations in meters, the *aspect* and *slope* are the angles of the surface normal clockwise from north and up from the

horizontal. Slope is defined as the change in elevation per unit distance along the path of steepest ascent or descent from a grid cell to one of its eight immediate neighbors, expressed as the arctangent. The angles are in units of degrees by default.

### Computing Gradient Data from a Regular Data Grid

The following example illustrates computation of gradient, slope, and aspect data grids for a regular data grid based on the MATLAB peaks function:

- 1 Construct a 100-by-100 grid using the MATLAB peaks function and construct a referencing vector for it:

```
datagrid = 500*peaks(100);  
gridrv = [ 1000 0 0];
```

- 2 Use `gradientm` to generate grids containing aspect, slope, gradients to north, and gradients to east:

```
[aspect,slope,gradN,gradE] = gradientm(datagrid,gridrv);  
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,gridrv)  
colormap (jet(64))  
colorbar('vert')  
title('Peaks: elevation')  
axis square
```

**4** Clear the frame and display the slope grid:

```
figure; axesm eqacyl
meshm(slope,gridrv)
colormap (jet(64))
colorbar('vert');
title('Peaks: slope')
```

**5** Map the aspect grid:

```
figure; axesm eqacyl
meshm(aspect,gridrv)
colormap (jet(64))
colorbar('vert');
title('Peaks: aspect')
```

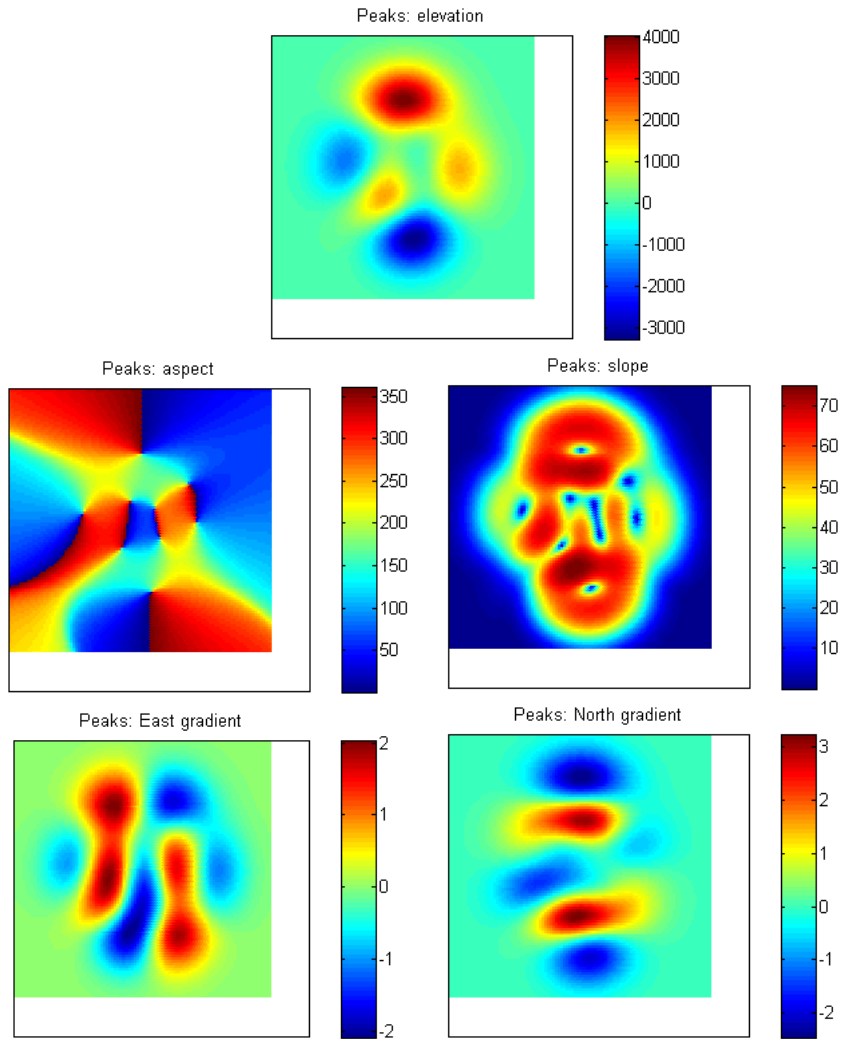
**6** Map the gradients to the north:

```
figure; axesm eqacyl
meshm(gradN,gridrv)
colormap (jet(64))
colorbar('vert');
title('Peaks: North gradient')
```

**7** Finally, map the gradients to the east:

```
figure; axesm eqacyl
meshm(gradE,gridrv)
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.



# 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.

What Is a Map Projection? (p. 8-3)	Flattening the Earth to comprehend its features
Quantitative Properties of Map Projections (p. 8-4)	What properties of maps the geometric construction of map projections influences and constrains
The Three Main Families of Map Projections (p. 8-6)	Making maps by projecting the globe onto cylinders, cones, and planes
Projection Aspect (p. 8-11)	How the orientation vector affects map displays
Projection Parameters (p. 8-19)	What parameters projections can have and how they influence the appearance and properties of maps
Visualizing and Quantifying Projection Distortions (p. 8-28)	Calculating and communicating the kinds of spatial error that map projections can have

Accessing, Computing, and Inverting Map Projection Data (p. 8-38)	Projecting coordinates using objects and retrieving projected coordinates from figure objects
Working with the UTM System (p. 8-52)	Understanding the Universal Transverse Mercator family of map projections
Summary and Guide to Projections (p. 8-64)	The properties of each projection supported by Mapping Toolbox

If you are not acquainted with the types, properties, and uses of map projections, read the first four sections. When constructing maps—especially in an environment in which a variety of projections are readily available—it is important to understand how to evaluate projections to select one appropriate to the contents and purpose of a given map.



## What Is a Map Projection?

Human beings have known that the shape of the Earth resembles a sphere and not a flat surface since classical times, and possibly much earlier than that. If the world were indeed flat, cartography would be much simpler because map projections would be unnecessary.

To represent a curved surface such as the Earth in two dimensions, you must geometrically transform (literally, and in the mathematical sense, “map”) that surface to a plane. Such a transformation is called a *map projection*. The term projection derives from the geometric methods that were traditionally used to construct maps, in the fashion of optical projections made with a device called *camera obscura* that Renaissance artists relied on to render three-dimensional perspective views on paper and canvas.

While many map projections no longer rely on physical projections, it is useful to think of map projections in geometric terms. This is because map projection consists of constructing points on geometric objects such as cylinders, cones, and circles that correspond to homologous points on the surface of the planet being mapped according to certain rules and formulas.

The following sections describe the basic properties of map projections, the surfaces onto which projections are developed, the types of parameters associated with different classes of projections, how projected data can be mapped back to the sphere or spheroid it represents, and details about one very widely used projection system, called Universal Transverse Mercator.

---

**Note** Most map projections in the toolbox are implemented as M-functions; however, these are only used by certain calling functions (such as `geoshow` and `axesm`), and thus have no documented public API.

---

For more detailed information on specific projections, browse the Chapter 12, “Map Projections — By Category” (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 8-64.

## The Three Main Families of Map Projections

### In this section...

“Unwrapping the Sphere to a Plane” on page 8-6

“Cylindrical Projections” on page 8-6

“Conic Projections” on page 8-8

“Azimuthal Projections” on page 8-9

### Unwrapping the Sphere to a Plane

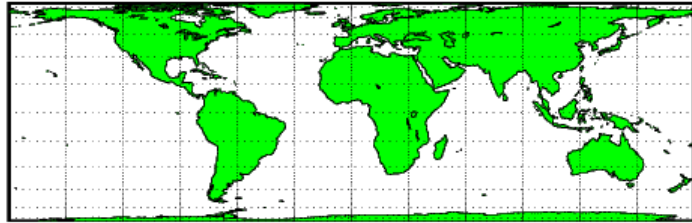
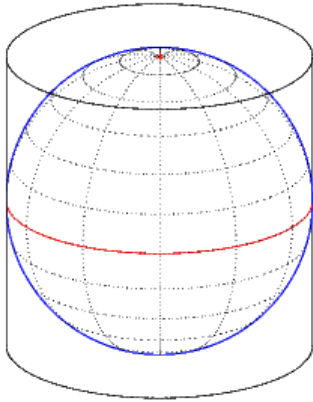
Mapmakers have developed hundreds of map projections, over several thousand years. Three large families of map projection, plus several smaller ones, are generally acknowledged. These are based on the types of geometric shapes that are used to transfer features from a sphere or spheroid to a plane. As 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 8-11.

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

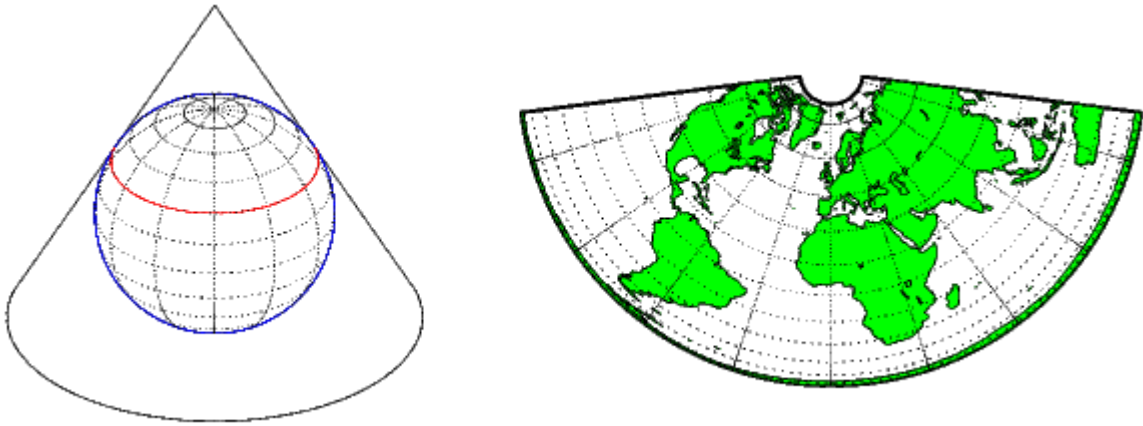
straight lines. Pseudocylindrical projections can be equal-area, but are not conformal or equidistant.

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).

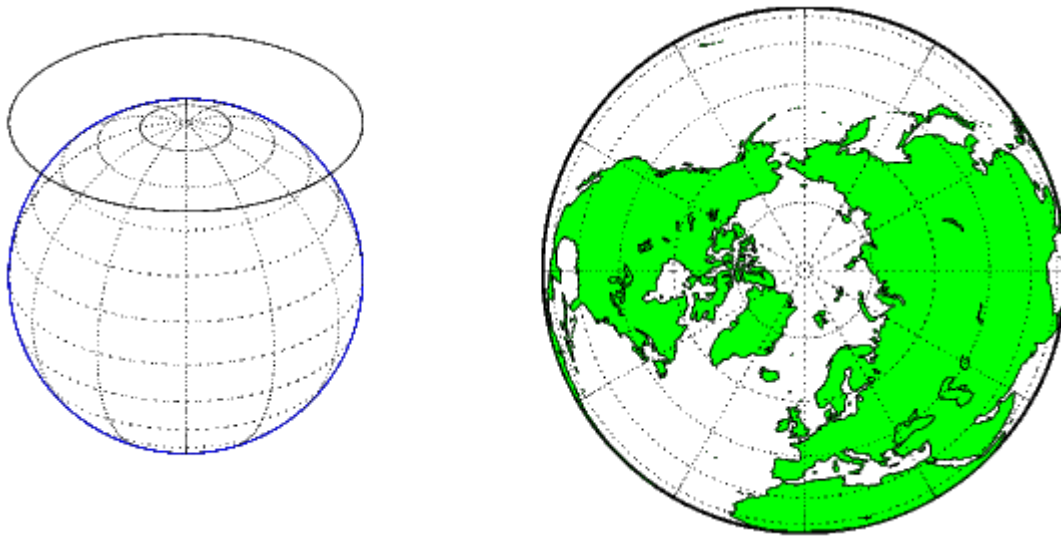


Some widely-used conic projections are

- 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 12, “Map Projections — By Category” (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.

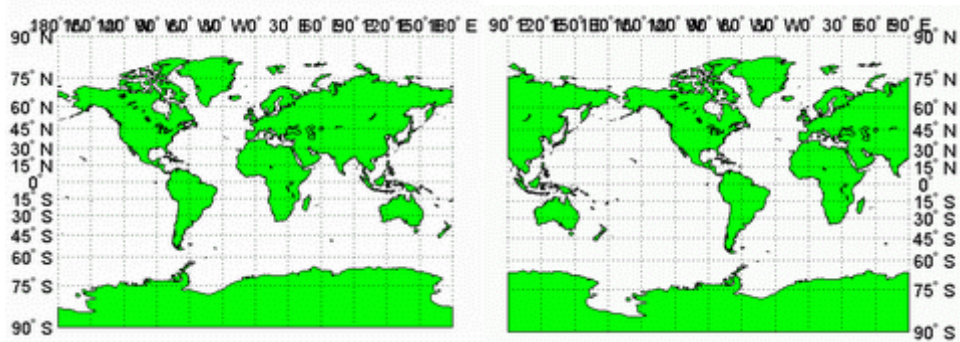
### The Orientation Vector

A map axes Origin property is a vector describing the geometry of the displayed projection. Mapping Toolbox calls this property the *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°,0°) 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°.

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



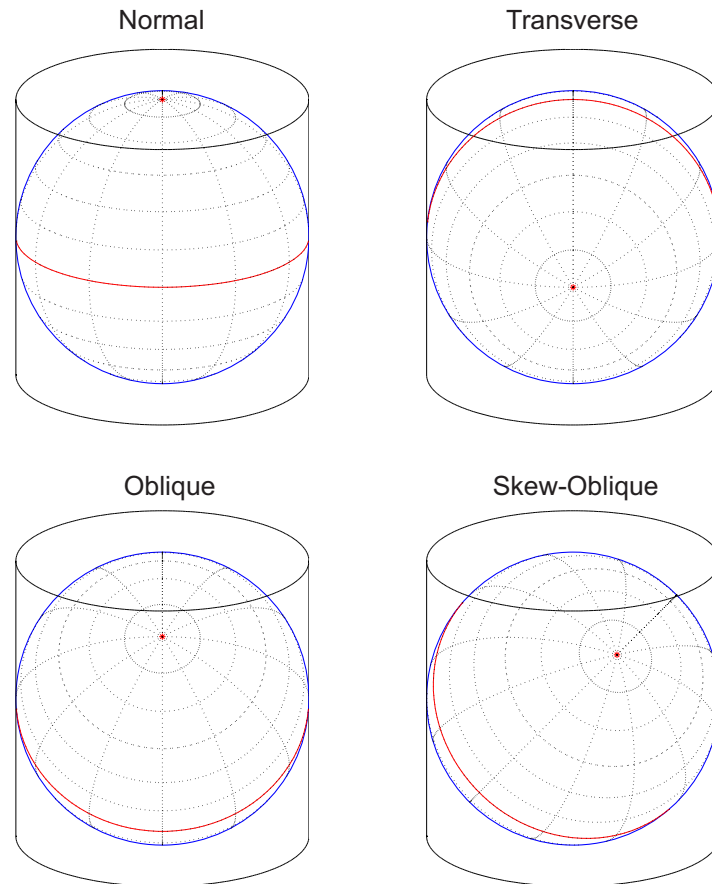
Origin at  $(0^\circ, 0^\circ)$  with a  $0^\circ$  orientation  
(orientation vector =  $[0 \ 0 \ 0]$ )

Origin at  $(0^\circ, 90^\circ\text{W})$  with a  $0^\circ$  orientation  
(orientation vector =  $[0 \ -90 \ 0]$ )

This makes sense if you think about a simple, true cylindrical projection. This is the projection of the globe onto a cylinder wrapped around it. For normal aspects, this cylinder is tangent to the globe at the Equator, and changing the origin longitude simply corresponds to rotating the sphere about the longitudinal axis of the cylinder. If you continue with the wrapped-cylinder model, you can understand the other aspects as well.

Following this description, a *transverse* projection can be thought of as a cylinder wrapped around the globe tangent at the poles and along a meridian and its antipodal meridian. Finally, when such a cylinder is tangent along any great circle other than a meridian, the result is an *oblique* projection.

Here are diagrams of the four cylindrical map orientations, or aspects:



Of course, few projections are true cylindrical projections, but the concept of the wrapped cylinder is nonetheless a convenient way to describe aspect.

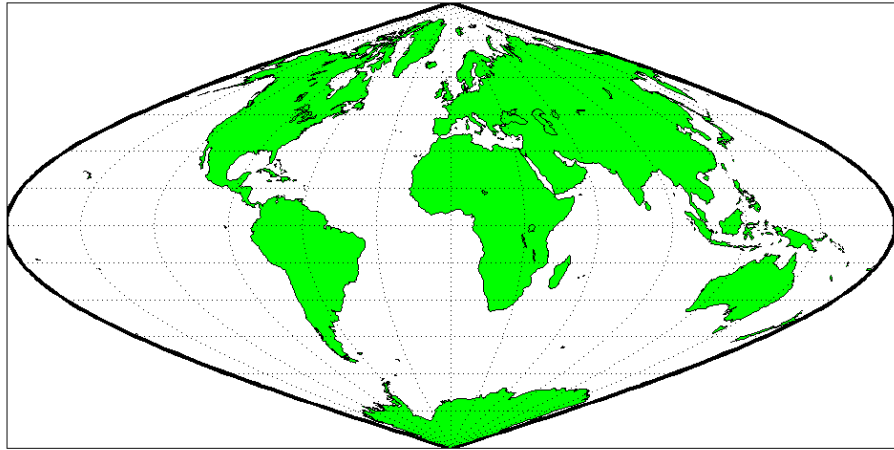
### Exploring Projection Aspect

Perhaps the best way to gain an understanding of projection aspect is to experiment with orientation vectors. For the following exercise, use a pseudocylindrical projection, the sinusoidal.

- 1 Create a default map axes in a sinusoidal projection, turn on the graticule, and display the coast data set as filled polygons:

```
figure;  
axesm sinusoid  
framem on; gridm on; tightmap tight  
load coast  
patchm(lat, long, 'g')
```

The continents and graticule appear in normal aspect, as shown below.



Normal aspect: origin at (0°,0°), orientation 0°  
(orientation vector = [0 0 0])

**2** Inspect the orientation vector from the map axes:

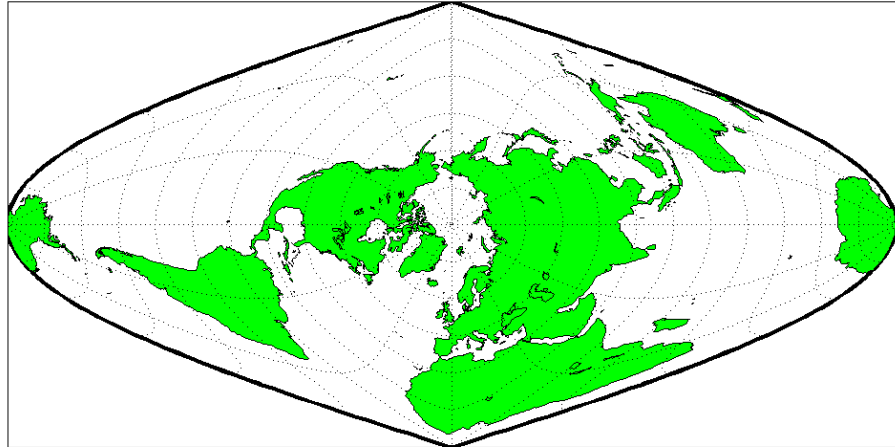
```
getm(gca, 'Origin')  
  
ans =  
    0    0    0
```

By default, the origin is set at (0°E, 0°N), oriented 0° from vertical.

**3** In the normal aspect, the North Pole is at the *top* of the image. To create a transverse aspect, imagine pulling the North Pole down to the center of the display, which was originally occupied by the point (0°,0°). Do this by setting the first element of `Origin` parameter to a latitude of 90°N:

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

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

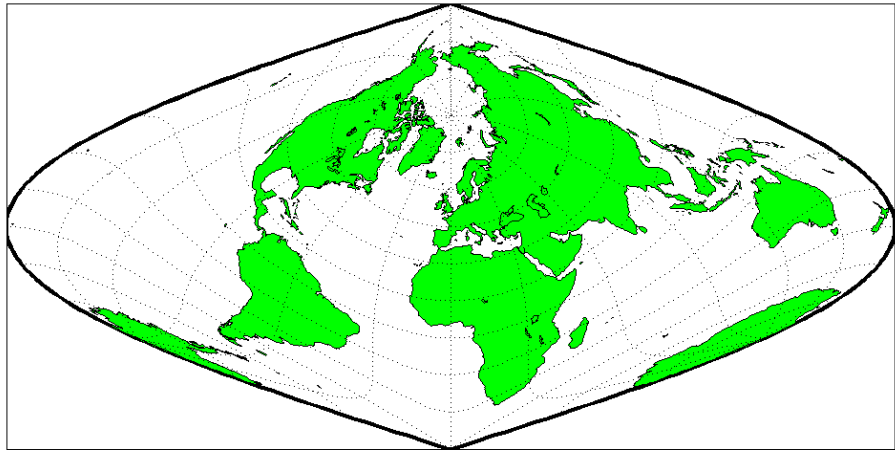


**Transverse aspect: origin at (90°N,0°), orientation 0°  
(orientation vector = [90 0 0])**

- 4** The normal and transverse aspects can be thought of as limiting conditions. Anything else is an oblique aspect. Conceptually, if you push the North Pole halfway back to its original position (to the position originally occupied by the point (45°N, 0°E) in the normal aspect), the result is a simple oblique aspect.

```
setm(gca, 'Origin', [45 0 0])
```

The oblique sinusoidal projection centered at (45°N, 0°E) is shown below.



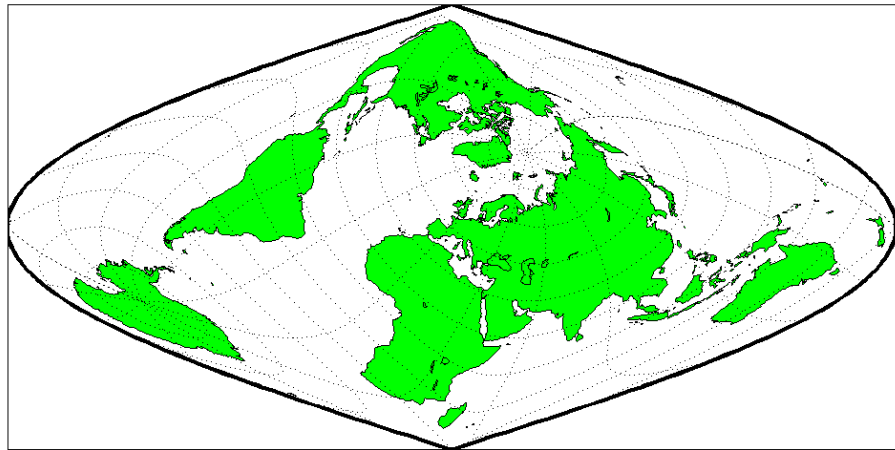
Oblique aspect: origin at (45°N, 0°), orientation 0°  
(orientation vector = [45 0 0])

You can think of this as pulling the new origin (45°N, 0°) to the center of the image, the place that (0°, 0°) occupied in the normal aspect.

- 5 The previous examples of projection aspect kept the aspect orientation at 0°. If the orientation is altered, an oblique aspect becomes a *skew-oblique*. Imagine the previous example with an orientation of 45°. Think of this as pulling the new origin (45°N, 0°E), down to the center of the projection and then rotating the projection until the North Pole lies at an angle of 45° clockwise from straight up with respect to the new origin.

```
setm(gca, 'Origin', [45 0 45])
```

As in the previous example, the location (45°N, 0°E) 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 90°, resulting in a *portrait* orientation. To display a map in the transverse aspect, combine the transverse aspect property with a -90° 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 38° 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.

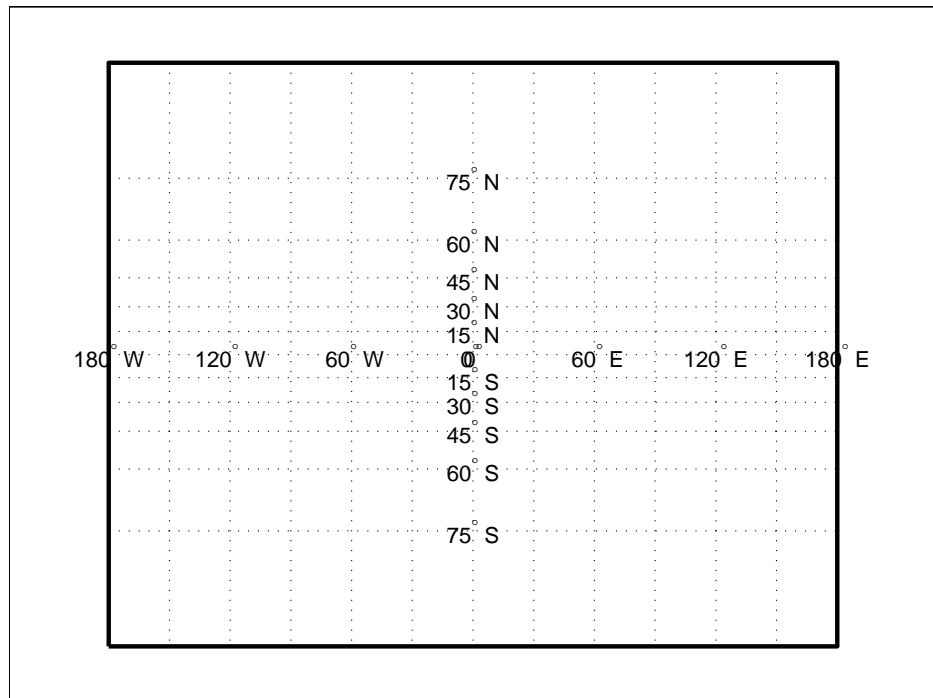
### Determining Projection Parameters

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 Mapping 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, 'falseeastng')  
  
ans =  
    0  
  
getm(h, 'zone')  
  
ans =  
    []  
  
getm(h, 'maplatlimit')  
  
ans =  
   -86    86  
  
getm(h, 'maplonlimit')
```

```
ans =
    -135    135

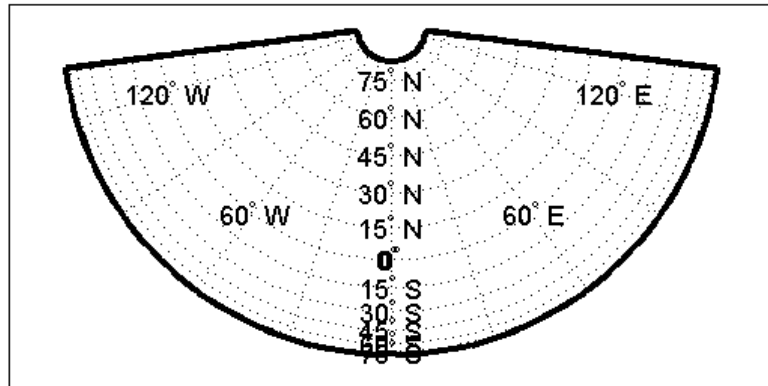
getm(h, 'Flatlimit')

ans =
    -86     86

getm(h, 'Flonlimit')

ans =
    -135    135
```

The equiconic projection has two standard parallels, at 15° and 75°. It also has reduced longitude limits (covering 270° rather than 360°). The resulting equiconic 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')
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,'falseeastings')  
  
ans =  
    0  
  
getm(h,'zone')  
  
ans =  
    []  
  
getm(h,'maplatlimit')  
  
ans =  
   -86    86
```

```
getm(h,'maplonlimit')
```

```
ans =  
-135 135
```

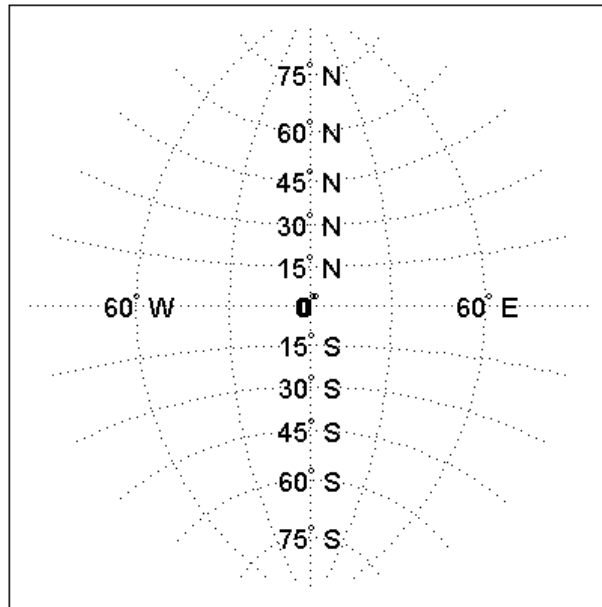
```
getm(h, 'Flatlimit')
```

```
ans =  
-86 86
```

```
getm(h, 'Flonlimit')
```

```
ans =  
-135 135
```

The 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 12, “Map Projections — By Category” (available online and in the PDF version of this document) lists and illustrates all map projections supported by Mapping Toolbox, including suggestions for parameter usage.

## Visualizing and Quantifying Projection Distortions

### In this section...

“Displays of Spatial Error in Maps” on page 8-28

“Quantifying Map Distortions at Point Locations” on page 8-32

### 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. Mapping Toolbox provides several functions that map projection distortions, and one that 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 `clm tissot`:

- 1 Set up a Sinusoidal projection in a skewed aspect, plotting the graticule:

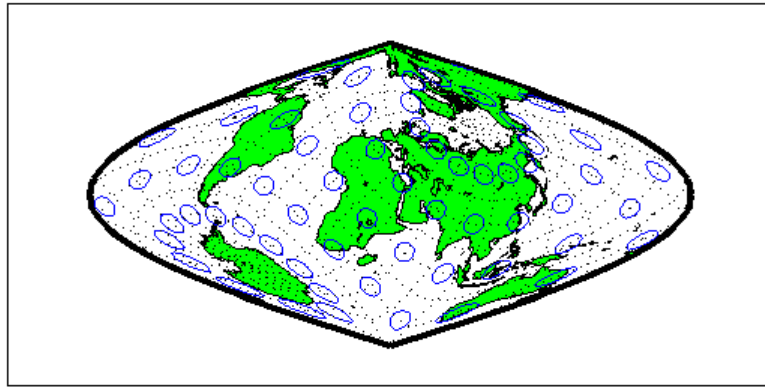
```
figure;  
axesm sinusoid  
gridm on;framem on;  
setm(gca,'Origin', [20 30 45])
```

- 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
```



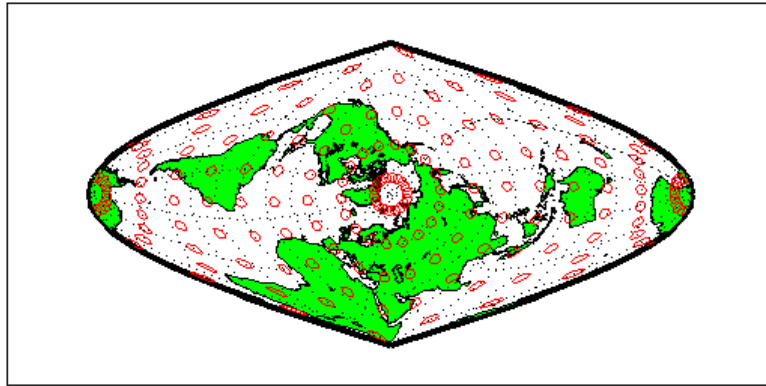
Notice that the circles vary considerably in shape. This indicates that the Sinusoidal projection is not conformal. Despite the distortions, however, the circles all cover equal amounts of area on the map, because the projection has the equal-area property.

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 referencing vector (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, Mapping Toolbox enables you to 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:

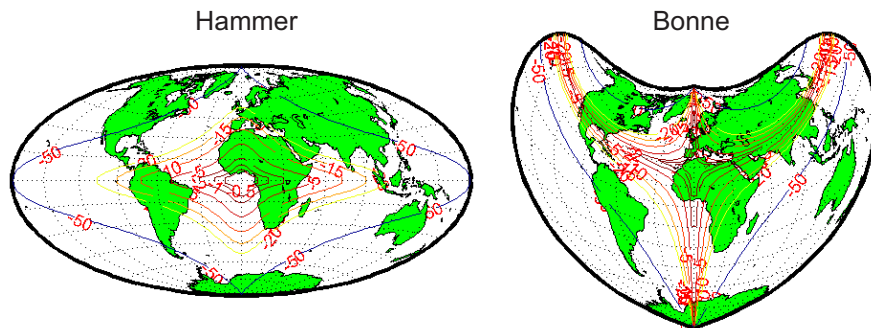
```
mdistort('scaleratio')
```

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.

Mapping Toolbox provides a function to return location-specific map error statistics from the current projection or an `mstruct`. The `distortcalc` function computes the same distortion statistics as `mdistort` does, but for specified locations provided as arguments. You provide the latitude-longitude locations one at a time or in vectors. The general form is

```
[areascale,angdef,maxscale,minscale,merscale,parscale] = ...  
    distortcalc(mstruct,lat,long)
```

However, if you are evaluating the current map figure, omit the `mstruct`. You need not specify any return values following the last one of interest to you.

### Using `distortcalc` to Determine Map Projection Geometric Distortions

The following exercise uses `distortcalc` to compute the maximum area distortion for a map of Argentina from the `landareas` data set.

- 1 Read the North and South America polygon:

```
Americas = shaperead('landareas', 'UseGeoCoords', true, ...  
    'Selector', {@(name) ...  
    strcmpi(name,{'north and south america'})}, 'Name');
```

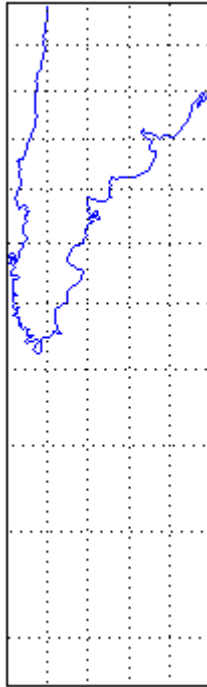
- 2** Set the spatial extent (map limits) to contain the southern part of South America and also include an area closer to the South Pole:

```
mlatlim = [-72.0 -20.0];  
mlonlim = [-75.0 -50.0];  
[alat, alon] = maptriml([Americas.Lat], ...  
    [Americas.Lon], mlatlim, mlonlim);
```

- 3** Create a Mercator cylindrical conformal projection using these limits, specify a five-degree graticule, and then plot the outline for reference:

```
figure;  
axesm('MapProjection','mercator', 'grid','on', ...  
    'MapLatLimit',mlatlim, 'MapLonLimit',mlonlim,...  
    'MLineLocation',5, 'PLineLocation',5)  
plotm(alat,alon,'b')
```

The map looks like this:



- 4** Sample every tenth point of the patch outline for analysis:

```
alats = alat(1:10:numel(alat));  
alons = alon(1:10:numel(alat));
```

- 5** Compute the area distortions (the first value returned by `distortcalc`) at the sample points:

```
adistort = distortcalc(alats, alons);
```

- 6** Find the range of area distortion across Argentina (percent of a unit area on, in this case, the equator):

```
adistortmm = [min(adistort) max(adistort)]  
  
adistortmm =  
    1.1790    2.7716
```

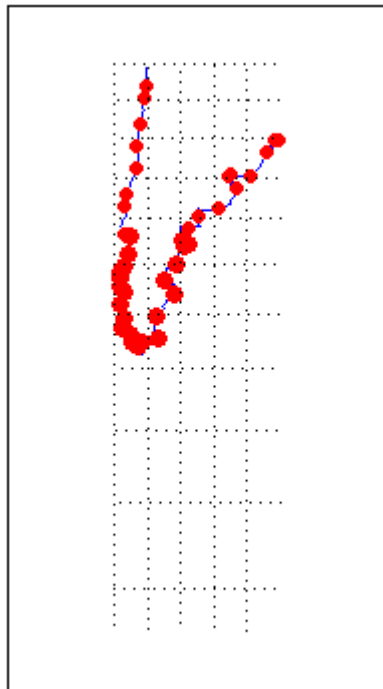


As Argentina occupies mid southern latitudes, its area on a Mercator map is overstated, and the errors vary noticeably from north to south.

- 7** Remove any NaNs from the coordinate arrays and plot symbols to represent the relative distortions as proportional circles, using scatterm:

```
nanIndex = isnan(adistort);
alats(nanIndex) = [];
alons(nanIndex) = [];
adistort(nanIndex) = [];
scatterm(alats,alons,20*adistort,'red','filled')
```

The resulting map is shown below:



- 8** The degree of area overstatement would be considerably larger if it extended farther toward the pole. To see how much larger, get the area distortion for 50°S, 60°S, and 70°S:

```
a=distortcalc(-50,-60)

a =
    2.4203

a=distortcalc(-60,-60)

a =
    4

>> a=distortcalc(-70,-60)

a =
    8.5485
```

---

**Note** You can only use `distortcalc` to query locations that are within the current map frame or `mstruct` limits. Outside points yield NaN as a result.

---

- 9 Using this technique, you can write a simple script that lets you query a map repeatedly to determine distortion at any desired location. You can select locations with the graphic cursor using `inputm`. For example,

```
[plat plon] = inputm(1)

plat =
   -62.225
plon =
   -72.301
>> a=distortcalc(plat,plon)

a =
    4.6048
```

Naturally the answer you get will vary depending on what point you pick. Using this technique, you can write a simple script that lets you query a map repeatedly to determine any distortion statistic at any desired location.

Try changing the map projection or even the orientation vector to see how the choice of projection affects map distortion. For further information, see the reference page for `distortcalc`.

## Accessing, Computing, and Inverting Map Projection Data

### In this section...

“Accessing Projected Coordinate Data” on page 8-38

“Projecting Coordinates Without a Map Axes” on page 8-40

“Inverse Map Projection” on page 8-42

“Coordinate Transformations” on page 8-46

### 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. For further information on specifying coordinate units and ellipsoids, see .

## 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 Geographic Data 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 the structure for projection computations without involving a map axes or a graphical display.

- 1 Begin by starting afresh with the coast data set:

```
figure;
load coast
```

- 2 Use `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.

- 3 Just as you can change the property settings of a map axes with `setm`, you can assign values to the entries of the map projection structure to control the projection properties. Change the map orientation to define a transverse aspect, and set the ellipsoid and coordinate units:

```
mstruct.origin = [-90 180 0];
mstruct.geoid = almanac('earth','grs80','kilometers');
```

- 4 Repopulate the rest of the structure fields with default property values.

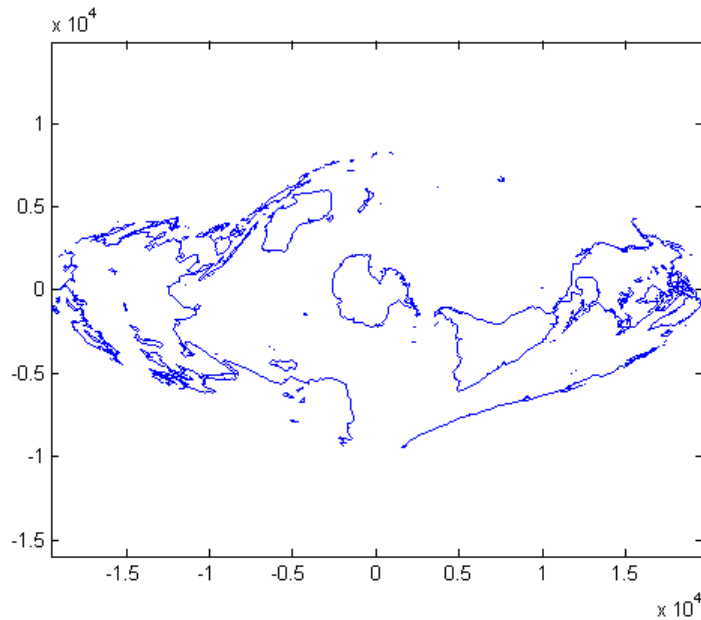
```
mstruct = defaultm(sinusoid(mstruct));
```

You must invoke `defaultm` a second time (recursively) to ensure that any side effects of properties you change are properly handled. For example, changing the origin can constrict the map limits on some projections.

- 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] = mfdtran(mstruct,lat,long,[],'line');
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 8-42, 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 transforms plane coordinates into geodetic coordinates with the `minvtran` function, a mirror image of `mfwdtran`, which is described in “Using `mfwdtran` with a Geographic Data 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 outside Mapping Toolbox, 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 `mfwtran` with a Geographic Data 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

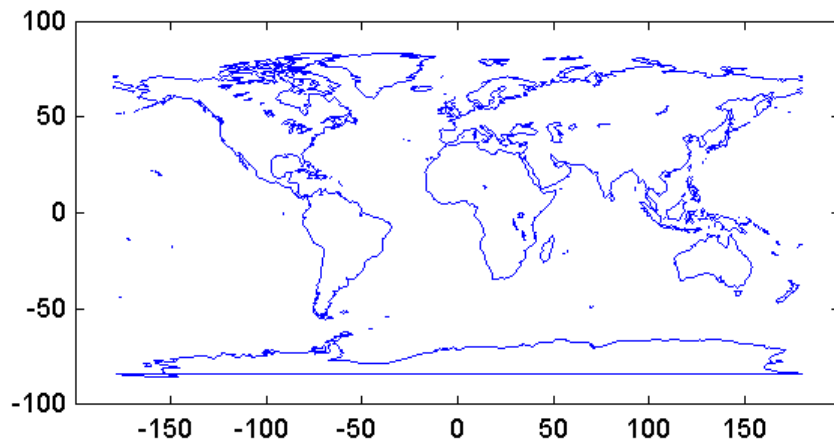
The difference in size between `lat` and `long` and `x` and `y` are due to clipping the `x-y` data to the map frame (NaNs are inserted at clip locations).

- 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  $360^\circ$  are geographically equivalent. The data from the inverse transformation process therefore jumps from  $180^\circ$  to  $-180^\circ$ , 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 `vfdtran` function, which takes latitudes, longitudes, and azimuths, computes angles that geographic vectors make on the projection plane.

To illustrate, define vectors pointing north ( $0^\circ$ ) and east ( $90^\circ$ ) at three locations and use `vfdtran` 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','maplatlim',[-10 45],'maplonlim',[-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 = vfdtran(lats,lons,northazs)

ans =
    59.614         90    120.39
```

North varies from about  $60^\circ$  from the  $x$ -axis, to vertical, to  $120^\circ$  from the  $x$ -axis, quite symmetrically.

**5** Compute projected direction of east for each location:

```
peast = vfdtran(lats,lons,eastazs)

ans =
   -30.385    0.0001931    30.386

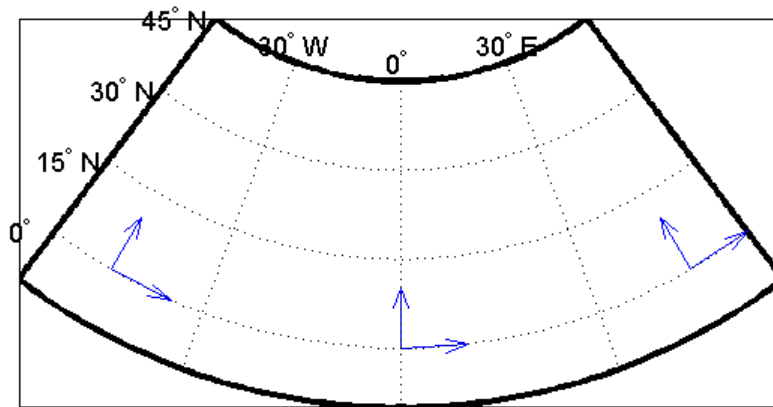
pnorth - peast

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 `vfdtran` and `quiverm`.

## Coordinate Transformations

In “The Orientation Vector” on page 8-11, 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  $90^\circ$  minus its transformed latitude (also known as a *colatitude*). The point antipodal to your town would become the *South Pole*, at  $-90^\circ$ . Its distance from your hometown is  $90^\circ - (-90^\circ)$ , or  $180^\circ$ , as expected. Points  $90^\circ$  distant from your hometown all have a transformed latitude of  $0^\circ$ , 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  $(32^\circ\text{N}, 102^\circ\text{W})$ . You have a brother in Tulsa  $(36.2^\circ\text{N}, 96^\circ\text{W})$  and a sister in New Orleans  $(30^\circ\text{N}, 90^\circ\text{W})$ .

- 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

[dist2newor1 az2newor1] = distance(midl_lat,midl_lon,...
                                   newo_lat,newo_lon)

dist2newor1 =
10.4727

az2newor1 =
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-az2newor1)

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 (58°N, 78°E). Midland is now at a *new latitude* of 90°.

- 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  $49.7258^\circ$ . 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 (7°N, 80°E) 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:

```
[map,lat,lon] = neworig(topo,topolegend,origin);
```

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

- 3 Display the new map:

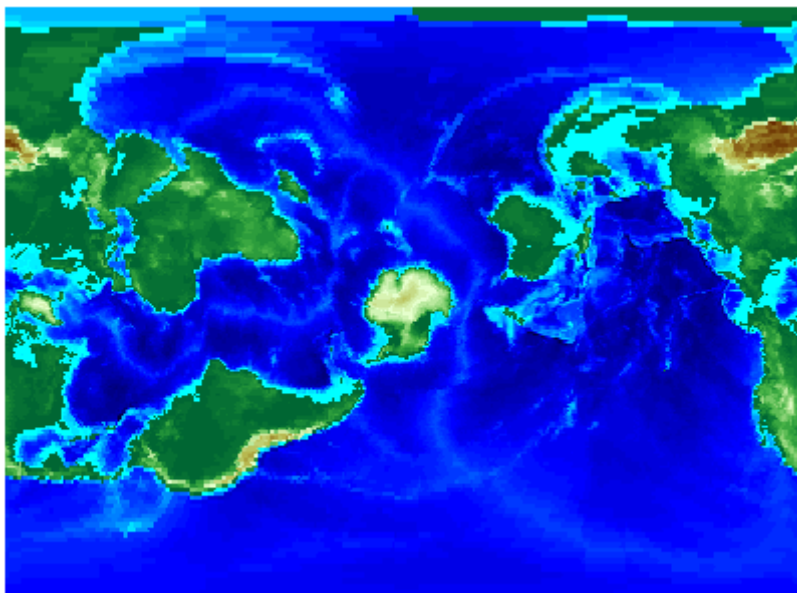
```
axesm miller
surfm(map,[30 30]); demcmap(topo)
```

- 4 This map is displayed in normal aspect, as its orientation vector shows:

```
mapprops = getm(gca);
mapprops.origin

ans =
     0     0     0
```





An interesting feature of this new grid is that every cell in its first row is  $0^{\circ}$ - $1^{\circ}$  distant from the point ( $7^{\circ}$ N, $80^{\circ}$ E), and every cell in its second row is  $1^{\circ}$ - $2^{\circ}$  distant, etc. Another feature is that every cell in a particular column has the same great circle azimuth from the point.

## Working with the UTM System

### In this section...

“What Is the Universal Transverse Mercator System?” on page 8-52

“Understanding UTM Parameters” on page 8-53

“Setting UTM Parameters with a GUI” on page 8-55

“Working in UTM Without a Map Axes” on page 8-60

“Mapping Across UTM Zones” on page 8-61

### 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 array 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
falseeastng: 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]
...
```

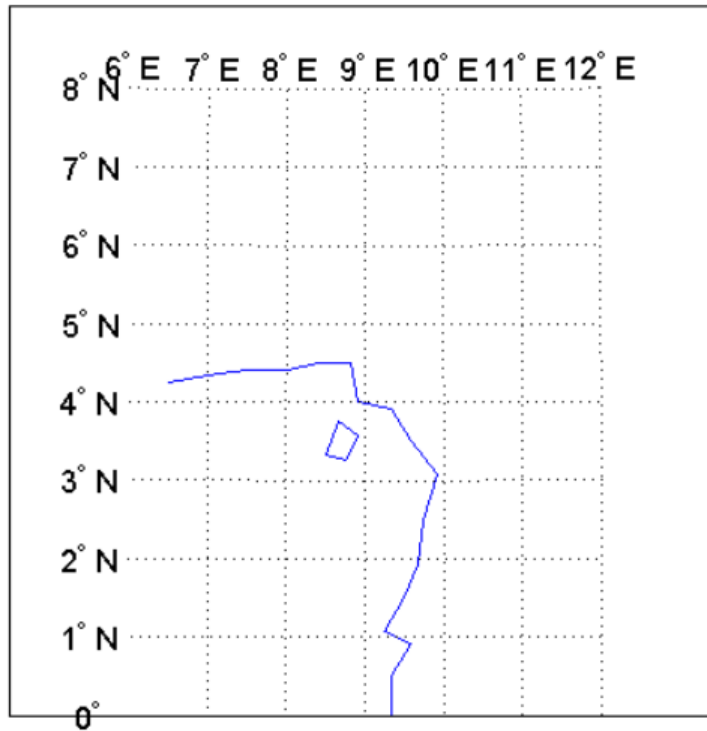
Note that the map origin and limits are adjusted for zone 32N.

**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 from 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
```

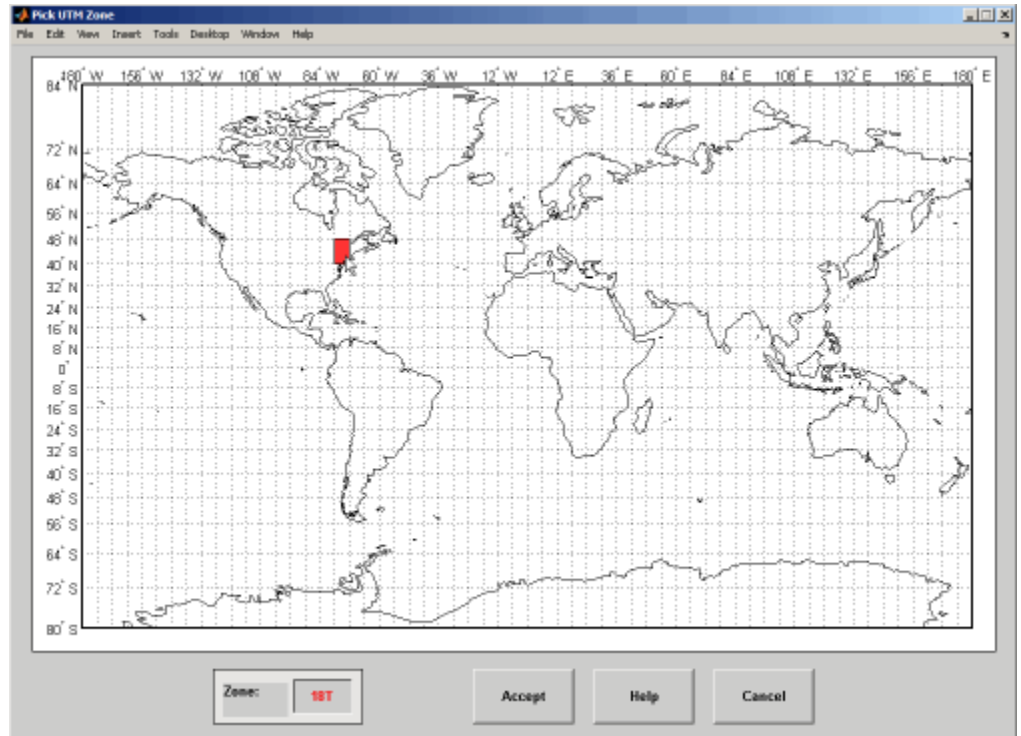
The **Map Projection** field is set to cylIn: Universal Transverse Mercator (UTM).

---

**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:

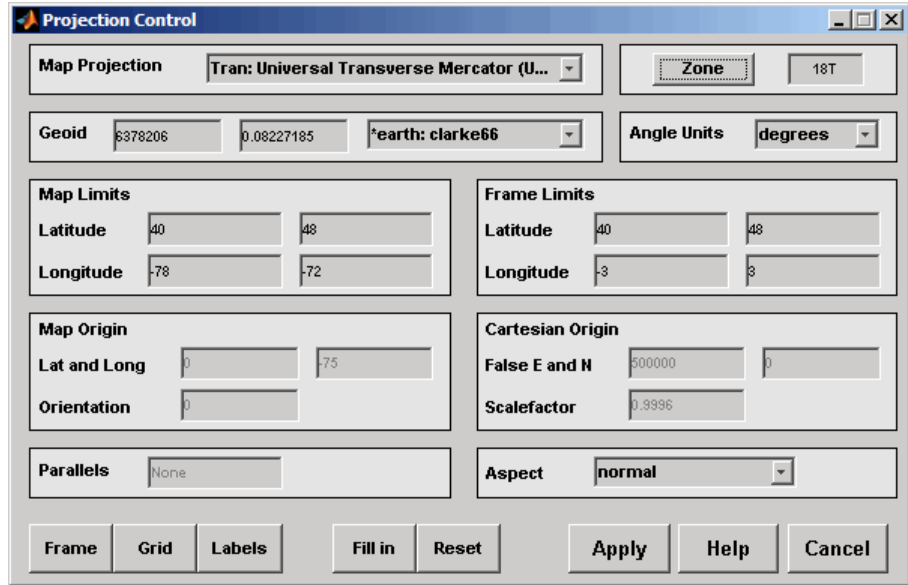


Note that while you can open the `utmzoneui` control panel from the command line, you then have to manually update the figure with the zone name it returns with a `setm` command:

```
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.



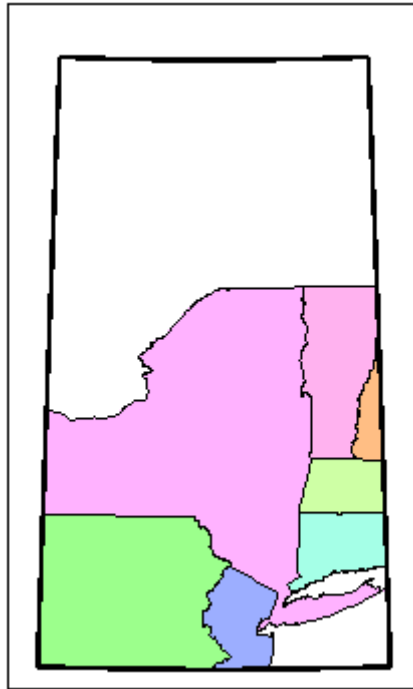
- 4 Click **Apply** to close the projection control panel.

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 projection based on this information:

```
utmstruct = defaultm('utm');  
utmstruct.zone = '18T';  
utmstruct.geoid = ellipsoid;  
utmstruct.flatlimit = [];  
utmstruct.maplatlimit = [];  
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:

```
[zonalats zonalons] = utmzone(utmzone(40.7, -74.0))

zonalats =
    40    48
zonalons =
   -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.

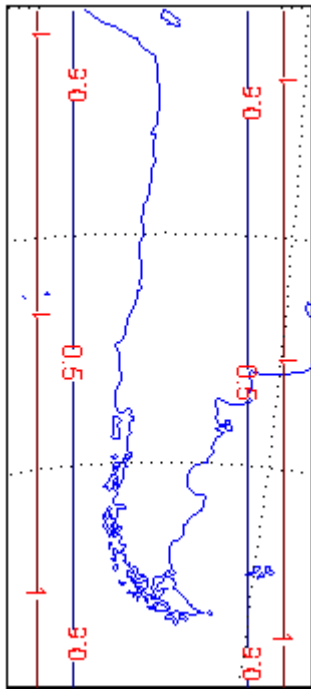
To display areas that extend across more than one UTM zone, it might be appropriate to use the Mercator projection in a transverse aspect. Of course, you do not obtain coordinates in meters that would match those of a UTM projection, but the results will be nearly as accurate. Here is an example of a 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.

---



## 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.

Mapping Toolbox provides about 60 different 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 map projections provided by Mapping Toolbox can be found in Chapter 12, “Map Projections — By Category” (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	•			

<b>Projection</b>	<b>Syntax</b>	<b>Type</b>	<b>Equal--Area</b>	<b>Con-formal</b>	<b>Equi-distant</b>	<b>Special Features</b>
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	•			

<b>Projection</b>	<b>Syntax</b>	<b>Type</b>	<b>Equal--Area</b>	<b>Con-formal</b>	<b>Equi-distant</b>	<b>Special Features</b>
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	flatplr	Pseudo-cylindrical	•			
McBryde-Thomas Flat-Polar Quartic	flatplr <sub>q</sub>	Pseudo-cylindrical	•			
McBryde-Thomas Flat-Polar Sinusoidal	flatplr <sub>s</sub>	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				



<b>Projection</b>	<b>Syntax</b>	<b>Type</b>	<b>Equal--Area</b>	<b>Con-formal</b>	<b>Equi-distant</b>	<b>Special Features</b>
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	wichel	Pseudo-azimuthal	•			
Aitoff	aitoff	Modified Azimuthal				

<b>Projection</b>	<b>Syntax</b>	<b>Type</b>	<b>Equal--Area</b>	<b>Con-formal</b>	<b>Equi-distant</b>	<b>Special Features</b>
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).

# 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 (p. 9-2)

Basic spatial statistics for the sphere and plane

Navigation (p. 9-11)

Functions for fixing, route planning, navigating, and reckoning

## Geographic Statistics

In this section...
“Statistics for Point Locations on a Sphere” on page 9-2
“Geographic Means” on page 9-2
“Geographic Standard Deviation” on page 9-4
“Equal-Areas in Geographic Statistics” on page 9-7

### Statistics for Point Locations on a Sphere

Mapping Toolbox provides functions for computing basic geographical measures for spatial analysis and for filtering and conditioning data. Since MATLAB itself can compute statistics like means, medians, and variances, why not use those functions in Mapping 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 provides functions for appropriately computing 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, 180° apart in longitude, for example (30°N,90°W) and (30°N,90°E). 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 90°W as 270°E,  $(90+270)/2=180$  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 30°N 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 (30°N,0°) 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 90°N, 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  $(0^\circ, 0^\circ)$ ,  $(0^\circ, 120^\circ)$ , and  $(0^\circ, 240^\circ)$ , 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)
```

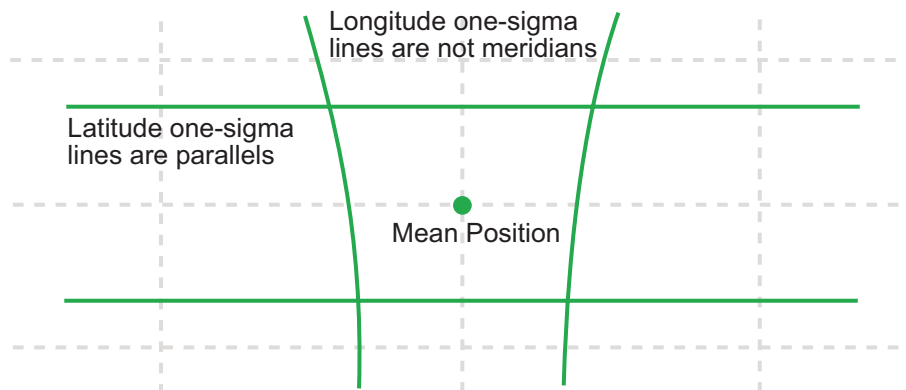
The function returns two deviations, one for latitudes and one for longitudes.

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  $60^\circ$  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

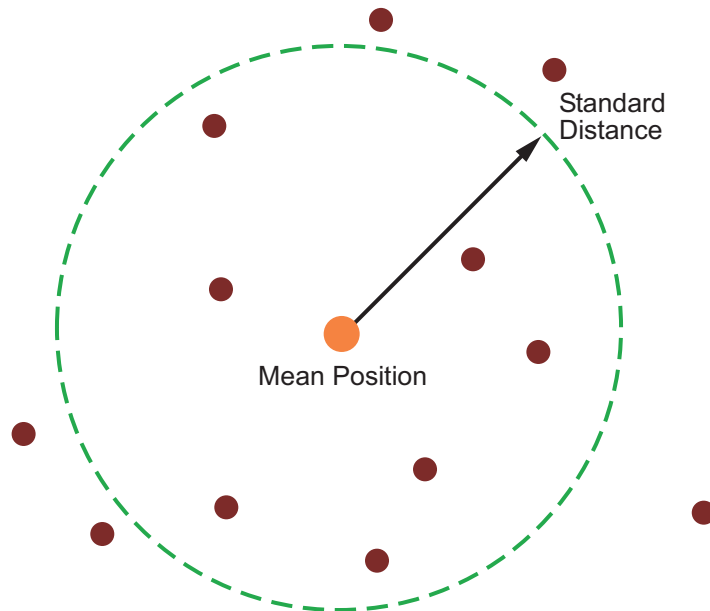
measure of deviation. The `stdm` handling is useful for many applications, especially when the data is not global. For instance, these potential difficulties would not be a danger for data points confined to the country of Mexico.

### 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

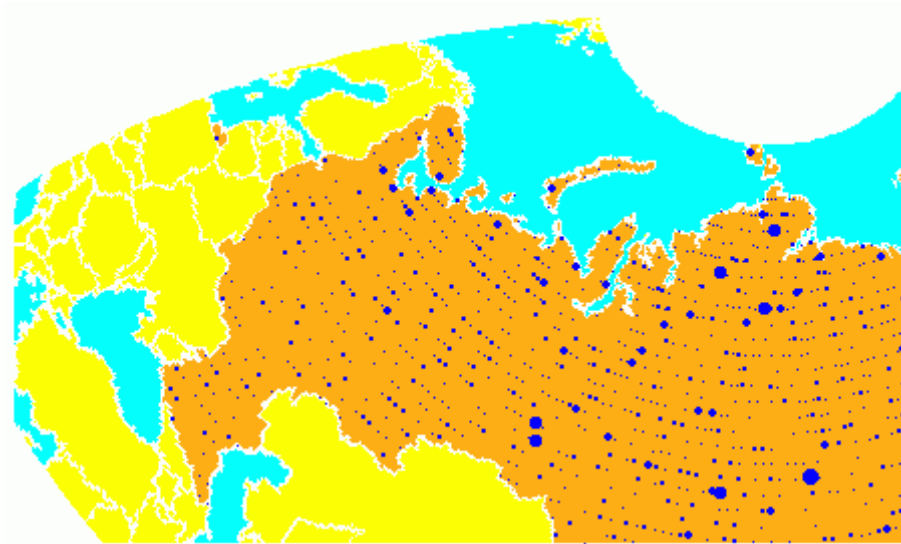
A common error in applying two-dimensional statistics to geographic data lies in ignoring equal-area treatment. It is often necessary to *bin* data to statistically analyze it. In a Cartesian plane, this is easily done by dividing the space into equal  $x$ - $y$  squares. The geographic equivalent of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

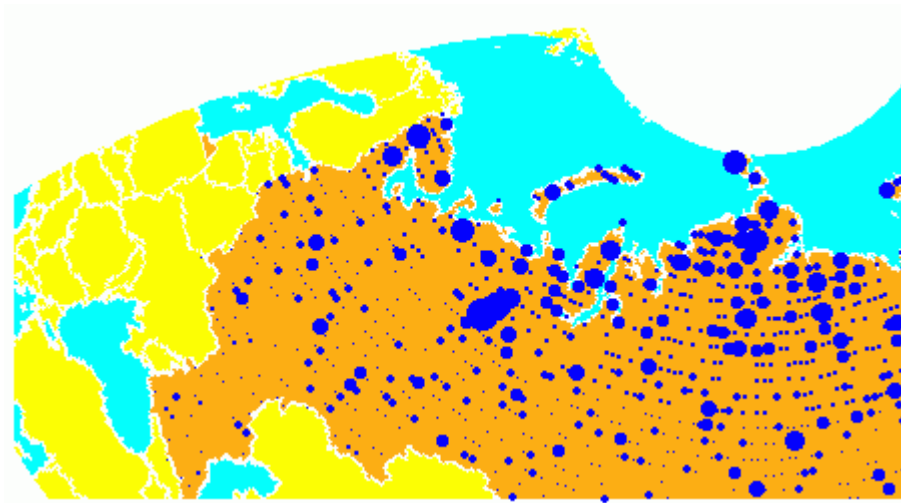


This is a portion of the whole map, displayed in an equal-area Bonne projection. The first step in creating data displays without area bias is to choose an equal-area projection. The proportionally sized symbols are a result of the specialized display function `scatterm`.

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:

$$[x,y] = \text{grn2eqa}(\text{lat}, \text{lon})$$

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 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?” on page 9-11

“Conventions for Navigational Functions” on page 9-12

“Fixing Position” on page 9-13

“Planning” on page 9-25

“Track Laydown – Displaying Navigational Tracks” on page 9-29

“Dead Reckoning” on page 9-31

“Drift Correction” on page 9-36

“Time Zones” on page 9-38

## 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. Mapping 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 contains functions to support these navigational activities as well.

## Conventions for Navigational Functions

### Units

Mapping Toolbox is, in general, very flexible in allowing a variety of angular and distance measurement units. The navigational support functions are

- `dreckon`
- `gcwaypts`
- `legs`
- `navfix`

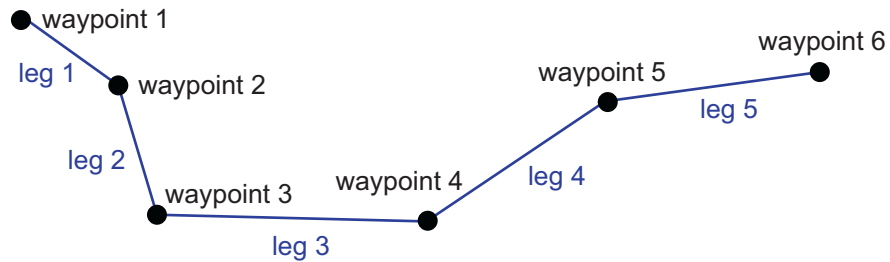
To make these functions easy to use, and to conform to common navigational practice, *for these specific functions only*, certain conventions are used:

- 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. In Mapping Toolbox navigation functions, angle units are always 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.

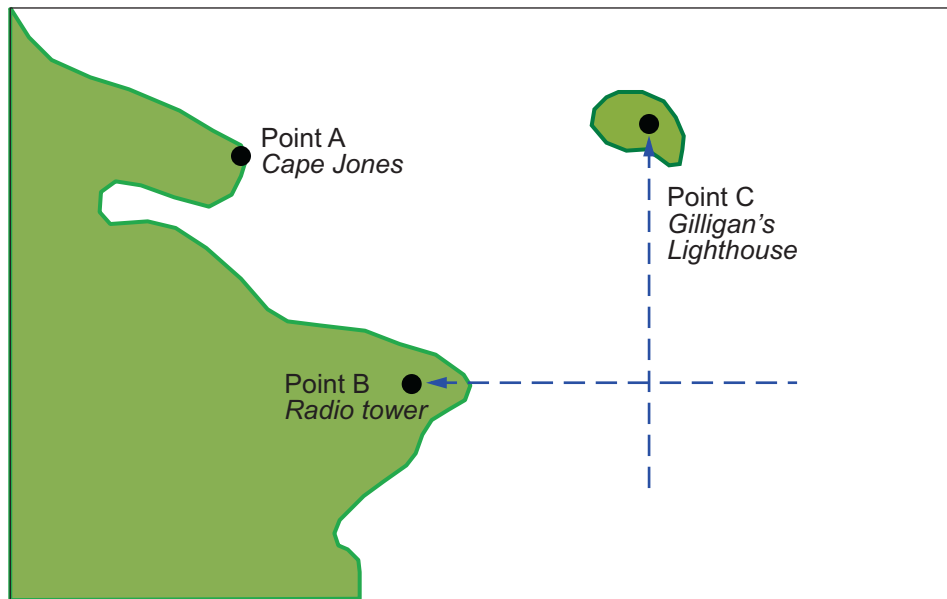
The Mercator was designed exactly for this purpose. Range circles, which might be determined with a radar, are assumed to plot as true circles on a Mercator chart. This allows the navigator to manually draw the range arc with a compass.

These assumptions also lead to computationally efficient methods for fixing positions with a computer. Mapping 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.

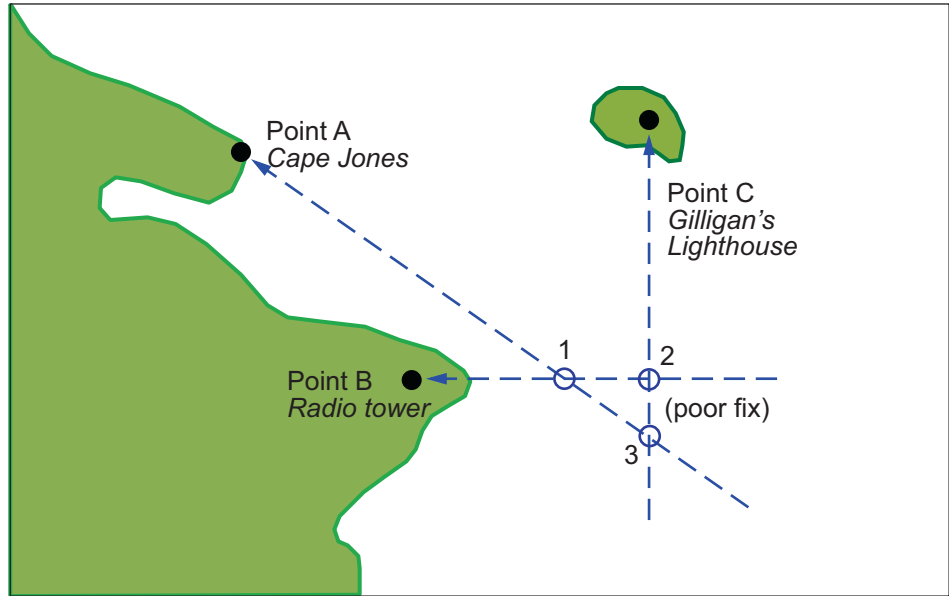
### Some Possible Situations

In this imaginary coastal region, you take a visual bearing on the radio tower of  $270^\circ$ . At the same time, Gilligan's Lighthouse bears  $0^\circ$ . If you plot a  $90^\circ$ - $270^\circ$  line through the radio tower and a  $0^\circ$ - $180^\circ$  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  $300^\circ$ . If that line exactly crosses the point of intersection of the first two lines, you will have a perfect fix.





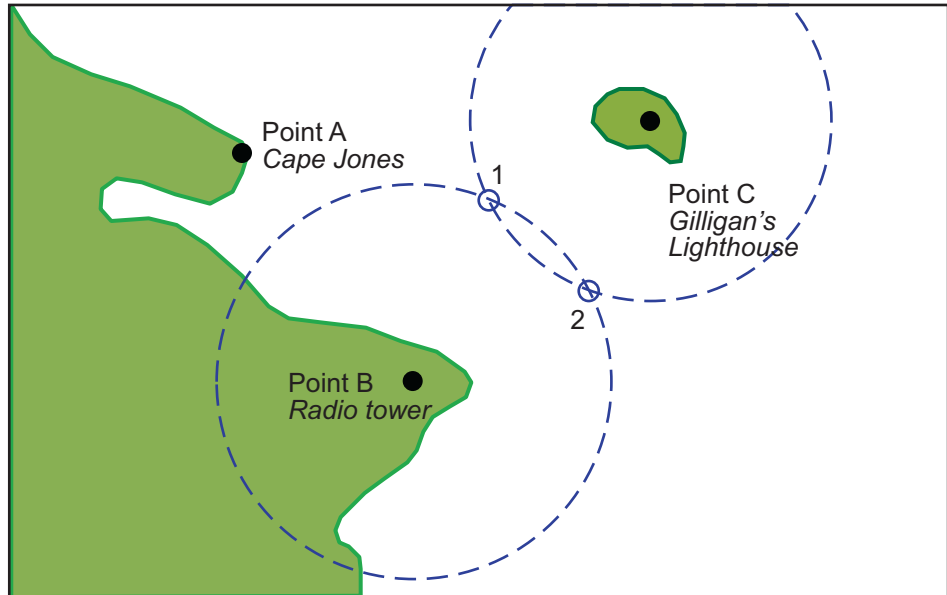
Whoops. What happened? Is your lookout in error? Possibly, but perhaps one or both of your bearings was slightly in error. This happens all the time. Which point, 1, 2, or 3, is correct? As far as you know, they are all equally valid.

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.

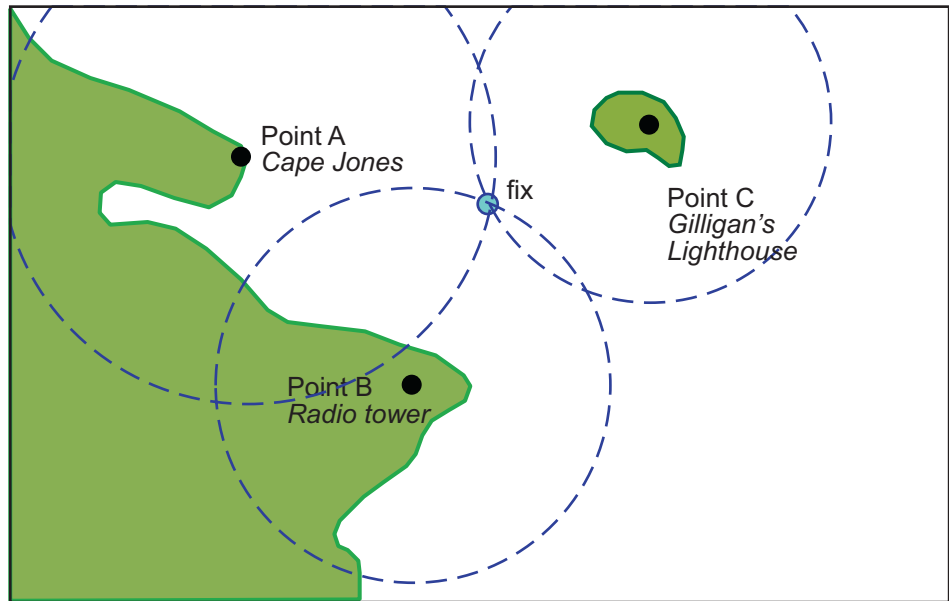
The next time you traverse these straits, it is a very foggy morning. You can't see any landmarks, but luckily, your navigational radar is operating. Each of these landmarks has a good radar signature, so you're not worried. You

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.

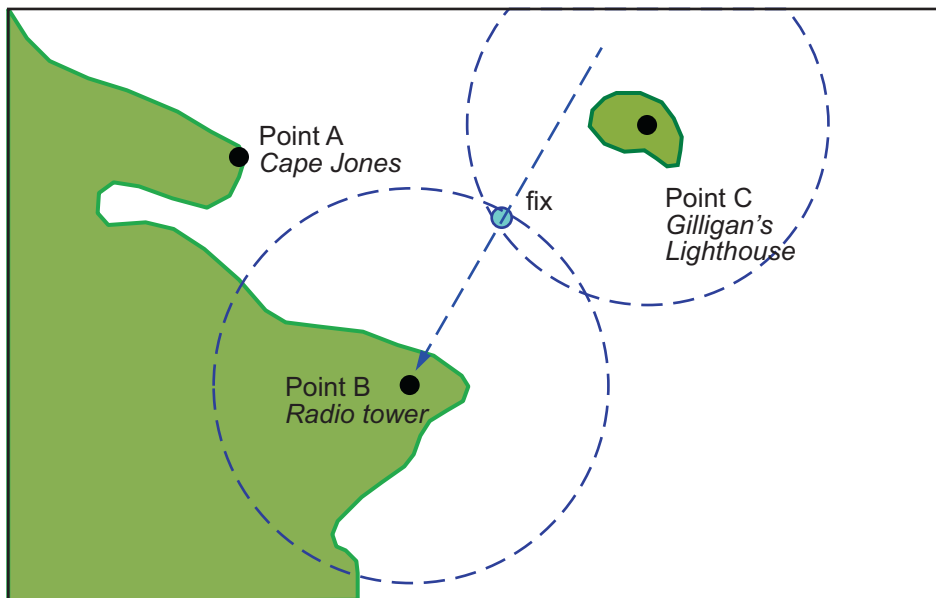
Luckily, your radar watch reports that he has Cape Jones at 18 nautical miles. This should resolve everything.



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,  $2 \times (n - \text{choose} - 2)$  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 radio tower of  $20^\circ$ , 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.

### 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 NaNs because it is used only for the two-intersection ambiguity associated with arcs.

For the range-arcs-only example, the syntax is

```

[latfix,lonfix] = navfix([latA latB latC],[lonA lonB lonC],...
                        [16 14 15],[0 0 0])

```

This defines the three points and their ranges as taken from the ship. The final argument indicates that the three cases are all ranges.

The outputs have the following form:

```

latfix =
  latfix11  latfix12      % A intersecting B
  latfix21  latfix22      % A intersecting C
  latfix31  latfix32      % B intersecting C
lonfix =
  lonfix11  lonfix12      % A intersecting B
  lonfix21  lonfix22      % A intersecting C
  lonfix31  lonfix32      % B intersecting C

```

Here, the second column is used, because each pair of arcs has two potential intersections.

For the bearings and ranges example, the syntax requires the final input to indicate which objects are lines of bearing (indicated with a 1) and which are range arcs (indicated with a 0):

```

[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                        [20 14 15],[1 0 0])

```

The resulting output is mixed:

```
latfix =
  latfix11      NaN          % Line B intersecting Arc B
  latfix21  latfix22        % Line B intersecting Arc C
  latfix31  latfix32        % Arc B intersecting Arc C
lonfix =
  lonfix11      NaN          % Line B intersecting Arc B
  lonfix21  lonfix22        % Line B intersecting Arc C
  lonfix31  lonfix32        % Arc B intersecting Arc C
```

Only one intersection is returned for the line from B with the arc about B, since the line originates inside the circle and intersects it once. The same line intersects the other circle twice, and hence it returns two points. The two circles taken together also return two points.

Usually, you have an idea as to where you are before you take the fix. For example, you might have a dead reckoning position for the time of the fix (see below). If you provide `navfix` with this estimated position, it chooses from each pair of ambiguous intersections the point closest to the estimate. Here's what it might look like:

```
[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                        [20 14 15],[1 0 0],drlat,drlon)

latfix =
  latfix11          % the only point
  latfix21          % the closer point
  latfix31          % the closer point
lonfix =
  lonfix11          % the only point
  lonfix21          % the closer point
  lonfix31          % the closer point
```

## A Numerical Example of Using `navfix`

**1** Define some specific points in the middle of the Atlantic Ocean. These are strictly arbitrary; perhaps they correspond to points in Atlantis:

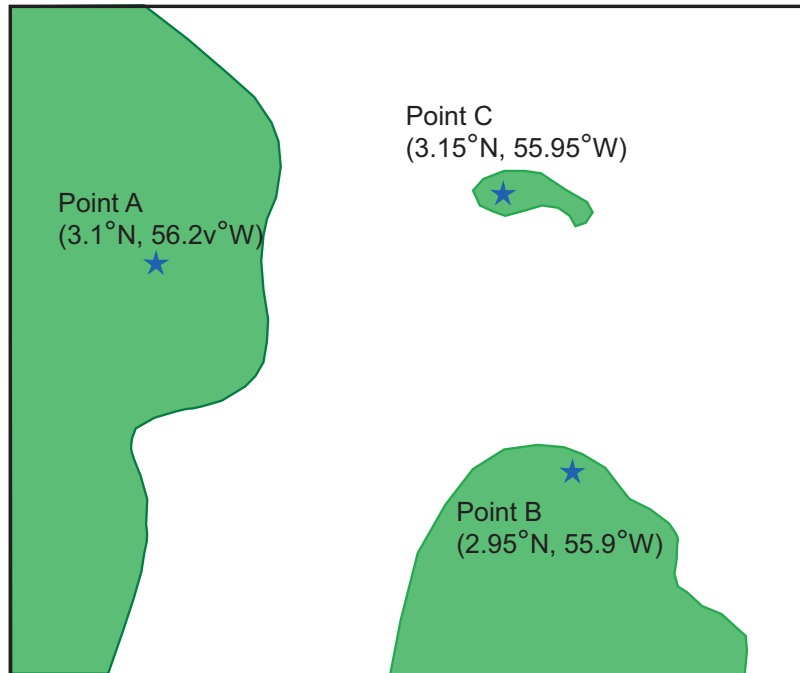
```
lata = 3.1;  lona = -56.2;
latb = 2.95; lonb = -55.9;
```

```
latc = 3.15; lonc = -55.95;
```

**2** Plot them on a Mercator projection:

```
axesm('MapProjection','mercator','Frame','on',...
      'MapLatLimit',[2.8 3.3],'MapLonLimit',[-56.3 -55.8])
plotm([lata latb latc],[lona lonb lonc],...
      'LineStyle','none','Marker','pentagram',...
      'MarkerEdgeColor','b','MarkerFaceColor','b',...
      'MarkerSize',12)
```

Here is what it looks like (the labeling and imaginary coastlines are added after the fact for illustration).



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

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [289 135 26.5],[1 1 1])
```

```

newlat =
  3.0214      NaN
  3.0340      NaN
  3.0499      NaN
newlong =
  -55.9715    NaN
  -56.0079    NaN
  -56.0000    NaN

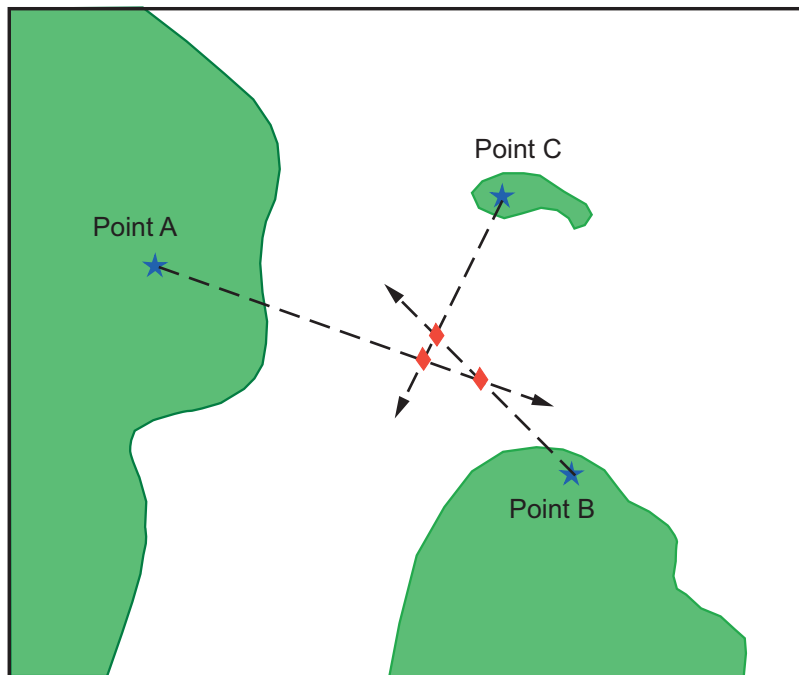
```

4 Add the bearing lines and intersection points to the map:

```

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

```



Notice that each pair of objects results in only one intersection, since all are lines of bearing.

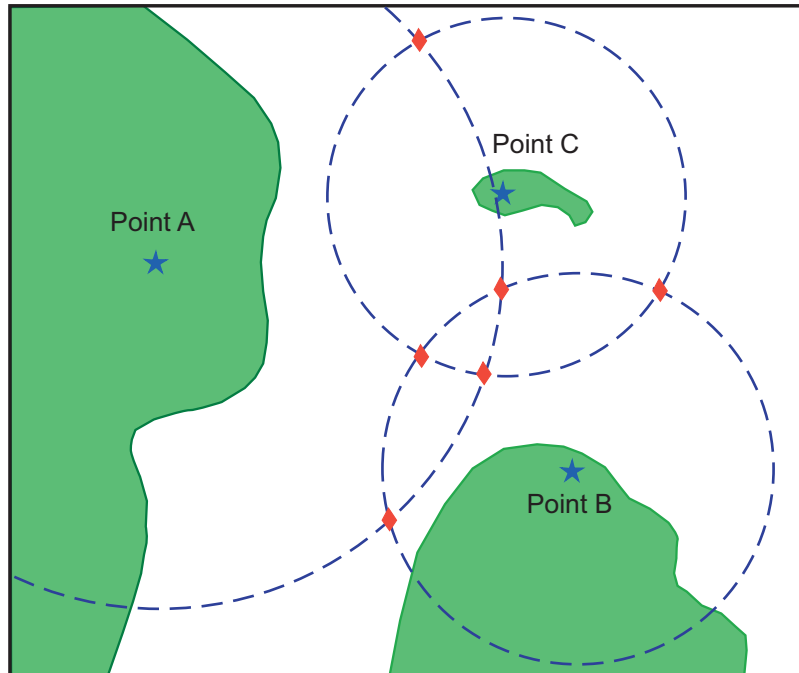


- 5 What if instead, you had ranges from the three points, A, B, and C, of 13 nmi, 9 nmi, and 7.5 nmi, respectively?

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...  
                          [13 9 7.5],[0 0 0])
```

```
newlat =  
  3.0739    2.9434  
  3.2413    3.0329  
  3.0443    3.0880  
newlong =  
 -55.9846  -56.0501  
 -56.0355  -55.9937  
 -56.0168  -55.8413
```

Here's what these points look like:

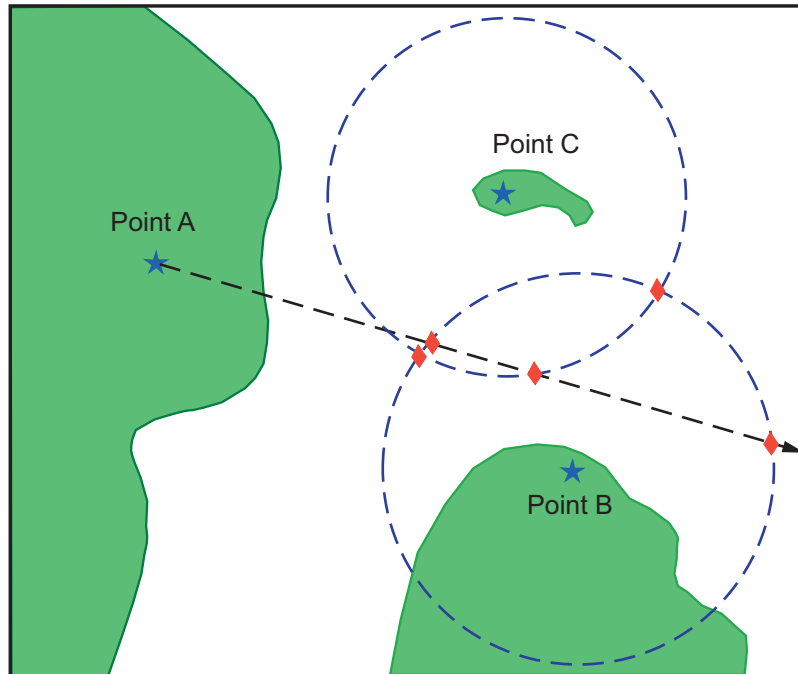


Three of these points look reasonable, three do not.

**6** What if, instead of a range from Point A, you had a bearing to it of 284°?

```
[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 (3.05°N,56.0°W), 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,dr lon)

newlat =
    3.0526
    3.0592
    3.0443
newlong =
   -56.0096
   -56.0360
   -56.0168
```

## Planning

You know that the shortest path between two geographic points is a great circle. Sailors and aviators are interested in minimizing distance traveled, and hence time elapsed. You also know that the rhumb line is a path of constant heading, the *natural* means of traveling. In general, to follow a great circle path, you would have to continuously alter course. This is impractical. However, you can approximate a great circle path by rhumb line segments so that the added distance is minor and the number of course changes minimal.

Surprisingly, very few rhumb line *track legs* are required to closely approximate the distance of the great circle path.

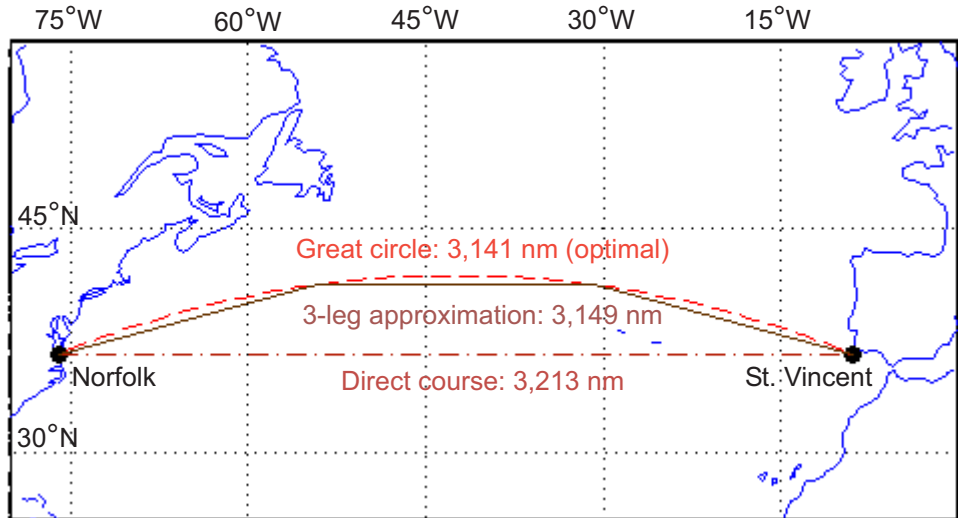
Consider the voyage from Norfolk, Virginia (37°N,76°W), to Cape St. Vincent, Portugal (37°N,9°W), one of the most heavily trafficked routes in the Atlantic. A due-east rhumb line track is 3,213 nautical miles, while the optimal great circle distance is 3,141 nautical miles.

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:



Mapping Toolbox provides the function `gcwaypts` to quickly calculate 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 do this with Mapping Toolbox, 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 4-61, or `inputm` in the Mapping Toolbox reference pages.

```

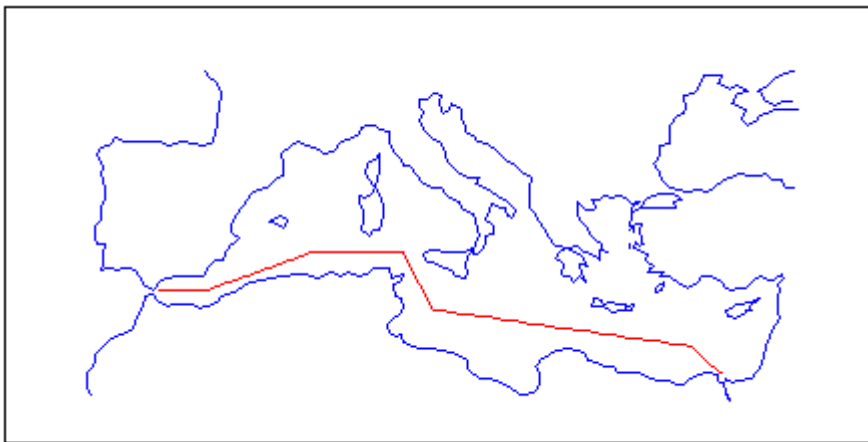
waypoints = [36 -5; 36 -2; 38 5; 38 11; 35 13; 33 30; 31.5 32]
waypoints =
    36.0000    -5.0000
    36.0000    -2.0000
    38.0000     5.0000
    38.0000    11.0000
    35.0000    13.0000
    33.0000    30.0000
    31.5000    32.0000
load coast
axesm('MapProjection','mercator',...
'MapLatLimit',[30 47],'MapLonLimit',[-10 37])
framem
plotm(lat,long)

[ltrk,ltrk] = track(waypoints);

```

```
plotm(lttrk,lntkr,'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
```



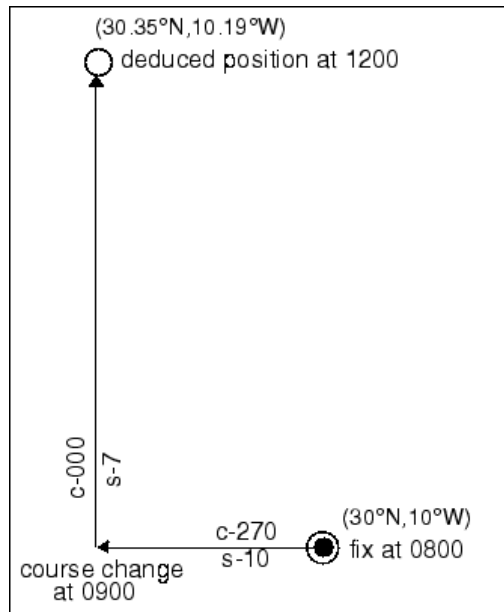
204.2073  
854.0092  
135.6415

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 (30°N,10°W) 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 (30.35°N,10.19°W) at 1200.



However, a sailor *shoots the sun* at local apparent noon and discovers that the ship's latitude is actually 30.29°N. 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?

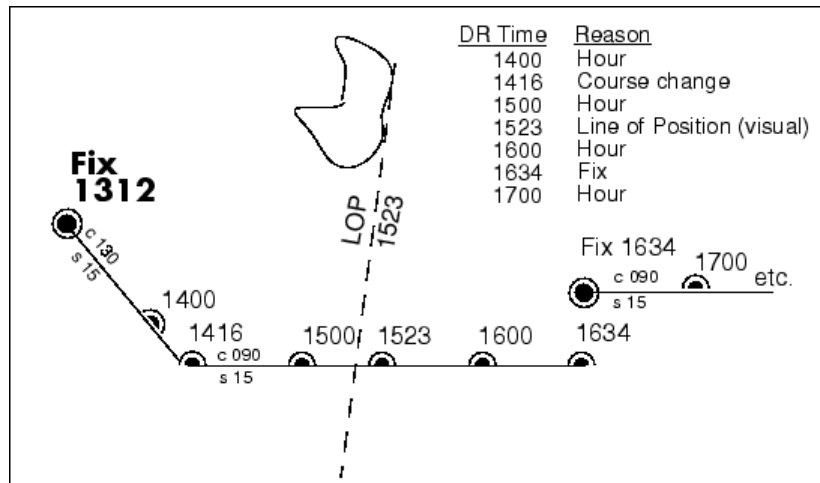
Leaving aside the difficulties in speed determination and the need to tack off course, even modern craft have to contend with winds and currents. However, despite these limitations, dead reckoning is still used for determining position between fixes and for forecasting future positions. This is because 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 *DRs*, 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 includes the function `dreckon`, which 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 (10°N,0°):

```
waypoints(1,:) = [10 0]; fixtime = 0;
```

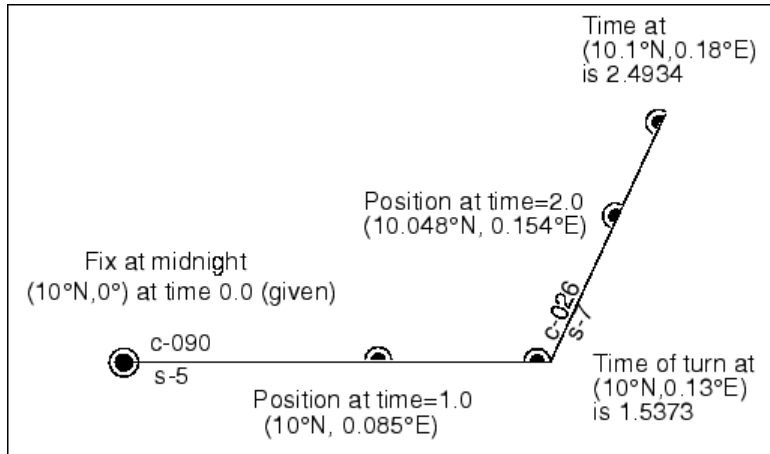
You intend to travel east and alter course at the point (10°N,0.13°E) and head for the point (10.1°N,0.18°E). On the first leg, you will travel at 5 knots, and on the second leg you will speed up to 7 knots.

```
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,drttime] = 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)?

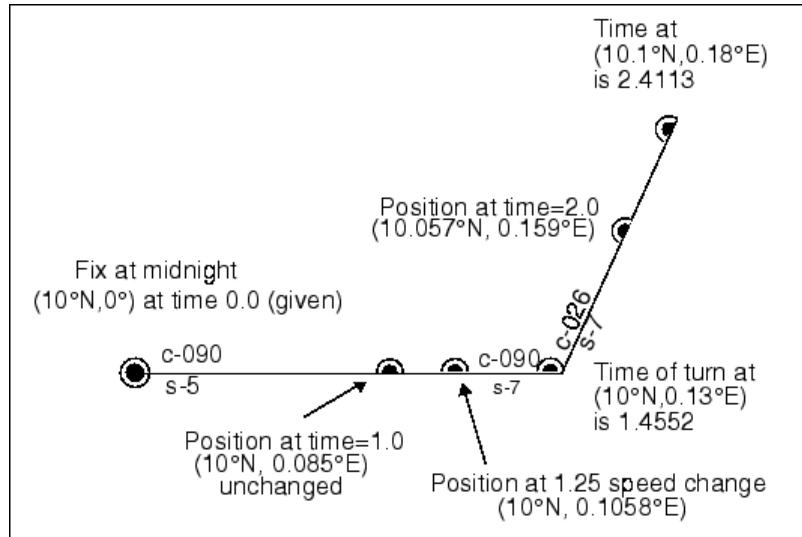
To indicate times for speed changes, another input is required, providing a time interval after the fix time at which each ordered speed is to end. The first speed, 5 knots, is to end 1.25 hours after midnight. Since you don't know when the rendezvous will be made under these circumstances, set the time 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,drttime] = dreckon(waypoints,fixtime,...
                               speeds,spdtimes);

[drlat,drlon,drttime]
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:



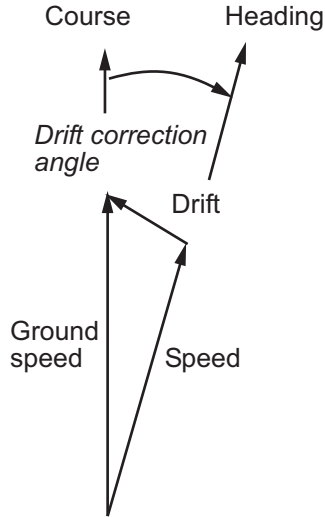
The times at planned positions after the speed change are a little earlier; the position at the known time (2 a.m.) is a little farther along. With this plan, you will arrive at the rendezvous about 4 1/2 minutes earlier, so you may want to consider a greater speed change.

## 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
```

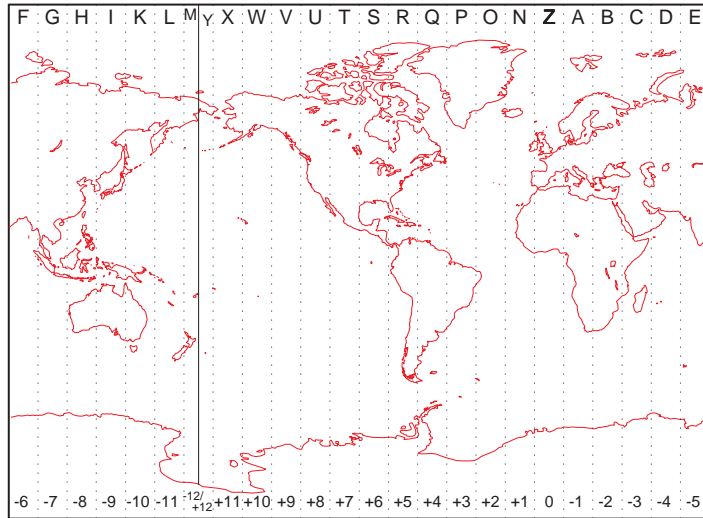
## Time Zones

Time zones used for navigation are uniform 15° 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 15° 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 7-1/2° 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 Mapping Toolbox does not contain any functions designed specifically for celestial navigation, a simple example can be devised.





It is possible with a sextant to determine *local apparent noon*. This is the moment when the Sun is at its zenith from your point of view. At the exact center longitude of a time zone, the phenomenon occurs exactly at noon, local time. Since the Sun traverses a 15° time zone in 1 hour, it crosses one degree every 4 minutes. So if you observe local apparent noon at 11:54, you must be 1.5° east of your center longitude.

You must know what time zone you are in before you can even attempt a fix. This concept has been understood since the spherical nature of the Earth was first accepted, but early sailors had no ability to keep accurate time on ship, and so were unable to determine their longitude. The invention of accurate chronometers in the 18th century solved this problem.

The timezone function is quite simple. It returns the description, *zd*, an integer for use in calculations, a string, *zltr*, of the zone designator, and a string fully naming the zone. For example, the information for a longitude 123°E is the following:

```
[zd,zltr,zone] = timezone(123)
zd =
```

```
-8  
zltr =  
H  
zone =  
-8 H
```

Returning to the simple celestial navigation example, the center longitude of this zone is:

```
-(zd*15)  
ans =  
120
```

This means that at our longitude, 123°E, we should experience local apparent noon at 11:48 a.m., 12 minutes early.

# Functions — By Category

---

Geospatial Data Import and Access (p. 10-2)	Readers, writers and associated utilities for geospatial file and data product formats
Vector Map Data and Geographic Data Structures (p. 10-6)	Manipulating polygons, geographic data structures and other vector geodata
Georeferenced Images and Data Grids (p. 10-8)	Constructing, georeferencing, analyzing, and manipulating raster geodata
Map Projections and Coordinates (p. 10-11)	Specifying, using, and analyzing map projections and geospatial coordinate transformations
Map Display and Interaction (p. 10-14)	Displaying geographic objects on maps and interacting with them
Geographic Calculations (p. 10-22)	Plane, spherical and ellipsoidal geometry
Utilities (p. 10-26)	Basic tasks, including time, angle, and distance conversions
Graphical User Interface Functions (p. 10-29)	GUI tools for selecting data and directly manipulating the content and appearance of maps

## Geospatial Data Import and Access

Standard File Formats (p. 10-2)	Reading and writing vector and raster geodata in widely used exchange formats
Gridded Terrain and Bathymetry Products (p. 10-3)	For reading raster data products distributed in special file formats
Vector Map Products (p. 10-4)	For reading vector data products distributed in special file formats
Miscellaneous Data Sets (p. 10-5)	For reading other data products distributed in special file formats
GUIs for Data Import (p. 10-5)	GUIs for browsing data products and selecting areas and objects of interest
File Reading Utilities (p. 10-5)	Low-level access functions for text and other data files
Ellipsoids, Radii, Areas, and Volumes (p. 10-5)	Geometric parameters of Earth, planets, Sun, and Moon

### Standard File Formats

<code>arcgridread</code>	Read gridded data set in Arc ASCII Grid Format
<code>geotiffinfo</code>	Information about GeoTIFF file
<code>geotiffread</code>	Read georeferenced image from GeoTIFF file
<code>getworldfilename</code>	Derive worldfile name from image filename
<code>kmlwrite</code>	Write geographic data to KML file
<code>makeattribspec</code>	Construct attribute specification from <code>geostruct</code>
<code>sdtsemread</code>	Read data from SDTS raster/DEM data set

<code>sdtinfo</code>	Information about SDTS data set
<code>shapeinfo</code>	Information about shapefile
<code>shaperead</code>	Read vector features and attributes from shapefile
<code>shapewrite</code>	Write geographic data structure to shapefile
<code>worldfileread</code>	Read worldfile and return referencing matrix
<code>worldfilewrite</code>	Construct worldfile from referencing matrix

## **Gridded Terrain and Bathymetry Products**

<code>dted</code>	Read U.S. Department of Defense Digital Terrain Elevation Data (DTED)
<code>dteds</code>	DTED filenames for latitude-longitude quadrangle
<code>etopo</code>	Read global 5-min or 2-min digital terrain data
<code>globedem</code>	Read Global Land One-km Base Elevation (GLOBE) data
<code>globedems</code>	GLOBE data filenames for latitude-longitude quadrangle
<code>gtopo30</code>	Read 30-arc-second global digital elevation data (GTOPO30)
<code>gtopo30s</code>	GTOPO30 data filenames for latitude-longitude quadrangle
<code>satbath</code>	Read 2-minute terrain/bathymetry from Smith and Sandwell
<code>tbase</code>	Read 5-minute global terrain elevations from TerrainBase

usgs24kdem	Read USGS 7.5-minute (30-m or 10-m) Digital Elevation Models
usgsdem	Read USGS 1-degree (3-arc-second) Digital Elevation Model
usgsdems	USGS 1-degree (3-arc-sec) DEM filenames for latitude-longitude quadrangle

## **Vector Map Products**

dcwdata	Read selected DCW worldwide basemap data
dcwgaz	Search DCW worldwide basemap gazette file
dcwread	Read DCW worldwide basemap file
dcwrhead	Read DCW worldwide basemap file headers
fipsname	Read Federal Information Processing Standard (FIPS) name file used with TIGER thinned boundary files
gshhs	Read Global Self-Consistent Hierarchical High-Resolution Shoreline
tgrline	Read TIGER/Line data
vmap0data	Read selected data from Vector Map Level 0
vmap0read	Read Vector Map Level 0 file
vmap0rhead	Read Vector Map Level 0 file headers

## Miscellaneous Data Sets

avhrrgoode	Read AVHRR data product stored in Goode Projection
avhrrlambert	Read AVHRR data product stored in eqaazim projection
egm96geoid	Read 15-minute gridded geoid heights from EGM96l
readfk5	Read Fifth Fundamental Catalog of Stars

## GUIs for Data Import

demdataui	UI for selecting digital elevation data
vmap0ui	UI for selecting data from Vector Map Level 0

## File Reading Utilities

grepfields	Identify matching fields in fixed record length files
readfields	Read fields or records from fixed-format files
readmtx	Read matrix stored in file
spread	Read columns of data from ASCII text file

## Ellipsoids, Radii, Areas, and Volumes

almanac	Parameters for Earth, planets, Sun, and Moon
---------	--

## Vector Map Data and Geographic Data Structures

Geographic Data Structures (p. 10-6)	For updating and obtaining fields from data structures
Data Manipulation (p. 10-6)	For altering, combining, and analyzing polygon and line data
Utilities for NaN-Separated Polygons and Lines (p. 10-7)	For structuring vectors defining multiple line or polygon objects

### Geographic Data Structures

<code>extractfield</code>	Field values from structure array
<code>extractm</code>	Vector data from Version 1 geographic data structure
<code>updategeostruct</code>	Convert geographic data structure from Version 1 to Version 2

### Data Manipulation

<code>bufferm</code>	Buffer zones for latitude-longitude polygons
<code>flatearthpoly</code>	Insert points along date line to pole
<code>interp</code>	Densify latitude-longitude sampling in lines or polygons
<code>intrplat</code>	Interpolate latitude at given longitude
<code>intrplon</code>	Interpolate longitude at given latitude
<code>ispolycw</code>	True if polygon vertices are in clockwise order
<code>nanclip</code>	Clip vector data with NaNs at specified pen-down locations



<code>poly2ccw</code>	Convert polygon contour to counterclockwise vertex ordering
<code>poly2cw</code>	Convert polygon contour to clockwise vertex ordering
<code>poly2fv</code>	Convert polygonal region to patch faces and vertices
<code>polybool</code>	Set operations on polygonal regions
<code>polycut</code>	Polygon branch cuts for holes
<code>polyjoin</code>	Convert polygon segments from cell array to vector format
<code>polymerge</code>	Merge line segments with matching endpoints
<code>polysplit</code>	Extract segments of NaN-delimited polygon vectors to cell arrays
<code>polyxpoly</code>	Compute line or polygon intersection points
<code>reducem</code>	Reduce density of points in vector data

## Utilities for NaN-Separated Polygons and Lines

<code>closePolygonParts</code>	Close all rings in multipart polygon
<code>isShapeMultipart</code>	True, if polygon or line has multiple parts
<code>removeExtraNaNSeparators</code>	Clean up NaN separators in polygons and lines

## Georeferenced Images and Data Grids

Spatial Referencing (p. 10-8)	Computing bounds and converting between geographic and raster coordinates for spatially referenced images and grids
Terrain Analysis (p. 10-9)	Computing slope, aspect, lines of sight, and terrain visibility
Other Analysis/Access (p. 10-9)	Computing areas and profiles, and selecting subsets of values from data grids
Construction and Modification (p. 10-10)	Constructing, encoding, seeding, reorienting, and converting data grids
Initialization (p. 10-10)	Generating data grids containing uniform values

### Spatial Referencing

<code>latlon2pix</code>	Convert latitude-longitude coordinates to pixel coordinates
<code>limitm</code>	Determine latitude and longitude limits of regular data grid
<code>makerefmat</code>	Construct affine spatial-referencing matrix
<code>map2pix</code>	Convert map coordinates to pixel coordinates
<code>mapbbox</code>	Compute bounding box of georeferenced image or data grid
<code>mapoutline</code>	Compute outline of georeferenced image or data grid
<code>meshgrat</code>	Construct map graticule for surface object display

<code>pix2map</code>	Convert pixel coordinates to map coordinates
<code>pixcenters</code>	Compute pixel centers for georeferenced image or data grid
<code>refmat2vec</code>	Convert referencing matrix to referencing vector
<code>refvec2mat</code>	Convert referencing vector to referencing matrix
<code>set1t1n</code>	Convert data grid rows and columns to latitude-longitude
<code>setpostn</code>	Convert latitude-longitude to data grid rows and columns

## **Terrain Analysis**

<code>gradientm</code>	Calculate gradient, slope and aspect of data grid
<code>los2</code>	Line-of-sight visibility between two points in terrain
<code>viewshed</code>	Areas visible from point on terrain elevation grid

## **Other Analysis/Access**

<code>areamat</code>	Surface area covered by nonzero values in binary data grid
<code>filterm</code>	Filter latitudes and longitudes based on underlying data grid
<code>findm</code>	Latitudes and longitudes of nonzero data grid elements

<code>ltln2val</code>	Extract data grid values for specified locations
<code>mapprofile</code>	Interpolate heights between waypoints on regular data grid

## Construction and Modification

<code>changem</code>	Substitute values in data array
<code>encodem</code>	Fill in regular data grid from seed values and locations
<code>geoloc2grid</code>	Convert geolocated data array to regular data grid
<code>imbedm</code>	Encode data points into regular data grid
<code>neworig</code>	Orient regular data grid to oblique aspect
<code>resizem</code>	Resize regular data grid
<code>sizem</code>	Row and column dimensions needed for regular data grid
<code>vec2mtx</code>	Convert latitude-longitude vectors to regular data grid

## Initialization

<code>nanm</code>	Construct regular data grid of NaNs
<code>onem</code>	Construct regular data grid of 1s
<code>spzerom</code>	Construct sparse regular data grid of 0s
<code>zerom</code>	Construct regular data grid of 0s

## Map Projections and Coordinates

Available Map Projections (p. 10-11)	Lists of map projections and characteristics
Map Projection Transformations (p. 10-12)	Forward and inverse map projection functions
Map Trimming (p. 10-12)	For trimming lines, polygons, and data grids to latitude-longitude quadrangles
Angles, Scales, and Distortions (p. 10-12)	Computing directions, angles, and distortions on projected maps
Visualizing Map Distortions (p. 10-13)	Generating displays of distortion statistics and Tissot ellipses
UTM System (p. 10-13)	Selecting zones and ellipsoids for the Universal Transverse Mercator system
Rotating Coordinates on the Sphere (p. 10-13)	Reorienting map data by solid-body rotations on the sphere
Trimming and Clipping (p. 10-13)	Removing and replacing data that extends outside a map frame

For specific map projections, see Chapter 12, “Map Projections — By Category”.

### Available Map Projections

<code>maplist</code>	Map projections available in Mapping Toolbox
<code>maps</code>	List available map projections and verify names
<code>projlist</code>	Map projections supported by <code>projfwd</code> and <code>projinv</code>

## Map Projection Transformations

<code>mfwdtran</code>	Project geographic features to map coordinates
<code>minvtran</code>	Unproject features from map to geographic coordinates
<code>projfwd</code>	Forward map projection using PROJ.4 map projection library
<code>projinv</code>	Inverse map projection using PROJ.4 map projection library

## Map Trimming

<code>maptriml</code>	Trim lines to latitude-longitude quadrangle
<code>maptrimp</code>	Trim polygons to latitude-longitude quadrangle
<code>maptrims</code>	Trim regular data grid to latitude-longitude quadrangle

## Angles, Scales, and Distortions

<code>distortcalc</code>	Distortion parameters for map projections
<code>vfdtran</code>	Direction angle in map plane from azimuth on ellipsoid
<code>vinvtran</code>	Azimuth on ellipsoid from direction angle in map plane

## Visualizing Map Distortions

<code>mdistort</code>	Display contours of constant map distortion
<code>tissot</code>	Project Tissot indicatrices on map axes

## UTM System

<code>utmgeoid</code>	Select ellipsoids for given UTM zone
<code>utmzone</code>	Select UTM zone given latitude and longitude

## Rotating Coordinates on the Sphere

<code>newpole</code>	Origin vector to place specific point at pole
<code>org2pol</code>	Location of north pole in rotated map
<code>putpole</code>	Origin vector to place north pole at specified point

## Trimming and Clipping

<code>clipdata</code>	Clip data at $\pm\pi$ in longitude, $\pm\pi$ in latitude
<code>trimcart</code>	Trim graphic objects to map frame
<code>trimdata</code>	Trim map data exceeding projection limits
<code>undoclip</code>	Remove object clips introduced by <code>clipdata</code>
<code>undotrim</code>	Remove object trims introduced by <code>trimdata</code>

## Map Display and Interaction

Map Creation and High-Level Display (p. 10-15)	Top-level functions that create map axes, project map data onto them, and control symbolization
Vector Symbolization (p. 10-15)	Functions that draw symbols for points, lines, and polygons (coordinate lists and geostructs)
Displaying Lines and Contours (p. 10-15)	Lower level line plotting and higher level contour plotting functions
Displaying Patch Data (p. 10-16)	Lower-level functions for plotting polygons as patches on map axes
Displaying Data Grids (p. 10-16)	For mapping regular and geolocated data grids in 2-D and 3-D
Displaying Light Objects and Lighted Surfaces (p. 10-16)	For mapping regular and geolocated data grids using lighting and shading
Displaying Thematic Maps (p. 10-17)	For making scatter, quiver, comet, and stem maps
Annotating Map Displays (p. 10-17)	For adding north arrows, graphic scales, text and other annotations to maps
Colormaps for Map Displays (p. 10-18)	For constructing colormaps appropriate for map displays
Interactive Map Positions (p. 10-18)	For graphic interaction with data in map axes
Interactive Track and Circle Definition (p. 10-19)	For constructing great and small circles, rhumb lines and other geographic tracks
Graphical User Interfaces (p. 10-19)	GUIs for specific functions and general GUIs for interactive mapping
Map Object and Projection Properties (p. 10-20)	For querying, setting, and modifying map axes objects and properties



Controlling Map Appearance (p. 10-21)	For controlling the view and map scale
Clearing Map Displays/Managing Visibility (p. 10-21)	For showing, hiding, and removing objects from map axes

## Map Creation and High-Level Display

<code>axesm</code>	Define map axes and set map properties
<code>displaym</code>	Display Version 1 geographic data structure
<code>geoshow</code>	Display map latitude and longitude data
<code>grid2image</code>	Display regular data grid as image
<code>mapview</code>	Interactive map viewer
<code>usamap</code>	Construct map axes for United States of America
<code>worldmap</code>	Construct map axes for given region of world

## Vector Symbolization

<code>makesymbolspec</code>	Construct vector layer symbolization specification
-----------------------------	---

## Displaying Lines and Contours

<code>contour3m</code>	Project 3-D contour plot of map data
<code>contourfm</code>	Project filled 2-D contour plot of map data
<code>contourm</code>	Project 2-D contour plot of map data

<code>linem</code>	Project line object on map axes
<code>plot3m</code>	Project 3-D lines and points on map axes
<code>plotm</code>	Project 2-D lines and points on map axes

## Displaying Patch Data

<code>fill3m</code>	Project filled 3-D patch objects on map axes
<code>fillm</code>	Project filled 2-D patch objects on map axes
<code>patchesm</code>	Project patches on map axes as individual objects
<code>patchm</code>	Project patch objects on map axes

## Displaying Data Grids

<code>meshm</code>	Project regular data grid on map axes
<code>pcolorm</code>	Project regular data grid on map axes in $z=0$ plane
<code>surfacem</code>	Project and add geolocated data grid to current map axes
<code>surfm</code>	Project geolocated data grid on map axes

## Displaying Light Objects and Lighted Surfaces

<code>lightm</code>	Project light objects on map axes
<code>meshl3m</code>	3-D lighted shaded relief of regular data grid

<code>shadere1</code>	Construct <code>cdata</code> and <code>colormap</code> for shaded relief
<code>surflm</code>	3-D shaded surface with lighting on map axes
<code>surflsrm</code>	3-D lighted shaded relief of geolocated data grid

## Displaying Thematic Maps

<code>comet3m</code>	Project 3-D comet plot on map axes
<code>cometm</code>	Project 2-D comet plot on map axes
<code>quiverm</code>	Project 2-D quiver plot on map axes
<code>scatterm</code>	Project point markers with variable color and area
<code>stem3m</code>	Project stem plot map on map axes
<code>symbolm</code>	Project point markers with variable size

## Annotating Map Displays

<code>clabelm</code>	Add contour label map contour display
<code>framem</code>	Toggle and control display of map frame
<code>gridm</code>	Toggle and control display of map grid
<code>lcolorbar</code>	Colorbar with text labels
<code>mlabel</code>	Toggle and control display of meridian labels
<code>mlabelzero22pi</code>	Convert meridian labels to 0-360 degree range

northarrow	Add graphic element pointing to geographic north pole
plabel	Toggle and control display of parallel labels
rotatetext	Rotate text to projected graticule
scaleruler	Add or modify graphic scale on map axes
textm	Project text annotation on map axes

## **Colormaps for Map Displays**

contourmap	Contour colormap and colorbar current axes
demcmap	Colormaps appropriate to terrain elevation data
polcmap	Colormaps appropriate to political regions

## **Interactive Map Positions**

gcpmap	Current mouse point from map axes
gtextm	Place text on map using mouse
inputm	Latitudes and longitudes of mouse-click locations

## Interactive Track and Circle Definition

<code>scircleg</code>	Small circle defined via mouse input
<code>sectorg</code>	Sector of small circle defined via mouse input
<code>trackg</code>	Great circle or rhumb line defined via mouse input

## Graphical User Interfaces

<code>clrmnu</code>	Add colormap menu to figure window
<code>colorm</code>	Create index map colormaps
<code>colorui</code>	Interactively define RGB color
<code>getseeds</code>	Interactively assign seeds for data grid encoding
<code>lightmui</code>	Control position of lights on globe or 3-D map
<code>maptool</code>	Add menu activated tools to map figure
<code>maptrim</code>	Interactively trim and convert map data from vector to raster format
<code>mlayers</code>	Interactive display of Version 1 geographic data structures
<code>mobjects</code>	Manipulate object sets displayed on map axes
<code>originui</code>	Interactively modify map origin
<code>panzoom</code>	Pan and zoom on map axes
<code>parallelui</code>	Interactively modify map parallels
<code>qrydata</code>	GUI to interactively perform data queries

<code>rootlayr</code>	Construct cell array of workspace variables for <code>mlayers</code> tool
<code>seedm</code>	GUI to fill data grids with seeded values
<code>surfdist</code>	Interactive distance, azimuth, and reckoning calculations
<code>uimaptbx</code>	Handle <code>buttondown</code> callbacks for mapped objects
<code>utmzoneui</code>	Choose or identify UTM zone by clicking map

## Map Object and Projection Properties

<code>cart2grn</code>	Transform projected coordinates to Greenwich system
<code>defaultm</code>	Initialize or reset projection properties to default values
<code>gcm</code>	Current map projection structure
<code>geotiff2mstruct</code>	Convert GeoTIFF information to map projection structure
<code>getm</code>	Map object properties
<code>handlem</code>	Handles of displayed map objects
<code>ismap</code>	True for axes with map projection
<code>ismapped</code>	True, if object is projected on map axes
<code>makemapped</code>	Convert ordinary graphics object to mapped object
<code>namem</code>	Determine names of valid graphics objects
<code>project</code>	Project displayed map graphics object

restack	Restack objects within map axes
rotatem	Transform vector map data to new origin and orientation
setm	Set properties of map axes and graphics objects
tagm	Set Tag property of map graphics object
zdatam	Adjust $z$ -plane of displayed map objects

## Controlling Map Appearance

axesscale	Resize axes for equivalent scale
camposm	Set camera position using geographic coordinates
camtargm	Set camera target using geographic coordinates
camupm	Set camera up vector using geographic coordinates
daspectm	Control vertical exaggeration in map display
paperscale	Set figure properties for printing at specified map scale
previewmap	View map at printed size
tightmap	Remove white space around map

## Clearing Map Displays/Managing Visibility

clma	Clear current map axes
clmo	Clear specified graphics objects from map axes

hidem	Hide specified graphic objects on map axes
showaxes	Toggle display of map coordinate axes
showm	Specify graphic objects to display on map axes

## Geographic Calculations

Geometry of Sphere and Ellipsoid (p. 10-22)	Distances, deviations, areas, and curves on the sphere or ellipsoid
3-D Coordinates (p. 10-24)	For converting between different 3-D coordinate systems
Ellipsoids and Latitudes (p. 10-24)	For converting ellipsoid parameters and auxiliary latitudes
Intersections in the Cartesian Plane (p. 10-25)	For determining planar intersections of circles and lines
Geographic Statistics (p. 10-25)	For computing geographic means, standard deviations, and histograms
Navigation (p. 10-26)	For determining positions, headings, drift, and navigational fixes and way points

## Geometry of Sphere and Ellipsoid

antipode	Point on opposite side of globe
areaint	Surface area of polygon on sphere or ellipsoid
areaquad	Surface area of latitude-longitude quadrangle



azimuth	Azimuth between points on sphere or ellipsoid
departure	Departure of longitudes at specified latitudes
distance	Distance between points on sphere or ellipsoid
ellipse1	Geographic ellipse from center, semimajor axes, eccentricity, and azimuth
gc2sc	Center and radius of great circle
gcxgc	Intersection points for pairs of great circles
gcxsc	Intersection points for great and small circle pairs
meridianarc	Ellipsoidal distance along meridian
meridianfwd	Reckon position along meridian
reckon	Point at specified azimuth, range on sphere or ellipsoid
rhxrh	Intersection points for pairs of rhumb lines
scircle1	Small circles from center, range, and azimuth
scircle2	Small circles from center and perimeter
scxsc	Intersection points for pairs of small circles
track1	Geographic tracks from starting point, azimuth, and range
track2	Geographic tracks from starting and ending points

### 3-D Coordinates

<code>ecef2geodetic</code>	Convert geocentric (ECEF) to geodetic coordinates
<code>ecef2lv</code>	Convert geocentric (ECEF) to local vertical coordinates
<code>elevation</code>	Local vertical elevation angle, range, and azimuth
<code>geodetic2ecef</code>	Convert geodetic to geocentric (ECEF) coordinates
<code>lv2ecef</code>	Convert local vertical to geocentric (ECEF) coordinates

### Ellipsoids and Latitudes

<code>axes2ecc</code>	Eccentricity of ellipse with given axis lengths
<code>convertlat</code>	Convert between geodetic and auxiliary latitudes
<code>ecc2flat</code>	Flattening of ellipse with given eccentricity
<code>ecc2n</code>	n-value of ellipse with given eccentricity
<code>flat2ecc</code>	Eccentricity of ellipse with given flattening
<code>geocentric2geodeticLat</code>	Convert geocentric to geodetic latitude
<code>geodetic2geocentricLat</code>	Convert geodetic to geocentric latitude
<code>majaxis</code>	Semimajor axis of ellipse given semiminor axis and eccentricity
<code>minaxis</code>	Semiminor axis of ellipse given semimajor axis and eccentricity

n2ecc	Eccentricity of ellipse with given n-value
rcurve	Radii of curvature of ellipsoid
rsphere	Radii of auxiliary spheres

## Intersections in the Cartesian Plane

circcirc	Intersections of circles in Cartesian plane
linecirc	Intersections of circles and lines in Cartesian plane

## Geographic Statistics

combntns	All possible combinations of set of values
eqa2grn	Convert from equal area to Greenwich coordinates
grn2eqa	Convert from Greenwich to equal area coordinates
hista	Histogram for geographic points with equal-area bins
meanm	Mean location of geographic coordinates
stdist	Standard distance for geographic points
stdm	Standard deviation for geographic points

## Navigation

<code>crossfix</code>	Cross-fix positions from bearings and ranges
<code>dreckon</code>	Dead reckoning positions for track
<code>driftcorr</code>	Heading to correct for wind or current drift
<code>driftvel</code>	Wind or current from heading, course, and speeds
<code>gcwaypts</code>	Equally spaced waypoints along great circle
<code>legs</code>	Courses and distances between navigational waypoints
<code>navfix</code>	Mercator-based navigational fix
<code>timezone</code>	Time zone based on longitude
<code>track</code>	Track segments to connect navigational waypoints

## Utilities

Angle Conversions (p. 10-27)	For converting angles between different units and encodings
Conversion Factors for Angles and Distances (p. 10-27)	Function to compute factor for converting between units of distance and angles
Data Precision (p. 10-28)	For managing data precision
Distance Conversions (p. 10-28)	For converting distances between different units and encodings
Image Conversion (p. 10-28)	Function for changing indexed images to uint8 true-color images

String Formatters (p. 10-28)

For formatting angles and distances as text suitable for annotations

Longitude or Azimuth Wrapping (p. 10-29)

For forcing angles to lie within specified intervals

## Angle Conversions

deg2rad

Convert angles from degrees to radians

degrees2dm

Convert degrees to degrees-minutes

degrees2dms

Convert degrees to degrees-minutes-seconds

dm2degrees

Convert degrees-minutes to degrees

dms2degrees

Convert degrees-minutes-seconds to degrees

fromDegrees

Convert angles from degrees

fromRadians

Convert angles from radians

rad2deg

Convert angle units from radians to degrees

str2angle

Convert strings to angles in degrees

toDegrees

Convert angles to degrees

toRadians

Convert angles to radians

## Conversion Factors for Angles and Distances

unitsratio

Unit conversion factors

## Data Precision

<code>epsm</code>	Accuracy in angle units for certain map computations
<code>roundn</code>	Round numbers to specified power of 10

## Distance Conversions

<code>deg2km</code> , <code>deg2nm</code> , <code>deg2sm</code>	Convert distance from degrees to kilometers, nautical miles, or statute miles
<code>km2deg</code> , <code>nm2deg</code> , <code>sm2deg</code>	Convert from distance units to degrees
<code>km2nm</code> , <code>km2sm</code> , <code>nm2km</code> , <code>nm2sm</code> , <code>sm2km</code> , <code>sm2nm</code>	Convert distance between kilometers and miles
<code>km2rad</code> , <code>nm2rad</code> , <code>sm2rad</code>	Convert from distance units to radians
<code>rad2km</code> , <code>rad2nm</code> , <code>rad2sm</code>	Convert distance from radians to kilometers, nautical miles, or statute miles

## Image Conversion

<code>ind2rgb8</code>	Convert indexed image to uint8 RGB image
-----------------------	--

## String Formatters

<code>angl2str</code>	Format angle strings
<code>dist2str</code>	Format distance strings

## Longitude or Azimuth Wrapping

<code>unwrapMultipart</code>	Unwrap vector of angles with NaN-delimited parts
<code>wrapTo180</code>	Wrap angle in degrees to [-180 180]
<code>wrapTo2Pi</code>	Wrap angle in radians to [0 2*pi]
<code>wrapTo360</code>	Wrap angle in degrees to [0 360]
<code>wrapToPi</code>	Wrap angle in radians to [-pi pi]

## Graphical User Interface Functions

Map Definition Tools (p. 10-30)	Selecting vector and raster data, defining map axes, and projection parameters
Mapping Tools (p. 10-30)	Displaying maps, manipulating layers, and querying map objects
Display Manipulation Tools (p. 10-30)	Controlling zoom levels, colormaps, and lighting
Object Property Tools (p. 10-31)	Showing, hiding, tagging, and clearing objects, and customizing colormaps
Track Tools (p. 10-31)	Plotting small and great circles, rhumb lines, and other navigational tracks
Map Data Construction Tools (p. 10-32)	Setting limits, trimming maps, and seeding grid values

## Map Definition Tools

<code>axesm, axesmui</code>	Define map axes and modify map projection and display properties
<code>demdataui</code>	UI for selecting digital elevation data
<code>originui</code>	Interactively modify map origin
<code>parallelui</code>	Interactively modify map parallels
<code>utmzoneui</code>	Choose or identify UTM zone by clicking map
<code>vmap0ui</code>	UI for selecting data from Vector Map Level 0

## Mapping Tools

<code>maptool</code>	Add menu activated tools to map figure
<code>maptrim</code>	Interactively trim and convert map data from vector to raster format
<code>mapview</code>	Interactive map viewer
<code>mayers</code>	Interactive display of Version 1 geographic data structures
<code>mobjects</code>	Manipulate object sets displayed on map axes
<code>qrydata</code>	GUI to interactively perform data queries

## Display Manipulation Tools

<code>clrmenu</code>	Add colormap menu to figure window
<code>hidem-ui</code>	Hide specified mapped objects



lightmui	Control position of lights on globe or 3-D map
panzoom	Pan and zoom on map axes

## Object Property Tools

clmo	Clear specified graphics objects from map axes
colorui	Interactively define RGB color
handlem	Handles of displayed map objects
handlem-ui	GUI for handles of specified mapped objects
hidem	Hide specified graphic objects on map axes
property editors	GUIs to edit properties of mapped objects
showm	Specify graphic objects to display on map axes
tagm	Set Tag property of map graphics object
zdatam	Adjust $z$ -plane of displayed map objects

## Track Tools

scircleg	Small circle defined via mouse input
scirclui	GUI to display small circles on map axes
sectorg	Sector of small circle defined via mouse input

<code>surfdist</code>	Interactive distance, azimuth, and reckoning calculations
<code>trackg</code>	Great circle or rhumb line defined via mouse input
<code>trackui</code>	GUI to display great circles and rhumb lines on map axes

## Map Data Construction Tools

<code>colorm</code>	Create index map colormaps
<code>seedm</code>	GUI to fill data grids with seeded values

# Functions — Alphabetical List

---

**Purpose** Parameters for Earth, planets, Sun, and Moon

**Syntax**

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

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

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

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

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

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

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

```
'everest'      1830 Everest ellipsoid
'bessel'       1841 Bessel ellipsoid
'airy'         1849 Airy ellipsoid
'clarke66'     1866 Clarke ellipsoid
```

'clarke80'	1880 Clarke ellipsoid
'international'	1924 International ellipsoid
'krasovsky'	1940 Krasovsky ellipsoid
'wgs60'	1960 World Geodetic System ellipsoid
'iau65'	1965 International Astronomical Union ellipsoid
'wgs66'	1966 World Geodetic System ellipsoid
'iau68'	1968 International Astronomical Union ellipsoid
'wgs72'	1972 World Geodetic System ellipsoid
'grs80'	1980 Geodetic Reference System ellipsoid
'wgs84'	1984 World Geodetic System ellipsoid

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

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

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

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

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

## Examples

The radius of the Earth (treated as a sphere) in kilometers is

```
almanac('earth','radius')
```

```
ans =  
6371
```

The default ellipsoid model for the Earth ([semimajor axis eccentricity]) is

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

```
ans =  
1.0e+03 *  
6.3781    0.0001
```

Note that the radius returned for any ellipsoid model reference body is the semimajor axis:

```
almanac('earth','radius','kilometers','ellipsoid')
```

```
Warning: Semimajor axis returned for radius parameter  
ans =  
6.3781e+03
```

Compare the tabulated values of the Earth's surface area with a spherical assumption and with the 1966 World Geodetic System ellipsoid model:

```
almanac('earth','surfarea','statutemiles','actual')
```

```
ans =  
1.969499232704451e+008
```

```
almanac('earth','surfarea','statutemiles','sphere')
```

```
ans =  
1.969362058529953e+008
```

```
almanac('earth', 'surfarea', 'statutemiles', 'wgs66')
```

```
ans =  
1.969371331484438e+008
```

Note that these values are so close that long notation is required to differentiate them.

Some lunar measurements are

```
almanac('moon', 'radius')
```

```
ans =  
1738
```

```
almanac('moon', 'surfarea')
```

```
ans =  
3.7959e+07
```

```
almanac('moon', 'volume')
```

```
ans =  
2.1991e+10
```

## Remarks

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

## See Also

distance

# angl2str

---

## Purpose

Format angle strings

## Syntax

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

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

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

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

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

`str = angl2str(angle,signcode,units,n)` uses the integer *n* to control the number of significant digits provided in the output. *n* is the power of 10 representing the last place of significance in the number of degrees or radians. For example, if *n* = -2 (the default), `angl2str` rounds to the nearest hundredth. If *n* = -0, `angl2str` rounds to the nearest integer. And if *n* == 1, `angl2str` rounds to the tens place, although positive values of *n* are of little practical use.



**Remarks**

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

**Examples**

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

```
a = -3:1.5:3;
str = angl2str(a, 'ns', 'degrees', -5)
str =
  3.00000{\circ} S
  1.50000{\circ} S
  0.00000{\circ}
  1.50000{\circ} N
  3.00000{\circ} N
```

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

```
3" 00' 00.00" S
1" 30' 00.00" S
0" 00' 00.00"
1" 30' 00.00" N
3" 00' 00.00" N
```

**See Also**

`str2angle`, `dist2str`

# angledim

---

**Purpose** Convert angles units

**Syntax**

---

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

---

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

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

**Example**

Convert from degrees to radians:

```
angledim(23.45134, 'degrees', 'radians')  
  
ans =  
    0.4093
```

**See Also**

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

**Purpose**

Point on opposite side of globe

**Syntax**

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

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

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

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

**Examples**

Given a point (43°N, 15°E), find its antipode:

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

or (43°S, 165°W).

**Example 1**

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

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

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

## Example 2

Find antipode of the Mathworks corporate headquarters in Natick, Massachusetts, USA, and map both places in orthographic projection. Begin by specifying its latitude and longitude as degree-minutes-seconds and converting them to decimal degrees:

```
mwlats = dms2degrees([ 42 18 2.5])
mwlons = dms2degrees([-71 21 7.9])

mwlats =
    42.3007
mwlons =
   -71.3522

[amwlats amwlons] = antipode(mwlats, mwlons)

amwlats =
   -42.3007
amwlons =
   108.6478
```

Prove that these points are antipodes:

```
dist = distance(mwlats,mwlons, amwlats,amwlons)

dist =
    180.0000
```

The distance function shows them to be 180 degrees apart.

Generate a map centered on the original point:

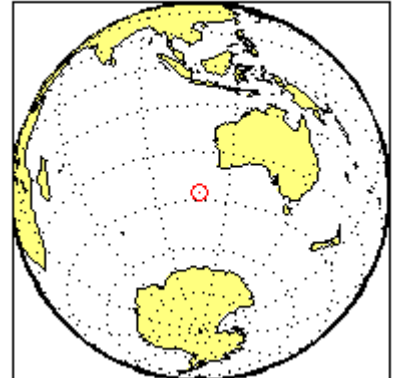
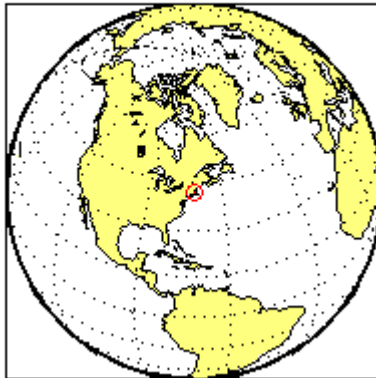
```
subplot(1,2,1)
axesm ('MapProjection','ortho', 'origin',[mwlats mwlons],...
      'frame','on', 'grid','on')
load coast
geoshow(lat, long, 'displaytype', 'polygon')
geoshow(mwlats, mwlons, 'Marker', 'o', 'Color', 'red')
```

```
s = ['Looking down at (' angl2str(mwlat,'ns') ...
    ',' angl2str(mwlon,'ew') ')'];
title(s)
```

Add a second map centered on the computed antipodal point:

```
subplot(1,2,2)
axesm ('MapProjection','ortho', 'origin',[amwlat amwlon],...
    'frame','on', 'grid','on')
geoshow(lat, long, 'displaytype', 'polygon')
geoshow(amwlat, amwlon, 'Marker', 'o', 'Color', 'red')
t = ['Looking down at (' angl2str(amwlat,'ns') ...
    ',' angl2str(amwlon,'ew') ')'];
title(t)
```

Looking down at ( $42.30^{\circ}$  N ,  $71.35^{\circ}$  W)      Looking down at ( $42.30^{\circ}$  S ,  $108.65^{\circ}$  E)



# arcgridread

---

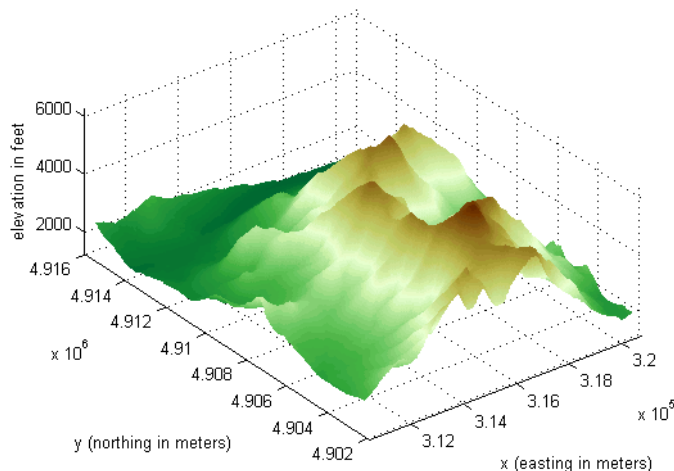
**Purpose** Read gridded data set in Arc ASCII Grid Format

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

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

**Example**

```
[Z,R] = arcgridread('MtWashington-ft.grd');  
mapshow(Z,R,'DisplayType','surface');  
xlabel('x (easting in meters)'); ylabel('y (northing in meters)')  
colormap(demcmap(Z))  
  
% View the terrain in 3-D  
axis normal; view(3); axis equal; grid on  
zlabel('elevation in feet')
```



**See Also** `makereformat`, `mapshow`, `sdtisdemread`

**Purpose**

Surface area of polygon on sphere or ellipsoid

**Syntax**

```
area = areaint(lats,longs)
area = areaint(lats,longs,ellipsoid)
area = areaint(lats,longs,ellipsoid,units)
```

`area = areaint(lats,longs)` returns the surface area enclosed by the polygon defined by the column vectors `lats` and `longs`. Multiple polygons can be delineated by NaNs. The output area is a fraction of the unit sphere's area of  $4\pi$ , so the result ranges from 0 to 1.

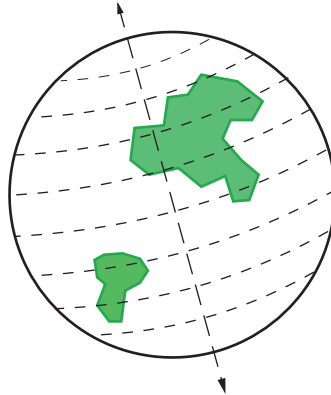
`area = areaint(lats,longs,ellipsoid)` allows the specification of the ellipsoid model with the two-element ellipsoid vector `ellipsoid`. When an ellipsoid is input, the resulting area is given in terms of the (squared) units of the ellipsoid. For example, if the ellipsoid `almanac('earth','ellipsoid','kilometers')` is used, the resulting area is in  $\text{km}^2$ . The default ellipsoid is the unit sphere.

`area = areaint(lats,longs,ellipsoid,units)` specifies the units of the inputs `lats` and `longs`, which are 'degrees' by default.

**Description**

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

Areas are computed for arbitrary polygons on the ellipsoid or sphere



An area is returned for each NaN-separated polygon

## Examples

Consider the area enclosed by a  $30^\circ$  lune from pole to pole and bounded by the prime meridian and  $30^\circ\text{E}$ . You can use the function `areaquad` to get an exact solution:

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

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

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

Now, a little finer, perhaps one point every  $1^\circ$  of latitude:

```
lats = [-90:1:90,89:-1:-89]';
```



```
longs = [zeros(1,181),30*ones(1,179)]';  
area = areaint(lats,longs)  
area =  
    0.0833
```

**Limitations**

As noted above, this is a line integral estimation, only as good as the accuracy and the density of the polygon vertex data. However, given sufficient data, the `areaint` function is the best method for determining the areas of complex polygons, such as continents, cloud cover, and other natural or derived features. The calculations in this function employ a spherical Earth assumption. For nonspherical ellipsoids, the latitude data is converted to the auxiliary authalic sphere.

**See Also**

`almanac`, `areamat`, `areaquad`

## Purpose

Surface area covered by nonzero values in binary data grid

## Syntax

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

`a = areamat(BW,refvec)` returns the surface area covered by the elements of the binary regular data grid `BW`, which contain the value 1 (true). `BW` can be the result of a logical expression such as `BW = (topo > 0)`. `refvec` is a 1-by-3 referencing vector containing elements [cells/degree north-latitude west-longitude] with latitude and longitude limits in degrees. The output `a` expresses surface area as a fraction of the surface area of the unit sphere ( $4\pi$ ), so the result ranges from 0 to 1.

`a = areamat(BW,refvec,ellipsoid)` uses the input `ellipsoid` vector to describe the sphere or reference ellipsoid. `ellipsoid` has the form [semi-major-axis-length, eccentricity]. The units of the output, `a`, are the square of the length units in which the semi-major axis is provided. For example, if `ellipsoid` is replaced with `almanac('earth','wgs84','kilometers')`, then `a` is in square kilometers.

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

## Remarks

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



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

## Examples

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

area =
    0.2411
```

Approximately 24% of the Earth has an altitude greater than 127 meters. What is the surface area of this portion of the Earth in square kilometers if a spherical ellipsoid is assumed? (Use the `almanac` function with the sphere as its reference body.)

```
earthgeoid = almanac('earth','ellipsoid','km','sphere');
area = areamat((topo>127),topolegend,earthgeoid)

area =
    1.2299e+08
```

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

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

area =
    1.0000
```

## areamat

---

```
cellarea =  
    0.0017  
    0.0048  
    0.0074  
    0.0091  
    0.0096  
    0.0091  
    0.0074  
    0.0048  
    0.0017
```

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

### See Also

`almanac`, `areaint`, `areaquad`

**Purpose**

Surface area of latitude-longitude quadrangle

**Syntax**

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

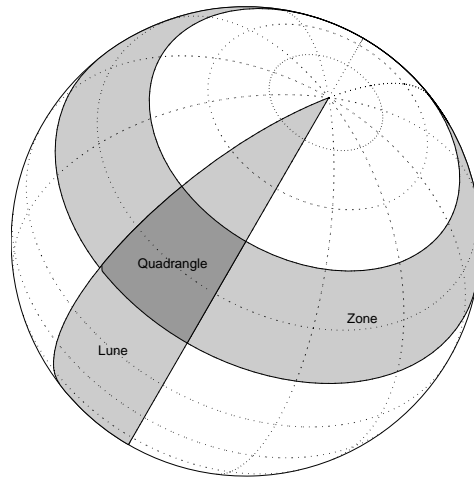
`area = areaquad(lat1,lon1,lat2,lon2)` returns the surface area bounded by the parallels `lat1` and `lat2` and the meridians `lon1` and `lon2`. The output `area` is a fraction of the unit sphere's area of  $4\pi$ , so the result ranges from 0 to 1.

`area = areaquad(lat1,lon1,lat2,lon2,ellipsoid)` allows the specification of the ellipsoid model with the two-element ellipsoid vector `ellipsoid`. When a `ellipsoid` is input, the resulting area is given in terms of the (squared) units of the ellipsoid. For example, if the ellipsoid `almanac('earth','ellipsoid','kilometers')` is used, the resulting area is in  $\text{km}^2$ . The default ellipsoid is the unit sphere.

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

**Description**

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



## Examples

What fraction of the Earth's surface lies between 30°N and 45°N, and also between 25°W and 60°E?

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

```
area =  
0.0245
```

About 2.5%. What is the surface area of the Earth in square kilometers if a spherical ellipsoid is assumed (use the almanac function with the sphere as its reference body)?

```
earthellipsoid = almanac('earth', 'ellipsoid', 'km', 'sphere');  
area = areaquad(-90, -180, 90, 180, earthellipsoid)
```

```
area =  
5.1006e+08
```

For comparison,

```
almanac('earth', 'surfarea', 'km')
```

```
ans =  
5.1006e+08
```

## Remarks

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

## See Also

almanac, areaint, areamat

# avhrrgoode

---

## Purpose

Read AVHRR data product stored in Goode Projection

## Syntax

```
[latgrat,longrat,z] = avhrrgoode  
avhrrgoode(region)  
avhrrgoode(region,filename)  
avhrrgoode(region,filename,scalefactor)  
avhrrgoode(region,filename,scalefactor,latlim,lonlim)  
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize)  
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,  
    fnrows,fncols)  
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,  
    fnrows,fncols,resolution)  
avhrrgoode(region,filename,scalefactor,latlim,lonlim,gsize,  
    fnrows,fncols,resolution,precision)
```

[*latgrat*,*longrat*,*z*] = avhrrgoode reads data from an AVHRR data set with a nominal resolution of 1 km. These files have 17347 rows and 40031 columns of data, or somewhat more than the capacity of one CD-ROM. The file is selected interactively. Data is returned as a general data grid with the graticule matrices in units of degrees.

avhrrgoode(*region*) reads data from a file with data covering the specified *region*. Valid regions are 'g' or 'global', 'af' or 'africa', 'ap' or 'australia/pacific', 'ea' or 'eurasia', 'na' or 'north america', and 'sa' or 'south america'. The file is selected interactively. If omitted, 'global' is assumed.

avhrrgoode(*region*,*filename*) uses the provided *filename*.

avhrrgoode(*region*,*filename*,*scalefactor*) uses the integer *scalefactor* to downsample the data. A scale factor of 1 returns every point. A scale factor of 10 returns every 10th point. If omitted, 100 is assumed.

avhrrgoode(*region*,*filename*,*scalefactor*,*latlim*,*lonlim*) returns data for the specified region. The returned data will extend somewhat beyond the requested area. If omitted, the entire area covered by the data file is returned. The limits are two-element vectors in units of



degrees, with `latlim` in the range [ 90 90] and `lonlim` in the range [ 180 180].

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

`avhrrgoode(region, filename, scalefactor, latlim, lonlim, gsize, fnrows, fncols)` overrides the standard file format for the selected region. This is useful for data stored on CD-ROM, which might have been truncated to fit. Some data was distributed with 16347 rows and 40031 columns of data on CD-ROMs. Nondimensional vegetation index data at 8 km spatial resolution has 2168 rows and 5004 columns.

`avhrrgoode(region, filename, scalefactor, latlim, lonlim, gsize, fnrows, fncols, resolution)` reads a data set with the spatial resolution specified in meters. If omitted, the full resolution of 1000 meters is assumed. Data is also available at 8000 meter resolution.

`avhrrgoode(region, filename, scalefactor, latlim, lonlim, gsize, fnrows, fncols, resolution, precision)` reads a data set with the integer `precision` specified. If omitted, 'uint8' is assumed. 'uint16' is appropriate for some files. Check the metadata (.txt or README) file in the ftp directory for specification of the file format and contents.

## Background

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

Goode projection with global and continental coverage at 1 km. It can also read 8 km data with global coverage.

## Remarks

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

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

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

Read a 1 km Global Land Cover Classification (GLCC) file using the default parameters. Select the file 'gusgs1\_2.img' interactively.

```
[latgrat,longrat,z] = avhrrgoode;
```

Read the same file at full resolution for just the island of Cyprus.

```
[latgrat,longrat,z] = avhrrgoode('g','gusgs1_2.img',1,...  
    [34.2 35.9],[32 35]);
```

Read the GLCC urban areas file covering North America in the Goode projection for just the area of eastern Massachusetts.

```
[latgrat,longrat,z] = avhrrgoode('north america',...  
    'naurban.img',1,[41.13 42.75],[-71.7 -69.8]);
```

Read the global data on the “Global Land 1-km AVHRR Data Set – Vegetation Index 6/21-30, 1992” CD-ROM (distributed by the Land Processes Distributed Active Archive Center, EROS Data Center, Sioux Falls, South Dakota, 57198, USA). Sample every 100th point for the entire globe, returning one lat and long for value. Provide the nonstandard number of rows and columns in the file.

```
[latgrat,longrat,z] = avhrrgoode('global','NDVI.IMG',...
```

```
100,[-90 90],[-180 180],[],16347,40031);
```

Read the global 8 km resolution nondimensional vegetation index. Sample every 10th point for the entire globe, returning one lat and long for value. Provide the nonstandard number of rows, columns, and resolution in the file.

```
[latgrat,longrat,z] = avhrrgoode('global',...
    'avhrrpf.ndvi.1ntfg1.940621',10,[-90 90],...
    [-180 180],[],2168,5004,8000);
```

Read the global 8 km resolution data for AVHRR sensor channel 4. Read at the full 8 km resolution for the island of Cyprus, returning one lat and long for value. Provide the nonstandard number of rows, columns, resolution, and integer precision in the file.

```
[latgrat,longrat,z] = avhrrgoode('global',...
    'avhrrpf.ch4.1ntfg1.840201',10,[34.2 35.9],...
    [32 35],[],2168,5004,8000,'uint16');
```

## Limitations

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

Subsets of the land cover data are available in both the Goode and the uninterrupted Lambert azimuthal projections. Data can be read more quickly from the Lambert projection using `avhrrlambert`.

This function does not have the proper projection parameters to read the regional 8 km resolution data sets.

## See Also

`avhrrlambert`

# avhrrlambert

---

## Purpose

Read AVHRR data product stored in eqaazim projection

## Syntax

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

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

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

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

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

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

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

## Background

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

## Remarks

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

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

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

### Example 1

Read and display every 100th point from the Global Land Cover Characteristics (GLCC) file covering North America with the USGS classification scheme, named `nausgs1_21.img`.

```
[latgrat, longrat, Z] = avhrrlambert('na','nausgs1_21.img');
```

Display the data using the Lambert Equal Area Azimuthal projection.

```
origin = [50 -100 0];  
ellipsoid = [6370997 0];  
figure  
axesm('MapProjection','eqaazim', 'Origin',origin, 'Geoid',ellipsoid)  
geoshow(latgrat, longrat, Z, 'DisplayType', 'texturemap');
```

### Example 2

Read and display every 50th point from the Global Land Cover Characteristics (GLCC) file covering Europe with the USGS classification scheme, named `eausgs1_21e.img`.

```
figure  
worldmap europe  
mstruct = gcm;  
latlim = mstruct.maplatlimit;  
lonlim = mstruct.maplonlimit;  
scalefactor = 50;
```

```
[latgrat, longrat, Z] = ... avhrrlambert('e', ...  
    'eausgs1_21e.img', scalefactor, latlim, lonlim);  
geoshow(latgrat, longrat, Z, 'DisplayType', 'texturemap');
```

Overlay the coastline.

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

## See Also

avhrrgoode

# axes2ecc

---

**Purpose** Eccentricity of ellipse with given axis lengths

**Syntax** `eccentricity = axes2ecc(semimajor,semiminor)`  
`eccentricity = axes2ecc(axes)`

**Description** Eccentricity, the second element of the standard ellipsoid vector in mapping toolbox, can be determined given both the semimajor and semiminor axes.

`eccentricity = axes2ecc(semimajor,semiminor)` returns the eccentricity associated with the input axes.

`eccentricity = axes2ecc(axes)` allows the axes inputs to be packed into a single two-column input of the form `[semimajor, semiminor]`.

**Examples** Using the axes for the default GRS 80 Earth model,

```
ecc = axes2ecc(6378.1370,6356.7523)
ecc =
    0.08181921804834
```

This is the eccentricity returned by `almanac('earth','ellipsoid')`.

**See Also** `almanac`, `ecc2n`, `majaxis`, `minaxis`



---

<b>Purpose</b>	Define map axes and set map properties
<b>Syntax</b>	<pre>axesm axesm(<i>PropertyName</i>,<i>PropertyValue</i>,...) axesm(<i>ProjectionFcn</i>,<i>PropertyName</i>,<i>PropertyValue</i>,...)</pre> <p>axesm with no input arguments, initiates the axesmui map axes graphical user interface, which can be used to set map axes properties. This is detailed on the axesmui reference page.</p> <p>axesm(<i>PropertyName</i>,<i>PropertyValue</i>,...) creates map axes with the specified property values. The MapProjection property must be the first input pair.</p> <p>axesm(<i>ProjectionFcn</i>,<i>PropertyName</i>,<i>PropertyValue</i>,...) allows omission of the MapProjection property name. The first input must be the identifying string of an available projection. For a list of these strings, see the table of projections in “Summary and Guide to Projections” on page 8-64.</p>
<b>Description</b>	The axesm function creates a map axes object complete with a map data structure. Maps must be displayed in map axes. All standard MATLAB axes properties of map axes are controlled by the axes function, along with set and get. Map axes properties are defined on creation with axesm and can be queried and changed after creation of a map axes using getm and setm.
<b>Axes Definition</b>	<p>Map axes are standard MATLAB axes with different default settings for some properties and a data structure for storing projection parameters and other data. The main differences in default settings are</p> <ul style="list-style-type: none"><li>• Axes properties XGrid, YGrid, XTick, YTick are set to 'off'.</li><li>• The properties XColor, YColor, and ZColor are set to the background color.</li><li>• The hold mode is on.</li></ul>

The map data structure stores the map axes properties, which, in addition to the special standard axes settings described here, allow Mapping Toolbox to recognize an axes or an opened FIG-file as a map axes. See “Map Axes Object Properties” on page 11-32, below, for descriptions of the map axes properties.

## Examples

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

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

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

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

A projection file includes default data for all properties. Any following property name/property value pairs are treated as overrides.

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

---

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

---

## Map Axes Object Properties

- “Properties That Control the Map Projection” on page 11-33
- “Properties That Control the Frame” on page 11-38
- “Properties That Control the Grid” on page 11-40
- “Properties That Control Grid Labeling” on page 11-44

## Properties That Control the Map Projection

### AngleUnits

{degrees} | radians

*Angular unit of measure* — Controls the units of measure used for angles (including latitudes and longitudes) in the map axes. All input data are assumed to be in the given units; 'degrees' is the default. For more information on angle units, see “Working with Angles: Units and Representations” on page 3-18 in the *Mapping Toolbox User’s Guide*.

### Aspect

{normal} | transverse

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

### FalseEasting

scalar {0}

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

FalseNorthing  
scalar {0}

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

FixedOrient  
scalar {[]} (read-only)

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

Geoid  
[semimajor\_axis eccentricity]

*Planet ellipsoid definition* — Sets the ellipsoid for calculating the projections of any displayed map objects. In Mapping Toolbox, the ellipsoid is approximated by a spheroid. The default ellipsoid is a sphere with a radius of 1. This is represented as [1 0]. Any semimajor axis, in any distance units, can be entered; eccentricity lies between 0 and 1.

MapLatLimit  
[south north] | [north south]

*Latitude limits of the displayed map* — Sets the north and south latitude limits of the map data. This information is useful for two

purposes. The default extents for the texture mapping functions `meshm`, `surfm`, `surfacem`, `surflm`, and `pcolorm` are set for the map axes; if the latitude limits match the actual data grid data limits, no graticule definitions are required when calling the above functions. Secondly, establishing map latitude limits sets the absolute limit on the extent of displayed meridians, regardless of the values of the meridian limits or the meridian exceptions. For nonazimuthal projections in the normal aspect, the map limits are truncated to the smaller of the map and frame limits. The default map latitude limits for most projections are at the poles, `[-90 90]`.

`MapLonLimit`  
`[west east]`

*Longitude limits of the displayed map* — Sets the east and west longitude limits of the map data. This information is useful for two purposes. The default extents for the texture mapping functions `meshm`, `surfm`, `surfacem`, `surflm`, and `pcolorm` are set for the map axes; if the longitude limits match the actual data grid data limits, no graticule definitions are required when calling the above functions. Secondly, establishing map longitude limits sets the absolute limit on the extent of displayed parallels, regardless of the values of the parallel limits or the parallel exceptions. For nonazimuthal projections in the normal aspect, the map limits are truncated to the smaller of the map and frame limits. The default map longitude limits for most projections are at the International Date Line, `[-180 180]`.

`MapParallels`  
`[lat] | [lat1 lat2]`

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

parallels are set to 15°N and 75°N, which biases the projection toward the northern hemisphere.

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

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

#### MapProjection

`projection_name` {no default}

*Map projection* — Sets the projection, and hence all transformation calculations, for the map axes object. It is required in the creation of map axes. The projection name is a string corresponding to an M-file appropriate to the projection. It must be a member of the recognized projection set, which you can list by typing `getm('MapProjection')` or `maps`. For more information on projections, see the *Mapping Toolbox User's Guide*. Some projections set their own defaults for other properties, such as parallels and trim limits.

#### Origin

[latitude longitude orientation]

*Origin and orientation for projection calculations* — Sets the map origin for all projection calculations. The latitude, longitude, and orientation should be in the map axes `AngleUnits`. Latitude and longitude refer to the coordinates of the map origin; orientation refers to an angle of skewness or rotation about the axis running

through the origin point and the center of the earth. The default origin is 0° latitude and a longitude centered between the map longitude limits. If a scalar is entered, it is assumed to refer to the longitude; if a two-element vector is entered, the default orientation is 0°, a normal projection. If an empty origin vector is entered, the origin is centered on the map longitude limits. For more information on the origin, see the *Mapping Toolbox User's Guide*.

#### Parallels

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

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

#### ScaleFactor

scalar {1}

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

#### TrimLat

[south north] (read-only, projection-dependent)

*Latitude trimming for certain projections* — This read-only property indicates the limits in latitude beyond which no plotting is attempted. This property is set by the projection and is usually used to avoid blowups to infinity. For example, the Mercator projection trims all data outside the range [ 86 86]. TrimLat is [-90 90] for most projections.

TrimLon

[west east] (read-only, projection-dependent)

*Longitude trimming for certain projections* — This read-only property indicates the limits in longitude beyond which no plotting is attempted. This property is set by the projection and is usually used to avoid blowups to infinity. It is less commonly used than the latitude trimming and is [-180 180] for most projections.

Zone

ZoneSpec | {[ ] or 31N}

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

## **Properties That Control the Frame**

Frame

on | {off}

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

FFill

*scalar plotting point density* {100}

*Frame plotting precision* — Sets the number of points to be used in plotting the frame for display. The default value is 100, which



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

**FEdgeColor**

ColorSpec | {[0 0 0]}

*Color of the displayed frame edge* — Specifies the color used for the displayed frame. You can specify a color using a vector of RGB values or one of the MATLAB predefined names. By default, the frame edge is displayed in black ([0 0 0]).

**FFaceColor**

ColorSpec | {none}

*Color of the displayed frame face* — Specifies the color used for the displayed frame face. You can specify a color using a vector of RGB values or one of the MATLAB predefined names. By default, the frame face is 'none', meaning no face color is filled in. Another useful color is 'cyan' ([0 1 1]), which looks like water.

**FLatLimit**

[south north] | [north south]

*Latitude limits of the base projection frame* — Sets the north and south latitude limits of the map frame. Latitudes refer to the base projection, and are [ 90 90] by default for most projections. The frame latitude limits determine where data is trimmed in a north-south sense. Data lying outside the latitude limits is not displayed. These limits also determine the latitude positions of the frame for its own display.

For nonazimuthal projections in the normal aspect, the frame limits are truncated to the smaller of the map and frame limits. Frame limits for nonazimuthal projections are specified in

geographic coordinates with respect to the map origin specified in the `Origin` property. Frame latitude limits for azimuthal projections are specified by `-Inf` and a radius in polar coordinates with respect to the map origin specified in the `Origin` property.

`FLineWidth`  
`scalar {2}`

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

`FLonLimit`  
`[east west] | [west east]`

*Longitude limits of the base projection frame* — Sets the east and west longitude limits of the map frame. Longitudes refer to the base projection, and are `[ 180 180]` by default for most projections. The frame longitude limits determine where data is trimmed in an east-west sense. Data lying outside the longitude limits is not displayed. These limits also determine the longitude positions of the frame for its own display.

For nonazimuthal projections in the normal aspect, the frame limits are truncated to the smaller of the map and frame limits. Frame limits for nonazimuthal projections are specified in geographic coordinates with respect to the map origin specified in the `Origin` property. Frame longitude limits are ignored for azimuthal projections.

### **Properties That Control the Grid**

`Grid`  
`on | {off}`

*Grid visibility* — Controls the visibility of the display grid. When the grid is `'off'` (the default), the grid is not displayed. When

the grid is 'on', meridians and parallels are visible. The grid is plotted as a set of line objects.

#### GAltitude

scalar z-axis value {Inf}

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

#### GColor

ColorSpec | {[0 0 0]}

*Color of the displayed grid* — Specifies the color used for the displayed grid. You can specify a color using a vector of RGB values or one of the MATLAB predefined names. By default, the map grid is displayed in black ([0 0 0]).

#### GLineStyle

LineStyle {:}

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

#### GLineWidth

scalar {0.5}

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

#### MLineException

vector of longitudes {[]}

*Exceptions to grid meridian limits* — Allows specific meridians of the displayed grid to extend beyond the grid meridian limits to the poles. The value must be a vector of longitudes in the appropriate

angle units. For longitudes so specified, grid lines extend from pole to pole regardless of the existence of any grid meridian limits. This vector is empty by default.

**MLineFill**

scalar plotting point density {100}

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

**MLineLimit**

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

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

**MLineLocation**

scalar interval or specific vector {30°}

*Grid meridian interval or specific locations* — Establishes the interval between displayed grid meridians. When a scalar interval is entered in the map axes **MLineLocation**, meridians are displayed, starting at 0° longitude and repeating every interval in both directions, which by default is 30°. Alternatively, you can enter a vector of longitudes, in which case a meridian is displayed for each element of the vector.

**PLineException**

vector of latitudes {[]}

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

#### PLineFill

scalar plotting point density {100}

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

#### PLineLimit

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

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

#### PLineLocation

scalar interval or specific vector {15°}

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

enter a vector of latitudes, in which case a parallel is displayed for each element of the vector.

### **Properties That Control Grid Labeling**

FontAngle

{normal} | italic | oblique

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

FontColor

ColorSpec | {black}

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

FontName

courier | {helvetica} | symbol | times

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

FontSize

scalar in units specified in FontUnits {9}

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

FontUnits

{points} | normalized | inches | centimeters | pixels

*Units used to interpret the FontSize property* — When set to normalized, the toolbox interprets the value of FontSize as a fraction of the height of the axes. For example, a normalized

FontSize of 0.1 sets the text characters to a font whose height is one-tenth of the axes' height. The default units (points) are equal to 1/72 of an inch.

FontWeight  
bold | {normal}

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

LabelFormat  
{compass} | signed | none

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

LabelRotation  
on | {off}

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

LabelUnits  
{degrees} | dm | dms | radians

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

<b>LabelUnits value</b>	<b>Label format</b>
'degrees'	decimal degrees
'dm'	degrees/decimal minutes
'dms'	degrees/minutes/decimal seconds
'radians'	decimal radians

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

MeridianLabel  
on | {off}

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

MLabelLocation  
scalar interval or vector of longitudes

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

MLabelParallel  
{north} | south | equator | scalar latitude

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



'south' is specified, the minimum of the MapLatLimit is used. If 'equator' is specified, a latitude of 0° is used.

MLabelRound  
integer scalar {0}

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

ParallelLabel  
on | {off}

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

PLabelLocation  
scalar interval or vector of latitudes

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

PLabelMeridian  
east | {west} | prime | scalar longitude

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

`PLabelRound`  
integer scalar {0}

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

### See Also

`axes` (MATLAB function), `gcm`, `getm`, `setm`

**Purpose** Resize axes for equivalent scale

**Syntax**

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

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

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

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

**Examples**

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

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

% Create a figure for the conterminous states
f1 = figure; hconus = usamap('conus'); tightmap
geoshow(states, 'FaceColor', [0.5 1 0.5]);
framem off; gridm off; mlabel off; plabel off
load conus gtlakelat gtlakelon
geoshow(gtlakelat, gtlakelon,...
    'DisplayType', 'polygon', 'FaceColor', 'cyan')
gridm off;
```

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

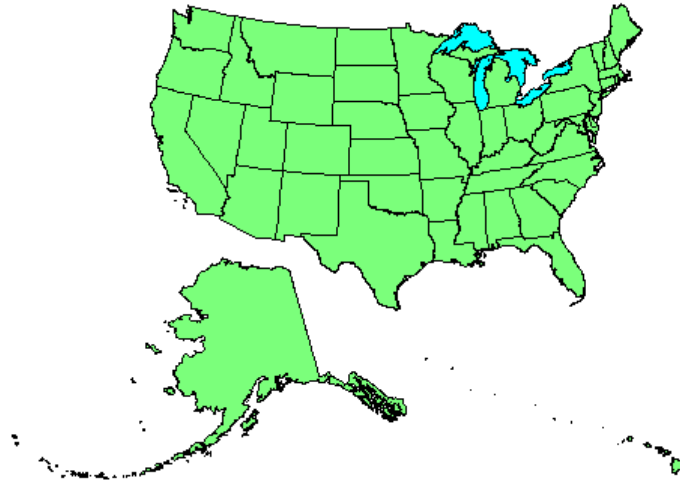
halaska = axes; usamap('alaska'); tightmap;
geoshow(alaska, 'FaceColor', [0.5 1 0.5]);
gridm off;
framem off; mlabel off; plabel off; gridm off;
set(halaska,'Parent',f1)

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

close(f2)

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

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

**Limitations**

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

**Remarks**

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

**See Also**

`paperscale`

# azimuth

---

## Purpose

Azimuth between points on sphere or ellipsoid

## Syntax

```
az = azimuth(pt1,pt2)
```

```
az = azimuth(lat1,lon1,lat2,lon2)
```

```
az = azimuth(pt1,pt2,ellipsoid)
```

```
az = azimuth(pt1,pt2,units)
```

```
az = azimuth(track,pt1,...)
```

`az = azimuth(pt1,pt2)` calculates the great circle azimuths from `pt1` to `pt2`. These two-column matrices should be of the form [latitude longitude].

`az = azimuth(lat1,lon1,lat2,lon2)` performs the same calculation for two pairs of latitude and longitude matrices.

`az = azimuth(pt1,pt2,ellipsoid)` specifies the elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. The default ellipsoid model is a unit sphere, which is sufficient for most applications.

`az = azimuth(pt1,pt2,units)` specifies the standard angle unit string. The default value is 'degrees'.

`az = azimuth(track,pt1,...)` specifies whether great circle azimuths or rhumb line azimuths are desired. Great circle azimuths, the default, are indicated with the standard *track* string 'gc'. Rhumb line azimuths are indicated with the standard *track* string 'rh'.

## Background

Azimuths are the bearings, or directions, between pairs of points. Azimuths are measured as angles, clockwise from true north. The North Pole has an azimuth of 0° from every other point on the globe.

Azimuth can be calculated in two ways. For great circles, the azimuth is the angle made between true north and the great circle passing through the two points *at the first point*. For rhumb lines, the azimuth is the *constant* angle made between true north and the entire rhumb line passing through the two points. For more information on this distinction, see the *Mapping Toolbox User's Guide*.

**Examples**

Consider two points on the same parallel, for example, (10°N,10°E) and (10°N,40°E). The azimuth between these two points depends upon the *track* string selected. Using the pt1,pt2 notation, the two cases result in

```
az = azimuth('gc',[10,10],[10,40])
az =
    87.3360
az = azimuth('rh',[10,10],[10,40])
az =
    90
```

The great circle path begins on an azimuth north of east to take the shortest route to the second point; the rhumb line proceeds along the parallel, on a constant due east heading.

Rhumb lines and great circles coincide along meridians and the equator. Consider two points on the same meridian, say (10°N,10°E) and (40°N,10°E), this time using the lat1,lon1,lat2,lon2 notation:

```
az = azimuth(10,10,40,10) % great circle sense
az =
    0
az = azimuth('rh',10,10,40,10)
az =
    0
```

The azimuths are the same because the paths coincide.

**See Also**

distance, elevation, reckon, track, track1, track2

# bufferm

---

## Purpose

Buffer zones for latitude-longitude polygons

## Syntax

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

[latb,lonb] = bufferm(lat,lon,dist,*direction*) computes the buffer zone around a polygon. A buffer zone for a closed polygon is defined as the locus of points that are a certain distance in or out of the polygon. A buffer zone for an open polygon is the locus of points a certain distance out from the polygon. The polygon is specified as vectors of latitude and longitude in units of degrees. The distance is a scalar specified in degrees of arc along the surface. Valid direction strings are 'in' and 'out'. The result is returned as NaN-clipped vectors in units of degrees.

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

[latb,lonb] = bufferm(lat,lon,dist,*direction*,npts,*outputformat*) controls the format of the returned buffer zones. *outputformat* 'vector' returns NaN-clipped vectors. *outputformat* 'cutvector' returns NaN-clipped vectors with cuts connecting holes to the exterior of the polygon. *outputformat* 'cell' returns cell arrays in which each element of the cell array is a separate polygon. Each polygon can consist of an outer contour followed by holes separated with NaNs.

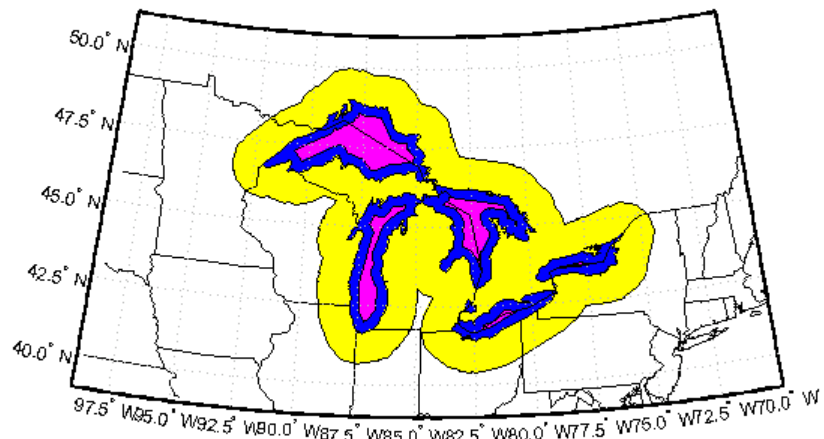
## Examples

Load the coordinates for the conterminous U.S. and its great lakes. Construct a 1-degree buffer zone around the great lakes, another buffer one-third of a degree wide inside the great lakes, and display the resulting buffers over the lake and state boundaries using geoshow:

```
load conus  
tol = 0.1; % Tolerance for simplifying polygon outlines
```



```
[reducedlat, reducedlon] = reducem(gtlakelat, gtlakelon, tol);  
dist = 1; % Buffer distance in degrees  
[latb, lonb] = bufferm(reducedlat, reducedlon, dist, 'out');  
[lati, loni] = bufferm(reducedlat, reducedlon, 0.3*dist, 'in');  
usamap({'MN','NY'})  
geoshow(latb, lonb, 'DisplayType', 'polygon',...  
         'FaceColor', 'yellow')  
geoshow(gtlakelat, gtlakelon,...  
         'DisplayType', 'polygon', 'FaceColor', 'blue')  
geoshow(lati, loni, 'DisplayType', 'polygon',...  
         'FaceColor', 'magenta')  
geoshow(uslat, uslon)  
geoshow(statelat, statelon)
```



**See Also**

polybool

**Purpose** Set camera position using geographic coordinates

**Syntax**

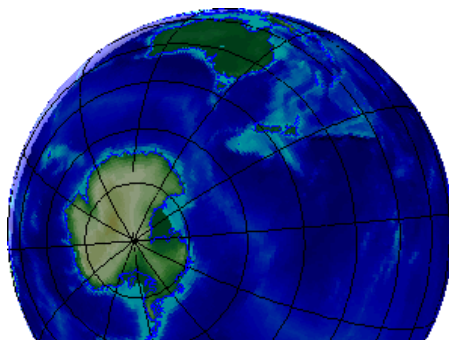
```
camposm(lat, long, alt)
[x, y, z] = camposm(lat, long, alt)
```

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

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

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

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



**See Also**

camtargm, camupm, campos, camva

# camtargm

---

**Purpose** Set camera target using geographic coordinates

**Syntax**

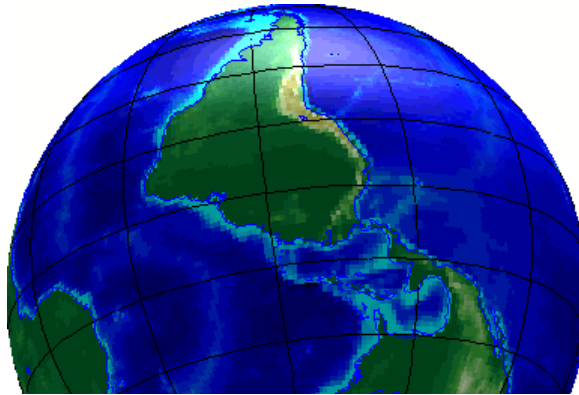
```
camtargm(lat, long, alt)
[x, y, z] = camtargm(lat, long, alt)
```

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

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

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

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



**See Also**

camposm, camupm, camtarget, camva

## Purpose

Set camera up vector using geographic coordinates

## Syntax

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

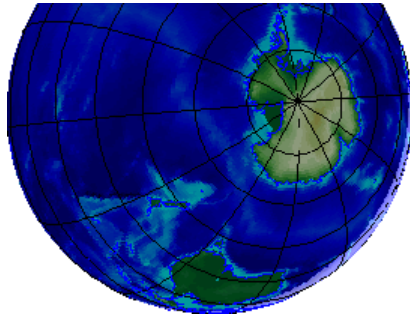
`camupm(lat, long)` sets the axes `CameraUpVector` property of the current map axes to the position specified in geographic coordinates. The inputs `lat` and `long` are assumed to be in the angle units of the current map axes.

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

## Examples

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

```
figure
axesm('globe', 'galt', 0)
gridm('glinestyle', '- -')
load topo
geoshow(topo, topolegend, 'DisplayType', 'texturemap');
demcmap(topo)
camlight;
material(0.6*[ 1 1 1])
plat = -50; plon = 160;
tlat = -10; tlon = 130;
[alat, alon] = antipode(tlat, tlon);
camtargm(tlat, tlon, 0);
camposm(plat, plon, 1);
camupm(alat, alon)
set(gca, 'CameraViewAngle', 80)
land = shaperead('landareas.shp', 'UseGeoCoords', true)
linem([land.Lat], [land.Lon])
axis off
```



**See Also**

cantargm, camposm, camup, camva

**Purpose** Transform projected coordinates to Greenwich system

**Syntax**

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

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

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

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

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

**See Also** gcm, mfwdtran, minvtran, project



**Purpose**           Substitute values in data array

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

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

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

**Examples**        Invent a map:

```
A = magic(3)
```

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

Replace instances of 8 or 9 with 0s:

```
B = changem(A,[0 0],[9 8])
```

```
B =
     0     1     6
     3     5     7
     4     0     2
```

# circirc

---

**Purpose** Intersections of circles in Cartesian plane

**Syntax** `[xout,yout] = circirc(x1,y1,r1,x2,y2,r2)`

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

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

**See Also** `linecirc`

**Purpose**

Add contour label map contour display

**Syntax**

```
h1 = clabelm(c,h)
h1 = clabelm(c,h,v)
h1 = clabelm(c,h,'manual')
h1 = clabelm(c), h1 = clabelm(c,v)
```

`h1 = clabelm(c,h)` rotates the labels and inserts them in line with the contour lines. The handles of the labels can be returned in `h1`.

`h1 = clabelm(c,h,v)` creates inline labels only for those levels specified in the vector `v`.

`h1 = clabelm(c,h,'manual')` places contour labels at locations you select with a mouse. You press the left mouse button (the only mouse button on a single-button mouse), or the space bar to label a contour at the closest location beneath the center of the cursor. Press the **Return** key while the cursor is within the figure window to terminate labeling. The labels are inserted in line with the contour lines.

`h1 = clabelm(c)`, `h1 = clabelm(c,v)`, and `h1 = clabelm(c,'manual')` operate as above, except that instead of rotating the labels and placing them in line with the contours, the labels are upright, and a + indicates the contour line the label is annotating.

**Description**

The `clabelm` function adds height labels to a two-dimensional contour plot. By default, `clabelm` labels all displayed contours and randomly selects label positions.

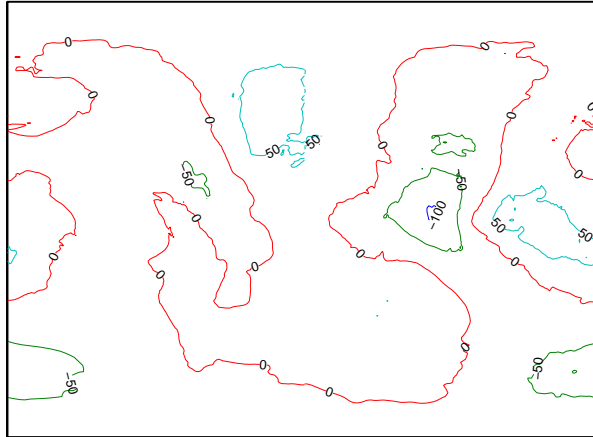
`c` is the contour matrix as described on the `contourm` reference page of this guide; `h` is the vector of handles for the displayed contours.

**Example**

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

# clabelm

---



## See Also

`clegendm`, `contourm`, `contour3m`, `clabel` (MATLAB function)

**Purpose** Add legend labels to map contour display

**Syntax**

```
clegendm(cs,h)
clegendm(cs,h,pos)
clegendm(...,unitstr)
clegendm(...,str)
```

**Description** The `clegendm` function displays a legend for a displayed contour map. `clegendm(cs,h)` displays a legend for the contour map defined by the two-column contour definition matrix, `cs`, and the handle(s) `h`. Both of these inputs are produced as the outputs of either `contourm` or `contour3m`.

`clegendm(cs,h,pos)` allows you to specify the position of the legend in the display. The input `pos` can be any of the following integers, with the indicated result:

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

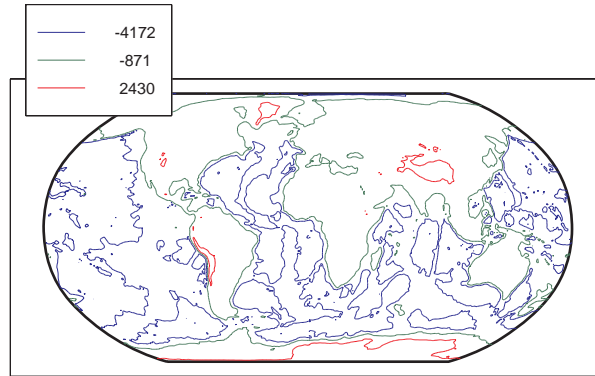
`clegendm(...,unitstr)` appends the character string `unitstr` to each entry in the legend.

`clegendm(...,str)` uses the strings specified in cell array `str` to label the legend. The cell array must have same number of entries as `h`.

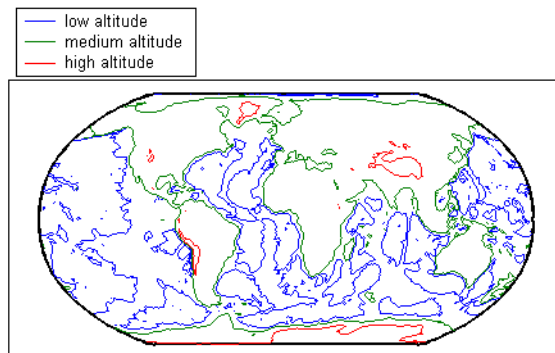
**Examples**

```
load topo
axesm robinson; framem
[cs,h] = contourm(topo,topolegend,3);
clegendm(cs,h,2)
```

# clegendm



```
% Example showing legend string usage
% Load topographic data measured in meters
load topo;
axesm robinson; framem
[cs,h] = contorm(topo,topolegend,3);
% Create Legend with user specified string
str = {'low altitude','medium altitude','high altitude'}
clegendm(cs,h,2,str);
```



**See Also**      clabelm, contourm, contour3m, contourc (MATLAB function)

# clipdata

---

**Purpose** Clip data at  $\pm \pi$  in longitude,  $\pm \pi$  in latitude

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

**Description** The input data must be in radians and properly transformed for the particular aspect and origin so that it fits in the specified clipping range. The output data is in radians, with clips placed at the proper locations. The output variable `splitpts` returns the row and column indices of the clipped elements (columns 1 and 2 respectively). These indices are necessary to restore the original data if the map parameters or projection are ever changed.

Allowable object strings are:

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

**See Also** `trimdata`, `undoclip`, `undotrim`



**Purpose**

Clear current map axes

**Syntax**

```
clma  
clma all  
clma purge
```

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

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

`clma purge` clears all displayed map objects and converts the map axes to standard axes. This is equivalent to `cla reset`.

**See Also**

`cla` (MATLAB function), `clmo`, `handlem`, `hidem`, `namem`, `showm`, `tagm`

# clmo

---

**Purpose** Clear specified graphics objects from map axes

**Syntax**

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

clmo deletes all displayed graphics objects on the current axes.

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

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

**See Also**

clma, handlem, hidem, namem, showm, tagm

**Purpose** Close all rings in multipart polygon

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

**Example**

```
xOpen = [1 0 2 NaN 0.5 0.5 1 1];
yOpen = [0 1 2 NaN 0.8 1 1 0.8];
[xClosed, yClosed] = closePolygonParts(xOpen, yOpen)
xClosed =
    Columns 1 through 7
    1.0000    0    2.0000    1.0000    NaN    0.5000    0.5000
    Columns 8 through 10
    1.0000    1.0000    0.5000

yClosed =
    Columns 1 through 7
    0    1.0000    2.0000         0    NaN    0.8000    1.0000
    Columns 8 through 10
    1.0000    0.8000    0.8000

whos
    Name          Size          Bytes  Class          Attributes

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

**See Also** `isshapemultipart`, `removeextrananseparators`

# cmapui

---

**Purpose** GUI to interactively generate colormaps

---

**Note** The `cmapui` function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox. To interactively construct a colormap, use `colormapeditor`. To set up a colormap appropriate for terrain displays, use the `demcmap` function. To generate an appropriate (but random) colormap for political maps, use the `polcmap` function.

---

**Syntax** `cmapui`

**Description** `cmapui` is a graphical user interface to create a colormap. The default size is five colors.

You select color slots in the colormap by clicking in the colorbar on the right side of the dialog. The current color slot is outlined in black. The color components for that color in HSV space are shown by the position of the dot in the color wheel and of the red bar in the value slider. To change the color, use the mouse to drag the dot and/or the red bar. To close the GUI and return the matrix of colors as RGB components, click the **Accept** button. Clicking **Cancel** closes the GUI and returns an empty matrix.

`cmapui` is a modal GUI. There is no access to the MATLAB command line while `cmapui` is active.

**Examples**

```
cmap = cmapui(20);  
cmap = cmapui(colorcube(10));
```

**See Also** `colormapeditor`, `demcmap`, `polcmap`

**Purpose** Interactively define RGB color

**Syntax**

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

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

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

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

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

**See Also** `uicolor`

# combntns

---

**Purpose** All possible combinations of set of values

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

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

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

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

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

```
combos =
```

```
    1    2    3
    1    2    4
    1    2    5
    1    3    4
    1    3    5
    1    4    5
    2    3    4
    2    3    5
    2    4    5
    3    4    5
```

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

```
ans =
```

10

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

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

```
combos =  
     2     2  
     2     5  
     2     5
```

**Remarks**

This is a recursive function.

# comet3m

---

**Purpose** Project 3-D comet plot on map axes

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

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

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

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

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

```
load coast
z = (1:length(lat))'/3000;
axesm miller
framem; gridm;
setm(gca,'galtitude',max(z)+.5)
view(3)
comet3m(lat,long,z,0.01)
```

**See Also** `comet3`, `cometm`



**Purpose**

Project 2-D comet plot on map axes

**Syntax**

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

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

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

**Description**

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

**Examples**

Create a comet plot of the coastlines data:

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

**See Also**

`comet`, `comet3m`

**Purpose** Project 3-D contour plot of map data

**Syntax**

```
contour3m(Z,R)
contour3m(lat,lon,Z)
contour3m(Z,R,n) or contour3m(lat,lon,Z,n)
contour3m(Z,V,R) or contour3m(lat,lon,Z,V)
contour3m(..., linespec)
contour3m(..., prop1, val1, prop2, val2,...)
C = contour3m(...)
[C,h] = contour3m(...)
```

`contour3m(Z,R)` displays a contour plot of the regular  $M$ -by- $N$  data grid,  $Z$ .  $R$  is a referencing matrix or referencing vector. If the current axis is a map axis, the coordinates of  $Z$  will be projected using the projection structure from the axis. The contours are drawn at their corresponding  $Z$  level.

`contour3m(lat,lon,Z)` displays a contour plot of the geolocated  $M$ -by- $N$  data grid,  $Z$ . `lat` and `lon` can be the size of  $Z$  or can specify the corresponding row and column dimensions for  $Z$ .

`contour3m(Z,R,n)` or `contour3m(lat,lon,Z,n)` where  $n$  is a scalar, draws  $n$  contour levels.

`contour3m(Z,V,R)` or `contour3m(lat,lon,Z,V)` where  $V$  is a vector, draws contours at the levels specified by the input vector  $v$ . Use  $V = [v \ v]$  to compute a single contour at level  $v$ .

`contour3m(..., linespec)` uses any valid LineSpec string to draw the contour lines.

`contour3m(..., prop1, val1, prop2, val2,...)` specifies property/value pairs that modify LINE graphics properties. Property names can be abbreviated and are case-insensitive.

`C = contour3m(...)` returns a standard contour matrix,  $C$ , with the first row representing longitude data and the second row representing latitude data.

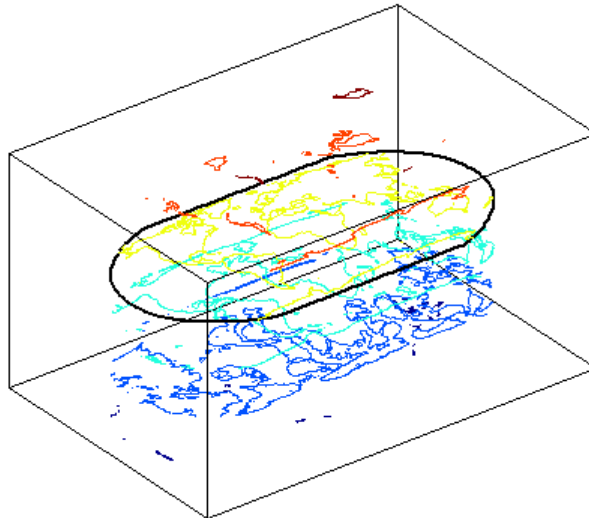
`[C,h] = contour3m(...)` returns the contour matrix and the line handles to the contour lines drawn onto the current axes.

## Examples

### Example 1

Make a default contour map of world topography data

```
load topo
axesm robinson; framem; view(3)
contour3m(topo,topolegend)
set(gca,'DataAspectRatio',[1 1 3000])
```

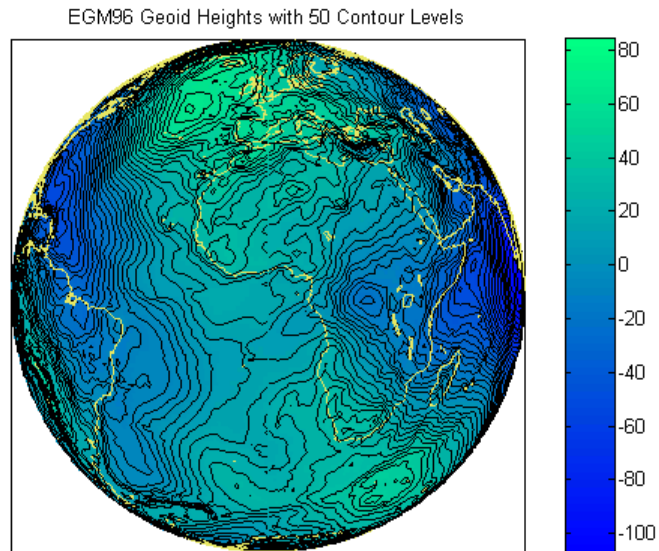


### Example 2

Contour EGM96 geoid heights as a 3-D surface with 50 levels, set contour patch edge color to black, show the geoid surface under and coastlines above the contour lines on an orthographic projection.

```
load geoid
axesm ortho
% Contour the geoid surface in black using 50 levels
[c,h]=contour3m(geoid, geoidrefvec, 50,'EdgeColor','black');
% Add the geoid surface.
hold on
```

```
geoshow(geoid,geoidrefvec,'DisplayType','surface')
% Add a title and colorbar.
title('EGM96 Geoid Heights with 50 Contour Levels');
colorbar
% Set the colormap to blue - green
colormap('winter')
% Set the Z-datum so that all contours show
zdatam(handlem('surface'),min(geoid(:)));
% Get world coastlines and plot them in gold
landareas = shaperead('landareas.shp','UseGeoCoords',true);
geoshow(landareas,'DisplayType','Polygon',...
        'FaceColor','None','EdgeColor',[.9 .9 .4])
```



### Example 3

Display the EGM96 geoid height contours in a default world map.

```

load geoid
figure
worldmap('world');

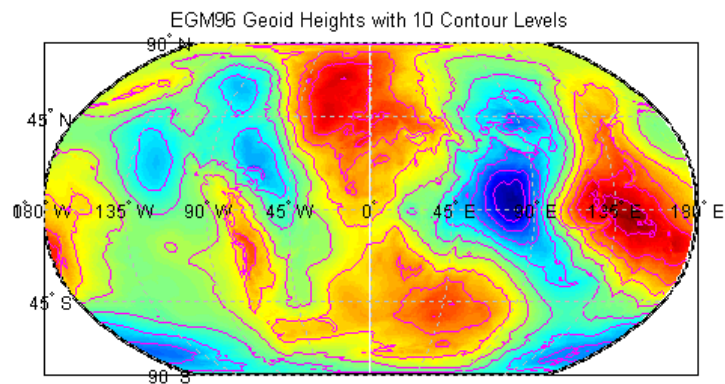
% Contour the geoid height with 10 levels and
% set the color to magenta.
[c,h]=contour3m(geoid, geoidrefvec, 10,'m');

% Add the geoid surface.
hold on
geoshow(geoid,geoidrefvec,'DisplayType','surface')

% Set the surface to the minimum height of the geoid.
% to keep the contours visible.
zdatam(handlem('surface'),min(geoid(:)));

% Add a title.
title('EGM96 Geoid Heights with 10 Contour Levels');

```



**See Also**

clabel, clabelm, clegendm, contour, contour3, contourm, geoshow, plot

**Purpose** Contour colormap and colorbar current axes

**Syntax**  
`contourcmap(cdelta,cmap)`  
`contourcmap(cdelta,cmap,property,value,...)`  
`hcb = contourcmap(...)`

`contourcmap(cdelta,cmap)` creates a contour colormap for the current axes. A contour colormap is a colormap with color changes aligned to the color data. If `cdelta` is a scalar, contours are generated at multiples of `cdelta`. If `cdelta` is a vector of evenly spaced values, contours are generated at those values. The string input `cmap` is the name of the colormap function used in the surface. Valid entries for `cmap` include 'pink', 'hsv', 'jet', or any similar colormap function.

`contourcmap(cdelta,cmap,property,value,...)` allows you to add a colorbar and control the colorbar's properties. You turn the colorbar on with the property-value pair 'Colorbar' and 'on'. The location of the colorbar is controlled by the 'Location' property. Valid entries for Location are 'vertical' (the default) or 'horizontal'. Properties 'TitleString', 'XLabelString', 'YLabelString' and 'ZLabelString' set the respective strings. Property 'ColorAlignment' controls whether the colorbar labels are centered on the color bands or the color breaks. Valid values for ColorAlignment are 'center' or 'ends'. Property 'SourceObject' controls which object is used to determine the color limits for the colormap. The SourceObject value is the handle of a currently displayed object. If omitted, gca is used. Other valid property-value pairs are any properties and values that can be applied to the title and labels of the colorbar axes.

`hcb = contourcmap(...)` returns a handle to the colorbar.

**Example** Create a colormap and set color limits to make the color changes occur at multiples of 20 for the geoid.

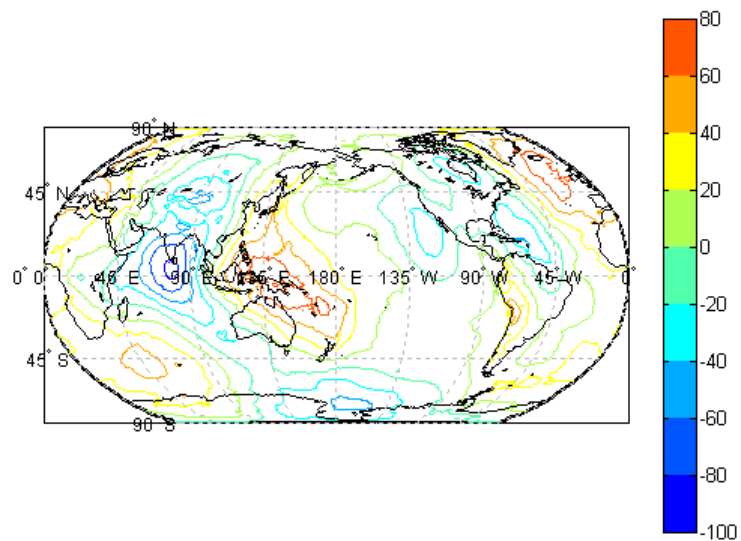
```
load geoid
figure
worldmap(geoid, geoidrefvec)
contourm(geoid, geoidrefvec, -120:20:100);
```

Add a colorbar, controlling the labels and font properties.

```
contourmap(20, 'jet', 'colorbar', 'on');
```

Load and plot coastlines on top.

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



## See Also

[contourfm](#), [contourm](#), [lcolorbar](#), [demcmap](#)

# contourfm

---

## Purpose

Project filled 2-D contour plot of map data

## Syntax

```
contourfm(lat,lon,Z)
contourfm(Z,R)
contourfm(lat,lon,Z,n,...)
contourfm(...,v,...)
contourfm(...,LineStyle)
c = contourfm(...)
[c,h] = contourfm(...)
```

`contourfm(lat,lon,Z)` produces a contour plot of map data projected onto the current map axes. The input latitude and longitude vectors can be the size of `Z` (as in a geolocated data grid), or can specify the corresponding row and column dimensions for the map.

`contourfm(Z,R)` creates a contour plot of the regular data grid, `Z`. `R` is a referencing matrix or referencing vector.

`contourfm(lat,lon,Z,n,...)` draws `n` contour levels, where `n` is a scalar.

`contourfm(...,v,...)` draws contours at the levels specified by the input vector `v`.

`contourfm(...,LineStyle)` uses any valid *LineStyle* string to draw the contour lines.

`c = contourfm(...)` returns a standard contour matrix, with the first row representing longitude data and the second row representing latitude data.

`[c,h] = contourfm(...)` returns the contour matrix and an array of handles to the contour patches drawn.

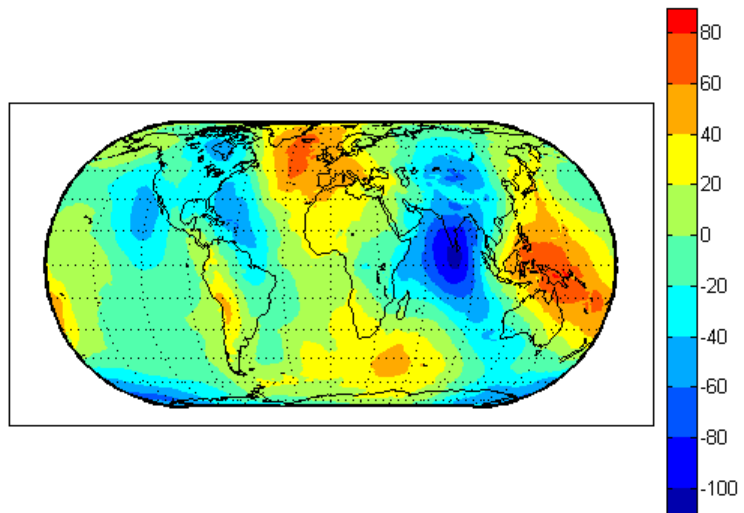
## Examples

Plot the Earth's geoid with filled contours. The data is in meters.

```
load geoid
figure
axesm eckert4
framem;gridm
```



```
load coast
plotm(lat,long,'k')
caxis([-120 100]);colormap(jet(11));colorbar
contourfm(geoid,geoidrefvec,-120:20:100);
```

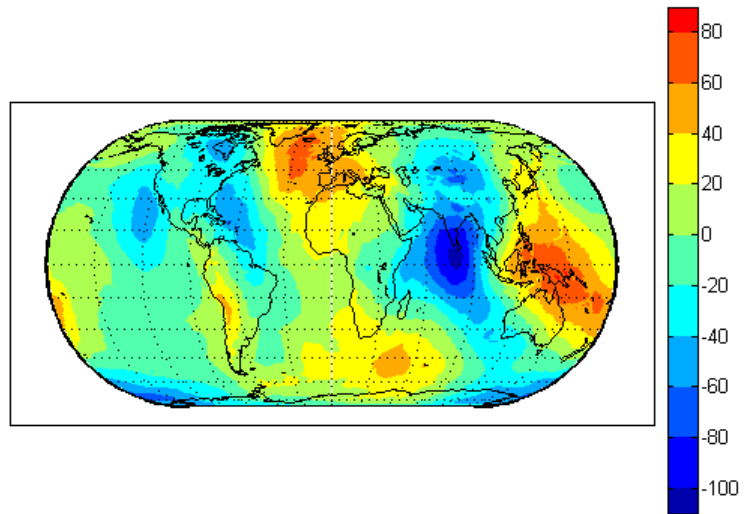


You can reproduce the filled contour display by using a surface instead of the patches created by `contourfm`.

```
figure
axesm eckert4
framem;gridm
load coast
plotm(lat,long,'k')
meshm(geoid,geoidrefvec,size(geoid),'Facecolor','interp')
contourcmap(20,'jet');colorbar
```

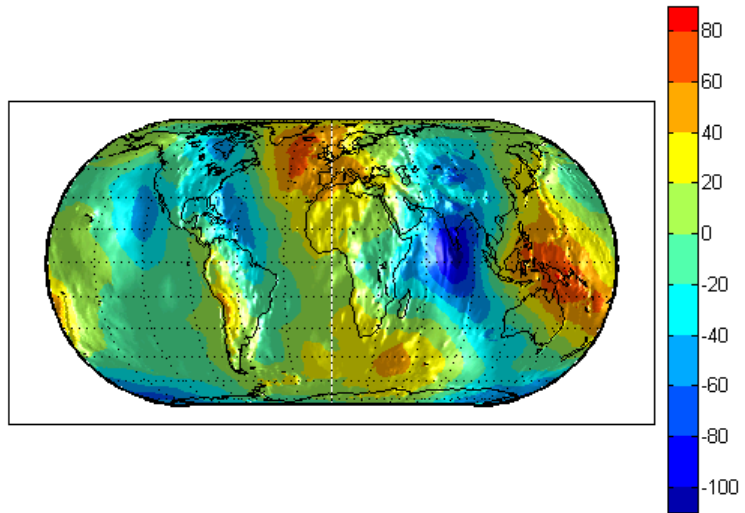
## contourfm

---



Surfaces also allow use of lighting to bring out the smaller variations in the data.

```
clmo surface
meshm(geoid,geoidrefvec,size(geoid),geoid,'Facecolor','interp')
light;lighting phong; material(0.6*[ 1 1 1])
set(gca,'dataaspectratio',[ 1 1 200])
gridm reset
zdatam(handlem('line'),max(geoid(:)))
```

**Limitations**

`contourfm` might not fill properly with azimuthal projections.

**Remarks**

By default, filled contour patches are displayed with no edge lines. To add contour lines, supply a `lineSpec` or specify an `EdgeColor` to `contourfm`. An `EdgeColor` may also be set later.

In most circumstances, contour plots made with surfaces are preferable to the filled patches created by `contourfm`. Surfaces are rendered more quickly and take less time to project and reproject. The use of surfaces also allows surface lighting to create shaded 3-D maps.

**See Also**

`contourm`, `contour3m`, `clabelm`, `meshm`, `surf`

# contourm

---

## Purpose

Project 2-D contour plot of map data

## Syntax

```
contourm(Z,R)
contourm(lat,lon,Z)
contourm(Z,R,n)
contourm(Z,R,V) or contourm(lat,lon,Z,V)
contourm(..., linespec)
contourm(..., prop1, val1, prop2, val2,...)
C = contourm(...)
[C,h] = contourm(...)
```

`contourm(Z,R)` creates a contour plot of the regular M-by-N data grid, Z. R is a referencing matrix or referencing vector. If the current axis is a map axis, the coordinates of Z will be projected using the projection structure from the axis. The contours are drawn at their corresponding Z level.

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

`contourm(Z,R,n)` or `contourm(lat,lon,Z,n)` where n is a scalar, draws n contour levels.

`contourm(Z,R,V)` or `contourm(lat,lon,Z,V)` where V is a vector, draws contours at the levels specified by the input vector v. Use `V = [v v]` to compute a single contour at level v.

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

`contourm(..., prop1, val1, prop2, val2,...)` specifies property/value pairs that modify `contourgroup` graphics properties. Property names can be abbreviated and are case-insensitive.

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

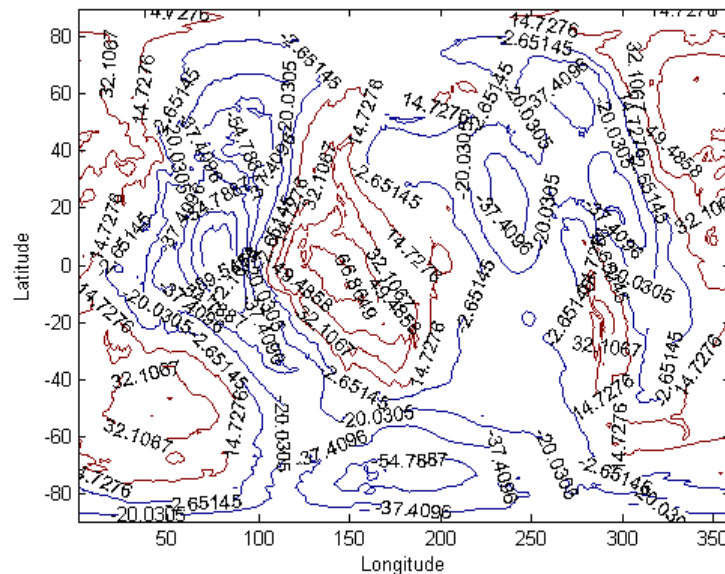
[C,h] = contourm(...) returns the contour matrix and the handle to the contour patches drawn onto the current axes. The handle is type hgroup.

## Examples

### Example 1

Contour EGM96 geoid heights as dotted lines and with 10 levels and set the contour labels on.

```
load geoid
figure
contourm(geoid, geoidrefvec, 10, ':','ShowText','on');
xlabel('Longitude');
ylabel('Latitude');
```



### Example 2

Contour the Korean bathymetry and elevation data:

# contourm

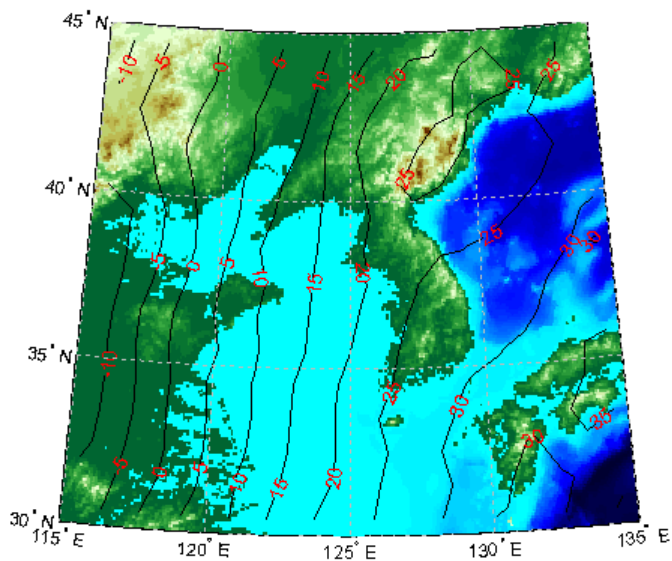
---

```
% Load the data.
load korea;
load geoid;

% Create a worldmap of Korea.
figure
worldmap(map, refvec);

% Display the digital elevation data and colormap.
geoshow(map, refvec, 'DisplayType', 'surface');
colormap(demcmap(map));
% Contour the geoid values from -100 to 100 in increments of 5.
[c,h] = contourm(geoid, geoidlegend, -100:5:100, 'k');

% Add red labels to the contours.
ht=clabel(c,h);
set(ht,'Color','r');
```



### Example 3

Contour the geoid and topography heights:

```
% Load the data.
load topo
load geoid

% Create a Miller projection with geoid contours as red lines,
% and topography contours as black lines.
figure; axesm miller
hold on
contourm(geoid, geoidrefvec, 'r');
contourm(topo, topolegend, 'k');

% Add the topography surface and color map.
geoshow(topo, topolegend, 'DisplayType', 'surface')
colormap(demcmap(topo))

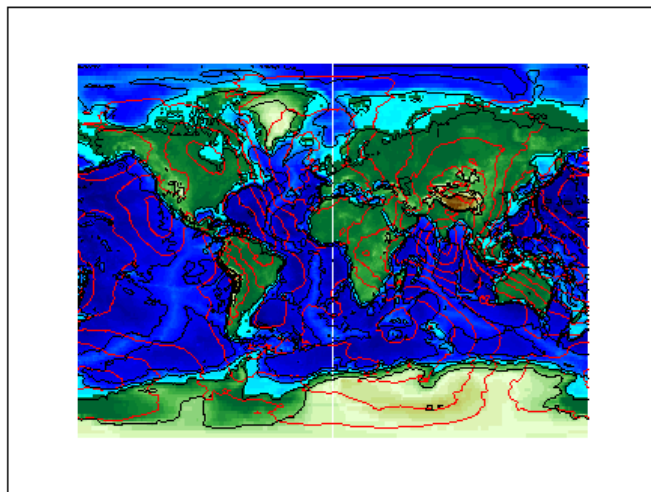
% Set the surface as the lowest value of topo
% to keep the contour lines visible.
zdatam(handlem('surface'), min(topo(:)))

% Add a title
title('Contour Plot of Topography and Geoid Heights');
```

# contourm

---

Contour Plot of Topography and Geoid Heights



## See Also

`clabelm`, `clegendm`, `contour`, `contourc`, `contour3`, `contour3m`,  
`geoshow`, `plot`



## Purpose

Convert between geodetic and auxiliary latitudes

## Syntax

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

`latout = convertlat(ellipsoid,latin,from,to,units)` converts latitude values in `latin` from type `FROM` to type `TO`. `ellipsoid` is a 1-by-2 ellipsoid vector of the form `[semimajoraxis eccentricity]`. (The `almanac` function offers a set of built-in ellipsoids covering most widely available map data.)

## Description

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

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

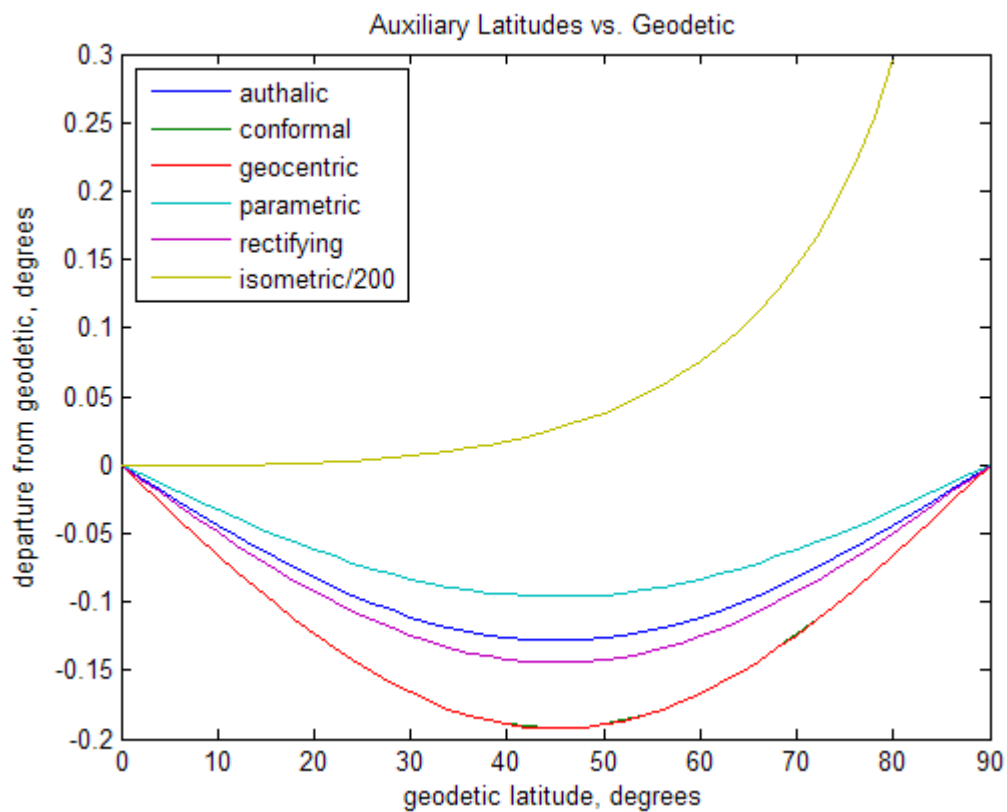
Latitude Type	Description
isometric	The isometric latitude is a nonlinear function of the geodetic latitude.
parametric	The parametric latitude of a point on the ellipsoid is the latitude on a sphere of radius $a$ , where $a$ is the semimajor axis of the ellipsoid, for which the parallel has the same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

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

## Example

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

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



**See Also** [almanac](#), [rsphere](#)

**Purpose**

Cross-fix positions from bearings and ranges

**Syntax**

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

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

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

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

```
[newlat,newlon] =
crossfix(lat,long,az,units), [newlat,newlon] =
```

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

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

## Description

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

## Example

Where do the small circles defined as all points 8° in distance from the points (0°,0°), (5°N,5°E), and (0°,10°E)" intersect?

```
figure('color','w');  
ha = axesm('mapproj','mercator', ...  
    'maplatlim',[-10 15], 'maplonlim',[-10 20],...  
    'MLineLocation',2, 'PLineLocation',2);  
axis off, gridm on, framem on;  
mlabel on, plabel on;  
latpts = [0;5;0];           % Define latitudes of three arbitrary points  
lonpts = [0;5;10];         % Define longitudes of three arbitrary points  
radii = [8;8;8];           % Define three radii, all 8 degrees  
  
% Obtain intersections of imagined small circles around these points  
[newlat, newlon] = crossfix(latpts, lonpts, radii, [0;0;0])  
  
% Draw red circle markers at the given points  
geoshow(latpts, lonpts, 'DisplayType', 'point', ...  
    'markeredgecolor', 'r', 'markerfacecolor', 'r', 'marker', 'o')
```

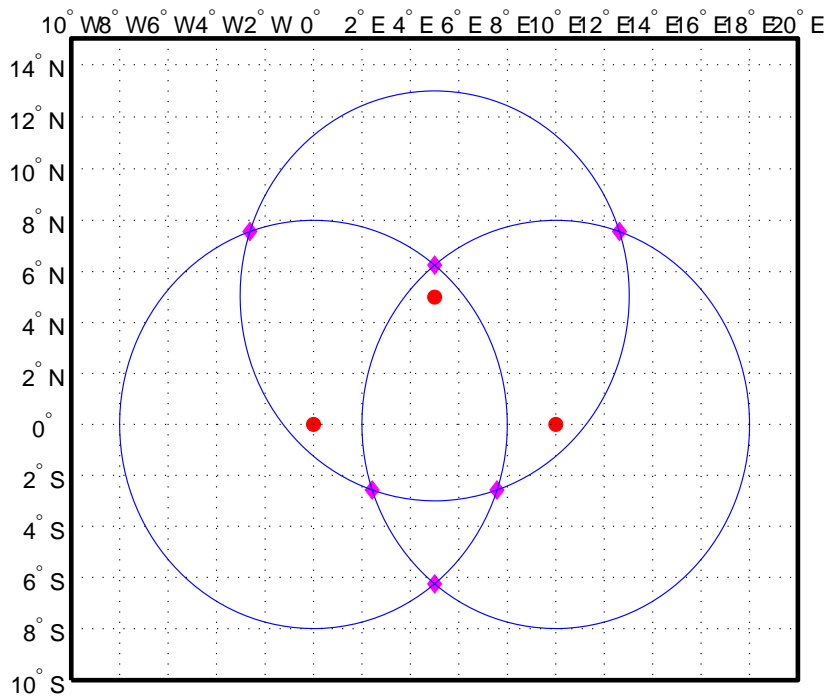
```
% Draw magenta diamond markers at intersection points just found
geoshow(reshape(newlat,6,1),reshape(newlon,6,1),'DisplayType','point',...
        'markeredgecolor','m','markerfacecolor','m','marker','d')

% Generate a small circle 8 deg radius for each original point
[latc1,lonc1] = scircle1(latpts(1),lonpts(1),radii(1));
[latc2,lonc2] = scircle1(latpts(2),lonpts(2),radii(2));
[latc3,lonc3] = scircle1(latpts(3),lonpts(3),radii(3));

% Plot the small circles to show the intersections are as determined
geoshow(latc1,lonc1,'DisplayType','line',...
        'color','b','linestyle','-')
geoshow(latc2,lonc2,'DisplayType','line',...
        'color','b','linestyle','-')
geoshow(latc3,lonc3,'DisplayType','line',...
        'color','b','linestyle','-')
```

The diagram shows why there are six intersections:

# crossfix



If a dead reckoning position is provided, say (0°,5°E), then one from each pair is returned (the closest one):

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

```
newlat =  
-2.5744  
6.2529  
-2.5744
```

```
newlong =  
7.5770  
5.0000
```



2.4230

**See Also**

gcxgc, gcxsc, scxsc, rhxrh, polyxpoly, navfix

# daspectm

---

## Purpose

Control vertical exaggeration in map display

## Syntax

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

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

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

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

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

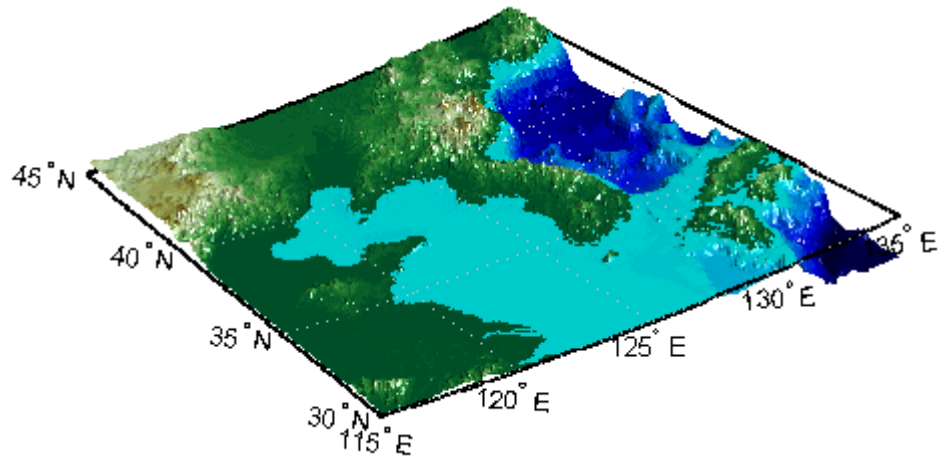
`daspectm(zunits,vfac,lat,long,az,gunits)` also specifies the units in which the geographic position and direction are given. If omitted, 'degrees' is assumed.

`daspectm(zunits,vfac,lat,long,az,gunits,radius)` uses the last input to determine the radius of the sphere. If *radius* is a string, then it is evaluated as an almanac body to determine the spherical radius. If numerical, it is the radius of the desired sphere in *zunits*. If omitted, the default radius of the Earth is used.

## Examples

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

```
load korea
[latlim,lonlim] = limitm(map,refvec);
worldmap(latlim,lonlim)
meshm(map,refvec,size(map),map)
demcmap(map)
view(3)
daspectm('m',30)
tightmap
camlight
```



### Limitations

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

### See Also

`daspect`, `paperscale`

## Purpose

Read selected DCW worldwide basemap data

## Syntax

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

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

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

```
[struct1, struct2,...] =  
dcwdata(...,{topolevel1,topolevel2,...})
```

reads several topology levels. The levels must be specified as a cell array with the entries 'patch', 'line', 'point', or 'text'. Entering {'all'} for the topology level argument is equivalent to {'patch', 'line', 'point', 'text'}. Upon output, the data structures are returned in the output arguments by topology level in the same order as they were requested.

## Background

The Digital Chart of the World (DCW) is a detailed and comprehensive source of publicly available global vector data. It was digitized from the Operational Navigation Charts (scale 1:1,000,000) and Jet Navigation

Charts (1:2,000,000), compiled by the U.S. Defense Mapping Agency (DMA) along with mapping agencies in Australia, Canada, and the United Kingdom. The digitized data was published on four CD-ROMS by the DMA and is distributed by the U.S. Geological Survey (USGS).

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

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

## Remarks

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

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

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

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

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

The DCW was published in 1992 based on data compiled some years earlier. The political boundaries do not reflect recent changes such as the dissolution of the Soviet Union, Czechoslovakia, and Yugoslavia. In

some cases, the boundaries of the successor nations are present as lower level political units. A new version, called VMAP0.

## Examples

On the Macintosh,

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

??? Error using ==> dcwdata
Theme not present in library NOAMER
Valid two-letter theme identifiers are:
PO: Political/Oceans
PP: Populated Places
LC: Land Cover
VG: Vegetation
RD: Roads
RR: Railroads
UT: Utilities
AE: Aeronautical
DQ: Data Quality
DN: Drainage
DS: Supplemental Drainage
HY: Hypsography
HS: Supplemental Hypsography
CL: Cultural Landmarks
OF: Ocean Features
PH: Physiography
TS: Transportation Structure
POpatch = dcwdata('NOAMER',[41 44],[-72 -69],'PO','patch')
POpatch =
1x234 struct array with fields:
    type
    otherproperty
    tag
    altitude
    lat
    long
    tag2
```

tag3

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

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

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

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

## References

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

## See Also

vmap0data, dcwgaz, dcwread, dcwrhead, extractm, mlayers, updategeostruct

## Purpose

Search DCW worldwide basemap gazette file

## Syntax

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

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

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

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

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

## Background

In addition to the geographic data, the Digital Chart of the World (DCW) also includes an extensive gazette feature. The gazette is a collection of the names of geographic items mentioned in the various themes of a DCW disk. One DCW disk can contain about 10,000 to 15,000 names. This function allows you to search the gazette for names beginning with a particular string.

## Remarks

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

## Examples

On the Macintosh,

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



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

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

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

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

# dcwgaz

---

## **See Also**

dcwdata, dcwread, dcwrhead, mlayers, updategeostruct

**Purpose**

Read DCW worldwide basemap file

**Syntax**

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

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

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

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

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

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

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

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

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

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

## Background

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

## Remarks

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

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

## Examples

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

```
s = dcwread('NOAMER:DCW:NOAMER:', 'GRT')
s =
    ID: 1
    DATA_TYPE: 'GEO'
    UNITS: '014'
    ELLIPSOID: 'WGS 84'
    ELLIPSOID_DETAIL: 'A=6378137,B=6356752 Meters'
    VERT_DATUM_REF: 'MEAN SEA LEVEL'
    VERT_DATUM_CODE: '015'
    SOUND_DATUM: 'MEAN SEA LEVEL'
    SOUND_DATUM_CODE: '015'
    GEO_DATUM_NAME: 'WGS 84'
    GEO_DATUM_CODE: 'WGE'
    PROJECTION_NAME: 'DECIMAL DEGREES'

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

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

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

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

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

# dcwread

---

```
AEPTDATE: '19900502000000000000'  
AEPTICAO: '1261'  
AEPTDKEY: 'BR17652'  
  TILE_ID: 94  
  END_ID: 1
```

```
s = dcwread('NOAMER:DCW:NOAMER:AE:', 'AEPOINT.PFT', {1,2})
```

```
s =
```

```
4678x1 struct array with fields:
```

```
  ID  
  AEPTTYPE
```

## See Also

dcwdata, dcwgaz, dcwrhead

---

<b>Purpose</b>	Read DCW worldwide basemap file headers
<b>Syntax</b>	<pre>dcwrhead dcwrhead(<i>filepath</i>,<i>filename</i>) dcwrhead(<i>filepath</i>,<i>filename</i>,<i>fid</i>) dcwrhead(...) str = dcwrhead(...)</pre> <p>dcwrhead allows the user to select the header file interactively.</p> <p>dcwrhead(<i>filepath</i>,<i>filename</i>) reads from the specified file. The combination [<i>filepath filename</i>] must form a valid complete filename.</p> <p>dcwrhead(<i>filepath</i>,<i>filename</i>,<i>fid</i>) reads from the already open file associated with <i>fid</i>.</p> <p>dcwrhead(...) with no output arguments displays the formatted header information on the screen.</p> <p>str = dcwrhead(...) returns a string containing the DCW header.</p>
<b>Background</b>	The Digital Chart of the World (DCW) uses header strings in most files to document the contents and format of that file. This function reads the header string, displays a formatted version in the command window, or returns it as a string.
<b>Remarks</b>	<p>This function reads all DCW files except index files (files with names ending in 'X'), thematic index files (files with names ending in 'TI'), and spatial index files (files with names ending in 'SI').</p> <p>File separators are platform dependent. The <i>filepath</i> input must use appropriate file separators, which you can determine using the MATLAB <code>filesep</code> function.</p>
<b>Examples</b>	<p>The following example uses the Macintosh file separators and pathname:</p> <pre>dcwrhead('NOAMER:DCW:NOAMER:AE:', 'AEPOINT.PFT') Aeronautical Points</pre>

```
AEPOINT.DOC
ID=I, 1,P,Row Identifier,-,-,
AEPTTYPE=I, 1,N,Airport Type,INT.VDT,-,
AEPTNAME=T, 50,N,Airport Name,-,-,
AEPTVAL=I, 1,N,Airport Elevation Value,-,-,
AEPTDATE=D, 1,N,Aeronautical Information Date,-,-,
AEPTICAO=T, 4,N,International Civil Organization Number,-,-,
AEPTDKEY=T, 7,N,DAFIF Reference Number,-,-,
TILE_ID=S, 1,F,Tile Reference Identifier,-,AEPOINT.PTI,
END_ID=I 1,F,Entity Node Primitive Foreign Key,-,-,
```

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

## See Also

dcwdata, dcwgaz, dcwread



**Purpose**

Initialize or reset projection properties to default values

**Syntax**

```
mstruct = defaultm
mstruct = defaultm(projection)
mstruct = defaultm(mstruct)
```

`mstruct = defaultm` creates an empty map projection structure.

`mstruct = defaultm(projection)` initializes the map structure for the specified map projection. *projection* is any valid projection string, such as 'sinusoid'.

`mstruct = defaultm(mstruct)` sets appropriate defaults based on existing parameter values in the map structure `mstruct`.

**Description**

The map projection structure contains all the information needed to project and display geographic data. It is often used indirectly via the map axes with which it is associated, but it can also be used directly to project data without displaying it.

**Examples**

Create an empty map projection structure for a Mercator projection:

```
mstruct = defaultm('mercator')

mstruct =
    mapprojection: 'mercator'
           zone: []
    angleunits: 'degrees'
           aspect: 'normal'
    falseeasting: []
    falsenorthing: []
           fixedorient: []
           geoid: [1 0]
    maplatlimit: []
    maplonlimit: []
    mapparallels: 0
           nparallels: 1
           origin: []
```

## defaultm

---

```
scalefactor: []
  trimlat: [-86 86]
  trimlon: [-180 180]
  frame: []
  ffill: 100
  fedgecolor: [0 0 0]
  ffacecolor: 'none'
  flatlimit: []
  flinewidth: 2
  flonlimit: []
  grid: []
  galtitude: Inf
  gcolor: [0 0 0]
  glinestyle: ':'
  glinewidth: 0.5000
mlineexception: []
  mlinefill: 100
  mlinelimit: []
  mlinelocation: []
  mlinevisible: 'on'
plineexception: []
  plinefill: 100
  plinelimit: []
  plinelocation: []
  plinevisible: 'on'
  fontangle: 'normal'
  fontcolor: [0 0 0]
  fontname: 'helvetica'
  fontsize: 9
  fontunits: 'points'
  fontweight: 'normal'
  labelformat: 'compass'
  labelrotation: 'off'
  labelunits: []
  meridianlabel: []
  mlabellocation: []
  mlabelparallel: []
```

```
mlabelround: 0
parallellabel: []
plabellocation: []
plabelmeridian: []
plabelround: 0
```

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

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

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

## defaultm

---

```
    galtitude: Inf
    gcolor: [0 0 0]
    glinestyle: ':'
    glinewidth: 0.500000000000000
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'
    labelrotation: 'off'
    labelunits: 'degrees'
meridianlabel: 'off'
mlabellocation: 30
mlabelparallel: 86
    mlabelround: 0
    parallellabel: 'off'
plabellocation: 15
plabelmeridian: -180
    plabelround: 0
```

### See Also

axesm, gcm, mfwdtran, minvtran, setm

**Purpose** Convert angles from degrees to deg:min or deg:min:sec encoding

---

**Note** The `deg2dm` and `deg2dms` functions are obsolete and error when used. Use `degrees2dm` or `degrees2dms` instead.

---

**Syntax**

```
angleOut = deg2dms(angleIn)
angleout = deg2dm(angleIn)
```

`angleOut = deg2dms(angleIn)` converts angles input in degrees to the equivalent measure in the degrees-minutes-seconds (*dms*) format.

`angleout = deg2dm(angleIn)` converts angles input in degrees to the equivalent measure in the degrees-minutes (*dm*) format. This is the *dms* format, properly rounded to just degrees and minutes.

**Example**

```
deg2dms(23.561)
ans =
    2333.40

deg2dm(23.561)
ans =
    2334.00
```

**See Also** `angledim`, `deg2rad`, `degrees2dm`

# deg2km, deg2nm, deg2sm

---

**Purpose** Convert distance from degrees to kilometers, nautical miles, or statute miles

**Syntax**

```
km = deg2km(deg)
nm = deg2nm(deg)
sm = deg2sm(deg)
km = deg2km(deg,radius)
nm = deg2nm(deg,radius)
sm = deg2sm(deg,radius)
km = deg2km(deg,sphere)
nm = deg2nm(deg,sphere)
sm = deg2sm(deg,sphere)
```

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

`nm = deg2nm(deg)` and `sm = deg2sm(deg)` work identically, except that the output units are nautical miles and statute miles, respectively.

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

For `nm = deg2nm(deg,radius)` and `sm = deg2sm(deg,radius)`, make sure your input radius is in the appropriate units.

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

`nm = deg2nm(deg,sphere)` and `sm = deg2sm(deg,sphere)` work identically, except that the output units are nautical miles and statute miles, respectively.

## Examples

A degree of arc length is about 60 nautical miles:

```
deg2nm(1)
```

```
ans =  
    60.0405
```

This is not true on Mercury, of course:

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

```
ans =  
    22.9852
```

## See Also

deg2nm, deg2rad, deg2sm, km2deg, sm2deg

# deg2rad

---

**Purpose** Convert angles from degrees to radians

**Syntax** `angleOut = deg2rad(angleIn)`  
`angleOut = deg2rad(angleIn)` converts angles input in degrees to the equivalent measure in radians.

**Remarks** This is both an angle conversion function and a distance conversion function, since arc length can be a measure of distance in either radians or degrees, provided that the radius is known.

**Example** Show that there are  $2\pi$  radians in a full circle:

```
2*pi - deg2rad(360)
```

```
ans =  
    0
```

**See Also** `degrees2dms`, `nm2km`, `sm2deg`, `rad2deg`



**Purpose** Convert degrees to degrees-minutes

**Syntax** `DM = degrees2dm(angleInDegrees)`

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

**Example**

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

dm =
    30.0000000000000    50.746335333336106
   -82.0000000000000     2.665137499997741
     0 -30.285390833333338
     0  0.247000000000020
```

**See Also** `dm2degrees`, `deg2rad` `degrees2dms`, `rad2deg`

# degrees2dms

---

**Purpose** Convert degrees to degrees-minutes-seconds

**Syntax** DMS = degrees2dms(angleInDegrees)

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

## Example

```
format long g
angleInDegrees = [ 30.8457722555556; ...
                  -82.0444189583333; ...
                  -0.504756513888889;...
                   0.004116666666667];

dms = degrees2dms(angleInDegrees)

dms =

           30           50          44.7801200001663
          -82            2          39.9082499998644
            0          -30          17.1234500000003
            0            0          14.8200000000012

% Convert angles to a string, with each angle on its own line.
nonnegative = all((dms >= 0),2);
hemisphere = repmat('N', size(nonnegative));
hemisphere(~nonnegative) = 'S';
absvalues = num2cell(abs(dms'));
values = [absvalues; num2cell(hemisphere')];
str = sprintf('%2.0fd%2.0fm%7.5fs%s\n', values{:})
```

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

% Split the string into cells as delimited by the newline
% character, then return to the original values using STR2ANGLE.
newline = sprintf('\n');
a = strread(str,'%s',-1,'delimiter',newline);
for k = 1:numel(a)
    str2angle(a{k})
end

ans =
    30.8457722555556

ans =
   -82.0444189583333

ans =
   -0.504756513888889

ans =
    0.00411666666666667
```

## See Also

dms2degrees, deg2rad degrees2dm, rad2deg

# demcmap

---

**Purpose** Colormaps appropriate to terrain elevation data

**Syntax**

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

`demcmap(Z)` creates and assigns a colormap for elevation data grid `Z`. The colormap has the number of land and sea colors in the same proportions as the maximum elevations and depths in the data grid. With no output arguments, the colormap is applied to the current figure and the color axis is set so that the interface between the land and sea is in the right place.

`demcmap(Z,ncolors)` makes a colormap with a length of `ncolors`. The default value is 64.

`demcmap(Z,ncolors,cmapsea,cmapland)` allows the default colors for sea and land to be replaced. The colors in the created colormap are interpolated from the RGB color matrix inputs, which can be of any length. You can retain default colors for either land or sea by providing an empty matrix in place of the color matrices. You can specify the current figure colormap by entering the string 'window' in place of either RGB matrix.

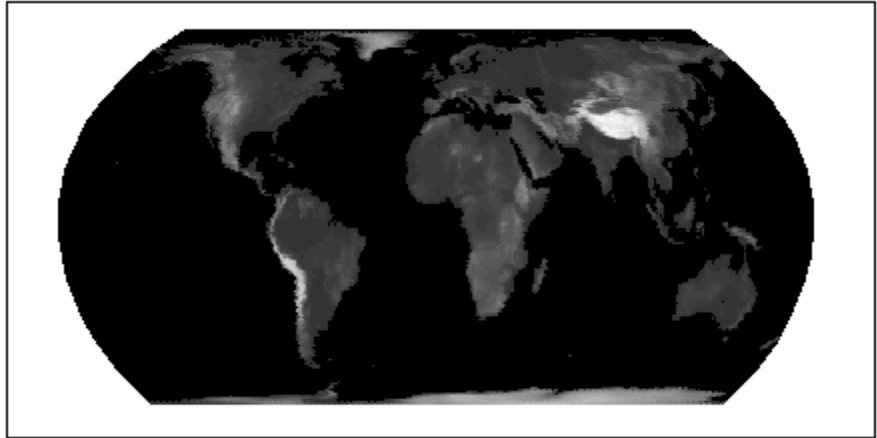
`demcmap(color,Z,spec)` uses the `color` string to define a colormap. If the string is set to 'size', `spec` is the length of the colormap. If it is set to 'inc', `spec` is the size of the altitude range assigned to each color. If omitted, `color` is 'size' by default.

`demcmap(color,Z,spec,cmapsea,cmapland)` allows for both coloring options along with specified colors.

**Examples** Display the world topographical map using grayscale colors:

```
load topo
axesm hatano
meshm(topo,topolegend)
```

```
demcmap(topo,64,[0 0 0],[.2 .2 .2; 1 1 1])
```

**See Also**

`caxis`, `colormap`, `meshlslrm`, `meshm`, `surflslrm`, `surfm`

# departure

---

## Purpose

Departure of longitudes at specified latitudes

## Syntax

```
dist = departure(long1,long2,lat)
dist = departure(long1,long2,lat,units)
dist = departure(long1,long2,lat,ellipsoid)
```

`dist = departure(long1,long2,lat)` returns the departure between two longitudes at a given latitude in degrees. Departure is dimensionless; the shorter of the two directions is taken from the first longitude to the second. The distance is given in degrees of arc length.

`dist = departure(long1,long2,lat,units)` specifies the valid angle units string to apply to the latitude, longitudes, and output distance.

`dist = departure(long1,long2,lat,ellipsoid)` specifies the elliptical definition of the Earth to be used with the two-element ellipsoid vector. The default ellipsoid model is a unit sphere, which is sufficient for most applications. When a ellipsoid model is input, the resulting distance is given in terms of the distance units in the ellipsoid vector, regardless of the angle units used.

## Description

*Departure* is the distance along a parallel between two points. Whereas a degree of latitude is always the same distance, a degree of longitude is different in length at different latitudes. In practice, this distance is usually given in nautical miles.

## Examples

On a spherical Earth, the departure is proportional to the cosine of the latitude:

```
distance = departure(0,10,0)
```

```
distance =
    10
```

```
distance = departure(0,10,60)
```

```
distance =
    5
```

When an ellipsoid is used, the result is more complicated. The distance at 60° is not exactly twice the 0° value:

```
distance = departure(0,10,0,almanac('earth','ellipsoid','nm'))
```

```
distance =  
    601.0772
```

```
distance = departure(0,10,60,almanac('earth','ellipsoid','nm'))
```

```
distance =  
    299.7819
```

## See Also

distance, stdm

# displaym

---

## Purpose

Display Version 1 geographic data structure

## Syntax

```
displaym(mstruct)
displaym(mstruct,'object')
displaym(mstruct,objects)
[lat,lon] = displaym(mstruct,objects,'exact')
h = displaym(mstruct,...)
```

`displaym(mstruct)` projects the data contained in the input structure onto the current axes. The current axes must have a valid map definition. The input `mstruct` must be a valid Mapping Toolbox geographic data structure.

`displaym(mstruct,'object')` displays vector data from entries in the Mapping Toolbox geographic data structure whose tags begin with the 'object' string. The output vectors use NaNs to separate the individual entries in the map structure. Matches of the tag string must be vector data (lines and patches) to be included in the output. The search is not case sensitive.

`displaym(mstruct,objects)` where `objects` is a character array or a cell array or strings, allows more than one object to be the basis for the search. Character array objects have trailing spaces stripped before matching.

`[lat,lon] = displaym(mstruct,objects,'exact')` requires an exact match to extract data.

`h = displaym(mstruct,...)` returns the handles to the objects projected.

## Remarks

A Mapping Toolbox Version 1 geographic data structure is a MATLAB structure that can contain line, patch, text, regular data grid, geolocated data grid, and light objects.

Object properties used in the display are taken from the `otherproperty` field of the structure. If a line or patch object's `otherproperty` field is empty, `displaym` uses default colors. A patch is assigned an index into the current colormap based on the structure's `tag` field. Lines are assigned colors from the current color order according to their tags.



**See Also**      `extractm, mlayers, updategeostruct`

# dist2str

---

## Purpose

Format distance strings

## Syntax

```
str = dist2str(distin)
str = dist2str(dist,format)
str = dist2str(dist,format,units)
str = dist2str(dist,format,digits)
str = dist2str(dist,format,units,digits)
```

`str = dist2str(distin)` converts a numerical vector of distances in kilometers, `distin`, to a string matrix. The output string matrix is useful for the display of distances.

`str = dist2str(dist,format)` uses the `format` string to specify the notation to be used for the string matrix. If blank or 'none', the result is a simple numerical representation (no indicator for positive distances, minus signs for negative distances). The only other format is 'pm' (for *plus-minus*) prefixes a + for positive distances.

`str = dist2str(dist,format,units)` defines the units in which the input distances are supplied, and which are encoded in the string matrix. Units must be one of the following: 'feet', 'kilometers', 'meters', 'nauticalmiles', 'statutemiles', 'degrees', or 'radians'. Note that statute miles are encoded as 'mi' in the string matrix, whereas in most Mapping Toolbox functions, 'mi' indicates international miles. If omitted or blank, 'kilometers' is assumed.

`str = dist2str(dist,format,digits)` or `str = dist2str(dist,format,units,digits)` uses the input `digits` to determine the number of decimal digits in the output matrix. `digits = -2` uses accuracy in the hundredths position, `digits = 0` uses accuracy in the units position. Default is `digits = -2`. For further discussion of specifying digits, see `roundn`.

## Description

The purpose of this function is to make distance-valued variables into strings suitable for map display.

## Examples

Create a vector of values and convert to strings:

```
d = [-3.7 2.95 87];
```

```
str = dist2str(d, 'none', 'km')
```

```
str =  
-3.70 km  
 2.95 km  
87.00 km
```

Now change the units to nautical miles, add plus signs to positive values, and truncate to the tenths ( $10^{-1}$ ) slot:

```
str = dist2str(d, 'pm', 'nm', -1)
```

```
str =  
-3.7 nm  
+3.0 nm  
+87.0 nm
```

**See Also**

`angl2str`, `roundn`

# distance

---

## Purpose

Distance between points on sphere or ellipsoid

## Syntax

```
[dist,az] = distance(lat1,lon1,lat2,lon2)
[dist,az] = distance(lat1,lon1,lat2,lon2,ellipsoid)
[dist,az] = distance(lat1,lon1,lat2,lon2,units)
[dist,az] = distance(lat1,lon1,lat2,lon2,ellipsoid,units)
[dist,az] = distance(pt1,pt2)
[dist,az] = distance(pt1,pt2,ellipsoid)
[dist,az] = distance(pt1,pt2,units)
[dist,az] = distance(pt1,pt2,ellipsoid,units)
[dist,az] = distance(track,...)
dist = distance(...)
```

## Background

Distance between two points can be calculated in two ways. For great circles (on the sphere) and geodesics (on the ellipsoid), the distance is the shortest surface distance between two points. For rhumb lines, the distance is measured along the rhumb line passing through the two points, which is not, in general, the shortest surface distance between them. For more information on this distinction, see “Great Circles, Rhumb Lines, and Small Circles” on page 3-32” in the *Mapping Toolbox User’s Guide*.

## Description

`[dist,az] = distance(lat1,lon1,lat2,lon2)` computes the great circle distance(s) between pairs of points on the surface of a sphere. The input latitudes and longitudes, `lat1`, `lon1`, `lat2`, and `lon2` are in degrees and can be scalars or arrays of equal size. The distance `dist` is expressed in degrees of arc length and will have the same size as the input arrays. `az` is the azimuth clockwise from north and in the specified angular units, from the first point to the second point. The value in `az` is identical to that obtained by calling `azimuth(...)`. When given a combination of scalar and array inputs, the scalar inputs are automatically expanded to match the size of the arrays.

`[dist,az] = distance(lat1,lon1,lat2,lon2,ellipsoid)` specifies the ellipsoidal shape of the Earth to be used with the two-element ellipsoid vector and computes the geodesic distance. The ellipsoid vector is of the form `[semimajor_axis, eccentricity]`. The resulting

distance and azimuth are expressed in the same units as the semimajor axis of the `ellipsoid` vector. The default ellipsoid is a unit sphere, which is sufficient for most applications. Use the `almanac` function to obtain an ellipsoid vector for ellipsoids by name.

`[dist,az] = distance(lat1,lon1,lat2,lon2,units)` uses the string *units* to define the angle units of the input latitudes and longitudes and the outputs `dist` and `az`. *units* may equal 'degrees' (the default value) or 'radians'.

`[dist,az] = distance(lat1,lon1,lat2,lon2,ellipsoid,units)` uses the *units* string to specify the units of the latitude-longitude coordinates, but the output range has the same units as the semimajor axis of the ellipsoid vector.

`[dist,az] = distance(pt1,pt2)` calculates the great circle distance and azimuth from `pt1` to `pt2`. These two-column matrices should be of the form [latitude longitude] in degrees. The resulting distance is returned in terms of degrees of arc length.

`[dist,az] = distance(pt1,pt2,ellipsoid)` calculates the great circle distance and azimuth from `pt1` to `pt2` on an ellipsoid specified with an ellipsoid vector.

`[dist,az] = distance(pt1,pt2,units)` specifies the angle units of `pt1` and `pt2`. The default value is 'degrees'. These units are also the distance units of the result (e.g., degrees of arc length) unless a ellipsoid vector is specified.

`[dist,az] = distance(pt1,pt2,ellipsoid,units)` is also a valid syntax.

`[dist,az] = distance(track,...)` specifies whether great circle distances or rhumb line distances are desired. Great circle distances, the default, are indicated with the standard *track* string 'gc'. Rhumb line distances are indicated with the standard *track* string 'rh'.

`dist = distance(...)` returns only the distance `dist`.

# distance

---

## Remarks

When you need to compute both distance and azimuth for the same point pair(s), it is more efficient to do so with a single call to `distance`. That is, use

```
[dist az] = distance(...);
```

rather than the slower

```
dist = distance(...)  
az = azimuth(...)
```

To express the output `dist` as an arc length expressed in either degrees or radians, omit the `ellipsoid` argument. This is possible only on a sphere. If `ellipsoid` is supplied, `dist` is a distance expressed in the same units as the semimajor axis of the ellipsoid. Specify `ellipsoid` as `[R 0]` to compute `dist` as a distance on a sphere of radius `R`, with `dist` having the same units as `R`.

## Examples

Imagine a trip from Norfolk, Virginia (37°N, 76°W), to Cape St. Vincent, Portugal (37°N, 9°W), just outside the Straits of Gibraltar. The distance between these two points depends upon the *track* string selected. Using the `pt1,pt2` notation, the two cases result in

```
dist = distance('gc',[37,-76],[37,-9])
```

```
dist =  
    52.3094
```

```
dist = distance('rh',[37,-76],[37,-9])
```

```
dist =  
    53.5086
```

The difference between these two tracks is 1.1992 degrees, or about 72 nautical miles. This represents about 2% of the total trip distance. The trade-off is that at the cost of those 72 miles, the entire trip can be made on a rhumb line with a fixed course of 90°, due east, while in

order to follow the shorter great circle path, the course must be changed continuously.

On a meridian and on the Equator, great circles and rhumb lines coincide, so the distances are the same. For example,

```
dist = distance(37,-76,67,-76) % great circle distance
```

```
dist =  
    30.0000
```

```
dist = distance('rh',37,-76,67,-76) % rhumb line distance
```

```
dist =  
    30.0000
```

The distances are the same, 30°, or about 1800 nautical miles (there are about 60 nautical miles in a degree of arc length).

## See Also

almanac, azimuth, elevation, reckon, track, track1, track2, trackg

# distortcalc

---

## Purpose

Distortion parameters for map projections

## Syntax

```
areascale = distortcalc(lat, long)
areascale = distortcalc(mstruct, lat, long)
[areascale, angdef, maxscale, minscale, merscale,
 parscale] = distortcalc(...)
```

`areascale = distortcalc(lat, long)` computes the area distortion for the current map projection at the specified geographic location. An area scale of 1 indicates no scale distortion. Latitude and longitude can be scalars, vectors, or matrices in the angle units of the defined map projection.

`areascale = distortcalc(mstruct, lat, long)` uses the projection defined in the map structure `mstruct`.

`[areascale, angdef, maxscale, minscale, merscale, parscale] = distortcalc(...)` computes the area scale, maximum angular deformation of right angles (in the angle units of the defined projection), the particular maximum and minimum scale distortions in any direction, and the particular scale along the meridian and parallel. You can also call `distortcalc` with fewer output arguments, in the order shown.

## Background

Map projections inevitably introduce distortions in the shapes and sizes of objects as they are transformed from three-dimensional spherical coordinates to two-dimensional Cartesian coordinates. The amount and type of distortion vary between projections, over the projection, and with the selection of projection parameters such as standard parallels. This function allows a quantitative evaluation of distortion parameters.

## Examples

At the equator, the Mercator projection is free of both area and angular distortion:

```
axesm mercator
[areascale, angdef] = distortcalc(0,0)

areascale =
```



```
1.0000  
angdef =  
8.5377e-007
```

At 60 degrees north, objects are shown at 400% of their true area. The projection is conformal, so angular distortion is still zero.

```
[areascale,angdef] = distortcalc(60,0)  
  
areascale =  
4.0000  
angdef =  
4.9720e-004
```

**Remarks**

This function uses a finite difference technique. The geographic coordinates are perturbed slightly in different directions and projected. A small amount of error is introduced by numerical computation of derivatives and the variation of map distortion parameters.

**See Also**

`mdistort`, `tissot`

# distdim

---

## Purpose

Convert length units

## Syntax

```
distOut = distdim(distIn, from, to)
distOut = distdim(distIn, from, to, radius)
distOut = distdim(distIn, from, to, sphere)
```

---

**Note** `distdim` has been replaced by `unitsratio`, but will be maintained for backward compatibility. See “Replacing `distdim`” on page 11-146 for details.

---

`distOut = distdim(distIn, from, to)` converts `distIn` from the units specified by the string *from* to the units specified by the string *to*. *from* and *to* are case-insensitive, and may equal any of the following:

```
'meters' or 'm'
'feet' or 'ft'              U.S. survey feet
'kilometers' or 'km'
'nauticalmiles' or 'nm'
'miles', 'statutemiles', 'mi', or 'sm'  Statute miles
'degrees' or 'deg'
'radians' or 'rad'
```

If either *from* or *to* indicates angular units ('degrees' or 'radians'), the conversion to or from linear distance is made along a great circle arc on a sphere with a radius of 6371 km, the mean radius of the Earth.

`distOut = distdim(distIn, from, to, radius)`, where one of the unit strings, either *from* or *to*, indicates angular units and the other unit string indicates length units, uses a great circle arc on a sphere of the given radius. The specified length units must apply to radius as well as to the input distance (when *from* indicates length) or output distance (when *to* indicates length). If neither *from* nor *to* indicates angular units, or if both do, then the value of radius is ignored.

`distOut = distdim(distIn, from, to, sphere)`, where either *from* or *to* indicates angular units, uses a great circle arc on a sphere approximating a body in the Solar System. *sphere* may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive. If neither *to* nor *from* is angular, *sphere* is ignored.

## Remarks

### Arc Lengths of Angles Not Constant

Distance is expressed in one of two general forms: as a linear measure in some unit (kilometers, miles, etc.) or as angular arc length (degrees or radians). While the use of linear units is generally understood, angular arc length is not always as clear. The conversion from angular units to linear units for the arc along any circle is the angle in radians multiplied by the radius of the circle. On the sphere, this means that radians of latitude are directly translatable to kilometers, say, by multiplying by the radius of the Earth in kilometers (about 6,371 km). However, the linear distance associated with radians of longitude changes with latitude; the radius in question is then not the radius of the Earth, but the (chord) radius of the small circle defining that parallel. In Mapping Toolbox, the angle in radians or degrees associated with any distance is the arc length of a great circle passing through the points of interest. Therefore, the radius in question always refers to the radius of the relevant sphere, consistent with the distance function.

### Exercise Caution with 'feet' and 'miles'

*Exercise caution with 'feet' and 'miles'.* `distdim` interprets 'feet' and 'ft' as U.S. survey feet, and does not support international feet at all. In contrast, `unitsratio` follows the opposite, and more standard approach, interpreting both 'feet' and 'ft' as international feet. `unitsratio` provides separate options, including 'surveyfeet' and 'sf', to indicate survey feet. By definition, one international foot is exactly 0.3048 meters and one U.S. survey foot is exactly 1200/3937 meters. For many applications, the difference is significant. Most projected coordinate systems use either the meter or the survey foot as a standard unit. International feet are less likely to be used, but do occur sometimes. Likewise, `distdim` interprets 'miles' and 'mi' as

statute miles (also known as U.S. survey miles), and does not support international miles at all. By definition, one international mile is 5,280 international feet and one statute mile is 5,280 survey feet. You can evaluate:

```
unitsratio('millimeter','statute mile') - ...
unitsratio('millimeter','mile')
```

to see that the difference between a statute mile and an international mile is just over three millimeters. This may seem like a very small amount over the length of a single mile, but mixing up these units could result in a significant error over a sufficiently long baseline. Originally, the behavior of `distdim` with respect to `'miles'` and `'mi'` was documented only indirectly, via the now-obsolete `unitstr` function. As with feet, `unitsratio` takes a more standard approach. `unitsratio` interprets `'miles'` and `'mi'` as international miles, and `'statute miles'` and `'sm'` as statute miles. (`unitsratio` accepts several other strings for each of these units; see the `unitsratio` help for further information.)

## Replacing `distdim`

If both *from* and *to* are known at the time of coding, then you may be able to replace `distdim` with a direct conversion utility, as in the following examples:

```
distdim(dist, 'nm', 'km') == km2nm(dist)
distdim(dist, 'sm', 'deg') == deg2sm(dist)
distdim(dist, 'rad', 'km', 'moon') == rad2km(dist, 'moon')
```

If there is no appropriate direct conversion utility, or you won't know the value of *from* and/or *to* until run time, you can generally replace

```
distdim(dist, FROM, TO)
```

with

```
unitsratio(TO, FROM) * dist
```

If you are using units of feet or miles, see the cautionary note above about how they are interpreted. For example, with `distIn` in meters and `distOut` in survey feet, `distOut = distdim(distIn, 'meters', 'feet')` should be replaced with `distOut = unitsratio('survey feet', 'meters') * distIn`. Saving a multiplicative factor computed with `unitsratio` and using it to convert in a separate step can make code cleaner and more efficient than using `distdim`. For example, replace

```
dist1_meters = distdim(dist1_nm, 'nm', 'meters');
dist2_meters = distdim(dist2_nm, 'nm', 'meters');
```

with

```
metersPerNM = unitsratio('meters', 'nm');
dist1_meters = metersPerNM * dist1_nm;
dist2_meters = metersPerNM * dist2_nm;
```

`unitsratio` does not perform great-circle conversion between units of length and angle, but it can be easily combined with other functions to do so. For example, to convert degrees to meters along a great-circle arc on a sphere approximating the planet Mars, you could replace

```
distdim(dist, 'degrees', 'meters', 'mars')
```

with

```
unitsratio('meters', 'km') * deg2km(dist, 'mars')
```

## Examples

Convert 100 kilometers to nautical miles:

```
distkm = 100
```

```
distkm =
```

# distdim

---

```
100
```

```
distnm = distdim(distkm, 'kilometers', 'nauticalmiles')
```

```
distnm =  
53.9957
```

A degree of arc length is about 60 nautical miles:

```
distnm = distdim(1, 'deg', 'nm')
```

```
distnm =  
60.0405
```

This is not accidental. It is the original definition of the nautical mile. Naturally, this assumption does not hold on other planets:

```
distnm = distdim(1, 'deg', 'nm', 'mars')
```

```
distnm =  
31.9474
```

## See Also

deg2km, deg2nm, deg2sm, km2deg, km2nm, km2rad, km2sm, nm2deg, nm2km, nm2rad, nm2sm, rad2km, rad2nm, rad2sm, sm2deg, sm2km, sm2nm, sm2rad, unitsratio

**Purpose** Convert degrees-minutes to degrees

**Syntax** `angleInDegrees = dm2degrees(DM)`

`angleInDegrees = dm2degrees(DM)` converts angles from degree-minutes representation to values in degrees which may include a fractional part (sometimes called “decimal degrees”). `DM` should be `N`-by-2 and real-valued, with one row per angle. The output will be an `N`-by-1 column vector whose  $k^{\text{th}}$  element corresponds to the  $k^{\text{th}}$  row of `DM`. The first column of `DM` contains the “degrees” element and should be integer-valued. The second column contains the “minutes” element and may have a fractional part. For an angle that is positive (north latitude or east longitude) or equal to zero, all elements in the row need to be nonnegative. For a negative angle (south latitude or west longitude), the first nonzero element in the row should be negative and the remaining value, if any, should be nonzero. Thus, for an input row with value `[D M]`, with integer-valued `D` and real `M`, the output value will be

$$\text{SGN} * (\text{abs}(D) + \text{abs}(M)/60)$$

where `SGN` is 1 if `D` and `M` are both nonnegative and -1 if the first nonzero element of `[D M]` is negative (an error results if a nonzero `D` is followed by a negative `M`). Any fractional parts in the first (degrees) columns of `DM` are ignored. An error results unless the absolute values of all elements in the second (minutes) column are less than 60.

### Example

```
dm = [ ...
      30 44.78012; ...
     -82 39.90825; ...
      0 -17.12345; ...
      0 14.82000];
format long g
angleInDegrees = dm2degrees(dm)

angleInDegrees =
    30.7463353333333
   -82.6651375
```

# dm2degrees

---

-0.2853908333333333  
0.247

## See Also

degrees2dm, deg2rad dms2degrees, str2angle



**Purpose** Convert degrees-minutes-seconds to degrees

**Syntax** `angleInDegrees = dms2degrees(DMS)`

`angleInDegrees = dms2degrees(DMS)` converts angles from degree-minutes-seconds representation to values in degrees which may include a fractional part (sometimes called “decimal degrees”). DMS should be N-by-3 and real-valued, with one row per angle. The output will be an N-by-1 column vector whose  $k^{\text{th}}$  element corresponds to the  $k^{\text{th}}$  row of DMS. The first column of DMS contains the “degrees” element and should be integer-valued. The second column contains the “minutes” element and should be integer-valued. The third column contains the “seconds” element and may have a fractional part. For an angle that is positive (north latitude or east longitude) or equal to zero, all elements in the row need to be nonnegative. For a negative angle (south latitude or west longitude), the first nonzero element in the row should be negative and the remaining values should be positive. Thus, for an input row with value `[D M S]`, with integer-valued D and M, and real D, M, and S, the output value will be

$$\text{SGN} * (\text{abs}(D) + \text{abs}(M)/60 + \text{abs}(S)/3600)$$

where SGN is 1 if D, M, and S are all nonnegative and -1 if the first nonzero element of `[D M S]` is negative (an error results if a nonzero element is followed by a negative element). Any fractional parts in the first (degrees) and second (minutes) columns of DMS are ignored. An error results unless the absolute values of all elements in the second (minutes) and third (seconds) columns are less than 60.

**Example**

```
dms = [ ...
        30  50 44.78012; ...
       -82   2 39.90825; ...
         0 -30 17.12345; ...
         0   0 14.82000];
format long g
angleInDegrees = dms2degrees(dms)
```

# dms2degrees

---

```
angleInDegrees =  
    30.8457722555556  
    -82.0444189583333  
    -0.504756513888889  
    0.00411666666666667
```

## See Also

degrees2dm, deg2rad dm2degrees, str2angle

**Purpose** Convert angles from deg:min:sec to deg:min encoding

---

**Note** The `dms2dm` function is obsolete and errors when used. Instead, combine `dms2degrees` and `degrees2dm`, as in `degrees2dm(dms2degrees([-29 42 18.7]))`

---

**Syntax** `angleOut = dms2dm(angleIn)`  
`angleOut = dms2dm(angleIn)` rounds angles input in degrees-minutes-seconds (*dms*) format to the appropriate value in degrees-minutes (*dm*) format. This special handling is needed because there are 60, and not 100, seconds in a minute.

**Example** Round 4° 45' 29" and 4° 45' 31" to dm format:

```
dms2dm(445.29)
```

```
ans =  
    445.00
```

```
dms2dm(445.31)
```

```
ans =  
    446.00
```

**See Also** `angledim`, `deg2rad`, `dms2degrees`

# dms2mat

---

## Purpose

Expand deg:min:sec encoded vector to [deg min sec] matrix

---

**Note** The `dms2mat` function is obsolete and errors when used. Instead use `degrees2dms` to convert degrees to degrees-minutes-seconds vector.

---

## Syntax

```
[d,m,s] = dms2mat(angleIn)
[d,m,s] = dms2mat(angleIn,n)
matout = dms2mat(angleIn,n)
```

`[d,m,s] = dms2mat(angleIn)` takes angles input in dms inputs and splits their components into three outputs, one each for degrees, minutes, and seconds.

`[d,m,s] = dms2mat(angleIn,n)` specifies the power of 10, `n`, to which the resulting seconds output should be rounded (that is, if a result is 12.567 seconds, and `n = -2`, the resulting seconds output would be 12.57). The default value of `n` is -5.

`matout = dms2mat(angleIn,n)` returns a three-column matrix, `matout`, in which the columns represent degrees, minutes, and seconds, respectively. In this case, `angleIn` must be a vector.

## Examples

```
angleIn = [12547.34; 54323.17];
```

```
[d,m,s] = dms2mat(angleIn)
```

```
d =
```

```
    125
```

```
    543
```

```
m =
```

```
    47
```

```
    23
```

```
s =
```

```
    34
```

```
    17
```

```
matout = dms2mat(angleIn)
```

```
matout =  
  125  47  34  
  543  23  17
```

**See Also** [dm2degrees](#), [dms2degrees](#)

# dms2deg, dms2rad

---

**Purpose** Convert angles from deg:min:sec encoding to degrees or radians

---

**Note** The `dms2deg` and `dms2rad` functions are obsolete and error when used. Instead use `dms2degrees` to convert degrees-minutes-seconds vector to degrees and then call `deg2rad` or multiply by  $\pi/180$ .

---

**Syntax**

```
angleOut = dms2deg(angleIn)
angleOut = dms2rad(angleIn)
```

`angleOut = dms2deg(angleIn)` converts angles input in degrees-minutes-seconds (*dms*) format to the equivalent measure in decimal degrees.

`angleOut = dms2rad(angleIn)` converts angles input in degrees-minutes-seconds (*dms*) format to the equivalent measure in radians.

**Remarks** The inputs can be in degrees-minutes (*dm*) format, because numerically they look like *dms* format in which seconds are always zero.

**Example**

```
dms2deg(430.00)

ans =

    4.50
```

**See Also** `angledim`, `angl2str`, `deg2rad`, `dms2degrees`

**Purpose**

Dead reckoning positions for track

**Syntax**

```
[drlat,drlong,drttime] = dreckon(waypoints,time,speed)
[drlat,drlong,drttime] = dreckon (waypoints,time,speed,
    spdtimes)
```

[drlat,drlong,drttime] = dreckon(waypoints,time,speed) returns the positions and times of required dead reckoning (DR) points for the input track that starts at the input time. The track should be in navigational track format (two columns, latitude then longitude, in order of traversal). These waypoints are the starting and ending points of each leg of the track. There is one fewer track leg than waypoints, as the last point included is the end of the track. In navigation, the first waypoint would be a navigational fix, taken at time. The speed input can be a scalar, in which case a constant speed is used throughout, or it can be a vector in which one speed is given for each track leg (that is, speed changes coincide with course changes).

```
[drlat,drlong,drttime] = dreckon
(waypoints,time,speed,spdtimes) allows speed
changes to occur independent of course changes. The elements of the
speed vector must have a one-to-one correspondence with the elements
of the spdtimes vector. This latter variable consists of the time
interval after time at which each speed order ends. For example, if
time is 6.75, and the first element of spdtimes is 1.35, then the first
speed element is in effect from 6.75 to 8.1 hours. When this syntax
is used, the last output DR is the earlier of the final spdtimes time
or the final waypoints point.
```

**Background**

This is a navigational function. It assumes that all latitudes and longitudes are in degrees, all distances are in nautical miles, all times are in hours, and all speeds are in knots, that is, nautical miles per hour.

Dead reckoning is an estimation of position at various times based on courses, speeds, and times elapsed from the last certain position, or fix. In navigational practice, a dead reckoning position, or DR, must be plotted at every course change, every speed change, and at every hour,

on the hour. Navigators also DR at other times that are not relevant to this function.

Often in practice, when two events occur that require DRs within a very short time, only one DR is generated. This function mimics that practice by setting a tolerance of 3 minutes (0.05 hours). No two DRs will fall closer than that.

Refer to “Navigation” on page 9-11 in the *Mapping Toolbox User’s Guide* for further information.

## Examples

Assume that a navigator gets a fix at noon, 1200Z, which is (10.3°N, 34.67°W). He’s in a hurry to make a 1330Z rendezvous with another ship at (9.9°N, 34.5°W), so he plans on a speed of 25 knots. After the rendezvous, both ships head for (0°, 37°W). The engineer wants to take an engine off line for maintenance at 1430Z, so at that time, speed must be reduced to 15 knots. At 1530Z, the maintenance will be done. Determine the DR points up to the end of the maintenance.

```
waypoints = [10.1 -34.6; 9.9 -34.5; 0 -37]

waypoints =
    10.1000  -34.6000    % Fix at noon
     9.9000  -34.5000    % Rendezvous point
     0       -37.0000    % Ultimate destination

speed = [25; 15];
spdtimes = [2.5; 3.5];    % Elapsed times after fix
noon = 12;
[drlat,drlong,dertime] = dreckon(waypoints,noon,speed,spdtimes);
[drlat,drlong,dertime]

ans =
    9.8999  -34.4999   12.5354    % Course change at waypoint
    9.7121  -34.5478   13.0000    % On the hour
    9.3080  -34.6508   14.0000    % On the hour
    9.1060  -34.7022   14.5000    % Speed change to 15 kts
    8.9847  -34.7330   15.0000    % On the hour
```



```
8.8635 -34.7639 15.5000 % Stop at final spdtime, last  
% waypoint has not been reached
```

**See Also**      legs, navfix, track

# driftcorr

---

## Purpose

Heading to correct for wind or current drift

## Syntax

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

heading = driftcorr(course,airspeed,windfrom,windspeed) computes the heading that corrects for drift due to wind (for aircraft) or current (for watercraft). course is the desired direction of movement (in degrees), airspeed is the speed of the vehicle relative to the moving air or water mass, windfrom is the direction facing into the wind or current (in degrees), and windspeed is the speed of the wind or current (in the same units as airspeed).

[heading,groundspeed,windcorrangle] = driftcorr(...) also returns the ground speed and wind correction angle. The wind correction angle is positive to the right, and negative to the left.

## Example

An aircraft cruising at a speed of 160 knots plans to fly to an airport due north of its current position. If the wind is blowing from 310 degrees at 45 knots, what heading should the aircraft fly to remain on course?

```
course=0; airspeed=160;windfrom=310; windspeed = 45;
[heading,groundspeed,windcorrangle] =
driftcorr(course,airspeed,windfrom,windspeed)
```

```
heading =
```

```
347.56
```

```
groundspeed =
```

```
127.32
```

```
windcorrangle =
```

```
-12.442
```

The required heading is 348 degrees, which amounts to a wind correction angle of 12 degrees to the left of course. The headwind component reduces the aircraft's ground speed to 127 knots.

**See Also**

driftvel

# driftvel

---

## Purpose

Wind or current from heading, course, and speeds

## Syntax

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

```
[windfrom,windspeed] = driftvel  
(course,groundspeed,heading,airspeed) computes  
the wind (for aircraft) or current (for watercraft) from course, heading,  
and speeds. course and groundspeed are the direction and speed  
of movement relative to the ground (in degrees), heading is the  
direction in which the vehicle is steered, and airspeed is the speed of  
the vehicle relative to the air mass or water. The output windfrom  
is the direction facing into the wind or current (in degrees), and  
windspeed is the speed of the wind or current (in the same units as  
airspeed and groundspeed).
```

## Example

An aircraft is cruising at a true air speed of 160 knots and a heading of 10 degrees. From the Global Positioning System (GPS) receiver, the pilot determines that the aircraft is progressing over the ground at 155 knots in a northerly direction. What is the wind aloft?

```
course = 0; groundspeed = 155; heading = 10; airspeed = 160;  
[windfrom,windspeed] =  
driftvel(course,groundspeed,heading,airspeed)
```

```
windfrom =  
84.717
```

```
windspeed =  
27.902
```

The wind is blowing from the right, 085 degrees at 28 knots.

## See Also

driftcorr

**Purpose**

Read U.S. Department of Defense Digital Terrain Elevation Data (DTED)

**Syntax**

```
[Z, refvec] = dted
[Z, refvec] = dted(filename)
[Z, refvec] = dted(filename, samplefactor)
[Z, refvec] = dted(filename, samplefactor, latlim, lonlim)
[Z, refvec] = dted(dirname, samplefactor, latlim, lonlim)
[Z, refvec, UHL, DSI, ACC] = dted(...)
```

[Z, refvec] = dted returns all of the elevation data in a DTED file as a regular data grid, Z, with elevations in meters. The file is selected interactively. This function reads the DTED elevation files, which generally have filenames ending in .dtN, where N is 0,1,2,3,... refvec is the associated three-element referencing vector that geolocates Z.

[Z, refvec] = dted(filename) returns all of the elevation data in the specified DTED file. The file must be found on the MATLAB path. If not found, the file may be selected interactively.

[Z, refvec] = dted(filename, samplefactor) subsamples data from the specified DTED file. samplefactor is a scalar integer. When samplefactor is 1 (the default), DTED reads the data at its full resolution. When samplefactor is an integer n greater than one, every nth point is read.

[Z, refvec] = dted(filename, samplefactor, latlim, lonlim) reads the data for the part of the DTED file within the latitude and longitude limits. The limits must be two-element vectors in units of degrees.

[Z, refvec] = dted(dirname, samplefactor, latlim, lonlim) reads and concatenates data from multiple files within a DTED CD-ROM or directory structure. The dirname input is a string with the name of a directory containing the DTED directory. Within the DTED directory are subdirectories for each degree of longitude, each of which contain files for each degree of latitude. For DTED CD-ROMs, dirname is the device name of the CD-ROM drive.

[Z, refvec, UHL, DSI, ACC] = dted(...) returns structures containing the DTED User Header Label (UHL), Data Set Identification (DSI) and ACCuracy metadata records.

## Background

The U. S. Department of Defense, through the National Geospatial Intelligence Agency, produces several kinds of digital cartographic data. One is digital elevation data, in a series called DTED, for Defense Digital Terrain Elevation Data. The data is available as 1-by-1 degree quadrangles at horizontal resolutions ranging from about 1 kilometer to 1 meter. The lowest resolution data is available to the public. Certain higher resolution data is restricted to the U.S. Department of Defense and its contractors.

DTED Level 0 files have 121-by-121 points. DTED Level 1 files have 1201-by-1201. The edges of adjacent tiles have redundant records. Maps extend a half a cell outside the requested map limits. The 1 kilometer data and some higher-resolution data is available online, as are product specifications and documentation. DTED files are binary. No line ending conversion or byte-swapping is required when downloading a DTED file.

## Remarks

### Latitude-Dependent Sampling

In DTED files north of 50° North and south of 50° South, where the meridians have converged significantly relative to the equator, the longitude sampling interval is reduced to half of the latitude sampling interval. In order to retain square output cells, this function reduces the latitude sampling to match the longitude sampling. For example, it will return a 121-by-121 elevation grid for a DTED file covering from 49 to 50 degrees north, but a 61-by-61 grid for a file covering from 50 to 51 degrees north. When you supply a directory name instead of a file name, and `latlim` spans either 50° North or 50° South, an error results.

### Snapping Latitude and Longitude Limits

If you call `dted` specifying arbitrary latitude-longitude limits for a region of interest, the grid and referencing vector returned will not exactly honor the limits you specified unless they fall precisely on grid cell boundaries. Because grid cells are discrete and cannot be arbitrarily

divided, the data grid returned will include all areas between your latitude-longitude limits and the next row or column of cells, potentially in all four directions.

### Data Sources and Information

DTED files contain digital elevation maps covering 1-by-1-degree quadrangles at horizontal resolutions ranging from about 1 kilometer to 1 meter. For details on locating DTED for download over the Internet, see the following documentation at the MathWorks Web site:

<http://www.mathworks.com/support/tech-notes/2100/2101.html>

### Null Data Values

Some DTED Level 1 and higher data tiles contain null data cells, coded with value -32767. When encountered, these null data values are converted to NaN.

### Nonconforming Data Encoding

DTED files from some sources may depart from the specification by using two's complement encoding for binary elevation files instead of "sign-bit" encoding. This difference affects the decoding of negative values, and incorrect decoding usually leads to nonsensical elevations.

Thus, if the DTED function determines that all the (nonnull) negative values in a file would otherwise be less than -12,000 meters, it issues a warning and assumes two's complement encoding.

### Examples

```
[Z,refvec] = dted('n38.dt0');
[Z,refvec,UHL,DSI,ACC] = dted('n38.dt0',1,[38.5 38.8],...
[-76.8 -76.6]);
[Z,refvec,UHL,DSI,ACC] = dted('f:',1,[38.5 38.8],...
[-76.8 -76.6]);
```

### See Also

usgsdem, gtopo30, tbase, etopo

# dteds

---

## Purpose

DTED filenames for latitude-longitude quadrangle

## Syntax

```
fname = dteds(latlim,lonlim)
fname = dteds(latlim,lonlim,level)
```

fname = dteds(latlim,lonlim) returns Level 0 DTED file names (directory and name) required to cover the geographic region specified by latlim and lonlim.

fname = dteds(latlim,lonlim,level) controls the level for which the file names are generated. Valid inputs for the level of the DTED files include 0, 1, or 2.

## Background

The U. S. Department of Defense produces several kinds of digital cartographic data. One is digital elevation data, in a series called DTED, for Defense Digital Terrain Elevation Data. The data is available as 1-by-1 degree quadrangles at horizontal resolutions ranging from about 1 kilometer to 1 meter. The lowest resolution data is available to the public. Higher resolution data is restricted to the U.S. Department of Defense and its contractors.

Determining the files needed to cover a particular region requires knowledge of the DTED database naming conventions. This function constructs the file names for a given geographic region based on these conventions.

## Examples

Which files are needed for Cape Cod?

```
latlim = [ 41.15 42.22]; lonlim = [-70.94 -69.68];
dteds(latlim,lonlim,1)

ans =
    '\DTED\W071\N41.dt1'
    '\DTED\W070\N41.dt1'
    '\DTED\W071\N42.dt1'
    '\DTED\W070\N42.dt1'
```

## See Also

dted



**Purpose**

Wrap longitudes to values east of specified meridian

---

**Note** The `eastof` function is obsolete and will be removed in a future release of Mapping Toolbox. Replace it with the following calls, which are also more efficient:

```
eastof(lon,meridian,'degrees') ==> meridian+mod(lon-meridian,360)
```

```
eastof(lon,meridian,'radians') ==> meridian+mod(lon-meridian,2*pi)
```

---

**Syntax**

```
lonWrapped = eastof(lon,meridian)
```

```
lonWrapped = eastof(lon,meridian,angleunits)
```

`lonWrapped = eastof(lon,meridian)` wraps angles in `lon` to values in the interval `[meridian meridian+360)`. `lon` is a scalar longitude or vector of longitude values. All inputs and outputs are in degrees.

`lonWrapped = eastof(lon,meridian,angleunits)` specifies the input and output units with the string *angleunits*. *angleunits* can be either 'degrees' or 'radians'. It may be abbreviated and is case-insensitive. If *angleunits* is 'radians', the input is wrapped to the interval `[meridian meridian+2*pi)`.

# ecc2flat

---

**Purpose** Flattening of ellipse with given eccentricity

**Syntax** `flattening = ecc2flat(eccentricity)`  
`flattening = ecc2flat(eccentricity)` returns the equivalent flattening for the input eccentricities. If the input, `eccentricity`, is a two-column vector, only the second column is used. This allows the standard two-element ellipsoid vectors to be used as rows of the input, because the second element of these vectors is the eccentricity. In all other cases, all columns of the input are used.

**Description** Flattening and eccentricity are two methods of defining an ellipsoid.

**Example**

```
flattening = ecc2flat(almanac('earth','ellipsoid'))

flattening =
    0.0034
```

**See Also** `almanac`, `ecc2n`, `majaxis`, `flat2ecc`

---

<b>Purpose</b>	n-value of ellipse with given eccentricity
<b>Syntax</b>	<pre>n = ecc2n(eccentricity)</pre> <p><code>n = ecc2n(eccentricity)</code> returns the equivalent <math>n</math> for the input eccentricities. If the input, <code>eccentricity</code>, is a two-column vector, only the second column is used. This allows the standard two-element ellipsoid vectors to be used as rows of the input, because the second element of these vectors is the eccentricity. In all other cases, all columns of the input are used.</p>
<b>Description</b>	<p>Eccentricity and the parameter <math>n</math> are two methods of defining an ellipsoid. The definition of <math>n</math> is</p> $\frac{(\text{semimajor axis} - \text{semiminor axis})}{(\text{semimajor axis} + \text{semiminor axis})}$
<b>Example</b>	<pre>n = ecc2n(almanac('earth','ellipsoid')) n =     0.00167922039463</pre>
<b>See Also</b>	<code>almanac</code> , <code>ecc2flat</code> , <code>majaxis</code> , <code>n2ecc</code>

# ecef2geodetic

---

<b>Purpose</b>	Convert geocentric (ECEF) to geodetic coordinates
<b>Syntax</b>	<pre>[phi, lambda, h] = ecef2geodetic (x,y,z,ellipsoid)</pre> <p>[phi, lambda, h] = ecef2geodetic (x,y,z,ellipsoid) converts point locations in geocentric Cartesian coordinates, stored in the coordinate arrays x, y, z, to geodetic coordinates phi (geodetic latitude in radians), lambda (longitude in radians), and h (height above the ellipsoid). The geodetic coordinates refer to the reference ellipsoid specified by ellipsoid (a row vector with the form [semimajor axis, eccentricity]). x, y, and z must use the same units as the semimajor axis; h will also be expressed in these units. x, y, and z must have the same shape; phi, lambda, and h will have this shape also.</p>
<b>Remarks</b>	For a definition of the geocentric system, also known as Earth-Centered, Earth-Fixed (ECEF), see the help for geodetic2ecef.
<b>See Also</b>	ecef2lv, geodetic2ecef, lv2ecef

---

<b>Purpose</b>	Convert geocentric (ECEF) to local vertical coordinates
<b>Syntax</b>	<pre>[x1, y1, z1] = ecef2lv(x,y,z,phi0,lambda0,h0,ellipsoid)</pre> <p><code>[x1, y1, z1] = ecef2lv(x,y,z,phi0,lambda0,h0,ellipsoid)</code> converts geocentric point locations specified by the coordinate arrays <code>x</code>, <code>y</code>, and <code>z</code> to the local vertical coordinate system with its origin at geodetic latitude <code>phi0</code>, longitude <code>lambda0</code>, and ellipsoidal height <code>h0</code>. <code>x</code>, <code>y</code>, and <code>z</code> may be arrays of any shape, as long as they all match in size. <code>phi0</code>, <code>lambda0</code>, and <code>H0</code> must be scalars. <code>ellipsoid</code> is a row vector with the form <code>[semimajor axis, eccentricity]</code>. <code>x</code>, <code>y</code>, <code>z</code>, and <code>h0</code> must have the same length units as the semimajor axis. <code>phi0</code> and <code>lambda0</code> must be in radians. The output coordinate arrays, <code>x1</code>, <code>y1</code>, and <code>z1</code> are the local vertical coordinates of the input points. They have the same size as <code>x</code>, <code>y</code>, and <code>z</code> and have the same length units as the semimajor axis.</p> <p>In the local vertical Cartesian system defined by <code>phi0</code>, <code>lambda0</code>, <code>h0</code>, and <code>ellipsoid</code>, the <code>x1</code> axis is parallel to the plane tangent to the ellipsoid at <code>(phi0, lambda0)</code> and points due east. The <code>y1</code> axis is parallel to the same plane and points due north. The <code>z1</code> axis is normal to the ellipsoid at <code>(phi0, lambda0)</code> and points outward into space. The local vertical system is sometimes referred to as east-north-up or ENU.</p>
<b>Remarks</b>	For a definition of the geocentric system, also known as Earth-Centered, Earth-Fixed (ECEF), see the help for <code>geodetic2ecef</code> .
<b>See Also</b>	<code>ecef2geodetic</code> , <code>elevation</code> , <code>geodetic2ecef</code> , <code>lv2ecef</code>

# egm96geoid

---

## Purpose

Read 15-minute gridded geoid heights from EGM96l

## Syntax

```
[Z,refvec] = egm96geoid(scalefactor)
```

```
[Z,refvec] = egm96geoid(scalefactor,latlim,lonlim)
```

`[Z,refvec] = egm96geoid(scalefactor)` reads the data for the entire world, downsampling the data by the scale factor. The result is returned as a regular data grid and associated three-element referencing vector that geolocates Z. Heights are given in meters in the tide-free system.

`[Z,refvec] = egm96geoid(scalefactor,latlim,lonlim)` reads the data for the part of the world within the latitude and longitude limits. The limits must be two-element vectors in units of degrees. Longitude limits can be defined in the range `[-180 180]` or `[0 360]`. For example, `lonlim = [170 190]` returns data centered on the date line, while `lonlim = [-10 10]` returns data centered on the prime meridian.

## Background

Although the Earth is round, it is not exactly a sphere. The shape of the Earth is usually defined by the geoid, which is defined as a gravitational equipotential surface, but can be conceptualized as the shape the ocean surface would take in the absence of waves, weather, and land. For cartographic purposes it is generally sufficient to treat the Earth as a sphere or ellipsoid of revolution. For other applications, a more detailed model of the geoid such as EGM 96 may be required. EGM 96 is a spherical harmonic model of the geoid complete to degree and order 360. This function reads from a file of gridded geoid heights derived from the EGM 96 harmonic coefficients.

## Examples

Read the EGM 96 geoid grid for the world, taking every 10th point.

```
[Z,refvec] = egm96geoid(10);
```

Read a subset of the geoid grid at full resolution and interpolate to find the geoid height at a point between grid points.

```
[Z,refvec] = egm96geoid(1,[-10 -12],[129 132]);  
z = ltln2val(Z,refvec,-11.1,130.22,'bicubic')
```

z =  
53.4809

**Remarks**

This function reads the 15-minute EGM96 grid file WW15MGH.GRD. The grid is available as either a DOS self-extracting compressed file or a UNIX compressed file. Do not modify the file once it has been extracted.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>

---

Maps will extend a half a cell outside the requested map limits.

There are 721 rows and 1441 columns of values in the grid at full resolution. The low resolution data in GEOID.MAT is derived from the EGM 96 grid.

**See Also**

1t1n2va1

# elevation

---

## Purpose

Local vertical elevation angle, range, and azimuth

## Syntax

```
[elevationangle, slanrange, azimuthangle] = ...  
    elevation(lat1, lon1, alt1, lat2, lon2, alt2)  
[...] = elevation(lat1, lon1, alt1, lat2, lon2, alt2,...  
    angleunits)  
[...] = elevation(lat1, lon1, alt1, lat2, lon2, alt2,...  
    angleunits, distanceunits)  
[...] = elevation(lat1, lon1, alt1, lat2, lon2, alt2, angleunits,...  
    ellipsoid)
```

```
[elevationangle, slanrange, azimuthangle] = ...  
    elevation(lat1, lon1, alt1, lat2, lon2, alt2) computes the  
elevation angle, slant range, and azimuth angle of point 2 (with geodetic  
coordinates lat2, lon2, and alt2) as viewed from point 1 (with geodetic  
coordinates lat1, lon1, and alt1). alt1 and alt2 are ellipsoidal  
heights. The elevation angle is the angle of the line of sight above the  
local horizontal at point 1. The slant range is the three-dimensional  
Cartesian distance between point 1 and point 2. The azimuth is the  
angle from north to the projection of the line of sight on the local  
horizontal. Angles are in units of degrees, altitudes and distances are  
in meters. The figure of the earth is the default ellipsoid (GRS 80) as  
defined by almanac.
```

Inputs can be vectors of points, or arrays of any shape, but must match in size, with the following exception: Elevation, range, and azimuth from a single point to a set of points can be computed very efficiently by providing scalar coordinate inputs for point 1 and vectors or arrays for point 2.

```
[...] = elevation(lat1, lon1, alt1, lat2, lon2, alt2,...  
    angleunits) uses the string angleunits to specify the units of the  
input and output angles. If omitted, 'degrees' is assumed.
```

```
[...] = elevation(lat1, lon1, alt1, lat2, lon2, alt2,...  
    angleunits, distanceunits) uses the string distanceunits  
to specify the altitude and slant-range units. If omitted, 'meters' is  
assumed. Any units string recognized by unitsratio may be used.
```



```
[...] = elevation(lat1, lon1, alt1,  
lat2, lon2, alt2, angleunits,...
```

ellipsoid) uses the vector ellipsoid, with form [semimajor axis, eccentricity], to specify the ellipsoid. If ellipsoid is supplied, the altitudes must be in the same units as the semimajor axis and the slant range will be returned in these units. If ellipsoid is omitted, the default earth ellipsoid defined by azimuth is used and distances are in meters unless otherwise specified.

---

**Note** The line-of-sight azimuth angles returned by elevation will generally differ slightly from the corresponding outputs of azimuth and distance, except for great-circle azimuths on a spherical earth.

---

## Example

What is the elevation angle of a point 90 degrees distant when both the observer and target are 1000 km altitude above the Earth?

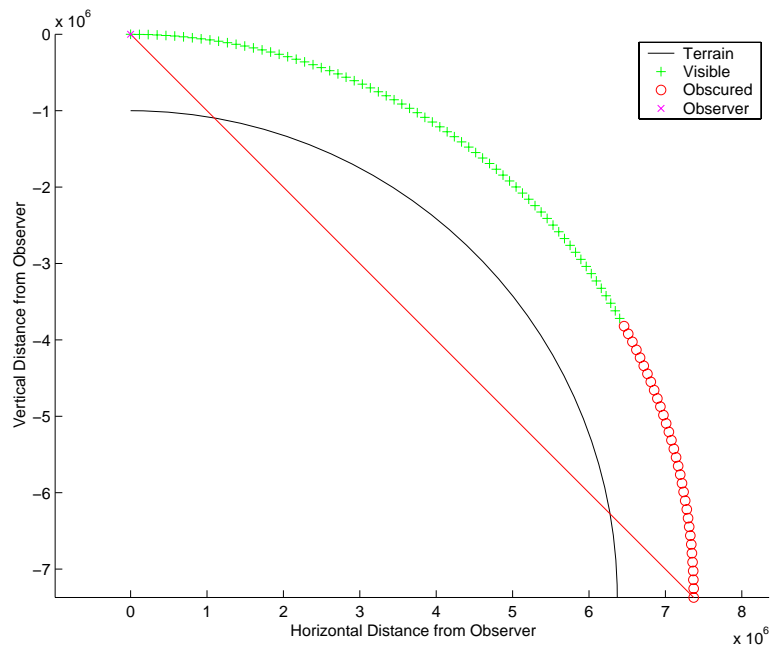
```
lat1 = 0; lon1 = 0; alt1 = 1000*1000;  
lat2 = 0; lon2 = 90; alt2 = 1000*1000;  
elevang = elevation(lat1,lon1,alt1,lat2,lon2,alt2)
```

```
elevang =  
-45
```

Visually check the result using the los2 line of sight function. Construct a data grid of zeros to represent the Earth's surface. The los2 function with no output arguments creates a figure displaying the geometry.

```
Z = zeros(180,360); refvec = [1 90 -180];  
los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt1);
```

# elevation



**See Also** almanac, azimuth, distance

## Purpose

Geographic ellipse from center, semimajor axes, eccentricity, and azimuth

## Syntax

```
[lat,lon] = ellipse1(lat0,lon0,ellipse)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,units)
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,
    units,npts)
[lat,lon] = ellipse1(track,...)
mat = ellipse1(...)
```

`[lat,lon] = ellipse1(lat0,lon0,ellipse)` computes ellipses with a center at the point `lat0, lon0`. The ellipse is defined by the third input, which is of the form `[semimajor-axis, eccentricity]`. The `lat0, lon0` inputs can be scalar or column vectors. The eccentricity input can be a two-element row vector or a two-column matrix. The ellipse input must have the same number of rows as the input `lat0` and `lon0`. The input semimajor axis is in degrees of arc length on a sphere. All ellipses are oriented so that their semimajor axis lies due north.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset)` computes the ellipses where the semimajor axis is rotated from due north by an azimuth `offset`. The `offset` angle is measured clockwise from due north. If `offset=[]`, then no offset is assumed.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az)` uses the input `az` to define the ellipse arcs computed. The arc azimuths are measured clockwise from due north. If `az` is a column vector, then the arc length is computed from due north. If `az` is a two-column matrix, then the ellipse arcs are computed starting at the azimuth in the first column and ending at the azimuth in the second column. If `az=[]`, then a complete ellipse is computed.

`[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid)` computes the ellipse on the ellipsoid defined by the input `ellipsoid`

# ellipse1

---

vector, of the form [semimajor-axis, eccentricity]. If omitted, the unit sphere, `ellipsoid=[1 0]`, is assumed. When an ellipsoid is supplied, the input semimajor axis must be in the same units as the ellipsoid semimajor axes. In this calling form, the units of the ellipse semimajor axis are not assumed to be in degrees.

```
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,units),  
[lat,lon] = ellipse1(lat0,lon0,ellipse,offset,az,units),  
and [lat,lon] =  
ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,units)
```

 are all valid calling forms, which use the input *units* to define the angle units of the inputs and outputs. If omitted, 'degrees' is assumed.

```
[lat,lon] =  
ellipse1(lat0,lon0,ellipse,offset,az,ellipsoid,units,npts)
```

 uses the input *npts* to determine the number of points per ellipse computed. The input *npts* is a scalar, and if omitted, *npts*=100.

```
[lat,lon] = ellipse1(track,...)
```

 uses the *track* string to define either a great circle or rhumb line distance from the ellipse center. If *track* = 'gc', then great circle distances are computed. If *track* = 'rh', then rhumb line distances are computed. If omitted, 'gc' is assumed.

`mat = ellipse1(...)` returns a single output argument where `mat=[lat lon]`. This is useful if only one ellipse is computed.

## Example

Create and plot the small ellipse centered at (0°,0°), with a semimajor axis of 10° and a semiminor axis of 5°.

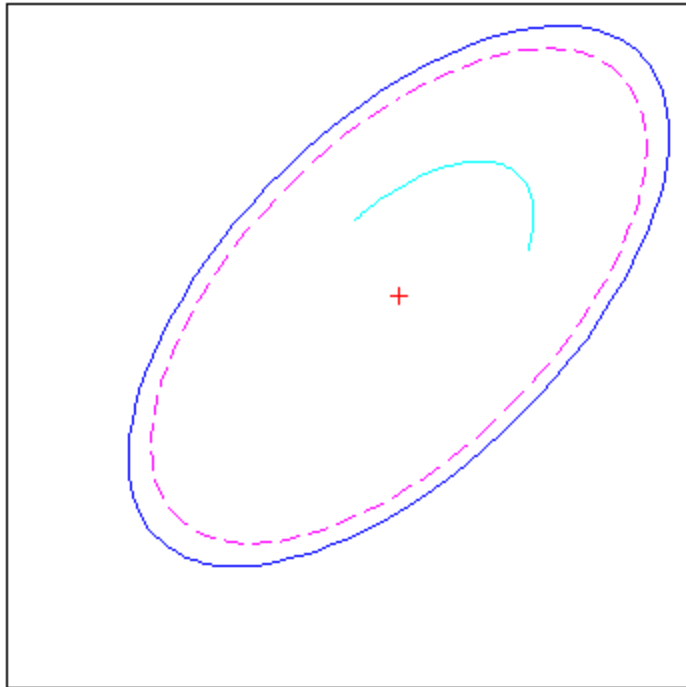
```
axesm mercator  
ecc = axes2ecc(10,5);  
plotm(0,0,'r+')  
[elat,elon] = ellipse1(0,0,[10 ecc],45);  
plotm(elat,elon)
```

If the desired radius is known in some nonangular distance unit, use the radius returned by the `almanac` function as the ellipsoid input to set the range units (use an empty azimuth entry to specify a full ellipse).

```
earthradius = almanac('earth','radius','nm');
[elat,elon] = ellipse1(0,0,[550 ecc],45,[],earthradius);
plotm(elat,elon,'m--')
```

For just an arc of the ellipse, enter an azimuth range:

```
[elat,elon] = ellipse1(0,0,[5 ecc],45,[-30 70]);
plotm(elat,elon,'c-')
```



## Remarks

This function extends the concept of the small circle, which is the locus of all points at an equal surface distance, to a “small ellipse.” You construct the small ellipse by computing the locus of points for which the distance from the center point varies as the parametric description of the ellipse.

# ellipse1

---

You can define multiple circles from a single starting point by providing scalar `lat0`, `lon0` inputs and a two-column matrix for the ellipse definitions.

## See Also

`scircle1`, `track1`, `axes2ecc`

**Purpose**

Fill in regular data grid from seed values and locations

**Syntax**

```
newgrid = encodem(Z,seedmat)
newgrid = encodem(Z,seedmat,stopvals)
```

`newgrid = encodem(Z,seedmat)` fills in regions of the input data grid, `Z`, with desired new values. The boundary consists of the edges of the matrix and any entries with the value 1. The *seeds*, or starting points, and the values associated with them, are specified by the three-column matrix `seedmat`, the rows of which have the form [row column value].

`newgrid = encodem(Z,seedmat,stopvals)` allows you to specify a vector, `stopvals`, of stopping values. Any value that is an element of `stopvals` will act as a boundary.

**Description**

This function *fills in* regions of data grids with desired values. If a *boundary* exists, the new value replaces all entries in all four directions until the boundary is reached. The boundary is made up of selected stopping values and the edges of the matrix. The new value tries to flood the region exhaustively, stopping only when no new spaces can be reached by moving up, down, left, or right without hitting a stopping value.

**Examples**

For this imaginary map, fill in the upper right region with 7s and the lower left region with 3s:

```
Z = eye(4)
```

```
Z =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

```
newgrid = encodem(Z,[4,1,3; 1,4,7])
```

```
newgrid =
     1     7     7     7
```

# encodem

---

3	1	7	7
3	3	1	7
3	3	3	1

## See Also

getseeds, imbedm



**Purpose** Accuracy in angle units for certain map computations

**Syntax** epsm  
 epsm(*units*)

epsm is the limit of map angular precision. It is useful in avoiding trigonometric singularities, among other things.

epsm(*units*) returns the same angle in units corresponding to any valid angle units string. The default is 'degrees'.

**Examples** The value of epsm is  $10^{-6}$  degrees. To put this in perspective, in terms of an angular arc length, the distance is

```
epsmkm = deg2km(epsm)

epsmkm =
    1.1119e-04      % kilometers
```

This is about 11 centimeters, a very small distance on a global scale.

**See Also** roundn

## Purpose

Convert from equal area to Greenwich coordinates

## Syntax

```
[lat,lon] = eqa2grn(x,y)
[lat,lon] = eqa2grn(x,y,origin)
[lat,lon] = eqa2grn(x,y,origin,ellipsoid)
[lat,lon] = eqa2grn(x,y,origin,units)
mat = eqa2grn(x,y,origin...)
```

[lat,lon] = eqa2grn(x,y) converts the equal-area coordinate points *x* and *y* to the Greenwich (standard geographic) coordinates *lat* and *lon*.

[lat,lon] = eqa2grn(x,y,origin) specifies the location in the Greenwich system of the *x-y* origin (0,0). The two-element vector *origin* must be of the form [latitude longitude]. The default places the origin at the Greenwich coordinates (0°,0°).

[lat,lon] = eqa2grn(x,y,origin,ellipsoid) specifies the two-element ellipsoid vector describing the ellipsoidal model of the figure of the Earth. The *ellipsoid* is spherical by default.

[lat,lon] = eqa2grn(x,y,origin,units) specifies the units for the outputs, where *units* is any valid angle units string. The default value is 'degrees'.

mat = eqa2grn(x,y,origin...) packs the outputs into a single variable.

## Description

This function converts data from equal-area *x-y* coordinates to geographic (latitude-longitude) coordinates. The opposite conversion can be performed with `grn2eqa`.

## Examples

```
[lat,lon] = eqa2grn(.5,.5)

lat =
    30.0000
lon =
    28.6479
```

**See Also**      grn2eqa, hista

## Purpose

Read global 5-min or 2-min digital terrain data

## Syntax

```
[Z, refvec] = etopo
[Z, refvec] = etopo(samplefactor)
[Z, refvec] = etopo(samplefactor, latlim, lonlim)
[Z, refvec] = etopo(directory, ...)
[Z, refvec] = etopo(file, ...)
```

[Z, refvec] = etopo reads the ETOPO data for the entire world from the ETOPO data in the current directory. The current directory is searched first for ETOPO2 binary data, followed by ETOPO5 binary data, followed by ETOPO5 ASCII data from the file names etopo5.northern.bat and etopo5.southern.bat. Once a match is found the data is read. The data grid, Z, is returned as an array of elevations. Data values are in whole meters, representing the elevation of the center of each cell. refvec is the associated three-element referencing vector that geolocates Z.

[Z, refvec] = etopo(samplefactor) reads the data for the entire world, downsampling the data by samplefactor. samplefactor is a scalar integer, which when equal to 1 gives the data at its full resolution (1080 by 4320 values for ETOPO5 data and 5400 by 10800 values for ETOPO2 data). When samplefactor is an integer n greater than one, every n<sup>th</sup> point is returned. samplefactor must divide evenly into the number of rows and columns of the data file. If samplefactor is omitted or empty, it defaults to 1.

[Z, refvec] = etopo(samplefactor, latlim, lonlim) reads the data for the part of the world within the specified latitude and longitude limits. The limits of the desired data are specified as two-element vectors of latitude, latlim, and longitude, lonlim, in degrees. The elements of latlim and lonlim must be in ascending order. lonlim must be specified in the range [0 360] for ETOPO5 data and [-180 180] for ETOPO2 data. If latlim is empty the latitude limits are [-90 90]. If lonlim is empty, the longitude limits are determined by the file type.

---

`[Z, refvec] = etopo(directory, ...)` allows the path for the ETOPO data file to be specified by directory rather than the current directory.

`[Z, refvec] = etopo(file, ...)` reads the ETOPO data from file, where file is a string or a cell array of strings containing the name or names of the ETOPO data files.

## Background

ETOPO5 is a global database of elevations and depths on a regular 5-minute grid. It is a compilation of data from a variety of different sources, including the U.S. Naval Oceanographic Office, U.S. Defense Mapping Agency, U.S. Navy Fleet Numerical Oceanographic Center, Bureau of Mineral Resources, Australia, and the Department of Industrial and Scientific Research, New Zealand. These databases were assembled by Margo Edwards at Washington University, St. Louis, Missouri.

## Remarks

ETOPO5 data values are in whole meters, representing the elevation of the center of each cell. Some parts of the world are represented by data with a horizontal resolution as coarse as 1 degree by 1 degree. The vertical resolution varies from 1 meter for Australia and New Zealand to as much as 150 meters for parts of Africa, Asia, and South America. Oceanographic data in areas shallower than 200 meters contains little detail, because of how depth contours were converted to gridded depths.

ETOPO5 is superseded by ETOPO2 and the TerrainBase digital terrain model. See the tbase external interface function for more information.

---

**Note** You can find links to more information about ETOPO files in the following page at the MathWorks Web site:  
<http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

### Example 1

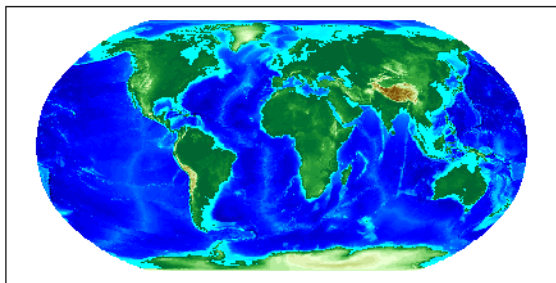
```
% Read and display the ETOPO5 data from the directory 'etopo5'  
% downsampled by a factor of 10.
```

```
[Z, refvec] = etopo('etopo5',10);  
whos
```

Name	Size	Bytes	Class
Z	216x432	746496	double array
refvec	1x3	24	double array

Grand total is 93315 elements using 746520 bytes

```
axesm robinson  
geoshow(Z, refvec, 'DisplayType', 'surface');  
colormap(demcmap(Z));
```



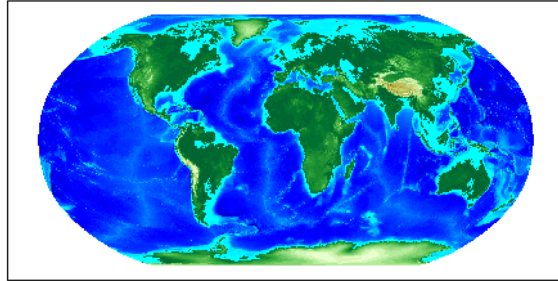
## Example 2

```
% From the current directory, read and display the  
% ETOPO2 binary data downsampled by a factor of 10.  
cd etopo2  
[Z, refvec] = etopo('ETOPO2.dos.bin', 10);  
whos
```

Name	Size	Bytes	Class
Z	540x1080	4665600	double array
refvec	1x3	24	double array

```
figure; axesm robinson  
geoshow(Z, refvec, 'DisplayType', 'surface');
```

```
colormap(demcmap(Z));
```



**See Also** `gtopo30`, `tbase`, `usgsdem`

## Purpose

Read global 5-min digital terrain data

## Syntax

---

**Note** etopo5 is obsolete; use etopo.

---

```
[Z, refvec] = etopo5
[Z, refvec] = etopo5(samplefactor)
[[Z, refvec] = etopo5(samplefactor, latlim, lonlim)
[Z, refvec] = etopo5(directory, ...)
[Z, refvec] = etopo5(file, ...)
```

[Z, refvec] = etopo5 reads the topography data for the entire world for the data in the current directory. The current directory is searched first for ETOPO2 binary data, followed by ETOPO5 binary data, followed by ETOPO5 ASCII data from the file names etopo5.northern.bat and etopo5.southern.bat. Once a match is found the data is read. The data grid, Z, is returned as an array of elevations. Data values are in whole meters, representing the elevation of the center of each cell. refvec is the associated three-element referencing vector that geolocates Z.

[Z, refvec] = etopo5(samplefactor) reads the data for the entire world, downsampling the data by samplefactor. samplefactor is a scalar integer, which when equal to 1 gives the data at its full resolution (1080 by 4320 values). When samplefactor is an integer n greater than one, every n<sup>th</sup> point is returned. samplefactor must divide evenly into the number of rows and columns of the data file. If samplefactor is omitted or empty, it defaults to 1.

[[Z, refvec] = etopo5(samplefactor, latlim, lonlim) reads the data for the part of the world within the specified latitude and longitude limits. The limits of the desired data are specified as two-element vectors of latitude, latlim, and longitude, lonlim, in degrees. The elements of latlim and lonlim must be in ascending order. If latlim is empty the latitude limits are [-90 90]. lonlim must be specified in the range [0 360]. If lonlim is empty, the longitude limits are [0 360].



`[Z, refvec] = etopo5(directory, ...)` allows the path for the data file to be specified by `directory` rather than the current directory.

`[Z, refvec] = etopo5(file, ...)` reads the data from `file`, where `file` is a string or a cell array of strings containing the name or names of the data files.

ETOPO5 is being superseded by ETOPO2 and the TerrainBase digital terrain model. See the `tbase` external interface function for more information.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web Site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>

---

## Examples

### Example 1

Read every tenth point in the data set:

```
% Read and display the ETOPO5 data from the directory 'etopo5'
% downsampled by a factor of 10.
[Z, refvec] = etopo5('etopo5',10);
axesm merc
geoshow(Z, refvec, 'DisplayType', 'surface');
colormap(demcmap(Z));
```

### Example 2

Read in data for Korea and Japan at the full resolution:

```
samplefactor = 1; latlim = [30 45]; lonlim = [115 145];
[Z,refvec] = etopo5(samplefactor,latlim,lonlim);
whos Z
```

Name	Size	Bytes	Class
Z	180x360	518400	double array

## See Also

`etopo`, `gtopo30`, `tbase`, `usgsdem`

# extractfield

---

**Purpose** Field values from structure array

**Syntax** `a = extractfield(s, name)`

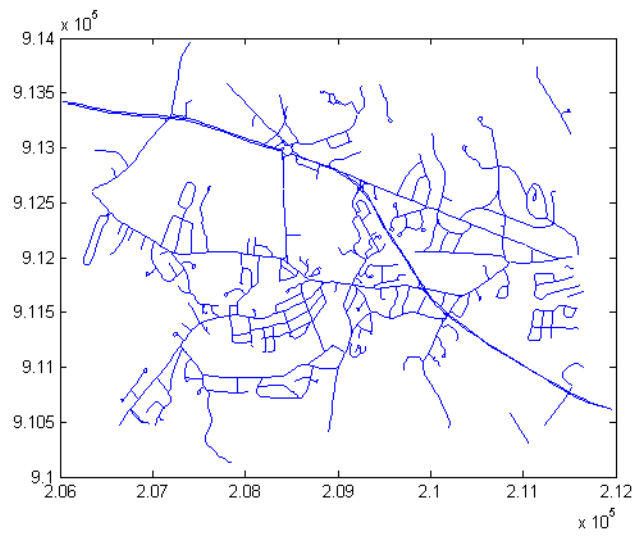
`a = extractfield(s, name)` returns the field values specified by the field named `name` into the 1-by-`n` output array `a`. `n` is the total number of elements in the field name of structure `s`, that is, `n = numel([s(:).(name)])`. `name` is a case-sensitive string defining the field name of the structure `s`. `a` is a cell array if any field values in the field name contain a string or if the field values are not uniform in type; otherwise `a` is the same type as the field values. The shape of the input field is not preserved in `a`.

## Examples

```
% Plot the X, Y coordinates of the road's shape
roads = shaperead('concord_roads.shp');
plot(extractfield(roads, 'X'), extractfield(roads, 'Y'));

% Extract the names of the roads
roads = shaperead('concord_roads.shp');
names = extractfield(roads, 'STREETNAME');

% Extract a mix-type field into a cell array
S(1).Type = 0;
S(2).Type = logical(0);
mixedType = extractfield(S, 'Type');
```



**See Also** `struct`, `shaperead`

# extractm

---

## Purpose

Vector data from Version 1 geographic data structure

## Syntax

```
[lat,lon] = extractm(gstruct,object)
[lat,lon] = extractm(gstruct,objects)
[lat,lon] = extractm(gstruct,objects,'exact')
[lat,lon,indx] = extractm(gstruct)
[lat,lon,indx] = extractm(...)
mat = extractm(...)
```

`[lat,lon] = extractm(gstruct,object)` extracts vector data from those entries in the Mapping Toolbox geographic data structure that have tags beginning with the *object* string. The output vectors use NaNs to separate the entries in the map structure. Matches of the tag string must be vector data (lines and patches) to be included in the output.

`[lat,lon] = extractm(gstruct,objects)` where *objects* is a character array, allows more than one object to be the basis for the search.

`[lat,lon] = extractm(gstruct,objects,'exact')` requires an exact match to extract data.

`[lat,lon,indx] = extractm(gstruct)` extracts all vector data from the input map structure.

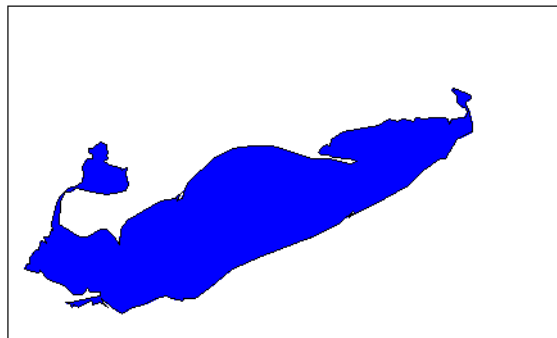
`[lat,lon,indx] = extractm(...)` also returns the vector *indx* identifying the entries in the structure that meet the selection criteria.

`mat = extractm(...)` returns the vector data in a single, two-column matrix, in which the first column contains latitudes and the second column longitudes.

## Example

Extract the District of Columbia from the low-resolution U.S. vector data:

```
load greatlakes
[lat, lon] = extractm(greatlakes, 'Erie');
axesm mercator
geoshow(lat,lon, 'DisplayType','polygon', 'FaceColor','blue')
```

**Remarks**

A Mapping Toolbox Version 1 geographic data structure is a MATLAB structure that can contain line, patch, text, regular data grid, geolocated data grid, light objects, and certain fixed attributes. Starting in Version 2, Mapping Toolbox updated this structure to a Version 2 geographic data structure, which has greater flexibility.

**See Also**

`extractfield`, `geoshow`, `mapshow`, `updategeostruct`, `mlayers`

# fill3m

---

## Purpose

Project filled 3-D patch objects on map axes

## Syntax

```
h = fill3m(lat,lon,z,cdata)
```

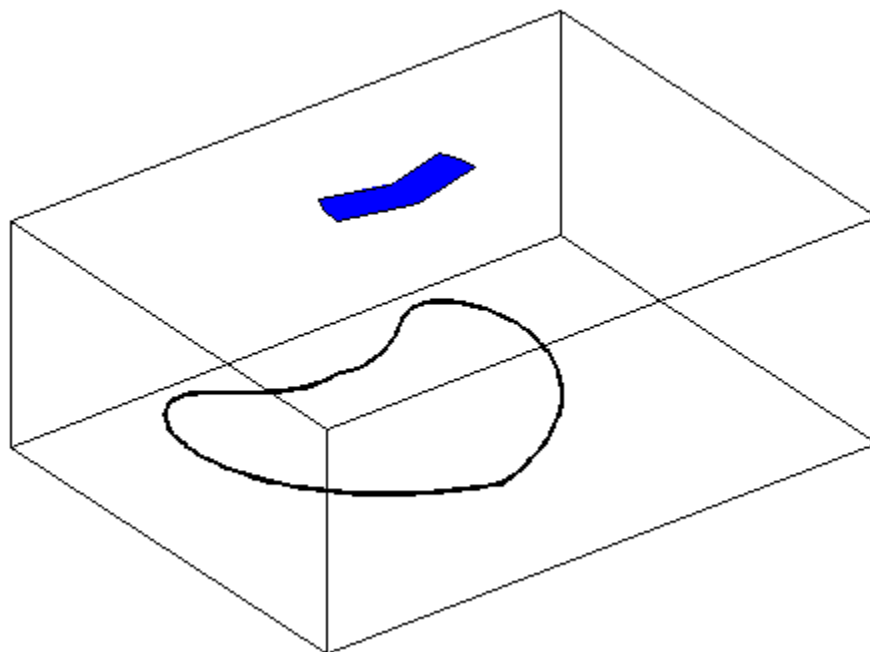
```
h = fill3m(lat,lon,z,PropertyName,PropertyValue,...)
```

`h = fill3m(lat,lon,z,cdata)` projects and displays any patch object with vertices defined by vectors `lat` and `lon` to the current map axes. The scalar `z` indicates the altitude plane at which the patch is displayed. The input `cdata` defines the patch face color. The patch handle or handles, `h`, can be returned.

`h = fill3m(lat,lon,z,PropertyName,PropertyValue,...)` allows any property name/property value pair supported by `patch` to be assigned to the `fill3m` object.

## Examples

```
lat = [30 15 0 0 0 15 30 30]';  
lon = [-60 -60 -60 0 60 60 60 0]';  
axesm bonne; framem  
view(3)  
fill3m(lat,lon,2,'b')
```



**See Also**

fillm, patchesm, patchm

# fillm

---

**Purpose** Project filled 2-D patch objects on map axes

**Syntax**

```
h = fillm(lat,lon,cdata)
h = fillm(lat,lon,'PropertyName',PropertyValue,...)
```

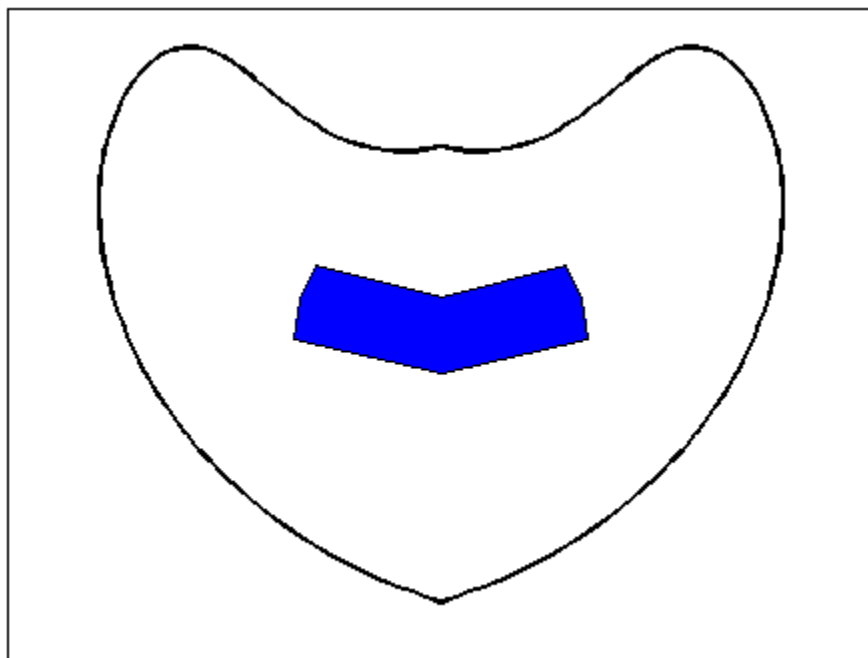
`h = fillm(lat,lon,cdata)` projects and displays any patch object with vertices defined by the vectors `lat` and `lon` to the current map axes. The input `cdata` defines the patch face color. The patch handle or handles, `h`, can be returned.

`h = fillm(lat,lon,'PropertyName',PropertyValue,...)` allows any property name/property value pair supported by patch to be assigned to the `fillm` object.

**Examples**

```
lat = [30 15 0 0 0 15 30 30]';
lon = [-60 -60 -60 0 60 60 60 0]';
axesm bonne; framem
fillm(lat,lon,'b')
```



**See Also**

`fill13m`, `patchesm`, `patchm`

# filterm

---

**Purpose** Filter latitudes and longitudes based on underlying data grid

**Syntax** `[newlat,newlong] = filterm(lat,long,Z,refvec,allowed)`  
`[newlat,newlong] = filterm(lat,long,Z,refvec,allowed)` filters geographic data based upon the corresponding entries of a regular data grid, `Z`, with a three-element referencing vector `refvec`. The data locations to be filtered are input in the vectors `lat` and `lon`. For those locations corresponding to entries of `map` equal to one of the values contained in the vector `allowed`, an output location is returned in `newlat` and `newlon`. Those locations not corresponding to such entries of `Z` are not returned in the outputs.

**Examples** Filter a random set of 100 geographic points. Use the topo map for starters:

```
load topo
```

Then generate 100 random points:

```
lat = -90+180*rand(100,1);  
long = -180+360*rand(100,1);
```

Make a land map, which is 1 where `topo>0` elevation:

```
land = topo>0;  
[newlat,newlong] = filterm(lat,long,land,topolegend,1);  
size(newlat)
```

```
ans =  
    15     1
```

15 of the 100 random points fall on *land*.

**See Also** `hista`, `histr`

**Purpose**

Latitudes and longitudes of nonzero data grid elements

**Syntax**

```
[lat,lon] = findm(Z,refvec)
[lat,lon,val] = findm(Z,refvec)
[lat,lon,val] = findm(Lats,Lons,Z)
mat = findm(...)
```

`[lat,lon] = findm(Z,refvec)` returns latitude and longitude vectors `lat` and `lon`, which provide the locations of all nonzero entries of the regular data grid `Z`, with three-element referencing vector `refvec`.

`[lat,lon,val] = findm(Z,refvec)` also returns the values `val` of the data grid corresponding to the `lat` and `lon` locations.

`[lat,lon,val] = findm(Lats,Lons,Z)` removes the *regular* matrix restriction. Two matrices, `Lats` and `Lons`, the same size as `Z`, must provide cell-by-cell latitude and longitude coordinates matched with the corresponding entries of `Z`.

`mat = findm(...)` returns a single matrix `mat` of the form `[lat,lon]`.

**Description**

This function works in two modes: with a regular data grid and with a geolocated data grid.

**Example**

The entered map can also be the result of a logical statement. Where is elevation greater than 5500 meters?

```
load topo
[lat lon] = findm((topo>5500),topolegend)

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

These points are in the Himalayas. Find the grid values at these locations with `setpostn`:

# findm

---

```
heights = topo(setpostn(topo, topolegend, lat, lon))
```

```
heights =  
    5559  
    5515  
    5523  
    5731
```

You must use a regular data grid in order to retrieve the elevations from `setpostn`.

## See Also

`find` (MATLAB function), `setpostn`

**Purpose** Read Federal Information Processing Standard (FIPS) name file used with TIGER thinned boundary files

**Syntax**

```
struc = fipsname
struc = fipsname(filename)

struc = fipsname opens a file selection window to pick the file, reads the FIPS codes, and returns them in a structure.

struc = fipsname(filename) reads the specified file.
```

**Background** The TIGER thinned boundary files provided by the U.S. Census use FIPS codes to identify geographic entities. This function reads the FIPS files as provided with the TIGER files. These files generally have names of the format *\_name.dat*.

**Remarks** The FIPS name files, along with TIGER thinned boundary files, are available over the Internet.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

**Example**

```
struc = fipsname('st_name.dat')

struc =
1x57 struct array with fields:
    name
    id

s(1)

ans =
    name: 'Alabama'
    id: 1
```

# flat2ecc

---

**Purpose** Eccentricity of ellipse with given flattening

**Syntax** `eccentricity = flat2ecc(flattening)`  
`eccentricity = flat2ecc(flattening)` returns the equivalent eccentricity for the input `flattening`. If the input, `flattening`, is a two-column vector, only the second column is used. This allows two-element vectors to be used as rows of the input, since the form `[semimajor-axis, flattening]` is a complete representation of an ellipsoid (but is not the standard form for ellipsoid vectors in Mapping Toolbox). In all other cases, all columns of the input are used.

**Description** Flattening and eccentricity are two methods of defining an ellipsoid.

**Example** `e = flat2ecc(0.003353)`

```
e =  
    0.08182149712026
```

This eccentricity is the default value for the Earth.

**See Also** `almanac`, `ecc2flat`, `ecc2n`, `majaxis`

## Purpose

Insert points along date line to pole

## Syntax

```
[latf,lonf] = flatearthpoly(lat,lon)
```

```
[latf,lonf] = flatearthpoly(lat,lon,longitudeOrigin)
```

[latf,lonf] = flatearthpoly(lat,lon) trims NaN-separated polygons specified by the latitude and longitude vectors lat and lon to the limits [-180 180] in longitude and [-90 90] in latitude, inserting straight segments along the +/- 180-degree meridians and at the poles. Inputs and outputs are in degrees.

[latf,lonf] = flatearthpoly(lat,lon,longitudeOrigin) centers the longitude limits on the longitude specified by the scalar longitudeOrigin.

## Remarks

The polygon topology for the input vectors must be valid. This means that vertices for outer rings (main polygon or “island” polygons) must be in clockwise order, and any inner rings (“lakes”) must run in counterclockwise order for the function to work properly. You can use the ispolycw function to check whether or not your lat, lon vectors meet this criterion, and the poly2cw and poly2ccw functions to correct any that run in the wrong direction.

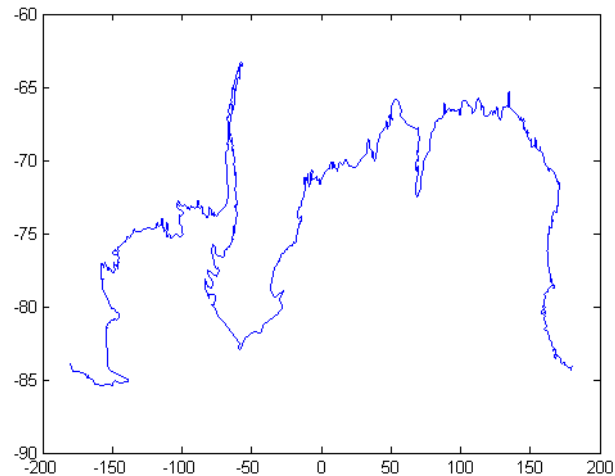
## Example

Vector data for geographic objects that encompass a pole will inevitably encounter or cross the date line. While Mapping Toolbox properly displays such polygons, they can cause problems for functions like the polygon intersection and Boolean operations that work with Cartesian coordinates. When these polygons are treated as Cartesian coordinates, the date line crossing results in a spurious line segment, and the polygon displayed as a patch does not have the interior filled correctly.

```
antarctica = shaperead('landareas', 'UseGeoCoords', true,...
    'Selector', {@(name) strcmp(name,'Antarctica')}, 'Name');
figure; plot(antarctica.Lon, antarctica.Lat); ylim([-100 -60])
```

# flatearthpoly

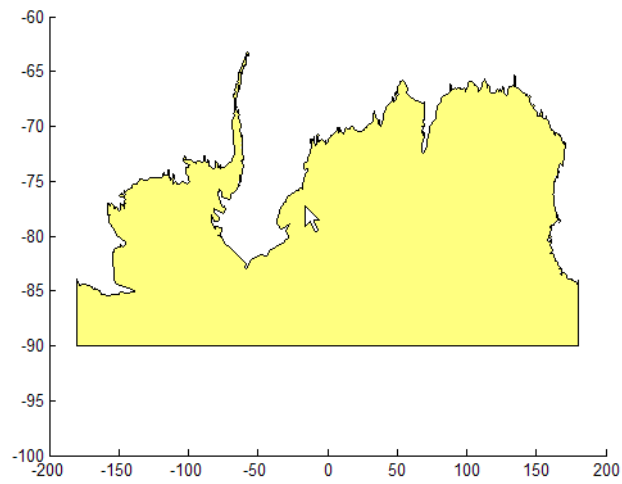
---



The polygons can be reformatted more appropriately for Cartesian coordinates using the `flatearthpoly` function. The result resembles a map display on a cylindrical projection. The polygon meets the date line, drops down to the pole, sweeps across the longitudes at the pole, and follows the date line up to the other side of the date line crossing.

```
[latf, lonf] = flatearthpoly(antarctica.Lat', antarctica.Lon');  
figure; mapshow(lonf, latf, 'DisplayType', 'polygon')  
ylim([-100 -60])
```





**See Also**

`ispolycw`, `maptrimp`, `poly2cw`, `poly2ccw`

# framem

---

## Purpose

Toggle and control display of map frame

## Syntax

```
framem  
framem('on')  
framem('off')  
framem('reset')  
framem(linespec)  
framem(PropertyName,PropertyValue,...)
```

`framem` toggles the visibility of the map frame by setting the map axes property `Frame` to 'on' or 'off'. The default setting for map axes is 'off'.

`framem('on')` sets the map axes property `Frame` to 'on'.

`framem('off')` sets the map axes property `Frame` to 'off'.

When called with the string argument 'off', the map axes property `Frame` is set to 'off'.

`framem('reset')` resets the entire frame using the current properties. This is essentially a *refresh* option.

`framem(linespec)` sets the map axes `FEdgeColor` property to the color component of any *linespec* string recognized by the MATLAB `line` function.

`framem(PropertyName,PropertyValue,...)` sets the appropriate map axes properties to the desired values. These property names and values are described on the `axesm` reference page.

## Remarks

You can also create or alter map frame properties using the `axesm` or `setm` functions.

## See Also

`axesm`, `setm`

**Purpose** Convert angles from degrees

**Syntax** `[angle1, angle2, ...] = fromDegrees(toUnits, angle1InDegrees, angle2InDegrees, ...)`

`[angle1, angle2, ...] = fromDegrees(toUnits, angle1InDegrees, angle2InDegrees, ...)` converts `angle1InDegrees`, `angle2InDegrees`, ... from degrees to the specified output ("to") angle units. *toUnits* can be either 'degrees' or 'radians' and may be abbreviated. The inputs `angle1InDegrees`, `angle2InDegrees`, ... and their corresponding outputs are numeric arrays of various sizes, with `size(angleN)` matching `size(angleNInDegrees)`.

**See Also** `deg2rad`, `fromRadians`, `toDegrees`, `toRadians`

# fromRadians

---

**Purpose** Convert angles from radians

**Syntax**

```
[angle1, angle2, ...] = fromRadians(toUnits,  
angle1InRadians,  
angle2InRadians, ...)
```

[angle1, angle2, ...] = fromRadians(*toUnits*, angle1InRadians, angle2InRadians, ...) converts angle1InRadians, angle2InRadians, ... from radians to the specified output ("to") angle units. *toUnits* can be either 'degrees' or 'radians' and may be abbreviated. The inputs angle1InRadians, angle2InRadians, ... and their corresponding outputs are numeric arrays of various sizes, with size(angleN) matching size(angleNInRadians).

**See Also** fromDegrees, rad2deg, toDegrees, toRadians

**Purpose**

Center and radius of great circle

**Syntax**

```
[centerlat,centerlong,radius] = gc2sc(lat,long,az)
[centerlat,centerlong,radius] = gc2sc(lat,long,az,units)

[centerlat,centerlong,radius] = gc2sc(lat,long,az) returns the
small circle notation for great circles entered in great circle notation.

[centerlat,centerlong,radius] = gc2sc(lat,long,az,units)
specifies the standard angle unit string. The default value is 'degrees'.
```

**Description**

Great circles are a subcategory of small circles, having a radius of 90°. Because of the computational circumstances under which these objects often arise, however, two different notations are convenient.

*Great circle notation* consists of a point on the great circle and the azimuth at that point along which the great circle proceeds.

*Small circle notation* consists of a center point and a radius in units of angular arc length.

**Examples**

Given a great circle passing through (25°S,70°W) on an azimuth of 45°, how can it be represented in small circle notation?

```
[newlat,newlong,range] = gc2sc(-25,-70,45)

newlat =
    -39.8557
newlong =
     42.9098
range =
     90
```

A great circle always bisects the sphere. As a demonstration of this statement, consider the equator, which passes through any point with a latitude of 0° and proceeds on an azimuth of 90° or 270°. In small circle notation, this is

```
[newlat,newlong,range] = gc2sc(0,-70,270)
```

```
newlat =  
    90  
newlong =  
    -145.9638  
range =  
    90
```

Not surprisingly, the small circle is centered on the North Pole. As always, at the poles, the longitude is arbitrary, because of the convergence of the meridians.

## Remarks

Note that the center coordinates returned by this function always lead to one of two possibilities. Since the great circle bisects the sphere, the antipode of the returned point is also a center with a radius of  $90^\circ$ . In the above example, the South Pole would also be a suitable center for the equator in small circle notation.

## See Also

antipode, gcxgc, gcxsc, rhxrh, crossfix

**Purpose**

Current map projection structure

**Syntax**

```
mapstruct = gcm
```

```
mapstruct = gcm(hndl)
```

`mapstruct = gcm` returns the map axes *map structure*, which contains the settings for all the current map axes properties.

`mapstruct = gcm(hndl)` specifies the map axes by axes handle.

**Examples**

Establish a map axes with default values, then look at the structure:

```
axesm mercator
```

```
mapstruct = gcm
```

```
mapstruct =
```

```
  mapprojection: 'mercator'  
        zone: []  
        angleunits: 'degrees'  
        aspect: 'normal'  
  fixedorient: []  
        geoid: [1 0]  
  maplatlimit: [-86 86]  
  maplonlimit: [-180 180]  
  mapparallels: 0  
        nparallels: 1  
        origin: [0 0 0]  
  falsenorthing: 0  
  falseeasting: 0  
  scalefactor: 1  
        trimlat: [-86 86]  
        trimlon: [-180 180]  
        frame: 'off'  
        ffill: 100  
  fedgecolor: [0 0 0]  
  ffacecolor: 'none'  
  flatlimit: [-86 86]  
  flinewidth: 2
```

```
    flonlimit: [-180 180]
      grid: 'off'
    galtitude: Inf
      gcolor: [0 0 0]
    glinestyle: ':'
    glinewidth: 0.5000000000000000
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'
  labelrotation: 'off'
  meridianlabel: 'off'
mlabellocation: 30
mlabelparallel: 86
  mlabelround: 0
  parallellabel: 'off'
plabellocation: 15
plabelmeridian: -180
  plabelround: 0
```

**Remarks**

You create map structure properties with the `axesm` function. You can query them with the `getm` function and modify them with the `setm` function.



**See Also**      axesm, getm, setm

# gcpmap

---

**Purpose** Current mouse point from map axes

**Syntax**

```
pt = gcpmap
pt = gcpmap(hndl)
```

pt = gcpmap returns the current point (the location of last button click) of the current map axes in the form [latitude longitude z-altitude].

pt = gcpmap(hndl) specifies the map axes in question by its handle.

**Remarks**

gcpmap works much like the standard MATLAB `get(gca, 'CurrentPoint')`, except that the returned matrix is in [lat lon z], not [x y z].

MATLAB updates the `CurrentPoint` property whenever a button-click event occurs. The pointer does not have to be within the axes, or even the figure window; MATLAB returns the coordinates with respect to the requested axes regardless of the pointer location. Likewise, gcpmap will return values that may look reasonable whether the current point is within the graticule bounds or not, and thus must be used with care.

**Example** Set up a map axes with a graticule and display a world map:

```
axesm robinson
gridm on
geoshow('landareas.shp')
```

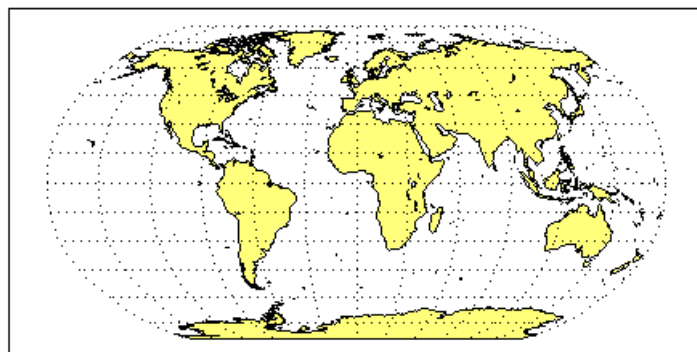
Click somewhere near Boston, Massachusetts to obtain a current point:

```
pt = gcpmap

pt =
    44.171    -69.967         2
    44.171    -69.967         0

whos
```

Name	Size	Bytes	Class	Attributes
pt	2x3	48	double array	



**See Also** `inputm`, Axes Properties

## Purpose

Equally spaced waypoints along great circle

## Syntax

```
[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2)
[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2,nlegs)
pts = gcwaypts(lat1,lon1,lat2,lon2...)
```

`[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2)` returns the coordinates of equally spaced points along a great circle path connecting two endpoints, (lat1,lon1) and (lat2,lon2).

`[lat,lon] = gcwaypts(lat1,lon1,lat2,lon2,nlegs)` specifies the number of equal-length track legs to calculate. `nlegs+1` output points are returned, since a final endpoint is required. The default number of legs is 10.

`pts = gcwaypts(lat1,lon1,lat2,lon2...)` packs the outputs, which are otherwise two-column vectors, into a two-column matrix of the form [latitude longitude]. This format for successive waypoints along a navigational track is called *navigational track format* in this guide. See the `navigational track format` reference page in this section for more information.

## Background

This is a navigational function. It assumes that all latitudes and longitudes are in degrees.

In navigational practice, great circle paths are often approximated by rhumb line segments. This is done to come reasonably close to the shortest distance between points without requiring course changes too frequently. The `gcwaypts` function provides an easy means of finding waypoints along a great circle path that can serve as endpoints for rhumb line segments (track legs).

## Examples

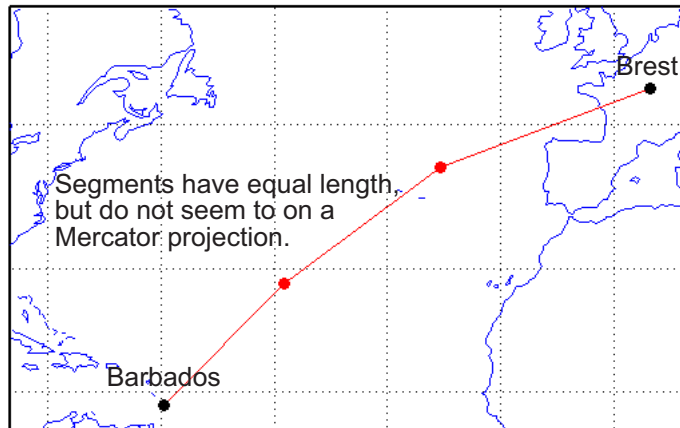
Imagine you own a sailing yacht and are planning a voyage from North Point, Barbados (13.33° N,59.62°W), to Brest, France (48.33°N,4.83°W). To divide the track into three equal-length segments,

```
figure('color','w');
ha = axesm('mapproj','mercator',...
    'maplatlim',[10 55],'maplonlim',[-80 10],...
```

```

'MLineLocation',15,'PLineLocation',15);
axis off, gridm on, framem on;
load coast;
hg = geoshow(lat,long,'displaytype','line','color','b');
% Define point locations for Barbados and Brest
barbados = [13.33 -59.62];
brest = [48.33 4.83];
[l,g] = gcwaypts(barbados(1),barbados(2),brest(1),brest(2),3);
geoshow(l,g,'displaytype','line','color','r',...
'markeredgecolor','r','markerfacecolor','r','marker','o');
geoshow(barbados(1),barbados(2),'DisplayType','point',...
'markeredgecolor','k','markerfacecolor','k','marker','o')
geoshow(brest(1),brest(2),'DisplayType','point',...
'markeredgecolor','k','markerfacecolor','k','marker','o')

```

**See Also**

dreckon, legs, navfix, track

**Purpose**

Intersection points for pairs of great circles

**Syntax**

```
[newlat,newlong] = gcxgc(lat1,long1,az1,lat2,long2,az2)
[newlat,newlong] =
gcxgc(lat1,long1,az1,lat2,long2,az2,units)
```

[newlat,newlong] = gcxgc(lat1,long1,az1,lat2,long2,az2) returns the two intersection points of pairs of great circles input in *great circle notation*. When the two great circles are identical (which is not, in general, apparent by inspection), two NaNs are returned instead and a warning is displayed. For multiple pairings, the inputs must be column vectors.

```
[newlat,newlong] =
gcxgc(lat1,long1,az1,lat2,long2,az2,units) specifies the
standard angle unit string. The default value is 'degrees'.
```

**Description**

For any pair of great circles, there are two possible intersection conditions: the circles are identical or they intersect exactly twice on the sphere.

*Great circle notation* consists of a point on the great circle and the azimuth at that point along which the great circle proceeds.

**Examples**

Given a great circle passing through (10°N,13°E) and proceeding on an azimuth of 10°, where does it intersect with a great circle passing through (0°, 20°E), on an azimuth of -23° (that is, 337°)?

```
[newlat,newlong] = gcxgc(10,13,10,0,20,-23)

newlat =
    14.3105   -14.3105
newlong =
    13.7838  -166.2162
```

Note that the two intersection points are always antipodes of each other. As a simple example, consider the intersection points of two meridians, which are just great circles with azimuths of 0° or 180°:

```
[newlat,newlong] = gcxgc(10,13,0,0,20,180)
```

```
newlat =  
-90    90
```

```
newlong =  
-174.4504    12.5094
```

The two meridians intersect at the North and South Poles, which is exactly correct.

**See Also**

antipode, gc2sc, scxsc, gcxsc, rhxrh, crossfix, polyxpoly

**Purpose**

Intersection points for great and small circle pairs

**Syntax**

`[newlat,newlong] = gcxsc(gclat,gclong,gcaz,sclat,sclong,scrangle)`

`[newlat,newlong] = gcxsc(...,units)`

`[newlat,newlong] = gcxsc(gclat,gclong,gcaz,sclat,sclong,scrangle)` returns the points of intersection of a great circle in *great circle notation* followed by a small circle in *small circle notation*. For multiple pairings, the inputs must be column vectors. The results are two-column matrices with the coordinates of the intersection points. If the circles do not intersect, or are identical, two NaNs are returned and a warning is displayed. If the two circles are tangent, the single intersection point is repeated twice.

`[newlat,newlong] = gcxsc(...,units)` specifies the standard angle unit string. The default value is 'degrees'.

**Description**

For a pairing of a great circle with a small circle, there are four possible intersection conditions: the circles are identical (possible because great circles are a subset of small circles), they do not intersect, they are tangent to each other (the small circle interior to the great circle) and hence they intersect once, or they intersect twice.

*Great circle notation* consists of a point on the great circle and the azimuth at that point along which the great circle proceeds.

*Small circle notation* consists of a center point and a radius in units of angular arc length.

**Examples**

Given a great circle passing through (43°N,0°) and proceeding on an azimuth of 10°, where does it intersect with a small circle centered at (47°N,3°E) with an arc length radius of 12°?

`[newlat,newlong] = gcxsc(43,0,10,47,3,12)`

```
newlat =
    35.5068    58.9143
newlong =
```



-1.6159 5.4039

**See Also**

gc2sc, gcxgc, scxsc, rhxrh, crossfix, polyxpoly

# geodetic2ecef

---

**Purpose** Convert geodetic to geocentric (ECEF) coordinates

**Syntax** `[x, y, z] = geodetic2ecef(phi, lambda, h, ellipsoid)`

**Description** `[x, y, z] = geodetic2ecef(phi, lambda, h, ellipsoid)`  
`[x, y, z] = geodetic2ecef(phi, lambda, h, ellipsoid)`

`[x, y, z] = geodetic2ecef(phi, lambda, h, ellipsoid)` converts geodetic point locations specified by the coordinate arrays `phi` (geodetic latitude in radians), `lambda` (longitude in radians), and `h` (ellipsoidal height) to geocentric Cartesian coordinates `x`, `y`, and `z`. The geodetic coordinates refer to the reference ellipsoid specified by `ellipsoid` (a row vector with the form [semimajor axis, eccentricity]). `h` must use the same units as the semimajor axis; `x`, `y`, and `z` will be expressed in these units also.

**Remarks** The geocentric Cartesian coordinate system is fixed with respect to the Earth, with its origin at the center of the ellipsoid and its  $x$ -,  $y$ -, and  $z$ -axes intersecting the surface at the Equator and the Prime Meridian (geodetic coordinate  $0^\circ, 0^\circ$ ), the Equator at  $90^\circ$  east ( $0^\circ, \pi/2^\circ$ ), and the North Pole ( $\pi/2^\circ, 0^\circ$ ), respectively. A common synonym is Earth-Centered, Earth-Fixed coordinates, or ECEF.

**See Also** `ecef2geodetic`, `ecef2lv`, `lv2ecef`, `geodetic2geocentricLat`

**Purpose**

Convert geocentric to geodetic latitude

**Syntax**

```
phiI = geocentric2geodeticlat(ecc, phi_g)
```

`phiI = geocentric2geodeticlat(ecc, phi_g)` converts an array of geocentric latitude in radians, `phi_g`, to geodetic latitude in radians, `phiI`, on a reference ellipsoid with first eccentricity `ecc`.

For conversion to/from other types of auxiliary latitude and, optionally, to work in degrees, use Mapping Toolbox function `convertlat`. For conversion from 3-D geocentric coordinates, see `ecef2geodetic`.

**See Also**

`convertlat`, `ecef2geodetic`, `geodetic2geocentricLat`

# geodetic2geocentricLat

---

**Purpose** Convert geodetic to geocentric latitude

**Syntax** `phi_g = geodetic2geocentriclat(ecc, phi)`  
`phi_g = geodetic2geocentriclat(ecc, phi)` converts an array of geodetic latitude in radians, `phi`, to geocentric latitude in radians, `phi_g`, on a reference ellipsoid with first eccentricity `ecc`.  
For conversion to/from other types of auxiliary latitude and, optionally, to work in degrees, use Mapping Toolbox function `convertlat`. For conversion to 3-D geocentric coordinates, see `geodetic2ecef`.

**See Also** `convertlat`, `geocentric2geodeticLat`, `geodetic2ecef`

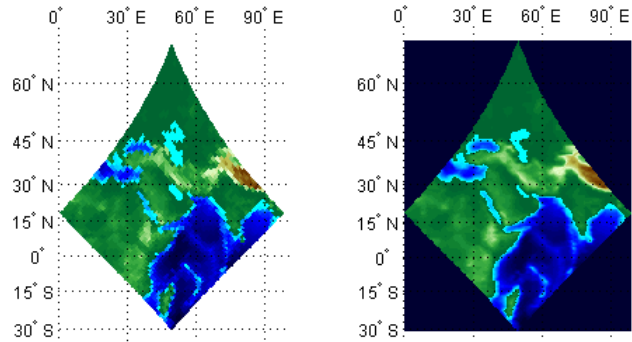
<b>Purpose</b>	Convert geolocated data array to regular data grid
<b>Syntax</b>	<pre>[Z, refvec] = geoloc2grid(lat, lon, A, cellsize)</pre> <p><code>[Z, refvec] = geoloc2grid(lat, lon, A, cellsize)</code> converts the geolocated data array <code>A</code>, given geolocation points in <code>lat</code> and <code>lon</code>, to produce a regular data grid, <code>Z</code>, and the corresponding three-element referencing vector <code>refvec</code>. <code>cellsize</code> is a scalar that specifies the width and height of data cells in the regular data grid, using the same angular units as <code>lat</code> and <code>lon</code>. Data cells in <code>Z</code> falling outside the area covered by <code>A</code> are set to NaN.</p>
<b>Remarks</b>	<code>geoloc2grid</code> provides an easy-to-use alternative to gridding geolocated data arrays with <code>imbedm</code> . There is no need to preallocate the output map; there are no data gaps in the output (even if <code>cellsize</code> is chosen to be very small), and the output map is smoother.
<b>Example</b>	<pre>% Load the geolocated data array 'map1' % and grid it to 1/2-degree cells. load mapmtx cellsize = 0.5; [Z, refvec] = geoloc2grid(lt1, lg1, map1, cellsize);  % Create a figure f = figure; [cmap, clim] = demcmap(map1); set(f, 'Colormap', cmap, 'Color', 'w')  % Define map limits latlim = [-35 70]; lonlim = [0 100];  % Display 'map1' as a geolocated data array in subplot 1 subplot(1,2,1) ax = axesm('mercator', 'MapLatLimit', latlim, 'MapLonLimit', lonlim, ... 'Grid', 'on', 'MeridianLabel', 'on', 'ParallelLabel', 'on');</pre>

# geoloc2grid

---

```
set(ax,'Visible','off')
geoshow(lt1, lg1, map1, 'DisplayType', 'texturemap');

% Display 'Z' as a regular data grid in subplot 2
subplot(1,2,2)
ax =
axesm('mercator','MapLatLimit',latlim,'MapLonLimit',lonlim,...
      'Grid','on','MeridianLabel','on','ParallelLabel','on');
set(ax,'Visible','off')
geoshow(Z, refvec, 'DisplayType', 'texturemap');
```



**Purpose**

Display map latitude and longitude data

**Syntax**

```

geoshow(lat,lon)
geoshow(lat,lon, ..., 'DisplayType', displaytype, ...)
geoshow(lat,lon,Z, ..., 'DisplayType', displaytype, ...)
geoshow(Z,R, ..., 'DisplayType', displaytype,...),
geoshow(lat,lon,I),geoshow(lat,lon,BW),geoshow(lat,lon,X,
    cmap), geoshow(lat,lon,RGB),
geoshow(... 'DisplayType', ...)
geoshow(I,R),geoshow(BW,R),geoshow(RGB,R),geoshow(A,CMAP,R),
geoshow(... `DisplayType', ...)
geoshow(s)
geoshow(s, ..., `SymbolSpec', symspec)
geoshow(filename)
geoshow(ax, ...)
geoshow(..., 'Parent', ax, ...)
h = geoshow(...)
geoshow(..., param1, val1, param2, val2, ...)

geoshow(lat,lon) or geoshow(lat,lon, ..., 'DisplayType',
displaytype, ...) project and display the latitude and longitude
vectors, lat and lon, using the projection stored in the axes. If there is
no projection, the latitudes and longitudes are projected using a default
Plate Carree projection. lat and lon must be of equal length, and may
contain embedded NaNs, delimiting individual lines or polygon parts.
DisplayType can be 'point', 'line', or 'polygon', and defaults to
'line'.

geoshow(lat,lon,Z, ..., 'DisplayType', displaytype, ...),
projects and displays a geolocated data grid. lat and lon are M-by-N
latitude-longitude arrays and Z is an M-by-N array of class double.
lat, lon, and Z may contain NaN values. DisplayType must be set to
'surface', 'mesh', 'texturemap', or 'contour'.

geoshow(Z,R, ..., 'DisplayType', displaytype,...), projects
and displays a regular data grid. Z is 2-D array of class double and R
is a referencing vector or a referencing matrix in latitude-longitude.
DisplayType must be set to 'surface', 'mesh', 'texturemap', or

```

'contour'. If `DisplayType` is 'texturemap', `geoshow` constructs a surface with `ZData` values set to 0.

`geoshow(lat,lon,I)`, `geoshow(lat,lon,BW)`, `geoshow(lat,lon,X,cmap)`, `geoshow(lat,lon,RGB)`, or `geoshow(... 'DisplayType', ...)` projects and display a geolocated image as a texturemap on a zero-elevation surface. `lat` and `lon` are latitude-longitude geolocation arrays and `I` is a grayscale image, `BW` is a logical image, `X` is an indexed image with colormap `cmap`, or `RGB` is a truecolor image. `lat`, `lon`, and the image array must match in size. If specified, `DisplayType` must be set to 'image'. Examples of geolocated images include a color composite from a satellite swath or an image originally referenced to a different coordinate system.

`geoshow(I,R)`, `geoshow(BW,R)`, `geoshow(RGB,R)`, `geoshow(A,CMAP,R)`, or `geoshow(... `DisplayType', ...)` project and display an image georeferenced to latitude-longitude through the referencing matrix `R`. The image is shown as a texturemap on a zero-elevation surface. If specified, `DisplayType` must be set to 'image'.

`geoshow(s)` or `geoshow(s, ..., `SymbolSpec', symspec)` display the vector geographic features stored in the geographic data structure `s` as points, multipoints, lines, or polygons according to the `Geometry` field of `s`. If `s` includes `Lat` and `Lon` fields, then the coordinate values are projected to map coordinates. If `s` includes `X` and `Y` fields they are plotted as (preprojected) map coordinates and a warning is issued. `symspec` is a structure returned by `makesymbolspec` that specifies the symbolization rules to be used for displaying vector data.

`geoshow(filename)` projects and displays data from `filename` according to the type of file format. The `DisplayType` parameter is automatically set, according to the following table:

Format	DisplayType
Shape file	'point', 'line', or 'polygon'
GeoTIFF	'image'



Format	DisplayType
TIFF/JPEG/PNG with a world file	'image'
ARC ASCII GRID	'surface' (can be overridden)
SDTS raster	'surface' (can be overridden)

`geoshow(ax, ...)` and `geoshow(..., 'Parent', ax, ...)` set the parent axes to `ax`.

`h = geoshow(...)` returns a handle to a MATLAB graphics object or, in the case of polygons, a modified patch object. If a geostruct or shapefile name is input, `geoshow` returns the handle to an hggroup object with one child per feature in the geostruct or shapefile, excluding any features that are completely trimmed away. In the case of a polygon geostruct or shapefile, each child is a modified patch object; otherwise it is a line object.

`geoshow(..., param1, val1, param2, val2, ...)` specifies parameter/value pairs that modify the type of display or set MATLAB graphics properties. Refer to the MATLAB Graphics documentation for line, patch, image, surface, mesh, and contour properties for full descriptions of these object properties and their values.

## Parameters

Parameter names can be abbreviated and are case insensitive. Parameters include

- **DisplayType:** The `DisplayType` parameter specifies the type of graphic display for the data. The value must be consistent with the type of data being displayed, as shown in the following table:

Data Type	Value(s)
Vector	'point', 'multipoint', 'line', or 'polygon'

Data Type	Value(s)
Image	'image'
Grid	'surface', 'mesh', 'texturemap', or 'contour'

- **SymbolSpec:** The SymbolSpec parameter specifies the symbolization rules used for vector data through a structure returned by makesymbolspec. It is used only for vector data stored in geographic data structures.

In cases where both SymbolSpec and one or more graphics properties are specified, the graphics properties override any settings in the symbolspec structure.

To change the default symbolization rule for a property name/property value pair in the symbolspec, prefix the word 'Default' to the graphics property name (listed in the preceding table). See Example 2 below.

## Graphics Properties

In addition to specifying a parent axes, you can set any appropriate property for a point, line, and polygon DisplayType, as follows:

DisplayType	Properties
'line'	Any MATLAB line property
'point'	Any MATLAB line marker property
'polygon'	Any MATLAB patch property

See the MATLAB Graphics Reference documentation for line, patch, image, and surface properties for complete descriptions of these properties and their values.

## Remarks

geoshow is often used to display vector geodata previously read from shapefiles using shaperead. When calling shaperead to read files that contain coordinates in latitude and longitude, be sure to specify the shaperead argument pair 'UseGeoCoords', true; if you do not include

this argument (or specify 'UseGeoCoords', false), shaperead will create a geostruct with coordinate fields labelled X and Y instead of Lon and Lat, causing geoshow to assume that the geostruct contains projected coordinates. In such cases, geoshow warns and calls mapshow to display the geostruct data without projecting it.

When projecting data onto a map axes, geoshow uses the projection stored with the map axes. When displaying on a regular axes, it constructs a default Plate Carrée projection with a scale factor of  $180/\pi$ , enabling direct readout of coordinates in degrees.

---

**Note** When you display vector data in a map axes using geoshow, you should not subsequently change the map projection using setm. You can, however, change the projection with setm for raster data. For more information, see “Changing Map Projections when Using geoshow” on page 4-25.

---

geoshow adds graphics to the current map axes (it does not clear it first), enabling you to create multiple raster and vector map layers. If you do not want geoshow to draw on top of an existing map, create a new figure or subplot before calling it.

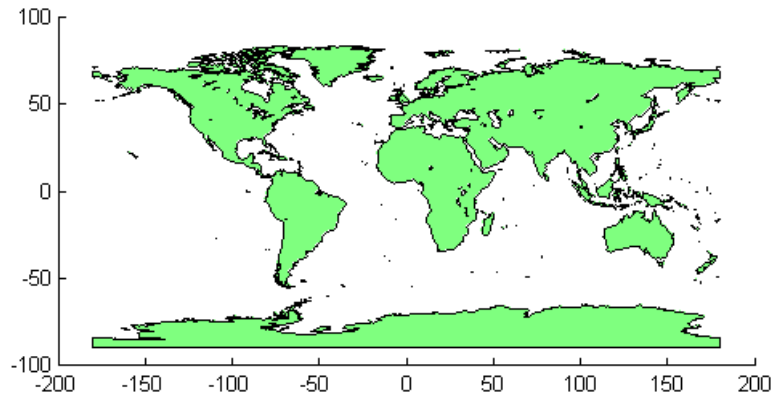
geoshow can generally be substituted for displaym. However, there are limitations where display of specific objects is concerned. See the remarks under updategeostruct for further information.

## Examples

### Example 1

Display world land areas using a default Plate Carree projection:

```
figure
geoshow('landareas.shp', 'FaceColor', [0.5 1.0 0.5]);
```



## Example 2

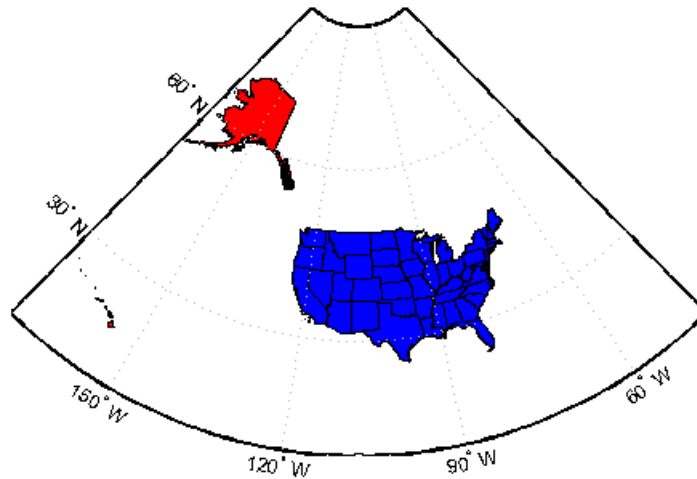
Override the symbolspec default rule:

```
% Create a worldmap of North America
figure
worldmap('na');

% Read the USA high resolution data
states = shaperead('usastatehi', 'UseGeoCoords', true);

% Create a symbolspec to make Alaska and Hawaii polygons red.
symspec = makesymbolspec('Polygon', ...
    {'Name', 'Alaska', 'FaceColor', 'red'}, ...
    {'Name', 'Hawaii', 'FaceColor', 'red'});

% Display all the other states in blue.
geoshow(states, 'SymbolSpec', symspec, ...
    'DefaultFaceColor', 'blue', ...
    'DefaultEdgeColor', 'black');
```



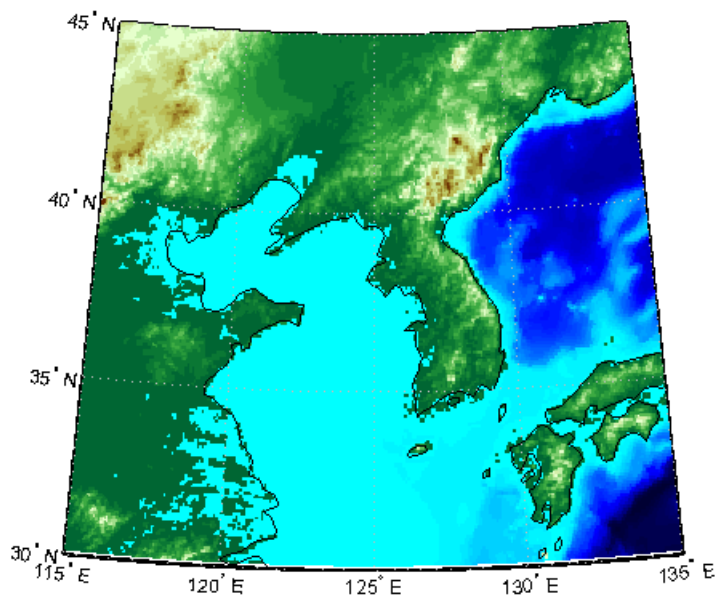
### Example 3

Create a worldmap of Korea and display the korea data grid as a texture map:

```
load korea
figure;
worldmap(map, refvec)

% Display the Korean data grid as a texture map.
geoshow(gca,map,refvec,'DisplayType','texturemap');
colormap(demcmap(map))

% Display the land area boundary as black lines.
S = shaperead('landareas','UseGeoCoords',true);
geoshow([S.Lat], [S.Lon],'Color','black');
```



## Example 4

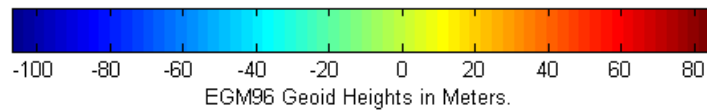
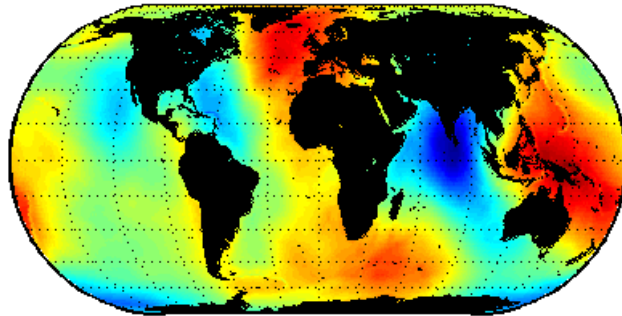
Display the EGM96 geoid heights, masking out land areas:

```
load geoid
% Create a figure with an Eckert projection.
figure
axesm eckert4;
framem; gridm;
axis off

% Display the geoid as a texture map.
geoshow(geoid, geoidrefvec, 'DisplayType', 'texturemap');

% Create a colorbar and title.
hcb = colorbar('horiz');
set(get(hcb, 'Xlabel'), 'String', 'EGM96 Geoid Heights in Meters.')
```

```
% Mask out all the land.  
geoshow('landareas.shp', 'FaceColor', 'black');
```

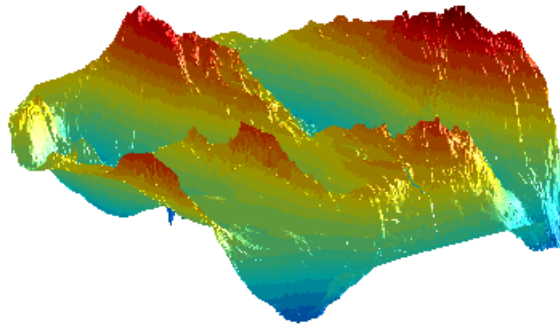


### Example 5

Display the EGM96 geoid heights as a 3-D surface using the Eckert IV projection:

```
load geoid  
  
% Create the figure with an Eckert projection.  
figure  
axesm eckert4;  
axis off  
  
% Display the geoid as a surface.  
h=geoshow(geoid, geoidrefvec, 'DisplayType','surface');  
  
% Add light and material.  
light; material(0.6*[ 1 1 1]);  
  
% View as a 3-D surface.
```

```
view(3)
axis normal
tightmap
```



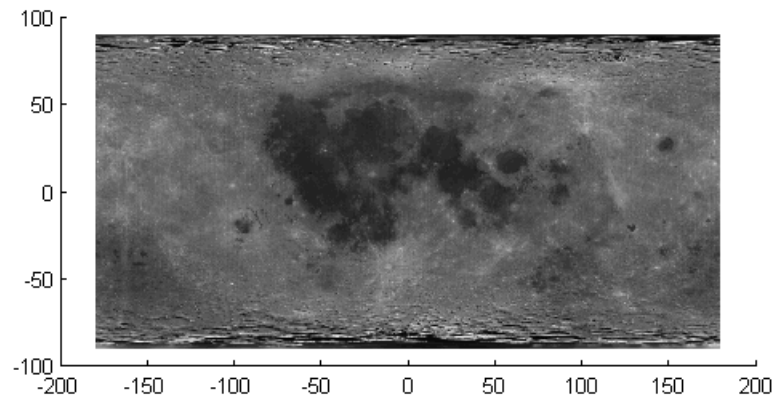
## Example 6

Display the moon albedo image projected using Plate Carree and in an orthographic projection.

```
load moonalb

% Projection not specified -- uses Plate Carree
figure
geoshow(moonalb,moonal Brefvec)
```





```
% Orthographic projection
figure
axesm ortho
geoshow(moonalb, moonalbrefvec, 'DisplayType', 'texturemap')
colormap(gray(256))
axis off
```



## Example 7

Read and display the San Francisco South 24K DEM data:

```
filenames = gunzip('sanfranciscos.dem.gz', tempdir);
demFilename = filenames{1};

% Read every point of the 1:24,000 DEM file.
[lat, lon,Z] = usgs24kdem(demFilename,2);

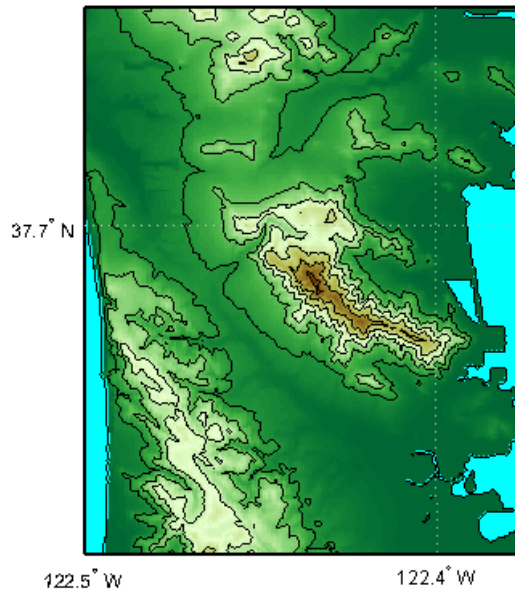
% Delete the temporary gunzipped file.
delete(demFilename);

% Move all points at sea level to -1 to color them blue.
Z(Z==0) = -1;

% Compute the latitude and longitude limits for the DEM.
latlim = [min(lat(:)) max(lat(:))];
```

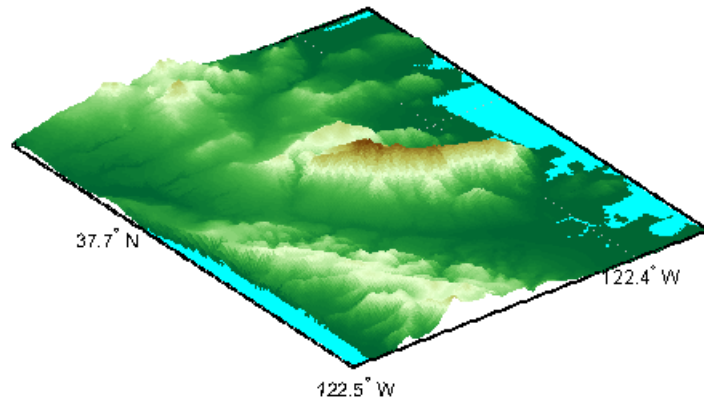
```
lonlim = [min(lon(:)) max(lon(:))];

% Display the DEM values as a texture map.
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, 'DisplayType','texturemap')
demcmap(Z)
daspectm('m',1)
% Overlay black contour lines onto the texturemap.
geoshow(lat, lon, Z, 'DisplayType', 'contour', ...
        'LineColor', 'black');
```



```
% View the DEM values in 3-D.
figure
usamap(latlim, lonlim)
geoshow(lat, lon, Z, 'DisplayType', 'surface')
demcmap(Z)
```

```
daspectm('m',1)  
view(3)
```



## See Also

`axesm`, `makesymbolspec`, `mapshow`, `mapview`, `updategeostruct`

**Purpose** Convert GeoTIFF information to map projection structure

**Syntax** `mstruct = geotiff2mstruct(proj)`  
`mstruct = geotiff2mstruct(proj)` converts the GeoTIFF projection structure, `proj`, to the map projection structure, `mstruct`. The unit of length of the `mstruct` projection is meter.

**Example**

```
% Compare inverse transform of points using projinv and minvtran.  
% Obtain the projection structure of 'boston.tif'.  
proj = geotiffinfo('boston.tif');  
  
% Convert the corner map coordinates to latitude and longitude.  
x = proj.CornerCoords.X;  
y = proj.CornerCoords.Y;  
[latProj, lonProj] = projinv(proj, x, y);  
  
% Obtain the mstruct from the GeoTIFF projection.  
mstruct = geotiff2mstruct(proj);  
  
% Convert the units of x and y to meter to match projection units.  
x = unitsratio('meter','sf') * x;  
y = unitsratio('meter','sf') * y;  
  
% Convert the corner map coordinates to latitude and longitude.  
[latMstruct, lonMstruct] = minvtran(mstruct, x, y);  
  
% Verify the values are within a tolerance of each other.  
abs(latProj - latMstruct) <= 1e-7  
abs(lonProj - lonMstruct) <= 1e-7  
  
ans =  
     1     1     1     1  
  
ans =  
     1     1     1     1
```

# geotiff2mstruct

---

## See Also

`axesm`, `defaultm`, `geotiffinfo`, `projfwd`, `projinv`, `projlist`

**Purpose**

Information about GeoTIFF file

**Syntax**

```
info = geotiffinfo(filename)
info = geotiffinfo(url)
```

`info = geotiffinfo(filename)` returns a structure whose fields contain image properties and cartographic information about a GeoTIFF file.

`filename` is a string that specifies the name of the GeoTIFF file. `filename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path. If the named file includes the extension `.TIF` or `.TIFF` (either upper- or lowercase), the extension can be omitted from `filename`.

If `filename` is a file containing more than one GeoTIFF image, `info` is a structure array with one element for each image in the file. For example, `info(3)` would contain information about the third image in the file. If more than one image exists in the file, it is assumed that each image will have the same cartographic information and the same image width and height.

`info = geotiffinfo(url)` reads the GeoTIFF image from an Internet URL. The `url` must include the protocol type (e.g., “`http://`”).

**Field Description**

The `info` structure contains the following fields:

<code>Filename</code>	String containing the name of the file
<code>FileModDate</code>	String containing the modification date of the file
<code>FileSize</code>	Integer indicating the size of the file in bytes
<code>Format</code>	String containing the file format, which should always be 'tiff'
<code>FormatVersion</code>	String or number specifying the file format version

Height	Integer indicating the height of the image in pixels
Width	Integer indicating the width of the image in pixels
BitDepth	Integer indicating the number of bits per pixel
<i>ColorType</i>	String indicating the type of image: 'truecolor' for a true-color (RGB) image, 'grayscale' for a grayscale image, or 'indexed' for an indexed image
<i>ModelType</i>	String indicating the type of coordinate system used to georeference the image: 'ModelTypeProjected', 'ModelTypeGeographic', or ''
PCS	String describing the projected coordinate system
Projection	String describing the EPSG identifier for the underlying projection method
<i>MapSys</i>	String indicating the map system, if applicable: 'STATE_PLANE_27', 'STATE_PLANE_83', 'UTM_NORTH', 'UTM_SOUTH', or ''
Zone	Double indicating the UTM or State Plane Zone number, empty ([]) if not applicable or unknown
CTProjection	String containing the GeoTIFF identifier for the underlying projection method
ProjParm	An N-by-1 double containing projection parameter values. The identity of each element is specified by the corresponding element of ProjParmId. Lengths are in meters, angles in decimal degrees.



---

ProjParmId	<p>An N-by-1 cell array listing the projection parameter identifier for each corresponding numerical element of ProjParm:</p> <ul style="list-style-type: none"><li>• 'ProjNatOriginLatGeoKey'</li><li>• 'ProjNatOriginLongGeoKey'</li><li>• 'ProjFalseEastingGeoKey'</li><li>• 'ProjFalseNorthingGeoKey'</li><li>• 'ProjFalseOriginLatGeoKey'</li><li>• 'ProjFalseOriginLongGeoKey'</li><li>• 'ProjCenterLatGeoKey'</li><li>• 'ProjCenterLongGeoKey'</li><li>• 'ProjAzimuthAngleGeoKey'</li><li>• 'ProjRectifiedGridAngleGeoKey'</li><li>• 'ProjScaleAtNatOriginGeoKey'</li><li>• 'ProjStdParallel1GeoKey'</li><li>• 'ProjStdParallel2GeoKey'</li></ul>
GCS	String indicating the geographic coordinate system
Datum	String indicating the projection datum type, such as 'North American Datum 1927' or 'North American Datum 1983'
Ellipsoid	String indicating the ellipsoid name as defined by the ellipsoid.csv EPSG file
SemiMajor	Double indicating the length of the semimajor axis of the ellipsoid, in meters
SemiMinor	Double indicating the length of the semiminor axis of the ellipsoid, in meters

PM	String indicating the prime meridian location, for example, 'Greenwich' or 'Paris'
PmLongToGreenwich	Double indicating the decimal degrees of longitude between this prime meridian and Greenwich. Prime meridians to the west of Greenwich are negative.
UOMLength	String indicating the units of length used in the projected coordinate system
UOMLengthInMeters	Double defining the UOMLength unit in meters
UOMAngle	String indicating the angular units used for geographic coordinates
UOMAngleInDegrees	Double defining the UOMAngle unit in degrees
TiePoints	Structure containing the image tiepoints. The structure contains these fields: <ul style="list-style-type: none"><li>• ImagePoints — A structure containing row and column coordinates of each image tiepoint. The ImagePoints structure contains these fields:<ul style="list-style-type: none"><li>▪ Row — A double array of size 1-by-N.</li><li>▪ Col — A double array of size 1-by-N.</li></ul></li><li>• WorldPoints — A structure containing the x and y world coordinates of the tiepoints. The WorldPoints structure contains these fields:<ul style="list-style-type: none"><li>▪ X — A double array of size 1-by-N</li><li>▪ Y — A double array of size 1-by-N</li></ul></li></ul>
PixelScale	3-by-1 double array that specifies the X, Y, Z pixel scale values

---

RefMatrix	3-by-2 double referencing matrix that must be unambiguously defined by the GeoTIFF file; otherwise it is returned empty ([ ]).
BoundingBox	2-by-2 double array that specifies the minimum (row 1) and maximum (row 2) values for each dimension of the image data in the GeoTIFF file
CornerCoords	<p>A structure with six fields that contains coordinates of the outer corners of the GeoTIFF image. Each field is a 1-by-4 double array, or empty ([ ]) if unknown. The arrays contain the coordinates of the outer corners of the corner pixels, starting from the (1,1) corner and proceeding clockwise:</p> <ul style="list-style-type: none"><li>• X — Horizontal coordinates in the projected coordinate system. X equals Lon (below) if <i>ModelType</i> is 'ModelTypeGeographic'.</li><li>• Y — Vertical coordinates in the projected coordinate system. Y equals Lat (below) if <i>ModelType</i> is 'ModelTypeGeographic'.</li><li>• Row — Row coordinates of the corner</li><li>• Col — Column coordinates of the corner</li><li>• Lat — Latitudes of the corner</li><li>• Lon — Longitudes of the corner</li></ul>

**GeoTIFFCodes** Structure containing raw numeric values for those GeoTIFF fields that are encoded numerically in the file. These raw values, converted to a string elsewhere in the INFO structure, are provided here for reference. The GeoTIFFCodes fields are:

- Model
- PCS
- GCS
- UOMLength
- UOMAngle
- Datum
- PM
- Ellipsoid
- ProjCode
- Projection
- CTProjection
- ProjParmId
- MapSys

Each is scalar, except for ProjParmId, which is a column vector.

**ImageDescription** String describing the image; if no description is included in the file, the field is omitted.

## Example

```
info = geotiffinfo('boston.tif')  
  
info =  
    Filename: [1x78 char]
```

```
FileModDate: '31-May-2007 03:25:30'  
  FileSize: 38729900  
    Format: 'tif'  
FormatVersion: []  
  Height: 2881  
  Width: 4481  
  BitDepth: 24  
  ColorType: 'truecolor'  
  ModelType: 'ModelTypeProjected'  
    PCS: 'NAD83 / Massachusetts Mainland'  
  Projection: 'SPCS83 Massachusetts Mainland zone (m)'  
    MapSys: 'STATE_PLANE_83'  
    Zone: 2001  
  CTProjection: 'CT_LambertConfConic_2SP'  
    ProjParm: [7x1 double]  
  ProjParmId: {7x1 cell}  
    GCS: 'NAD83'  
    Datum: 'North American Datum 1983'  
    Ellipsoid: 'GRS 1980'  
    SemiMajor: 6378137  
    SemiMinor: 6.3568e+006  
    PM: 'Greenwich'  
  PMLongToGreenwich: 0  
    UOMLength: 'US survey foot'  
  UOMLengthInMeters: 0.3048  
    UOMAngle: 'degree'  
  UOMAngleInDegrees: 1.0000  
    TiePoints: [1x1 struct]  
    PixelScale: [3x1 double]  
    RefMatrix: [3x2 double]  
    BoundingBox: [2x2 double]  
    CornerCoords: [1x1 struct]  
    GeoTIFFCodes: [1x1 struct]  
  ImageDescription: '"GeoEye"'
```

**See Also**

imfinfo, geotiffread, makerefmat, projfwd, projinv, projlist

## Purpose

Read georeferenced image from GeoTIFF file

## Syntax

```
A = geotiffread(filename)
[X, cmap] = geotiffread(filename)
[X, cmap, R, bbox] = geotiffread(filename)
[A, R, bbox] = geotiffread(filename)
[...] = geotiffread(filename, idx)
[...] = geotiffread(url, ...)
```

`A = geotiffread(filename)` reads the GeoTIFF image in `filename` into `A`. If the file contains a grayscale image, `A` is a two-dimensional array. If the file contains a true-color (RGB) image, `A` is a three-dimensional (M-by-N-by-3) array.

`filename` is a string that specifies the name of the GeoTIFF file. `filename` can include the directory name; otherwise, the file must be in the current directory or in a directory on the MATLAB path. If the named file includes the extension `.TIF` or `.TIFF` (either upper- or lowercase), the extension can be omitted from `filename`.

`[X, cmap] = geotiffread(filename)` reads the indexed image in `filename` into `X` and its associated colormap into `cmap`. Colormap values in the image file are automatically rescaled into the range `[0,1]`.

`[X, cmap, R, bbox] = geotiffread(filename)` reads the indexed image into `X`, the associated colormap into `cmap`, the referencing matrix into `R`, and the bounding box into `bbox`. The referencing matrix must be unambiguously defined by the GeoTIFF file; otherwise, it and the bounding box are returned empty (`[]`).

`[A, R, bbox] = geotiffread(filename)` reads the grayscale or RGB image into `A`, the referencing matrix into `R`, and the bounding box into `bbox`.

`[...] = geotiffread(filename, idx)` reads in one image from a multiimage GeoTIFF file. `idx` is an integer value that specifies the order that the image appears in the file. For example, if `idx` is 3, `geotiffread` reads the third image in the file. If you omit this argument, `geotiffread` reads the first image in the file.

[...] = geotiffread(url, ...) reads the GeoTIFF image from an Internet URL. The URL must include the protocol type (e.g., “http://”).

---

**Note** geotiffread imports pixel data using the TIFF-reading capabilities of the MATLAB function imread. Consequently, it shares the limitations of imread. Consult the imread documentation for details on the types of TIFF images that imread can import.

---

## Example

Read and display the Boston GeoTIFF image:

```
[boston, R, bbox] = geotiffread('boston.tif');  
figure  
mapshow(boston, R);  
axis image off
```



boston.tif copyright © GeoEye, all rights reserved.

# geotiffread

---

## See Also

`geotiffinfo`, `imread`, `mapview`, `mapshow`, `geoshow`



**Purpose**

Map object properties

**Syntax**

```
mat = getm(h)
mat = getm(h,MapPropertyName)
getm('MapProjection')
getm('axes')
getm('units')
```

`mat = getm(h)` returns the map structure of the map axes specified by its handle. If the handle of a child of the map axes is specified, only its properties are returned.

`mat = getm(h,MapPropertyName)` returns the specified property value.

`getm('MapProjection')` lists all available projections.

`getm('axes')` lists the map axes properties by property name.

`getm('units')` lists the available units.

**Examples**

Create a default map axes and query a property value:

```
axesm('mercator','AngleUnits','degrees')
getm(gca,'MapParallels')
```

```
ans =
     0
```

**See Also**

`axesm`, `setm`

# getseeds

---

## Purpose

Interactively assign seeds for data grid encoding

## Syntax

```
[row,col,val] = getseeds(Z,refvec,nseeds)
[row,col,val] = getseeds(Z,refvec,nseeds,seedval)
seedmat = getseeds(...)
```

`[row,col,val] = getseeds(Z,refvec,nseeds)` prompts the user for a number, `nseeds`, of mouse-input locations on the current map axes. After the locations are selected, the user is prompted for a value to associate with each location. The outputs are the row and column, `row` and `col`, of the input regular data grid, `Z`, with its associated three-element referencing vector, `refvec`, corresponding to the input locations. The third output, `val`, returns the selected value for each location.

`[row,col,val] = getseeds(Z,refvec,nseeds,seedval)` predefines the values of the locations. If `seedval` is a scalar, the same value is assigned to all points. If it is a vector with a length of `nseeds`, each entry corresponds to a particular location.

`seedmat = getseeds(...)` packs the outputs into a single, three-column matrix, `seedmat`, that is a suitable input for the `encodem` function. The form of this matrix is `[lat lon val]`.

## Description

The `getseeds` function allows you to interactively create the seed matrix values used by the `encodem` function to fill in regions of data grids.

## Examples

Demonstrate this for yourself by typing the following and interactively selecting points:

```
load topo
axesm('gortho','grid','on')
seedmat = getseeds(topo,topolegend,3)
```

When you have selected three points, you are prompted for their values. The regular data grid need not be displayed to execute `getseeds` on it.

## See Also

`encodem`

**Purpose** Derive worldfile name from image filename

**Syntax** `worldfilename = getworldfilename(imagefilename)`  
`worldfilename = getworldfilename(imagefilename)` returns the name of the corresponding worldfile derived from the name of an image file.

The worldfile and the image file have the same base name. If `imagefilename` follows the “.3” convention, then you create the worldfile extension by removing the middle letter and appending the letter 'w'.

If `imagefilename` has an extension that does not follow the “.3” convention, then a 'w' is appended to the full image name to construct the worldfile name.

If `imagefilename` has no extension, then '.wld' is appended to construct a worldfile name.

**Examples** Given the following image filenames, `worldfilename` would return these worldfile names:

Image File Name	Worldfile Name
myimage.tif	myimage.tfw
myimage.jpeg	myimage.jpegw
myimage	myimage.wld

**See Also** `mapshow`, `mapview`, `worldfileread`, `worldfilewrite`

## Purpose

Read Global Land One-km Base Elevation (GLOBE) data

## Syntax

```
[Z,refvec] = globedem(filename,scalefactor)
[Z,refvec] = globedem(filename,scalefactor,latlim,lonlim)
[Z,refvec] = globedem(dirname,scalefactor,latlim,lonlim)
```

[Z,refvec] = globedem(filename,scalefactor) reads the GLOBE DEM files and returns the result as a regular data grid. The filename is given as a string that does not include an extension. GLOBEDEM first reads the ESRI header file found in the subdirectory '/esri/hdr/' and then the binary data file filename. If the files are not found on the MATLAB path, they can be selected interactively. scalefactor is an integer that when equal to 1 gives the data at its full resolution. When scalefactor is an integer n larger than 1, every nth point is returned. The map data is returned as an array of elevations and associated three-element referencing vector. Elevations are given in meters above mean sea level, using WGS 84 as a horizontal datum.

[Z,refvec] = globedem(filename,scalefactor,latlim,lonlim) allows a subset of the map data to be read. The limits of the desired data are specified as vectors of latitude and longitude in degrees. The elements of latlim and lonlim must be in ascending order.

[Z,refvec] = globedem(dirname,scalefactor,latlim,lonlim) reads and concatenates data from multiple files within a GLOBE directory tree. The dirname input is a string with the name of the directory that contains both the uncompressed data files and the ESRI header files.

## Background

GLOBE, the Global Land One-km Base Elevation data, was compiled by the National Geophysical Data Center from more than 10 different sources of gridded elevation data. GLOBE can be considered a higher resolution successor to TerrainBase. The data set consists of 16 tiles, each covering 50 by 90 degrees. Tiles require as much as 60 MB of storage. Uncompressed tiles take between 100 and 130 MB.

## Remarks

Mapping Toolbox reads data from GLOBE Version 1.0. The data is for elevations only. Elevations are given in meters above mean sea level

using WGS 84 as a horizontal datum. Areas with no data, such as the oceans, are coded with NaNs.

The data set and documentation are available over the Internet.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

Determine the file that contains the area around Cape Cod.

```
latlim = [41 42.5]; lonlim = [-73 -69.9];
globedems(latlim,lonlim)

ans =
    'f10g'
```

Extract every 20th point from the tile covering the northeastern United States and eastern Canada. Provide an empty file name, and select the file interactively.

```
[Z,refvec] = globedem([],20);
size(Z)

ans =
    300    540
```

Extract a subset of the data for Massachusetts at the full resolution.

```
latlim = [41 42.5]; lonlim = [-73 -69.9];
[Z,refvec] = globedem('f10g',1,latlim,lonlim);
size(Z)

ans =
    181    373
```

Replace the NaNs in the ocean with -1 to color them blue.

```
Z(isnan(Z)) = -1;
```

Extract some data for southern Louisiana in an area that straddles two tiles. Provide the name of the directory containing the data files, and let globedem determine which files are required, read from the files, and concatenate the data into a single regular data grid.

```
latlim =[28.61 31.31]; lonlim = [-91.24 -88.62];
globedems(latlim,lonlim)

ans =
    'e10g'
    'f10g'

[Z,refvec] =
globedem('d:\externalData\globe\elev',1,latlim,lonlim);
size(Z)

ans =
    325.00    315.00
```

## References

See Web site for the National Oceanic and Atmospheric Administration, National Geophysical Data Center

## See Also

demdataui, dted, gtopo30, satbath, tbase, usgsdem

<b>Purpose</b>	GLOBE data filenames for latitude-longitude quadrangle
<b>Syntax</b>	<pre>fname = globedems(latlim,lonlim)</pre> <p>fname = globedems(latlim,lonlim) returns a cell array of the filenames covering the geographic region for GLOBE DEM digital elevation maps. The region is specified by scalar latitude and longitude points, or two-element vectors of latitude and longitude limits in units of degrees.</p>
<b>Background</b>	GLOBE, the Global Land One-km Base Elevation data, was compiled by the National Geophysical Data Center from more than 10 different sources of gridded elevation data. The data set consists of 16 tiles, each covering 50 by 90 degrees. Determining which files are needed to cover a particular region generally requires consulting an index map. This function takes the place of such a reference by returning the filenames for a given geographic region.
<b>Remarks</b>	Mapping Toolbox reads data from GLOBE Version 1.0. GLOBE DEM first reads the corresponding ESRI header file found in the subdirectory '/esri/hdr/' and then the binary data file (with no extension).
<b>Examples</b>	<p>Which files are needed for southern Louisiana?</p> <pre>latlim =[28.61 31.31]; lonlim = [-91.24 -88.62]; globedems(latlim,lonlim)</pre> <pre>ans =     'e10g'     'f10g'</pre>
<b>References</b>	See Web site for the National Oceanic and Atmospheric Administration, National Geophysical Data Center
<b>See Also</b>	globedem

# gradientm

---

## Purpose

Calculate gradient, slope and aspect of data grid

`[ASPECT, SLOPE, gradN, gradE] = gradientm(Z, refvec)` computes the slope, aspect and north and east components of the gradient for a regular data grid `Z` with three-element referencing vector `refvec`. If the grid contains elevations in meters, the resulting aspect and slope are in units of degrees clockwise from north and up from the horizontal. The north and east gradient components are the change in the map variable per meter of distance in the north and east directions. The computation uses finite differences for the map variable on the default earth ellipsoid.

`[...] = gradientm(lat, lon, Z)` does the computation for a geolocated data grid. `lat` and `lon`, the latitudes and longitudes of the geolocation points, are in degrees.

`[...] = gradientm(...,ellipsoid)` uses the specified ellipsoid vector, `ellipsoid`, a 1-by-2 vector of the form `[semimajor-axis, eccentricity]`. If the map contains elevations in the same units as `ellipsoid(1)`, the slope and aspect are in units of degrees. This calling form is most useful for computations on bodies other than the earth.

`[...] = gradientm(lat, lon, Z, ellipsoid, units)` specifies the angle units of the latitude and longitude inputs. If omitted, 'degrees' are assumed. For elevation maps in the same units as `ellipsoid(1)`, the resulting slope and aspect are in the specified units. The components of the gradient are the change in the map variable per unit of `ellipsoid(1)`.

## Remarks

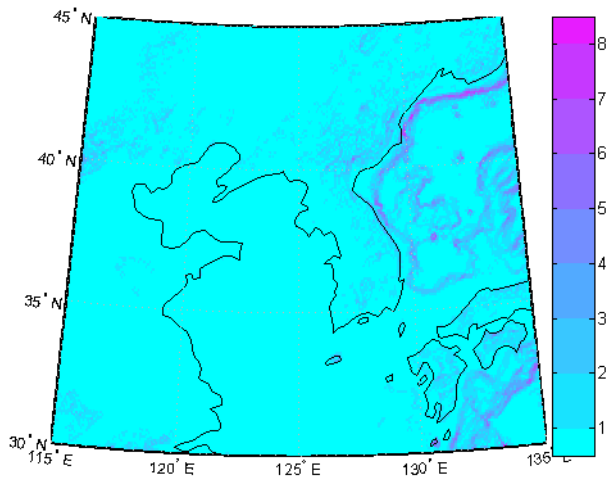
Coarse digital elevation models can considerably underestimate the local slope. For the preceding map, the elevation points are separated by about 10 kilometers. The terrain between two adjacent points is modeled as a linear variation, while actual terrain can vary much more abruptly over such a distance.

## Example

Compute and display the slope for the 30 arc-second (10 km) Korea elevation data. Slopes in the Sea of Japan are up to 8 degrees at this grid resolution.



```
load korea
[aspect, slope, gradN, gradE] = gradientm(map, refvec);
worldmap(slope, refvec)
geoshow(slope, refvec, 'DisplayType', 'texturemap')
cmap = cool(10);
demcmap('inc', slope, 1, [], cmap)
colorbar
latlim = getm(gca, 'maplatlimit');
lonlim = getm(gca, 'maplonlimit');
land = shaperead('landareas',...
    'UseGeoCoords', true, 'BoundingBox', [lonlim' latlim]);
geoshow(land, 'FaceColor', 'none')
set(gca, 'Visible', 'off')
```



## See Also

[viewshed](#)

# grepfields

---

## Purpose

Identify matching fields in fixed record length files

## Syntax

```
grepfields(filename,searchstring)
grepfields(filename,searchstring,casesens)
grepfields(filename,searchstring,casesens,startcol)
grepfields(filename,searchstring,casesens,startfield,fields)
grepfields(filename,searchstring,casesens,startfield,fields,
    machineformat)
indx = grepfields(...)
```

`grepfields(filename,searchstring)` displays lines in the file that begin with the search string. The file must have fixed-length records with line endings.

`grepfields(filename,searchstring,casesens)`, with `casesens` 'matchcase', specifies a case-sensitive search. If omitted or 'none', the search string matches regardless of the case.

`grepfields(filename,searchstring,casesens,startcol)` searches starting with the specified column. `startcol` is an integer between 1 and the bytes per record in the file. In this calling form, the file is regarded as a text file with line endings.

`grepfields(filename,searchstring,casesens,startfield,fields)` searches within the specified field. `startfield` is an integer between 1 and the number of fields per record. The format of the file is described by the `fields` structure. See `readfields` for recognized fields structure entries. In this calling form, the file can be binary and lack line endings. The search is within `startfield`, which must be a character field.

`grepfields(filename,searchstring,casesens,startfield,fields,machineformat)` opens the file with the specified machine format. `machineformat` must be recognized by `fopen`.

`indx = grepfields(...)` returns the record numbers of matched records instead of displaying them on screen.

## Example

Write a binary file and read it:

```
fid = fopen('testbin','wb');
```

```

for i = 1:3
    fwrite(fid,['character' num2str(i) ],'char');
    fwrite(fid,i,'int8');
    fwrite(fid,[i i],'int16');
    fwrite(fid,i,'integer*4');
    fwrite(fid,i,'real*8');
end
fclose(fid);

fs(1).length = 10;fs(1).type = 'char';fs(1).name = 'field 1';
fs(2).length = 1;fs(2).type = 'int8';fs(2).name = 'field 2';
fs(3).length = 2;fs(3).type = 'int16';fs(3).name = 'field 3';
fs(4).length = 1;fs(4).type = 'integer*4';fs(4).name = 'field 4';
fs(5).length = 1;fs(5).type = 'float64';fs(5).name = 'field 5';

```

Find the record matching the string 'character2'. The record contains binary data, which cannot be properly displayed.

```

grepfields('testbin','character2','none',1,fs)
character2? ? ?  ?@

indx = grepfields('testbin','character2','none',1,fs)
indx =
    2

```

Read the formatted file containing the following:

```

-----
character data 1  1  2  3 1e6 10D6

character data 2 11 22 33 2e6 20D6

character data 3111222333 3e6 30D6
-----

```

```

fs(1).length = 16;fs(1).type = 'char';fs(1).name = 'field 1';
fs(2).length = 3;fs(2).type = '%3d';fs(2).name = 'field 2';
fs(3).length = 1;fs(3).type = '%4g';fs(3).name = 'field 3';

```

# grepfields

---

```
fs(4).length = 1;fs(4).type = '%5D'; fs(4).name = 'field 4';  
fs(5).length = 1;fs(5).type = 'char';fs(5).name = '';
```

Find the records that match at the beginning of the line.

```
grepfields('testfile1','character')  
character data 1 1 2 3 1e6 10D6  
character data 2 11 22 33 2e6 20D6  
character data 3111222333 3e6 30D6
```

```
grepfields('testfile1','character data 2')  
character data 2 11 22 33 2e6 20D6
```

Find the records that match, starting the search in column 11.

```
grepfields('testfile1','data 2','none',11)  
character data 2 11 22 33 2e6 20D6
```

Search record number 1.

```
grepfields('testfile1','character data 2','none',1,fs)  
character data 2 11 22 33 2e6 20D6
```

## Limitations

Searches are limited to fields containing character data.

## Remarks

See `readfields` for a complete discussion of the format and contents of the `fields` argument.

## See Also

`readfields`, `fopen`

---

<b>Purpose</b>	Toggle and control display of map grid
<b>Syntax</b>	<pre>gridm gridm('on') gridm('off') gridm('reset') gridm(<i>linestyle</i>) gridm(<i>PropertyName,PropertyValue,...</i>)</pre> <p>gridm toggles the visibility of the map grid by setting the map axes property Grid to 'on' or 'off'. The default setting for map axes is 'off'.</p> <p>gridm('on') sets the map axes Grid property to 'on'.</p> <p>gridm('off') sets the map axes Grid property to 'off'.</p> <p>gridm('reset') resets the entire grid using the current properties. This is essentially a refresh option.</p> <p>gridm(<i>linestyle</i>) sets the map axes GridLineStyle property to any line style string recognized by the MATLAB line function.</p> <p>gridm(<i>PropertyName,PropertyValue,...</i>) sets the appropriate map axes properties to the desired values. These property names and values are described on the axesm reference page of this guide.</p>
<b>Remarks</b>	You can also create or alter map grid properties using the axesm or setm functions.
<b>See Also</b>	axesm, setm

# grid2image

---

**Purpose** Display regular data grid as image

**Syntax**  
`grid2image(Z,R)`  
`grid2image(Z,R,'PropertyName',PropertyValue,...)`  
`h = grid2image(...)`

`grid2image(Z,R)` displays a regular data grid as an image. `Z` can be a matrix of dimension `M-by-N` or `M-by-N-by-3`, and can contain `double`, `uint8`, or `uint16` data. `R` is a `1-by-3` referencing vector defined as `[cells/angle units north-latitude west-longitude]`, or a `3-by-2` referencing matrix, defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. The displayed map is a Plate Carrée projection, treating longitude as `X` and latitude as `Y`. This projection produces significant distortion near the poles.

`grid2image(Z,R,'PropertyName',PropertyValue,...)` uses the specified image properties to display the map. See the `image` function reference page for a list of properties that can be changed.

`h = grid2image(...)` returns the handle of the image object displayed.

**See Also** `image`, `mapshow`, `mapview`, `meshm`, `surfacem`, `surfm`

**Purpose**

Convert from Greenwich to equal area coordinates

**Syntax**

```
[x,y] = grn2eqa(lat,lon)
[x,y] = grn2eqa(lat,lon,origin)
[x,y] = grn2eqa(lat,lon,origin,ellipsoid)
[x,y] = grn2eqa(lat,lon,origin,units)
mat = grn2eqa(lat,lon,origin...)
```

`[x,y] = grn2eqa(lat,lon)` converts the Greenwich coordinates `lat` and `lon` to the equal-area coordinate points `x` and `y`.

`[x,y] = grn2eqa(lat,lon,origin)` specifies the location in the Greenwich system of the  $x$ - $y$  origin (0,0). The two-element vector `origin` must be of the form [`latitude`, `longitude`]. The default places the origin at the Greenwich coordinates (0°,0°).

`[x,y] = grn2eqa(lat,lon,origin,ellipsoid)` specifies the two-element ellipsoid vector describing the ellipsoidal model of the figure of the Earth. The ellipsoid is spherical by default.

`[x,y] = grn2eqa(lat,lon,origin,units)` specifies the units for the inputs, where `units` is any valid angle units string. The default value is 'degrees'.

`mat = grn2eqa(lat,lon,origin...)` packs the outputs into a single variable.

**Description**

The `grn2eqa` function converts data from Greenwich-based latitude-longitude coordinates to equal-area  $x$ - $y$  coordinates. The opposite conversion can be performed with `eqa2grn`.

**Examples**

```
lats = [56 34]; longs = [-140 23];
[x,y] = grn2eqa(lats,longs)

x =
    -2.4435    0.4014
y =
    0.8290    0.5592
```

# grn2eqa

---

## See Also

eqa2grn, hista



**Purpose** Read Global Self-Consistent Hierarchical High-Resolution Shoreline

**Syntax**

```
S = gshhs(filename)
S = gshhs(filename, latlim, lonlim)
indexfilename = gshhs(filename, 'createindex')
```

`S = gshhs(filename)` reads GSHHS version 1.3 and earlier vector data for the entire world from `filename`. GSHHS files have names of the form `gshhs_X.b`, where `X` is one of the letters `c`, `l`, `i`, `h` and `f`, corresponding to increasing resolution (and file size). The result returned in `S` is a polygon Version 2 geographic data structure array (`geostruct2`).

`S = gshhs(filename, latlim, lonlim)` reads a subset of the vector data from `filename`. The limits of the desired data are specified as two-element vectors of latitude, `latlim`, and longitude, `lonlim`, in degrees. The elements of `latlim` and `lonlim` must be in ascending order. Longitude limits range from `[-180 195]`. If `latlim` is empty the latitude limits are `[-90 90]`. If `lonlim` is empty, the longitude limits are `[-180 195]`.

`indexfilename = gshhs(filename, 'createindex')` creates an index file for faster data access when requesting a subset of a larger dataset. The index file has the same name as the GSHHS data file, but with the extension `'i'`, instead of `'b'` and is written in the same directory as `filename`. The name of the index file is returned, but no coastline data are read. A call using this option should be followed by an additional call to `gshhs` to import actual data. On that and subsequent calls, `gshhs` detects the presence of the index file and uses it to access records by location much faster than it would without an index.

## Output Structure

The geostruct2 output structure S contains the following fields; all latitude and longitude values are in degrees:

Field Name	Field Contents
Geometry	'Polygon'
BoundingBox	[minLon minLat; maxLon maxLat]
Lon	Coordinate vector
Lat	Coordinate vector
South	Southern latitude boundary
North	Northern latitude boundary
West	Western longitude boundary
East	Eastern longitude boundary
Area	Area of polygon in square kilometers
Level	Scalar value ranging from 1 to 4, indicates level in topological hierarchy
LevelString	'land' or 'lake', or 'island_in_lake', or 'pond_in_island_in_lake' or 'other'
NumPoints	Number of points in the polygon
FormatVersion	Format version of data: empty if unspecified
Source	Source of data: 'WDBII' or 'WVS'
CrossGreenwich	Scalar flag: true if the polygon crosses the prime meridian, false otherwise
GSHHS_ID	Unique polygon scalar id number, starting at 0

## Remarks

If you are extracting data within specified geographic limits and using data other than coarse resolution, consider creating an index file first.

Also, to speed rendering when mapping very large amounts of data, you might want to plot the data as NaN-clipped lines rather than as patches.

Note that when you specify latitude-longitude limits, polygons that completely fall outside those limits are excluded, but no trimming of features that partially traverse the region is performed. If you want to eliminate data outside of a rectangular region of interest, you can use `maptrim` with the `Lat` and `Lon` fields of the `geostruct` returned by `gshhs` to clip the data to your region and still maintain polygon topology.

## Background

The Global Self-Consistent Hierarchical High-Resolution Shoreline was created by Paul Wessel of the University of Hawaii and Walter H.F. Smith of the NOAA Geosciences Lab. At the full resolution the data requires 85 MB uncompressed, but lower resolution versions are also provided. This database includes coastlines, major rivers, and lakes. The GSHHS data in various resolutions is available over the Internet from the National Oceanic and Atmospheric Administration, National Geophysical Data Center Web site.

Version 1.3 of the `gshhs_c.b` (coarse) data set ships with Mapping Toolbox in the `toolbox/map/mapdemos` directory. For details, type

```
type gshhs_c.txt
```

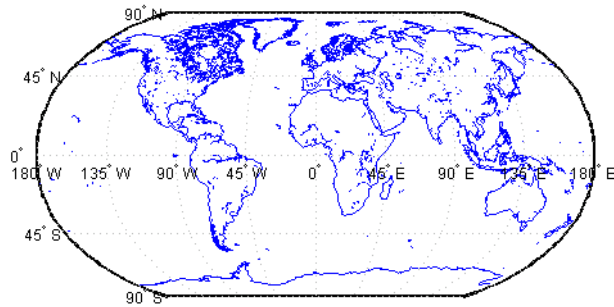
at the MATLAB command prompt.

## Examples

### Example 1

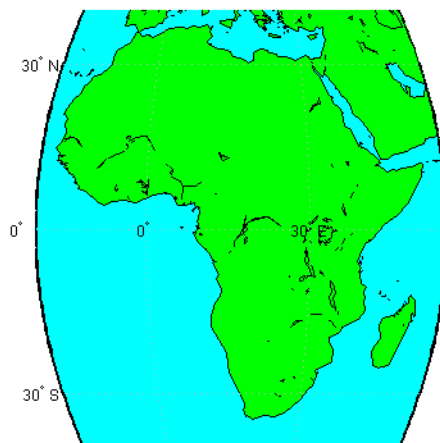
Read the entire coarse data set (located on the MATLAB path in `matlabroot/toolbox/map/mapdemos`) and display as a coastline:

```
filename = gunzip('gshhs_c.b.gz', tempdir);  
world = gshhs(filename{1});  
delete(filename{1})  
figure  
worldmap world  
geoshow([world.Lat], [world.Lon])
```



After creating an index file, read and display Africa as a green polygon; note that gshhs detects and uses the index file automatically:

```
filename = gunzip('gshhs_c.b.gz', tempdir);
indexname = gshhs(filename{1}, 'createindex');
figure
worldmap Africa
projection = gcm;
latlim = projection.maplatlimit;
lonlim = projection.maplonlimit;
africa = gshhs(filename{1}, latlim, lonlim);
delete(filename{1})
delete(indexname)
geoshow(africa, 'FaceColor', 'green')
setm(gca, 'FFaceColor', 'cyan')
```



---

**Note** The following examples use publicly available GSHHS data files that do not ship with Mapping Toolbox. For details on locating GSHHS data for download over the Internet, see the following documentation at the MathWorks Web site:  
<http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

### Example 2

Read all of the lowest resolution database:

```
s = gshhs('gshhs_c.b')
```

### Example 3

Read the intermediate resolution database for South America:

```
s = gshhs('gshhs_i.b',[-60 -15],[-90 -30])
```

### Example 4

Read the full-resolution file for East and West Falkland Islands (Islas Malvinas):

# gshhs

---

```
s = gshhs('gshhs_f.b', [-55 -50], [-65 -55])
```

## **Example 5**

Create the index file for the high-resolution database:

```
gshhs('gshhs_h.b', 'createindex')
```

## **See Also**

dcwdata, geoshow, maptrimp, shaperead, vmap0data, worldmap

**Purpose** Place text on map using mouse

**Syntax**

```
h = gtextm(string)
```

```
h = gtextm(string,PropertyName,PropertyValue,...)
```

`h = gtextm(string)` places the text object *string* at the position selected by mouse input. When this function is called, the current map axes are brought up and the cursor is activated for mouse-click position entry. The text object's handle is returned.

`h = gtextm(string,PropertyName,PropertyValue,...)` allows the specification of any properties supported by the MATLAB `text` function.

**Example**

Create map axes:

```
axesm('sinusoid','FEdgeColor','red')  
gtextm('hello world','FontWeight','bold')
```

Click inside the frame and the text appears.

**See Also**

`axesm`, `textm`

## Purpose

Read 30-arc-second global digital elevation data (GTOPO30)

## Syntax

```
[Z, refvec] = gtopo30(tilename)
[Z, refvec] = gtopo30(tilename, samplefactor)
[Z, refvec] = gtopo30(tilename, samplefactor, latlim, lonlim)
[Z, refvec] = gtopo30(dirname, samplefactor, latlim, lonlim)
```

[Z, refvec] = gtopo30(tilename) reads the GTOPO30 tile specified by tilename and returns the result as a regular data grid. tilename is a string which does not include an extension and indicates a GTOPO30 tile in the current directory or on the MATLAB path. If tilename is empty or omitted, a file browser will open for interactive selection of the GTOPO30 header file. The data is returned at full resolution with the latitude and longitude limits determined from the GTOPO30 tile. The data grid, Z, is returned as an array of elevations. Elevations are given in meters above mean sea level using WGS84 as a horizontal datum. refvec is the three-element referencing vector that geolocates Z.

[Z, refvec] = gtopo30(tilename, samplefactor) reads a subset of the elevation data from tilename. samplefactor is a scalar integer, which when equal to 1 reads the data at its full resolution. When samplefactor is an integer n greater than one, every nth point is read. If samplefactor is omitted or empty, it defaults to 1.

[Z, refvec] = gtopo30(tilename, samplefactor, latlim, lonlim) reads a subset of the elevation data from tilename. The limits of the desired data are specified as two-element vectors of latitude, latlim, and longitude, lonlim, in degrees. The elements of latlim and LONLIM must be in ascending order. Longitude limits range from [-180 180]. If latlim or lonlim is empty, the coordinate limits are determined from the file.

[Z, refvec] = gtopo30(dirname, samplefactor, latlim, lonlim) reads and concatenates data from multiple tiles within a GTOPO30 CD-ROM or directory structure. The dirname input is a string with the name of the directory which contains the GTOPO30 tile directories or GTOPO30 tiles. Within the tile directories are the uncompressed data files. The dirname for CD-ROMs distributed by the USGS is the device name of the CD-ROM drive. samplefactor if



omitted or empty defaults to 1. latlim if omitted or empty defaults to [-90 90]. lonlim if omitted or empty defaults to [-180 180].

When directory dirname contains no GTOPO30 data or tilename identifies a file with a .DEM extension that is not a GTOPO30 file, the function returns a single NaN in Z, a canonical refvec, and issues a warning.

The data and documentation are available over the Internet via http and anonymous ftp, as well as for purchase on CD-ROM.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

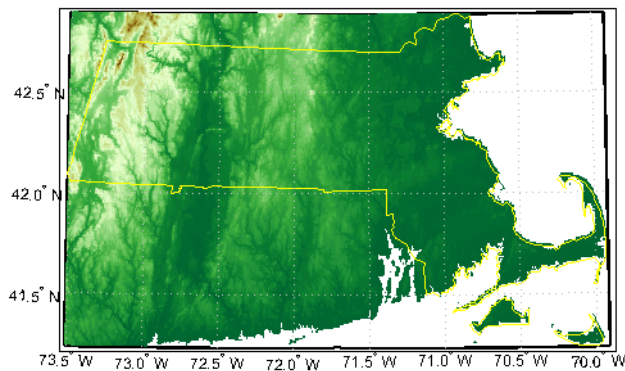
### Example 1

Extract and display full resolution data for the state of Massachusetts:

```
% Read stateline polygon boundary and calculate boundary limits.
Massachusetts = shaperead('usastatehi', 'UseGeoCoords', true, ...
    'Selector',{@(name) strcmpi(name,'Massachusetts'), 'Name'});
latlim = [min(Massachusetts.Lat(:)) max(Massachusetts.Lat(:))];
lonlim = [min(Massachusetts.Lon(:)) max(Massachusetts.Lon(:))];

% Read the GTOPO30 data at full resolution.
[Z,refvec] = gtopo30('W100N90',1,latlim,lonlim);

% Display the data grid and overlay the stateline boundary.
figure
usamap(Z,refvec);
geoshow(Z, refvec, 'DisplayType', 'surface')
colormap(demcmap(Z))
geoshow(Massachusetts,'DisplayType','polygon',...
    'facecolor','none','edgecolor','y')
```

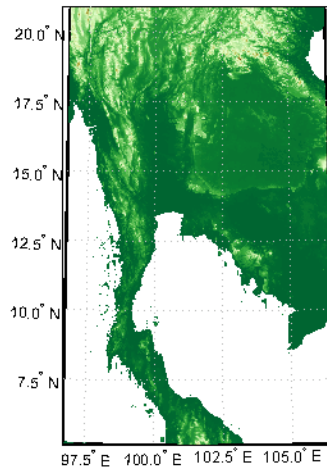


## Example 2

```
% Extract every 20th point from a tile.  
% Provide an empty filename and select the file interactively.  
[Z,refvec] = gtopo30([],20);
```

## Example 3

```
% Extract data for Thailand, an area which straddles two tiles.  
% The data is on CD number 3 distributed by the USGS.  
% The CD-device is 'F:\'  
latlim = [5.22 20.90];  
lonlim = [96.72 106.38];  
gtopo30s(latlim,lonlim)  
% Extract every fifth data point for Thailand.  
% Specify actual directory or mapped drive if not "F:\'  
[Z,refvec] = gtopo30('F:\',5,latlim,lonlim);  
worldmap(Z,refvec);  
geoshow(Z, refvec, 'DisplayType', 'surface')  
colormap(demcmap(Z))
```



#### Example 4

```
% Extract every 10th point from a column of data 5 degrees around  
% the prime meridian. The current directory contains GTOP030 data.  
[Z, refvec] = gtopo30(pwd, 10, [], [-5 5]);
```

#### See Also

gtopo30s, globedem, dted, satbath, tbase, usgsdem

# gtopo30s

---

**Purpose** GTOPO30 data filenames for latitude-longitude quadrangle

**Syntax** `fname = gtopo30s(latlim,lonlim)`  
`fname = gtopo30s(latlim,lonlim)` returns a cell array of the filenames covering the geographic region for GTOPO30 digital elevation maps (also referred to as “30-arc second” DEMs). `latlim` and `lonlim` specify the region as scalar latitude and longitude points, or two-element vectors of latitude and longitude limits in units of degrees.

**Remarks** The data and documentation are available over the Internet via `http` and anonymous `ftp`.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

**See Also** `gtopo30`

**Purpose** Handles of displayed map objects

**Syntax**

```
handlem or handlem('taglist')
h = handlem('prompt')
h = handlem(object)
handlem('object', axes)
handlem('object', axes, 'searchmethod')
h = handlem(handles)
```

handlem or handlem('taglist') displays a dialog box for selecting the objects for which you want handles.

h = handlem('prompt') displays another dialog box, which allows greater control of object selection.

h = handlem(object) returns the handles of those objects specified by the input string. The options for the object string are

'all'	All children of the current axes
'clabels'	Contour labels on the current map axes
'contour'	Contourgroup for contours on the current map axes
'contour3d'	3-D contour lines on the current map axes
'cpatches'	Filled contour patches on the current map axes
'frame'	Map frame
'grid'	Map grid lines
'hggroup'	All hggroup objects
'hidden'	Hidden objects on the current axes
'image'	Image objects on the current axes
'light'	Light objects on the current axes
'line'	Line objects on the current axes

# handlem

---

'map'	All objects on the map, excluding the frame (default)
'meridian'	Longitude grid lines
'mlabel'	Longitude labels
'parallel'	Latitude grid lines
'patch'	Patch objects on the current axes
'plabel'	Latitude labels
'scaleruler'	Scaleruler objects
'surface'	Surface objects on the current axes
'text'	Text objects on the current axes
'tissot'	Tissot indicatrices on the current map axes
'visible'	Visible objects on the current axes

Or any user-defined object tag string.

A prefix of 'all' can be applied to strings defining a Handle Graphics object type ('allimage', 'allline', 'allsurface', 'allpatch', 'alltext') to determine all object handles that meet the type criteria. Without the 'all' prefix, those objects named by the user with the tagm function are not included (e.g., a line with the tag 'route' would not be included for object string 'line', but would be for 'allline').

handlem('object', axesh) searches within the axes specified by the input handle axesh.

handlem('object', axesh, 'searchmethod') controls the method used to match the 'str' input. If omitted, 'exact' is assumed. Search method 'strmatch' searches for matches at the beginning of the tag, similar to the MATLAB strmatch function. Search method 'findstr' searches within the tag, similar to the MATLAB findstr function.

h = handlem(handles) returns those elements of an input vector of handles that are still valid.

**See Also**      clma, clmo, hidem, namem, showm, tagm

# hidem

---

## **Purpose**

Hide specified graphic objects on map axes

## **Syntax**

```
hidem  
hidem(handle)  
hidem(object)
```

hidem brings up a dialog box for selecting the objects to hide (set their `Visible` property to 'off').

hidem(handle) hides the objects specified by a vector of handles.

hidem(*object*) hides those objects specified by the *object* string, which can be any string recognized by the `handlem` function.

## **See Also**

clma, clmo, handlem, namem, showm, tagm



**Purpose**

Histogram for geographic points with equal-area bins

**Syntax**

```
[lat,lon,num] = hista(lats,lons)
```

```
[lat,lon,num] = hista(lats,lons,binarea)
```

```
[lat,lon,num] = hista(lats,lons,binarea,ellipsoid)
```

```
[lat,lon,num] = hista(lats,lons,binarea,units)
```

`[lat,lon,num] = hista(lats,lons)` returns the center coordinates of equal-area bins and the number of observations falling in each based on the geographically distributed input data.

`[lat,lon,num] = hista(lats,lons,binarea)` specifies the equal-area bin size, in square kilometers. It is 100 km<sup>2</sup> by default.

`[lat,lon,num] = hista(lats,lons,binarea,ellipsoid)` specifies the elliptical definition of the Earth to be used with the two-element ellipsoid vector. The default ellipsoid model is a spherical Earth, which is sufficient for most applications.

`[lat,lon,num] = hista(lats,lons,binarea,units)` specifies the standard angle unit string. The default value is 'degrees'.

**Examples**

Create random data:

```
lats = rand(4)
```

```
lats =
```

```
    0.4451    0.8462    0.8381    0.8318  
    0.9318    0.5252    0.0196    0.5028  
    0.4660    0.2026    0.6813    0.7095  
    0.4186    0.6721    0.3795    0.4289
```

```
longs = rand(4)
```

```
longs =
```

```
    0.3046    0.3028    0.3784    0.4966  
    0.1897    0.5417    0.8600    0.8998  
    0.1934    0.1509    0.8537    0.8216  
    0.6822    0.6979    0.5936    0.6449
```

# hista

---

Bin the data in 50-by-50 km cells (2500 sq km):

```
[lat,lon,num] = hista(lats,longs,2500);  
[lat lon num]
```

```
ans =
```

```
0.2574    0.3757    4.0000  
0.7070    0.3757    5.0000  
-0.1923    0.8253    1.0000  
0.2573    0.8253    2.0000  
0.7070    0.8254    4.0000
```

## See Also

eqa2grn, grn2eqa, histr

**Purpose**

Histogram for geographic points with equirectangular bins

**Syntax**

```
[lat,lon,num,wnum] = histr(lats,lons)
```

```
[lat,lon,num,wnum] = histr(lats,lons,units)
```

```
[lat,lon,num,wnum] = histr(lats,lons,bindensty)
```

`[lat,lon,num,wnum] = histr(lats,lons)` returns the center coordinates of equal-rectangular bins and the number of observations, `num`, falling in each based on the geographically distributed input data. Additionally, an area-weighted observation value, `wnum`, is returned. `wnum` is the bin's `num` divided by its normalized area. The largest bin has the same `num` and `wnum`; a smaller bin has a larger `wnum` than `num`.

`[lat,lon,num,wnum] = histr(lats,lons,units)` specifies the standard angle unit string. The default value is 'degrees'.

`[lat,lon,num,wnum] = histr(lats,lons,bindensty)` sets the number of bins per angular unit. For example, if `units` is 'degrees', a `bindensty` of 10 would be 10 bins per degree of latitude or longitude, resulting in 100 bins per *square* degree. The default is one cell per angular unit.

**Description**

The `histr` function sorts geographic data into equirectangular bins for histogram purposes. Equirectangular in this context means that each bin has the same angular measurement on each side (e.g., 1°-by-1°). Consequently, the result is not an equal-area histogram. The `hista` function provides that capability. However, the results of `histr` can be weighted by their area bias to correct for this, in some sense.

**Examples**

Create random data:

```
lats = rand(4)
```

```
lats =
```

```
0.4451    0.8462    0.8381    0.8318
0.9318    0.5252    0.0196    0.5028
0.4660    0.2026    0.6813    0.7095
0.4186    0.6721    0.3795    0.4289
```

# histr

---

```
longs = rand(4)
```

```
longs =  
    0.3046    0.3028    0.3784    0.4966  
    0.1897    0.5417    0.8600    0.8998  
    0.1934    0.1509    0.8537    0.8216  
    0.6822    0.6979    0.5936    0.6449
```

Bin the data in 0.5-by-0.5 degree cells (two bins per degree):

```
[lat,lon,num,wnum] = histr(lats,longs,2);  
[lat,lon,num,wnum]
```

```
ans =  
    0.2500    0.2500    3.0000    3.0000  
    0.7500    0.2500    4.0000    4.0003  
    0.2500    0.7500    4.0000    4.0000  
    0.7500    0.7500    5.0000    5.0004
```

The bins centered at 0.75°N are slightly smaller in area than the others. wnum reflects the relative count per normalized unit area.

## See Also

filterm, hista

**Purpose** Convert time from hrs:min:sec to hrs:min encoding

**Syntax**

---

**Note** The `hms2hm` function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox.

---

```
timeout = hms2hm(timein)
```

`timeout = hms2hm(timein)` rounds times input in hours-minutes-seconds (*hms*) format to the appropriate value in hours-minutes (*hm*) format. This special handling is needed because there are 60, not 100, seconds in a minute.

**Example**

Round 12:34:29 and 12:34:31 to hm format:

```
timeout = hms2hm([1234.29 1234.31])
```

```
timeout =  
    1234      1235
```

# hms2hr, hms2sec

---

## Purpose

Convert time from hrs:min:sec encoding to hours or seconds

## Syntax

---

**Note** The `hms2hr` and `hms2sec` functions are obsolete and error when used. They will be completely removed from the next version of Mapping Toolbox..

---

```
timeout = hms2hr(timein)
```

```
timeout = hms2sec(timein)
```

`timeout = hms2hr(timein)` converts times input in hours-minutes-seconds (*hms*) format to the equivalent measure in decimal hours.

`timeout = hms2sec(timein)` converts times input in hours-minutes-seconds (*hms*) format to the equivalent measure in seconds.

## Remarks

The inputs can be in hours-minutes (*hm*) format, since numerically they look like *hms* format, in which seconds are always zero.

## Example

```
hms2hr(1230)
```

```
ans =  
    12.5000
```

```
hms2sec(100.10)
```

```
ans =  
    3610
```

**Purpose**

Expand hrs:min:sec encoded vector to [hrs min sec] matrix

**Syntax**

---

**Note** The hms2mat function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox.

---

```
[h,m,s] = hms2mat(timein)
[h,m,s] = hms2mat(timein,n)
matout = hms2mat(timein,n)
```

[h,m,s] = hms2mat(timein) takes times in hms format and splits their components into three outputs, one each for hours, minutes, and seconds.

[h,m,s] = hms2mat(timein,n) specifies the power of 10, n, to which the resulting seconds output should be rounded (that is, if a result is 12.567 seconds and n = -2, the resulting seconds output would be 12.57). The default value of n is -5.

matout = hms2mat(timein,n) returns a three-column matrix, matout, in which the columns represent hours, minutes, and seconds, respectively. In this case, timein must be a vector.

**Examples**

```
[h,m,s] = hms2mat(1234.567)
```

```
h =
    12
m =
    34
s =
   56.7000
```

```
matout = hms2mat(1234.567)
```

```
matout =
   12.0000   34.0000   56.7000
```

# hr2hms, hr2hm

---

## Purpose

Convert time from hours to hrs:min:sec or hrs:min encoding

## Syntax

---

**Note** The `hr2hm` and `hr2hms` functions are obsolete and error when used. They will be completely removed from the next version of Mapping Toolbox.

---

```
timeout = hr2hms(timein)
timeout = hr2hm(timein)
```

`timeout = hr2hms(timein)` converts times input in hours to the equivalent measure in the hours-minutes-seconds (*hms*) format.

`timeout = hr2hm(timein)` converts times input in hours to the equivalent measure in the hours-minutes (*hm*) format. This is the *hms* format, properly rounded to just hours and minutes.

## Example

```
hr2hms(12.51)

ans =
    1230.36

hr2hm(12.51)

ans =
    1231.00
```



**Purpose** Convert time from hours to seconds

**Syntax**

---

**Note** The hr2sec function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox.

---

```
timeout = hr2sec(timein)
```

timeout = hr2sec(timein) converts times input in hours to the equivalent measure in seconds.

**Example**

```
hr2sec(1)
```

```
ans =  
    3600
```

# imbedm

---

## Purpose

Encode data points into regular data grid

## Syntax

```
newgrid = imbedm(lat,lon,value,Z,refvec)
newgrid = imbedm(lat,lon,value,Z,refvec,units)
[newgrid,badindx] = imbedm(lat,lon,value,Z,refvec,units)
```

`newgrid = imbedm(lat,lon,value,Z,refvec)` resets certain entries of a regular data grid, `Z`. The entries to be reset correspond to the locations defined by the latitude and longitude position vectors `lat` and `lon`. The entries are reset to the same number if `value` is a scalar, or to individually specified numbers if `value` is a vector the same size as `lat` and `lon`. If any points lie outside the input grid `Z`, a warning is displayed.

`newgrid = imbedm(lat,lon,value,Z,refvec,units)` specifies the units of the vectors `lat` and `lon`, where `units` is any valid angle units string ('degrees' by default).

`[newgrid,badindx] = imbedm(lat,lon,value,Z,refvec,units)` returns the indices of `lat` and `lon` corresponding to points outside grid `Z` in the variable `badindx`.

## Examples

Create a simple grid map and embed new values in it:

```
Z = ones(3,6)

Z =
     1     1     1     1     1     1
     1     1     1     1     1     1
     1     1     1     1     1     1
refvec = [1/60 90 -180]

refvec =
    0.0167    90.0000 -180.0000

newgrid = imbedm([23 -23], [45 -45],[5 5],Z,refvec)

newgrid =
```

1	1	1	1	1	1
1	1	5	5	1	1
1	1	1	1	1	1

**See Also** `ltln2val`, `setpostn`

# ind2rgb8

---

**Purpose** Convert indexed image to uint8 RGB image

**Syntax** `RGB = ind2rgb8(X,cmap)`  
`RGB = ind2rgb8(X,cmap)` creates an RGB image of class `uint8`. `X` must be `uint8`, `uint16`, or `double`, and `cmap` must be a valid MATLAB colormap.

**Example**

```
% Convert the 'concord_ortho_e.tif' image to RGB.  
[X, cmap] = imread('concord_ortho_e.tif');  
RGB = ind2rgb8(X, cmap);  
R = worldfileread('concord_ortho_e.tfw');  
mapshow(RGB, R);
```

**See Also** `ind2rgb`

---

<b>Purpose</b>	Latitudes and longitudes of mouse-click locations
<b>Syntax</b>	<pre>[lat, lon] = inputm [lat, lon] = inputm(n) [lat, lon] = inputm(n,h) [lat, lon, button] = inputm(n) MAT = inputm(...)</pre> <p>[lat, lon] = inputm returns the latitudes and longitudes in geographic coordinates of points selected by mouse clicks on a displayed grid. The point selection continues until the return key is pressed.</p> <p>[lat, lon] = inputm(n) returns n points specified by mouse clicks.</p> <p>[lat, lon] = inputm(n,h) prompts for points from the map axes specified by the handle h. If omitted, the current axes (gca) is assumed.</p> <p>[lat, lon, button] = inputm(n) returns a third result, button, that contains a vector of integers specifying which mouse button was used (1,2,3 from left) or ASCII numbers if a key on the keyboard was used.</p> <p>MAT = inputm(...) returns a single matrix, where MAT = [lat lon].</p>
<b>Remarks</b>	<p>inputm works much like the standard MATLAB ginput, except that the returned values are latitudes and longitudes extracted from the projection, rather than axes <i>x-y</i> coordinates. If you click outside of the projection bounds (beyond the map frame in the corners of a Robinson projection, for example), no coordinates are returned for that location.</p> <p>inputm cannot be used with a 3-D display, including those created using globe.</p>
<b>See Also</b>	gcpmap, ginput (MATLAB function)

# interp

---

## Purpose

Densify latitude-longitude sampling in lines or polygons

## Syntax

```
[latout,lonout] = interp(lat,lon,maxdiff)
[latout,lonout] = interp(lat,lon,maxdiff,method)
[latout,lonout] = interp(lat,lon,maxdiff,method,units)
```

[latout,lonout] = interp(lat,lon,maxdiff) fills in any gaps in latitude (*lat*) or longitude (*lon*) data vectors that are greater than a defined tolerance *maxdiff* apart in either dimension. The angle units of the three inputs need not be specified, but they must be identical. *latout* and *lonout* are the new latitude and longitude data vectors, in which any gaps larger than *maxdiff* in the original vectors have been filled with additional points. The default method of interpolation used by *interp* is linear.

[latout,lonout] = interp(lat,lon,maxdiff,method) interpolates between vector data coordinate points using a specified interpolation method. Valid interpolation method strings are 'gc' for great circle, 'rh' for rhumb line, and 'lin' for linear interpolation.

[latout,lonout] = interp(lat,lon,maxdiff,method,units) specifies the units used, where *units* is any valid angle units string. The default is 'degrees'.

## Examples

```
lat = [1 2 4 5]; lon = [7 8 9 11];
[latout,lonout] = interp(lat,lon,1);
[latout lonout]
```

```
ans =
    1.0000    7.0000
    2.0000    8.0000
    3.0000    8.5000
    4.0000    9.0000
    4.5000   10.0000
    5.0000   11.0000
```

## See Also

intrplat, intrplon

**Purpose**

Interpolate latitude at given longitude

**Syntax**

```
newlat = intrplat(long,lat,newlong)
newlat = intrplat(long,lat,newlong,method)
newlat = intrplat(long,lat,newlong,method,units)
```

`newlat = intrplat(long,lat,newlong)` returns an interpolated latitude, `newlat`, corresponding to a longitude `newlong`. `long` must be a monotonic vector of longitude values. The actual entries must be monotonic; that is, the longitude vector [350 357 3 10] is not allowed even though the geographic *direction* is unchanged (use [350 357 363 370] instead). `lat` is a vector of the latitude values paired with each entry in `long`.

`newlat = intrplat(long,lat,newlong,method)` specifies the method of interpolation employed. The default value of the `method` string is 'linear', which results in linear, or Cartesian, interpolation between the numerical values entered. This is really just a pass-through to the MATLAB `interp1` function. Similarly, 'spline' and 'cubic' perform cubic spline and cubic interpolation, respectively. The 'rh' method returns interpolated points that lie on rhumb lines between input data. Similarly, the 'gc' method returns interpolated points that lie on great circles between input data.

`newlat = intrplat(long,lat,newlong,method,units)` specifies the units used, where `units` is any valid angle units string. The default is 'degrees'.

**Description**

The function `intrplat` is a geographic data analogy of the standard MATLAB function `interp1`.

**Examples**

Compare the results of the various methods:

```
lats = [25 45]; longs = [30 60];
newlat = intrplat(longs,lats,45,'linear')
```

```
newlat =
    35
```

# intrplat

---

```
newlat = intrplat(longs,lats,45,'rh')
```

```
newlat =  
    35.6213
```

```
newlat = intrplat(longs,lats,45,'gc')
```

```
newlat =  
    37.1991
```

## Remarks

There are separate functions for interpolating latitudes and longitudes, for although the cases are identical when using those methods supported by `interp1`, when latitudes and longitudes are treated like the spherical angles they are (using `'rh'` or `'gc'`), the results are different. Compare the example above to the example under `intrplon`, which reverses the values of latitude and longitude.

## See Also

`interp1`, `intrplon`



**Purpose**

Interpolate longitude at given latitude

**Syntax**

```
newlon = intrplon(lat,lon,newlat)
newlon = intrplon(lat,lon,newlat,method)
newlon = intrplon(lat,lon,newlat,method,units)
```

`newlon = intrplon(lat,lon,newlat)` returns an interpolated longitude, `newlon`, corresponding to a latitude `newlat`. `lat` must be a monotonic vector of longitude values. `lon` is a vector of the longitude values paired with each entry in `lat`.

`newlon = intrplon(lat,lon,newlat,method)` specifies the method of interpolation employed. The default value of the `method` string is 'linear', which results in linear, or Cartesian, interpolation between the numerical values entered. This is really just a pass-through to the MATLAB `interp1` function. Similarly, 'spline' and 'cubic' perform cubic spline and cubic interpolation, respectively. The 'rh' method returns interpolated points that lie on rhumb lines between input data. Similarly, the 'gc' method returns interpolated points that lie on great circles between input data.

`newlon = intrplon(lat,lon,newlat,method,units)` specifies the units used, where `units` is any valid angle units string. The default is 'degrees'.

**Description**

The function `intrplon` is a geographic data analogy of the MATLAB function `interp1`.

**Examples**

Compare the results of the various methods:

```
long = [25 45]; lat = [30 60];
newlon = intrplon(lat,long,45,'linear')
```

```
newlon =
    35
```

```
newlon = intrplon(lat,long,45,'rh')
```

# intrplon

---

```
newlon =  
    33.6515  
  
newlon = intrplon(lat,long,45,'gc')  
  
newlon =  
    32.0526
```

## Remarks

There are separate functions for interpolating latitudes and longitudes, for although the cases are identical when using those methods supported by `interp1`, when latitudes and longitudes are treated like the spherical angles they are (using 'rh' or 'gc'), the results are different. Compare the previous example to the example under `intrplat`, which reverses the values of latitude and longitude.

## See Also

`interp1`, `intrplat`

<b>Purpose</b>	True for axes with map projection
<b>Syntax</b>	<pre>mflag = ismap mflag = ismap(hndl) [mflag,msg] = ismap(hndl)</pre> <p>mflag = ismap returns a 1 if the current axes is a map axes, and 0 otherwise.</p> <p>mflag = ismap(hndl) specifies the handle of the axes to be tested.</p> <p>[mflag,msg] = ismap(hndl) returns a string message if the axes is not a map axes, specifying why not.</p>
<b>Description</b>	The ismap function tests an axes object to determine whether it is a map axes.
<b>See Also</b>	gcm, ismapped

# ismapped

---

## **Purpose**

True, if object is projected on map axes

## **Syntax**

```
mflag = ismapped
```

```
mflag = ismapped(hndl)
```

```
[mflag,msg] = ismapped(hndl)
```

`mflag = ismapped` returns a 1 if the current object is projected on a map axes, and 0 otherwise.

`mflag = ismapped(hndl)` specifies the handle of the object to be tested.

`[mflag,msg] = ismapped(hndl)` returns a string message if the axes is not projected on a map axes, specifying why not.

## **Description**

The `ismapped` function tests an object to determine whether it is projected on map axes.

## **See Also**

`gcm`, `ismap`

**Purpose**

True if polygon vertices are in clockwise order

**Syntax**

```
tf = ispolycw(x, y)
```

`tf = ispolycw(x, y)` returns true if the polygonal contour vertices represented by `x` and `y` are ordered in the clockwise direction. `x` and `y` are numeric vectors with the same number of elements.

Alternatively, `x` and `y` can contain multiple contours, either in NaN-separated vector form or in cell array form. In that case, `ispolycw` returns a logical array containing one true or false value per contour.

`ispolycw` always returns true for polygonal contours containing two or fewer vertices.

Vertex ordering is not well defined for self-intersecting polygonal contours. For such contours, `ispolycw` returns a result based on the order of vertices immediately before and after the left-most of the lowest vertices. In other words, of the vertices with the lowest `y` value, find the vertex with the lowest `x` value. For a few special cases of self-intersecting contours, the vertex ordering cannot be determined using only the left-most of the lowest vertices; for these cases, `ispolycw` uses a signed area test to determine the ordering.

**Class Support**

`x` and `y` may be any numeric class.

**Example**

Orientation of a square:

```
x = [0 1 1 0 0];  
y = [0 0 1 1 0];  
ispolycw(x, y)           % Returns 0  
ispolycw(fliplr(x), fliplr(y)) % Returns 1
```

**See Also**

`poly2cw`, `poly2ccw`, `polybool`

# isShapeMultipart

---

**Purpose** True, if polygon or line has multiple parts

**Syntax** `tf = isShapeMultipart(xdata, ydata)`  
`tf = isShapeMultipart(xdata, ydata)` returns 1 (true) if the polygon or line shape specified by `xdata` and `ydata` consists of multiple NaN-separated parts (i.e. has inner or multiple polygon rings or multiple line segments). The coordinate arrays `xdata` and `ydata` must match in size and have identical NaN locations.

**Examples**

```
isShapeMultipart([0 0 1],[0 1 0])

ans =
     0

isShapeMultipart([0 0 1 NaN 2 2 3 3],[0 1 0 NaN 2 3 3 2])

ans =
     1

load coast
isShapeMultipart(lat, long)

ans =
     1

S = shaperead('concord_hydro_area');
isShapeMultipart( S(1).X, S(1).Y)

ans =
     0

isShapeMultipart(S(14).X, S(14).Y)

ans =
     1
```

**See Also**      polysplit

# km2deg, nm2deg, sm2deg

---

**Purpose** Convert from distance units to degrees

**Syntax**

```
deg = km2deg(km)
deg = nm2deg(nm)
deg = sm2deg(sm)
deg = km2deg(km, radius)
deg = nm2deg(nm, radius)
deg = sm2deg(sm, radius)
deg = km2deg(km, sphere)
deg = nm2deg(nm, sphere)
deg = sm2deg(sm, sphere)
```

**Description** `deg = km2deg(km)` converts distances from kilometers to degrees as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`deg = nm2deg(nm)` converts distances from nautical miles to degrees as measured along a great circle on a sphere with a radius of 6371 km (3440.065 nm), the mean radius of the Earth.

`deg = sm2deg(sm)` converts distances from statute miles to degrees as measured along a great circle on a sphere with a radius of 6371 km (3958.748 sm), the mean radius of the Earth.

`deg = km2deg(km, radius)` converts distances from kilometers to degrees as measured along a great circle on a sphere having the specified radius. `radius` must be in units of kilometers.

`deg = nm2deg(nm, radius)` and `deg = sm2deg(sm, radius)` work identically, except that both the input distance and radius must be in nautical miles and statute miles, respectively.

`deg = km2deg(km, sphere)` converts distances from kilometers to degrees, as measured along a great circle on a sphere approximating an object in the Solar System. `sphere` may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive.



`deg = nm2deg(nm, sphere)` and `deg = sm2deg(sm, sphere)` work identically, except that the input units are nautical miles and statute miles, respectively.

## Examples

Two cities are 340 km apart. How many degrees of arc is that? How many degrees would it be if the cities were on Mars?

```
deg = km2deg(340)
```

```
deg =  
3.0577
```

```
deg = km2deg(340, 'mars')
```

```
deg =  
5.7465
```

## See Also

`deg2rad`, `rad2deg`, `deg2km`, `km2rad`, `km2nm`, `km2sm`, `deg2nm`, `nm2deg`, `nm2km`, `nm2sm`, `deg2sm`, `sm2deg`, `sm2km`, `sm2nm`

# km2rad, nm2rad, sm2rad

---

**Purpose** Convert from distance units to radians

**Syntax**

```
rad = km2rad(km)
rad = nm2rad(nm)
rad = sm2rad(sm)
rad = km2rad(km,radius)
rad = nm2rad(nm,radius)
rad = sm2rad(sm,radius)
rad = km2rad(km,sphere)
rad = nm2rad(nm,sphere)
rad = sm2rad(sm,sphere)
```

**Description** `rad = km2rad(km)` converts distances from kilometers to radians as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`rad = nm2rad(nm)`, and `rad = sm2rad(sm)` work identically, except that the input units are nautical miles and statute miles, respectively.

`rad = km2rad(km,radius)` converts distances from kilometers to radians as measured along a great circle on a sphere having the specified radius. `radius` must be in units of kilometers.

`rad = nm2rad(nm,radius)` and `rad = sm2rad(sm,radius)` work identically, except that both the input distance and radius must be in nautical miles and statute miles, respectively.

`rad = km2rad(km,sphere)` converts distances from kilometers to radians, as measured along a great circle on a sphere approximating an object in the Solar System. `sphere` may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive.

`rad = nm2rad(nm,sphere)` and `rad = sm2rad(sm,sphere)` work identically, except that the input units must be nautical miles and statute miles, respectively.

## Examples

How many radians does 1,000 km span on the Earth and on the Moon?

```
rad = km2rad(1000)
```

```
rad =  
0.1570
```

```
rad = km2rad(1000, 'moon')
```

```
rad =  
0.5754
```

## See Also

deg2rad, rad2deg, rad2km, km2deg, km2nm, km2sm, rad2nm, nm2deg,  
nm2km, nm2sm, rad2sm, sm2deg, sm2km, sm2nm

# km2nm, km2sm, nm2km, nm2sm, sm2km, sm2nm

---

**Purpose** Convert distance between kilometers and miles

**Syntax**

$$\begin{aligned} \text{nm} &= \text{km2nm}(\text{km}) \\ \text{sm} &= \text{km2sm}(\text{km}) \\ \text{km} &= \text{nm2km}(\text{nm}) \\ \text{sm} &= \text{nm2sm}(\text{nm}) \\ \text{km} &= \text{sm2km}(\text{sm}) \\ \text{nm} &= \text{sm2nm}(\text{sm}) \end{aligned}$$

**Description**

$\text{nm} = \text{km2nm}(\text{km})$  converts distances from kilometers to nautical miles.

$\text{sm} = \text{km2sm}(\text{km})$  works identically, except that the output units are statute miles.

$\text{km} = \text{nm2km}(\text{nm})$  converts distances from nautical miles to kilometers.

$\text{sm} = \text{nm2sm}(\text{nm})$  works identically, except that the output units are statute miles.

$\text{km} = \text{sm2km}(\text{sm})$  converts distances from statute miles to kilometers.

$\text{nm} = \text{sm2nm}(\text{sm})$  works identically, except that the output units are nautical miles.

**Examples** How many statute miles is a *10k run*?

$$\text{sm} = \text{km2sm}(10)$$
$$\begin{aligned} \text{sm} &= \\ &6.2137 \end{aligned}$$

How fast is 30 knots (nautical miles per hour) in kph?

$$\text{km} = \text{nm2km}(30)$$
$$\begin{aligned} \text{km} &= \\ &55.5600 \end{aligned}$$

**See Also**  $\text{deg2km}$ ,  $\text{km2deg}$ ,  $\text{km2rad}$ ,  $\text{rad2km}$ ,  $\text{deg2nm}$ ,  $\text{nm2deg}$ ,  $\text{nm2rad}$ ,  $\text{rad2nm}$ ,  $\text{deg2sm}$ ,  $\text{sm2deg}$ ,  $\text{deg2sm}$ ,  $\text{sm2rad}$ ,  $\text{rad2sm}$

**Purpose**

Write geographic data to KML file

**Syntax**

```
kmlwrite(filename, lat, lon)
```

```
kmlwrite(filename, S)
```

```
kmlwrite(filename, address)
```

```
kmlwrite(..., param1, val1, param2, val2, ...)
```

`kmlwrite(filename, lat, lon)` writes the latitude and longitude points `lat` and `lon`, to disk in KML format. KML is an XML dialect used by the Google Earth™ and Google Maps™ mapping services and similar applications. `lat` and `lon` are numeric vectors, specified in degrees. `lat` must be in the range `[-90, 90]`. There is no range constraint on `lon`; all longitudes are automatically wrapped to the range `[-180, 180]`, to adhere to the KML specification. `filename` must be a character string specifying the output file name and location. If an extension is included, it must be `.kml`.

`kmlwrite(filename, S)` writes a point or multipoint geographic data structure to disk in KML format. The Geometry field of `S` must be either `'Point'` or `'Multipoint'`. `S` must include `Lat` and `Lon` fields. (If `S` includes `X` and `Y` fields an error is issued). The attribute fields of `S` are presented as a table in the description tag of the placemark displayed for each element of `S`, in the same order as they appear in `S`.

`kmlwrite(filename, address)` specifies the location of a KML Placemark via an address string or cell array of strings. Each string represents an unstructured address with city, state, and/or postal code. If `address` is a cell array, each cell contains the address of a unique point.

`kmlwrite(..., param1, val1, param2, val2, ...)` specifies parameter-value pairs that set additional KML feature properties. Parameter names can be abbreviated and are case-insensitive.

The parameter-value pairs are listed below:

- **Name** — A string or cell array of strings that specifies a name displayed in the viewer as the label for the object. If the value is a string, the name is applied to all objects. If the value is a cell array, it must match in size to `lat` and `lon`, `S`, or `address`.

- **Description** — A string, cell array of strings, or an attribute spec, that specifies the contents to be displayed in the feature's description tag(s). The description appears in the description balloon when the user clicks either the feature name in the Google Earth Places panel or clicks the placemark icon in the viewer window. If the value is a string, the description is applied to all objects. If the value is a cell array, it must match the size of lat and lon, S, or address. Use a cell array to customize descriptive tags for different placemarks.

Description elements can be either plain text or marked up with HTML. When it is plain text, Google Earth applies basic formatting, replacing each newline with `<br>` and giving anchor tags to all valid URLs for the World Wide Web. The URL strings are converted to hyperlinks. This means that you do not need to surround a URL with `<A HREF>` tags in order to create a simple link. Examples of HTML tags recognized by Google Earth are provided on its Web site, <http://earth.google.com>.

- **Icon** — A string or cell array of strings that specifies a custom icon filename. If the value is a string, the value is applied to all objects. If the value is a cell array, it must have the same size as lat and lon, S, or address. If the icon filename is not in the current directory, or in a directory on the MATLAB path, specify a full or relative path name. The string can be an Internet URL. The URL must include the protocol prefix (e.g., `http://`).
- **IconScale** — A positive numeric scalar or array that specifies a scaling factor for the icon. If the value is a scalar, the value is applied to all objects. If the value is an array, it must have the same size as lat and lon, S, or address.

## Remarks

### Using an Attribute Spec to Control Formatting of Attributes

An attribute spec is a structure with field names of attributes that controls how the table is displayed in its description balloon. In it, each field name you want to display has two fields, `Format` and `AttributeLabel`.

When you provide an attribute spec, the attribute fields of S are displayed as a table in the description tag of the placemark for each element of S. It only works with geostruct input; if you specify an attribute spec with lat and lon input syntax, the attribute spec is ignored. The attribute spec can control:

- Which attributes are included in the table
- The name for the attribute
- The order in which attributes appear
- The formatting of attributes

The easiest way to construct an attribute spec is to call `makeattribspec`, and then modify the output to remove attributes or change the Format field for one or more attributes. The lat and lon fields of S are never treated as attributes.

### **Viewing the KML file with a Google Earth browser**

A KML file may be displayed in a Google Earth browser. Google Earth must be installed on the system. On Windows platforms you can display the KML file with:

```
winopen(filename)
```

For Unix and MAC users, display the KML file with:

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

### **Viewing the KML file with Google Maps**

You can view KML files at the Google Maps mapping service Web site in addition to using Google Earth (an installed application). To do so, the file must be located on a web server that is accessible from the Internet. A private intranet server will not suffice, because Google's server must be able to access the URL that you provide to it. Here is a template for using Google Maps:

```
GMAPS_URL = 'http://maps.google.com/maps?q=';  
KML_URL = 'http://<your web server and path to your KML file>';  
web([GMAPS KML_URL])
```

You can only display a limited number of placemarks on a Google Maps page, and all placemarks must be geolocated using latitude-longitude coordinates (address-based placemarks are not supported). Google Maps for Mobile has further restrictions. See the Google KML documentation for more information.

## Examples

### Example 1 – Write a single point to a KML file

Add a description containing HTML markup, a name, and provide the location of an icon to display. Specifying an icon as a URL from the Web (as opposed to specifying one from a local file) makes the icon accessible to users of Google Maps as well as to Google Earth users.

```
% Write a single point to a KML file.  
% Add a description containing HTML, a name and an icon.  
lat = 42.299827;  
lon = -71.350273;  
description = sprintf('%s<br>%s</br><br>%s</br>', ...  
    '3 Apple Hill Drive', 'Natick, MA. 01760', ...  
    'http://www.mathworks.com');  
name = 'The MathWorks, Inc.';  
filename = 'The_MathWorks.kml';  
kmlwrite(filename, lat, lon, ...  
    'Description', description, 'Name', name, 'Icon', ...  
    'http://www.mathworks.com/products/product_listing/images/ml_icon.gif');
```

### Example 2 – Write the locations of major European cities to a KML file

Include the names of the cities, and remove the default description table:

```
latlim = [ 30; 75];  
lonlim = [-25; 45];  
cities = shaperead('worldcities.shp','UseGeoCoords', true, ...
```



```

    'BoundingBox', [lonlim, latlim]);
filename = 'European_Cities.kml';
kmlwrite(filename, cities, 'Name', {cities.Name}, 'Description', {});

```

### Example 3 – Write the locations of several Australian cities to a KML file

List the addresses to be displayed in a cell array:

```

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

```

### Example 4 – Unproject locations of Boston landmarks and write to a KML file

The Boston placenames file contains points stored in projected coordinates of meters, but Earth browser require geographic coordinates (latitudes and longitudes). Begin by converting coordinates from meters to survey feet, inverting the projection to latitudes and longitudes, and then adding the latitudes and longitudes to the geostruct. To unproject properly, use the projection information extracted from the GeoTIFF file `boston.tif`:

```

S = shaperead('boston_placenames');
proj = geotiffinfo('boston.tif');
surveyFeetPerMeter = unitsratio('sf','meter');
for k=1:numel(S)
    x = surveyFeetPerMeter * S(k).X;
    y = surveyFeetPerMeter * S(k).Y;
    [S(k).Lat, S(k).Lon] = projinv(proj, x, y);
end
filename = 'Boston_Placenames.kml';
kmlwrite(filename, S, 'Name', {S.NAME});

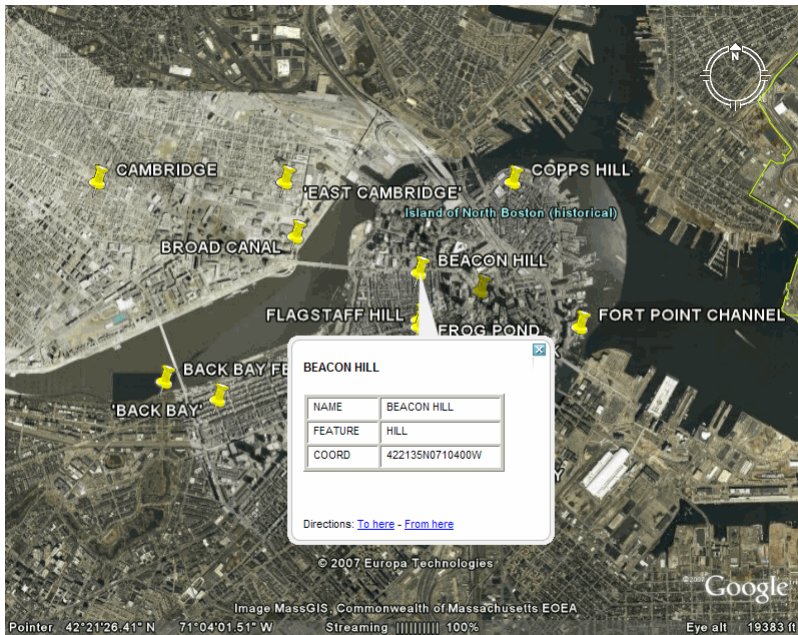
```

If you have Google Earth installed, you can view the file on Windows as follows:

```
winopen(filename)
```

On Unix or MAC, use:

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



For a different view of this location and placename data, see “Tour Boston with the Map Viewer” on page 1-9.

## See Also

geoshow, makeattribspec, shaperead, shapewrite

**Purpose**

Convert latitude-longitude coordinates to pixel coordinates

**Syntax**

```
[row, col ] = latlon2pix(R,lat,lon)
```

`[row, col ] = latlon2pix(R,lat,lon)` calculates pixel coordinates `row`, `col` from latitude-longitude coordinates `lat`, `lon`. `R` is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. `lat` and `lon` are vectors or arrays of matching size. The outputs `row` and `col` have the same size as `lat` and `lon`. `lat` and `lon` must be in degrees.

**Description**

Longitude wrapping is handled in the following way: Results are invariant under the substitution `lon = lon +/- n * 360` where `n` is an integer. Any point on the Earth that is included in the image or gridded data set corresponding to `r` will yield row/column values between 0.5 and 0.5 + the image height/width, regardless of what longitude convention is used.

**Example**

```
% Find the pixel coordinates of the upper left and lower right
% outer corners of a 2-by-2 degree gridded data set.
R = makerefmat(1, 89, 2, 2);
[UL_row, UL_col] = latlon2pix(R, 90, 0)    % Upper left
[LR_row, LR_col] = latlon2pix(R, -90, 360) % Lower right
[LL_row, LL_col] = latlon2pix(R, -90, 0)   % Lower left
% Note that in both the 2nd case and 3rd case we get a column
% value of 0.5, because the left and right edges are on the same
% meridian and (-90, 360) is the same point as (-90, 0).
```

**See Also**

`makerefmat`, `pix2latlon`, `map2pix`

# lcolorbar

---

**Purpose** Colorbar with text labels

**Syntax**  
`lcolorbar(labels)`  
`lcolorbar(labels,'property',value,...)`  
`hcb = lcolorbar(...)`

`lcolorbar(labels)` appends a colorbar with text labels. The labels input is a cell array of label strings. The colorbar is constructed using the current colormap with the label strings applied at the centers of the color bands.

`lcolorbar(labels,'property',value,...)` controls the colorbar's properties. The location of the colorbar is controlled by the `Location` property. Valid entries for `Location` are `'vertical'` (the default) or `'horizontal'`. Properties `TitleString`, `XLabelString`, `YLabelString` and `ZLabelString` set the respective strings. Property `ColorAlignment` controls whether the colorbar labels are centered on the color bands or the color breaks. Valid values for `ColorAlignment` are `'center'` and `'ends'`.

Other valid property-value pairs are any properties and values that can be applied to the title and labels of the colorbar axes.

`hcb = lcolorbar(...)` returns a handle to the colorbar axes.

**Example**

```
figure; colormap(jet(5))
labels = {'apples','oranges','grapes','peachs','melons'};
lcolorbar(labels,'fontweight','bold');
```

**See Also** `contourcmap`, `colormapeditor` (MATLAB function)

**Purpose**

Courses and distances between navigational waypoints

**Syntax**

```
[course,dist] = legs(lat,lon)
[course,dist] = legs(lat,lon,method)
[course,dist] = legs(pts) and [course,dist] = legs(pts,
    method)
mat = legs(lat,...)
```

`[course,dist] = legs(lat,lon)` returns the azimuths (*course*) and distances (*dist*) between navigational waypoints, which are specified by the column vectors *lat* and *lon*.

`[course,dist] = legs(lat,lon,method)` specifies the logic for the leg characteristics. If the string *method* is 'rh' (the default), *course* and *dist* are calculated in a rhumb line sense. If *method* is 'gc', great circle calculations are used.

`[course,dist] = legs(pts)` and `[course,dist] = legs(pts,method)` allow you to input the waypoints in a single two-column matrix, *pts*.

`mat = legs(lat,...)` packs up the outputs into a single two-column matrix, *mat*.

**Description**

This is a navigation function. All angles are in degrees, and all distances are in nautical miles. Track legs are the courses and distances traveled between navigational waypoints.

**Examples**

Imagine an airplane taking off from Logan International Airport in Boston (42.3°N,71°W) and traveling to LAX in Los Angeles (34°N,118°W). The pilot wants to file a flight plan that takes the plane over O'Hare Airport in Chicago (42°N,88°W) for a navigational update, while maintaining a constant heading on each of the two legs of the trip.

What are those headings and how long are the legs?

```
lat = [42.3; 42; 34]; long = [-71; -88; -118];
[course,dist] = legs(lat,long,'rh')
```

# legs

---

```
course =
  268.6365
  251.2724
dist =
  1.0e+003 *
  0.7569
  1.4960
```

Upon takeoff, the plane should proceed on a heading of about 269° for 756 nautical miles, then alter course to 251° for another 1495 miles.

How much farther is it traveling by not following a great circle path between waypoints? Using rhumb lines, it is traveling

```
totalrh = sum(dist)

totalrh =
  2.2530e+003
```

For a great circle route,

```
[coursegc,distgc] = legs(lat,long,'gc'); totalgc = sum(distgc)

totalgc =
  2.2451e+003
```

The great circle path is less than one-half of one percent shorter.

## See Also

`dreckon`, `gcwaypts`, `navfix`, `track`

**Purpose**

Project light objects on map axes

**Syntax**

```
h = lightm(lat,lon)
```

```
h = lightm(lat,lon,PropertyName,PropertyValue,...)
```

```
h = lightm(lat,lon,alt)
```

`h = lightm(lat,lon)` projects a light object at the coordinates `lat` and `lon`. The handle, `h`, of the object can be returned.

`h = lightm(lat,lon,PropertyName,PropertyValue,...)` allows the specification of any property name/property value pair supported by the standard MATLAB `light` function.

`h = lightm(lat,lon,alt)` allows the specification of an altitude, `alt`, for the light object. When omitted, the default is an infinite light source altitude.

**Examples**

```
load topo
axesm globe; view(120,30)
meshm(topo,topolegend); demcmap(topo)
lightm(0,90,'color','yellow')
material([.5 .5 1]); lighting phong
```



# lightm

---

## **See Also**

`light` (MATLAB function), `lightmui`



**Purpose** Determine latitude and longitude limits of regular data grid

**Syntax**

```
[latlimits,lonlimits] = limitm(Z,refvec)
limvec = limitm(Z,refvec)
```

`[latlimits,lonlimits] = limitm(Z,refvec)` returns two-element limit vectors `latlimits` and `lonlimits` describing the extremes of the input regular data grid with a three-element referencing vector `refvec`.

`latlimits` and `lonlimits` are of the form [south-limit north-limit] and [west-limit east-limit], respectively. All elements are in degrees, because this function deals only with regular data grids.

`limvec = limitm(Z,refvec)` returns a single four-element output vector of the form [south-limit north-limit west-limit east-limit].

**Examples** Using a familiar data grid,

```
load topo
[latlimits,lonlimits] = limitm(topo,topolegend
)
latlimits =
    -90    90
lonlimits =
     0   360
```

Which is expected, because `topo` covers the whole globe.

**See Also** `nanm`, `onem`, `spzerom`, `zerom`, `sizem`

# linecirc

---

**Purpose** Intersections of circles and lines in Cartesian plane

**Syntax** `[xout,yout] = linecirc(slope,intercpt,centerx,centery,radius)`  
`[xout,yout] = linecirc(slope,intercpt,centerx,centery,radius)` finds the points of intersection given a circle defined by a center and radius in  $x$ - $y$  coordinates, and a line defined by slope and  $y$ -intercept, or a slope of “inf” and an  $x$ -intercept. Two points are returned. When the objects do not intersect, NaNs are returned.

When the line is tangent to the circle, two identical points are returned. All inputs must be scalars.

**See Also** `circcirc`

**Purpose**

Project line object on map axes

**Syntax**

```
h = linem(lat,lon)
h = linem(lat,lon,linetype)
h = linem(lat,lon,PropertyName,PropertyValue,...)
h = linem(lat,lon,z)
```

`h = linem(lat,lon)` displays projected line objects on the current map axes. `lat` and `lon` are the latitude and longitude coordinates, respectively, of the line object to be projected. Note that this ordering is conceptually reversed from the MATLAB line function, because the *vertical* ( $y$ ) coordinate comes first. However, the ordering latitude, then longitude, is standard geographic usage. `lat` and `lon` must be the same size and in the `AngleUnits` of the map axes. The object handle for the displayed line can be returned in `h`.

`h = linem(lat,lon,linetype)` allows the specification of the line style, where *linetype* is any string recognized by the MATLAB line function.

`h = linem(lat,lon,PropertyName,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB line function except for `XData`, `YData`, and `ZData`.

`h = linem(lat,lon,z)` displays a line object in three dimensions, where `z` is the same size as `lat` and `lon` and contains the desired altitude data. `z` is independent of `AngleUnits`. If omitted, all points are assigned a `z`-value of 0 by default.

The units of `z` are arbitrary, except when using the globe projection. In the case of globe, `z` should have the same units as the radius of the earth or semimajor axis specified in the 'geoid' (reference ellipsoid) property of the map axes. This implies that for a reference ellipsoid vector of [1 0] (a unit sphere), the units of `z` are earth radii.

**Description**

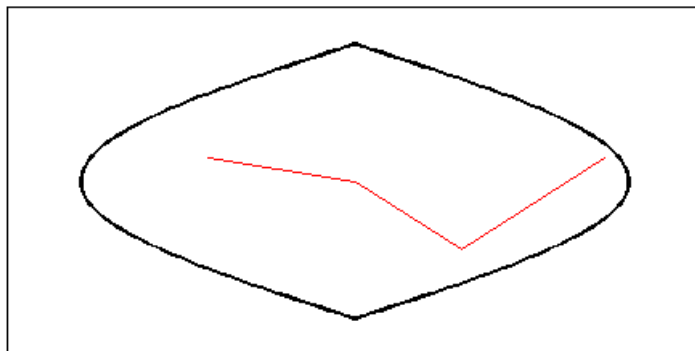
`linem` is the mapping equivalent of the MATLAB line function. It is a low-level graphics function for displaying line objects in map projections. Ordinarily, it is not used directly. Use `plotm` or `plot3m` instead.

# linem

---

## Examples

```
axesm sinusoid; framem  
linem([15; 0; -45; 15],[-100; 0; 100; 170],'r-')
```



## See Also

`line`, `plot3m`, `plotm`

**Purpose**

Line-of-sight visibility between two points in terrain

**Syntax**

```
vis = los2(Z,refvec,lat1,lon1,lat2,lon2)
vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1)
vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2)
vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2, alt1opt)
vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt,
    alt2opt)
vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2,
    alt1opt... alt2opt, actualradius)
vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2,
    alt1opt... alt2opt,actualradius,effectiveradius)
[vis,visprofile,dist,H,lattrk,lontrk] = los2(...)
los2(...)
```

LOS2 computes the mutual visibility between two points on a displayed digital elevation map. LOS2 uses the current object if it is a regular data grid, or the first regular data grid found on the current axes. The grid's zdata is used for the profile. The color data is used in the absence of data in z. The two points are selected by clicking on the map. The result is displayed in a new figure. Markers indicate visible and obscured points along the profile. The profile is shown in a Cartesian coordinate system with the origin at the observer's location. The displayed z coordinate accounts for the elevation of the terrain and the curvature of the body.

vis = los2(Z,refvec,lat1,lon1,lat2,lon2) computes the mutual visibility between pairs of points on a digital elevation map. The elevations are provided as a regular data grid Z containing elevations in units of meters. The two points are provided as vectors of latitudes and longitudes in units of degrees. The resulting logical variable vis is equal to one when the two points are visible to each other, and zero when the line of sight is obscured by terrain. If any of the input arguments are empty, los2 attempts to gather the data from the current axes. With one or more output arguments, no figures are created and only the data is returned.

vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1) places the first point at the specified altitude in meters above the surface (on a tower,

for instance). This is equivalent to putting the point on a tower. If omitted, point 1 is assumed to be on the surface. alt1 may be either a scalar or a vector with the same length as lat1, lon1, lat2, and lon2.

vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2) places both points at a specified altitudes in meters above the surface. alt2 may be either a scalar or a vector with the same length as lat1, lon1, lat2, and lon2. If alt2 is omitted, point 2 is assumed to be on the surface.

vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2, alt1opt) controls the interpretation of alt1 as either a relative altitude (alt1opt equals 'AGL', the default) or an absolute altitude (alt1opt equals 'MSL'). If the altitude option is 'AGL', alt1 is interpreted as the altitude of point 1 in meters above the terrain (“above ground level”). If alt1opt is 'MSL', alt1 is interpreted as altitude above zero (“mean sea level”).

vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt,alt2opt) controls the interpretation ALT2.

vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt... alt2opt, actualradius) does the visibility calculation on a sphere with the specified radius. If omitted, the radius of the earth in meters is assumed. The altitudes, elevations and the radius should be in the same units. This calling form is most useful for computations on bodies other than the earth.

vis = los2(Z,refvec,lat1,lon1,lat2,lon2,alt1,alt2,alt1opt... alt2opt,actualradius,effectiveradius) assumes a larger radius for propagation of the line of sight. This can account for the curvature of the signal path due to refraction in the atmosphere. For example, radio propagation in the atmosphere is commonly treated as straight line propagation on a sphere with 4/3 the radius of the earth. In that case the last two arguments would be R\_e and 4/3\*R\_e, where R\_e is the radius of the earth. Use Inf as the effective radius for flat earth visibility calculations. The altitudes, elevations and radii should be in the same units.

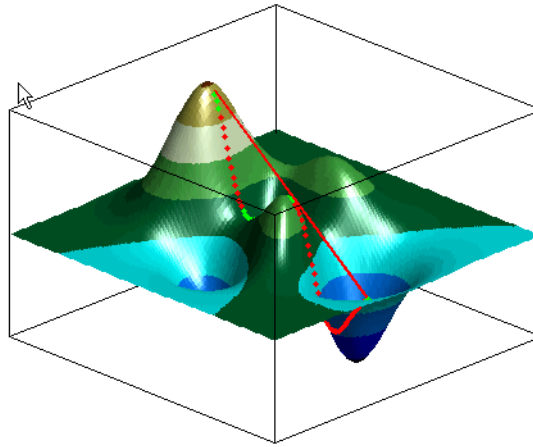
`[vis,visprofile,dist,H,lattrk,lontrk] = los2(...)`, for scalar inputs (`lat1`, `lon1`, etc.), returns vectors of points along the path between the two points. `visprofile` is a logical vector containing true (`logical(1)`) where the intermediate points are visible and false (`logical(0)`) otherwise. `dist` is the distance along the path (in meters or the units of the radius). `H` contains the terrain profile relative to the vertical datum along the path. `lattrk` and `lontrk` are the latitudes and longitudes of the points along the path. For vector inputs `los2` returns `visprofile`, `DIST`, `H`, `lattrk`, and `lontrk` as cell arrays, with one cell per element of `lat1`, `lon1`, etc.

`los2(...)`, with no output arguments, displays the visibility profile between the two points in a new figure.

### Example

```
Z = 500*peaks(100);
refvec = [1000 0 0];
[lat1, lon1, lat2, lon2] = deal(-0.027, 0.05, -0.093, 0.042);
los2(Z,refvec,lat1,lon1,lat2,lon2,100);
figure;
axesm('globe','geoid',almanac('earth','sphere','meters'))
meshm(Z, refvec, size(Z), Z); axis tight
camposm(-10,-10,1e6); camupm(0,0)
demcmap('inc', Z, 1000); shading interp; camlight

[vis,visprofile,dist,h,lattrk,lontrk] = ...
los2(Z,refvec,lat1,lon1,lat2,lon2,100);
plot3m(lattrk([1;end]),lontrk([1; end]),...
h([1; end])+[100; 0],'r','linewidth',2)
plotm(lattrk(~visprofile),lontrk(~visprofile),...
h(~visprofile),'r.','markersize',10)
plotm(lattrk(visprofile),lontrk(visprofile),...
h(visprofile),'g.','markersize',10)
```



**See Also**

`viewshed`, `mapprofile`



**Purpose**

Extract data grid values for specified locations

**Syntax**

```
value = 1t1n2val(Z,refvec,lat,lon)
value = 1t1n2val(Z,refvec,lat,lon,method)
```

`value = 1t1n2val(Z,refvec,lat,lon)` returns the values of the regular data grid `Z` corresponding to the locations specified by the vectors `lat` and `lon`.

`value = 1t1n2val(Z,refvec,lat,lon,method)` specifies the method for determining the returned value. The default *method* is 'nearest', which returns the unaltered value of the cell containing the coordinates `lat` and `lon`. Using a *method* of 'linear' or 'cubic' results in values that are linearly and cubically interpolated between cells, respectively. The function returns NaN for any `lat` or `lon` input that contains NaN or lies outside the grid limits. NaN is also returned if the grid cell occupied by any (`lat`, `lon`) contains NaN.

**Examples**

Find the elevations in `topo` associated with three European cities—Milan, Bern, and Prague (topo elevations are in meters):

```
load topo
% The city locations, [Milan Bern Prague]
lats = [45.45; 46.95; 50.1];
longs = [9.2; 7.4; 14.45];
elevations = 1t1n2val(topo,topolegend,lats,longs)

elevations =
    313
    1660
    297
```

**See Also**

`findm`

# lv2ecef

---

**Purpose**

Convert local vertical to geocentric (ECEF) coordinates

**Syntax**

```
[x, y, z] = lv2ecef(xl, yl, zl, phi0, lambda0, h0, ellipsoid)
```

[x, y, z] = lv2ecef(xl, yl, zl, phi0, lambda0, h0, ellipsoid) converts point locations specified by the coordinate arrays xl, yl, and zl relative to the local vertical system with its origin at geodetic latitude phi0, longitude lambda0, and ellipsoidal height h0. xl, yl, and zl may be arrays of any shape, as long as they are all be the same size. phi0, lambda0, and h0 must be scalars. ellipsoid is a row vector with the form [semimajor axis, eccentricity]. xl, yl, zl, and h0 must have the same length units as the semimajor axis. phi0 and lambda0 must be in radians. The coordinates x, y, and z are in the geocentric system, with the same units as the semimajor axis.

**Remarks**

For a definition of the local vertical system, also known as east-north-up (ENU), see the help for ecef2lv. For a definition of the geocentric system, also known as earth-centered, earth-fixed, see the help for geodetic2ecef.

**See Also**

ecef2geodetic, ecef2lv, elevation, geodetic2ecef

**Purpose** Semimajor axis of ellipse given semiminor axis and eccentricity

**Syntax**

```
semimajor = majaxis(semiminor, eccentricity)  
semimajor = majaxis([semiminor eccentricity])
```

`semimajor = majaxis(semiminor, eccentricity)` returns the semimajor axis length corresponding to the input semiminor axis and eccentricity.

`semimajor = majaxis([semiminor eccentricity])` allows the inputs to be packed into a single two-column input of the form `[semiminor eccentricity]`.

**Description**

The semimajor axis, the first element of the standard ellipsoid vector in Mapping Toolbox, can be determined given both the semiminor axis and the eccentricity.

**Examples**

Using the default values for the Earth,

```
semimajor = majaxis(6356.7523, 0.0818192)  
semimajor =  
    6.3781e+03
```

This is the default semimajor axis.

**See Also**

`almanac`, `axes2ecc`, `minaxis`

# makeattribspec

---

**Purpose** Construct attribute specification from geostruct

**Syntax** `attribspec = makeattribspec(S)`

**Description** `attribspec = makeattribspec(S)` analyzes a geographic data structure `S` and constructs an attribute specification suitable for use with `kmlwrite`. `kmlwrite`, given `geostruct` input, constructs an HTML table that consists of a label for the attribute in the first column and the string value of the attribute in the second column. You can modify `attribspec`, and then pass it to `kmlwrite` to exert control over which `geostruct` attribute fields are written to the HTML table and the format of the string conversion.

`attribspec` is a scalar MATLAB structure with two levels. The top level consists of a field for each attribute in `S`. Each of these fields, on the next level, contains a scalar structure with a fixed pair of fields:

`AttributeLabel` A string that corresponds to the name of the attribute field in the `geostruct`. With `kmlwrite`, the string is used to label the attribute in the first column of the HTML table. The string may be modified prior to calling `kmlwrite`. You might modify an attribute label, for example, because you want to use spaces in your HTML table, but the attribute field names in `S` must be valid MATLAB variable names and cannot have spaces themselves.

`Format` The `sprintf` format character string that converts the attribute value to a string.

**Example** In this example, a shapefile containing line data (road segments for Concord, Massachusetts) is first processed to remove all but one occurrence of each road, and then unprojected from map coordinates (in meters) into geographic coordinates (degrees of latitude and longitude). You obtain projection parameters from a GeoTIFF image that uses the

same coordinate system, except that its units are in survey feet instead of in meters.

Begin by importing a shapefile representing a small network of road segments and noting its attributes:

```
roads = shaperead('concord_roads')

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

Change its Geometry type to 'Point' and eliminate all but one instance of each street in the geostruct:

```
[roads.Geometry] = deal('Point');
[U, i] = unique({roads.STREETNAME});
roads = roads(i);
```

The coordinates of the roads are in projected map coordinates, and must be unprojected from meters to latitude-longitude. Obtain projection parameters from a related GeoTIFF file, which enables a mapping between latitude-longitude and map coordinates in U.S. survey feet:

```
proj = geotiffinfo('boston.tif');
% Loop through roads, assigning Lat and Lon fields.
feetPerMeter = unitsratio('survey foot','meter');
for k=1:numel(roads)
    % Take the first point on each road and convert its
    % coordinates from meters to survey feet to make them
```

# makeattribspec

---

```
% compatible with proj.  
x = feetPerMeter * roads(k).X(1);  
y = feetPerMeter * roads(k).Y(1);  
[roads(k).Lat, roads(k).Lon] = projinv(proj, x, y);  
end
```

Create the attribute specification for geostruct roads:

```
attribspec = makeattribspec(roads)  
  
attribspec =  
    STREETNAME: [1x1 struct]  
    RT_NUMBER: [1x1 struct]  
    CLASS: [1x1 struct]  
    ADMIN_TYPE: [1x1 struct]  
    LENGTH: [1x1 struct]
```

Note that `makeattribspec` automatically excludes the `Geometry`, `BoundingBox`, `X`, `Y`, `Lat`, and `Lon` fields of the geostruct. Examine the fields of this default attribute specification:

```
attribspec.STREETNAME  
  
ans =  
    AttributeLabel: 'STREETNAME'  
           Format: '%s'  
  
attribspec.RT_NUMBER  
  
ans =  
    AttributeLabel: 'RT_NUMBER'  
           Format: '%s'  
  
attribspec.CLASS  
  
ans =  
    AttributeLabel: 'CLASS'
```

```

                Format: '%.15g'

attribspec.ADMIN_TYPE

ans =
    AttributeLabel: 'ADMIN_TYPE'
                Format: '%.15g'

attribspec.LENGTH

ans =
    AttributeLabel: 'LENGTH'
                Format: '%.15g'

```

Modify the attribute spec to

- 1** Eliminate the 'ADMIN\_TYPE' attribute.
- 2** Rename the 'STREETNAME' field to 'Street Name'.
- 3** Rename the 'RT\_NUMBER' field to 'Route Number'.
- 4** Rename all other fields with first letter upper case only.
- 5** Highlight each attribute label with a bold font.
- 6** Reduce the number of decimal places used to display road lengths.
- 7** Add Meters to the format specifier:

```

attribspec = rmfield(attribspec, 'ADMIN_TYPE');
attribspec.STREETNAME.AttributeLabel = '<b>Street Name</b>';
attribspec.RT_NUMBER.AttributeLabel = '<b>Route Number</b>';
attribspec.CLASS.AttributeLabel = '<b>Class</b>';
attribspec.LENGTH.AttributeLabel = '<b>Length</b>';
attribspec.LENGTH.Format = '%.2f Meters';

```

Inspect the contents of the modified attribute specification:

# makeattribspec

---

```
attribspec =
    STREETNAME: [1x1 struct]
    RT_NUMBER: [1x1 struct]
    CLASS: [1x1 struct]
    LENGTH: [1x1 struct]

attribspec.STREETNAME

ans =
    AttributeLabel: '<b>Street Name</b>'
    Format: '%s'

attribspec.RT_NUMBER

ans =
    AttributeLabel: '<b>Route Number</b>'
    Format: '%s'

attribspec.CLASS

ans =
    AttributeLabel: '<b>Class</b>'
    Format: '%.15g'

attribspec.LENGTH

ans =
    AttributeLabel: '<b>Length</b>'
    Format: '%.2f meters'
```

Export the road network to a KML file:

```
filename = 'concord_roads.kml';
kmlwrite(filename, roads, 'Description', ...
    attribspec, 'Name', {roads.STREETNAME})
```

## See also

kmlwrite, makedbfspec, shapewrite



**Purpose** Construct default DBF specification from geostruct

**Syntax** `dbfspec = makedbfspec(S)`

`dbfspec = makedbfspec(S)` analyzes a geographic data structure `S` (a `geostruct2`) and constructs a DBF specification suitable for use with `shapewrite`. You can modify `dbfspec`, then pass it to `shapewrite` to exert control over which `geostruct` attribute fields are written to the DBF component of the shapefile, the field-widths, and the precision used for numerical values.

`dbfspec` is a scalar MATLAB structure with two levels. The top level consists of a field for each attribute in `S`. Each of these fields, in turn, contains a scalar structure with a fixed set of four fields:

<b>dbfspec field</b>	<b>Contents</b>
FieldName	The field name to be used within the DBF file. This will be identical to the name of the corresponding attribute, but may be modified prior to calling <code>shapewrite</code> . This might be necessary, for example, because you want to use spaces in your DBF field names, but the attribute fieldnames in <code>S</code> must be valid MATLAB variable names and cannot have spaces themselves.
<i>FieldType</i>	The field type to be used in the file, either 'N' (numeric) or 'C' (character).
FieldLength	The number of bytes that each instance of the field will occupy in the file.
FieldDecimalCount	The number of digits to the right of the decimal place that are kept in a numeric field. Zero for integer-valued fields and character fields. The default value for noninteger numeric fields is 6.

**Example** Import a shapefile representing a small network of road segments, and construct a DBF specification.

# makedbfspec

---

```
s = shaperead('concord_roads')

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

dbfspec = makedbfspec(s)

dbfspec =
    STREETNAME: [1x1 struct]
    RT_NUMBER: [1x1 struct]
    CLASS: [1x1 struct]
    ADMIN_TYPE: [1x1 struct]
    LENGTH: [1x1 struct]
```

Modify the DBF spec to (a) eliminate the 'ADMIN\_TYPE' attribute, (b) rename the 'STREETNAME' field to 'Street Name', and (c) reduce the number of decimal places used to store road lengths.

```
dbfspec = rmfield(dbfspec, 'ADMIN_TYPE')

dbfspec =
    STREETNAME: [1x1 struct]
    RT_NUMBER: [1x1 struct]
    CLASS: [1x1 struct]
    LENGTH: [1x1 struct]

dbfspec.STREETNAME.FieldName = 'Street Name';
dbfspec.LENGTH.FieldDecimalCount = 1;
```

Export the road network back to a modified shapefile. (Actually, only the DBF component will be different.)

```
shapewrite(s, 'concord_roads_modified', 'DbfSpec', dbfspec)
```

Verify the changes you made. Notice the appearance of 'Street Name' in the field names reported by shapeinfo, the absence of the 'ADMIN\_TYPE' field, and the reduction in the precision of the road lengths.

```
info = shapeinfo('concord_roads_modified')
info =
    Filename: [3x28 char]
    ShapeType: 'PolyLine'
    BoundingBox: [2x2 double]
    NumFeatures: 609
    Attributes: [4x1 struct]

{info.Attributes.Name}

ans =
    'Street Name'    'RT_NUMBER'    'CLASS'    'LENGTH'

r = shaperead('concord_roads_modified')

r =
609x1 struct array with fields:
    Geometry
    BoundingBox
    X
    Y
    StreetName
    RT_NUMBER
    CLASS
    LENGTH

s(33).LENGTH
```

# makedbfspec

---

```
ans =  
    3.492817400000000e+002
```

```
r(33).LENGTH
```

```
ans =  
    3.493000000000000e+002
```

**See also**      `shapeinfo`, `shapewrite`

**Purpose** Convert ordinary graphics object to mapped object

**Syntax** `makemapped(h)`

`makemapped(h)` modifies the graphic object(s) associated with `h` such that upon subsequent modification of map axes properties, they are automatically reprojected appropriately. The object's coordinates are not changed by `makemapped`, but will change should you modify the map projection. `h` can be a handle, vector of handles, or any name string recognized by `handlem`. The objects are then considered to be geographic data. You should first trim objects extending outside the map frame to the map frame using `trimcart`.

**Example**

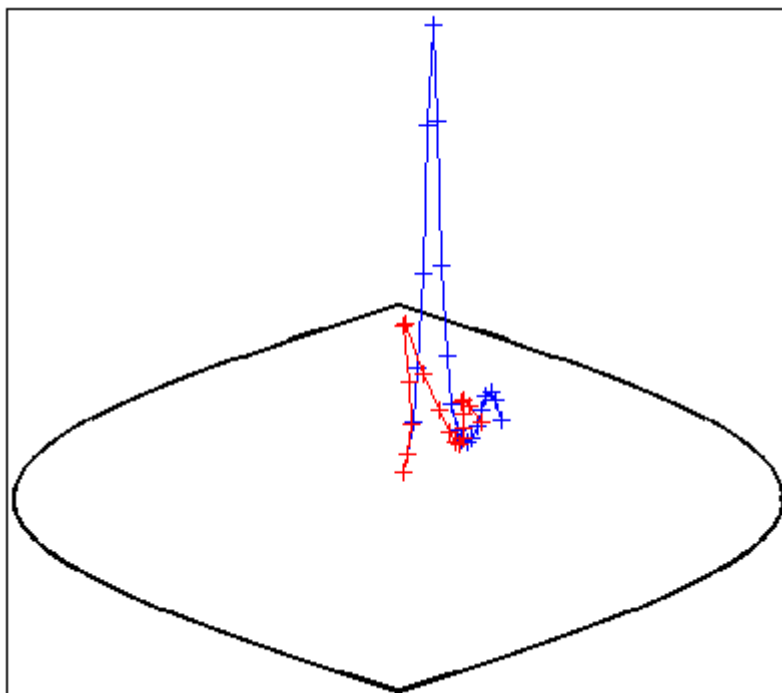
```
axesm('miller','geoid',[25 0])
framem
plot(humps,'b+-')

h = plot(humps,'r+-');
trimcart(h)
makemapped(h)

setm(gca,'MapProjection','sinusoid')
```

# makemapped

---



## Remarks

Objects should first be trimmed to the map frame using `trimcart`. This avoids problems in taking inverse map projections with out-of-range data.

## See Also

`trimcart`, `handlem`, `cart2grn`

**Purpose** Construct affine spatial-referencing matrix

**Syntax** `R = makereformat(x11, y11, dx, dy)`  
`R = makereformat(x11, y11, dx, dy)`

**Description** `R = makereformat(x11, y11, dx, dy)` with scalars `dx` and `dy` constructs a referencing matrix that aligns image/data grid rows to map  $x$  and columns to map  $y$ . `x11` and `y11` are scalars that specify the map location of the center of the first (1,1) pixel in the image or first element of the data grid, so that

$$[x11 \ y11] = \text{pix2map}(R, 1, 1)$$

`dx` is the difference in  $x$  (or longitude) between pixels in successive columns and `dy` is the difference in  $y$  (or latitude) between pixels in successive rows. More abstractly, `R` is defined such that

$$[x11 + (\text{col}-1) * dx, y11 + (\text{row}-1) * dy] = \text{pix2map}(R, \text{row}, \text{col})$$

Pixels cover squares on the map when  $\text{abs}(dx) = \text{abs}(dy)$ . To achieve the most typical kind of alignment, where  $x$  increases from column to column and  $y$  decreases from row to row, make `dx` positive and `dy` negative. In order to specify such an alignment along with square pixels, make `dx` positive and make `dy` equal to `-dx`:

$$R = \text{makereformat}(x11, y11, dx, -dx)$$

`R = makereformat(x11, y11, dx, dy)` with two-element vectors `dx` and `dy` constructs the most general possible kind of referencing matrix, for which

$$[x11 + ([\text{row} \ \text{col}]-1) * dx(:), y11 + ([\text{row} \ \text{col}]-1) * dy(:)] \dots \\ = \text{pix2map}(R, \text{row}, \text{col})$$

A spatial referencing matrix `R` ties the row and column subscripts of an image or regular data grid to 2-D map coordinates or to geographic coordinates (longitude and geodetic latitude). `R` is a 3-by-2 affine

transformation matrix. R either transforms pixel subscripts (row, column) to/from map coordinates (x,y) according to

$$[x \ y] = [\text{row} \ \text{col} \ 1] * R$$

or transforms pixel subscripts to/from geographic coordinates according to

$$[\text{lon} \ \text{lat}] = [\text{row} \ \text{col} \ 1] * R$$

To construct a referencing matrix for use with geographic coordinates, use longitude in place of X and latitude in place of Y, as shown in the third syntax below. This is one of the few places where longitude precedes latitude in a function call.

## Remarks

In this general case, each pixel can become a parallelogram on the map, with neither edge necessarily aligned to map  $x$  or  $y$ . The vector  $[dx(1) \ dy(1)]$  is the difference in map location between a pixel in one row and its neighbor in the preceding row. Likewise,  $[dx(2) \ dy(2)]$  is the difference in map location between a pixel in one column and its neighbor in the preceding column.

To specify pixels that are rectangular or square (but possibly rotated), choose  $dx$  and  $dy$  such that  $\text{prod}(dx) + \text{prod}(dy) = 0$ . To specify square (but possibly rotated) pixels, choose  $dx$  and  $dy$  such that the 2-by-2 matrix  $[dx(:) \ dy(:)]$  is a scalar multiple of an orthogonal matrix (that is, its two eigenvalues are real, nonzero, and equal in absolute value). This amounts to either rotation, a mirror image, or a combination of both. Note that for scalars  $dx$  and  $dy$ ,

$$R = \text{makereformat}(x11, y11, [0 \ dx], [dy \ 0])$$

is equivalent to

$$R = \text{makereformat}(x11, y11, dx, dy)$$

$R = \text{makereformat}(\text{lon}11, \text{lat}11, d\text{lon}, d\text{lat})$ , with longitude preceding latitude, constructs a referencing matrix for use with geographic coordinates. In this case,



```
[lat11,lon11] = pix2latlon(R,1,1),
[lat11+(row-1)*dlat,lon11+(col-1)*dlon] = pix2latlon(R,row,col)
```

for scalar dlat and dlon, and

```
[lat11+[row col]-1)*dlat,lon11+([row col]-1)*dlon] = ...
pix2latlon(R, row,col)
```

for vector dlat and dlon.

Images or data grids aligned with latitude and longitude might already have referencing vectors. In this case you can use function `refvec2mat` to convert to a referencing matrix.

## Examples

### Example 1

```
% Create a referencing matrix for an image with square,
% four-meter pixels and with its upper left corner (in a map
% coordinate system) at x = 207000 meters, y = 913000
% meters. The image follows the typical orientation:
% x increasing from column to column and y decreasing
% from row to row.
```

```
x11 = 207002; % Two meters east of the upper left corner
y11 = 912998; % Two meters south of the upper left corner
dx = 4;
dy = -4;
R = makerefmat(x11, y11, dx, dy)
```

### Example 2

```
% Create a referencing matrix for a global geoid grid.
```

```
load geoid % Adds array 'geoid' to the workspace
```

```
% 'geoid' contains a model of the Earth's geoid sampled in
% one-degree-by-one-degree cells. Each column of 'geoid'
% contains geoid heights in meters for 180 cells starting at
% latitude -90 degrees and extending to +90 degrees, for a
```

```
% given latitude.
% Each row contains geoid heights for 360 cells starting at
% longitude 0 and extending 360 degrees.

lat11 = -89.5; % Cell-center latitude corresponding to geoid(1,1)
lon11 = 0.5; % Cell-center longitude corresponding to
geoid(1,1)
dLat = 1; % From row to row moving north by one degree
dLon = 1; % From column to column moving east by one degree
geoidR = makerefmat(lon11, lat11, dLon, dLat)

% It's well known that at its most extreme the geoid reaches
% a minimum of slightly less than -100 meters, and that the
% minimum occurs in the Indian Ocean at approximately
% 4.5 degrees latitude, 78.5 degrees longitude. Check the
% geoid height at this location by using LATLON2PIX with
% the new referencing matrix:

[row, col] = latlon2pix(geoidR, 4.5, 78.5)
geoid(round(row),round(col))
```

### Example 3

```
% Create a half-resolution version of a georeferenced TIFF
% image, using Image Processing Toolbox functions IND2GRAY
% and IMRESIZE.

% Read the indexed-color TIFF image and convert it to grayscale.
% The size of the image is 2000-by-2000.
[X, cmap] = imread('1_209910_sub.tif');
I_orig = ind2gray(X, cmap);

% Read the corresponding worldfile. Each image pixel covers a
% one-meter square on the map.
R_orig = worldfileread('1_209910_sub.tfw')

% Halve the resolution, creating a smaller (1000-by-1000) image.
I_half = imresize(I_orig, size(I_orig)/2, 'bicubic');
```

```
% Find the map coordinates of the center of pixel (1,1) in the
% resized image: halfway between the centers of pixels (1,1) and
% (2,2) in the original image.
[x11_orig, y11_orig] = pix2map(R_orig, 1, 1)
[x22_orig, y22_orig] = pix2map(R_orig, 2, 2)

% Average these to determine the center of pixel (1,1) in the new
% image.
x11_half = (x11_orig + x22_orig) / 2
y11_half = (y11_orig + y22_orig) / 2

% Make a referencing matrix for the new image, noting that its
% pixels are each two meters square.
R_half = makereformat(x11_half, y11_half, 2, -2)

% Display each image in map coordinates.
figure;
subplot(2,1,1); h1 = mapshow(I_orig,R_orig); ax1 =
get(h1,'Parent');
subplot(2,1,2); h2 = mapshow(I_half,R_half); ax2 =
get(h2,'Parent');
set(ax1, 'XLim', [208000 208250], 'YLim', [911800 911950])
set(ax2, 'XLim', [208000 208250], 'YLim', [911800 911950])

% Mark the same map location on top of each image.
x = 208202.21;
y = 911862.70;
line(x, y, 'Parent', ax1, 'Marker', '+', 'MarkerEdgeColor', 'r');
line(x, y, 'Parent', ax2, 'Marker', '+', 'MarkerEdgeColor', 'r');

% Graphically, they coincide, even though the same map location
% corresponds to two different pixel coordinates.
[row1, col1] = map2pix(R_orig, x, y)
[row2, col2] = map2pix(R_half, x, y)
```

# makerefmat

---

## **See Also**

latlon2pix, map2pix, pix2latlon, pix2map, refvec2mat,  
worldfileread, worldfilewrite

## Purpose

Construct vector layer symbolization specification

## Syntax

```
symbolspec = makesymbolspec(geometry,rule1,rule2,...ruleN)
```

`symbolspec = makesymbolspec(geometry,rule1,rule2,...ruleN)` constructs a symbol specification structure (`symbolspec`) for symbolizing a (vector) shape layer in the Map Viewer or when using `mapshow`. `geometry` is one of 'Point', 'Line', 'PolyLine', 'Polygon', or 'Patch'. Rules, defined in detail below, specify the graphics properties for each feature of the layer. A rule can be a default rule that is applied to all features in the layer or it may limit the symbolization to only those features that have a particular value for a specified attribute. Features that do not match any rules are displayed using the default graphics properties.

To create a rule that applies to all features, a default rule, use the following syntax:

```
{'Default',Property1,Value1,Property2,Value2,...
    PropertyN,ValueN}
```

To create a rule that applies only to features that have a particular value or range of values for a specified attribute, use the following syntax:

```
{AttributeName,AttributeValue,
    Property1,Value1,Property2,Value2,...,PropertyN,ValueN}
```

`AttributeValue` and `ValueN` can each be a two-element vector, [`low high`], specifying a range. If `AttributeValue` is a range, `ValueN` might or might not be a range.

The following is a list of allowable values for `PropertyN`.

- Points or Multipoints: 'Marker', 'Color', 'MarkerEdgeColor', 'MarkerFaceColor', 'MarkerSize', and 'Visible'
- Lines or PolyLines: 'Color', 'LineStyle', 'LineWidth', and 'Visible'

# makesymbolspec

---

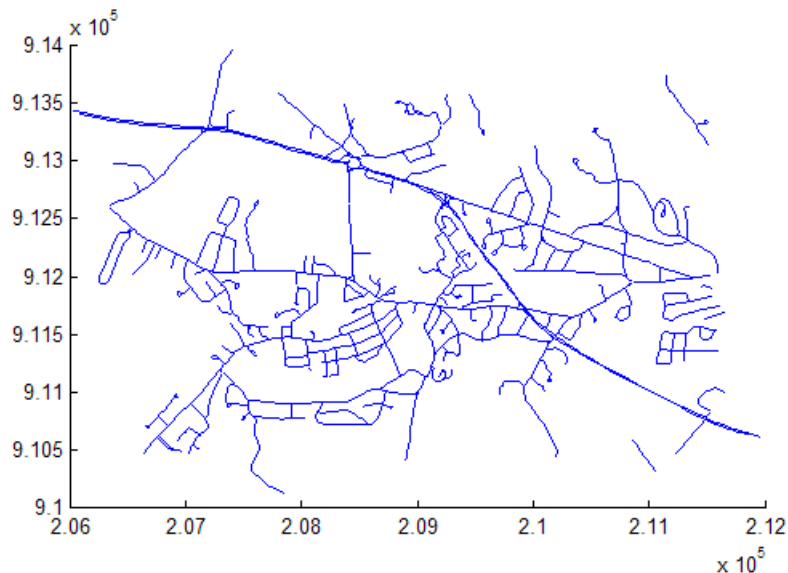
- Polygons: 'FaceColor', 'FaceAlpha', 'LineStyle', 'LineWidth', 'EdgeColor', 'EdgeAlpha', and 'Visible'

## Examples

The following examples import a shapefile containing road data and symbolize it in several ways using symbol specifications.

### Example 1 – Default Color

```
roads = shaperead('concord_roads.shp');
blueRoads = makesymbolspec('Line',{ 'Default','Color',[0 0 1]});
mapshow(roads, 'SymbolSpec',blueRoads);
```



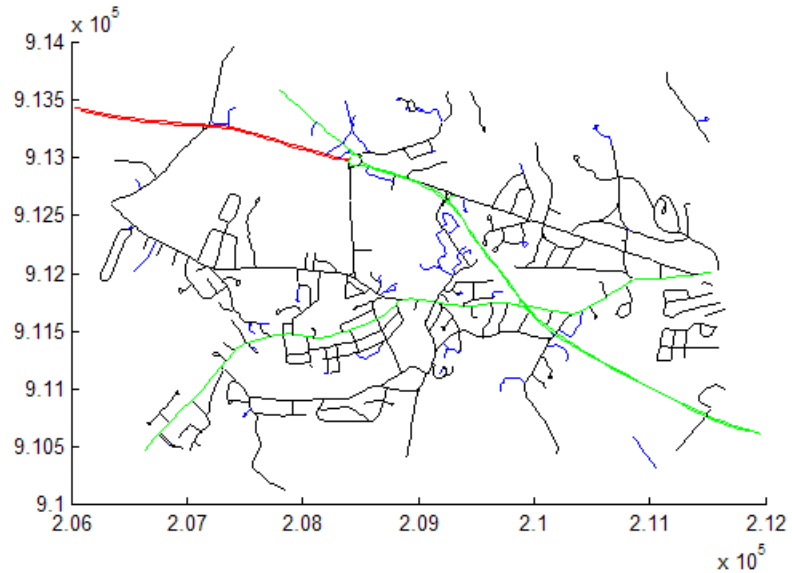
### Example 2 – Discrete Attribute Based

```
roads = shaperead('concord_roads.shp');
roadColors = ...
makesymbolspec('Line',{ 'CLASS',2,'Color','r'},...
               {'CLASS',3,'Color','g'},...
               {'CLASS',6,'Color','b'},...)
```

```

        {'Default','Color','k'});
mapshow(roads,'SymbolSpec',roadColors);

```

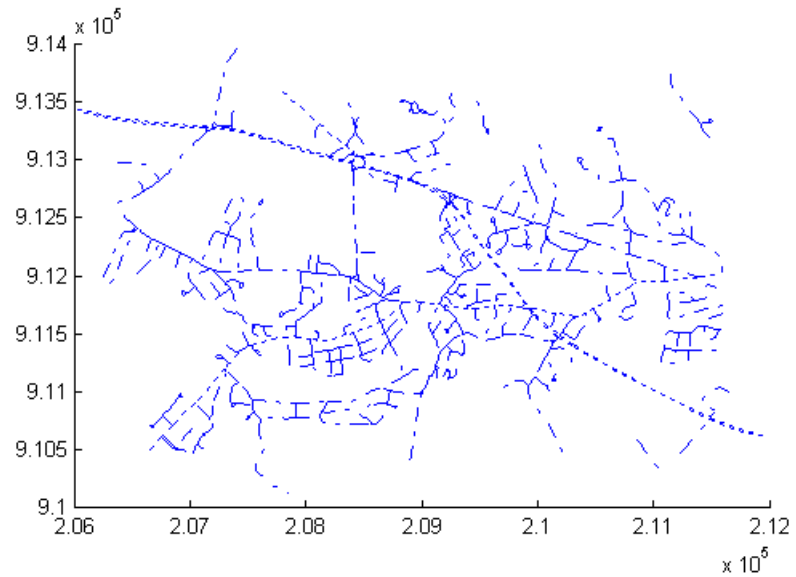


### Example 3 – Using a Range of Attribute Values

```

roads = shaperead('concord_roads.shp');
lineStyle = makesymbolspec('Line',...
    {'CLASS',[1 3], 'LineStyle',':'},...
    {'CLASS',[4 6], 'LineStyle','-.'});
mapshow(roads,'SymbolSpec',lineStyle);

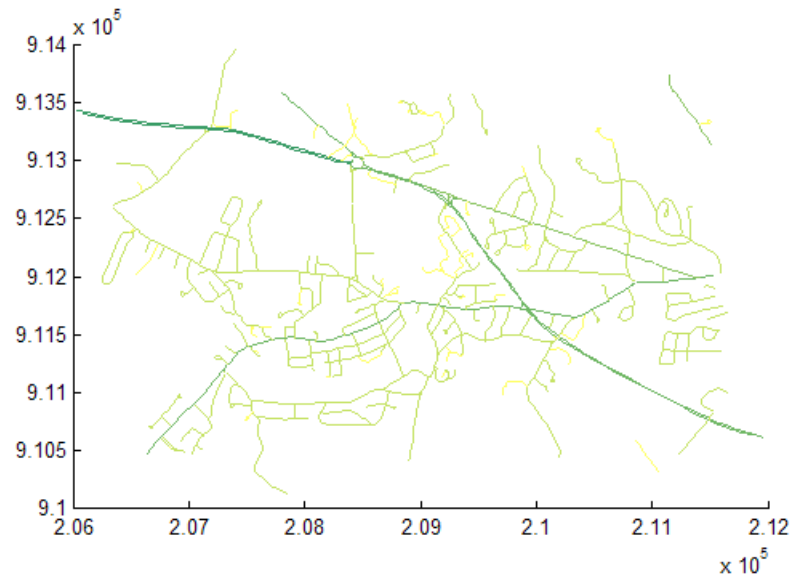
```



## Example 4 – Using a Range of Attribute Values and a Range of Property Values

```
roads = shaperead('concord_roads.shp');
colorRange = makesymbolspec('Line',...
    {'CLASS',[1 6],'Color',summer(10)});
mapshow(roads,'SymbolSpec',colorRange);
```





**See Also**

mapshow, geoshow, mapview

# map2pix

---

**Purpose** Convert map coordinates to pixel coordinates

**Syntax**

```
[row,col] = map2pix(R,x,y)
p = map2pix(R,x,y)
[...] = map2pix(R,s)
```

`[row,col] = map2pix(R,x,y)` calculates pixel coordinates `row`, `col` from map coordinates `x,y`. `R` is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to map coordinates. `x` and `y` are vectors or arrays of matching size. The outputs `row` and `col` have the same size as `x` and `y`.

`p = map2pix(R,x,y)` combines `row` and `col` into a single array `p`. If `x` and `y` are column vectors of length `n`, then `p` is an `n`-by-2 matrix and each `p(k,:)` specifies the pixel coordinates of a single point. Otherwise, `p` has size `[size(row) 2]`, and `p(k1,k2,...,kn,:)` contains the pixel coordinates of a single point.

`[...] = map2pix(R,s)` combines `x` and `y` into a single array `s`. If `x` and `y` are column vectors of length `n`, the `s` should be an `n`-by-2 matrix such that each row (`s(k,:)`) specifies the map coordinates of a single point. Otherwise, `s` should have size `[size(X) 2]`, and `s(k1,k2,...,kn,:)` should contain the map coordinates of a single point.

**Example**

```
% Find the pixel coordinates for the spatial coordinates
% (207050, 912900)
R = worldfileread('concord_ortho_w.tfw');
[r,c] = map2pix(R, 207050, 912900);
```

**See Also** `latlon2pix`, `makerefmat`, `pix2map`, `worldfileread`

**Purpose**

Compute bounding box of georeferenced image or data grid

**Syntax**

```
bbox = mapbbox(R, height, width)
```

```
bbox = mapbbox(R, sizea)
```

`bbox = mapbbox(R, height, width)` computes the 2-by-2 bounding box of a georeferenced image or regular gridded data set. `R` is a 3-by-2 affine referencing matrix. `height` and `width` are the image dimensions. `bbox` bounds the outer edges of the image in map coordinates:

```
[minX minY
maxX maxY]
```

`bbox = mapbbox(R, sizea)` accepts `sizea = [height, width, ...]` instead of `height` and `width`.

`BBOX = mapbbox(info)` accepts a scalar struct array with the fields

'RefMatrix'	3-by-2 referencing matrix
'Height'	Scalar number
'Width'	Scalar number

**See Also**

`geotiffinfo`, `makerefmat`, `mapoutline`, `pixcenters`, `pix2map`

# maplist

---

## Purpose

Map projections available in Mapping Toolbox

## Syntax

```
list = maplist  
[list,defproj] = maplist
```

`list = maplist` returns a structure that defines the map projections available in Mapping Toolbox. The list structure is `list.Name`, `list.IdString`, `list.Classification`, `list.ClassCode`. This list structure is used by the functions `maps` and `axesmui` when processing map projection identifiers during operation of the toolbox functions.

`[list,defproj] = maplist` also returns the default projection's `IdString`.

`list.Name` defines the full name of the projection. This entry is used in the command-line table display and in the Projection Control Box.

`list.IdString` defines the name of the M-file that computes the projection.

`list.Classification` defines the projection classification that is used in the command-line table display.

`list.ClassCode` defines the character string that is used to label the classes of projections in the Projection Control Box. The eight class codes are

- `Azim` — Azimuthal
- `Coni` — Conic
- `Cyln` — Cylindrical
- `Mazi` — Modified azimuthal
- `Pazi` — Pseudoazimuthal
- `Pcon` — Pseudoconic
- `Pcy` — Pseudocylindrical
- `Poly` — Polyconic

When map projections are added to the toolbox, the list structure needs to be extended. For example, if a new projection is added to the default list, then a new entry in the list structure would be

```
list.Name(61)           = 'My Projection'  
list.IdString(61)      = 'newprojection';  
list.Classification(61) = 'New Projection Type';  
list.ClassCode(61)     = 'Code';
```

**See Also**      maps, axesmui

# mapoutline

---

## Purpose

Compute outline of georeferenced image or data grid

## Syntax

```
[x,y] = mapoutline(R, height, width)
```

```
[x,y] = mapoutline(R, sizea)
```

```
[x,y] = mapoutline(info)
```

```
[x,y] = mapoutline(...,'close')
```

```
[lon,lat] = mapoutline(R,...)
```

```
outline = mapoutline(...)
```

`[x,y] = mapoutline(R, height, width)` computes the outline of a georeferenced image or regular gridded data set in map coordinates. `R` is a 3-by-2 affine referencing matrix. `height` and `width` are the image dimensions. `x` and `y` are 4-by-1 column vectors containing the map coordinates of the outer corners of the corner pixels, in the following order:

```
(1,1), (height,1), (height, width), (1, width).
```

`[x,y] = mapoutline(R, sizea)` accepts `SIZEA = [height, width, ...]` instead of `height` and `width`.

`[x,y] = mapoutline(info)` accepts a scalar struct array with the fields

'RefMatrix'	3-by-2 referencing matrix
'Height'	Scalar number
'Width'	Scalar number

`[x,y] = mapoutline(...,'close')` returns `x` and `y` as 5-by-1 vectors, appending the coordinates of the first of the four corners to the end.

`[lon,lat] = mapoutline(R,...)`, where `R` georeferences pixels to longitude and latitude rather than map coordinates, returns the outline in geographic coordinates. Longitude must precede latitude in the output argument list.

`outline = mapoutline(...)` returns the corner coordinates in a 4-by-2 or 5-by-2 array.

**Example**

Draw a red outline delineating the Boston GeoTIFF image, which is referenced to the Massachusetts Mainland State Plane coordinate system with units of survey feet.

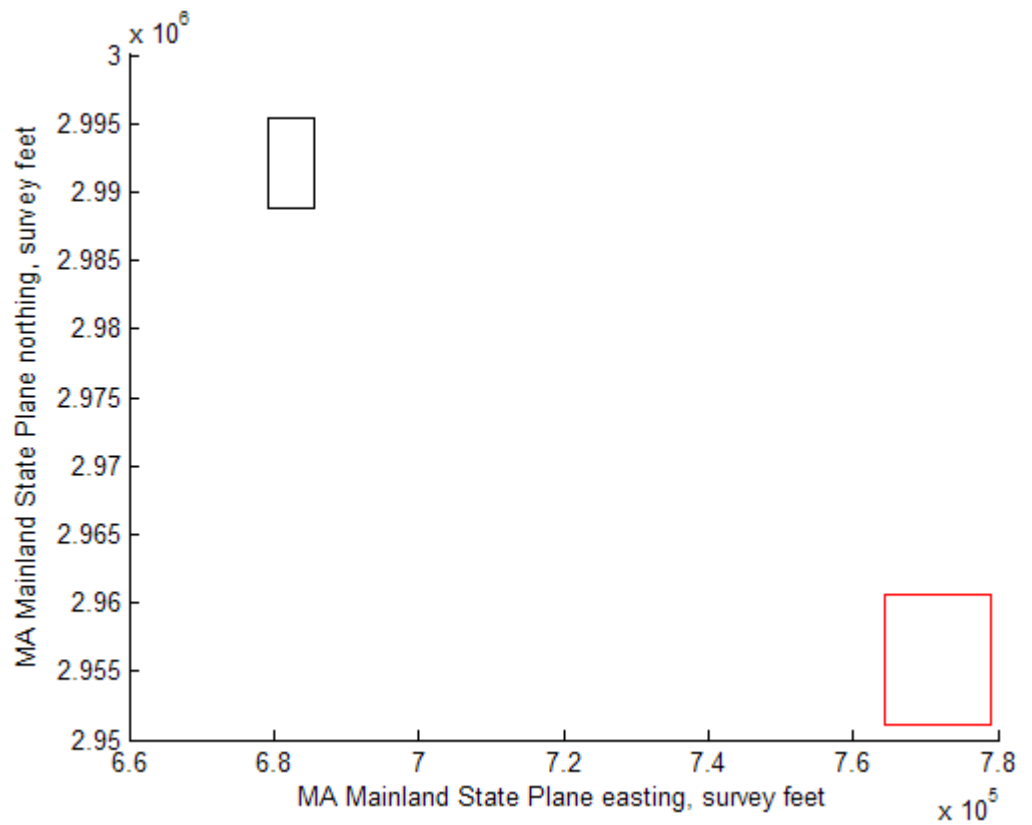
```
figure
info = geotiffinfo('boston.tif');
[x,y] = mapoutline(info, 'close');
hold on
plot(x,y,'r')
xlabel('MA Mainland State Plane easting, survey feet')
ylabel('MA Mainland State Plane northing, survey feet')
```

Draw a black outline delineating a TIFF image of Concord, Massachusetts, while lies roughly 25 km north west of Boston. Convert world file units to survey feet from meters to be consistent with the Boston image.

```
info = imfinfo('concord_ortho_w.tif');
R = worldfileread('concord_ortho_w.tfw');
R = R * unitsratio('sf','meter');
[x,y] = mapoutline(R, info.Height, info.Width, 'close');
plot(x,y,'k')
```

# mapoutline

---



## See Also

`makereformat`, `mapbbox`, `pixcenters`, `pix2map`



## Purpose

Interpolate heights between waypoints on regular data grid

## Syntax

```
[z,rng,lat,lon] = mapprofile
[z,rng,lat,lon] = mapprofile(Z,refvec,lat,lon)
[z,rng,lat,lon] = mapprofile(Z,refvec,lat,lon,rngunits)
[z,rng,lat,lon] = mapprofile(Z,refvec,lat,lon,ellipsoid)
[z,rng,lat,lon] = ... mapprofile(Z,refvec,lat,lon,rngunits,
    trackmethod,interpmethod)
[z,rng,lat,lon] = ... mapprofile(Z,refvec,lat,lon,ellipsoid,
    trackmethod,interpmethod)
```

mapprofile computes or plots a profile of values between waypoints on a displayed regular data grid. mapprofile uses the current object if it is a regular data grid, or the first regular data grid found on the current axes. The map's ZData is used for the profile. The color data is used in the absence of data in z. The result is displayed in a new 3-D figure. When no input arguments are given, cross hairs appear on the map in the current figure. Digitize a transect with as many waypoints as you need, and after the final one press **Enter** to complete the command.

[z,rng,lat,lon] = mapprofile returns the values of the profile without displaying them. The output z contains interpolated values from ZData along great circles between the waypoints. rng is a vector of associated distances from the first waypoint in units of degrees of arc along the surface. lat and lon are the corresponding latitudes and longitudes. This too is an interactive version of mapprofile, in which you digitize a transect on the current map.

[z,rng,lat,lon] = mapprofile(Z,refvec,lat,lon) takes as input a regular data grid, Z, a three-element referencing vector, refvec, and waypoint vectors. No displayed map is required. Sets of waypoints can be separated by NaNs into line sequences. The output ranges are measured from the first waypoint within a sequence.

[z,rng,lat,lon] = mapprofile(Z,refvec,lat,lon,rngunits) specifies the units of the output ranges along the profile. Valid range units inputs are any distance string recognized by unitsratio. Surface distances are computed using the default radius of the grid. If omitted, 'degrees' is assumed.

# mapprofile

---

`[z,rng,lat,lon] = mapprofile(Z,refvec,lat,lon,ellipsoid)` uses the provided ellipsoid definition in computing the range along the profile. The ellipsoid vector is of the form [semimajor axes, eccentricity]. The output range is reported in the same distance units as the semimajor axes of the ellipsoid vector. If omitted, the range vector is for a sphere.

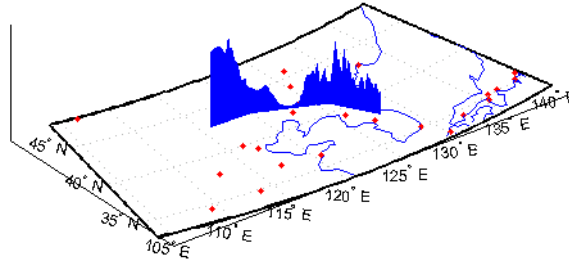
`[z,rng,lat,lon] = ...`  
`mapprofile(Z,refvec,lat,lon,rngunits,trackmethod,interpmethod)`  
and `[z,rng,lat,lon] = ...`  
`mapprofile(Z,refvec,lat,lon,ellipsoid,trackmethod,interpmethod)`  
control the interpolation methods used. Valid trackmethods are 'gc' for great circle tracks between waypoints, and 'rh' for rhumb lines. Valid interpmethods for interpolation within the data grid are 'bilinear' for linear interpolation, 'bicubic' for cubic interpolation, and 'nearest' for nearest neighbor interpolation. If omitted, 'gc' and 'bilinear' are assumed.

## Examples

### Example 1

Create a map axes for the Korean peninsula. Specify an elevation profile across the sample Korean digital elevation data and plot it, combined with a coastline and city markers:

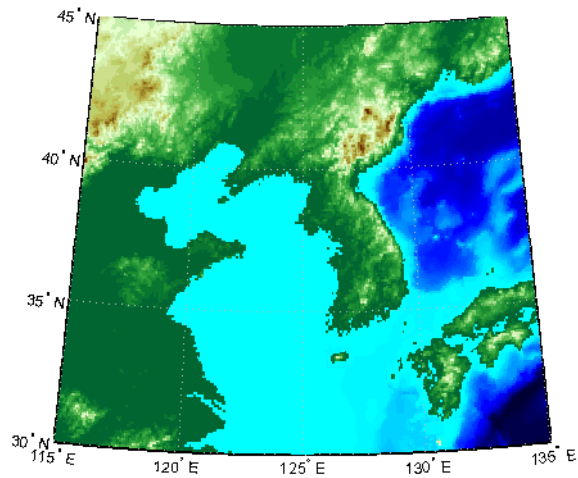
```
load korea
h = worldmap(map, refvec); % The figure has no map content
plat = [ 43  43  41  38];
plon = [116 120 126 128];
mapprofile(map, refvec, plat, plon)
load coast
plotm(lat, long)
geoshow('worldcities.shp', 'Marker', '.', 'Color', 'red')
```



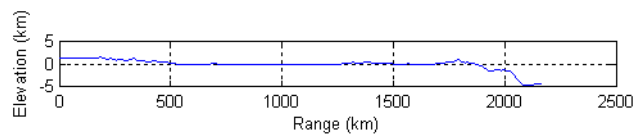
When you select more than two waypoints, the automatically generated figure displays the result in three dimensions. The following example shows the relative sizes of the mountains in northern China compared to the depths of the Sea of Japan. The call to `mapprofile` without input arguments requires you to interactively pick waypoints on the figure using the mouse, and press **Enter** after you select the final point:

```
axes(h);
meshm(map, refvec, size(map))
demcmap(map)
[z,rng,lat,lon] = mapprofile;
```

Adding output arguments suppresses the display of the results in a new figure. You can then use the results in further calculations or display the results yourself. Here the profile from the upper left to lower right is computed from waypoints interactively picked on the map (your profile will not be identical to what is shown below). The example converts ranges and elevations to kilometers and displays them in a new figure, setting the vertical exaggeration factor to 20. With no vertical exaggeration, the changes in elevation would be almost too small to see.



```
figure
plot(deg2km(rng),z/1000)
daspect([ 1 1/20 1 ]);
xlabel 'Range (km)'
ylabel 'Elevation (km)'
```



Naturally, the profile you get depends on the transect locations you pick.

## Example 2

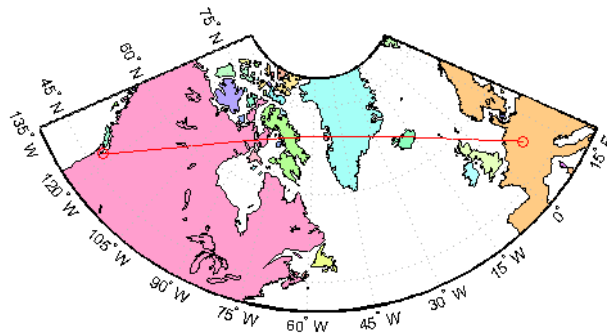
You can compute values along a path without reference to an existing figure by providing a regular data grid and vectors of waypoint coordinates. Optional arguments allow control over the units of the range output and interpolation methods between waypoints and data grid elements.

Show what land and ocean areas lie under a great circle track from Frankfurt to Seattle:

```

cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
Seattle = strmatch('Seattle', {cities(:).Name});
Frankfurt = strmatch('Frankfurt', {cities(:).Name});
lat = [cities(Seattle).Lat cities(Frankfurt).Lat]
lon = [cities(Seattle).Lon cities(Frankfurt).Lon]
load topo
[valp, rngp, latp, lonp] = ...
    mapprofile(double(topo), topolegend, ...
        lat, lon, 'km', 'gc', 'nearest');
figure
worldmap([40 80], [-135 20])
land = shaperead('landareas.shp', 'UseGeoCoords', true);
faceColors = makesymbolspec('Polygon', ...
    {'INDEX', [1 numel(land)], 'FaceColor', ...
    polcmap(numel(land))});
geoshow(land, 'SymbolSpec', faceColors)
plotm(latp, lonp, 'r')
plotm(lat, lon, 'ro')
axis off

```



## See Also

1t1n2val, los2

# maps

---

## Purpose

List available map projections and verify names

## Syntax

```
strmat = maps('namelist')
strmat = maps('idlist')
stdstr = maps(id_string)
```

`maps` displays in the Command Window a table describing all projections available for use.

`strmat = maps('namelist')` returns the English names for the available projections as a matrix of strings.

`strmat = maps('idlist')` returns the standard projection identification strings for the available projections as a matrix of strings.

`stdstr = maps(id_string)` returns the specific standard projection identification string associated with a unique truncation abbreviation.

## Examples

To show the first five entries of the projections name list,

```
str1 = maps('namelist');
str1(1:5,:)
ans =
Balthasart Cylindrical
Behrmann Cylindrical
Bolshoi Sovietskii Atlas Mira
Braun Perspective Cylindrical
Cassini Cylindrical
```

The corresponding shorthand names are

```
str2 = maps('idlist');
str2(1:5,:)
ans =
balthsrt
behrmann
bsam
braun
cassini
```

These are the strings used, for example, when setting the `axesm` property `MapProjection`.

The functions `setm` and `axesm` recognize unique abbreviations (truncations) of these strings. The `maps` function can be used to convert such an abbreviation to the standard ID string:

```
stdstr = maps('merc')
stdstr =
mercator
```

When the function name alone is used,

```
maps

MapTools Projections
CLASS          NAME                                     ID STRING
Cylindrical    Balthasart Cylindrical                            balthsrt
Cylindrical    Behrmann Cylindrical                              behrmann
Cylindrical    Bolshoi Sovietskii Atlas Mira*                    bsam
Cylindrical    Braun Perspective Cylindrical*                   braun
Cylindrical    Cassini Cylindrical                               cassini
Cylindrical    Central Cylindrical*                              ccylin
Cylindrical    Equal Area Cylindrical                            eqacylin
Cylindrical    Equidistant Cylindrical                          eqdcylin
Cylindrical    Gall Isographic                                   giso...
```

The actual result contains all defined projections.

## See Also

`axesm`, `setm`

# mapshow

---

**Purpose** Display map data without projection

**Syntax**

```
mapshow(x,y)
mapshow(x,y, ..., 'DisplayType', displaytype, ...)
mapshow(x,y,z, ..., 'DisplayType', displaytype, ...)
mapshow(Z,R, ..., 'DisplayType', displaytype,...)
mapshow(x,y,I)
mapshow(x,y,BW)
mapshow(x,y,A,cmap)
mapshow(x,y,RGB)
mapshow(I,R)
mapshow(BW,R)
mapshow(RGB,R)
mapshow(A,cmap,R)
mapshow(s)
MAPSHOW(s,...,'SymbolSpec',symspec, ...)
mapshow(filename)
mapshow(..., param1, val1, param2, val2, ...)
mapshow(ax, ...)
mapshow(..., 'Parent', ax, ...)
h = mapshow(...)
```

`mapshow(x,y)` or `mapshow(x,y, ..., 'DisplayType', displaytype, ...)` displays the coordinate vectors `x` and `y`. `x` and `y` can contain embedded NaNs, delimiting individual lines or polygon parts. `displaytype` can be 'point', 'line', or 'polygon' and defaults to 'line'.

`mapshow(x,y,z, ..., 'DisplayType', displaytype, ...)` displays a geolocated data grid. `x` and `y` are M-by-N coordinate arrays, `z` is an M-by-N array of class double, and `displaytype` is 'surface', 'mesh', 'texturemap', or 'contour'. `z` can contain NaN values.

`mapshow(Z,R, ..., 'DisplayType', displaytype,...)` displays a regular data grid, `Z`. `Z` is class double and `displaytype` can be 'surface', 'mesh', 'texturemap', or 'contour'. `R` is a referencing matrix that relates the subscripts of `Z` to map coordinates. If



`DisplayType` is `'texturemap'`, `mapshow` constructs a surface with `ZData` values set to 0.

`mapshow(x,y,I)`, `mapshow(x,y,BW)`, `mapshow(x,y,A,cmap)`, and `mapshow(x,y,RGB)` display a geolocated image as a `texturemap` on a zero-elevation surface. `x` and `y` are geolocation arrays in map coordinates; `I` is a grayscale image, `BW` is a logical image, `A` is an indexed image with colormap `cmap`, or `rgb` is a truecolor image. `x`, `y`, and the image array must match in size. If specified, `DisplayType` must be set to `'image'`. Examples of geolocated images include a color composite from a satellite swath or an image originally referenced to a different coordinate system.

`mapshow(I,R)`, `mapshow(BW,R)`, `mapshow(RGB,R)`, and `mapshow(A,cmap,R)` display an image georeferenced to map coordinates through the referencing matrix `R`. It constructs an image object if the display geometry permits; otherwise, the image is shown as a `texturemap` on a zero-elevation surface. If specified, `'DisplayType'` must be set to `'image'`.

`mapshow(s)` or `MAPSHOW(s,...,'SymbolSpec',symspec,...)` display the vector geographic features stored in the geographic data structure `s` as points, multipoints, lines, or polygons according to the `Geometry` field of `s`. If `s` includes `X` and `Y` fields, then they are used directly to plot features in map coordinates. If `Lat` and `Lon` fields are present in `s` instead, the coordinates are projected using the Plate Carree projection and a warning is issued. `symspec` specifies the symbolization rules used for the vector data through a structure returned by `makesymbolspec`.

If `s` includes `Lat` and `Lon` fields, it may be more appropriate to use `geoshow` to display them. You can project latitude and longitude coordinate values to map coordinates by displaying with `geoshow` on a map axes.

`mapshow(filename)` displays data from `filename`, according to the type of file format. The `DisplayType` parameter is automatically set according to the following table:

Format	DisplayType
Shapefile	'point', 'line', or 'polygon'
GeoTIFF	'image'
TIFF/JPEG/PNG with a world file	'image'
ARC ASCII GRID	'surface' (can be overridden)
SDTS raster	'surface' (can be overridden)

`mapshow(..., param1, val1, param2, val2, ...)` specifies parameter/value pairs that modify the type of display or set MATLAB graphics properties. Parameter names can be abbreviated and are not case-sensitive. Refer to the MATLAB Graphics documentation on `line`, `patch`, `image`, `surface`, `mesh`, and `contour` Handle Graphics object properties for a complete description of these properties and their values.

`mapshow(ax, ...)` and `mapshow(..., 'Parent', ax, ...)` set the axes parent to `ax`.

`h = mapshow(...)` returns a handle to a MATLAB graphics object or, in the case of polygons, a modified patch object. If a `geostruct` or `shapefile` name is input, `mapshow` returns the handle to an `hgroup` object with one child per feature in the `geostruct` or `shapefile`. In the case of a polygon `geostruct` or `shapefile`, each child is a modified patch object; otherwise it is a line object.

## Parameters

Parameters for `mapshow` include

- **DisplayType:** The `DisplayType` parameter specifies the type of graphic display for the data. The value must be consistent with the type of data being displayed, as shown in the following table:

Data Type	Value
vector	'point', 'line', or 'polygon'
image	'image'
grid	'surface', 'mesh', 'texturemap', or 'contour'

- **SymbolSpec:** The SymbolSpec parameter specifies the symbolization rules used for vector data through a structure returned by makesymbolspec. It is used only for vector data.

When both SymbolSpec and one or more graphics properties are specified, the graphics properties will override any settings in the symbolspec structure.

To change the default symbolization rule for a property name/property value pair in the symbolspec, prefix the word 'Default' to the graphics property name (listed in the preceding table).

If PropertyN is 'SymbolSpec', then ValueN must be symspec. symspec should conform to the structure returned by makesymbolspec.

When you use 'SymbolSpec'/symspec and other property name/property value pairs together, the property name/property value pairs override any settings in symspec.

## Graphics Properties

In addition to specifying a parent axes, you can set any appropriate property for a point, line, and polygon DisplayType, as follows:

DisplayType	Properties
'line'	Any MATLAB line property
'point'	Any MATLAB line marker property
'polygon'	Any MATLAB patch property

See the MATLAB Graphics documentation for line, patch, image, and surface properties for complete descriptions of these properties and their values.

## Remarks

You can use `mapshow` to display vector data in an axes figure. However, you should not subsequently change the map projection using `setm`.

`mapshow` adds graphics to the current axes (it does not clear it first), enabling you to create multiple raster and vector map layers. If you do not want `mapshow` to draw on top of an existing map, create a new figure or subplot before calling it.

## Examples

### Example 1

Overlay Boston roads on an orthophoto. You need to convert Boston road vectors to units of survey feet before overlaying them on the image. Note that `mapshow` draws a new layer in the axes rather than replacing its contents:

```
figure
mapshow boston.tif
axis image off

% The orthophoto is in survey feet, the roads are in meters;
% convert the road units to feet before overlaying them.
S = shaperead('boston_roads.shp');
surveyFeetPerMeter = unitsratio('sf','meter');
x = surveyFeetPerMeter * [S.X];
y = surveyFeetPerMeter * [S.Y];
mapshow(x,y)
```

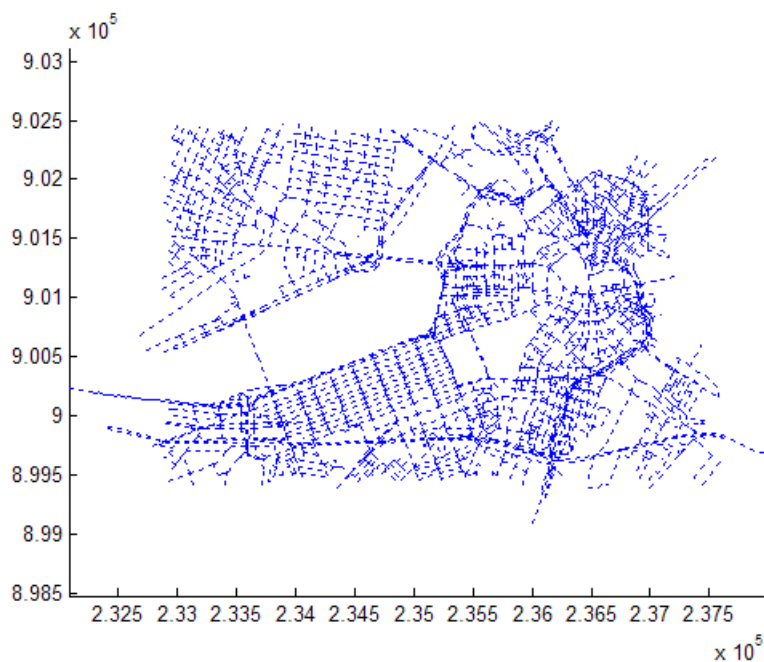


boston.tif image copyright © GeoEye, all rights reserved.

## Example 2

Display Boston roads and change the line style:

```
roads = shaperead('boston_roads.shp');  
figure  
mapshow(roads, 'LineStyle', ':');
```

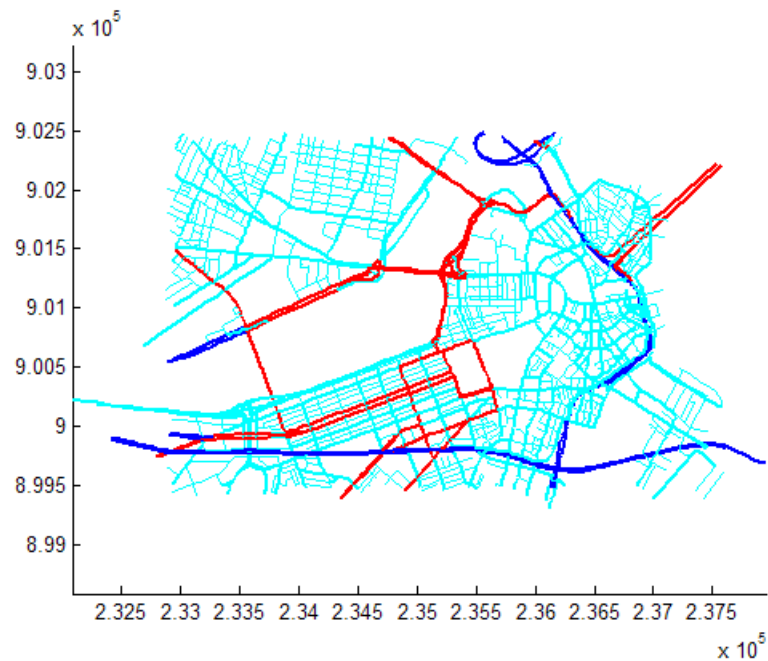


### Example 3

Display the Boston roads shapes using a symbolspec:

```
% Create a SymbolSpec to color local roads:
% (ADMIN_TYPE=0) cyan, state roads (ADMIN_TYPE=3) red.
% Hide very minor roads (CLASS=6).
% Make all roads that are major or larger (CLASS=1-4)
% have a LineWidth of 2.
roadspec = makesymbolspec('Line',...
    {'ADMIN_TYPE',0,'Color','cyan'}, ...
    {'ADMIN_TYPE',3,'Color','red'},...
    {'CLASS',6,'Visible','off'},...
    {'CLASS',[1 4],'LineWidth',2});

figure
mapshow('boston_roads.shp','SymbolSpec',roadspec);
```

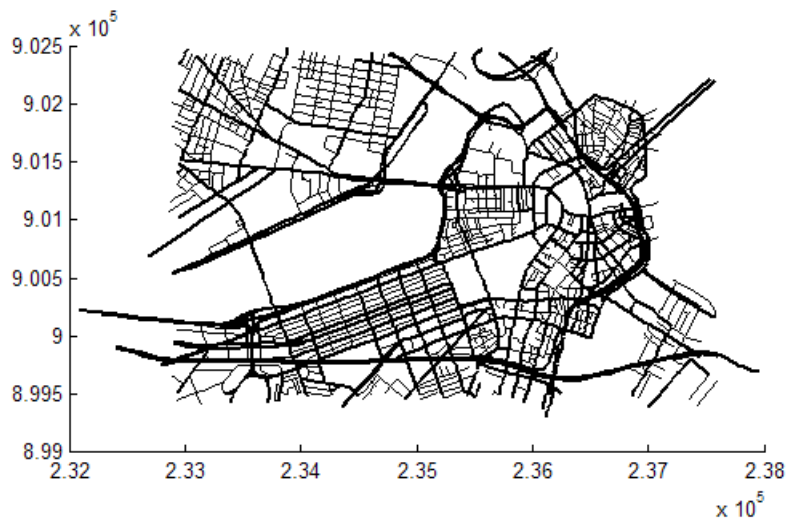


#### Example 4

Override default properties in combination with a symbolspec.

```
roadspec = makesymbolspec('Line',...
    {'Default', 'Color', 'yellow'}, ...
    {'ADMIN_TYPE',0,'Color','c'}, ...
    {'ADMIN_TYPE',3,'Color','r'},...
    {'CLASS',6,'Visible','off'},...
    {'CLASS',[1 4],'LineWidth',2});

figure
mapshow('boston_roads.shp', 'Color', 'black', ...
'SymbolSpec', roadspec);
```



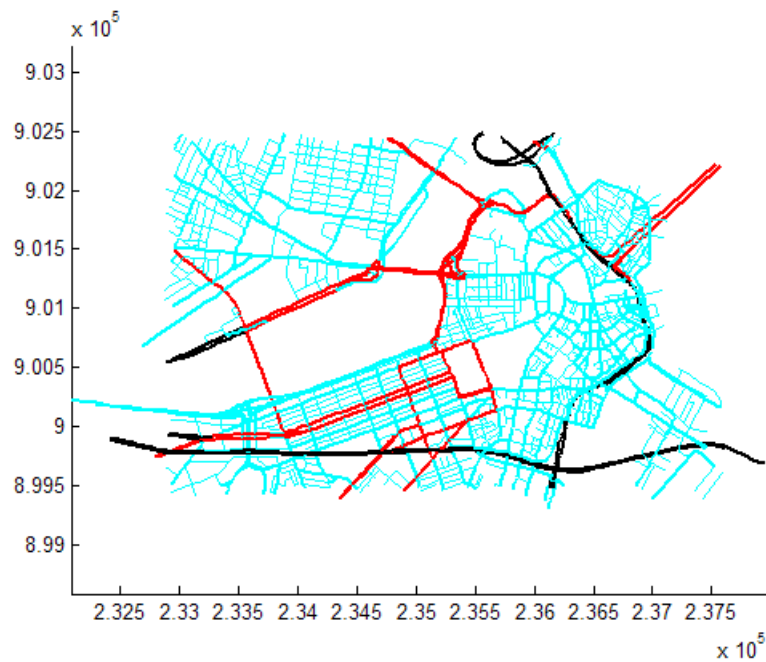
## Example 5

Override default properties of the line with a symbolspec:

```
roadspec = makesymbolspec('Line',...
    {'Default', 'Color', 'black'}, ...
    {'ADMIN_TYPE',0,'Color','c'}, ...
    {'ADMIN_TYPE',3,'Color','r'},...
    {'CLASS',6,'Visible','off'},...
    {'CLASS',[1 4],'LineWidth',2});

figure
mapshow('boston_roads.shp','SymbolSpec',roadspec);
```





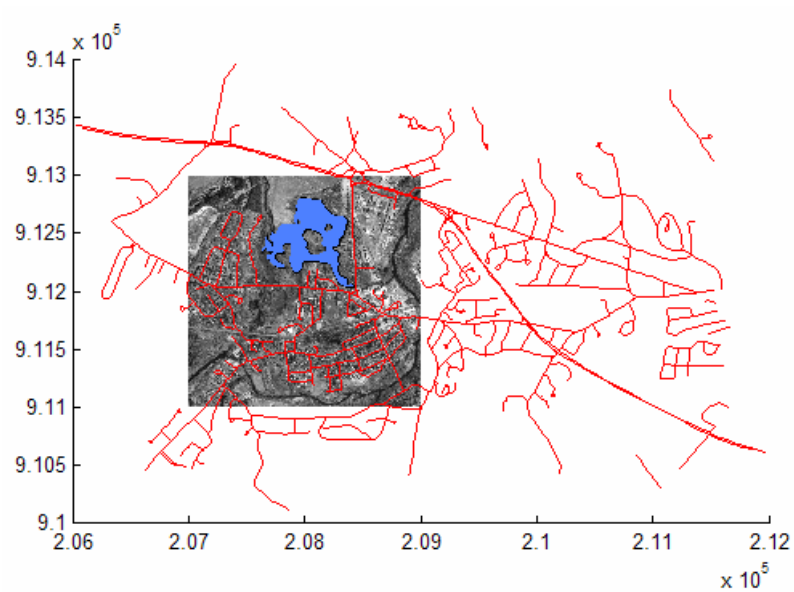
### Example 6

Display an orthophoto of Concord, MA, including a pond with three large islands:

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

% Overlay a polygon representing the same pond
% (feature 14 in the concord_hydro_area shapefile).
% Note that the islands are visible in the orthophoto
% through three "holes" in the pond polygon.
pond = shaperead('concord_hydro_area.shp', 'RecordNumbers', 14);
mapshow(pond, 'FaceColor', [0.3 0.5 1], 'EdgeColor', 'black')
```

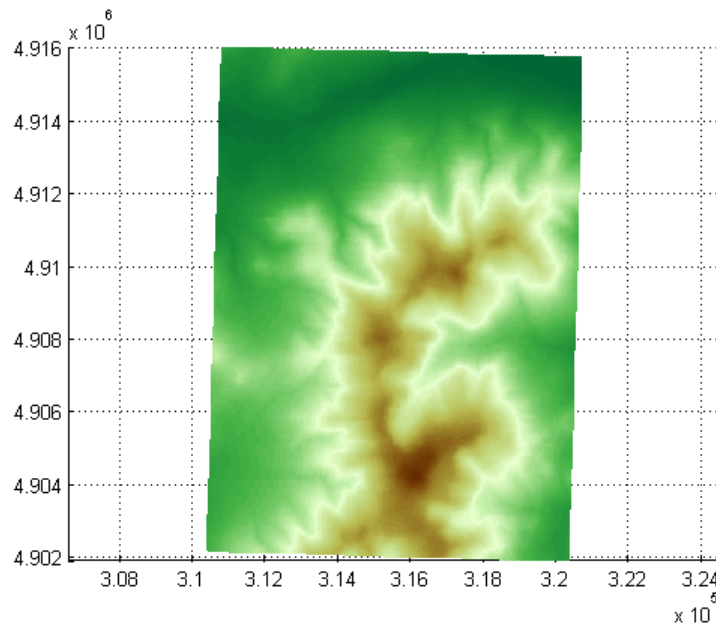
```
% Overlay roads in the same figure.  
mapshow('concord_roads.shp', 'Color', 'red', 'LineWidth', 1);
```



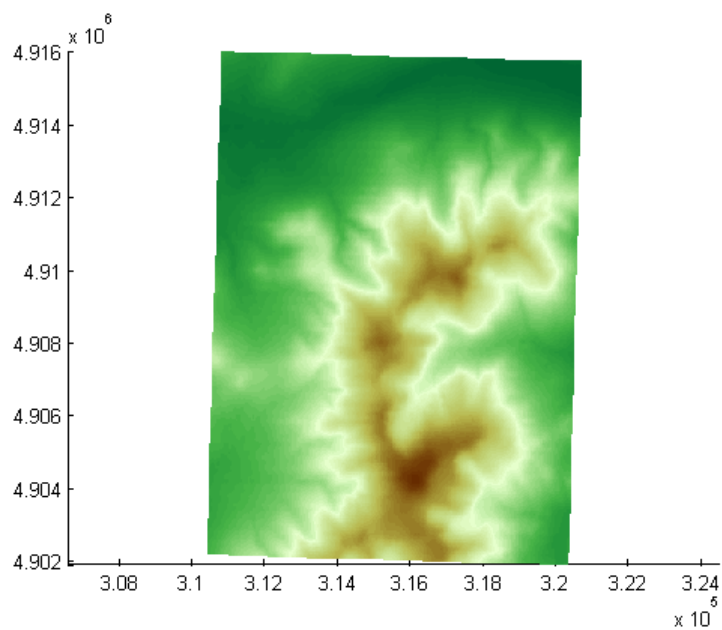
## Example 7

Read and view the Mount Washington SDTS DEM terrain data three different ways:

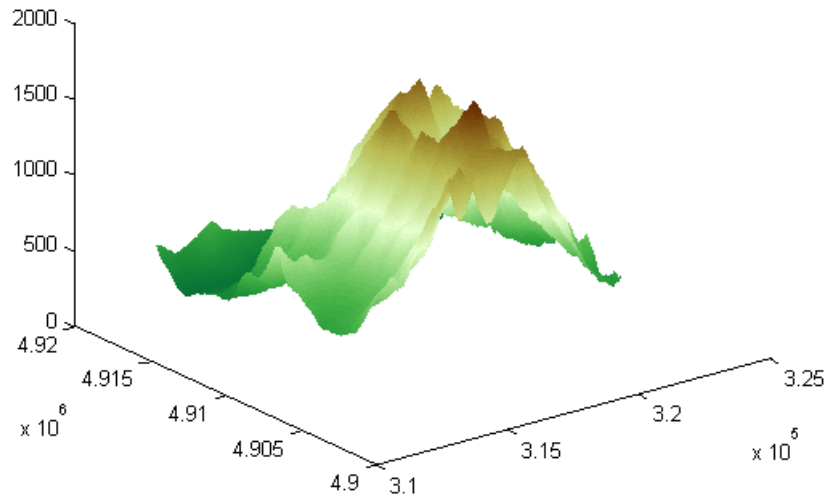
```
[Z, R] = sdtsemread('9129CATD.DDF');  
  
% View the Mount Washington terrain data as a mesh.  
figure  
mapshow(Z, R, 'DisplayType', 'mesh');  
colormap(demcmap(Z))
```



```
% View the Mount Washington terrain data as a surface.  
figure  
mapshow(Z, R, 'DisplayType', 'surface');  
colormap(demcmap(Z))
```



```
% View as a 3-D surface.  
view(3);  
axis normal
```

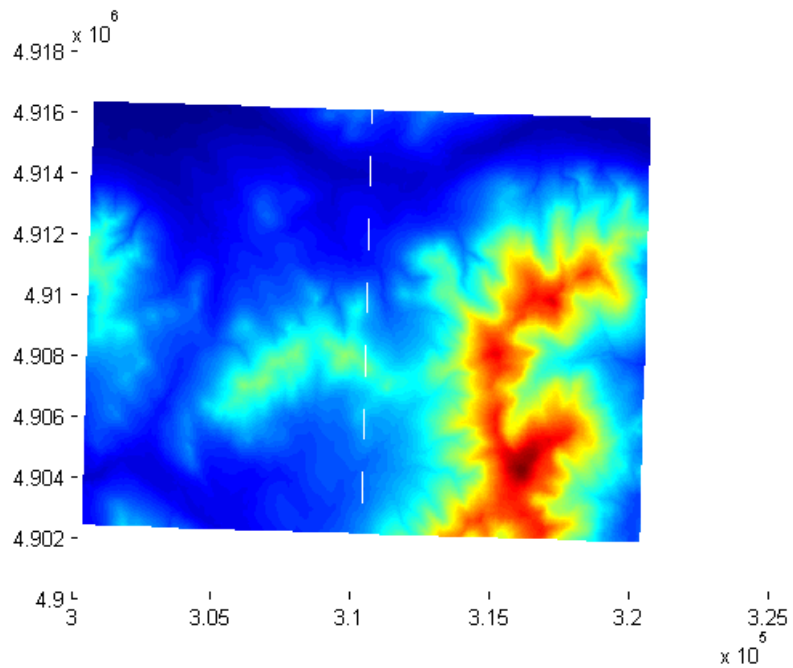


### Example 8

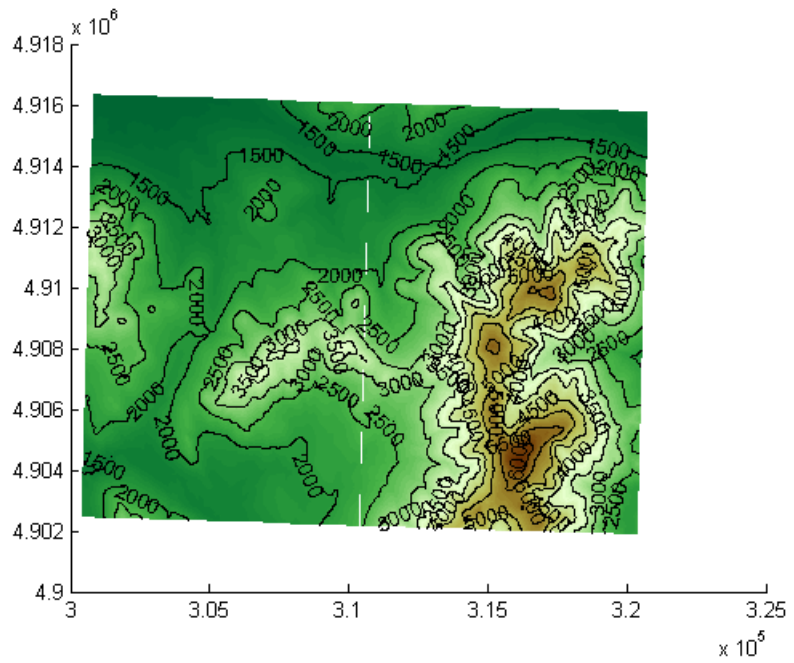
Display the grid and contour lines of Mount Washington and Mount Dartmouth.

```
% Read the terrain data files.
[Z_W, R_W] = arcgridread('MtWashington-ft.grd');
[Z_D, R_D] = arcgridread('MountDartmouth-ft.grd');

% Display the terrain data as a texture map.
figure
hold on
h1 = mapshow(Z_W, R_W, 'DisplayType', 'texturemap');
h2 = mapshow(Z_D, R_D, 'DisplayType', 'texturemap');
set([h1, h2], 'FaceColor', 'flat');
```



```
% Overlay black contour lines with labels onto the texturemap.  
mapshow(Z_W, R_W, 'DisplayType', 'contour', ...  
        'LineColor','black', 'ShowText', 'on');  
mapshow(Z_D, R_D, 'DisplayType', 'contour', ...  
        'LineColor','black', 'ShowText', 'on');  
  
% Set the colormap appropriate to terrain elevation.  
colormap(demcmap(Z_W))
```

**See Also**

geoshow, makesymbolspec, mapview, shaperead

# maptriml

---

**Purpose** Trim lines to latitude-longitude quadrangle

**Syntax** `[lat,lon] = maptriml(lat0,lon0,latlim,lonlim)`  
`[lat,lon] = maptriml(lat0,lon0,latlim,lonlim)` returns *filtered* NaN-delimited vector map data sets from which all points lying outside the desired latitude and longitude limits have been discarded. These limits are specified by the two-element vectors `latlim` and `lonlim`, which have the form `[south-limit north-limit]` and `[west-limit east-limit]`, respectively.

**Examples** Following is a simple example:

```
lat0 = [1:10,9:-1:0]; lon0 = -30:-11;  
[lat,lon] = maptriml(lat0,lon0,[3 7],[-29 -12]);  
[lat lon]
```

```
ans =  
    NaN    NaN  
     3    -28  
     4    -27  
     5    -26  
     6    -25  
     7    -24  
    NaN    NaN  
     7    -18  
     6    -17  
     5    -16  
     4    -15  
     3    -14  
    NaN    NaN
```

Notice that trimmed line segment ends have NaNs inserted at trim points.

**See Also** `maptrimp`, `maptrims`



**Purpose**

Trim polygons to latitude-longitude quadrangle

**Syntax**

```
[latTrimmed,lonTrimmed] = maptrimp(lat,lon,latlim,lonlim)
[latTrimmed,lonTrimmed] = maptrimp(lat,lon,latlim,lonlim)

```

trims the polygons in `lat` and `lon` to the quadrangle specified by `latlim` and `lonlim`. `latlim` and `lonlim` are two-element vectors, defining the latitude and longitude limits respectively. `lat` and `lon` must be vectors that represent valid polygons.

**Remarks**

`maptrimp` conditions the longitude limits such that:

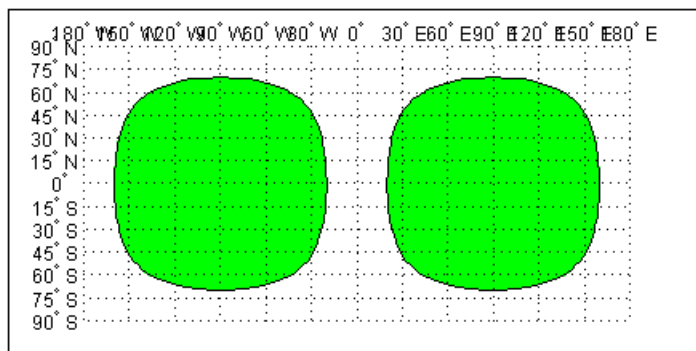
- `lonlim(2)` always exceeds `lonlim(1)`
- `lonlim(2)` never exceeds `lonlim(1)` by more than 360
- `lonlim(1) < 180` or `lonlim(2) > -180`
- Should the quadrangle span the Greenwich meridian, then that meridian appears at longitude = 0.

**Examples**

Make two polygons using the `scircle1` function, and display them:

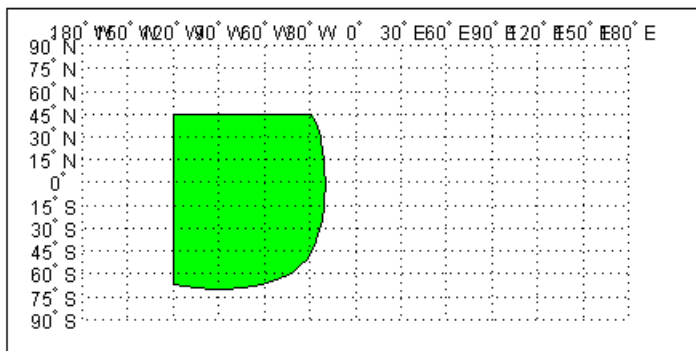
```
[latTrimmed,lonTrimmed] = scircle1([0 0]',[-90 90]',[70 70]');
axesm('pcarree','Grid','on',...
      'MeridianLabel','on','ParallelLabel','on')
h = fillm(latTrimmed,lonTrimmed,'green');
```

# maptrim



Now trim the data to lie between 80°S and 45°N latitude, and 120°W and 0° longitude. The coordinates are in two-column arrays coming out of `scircle1`, which you must first turn into NaN-delimited vectors:

```
latTrimmed = [latTrimmed; NaN NaN];  
lonTrimmed = [lonTrimmed; NaN NaN];  
[lat,lon] = maptrim(latTrimmed(:),lonTrimmed(:),...  
                  [-80 45],[-120 0]);  
clmo(h)  
fillm(lat,lon,'green')
```



Notice that the patch face to the east, lying completely outside the allowed area, was removed. The western patch was trimmed to the required area.

**See Also**

maptrim1, maptrims

# maptrims

---

## Purpose

Trim regular data grid to latitude-longitude quadrangle

## Syntax

```
[subgrid,subrefvec] = maptrims(Z,refvec,latlim,lonlim)
[subgrid,subrefvec] = maptrims(Z,refvec,latlim,lonlim,scale)
```

[subgrid,subrefvec] = maptrims(Z,refvec,latlim,lonlim) returns the subset of the input regular data grid between the latitude and longitude limits, in degrees, defined by the two-element vectors latlim and lonlim. refvec is a three-element referencing vector that geolocates the input data grid; subrefvec is the referencing vector of the output data grid.

[subgrid,subrefvec] = maptrims(Z,refvec,latlim,lonlim,scale) is a means of further reducing the size of the output matrix. The cells-per-degree scale of the original matrix is given by the first element of refvec. The desired cells-per-degree scale in the output grid is given by scale, which must equally divide refvec(1). For example, if refvec(1) were 20 (cells per degree), then scale could be 1, 2, 4, 5, 10, or 20.

## Description

The maptrims function selects a portion of a larger data grid defined by a latitude-longitude quadrangle.

The reduced matrix is created using `resizem` with a 'nearest' interpolation method.

## Examples

```
load topo
[subgrid,subrefvec] = maptrims(topo,topolegend,...
                               [80.25 85.3],[165.2 170.7])
```

```
subgrid =
    -2826    -2810    -2802    -2793
    -2915    -2913    -2905    -2884
    -3192    -3186    -3165    -3122
    -3399    -3324    -3273    -3214
```

```
subrefvec =
```

1 85 166

The upper left corner of the grid might differ slightly from that of the requested region. `maptrims` uses the corner coordinates of the first cell inside the limits.

**See Also**

`maptriml`, `maptrimp`, `resizem`

**Purpose** Interactive map viewer

**Description** Use the Map Viewer to work with vector, image, and raster data grids in a map coordinate system: load data, pan and zoom on the map, control the map scale of your screen display, control the order, visibility, and symbolization of map layers, annotate your map, and click to learn more about individual vector features. `mapview` complements `mapshow` and `geoshow`, which are for constructing maps in ordinary figure windows in a less interactive, script-oriented way.

**Syntax** `mapview`

`mapview` (with no arguments) starts a new Map Viewer in an empty state. The Map Viewer is a self-contained GUI for viewing geospatial data in map ( $x$ - $y$ ) coordinates. For usage information, see the following sections. You can also work through the Map Viewer tutorial, “Tour Boston with the Map Viewer” on page 1-9.

**Importing Data** The Map Viewer opens with no data loaded and an empty map display window. The first step is to import a data set. Use the options in the **File** menu to select data from a file or from the MATLAB workspace:

### **Import From File**

Use the file browsing dialog to open a file in one of the following formats: Shapefile, GeoTIFF, SDTS DEM, Arc ASCII Grid, TIFF, JPEG, or PNG with world file. This option imports the data into the viewer but does not add it to your workspace.

To view standard-format geodata files provided with Mapping Toolbox, set your working directory or navigate the Map Viewer Open dialog to

`matlabroot/toolbox/map/mapdemos`

## Import From Workspace

**Images.** Use the **Raster Data > Image** import dialog to select a **referencing matrix and data name** for the image from the list of workspace variables. If the image type is truecolor (RGB), specify which band represents the red, green, and blue intensities.

**Data grids.** Use the **Raster Data > Grid** import dialog to select X and Y geolocation and data grid array names from the list of workspace variables.

**Vector data.** Use the **Vector Data > Map coordinates** import dialog to select X and Y variables for map coordinates from the list of workspace variables and identify the type of geometry to be displayed (**Point, Line, or Polygon**). The X and Y variables can specify multiple line segments or multiple polygons if they contain NaNs at matching locations in the coordinate vectors.

**Vector geographic data structure.** Use the **Vector Data > Geographic data structure** import dialog to select the struct that contains vector map data from the list of workspace variables.

Once you import your first data set, the Map Viewer automatically sets the limits of its map display window to the spatial extent of the imported data.

## Working in Map Coordinates

As you move any of the Map Viewer cursors across the map display area, the coordinate readout in the lower left corners shows you the cursor position in map X and Y coordinates.

The Map Viewer requires that all currently viewed data sets possess the same coordinate system and length units. This is likely to be the case for data sets that originated from a common source. If it is not the case, you will need to adjust coordinates before importing data into the Map Viewer.

If some or all of your data is in geographic coordinates, use `projfw` or `mfwdtran` to project latitudes and longitudes to your desired map

coordinate system before you import it. When starting from a different projection, you must first unproject to latitude and longitude using `projinv` or `minvtran`, then reproject with `projfwd` or `mfwdtran`. You might also need to adjust the horizontal datum of your data using, for example, the free GEOTRANS (Geographic Translator) application from the Geospatial Sciences Division of the U.S. National Geospatial-Intelligence Agency (NGA). If you simply need a change of units, multiply by the appropriate conversion factor obtained from `unitsratio`.

`mapview` can also display data in unprojected geographic coordinates, if you consistently substitute longitude for map X and latitude for map Y. Geographic coordinates must be consistently expressed in either degrees or radians (not both at once). When using geographic coordinates, do not specify the viewer's map units (see below); you can only use the Map Viewer's map scale display when working in linear units of length.

## Setting Map Units and Scale

If you tell the Map Viewer which length unit you are using, it can calculate an approximate map scale for your onscreen display. Set the map units with either the drop-down menu at the bottom of the display or the **Set Map Units** item in the **Tools** menu.

The scale computed by the Map Viewer is displayed in the window just above the map units drop-down. To change your display scale while keeping the center of the map display fixed, simply edit this text box.

Make sure to format your text in the standard way ( $1:N$ , where  $N$  is a positive number such that a distance on the ground is  $N$  times the same distance on your screen, e.g.,  $1:24000$ ).

The scale is approximate because it depends on the MATLAB estimate of the size of your screen pixels. It is also approximate if your projection introduces significant distortion. If your data falls in a fairly small area and you use a conformal projection (e.g., UTM with all data in a single zone), the scale will be very consistent across your entire map.

## Navigating Your Map

By default, the Map Viewer sets the limits of your map window to match the extent of the first data set that you load. You will probably want to adjust this to see some areas in greater detail.



The Map Viewer provides several tools to control the limits of your map window and the map scale of the data display. Some are familiar from standard MATLAB figure windows.

- **Zoom in:** Drag a box to zoom in on a specific area or click a point to zoom in with that point centered in the map display.
- **Zoom out:** Click a point to zoom out with that point centered in the map display.
- **Pan tool:** Click, hold, and drag to reposition the selected point in the display window, while holding the map scale fixed. Release when you are satisfied with new display limits.
- **Fit to window:** Set the map display to enclose all currently loaded data layers. This is equivalent to selecting **Fit to Window** in the **View** menu.
- **Back to previous view:** Click this button once to return the map scale and display center to their values prior to the most recent zoom, pan, or scale change. Click repeatedly to undo earlier changes. This is equivalent to selecting **Previous View** in the **View** menu.

Another way to zoom in or out while keeping the center of the view fixed at the same map coordinates is to directly edit the map scale box at the bottom of the screen.

## Managing Map Layers

Each time you import a set of vectors, an image, or a data grid into the Map Viewer, the new data is stored in a new map layer. The layers form an ordered stack. Each layer is listed as an item in the **Layers** menu, with its position in the menu indicating its position in the stack.

When you import a new layer, the Map Viewer automatically places it at the top of the layer stack. To reposition a layer in the stack, select it in the **Layers** menu, slide right, and select **To Top**, **To Bottom**, **Move Up**, or **Move Down** from the pop-up submenu.

The vector features or raster in a given layer obscure coincident elements of any underlying layers. To control layers that are obscuring one another, you can also toggle layer visibility on and off. Use the item

**Visible** in the slide-right menu. Or, simply remove a layer from the Map Viewer via the **Remove** item in the slide-right menu. Remember that even if a layer's visibility is *on*, the layer does not appear if its contents are located completely outside the current display limits or are obscured by another layer.

## Symbolizing Vector Features

When point, line, and polygon layers are loaded, the Map Viewer initializes their graphics properties as follows:

Geometry	Properties
Point (line objects)	LineStyle = 'none' Marker = 'x' MarkerEdgeColor = <randomly generated value> MarkerFaceColor = 'none'
Line (line objects)	Color = <randomly generated value> LineStyle = '-' Marker = 'none'
Polygon (patch objects)	EdgeColor = [0 0 0] FaceColor = <randomly generated value>

To override symbolism defaults for a vector layer, use `makesymbolspec` to create a symbol specification in the workspace. A `symbolspec` contains a set of rules for setting vector graphics properties based on the values of feature attributes. For instance, if you have a line layer representing roads of various classes (e.g., major highway, secondary road, etc.), you can create a `symbolspec` to use a different color and/or line width and/or line style for each road class. See the `makesymbolspec` help for examples and to learn how to construct a `symbolspec`. If you regularly work with data sets sharing a common set of feature attributes, you might want to save one or more `symbolspec`s in a MAT-file (or save calls to `makesymbolspec` in an M-file).

Once you have a `symbolspec` in your workspace, select your vector layer in the **Layers** menu, then slide right and click **Set Symbol Spec**,

which opens a dialog box. Use the dialog box to select the symbolspec from your workspace.

## Getting Information About Vector Features

The **Datatip** tool and the **Info** tool provide different ways to check the attributes of vector features that you select graphically. Before using either tool you must designate one of your vector layers as *active*. (The default active layer is the first one that you imported.) Either use the **Active Layer** drop-down menu at the bottom of your screen or select the layer in the **Layers** menu, slide right, and select **Active**. Having a designated active layer ensures that when you click a feature you don't inadvertently select an overlapping feature from a different layer.

- **Datatip tool:** The **Datatip** tool displays a feature attribute in a text label each time you click a vector feature. By default the attribute is the first one in the layer's attribute list. To change which attribute is used, select the layer in the **Layers** menu, slide right, and click **Set Layer Attribute**. In the dialog that follows, select a different attribute, or **Index**. If you choose **Index**, the Map Viewer displays the one-based index value corresponding to a given feature—based on its position in the input file or workspace array. To remove a text label, right-click it and choose **Delete datatip** from the context menu. Or choose **Delete all datatips** from the context menu or the **Tools** menu.
- **Info tool:** The **Info** tool opens a separate text window each time you click a vector feature. The window displays all the attribute names and values for that feature, in contrast to the **Datatip** tool, which displays only the value of a single attribute. If you need to compare two or more features, simply click each one and view the info windows together. Use its close button to close an info window when you're done with it, or choose **Close All Info Windows** from the **Tools** menu.

## Annotating Your Map

Use the **text**, **line**, or **arrow** annotation tools to mark and highlight points of interest on your map, or select the corresponding items in the **Insert** menu. Note that to insert an additional object of the same type, you must reselect the appropriate tool. In addition, the **Insert** menu

allows you to insert axis labels and a title. Use the **Select annotations** tool and **Edit** menu to modify or remove your annotations. The Map Viewer manages annotations separately from data layers; annotations always stay on top. Note that annotations cannot be saved as graphic objects, although you can export maps containing annotations to an image format as described below.

## Creating and Using Additional Views

Use **New View** on the **File** menu to create an additional Map Viewer window linked to an existing window. Consider using an additional window when you want to see your map at different scales at the same time (e.g., a detailed view plus an overview), or when you want to simultaneously see different areas of the map at large scale. You can create as many additional windows as you need, and close them when you want. Your mapview session ends when you close the last window.

Options for creating a new viewer window include: **Duplicate Current View**, **Full Extent**, **Full Extent of Active Layer**, and **Selected Area**. Click and drag with the **Select area** tool to define a selected area.

A new viewer window differs from existing windows mainly in terms of the visible map extent and scale (it also omits annotations and any labels you added with the datatip tool). You will see the same layers in the same order with the same settings (including the active layer). Updates to layers (insertion/removal, order, visibility, label attribute, and symbolization) in one viewer window are propagated automatically to all the windows with which it is linked. Updates to annotations and datatip labels are not propagated between viewers. If you need two different layer configurations in different windows, launch a second mapview from the command line instead of creating an additional window. The views it contains will not be linked to previous ones.

## Exporting Your Map

The Map Viewer allows you to export all or part of your map for use in a publication or on a Web page. Use **File > Save As Raster Map** to export an image of either the current display extent or an area outlined with the **Select area** tool. Select a format (PNG, TIFF, JPEG) from the drop-down menu in the export dialog. For maps including vector layers, PNG (Portable Network Graphics) is often the best choice. This format provides excellent quality, good compression, and is well supported

by modern Web browsers. The export process automatically creates a world file (ending with suffix `tfw`, `jpgw`, or `pgw`) as well; the pair of files constitute a georeferenced image that itself can be displayed with `mapview`, `mapshow`, and many external GIS packages.

**See Also**

`arcgridread`, `geoshow`, `geotiffread`, `makesymbolspec`, `mapshow`, `sdtsemread`, `shaperead`, `updategeostruct`, `worldfileread`

# mat2dms

---

## Purpose

Convert [deg min sec] matrix to deg:min:sec encoding

---

**Note** The `mat2dms` function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox. Instead use `dms2degrees` to convert degrees-minutes-seconds vector to degrees.

---

## Syntax

```
anglout = mat2dms(d,m,s)
anglout = mat2dms(d,m,s,n)
anglout = mat2dms([d,m,s],n)
```

`anglout = mat2dms(d,m,s)` takes angles separated into three inputs, one each for degrees, minutes, and seconds, and converts them to single dms values.

`anglout = mat2dms(d,m,s,n)` specifies the power of 10, `n`, to which the input seconds should be rounded before they are converted (that is, if a result is 12.567 seconds, and `n = -2`, the resulting seconds output would be 12.57). The default value of `n` is -2.

`anglout = mat2dms([d,m,s],n)` allows the inputs to be packed into a three-column matrix in which the columns represent degrees, minutes, and seconds, respectively.

## Examples

```
anglout = mat2dms(23,45,17.5)
```

```
anglout =
           2345.175
```

## See Also

`dms2degrees` , `degrees2dms`

**Purpose** Convert [hrs min sec] matrix to hms format

---

**Note** The mat2hms function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox.

---

## Syntax

```
timeout = mat2hms(h,m,s)
timeout = mat2hms(h,m,s,n)
timeout = mat2hms([h,m,s],n)
```

timeout = mat2hms(h,m,s) takes times separated into three inputs, one each for hours, minutes, and seconds, and converts them to single hms values.

timeout = mat2hms(h,m,s,n) specifies the power of 10, n, to which the input seconds should be rounded before they are converted (that is, if a result is 12.567 seconds, and n = -2, the resulting seconds output would be 12.57). The default value of n is -2.

timeout = mat2hms([h,m,s],n) allows the inputs to be packed into a three-column matrix in which the columns represent hours, minutes, and seconds, respectively.

## Examples

```
timeout = mat2hms([13 35],[34 18],[29.8 17.0])
timeout =
           1334.298           3518.17
```

# mdistort

---

**Purpose** Display contours of constant map distortion

**Syntax**

```
mdistort
mdistort off
mdistort(parameter)
mdistort parameter
mdistort(parameter,levels)
mdistort(parameter,levels,gsiz)
[h,ht] = mdistort(...)
```

`mdistort`, with no input arguments, toggles the display of contours of projection-induced distortion on the current map axes. The magnitude of the distortion is reported in percent.

`mdistort off` removes the contours.

`mdistort(parameter)` or `mdistort parameter` displays contours of distortion for the specified parameter. Recognized *parameter* strings are 'area', 'angles' for the maximum angular distortion of right angles, 'scale' or 'maxscale' for the maximum scale, 'minscale' for the minimum scale, 'parscale' for scale along the parallels, 'merscale' for scale along the meridians, and 'scaleratio' for the ratio of maximum and minimum scale. If omitted, the 'maxscale' parameter is displayed. All parameters are displayed as percent distortion except angles, which are displayed in degrees.

`mdistort(parameter,levels)` specifies the levels for which the contours are drawn. *levels* is a vector of values as used by `contour`. If empty, the default levels are used.

`mdistort(parameter,levels,gsiz)` controls the size of the underlying graticule matrix used to compute the contours. *gsiz* is a two-element vector containing the number of rows and columns. If omitted, the default Mapping Toolbox graticule size of [50 100] is assumed.

`[h,ht] = mdistort(...)` returns the handles to the line and text objects.



## Background

Map projections inevitably introduce distortions in the shape and size of objects as they are transformed from three-dimensional spherical coordinates to two-dimensional Cartesian coordinates. The amount and type of distortion vary between projections, over the projection, and with the selection of projection parameters such as standard parallels. This function provides a quantitative graphical display of distortion parameters.

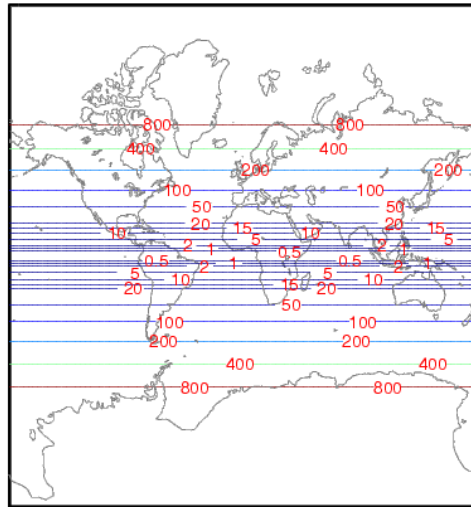
`mdistort` is not intended for use with UTM. Distortion is minimal within a given UTM zone. `mdistort` issues a warning if a UTM projection is encountered.

## Examples

### Example 1

Note the extreme area distortion of the Mercator projection. This makes it ill-suited for global displays.

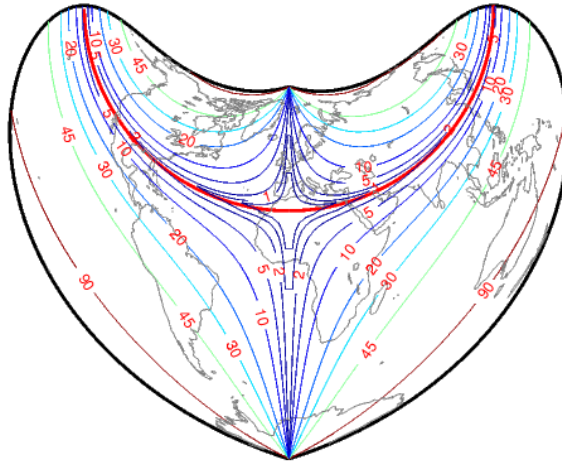
```
figure
axesm mercator
load coast
framem;plotm(lat, long, 'color', .5*[1 1 1])
mdistort area
```



## Example 2

The lines of zero distortion for the Bonne projection follow the central meridian and the standard parallel.

```
figure
axesm bonne
load coast
framem;plotm(lat, long,'color',.5*[1 1 1])
mdistort angles
parallelui
```



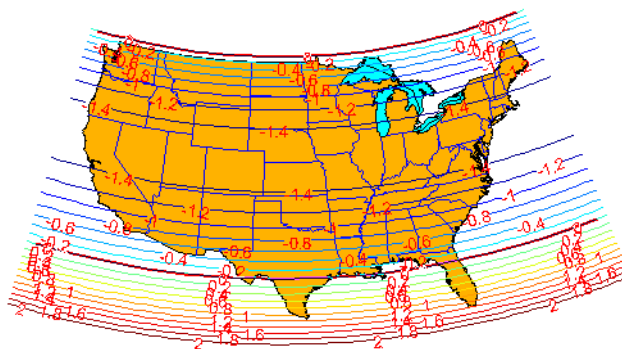
### Example 3

An equidistant conic projection with properly chosen parallels can map the conterminous United States with less than 1.5% distortion.

```
figure
usamap conus
load conus
patchm(uslat, uslon, [1 0.7 0])
plotm(statelat, statelon)
patchm(gtlakelat, gtlakelon, 'cyan')
framem off; gridm off; mlabel off; plabel off
mdistort('parscale', -2:.2:2)
parallelui
```

# mdistort

---



## Remarks

`mdistort` can help in the placement of standard parallels for projections. Standard parallels are generally placed to minimize distortion over the region of interest. The default parallel locations might not be appropriate for maps of smaller regions. By using `mdistort` and `parallelui`, you can immediately see how the movement of parallels reduces distortion.

## See Also

`tissot`, `distortcalc`, `vfdtran`

**Purpose**

Mean location of geographic coordinates

**Syntax**

```
[latmean,lonmean] = meanm(lat,lon)
[latmean,lonmean] = meanm(lat,lon,units)
[latmean,lonmean] = meanm(lat,lon,ellipsoid)
```

[latmean,lonmean] = meanm(lat,lon) returns row vectors of the geographic mean positions of the columns of the input latitude and longitude points.

[latmean,lonmean] = meanm(lat,lon,units) indicates the angular units of the data. When the standard angle string *units* is omitted, 'degrees' is assumed.

[latmean,lonmean] = meanm(lat,lon,ellipsoid) specifies the elliptical definition of the Earth to be used with the two-element ellipsoid vector. The default ellipsoid model is a spherical Earth, which is sufficient for most applications.

If a single output argument is used, then `geomeans = [latmean,longmean]`. This is particularly useful if the original `lat` and `lon` inputs are column vectors.

**Background**

Finding the mean position of geographic points is more complicated than simply averaging the latitudes and longitudes. `meanm` determines mean position through three-dimensional vector addition. See “Geographic Statistics” on page 9-2 in the *Mapping Toolbox User’s Guide*.

**Examples**

Create random latitude and longitude matrices:

```
lat = rand(3)

lat =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

lon = rand(3)
```

## meanm

---

```
lon =  
    0.4447    0.9218    0.4057  
    0.6154    0.7382    0.9355  
    0.7919    0.1763    0.9169  
  
[latmean,lonmean] = meanm(lat,lon,'radians')  
  
latmean =  
    0.6004    0.7395    0.4448  
lonmean =  
    0.6347    0.6324    0.7478
```

### See Also

`filterm`, `hista`, `histr`, `stdist`, `stdm`

**Purpose** Ellipsoidal distance along meridian

**Syntax** `s = meridianarc(phi1, phi2, ellipsoid)`  
`s = meridianarc(phi1, phi2, ellipsoid)` calculates the (signed) distance `s` between latitudes `phi1` and `phi2` along a meridian on the ellipsoid defined by the 1-by-2 vector `ellipsoid`. `phi1` and `phi2` are in radians. `s` has the same units as the semimajor axis of the ellipsoid. `s` is negative if `phi2` is less than `phi1`.

**See Also** `meridianfwd`

# meridianfwd

---

**Purpose** Reckon position along meridian

**Syntax** `phi2 = meridianfwd(phi1, s, ellipsoid)`  
`phi2 = meridianfwd(phi1, s, ellipsoid)` determines the geodetic latitude `phi2` reached by starting at geodetic latitude `phi1` and traveling distance `s` north (positive `s`) or south (negative `s`) along an a meridian on the specified `ellipsoid`. `phi1` and `phi2` are in radians and `s` has the same units as the semimajor axis of the ellipsoid.

**See Also** `meridianarc`



**Purpose**

Construct map graticule for surface object display

**Syntax**

```
[latgrat,longrat] = meshgrat(Z,refvec)
[latgrat,longrat] = meshgrat(Z,refvec,npts)
[latgrat,longrat] = meshgrat(lat,lon)
[latgrat,longrat] = meshgrat(latlim,lonlim,npts)
[latgrat,longrat] = meshgrat(lat,lon,units)
[latgrat,longrat] = meshgrat(latlim,lonlim,npts,units)
```

[latgrat,longrat] = meshgrat(Z,refvec) constructs a graticule for the regular data grid `grid` with the associated three-element referencing vector `refvec`. The default graticule size is equal to the size of the grid.

[latgrat,longrat] = meshgrat(Z,refvec,npts) returns a graticule mesh of size `npts`. The input `npts` is a two-element vector of the form [latitude-points longitude-points]. If `npts` is set to an empty matrix, then the graticule returned is the Mapping Toolbox default graticule size [50 100].

[latgrat,longrat] = meshgrat(lat,lon) can be used for data grids that are not regular in spacing (e.g., row one represents 1°, row two represents 1.34°) but are regular in orientation (rows are north-south, columns are east-west). The inputs `lat` and `lon` are vectors describing the latitudes and longitudes on a row-by-row and column-by-column basis for the data grid to be displayed. Regardless of the variable spacing of the matrix, the graticule is evenly spaced. In this form, `meshgrat` is similar to the MATLAB function `meshgrid`.

[latgrat,longrat] = meshgrat(latlim,lonlim,npts) returns a graticule mesh of size `npts`. The input vectors `latlim` and `lonlim` are two-element vectors specifying the graticule latitude and longitude limits. The input `npts` is a two-element vector of the form [latitude-points longitude-points]. If `npts` is set to an empty matrix, then the graticule returned is the Mapping Toolbox default graticule size [50 100].

[latgrat,longrat] = meshgrat(lat,lon,units) and  
[latgrat,longrat] = meshgrat(latlim,lonlim,npts,units) use

# meshgrat

---

the input *units* to specify the angle units of the input and output parameters. If omitted, 'degrees' is assumed.

## Description

The graticule mesh is a grid of points that are projected on a map axes and to which surface map objects are warped. The fineness, or resolution, of this grid determines the quality of the projection and the speed of plotting. There is no hard and fast rule for sufficient graticule resolution, but in general, cylindrical projections need very few graticules in the longitudinal direction, while complex curve-generating projections require more.

## Examples

Make a (coarse) graticule for the entire world:

```
latlim = [-90 90]; longlim = [-180 180];
[latgrat,longrat] = meshgrat(latlim,longlim,[3 6])

latgrat =
  -90.0000  -90.0000  -90.0000  -90.0000  -90.0000  -90.0000
           0           0           0           0           0           0
   90.0000   90.0000   90.0000   90.0000   90.0000   90.0000
longrat =
 -180.0000 -108.0000  -36.0000   36.0000  108.0000  180.0000
 -180.0000 -108.0000  -36.0000   36.0000  108.0000  180.0000
 -180.0000 -108.0000  -36.0000   36.0000  108.0000  180.0000
```

These paired coordinates are the graticule vertices, which are projected according to the requirements of the desired map projection. Then a surface object like the topo map can be warped to the grid.

## See Also

meshm, pcolorm, surfacem, surfm

**Purpose**

3-D lighted shaded relief of regular data grid

**Syntax**

```
meshlsrm(Z,refvec)
meshlsrm(Z,refvec,[azim elev])
meshlsrm(Z,refvec,[azim elev],cmap)
meshlsrm(Z,refvec,[azim elev],cmap,clim)
h = meshlsrm(...)
```

`meshlsrm(Z,refvec)` displays a regular data grid, `Z`, geolocated by a three-element referencing vector, `refvec`, coloring the map according to elevation and surface slopes. The current axes must have a valid map projection definition.

`meshlsrm(Z,refvec,[azim elev])` displays the regular data grid with the light coming from the specified azimuth and elevation. Lighting is applied before the data is projected. Angles are in degrees, with the azimuth measured clockwise from North and elevation up from the zero plane of the surface. By default, the direction of the light source is East (90° azimuth) at an elevation of 45°.

`meshlsrm(Z,refvec,[azim elev],cmap)` displays the regular data grid using the provided colormap. The number of grayscales is chosen to keep the size of the shaded colormap below 256. By default, the colormap is constructed from 16 colors and 16 grays. If the vector of azimuth and elevation is empty, the default locations are used.

`meshlsrm(Z,refvec,[azim elev],cmap,clim)` uses the provided color axis limits, which by default are computed from the data.

`h = meshlsrm(...)` returns the handle to the surface drawn.

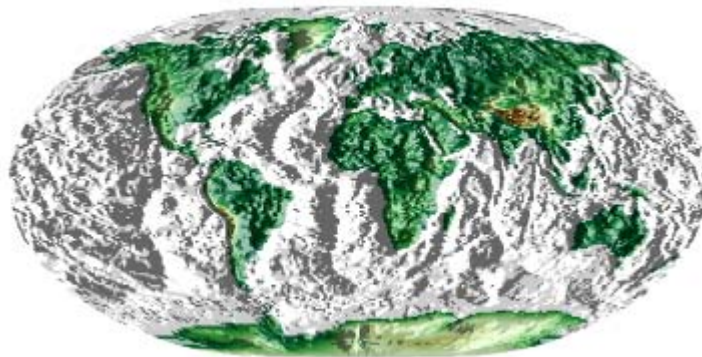
**Remarks**

This function effectively multiplies two colormaps, one with color based on elevation, the other with a grayscale based on the slope of the surface, to create a new colormap. This produces an effect similar to using a light on a surface, but with all of the visible colors actually in the colormap. Lighting calculations are performed on the unprojected data.

## Examples

Create a new colormap using `demcmap`, with white colors for the sea and default colors for land. Use this colormap for a lighted shaded relief map of the world.

```
load topo
[cmap,clim] = demcmap(topo,[],[1 1 1],[]);
axesm loximuth
meshlsrcm(topo,topolegend,[],cmap,clim)
```



## See Also

`meshm`, `pcolorm`, `shaderel`, `surfacem`, `surflm`, `surfm`, `surflsrcm`

**Purpose**

Project regular data grid on map axes

**Syntax**

```
h = meshm(Z,refvec)
h = meshm(Z,refvec,npts)
h = meshm(Z,refvec,npts,alt)
h = meshm(Z,refvec,PropertyName,PropertyValue,...)
```

`h = meshm(Z,refvec)` projects a regular data grid, `Z`, geolocated by a three-element Referencing vector, `refvec`, onto the current map axes. The handle, `h`, of the displayed surface can be returned.

`h = meshm(Z,refvec,npts)` specifies the resolution of the graticule grid. The input `npts` is of the form [latitude-points longitude-points]. The default value of `npts` is [50 100] (the graticule has 50 vertices in the latitude direction and 100 vertices in the longitude direction).

`h = meshm(Z,refvec,npts,alt)` sets the  $z$ -axis altitude of the graticule mesh. `alt` can be a scalar, in which case the map is plotted on a  $z = alt$  plane, or `alt` can be a matrix of size `alt` = `npts`, in which case the graticule mesh is plotted in 3-D.

`h = meshm(Z,refvec,PropertyName,PropertyValue,...)` allows the input of property name/property value pairs to control the surface object properties. Any property supported by the standard MATLAB function `surface` except `XData`, `YData`, and `ZData` can be altered in this manner.

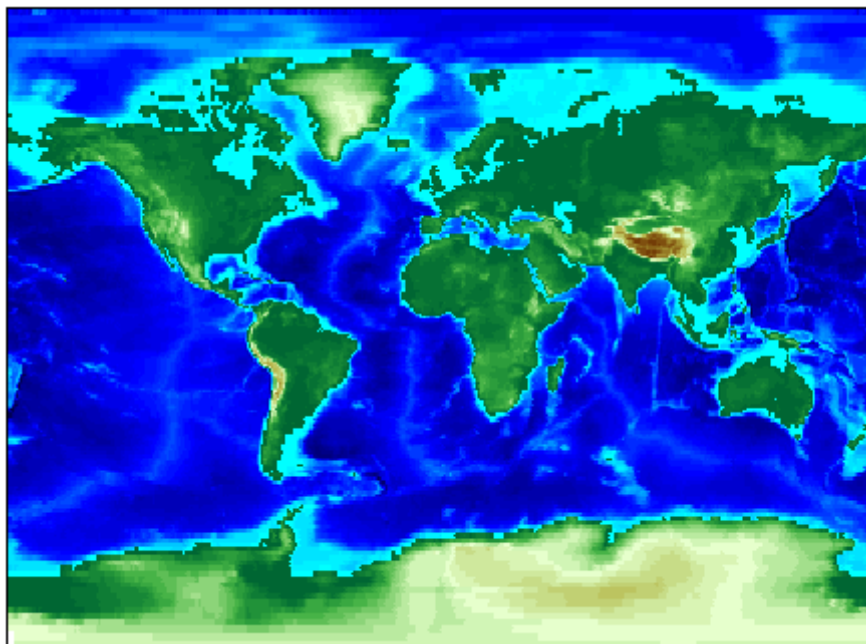
**Description**

The `meshm` function warps a regular data grid to a graticule mesh, which is itself projected according to the `MapProjection` property of the current map axes. The fineness, or resolution, of this grid determines the quality of the projection and the speed of plotting it. There is no hard and fast rule for sufficient graticule resolution, but in general, cylindrical projections need very few graticule points in the longitudinal direction, while complex curve-generating projections require more.

**Examples**

```
load topo
axesm miller
meshm(topo,topolegend,[90 180])
```

demcmap(topo)  
tightmap



**See Also**

geoshow, mapshow, meshgrat, pcolorm, surfacem, surfm

**Purpose**

Project geographic features to map coordinates

**Syntax**

```
[x,y] = mfwdtran(lat,lon)
[x,y,z] = mfwdtran(lat,lon,alt)
[x,y,z,struct] = mfwdtran(lat,lon,alt,object)
[...] = mfwdtran(mstruct,...)
```

`[x,y] = mfwdtran(lat,lon)` transforms unprojected geographic data to a projected  $x$ - $y$  coordinate system using the map projection defined for the current axes. No clipping or trimming of data is performed with this calling form.

`[x,y,z] = mfwdtran(lat,lon,alt)` transforms the three-dimensional data to a projected Cartesian coordinate system using the map projection defined for the current axes. If `alt = []` or `alt` is omitted, the default `alt = 0` is used.

`[x,y,z,struct] = mfwdtran(lat,lon,alt,object)` clips and trims the data during the transformation process. Allowable *object* strings are 'surface', 'line', 'patch', 'light', 'text', and 'none'. 'none' results in no clipping or trimming of the input data. The output `struct` is a structure containing information about the clips and trims associated with the transformed object.

`[...] = mfwdtran(mstruct,...)` requires a valid map projection structure as the first argument. This structure is used to define the map projection calculations performed. No map axes need be displayed when using this calling form.

**Examples**

The following latitude and longitude data for the District of Columbia is obtained from the `usastatelo` demo shapefile:

```
dc = shaperead('usastatelo', 'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'District of Columbia'),...
    'Name'});
lat = [dc.Lat]';
lon = [dc.Lon]';
[lat lon]
```

```
ans =  
    38.9000  -77.0700  
    38.9500  -77.1200  
    39.0000  -77.0300  
    38.9000  -76.9000  
    38.7800  -77.0300  
    38.8000  -77.0200  
    38.8700  -77.0200  
    38.9000  -77.0700  
    38.9000  -77.0500  
    38.9000  -77.0700  
         NaN         NaN
```

Before projecting the data, it is necessary to define projection parameters. You can do this with the `axesm` function or with the `defaultm` function:

```
mstruct = defaultm('mercator');  
mstruct.origin = [38.89 -77.04 0];  
mstruct = defaultm(mstruct);
```

Now that the projection parameters have been set, transform the District of Columbia data into map coordinates using the Mercator projection:

```
[x,y] = mfdtran(mstruct,lat,lon);  
[x y]
```

```
ans =  
   -0.0004    0.0002  
   -0.0011    0.0010  
    0.0001    0.0019  
    0.0019    0.0002  
    0.0001   -0.0019  
    0.0003   -0.0016  
    0.0003   -0.0003  
   -0.0004    0.0002
```



-0.0001	0.0002
-0.0004	0.0002
NaN	NaN

**See Also**

defaultm, gcm, minvtran, projfwd, projinv, vfwdtran, vinvtran

# minaxis

---

## Purpose

Semiminor axis of ellipse given semimajor axis and eccentricity

## Syntax

```
semiminor = minaxis(semimajor, eccentricity)
semiminor = minaxis([semimajor, eccentricity])
```

`semiminor = minaxis(semimajor, eccentricity)` returns the semiminor axis length corresponding to the input semimajor axis and eccentricity.

`semiminor = minaxis([semimajor, eccentricity])` allows the inputs to be packed into a single two-column input of the form `[semimajor, eccentricity]`.

## Description

The semiminor axis can be determined given both the semimajor axis and the eccentricity, the two elements of a standard ellipsoid vector in Mapping Toolbox.

## Examples

Using the default values for the Earth,

```
semiminor = minaxis(almanac('earth', 'ellipsoid'))
semiminor =
    6.3568e+03
```

## See Also

`almanac`, `axes2ecc`, `majaxis`

**Purpose**

Unproject features from map to geographic coordinates

**Syntax**

```
[lat,lon] = minvtran(x,y)
[lat,lon,alt] = minvtran(x,y,z)
[lat,lon,alt] = minvtran(x,y,z,object,struct)
[...] = minvtran(mstruct,...)
```

[lat,lon] = minvtran(x,y) transforms projected  $x$ - $y$  data to an unprojected geographic coordinate system using the map projection defined for the current axes. No data clips or trims are removed with this calling form.

[lat,lon,alt] = minvtran(x,y,z) transforms the three-dimensional data to an unprojected geographic coordinate system using the map projection defined for the current axes. If  $z = []$  or  $z$  is omitted, the default  $z = 0$  is used.

[lat,lon,alt] = minvtran(x,y,z,object,struct) removes all clips and trims from the input data. Allowable *object* strings are 'surface', 'line', 'patch', 'light', 'text', and 'none'. 'none' results in no removal of any clips or trims of the input data. The output struct is a structure containing information about the clips and trims associated with the transformed object, and is created by the function `mfwdtran`.

[...] = minvtran(mstruct,...) requires a valid map projection structure as the first argument. This structure is used to define the map projection calculations performed. No map axes need be displayed when using this calling form.

**Examples**

Before using any transformation functions, it is necessary to create a map projection structure. You can do this with `axesm` or the `defaultm` function:

```
mstruct = defaultm('mercator');
mstruct.origin = [38.89 -77.04 0];
mstruct = defaultm(mstruct);
```

The following latitude and longitude data for the District of Columbia is obtained from the `usastatelo` shapefile:

```
dc = shaperead('usastatelo', 'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'District of Columbia'),...
    'Name'});
lat = [dc.Lat]';
lon = [dc.Lon]';
[lat lon]
```

```
ans =
    38.9000  -77.0700
    38.9500  -77.1200
    39.0000  -77.0300
    38.9000  -76.9000
    38.7800  -77.0300
    38.8000  -77.0200
    38.8700  -77.0200
    38.9000  -77.0700
    38.9000  -77.0500
    38.9000  -77.0700
         NaN         NaN
```

This data can be projected into Cartesian coordinates of the Mercator projection using the `mfwdtran` function:

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

```
ans =
   -0.0004    0.0002
   -0.0011    0.0010
    0.0001    0.0019
    0.0019    0.0002
    0.0001   -0.0019
    0.0003   -0.0016
    0.0003   -0.0003
   -0.0004    0.0002
   -0.0001    0.0002
   -0.0004    0.0002
```

```
NaN NaN
```

To transform the projected *x-y* data back into the unprojected geographic system, use the `minvtran` function:

```
[lat2,lon2] = minvtran(mstruct,x,y);  
[lat2 lon2]
```

```
ans =  
38.9000 -77.0700  
38.9500 -77.1200  
39.0000 -77.0300  
38.9000 -76.9000  
38.7800 -77.0300  
38.8000 -77.0200  
38.8700 -77.0200  
38.9000 -77.0700  
38.9000 -77.0500  
38.9000 -77.0700  
NaN NaN
```

## See Also

`axesm`, `defaultm`, `gcm`, `mfwdtran`, `projfwd`, `projinv`, `vfwdtran`, `vinvtran`

# mlabel

---

**Purpose** Toggle and control display of meridian labels

**Syntax**

```
mlabel  
mlabel('on')  
mlabel('off')  
mlabel('reset')  
mlabel(parallel)  
mlabel(MapAxesPropertyName, PropertyValue, ...)
```

mlabel toggles the visibility of meridian labeling on the current map axes.

mlabel('on') sets the visibility of meridian labels to 'on'.

mlabel('off') sets the visibility of meridian labels to 'off'.

mlabel('reset') resets the displayed meridian labels using the currently defined meridian label properties.

mlabel(parallel) sets the value of the MLabelParallel property of the map axes to the value of parallel. This determines the parallel upon which the labels are placed (see axesm). The options for parallel are a scalar latitude or the strings 'north', 'south', or 'equator'.

mlabel(*MapAxesPropertyName*, *PropertyValue*, ...) allows paired map axes' property names and property values to be passed in. For a complete description of map axes properties, see the axesm reference page in this guide.

Meridian label handles can be returned in h if desired.

**See Also** axesm, mlabelzero22pi, plabel, setm

## Purpose

Convert meridian labels to 0-360 degree range

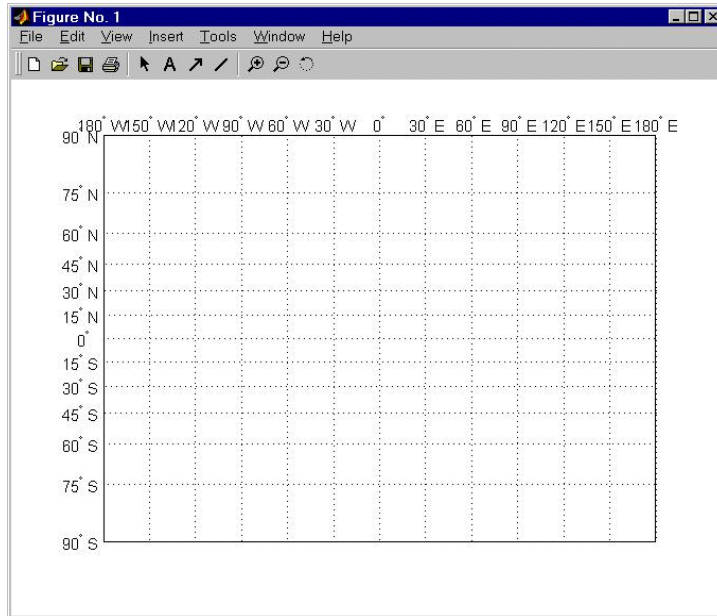
## Syntax

`mlabelzero22pi`

`mlabelzero22pi` displays longitude labels in the range of 0 to 360 degrees east of the prime meridian.

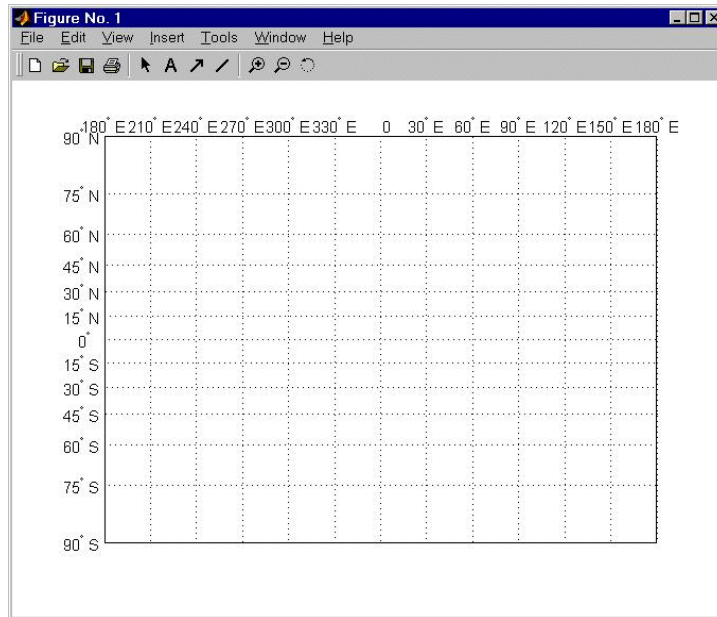
## Example

```
% create a map
figure('color','w'); axesm('miller','grid','on'); tightmap;
mlabel on; plabel on
```



```
% Display longitude labels in the range of 0 to 360 degrees
mlabelzero22pi
```

# mlabelzero22pi



**See Also**

mlabel



---

<b>Purpose</b>	Eccentricity of ellipse with given <i>n</i> -value
<b>Syntax</b>	<pre>eccentricity = n2ecc(n)</pre> <p><code>eccentricity = n2ecc(n)</code> returns the equivalent eccentricities for the input <i>n</i> parameters. If the input <i>n</i> is a two-column vector, only the second column is used. This allows two-element vectors to be used as rows of the input, because the form [semimajor-axis, <i>n</i>] is a complete representation of an ellipsoid (but is not the standard form for ellipsoid vectors in Mapping Toolbox). In all other cases, all columns of the input are used.</p>
<b>Description</b>	<p>Eccentricity and the parameter <i>n</i> are two methods of defining an ellipsoid. The definition of <i>n</i> is</p> $(\text{semimajor axis} - \text{semiminor axis}) / (\text{semimajor axis} + \text{semiminor axis})$
<b>Example</b>	<pre>ecc = n2ecc(0.00167922039463)</pre> <pre>ecc =     0.08181919104285</pre> <p>This eccentricity is the default value for the Earth.</p>
<b>See Also</b>	<code>almanac</code> , <code>ecc2flat</code> , <code>majaxis</code> , <code>ecc2n</code>

# namem

---

## **Purpose**

Determine names of valid graphics objects

## **Syntax**

```
objects = namem  
objects = namem(handles)
```

`objects = namem` returns the object names for all objects on the current axes. The object name is defined as its tag, if the object `Tag` property is supplied. Otherwise, it is the object `Type`. Duplicate object names are removed from the output string matrix.

`objects = namem(handles)` returns the object names for the objects specified by the input `handles`.

The names returned are either set at object creation or defined by the user with the `tagm` function.

## **See Also**

`clma`, `clmo`, `handlem`, `hidem`, `showm`, `tagm`

**Purpose** Clip vector data with NaNs at specified pen-down locations

**Syntax**

```
dataout = nanclip(datain)
dataout = nanclip(datain,pendowncmd)

dataout = nanclip(datain) and dataout =
nanclip(datain,pendowncmd) return the pen-down
delimited data in the matrix datain as NaN-delimited data in dataout.
When the first column of datain equals pendowncmd, a segment is
started and a NaN is inserted in all columns of dataout. The default
pendowncmd is -1.
```

**Description** Pen-down delimited data is a matrix with a first column consisting of pen commands. At the beginning of each segment in the data, this first column has an entry corresponding to a pen-down command. Other entries indicate that the segment is continuing. NaN-delimited data consists of columns of data, each segment of which ends in a NaN in every data column. Since there is no pen command column, the NaN-delimited format can represent the same data in one fewer columns; the remaining columns have more entries, one for each NaN (that is, for each segment).

**Examples**

```
datain = [-1 45 67; 0 23 54; 0 28 97; -1 47 89; 0 56 12]

datain =
    -1    45    67           % Begin first segment
     0    23    54
     0    28    97
    -1    47    89           % Begin second segment
     0    56    12

dataout = nanclip(datain)

dataout =
    45    67
    23    54
    28    97
   NaN   NaN           % End first segment
```

# nanclip

---

```
47 89
56 12
NaN NaN % End second segment
```

**See Also** `spread`

**Purpose** Construct regular data grid of NaNs

**Syntax** `[Z,refvec] = nanm(latlim,lonlim,scale)`  
`[Z,refvec] = nanm(latlim,lonlim,scale)` returns a regular data grid consisting entirely of NaNs and a three-element referencing vector for the returned Z. The two-element vectors `latlim` and `lonlim` define the latitude and longitude limits of the geographic region. They should be of the form `[south north]` and `[west east]`, respectively. The scalar `scale` specifies the number of rows and columns per degree of latitude and longitude.

**Example** `[Z,refvec] = nanm([46,51],[-79,-75],1)`

```
Z =  
    NaN    NaN    NaN    NaN  
    NaN    NaN    NaN    NaN  
    NaN    NaN    NaN    NaN  
    NaN    NaN    NaN    NaN  
    NaN    NaN    NaN    NaN  
refvec =  
     1     51    -79
```

**See Also** `limitm`, `onem`, `szem`, `spzerom`, `zerom`

## Purpose

Mercator-based navigational fix

## Syntax

```
[latfix,lonfix] = navfix(lat,long,az)
[latfix,lonfix] = navfix(lat,long,range,casetype)
[latfix,lonfix] = navfix(lat,long,az_range,casetype)
[latfix,lonfix] = navfix(lat,long,az_range,casetype,drlat,
    drlon)
```

[latfix,lonfix] = navfix(lat,long,az) returns the intersection points of rhumb lines drawn parallel to the observed bearings, az, of the landmarks located at the points lat and long and passing through these points. One bearing is required for each landmark. Each possible pairing of the  $n$  landmarks generates one intersection, so the total number of resulting intersection points is the combinatorial  $n$  choose 2. The calculation time therefore grows rapidly with  $n$ .

[latfix,lonfix] = navfix(lat,long,range,casetype) returns the intersection points of Mercator projection circles with radii defined by range, centered on the landmarks located at the points lat and long. One range value is required for each landmark. Each possible pairing of the  $n$  landmarks generates up to two intersections (circles can intersect twice), so the total number of resulting intersection points is the combinatorial 2 times ( $n$  choose 2). The calculation time therefore grows rapidly with  $n$ . In this case, the variable casetype is a vector of 0s the same size as the variable range.

[latfix,lonfix] = navfix(lat,long,az\_range,casetype) combines ranges and bearings. For each element of casetype equal to 1, the corresponding element of az\_range represents an azimuth to the associated landmark. Where casetype is a 0, az\_range is a range.

[latfix,lonfix] = navfix(lat,long,az\_range,casetype,drlat,drlon) returns for each possible pairing of landmarks only the intersection that lies closest to the dead reckoning position indicated by drlat and drlon. When this syntax is used, all included landmarks' bearing lines or range arcs must intersect. If any possible pairing fails, the warning No Fix is displayed.

## Background

This is a navigational function. It assumes that all latitudes and longitudes are in degrees and all distances are in nautical miles. In navigation, piloting is the practice of fixing one's position based on the observed bearing and ranges *to* fixed landmarks (points of land, lighthouses, smokestacks, etc.) *from* the navigator's vessel. In conformance with navigational practice, bearings are treated as rhumb lines and ranges are treated as the radii of circles on a Mercator projection.

In practice, at least three azimuths (bearings) and/or ranges are required for a usable fix. The resulting intersections are unlikely to coincide exactly. Refer to "Navigation" on page 9-11 in the *Mapping Toolbox User's Guide* for a more complete description of the use of this function.

## Remarks

The outputs of this function are matrices providing the locations of the intersections for all possible pairings of the  $n$  entered lines of bearing and range arcs. These matrices therefore have  $n$ -choose-2 rows. In order to allow for two intersections per combination, these matrices have two columns. Whenever there are fewer than two intersections for that combination, one or two NaNs are returned in that row.

When a dead reckoning position is included, these matrices are column vectors.

## Examples

For a fully illustrated example of the application of this function, refer to the "Navigation" on page 9-11 section in the *Mapping Toolbox User's Guide*.

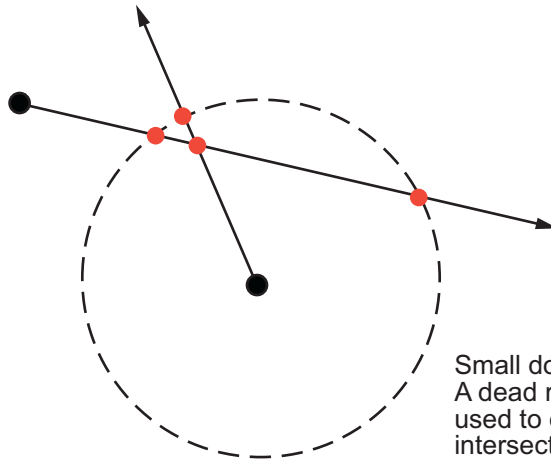
Imagine you have two landmarks, at (15°N,30.4°W) and (14.8°N,30.1°W). You have a visual bearing to the first of 280° and to the second of 160°. Additionally, you have a range to the second of 12 nm. Find the intersection points:

```
[latfix,lonfix] = navfix([15 14.8 14.8],[ -30.4 -30.1 -30.1],...
                        [280 160 12],[1 1 0])
```

```
latfix =
```

```
14.9591      NaN
14.9680    14.9208
14.9879      NaN
lonfix =
-30.1599      NaN
-30.2121   -29.9352
-30.1708      NaN
```

Here is an illustration of the geometry:



Small dots are intersection points. A dead reckoning position could be used to eliminate the inconsistent intersection.

## Limitations

Traditional plotting and the `navfix` function are limited to relatively short distances. Visual bearings are in fact great circle azimuths, not rhumb lines, and range arcs are actually arcs of small circles, not of the planar circles plotted on the chart. However, the mechanical ease of the process and the practical limits of visual bearing ranges and navigational radar ranges (~30 nm) make this limitation moot in practice. The error contributed because of these assumptions is minuscule at that scale.

## See Also

`crossfix`, `gcxgc`, `gcxsc`, `scxsc`, `rhxrh`, `polyxpoly`, `dreckon`, `gwaypts`, `legs`, `track`



**Purpose**

Orient regular data grid to oblique aspect

**Syntax**

```
[Z,lat,lon] = neworig(Z0,refvec,origin)
[Z,lat,lon] = neworig(Z0,refvec,origin,direction)
[Z,lat,lon] = neworig(Z0,refvec,origin,direction,units)
```

[Z,lat,lon] = neworig(Z0,refvec,origin) returns the data in the original regular data grid Z0, with its three-element referencing vector refvec, reallocated to the cells of the new (same-sized) data grid. This transformation is governed by the input origin. This is a three- (or two-) element vector of the form [latitude longitude orientation]. The latitude and longitude are the coordinates of the point in the original system that is the center of the output system. The orientation is the azimuth from the new origin point to the original North Pole in the new system. If origin has only two elements, the orientation is assumed to be 0°. This origin vector might be the output of putpole or newpole. The outputs lat and lon are matrices the size of Z that give a cell-by-cell registration of Z to the coordinates of the original (Z0) system in latitude and longitude, respectively.

[Z,lat,lon] = neworig(Z0,refvec,origin,direction) allows the specification of the operation. If the string *direction* is 'forward' (the default), the transformation occurs as described above. If the *direction* is 'inverse', then the output Z is the *original* system from which a transformed matrix grid0 was derived, via the input origin. Note that if the matrix Z1 is transformed forward to Z2, and Z2 is transformed inversely to Z3, Z3 will look very much like grid1, but the two matrices will not be identical. This is because neworig is in fact projecting the values of the cells twice, rather than *undoing* the first transformation, and matrix data has granularity.

[Z,lat,lon] = neworig(Z0,refvec,origin,direction,units) allows the specification of the angular units of the origin vector, where *units* is any valid angle units string. The default is 'degrees'.

**Description**

The neworig function transforms a regular data grid into a new matrix in an altered coordinate system. An analytical use of the new matrix can be realized in conjunction with the newpole function. If a selected

point is made the *north pole* of the new system, then when a new matrix is created with `neworig`, each row of the new matrix is a constant distance from the selected point, and each column is a constant azimuth from that point.

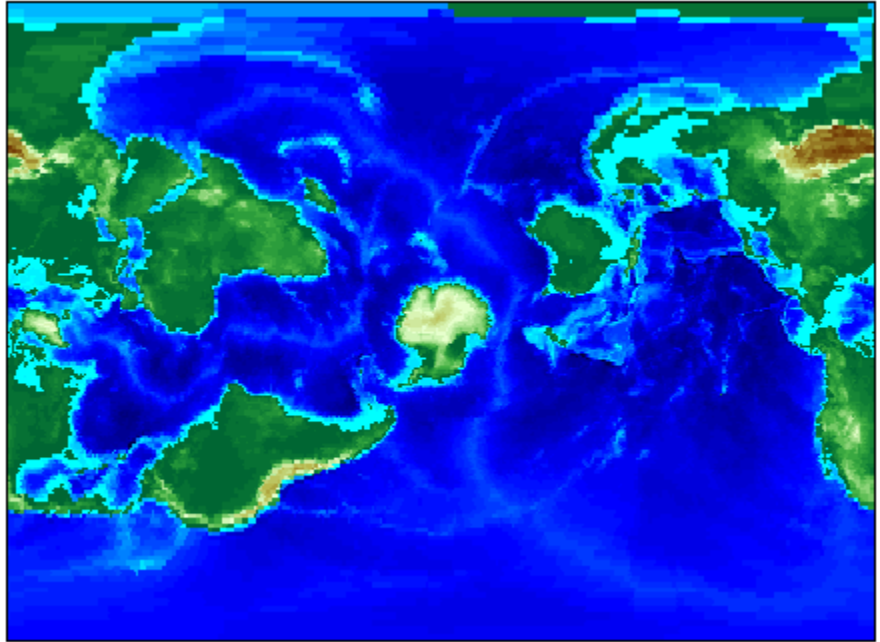
## Limitations

`neworig` only supports data grids that cover the entire globe.

## Example

This is the topo map transformed to put Sri Lanka at the North Pole:

```
load topo
origin = newpole(7,80)
origin =
    83.0000 -100.0000      0
[Z,lat,lon] = neworig(topo,topolegend,origin);
axesm miller
surfm(Z,[30 30])
demcmap(topo)
tightmap
```



**See Also**

`newpole`, `org2pol`, `putpole`, `rotatem`

# newpole

---

## Purpose

Origin vector to place specific point at pole

## Syntax

```
origin = newpole(polelat,polelon)
origin = newpole(polelat,polelon,units)
```

`origin = newpole(polelat,polelon)` provides the origin vector for a transformed coordinate system based upon moving the point (`polelat`, `polelon`) to become the north pole singularity in the new system. The origin is a three-element vector of the form [latitude longitude orientation], where the latitude and longitude are the coordinates the new center (`origin`) had in the untransformed system, and the orientation is the azimuth of the true North Pole from the new origin point. For the `newpole` calculation, this orientation is constrained to be always 0°.

`origin = newpole(polelat,polelon,units)` specifies the units of the inputs and output, where *units* is any valid angle units string. The default is 'degrees'.

## Description

When developing transverse or oblique projections, you need transformed coordinate systems. One way to define these systems is to establish the point in the original (untransformed) system that will become the new (transformed) *north pole*.

## Examples

Take a point and make it the new North Pole:

```
origin = newpole(60,180)

origin =
    30.0000         0         0
```

This makes sense: as a point 30° beyond the true North Pole on the original origin's meridian is pulled up to become the *pole*, the point originally 30° above the origin is pulled down into the origin spot.

## See Also

`neworig`, `org2pol`, `putpole`

**Purpose**

Add graphic element pointing to geographic north pole

**Syntax**

```
northarrow  
northarrow('property',value,...)
```

northarrow creates a default north arrow.

northarrow('property',value,...) creates a north arrow using the specified property/value pairs. Valid entries for properties are 'latitude', 'longitude', 'facecolor', 'edgecolor', 'linewidth', and 'scaleratio'. The 'latitude' and 'longitude' properties specify the location of the north arrow. The 'facecolor', 'edgecolor', and 'linewidth' properties control the appearance of the north arrow. The 'scaleratio' property represents the size of the north arrow as a fraction of the size of the axes. A 'scaleratio' value of 0.10 creates a north arrow one-tenth (1/10) the size of the axes. You can change the appearance ('facecolor', 'edgecolor', and 'linewidth') of the north arrow using the set command.

**Description**

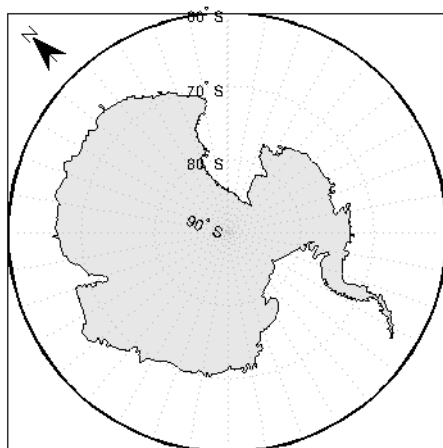
northarrow creates a north arrow symbol at the map origin on the displayed map. You can reposition the north arrow symbol by clicking and dragging its icon. Alternate clicking the icon creates an input dialog box that you can also use to change the location of the north arrow.

Modifying some of the properties of the north arrow results in replacement of the original object. Use HANDLEM('NorthArrow') to get the handles associated with the north arrow.

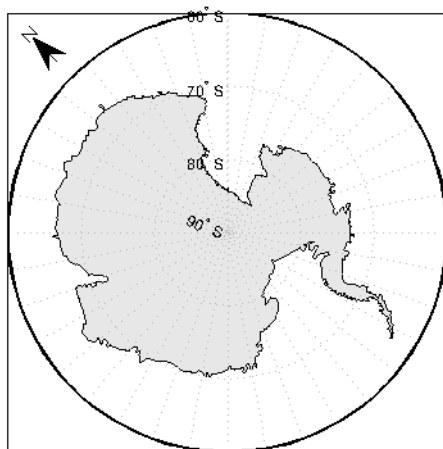
**Examples**

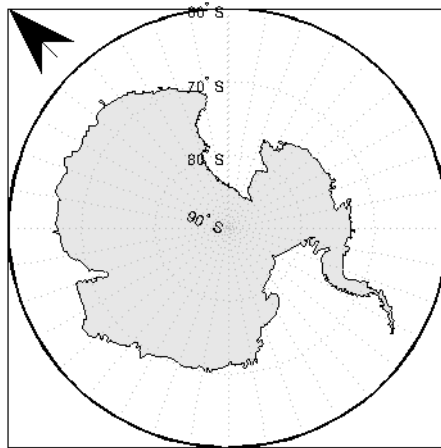
Create a map of the South Pole and then add the north arrow in the upper left of the map.

```
Antarctica = shaperead('landareas', 'UseGeoCoords', true, ...  
    'Selector',{@(name) strcmpi(name,{'Antarctica'})}, 'Name');  
figure;  
worldmap('south pole')  
geoshow(Antarctica,'FaceColor',[.9 .9 .9])  
northarrow('latitude', -57, 'longitude', 135);
```



Right-click the north arrow icon to activate the input dialog box. Increase the size of the north arrow symbol by changing the 'ScaleRatio' property.



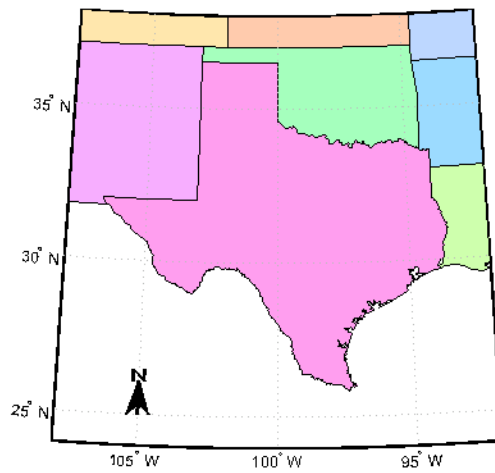


Create a map of Texas and add the north arrow in the lower left of the map.

```
figure; usamap('texas')
states = shaperead('usastatelo.shp','UseGeoCoords',true);
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))});
geoshow(states, 'DisplayType', 'polygon', ...
    'SymbolSpec', faceColors)
northarrow('latitude',25,'longitude',-105,'linewidth',1.5);
```

# northarrow

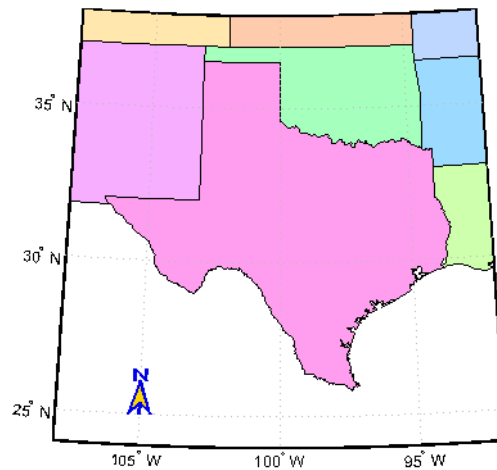
---



Change the 'FaceColor' and 'EdgeColor' properties of the north arrow.

```
h = handle('NorthArrow');  
set(h,'FaceColor',[1.000 0.8431 0.0000],...  
    'EdgeColor',[0.0100 0.0100 0.9000])
```



**Limitations**

You can draw multiple north arrows on the map. However, the callbacks will only work with the most recently created north arrow. In addition, since it can be displayed outside the map frame limits, the north arrow is not converted into a “mapped” object. Hence, the location and orientation of the north arrow have to be updated manually if the map origin or projection changes.

**See Also**

`scaleruler`

# npi2pi

---

## Purpose

Wrap longitudes to [-180 180] degree interval

---

**Note** The `npi2pi` function has been replaced by `wrapTo180` and `wrapToPi`.

---

## Syntax

```
anglout = npi2pi(anglin)
anglout = npi2pi(anglin,units)
anglout = npi2pi(anglin,units,method)
```

`anglout = npi2pi(anglin)` wraps the input angle `anglin` (typically representing a longitude) to lie on the range -180 to 180 (e.g., 270° is renamed -90°).

`anglout = npi2pi(anglin,units)` specifies the angle units with any valid angle units string `units`. The default is 'degrees'.

`anglout = npi2pi(anglin,units,method)` allows special alternative computations to be used when `npi2pi` is called from within certain Mapping Toolbox functions. `method` can be one of the following strings:

- 'exact', for exact wrapping (the default value)
- 'inward', where angles are scaled by a factor of  $(1 - \text{epsm}('radians'))$  before wrapping
- 'outward', where angles are scaled by a factor of  $(1 + \text{epsm}('radians'))$  before wrapping

## Examples

```
npi2pi(315)
```

```
ans =
    -45
```

```
npi2pi(181)
```

```
ans =
   -179
```

**See Also**      `wrapToPi`, `wrapTo180`

**Purpose** Construct regular data grid of 1s

**Syntax** `[Z,refvec] = onem(latlim,lonlim,scale)`

`[Z,refvec] = onem(latlim,lonlim,scale)` returns a regular data grid consisting entirely of 1s and a three-element referencing vector for the returned data grid, Z.. The two-element vectors `latlim` and `lonlim` define the latitude and longitude limits of the geographic region. They should be of the form `[south north]` and `[west east]`, respectively. The scalar `scale` specifies the number of rows and columns per degree of latitude and longitude.

**Examples** `[Z,refvec] = onem([46,51],[-79,-75],1)`

```
Z =
    1     1     1     1
    1     1     1     1
    1     1     1     1
    1     1     1     1
    1     1     1     1
refvec =
    1    51   -79
```

**See Also** `limitm`, `nanm`, `sizem`, `spzerom`, `zerom`

**Purpose**

Location of north pole in rotated map

**Syntax**

```
pole = org2pol(origin)
pole = org2pol(origin, units)
```

`pole = org2pol(origin)` returns the location of the North Pole in terms of the coordinate system after transformation based on the input `origin`. The `origin` is a three-element vector of the form `[latitude longitude orientation]`, where `latitude` and `longitude` are the coordinates that the new center (`origin`) had in the untransformed system, and `orientation` is the azimuth of the true North Pole from the new origin point in the transformed system. The output `pole` is a three-element vector of the form `[latitude longitude meridian]`, which gives the latitude and longitude point in terms of the original untransformed system of the new location of the true North Pole. The `meridian` is the longitude from the original system upon which the new system is centered.

`pole = org2pol(origin, units)` allows the specification of the angular units of the origin vector, where *units* is any valid angle units string. The default is 'degrees'.

**Description**

When developing transverse or oblique projections, transformed coordinate systems are required. One way to define these systems is to establish the point at which, in terms of the original (untransformed) system, the (transformed) true North Pole will lie.

**Examples**

Perhaps you want to make (30°N,0°) the new origin. Where does the North Pole end up in terms of the original coordinate system?

```
pole = org2pol([30 0 0])

pole =
    60.0000         0         0
```

This makes sense: pull a point 30° down to the origin, and the North Pole is pulled down 30°. A little less obvious example is the following:

# org2pol

---

```
pole = org2pol([5 40 30])
```

```
pole =  
    59.6245    80.0750    40.0000
```

**See Also**      neworig, putpole

**Purpose**

Set figure properties for printing at specified map scale

**Syntax**

```
paperscale(paperdist,punits,surfdist,sunits)
paperscale(paperdist,punits,surfdist,sunits,lat,long)
paperscale(paperdist,punits,surfdist,sunits,lat,long,az)
paperscale(paperdist,punits,surfdist,sunits,lat,long,az,
           gunits)
paperscale(paperdist,punits,surfdist,sunits,lat,long,az,gunits,
           radius)
paperscale(scale,...)
[paperXdim,paperYdim] = paperscale(...)
```

`paperscale(paperdist,punits,surfdist,sunits)` sets the figure paper position to print the map in the current axes at the desired scale. The scale is described by the geographic distance that corresponds to a paper distance. For example, a scale of 1 inch = 10 kilometers is specified as `degrees(1,'inch',10,'km')`. See below for an alternate method of specifying the map scale. The surface distance units string *sunits* can be any string recognized by `unitsratio`. The paper units string *punits* can be any dimensional units string recognized for the figure `PaperUnits` property.

`paperscale(paperdist,punits,surfdist,sunits,lat,long)` sets the paper position so that the scale is correct at the specified geographic location. If omitted, the default is the center of the map limits.

`paperscale(paperdist,punits,surfdist,sunits,lat,long,az)` also specifies the direction along which the scale is correct. If omitted, 90 degrees (east) is assumed.

`paperscale(paperdist,punits,surfdist,sunits,lat,long,az,gunits)` also specifies the units in which the geographic position and direction are given. If omitted, 'degrees' is assumed.

`paperscale(paperdist,punits,surfdist,sunits,lat,long,az,gunits,radius)` uses the last input to determine the radius of the sphere. If `radius` is a string, then it is evaluated as an almanac body to determine the spherical radius. If numerical, it is the radius of the desired

sphere in the same units as the surface distance. If omitted, the default radius of the Earth is used.

`paperscale(scale, ...)`, where the numeric scale replaces the two property/value pairs, specifies the scale as a ratio between distance on the sphere and on paper. This is commonly notated on maps as 1:scale (e.g. 1:100 000, or 1:1 000 000). For example, `paperscale(100000)` or `paperscale(100000,lat,long)`.

`[paperXdim,paperYdim] = paperscale(...)` returns the computed paper dimensions. The dimensions are in the paper units specified. For the scale calling form, the returned dimensions are in centimeters.

## Background

Maps are usually printed at a size that allows an easy comparison of distances measured on paper to distances on the Earth. The relationship of geographic distance and paper distance is termed *scale*. It is usually expressed as a ratio, such as 1 to 100,000 or 1:100,000 or 1 cm = 1 km.

## Examples

The small circle measures 10 cm across when printed.

```
axesm mercator
[lat,lon] = scircle1(0,0,km2deg(5));
plotm(lat,lon)
[x,y] = paperscale(1,'centimeter',1,'km'); [x y]

ans =
    13.154    12.509

set(gca,'pos',[ 0 0 1 1])
[x,y] = paperscale(1,'centimeter',1,'km'); [x y]

ans =
    10.195    10.195
```

## Limitations

The relationship between the paper and geographic coordinates holds only as long as there are no changes to the display that affect the axes limits or the relationship between geographic coordinates and projected coordinates. Changes of this type include the ellipsoid or scale factor



properties of the map axes, or adding elements to the display that cause MATLAB to modify the axes autoscaling. To be sure that the scale is correct, execute `paperscale` just before printing.

## See Also

`pagesetupdlg`, `axesscale`, `daspectm`

# patchesm

---

## Purpose

Project patches on map axes as individual objects

## Syntax

```
patchesm(lat,lon,cdata)
patchesm(lat,lon,z,cdata)
patchesm(...,'PropertyName',PropertyValue,...)
h = patchesm(...)
```

`patchesm(lat,lon,cdata)` projects 2-D patch objects onto the current map axes. The input latitude and longitude data must be in the same units as specified in the current map axes. The input `cdata` defines the patch face color. If the input vectors are NaN clipped, then multiple patches are drawn each with a single face. Unlike `fillm` and `fill3m`, `patchesm` will always add the patches to the current map regardless of the current hold state.

`patchesm(lat,lon,z,cdata)` projects 3-D planar patches at the uniform elevation given by scalar `z`.

`patchesm(...,'PropertyName',PropertyValue,...)` uses the patch properties supplied to display the patch. Except for `xdata`, `ydata`, and `zdata`, all patch properties available through `patch` are supported by `patchesm`.

`h = patchesm(...)` returns the handles to the patch objects drawn.

## Remarks

### Differences between patchesm and patchm

The `patchesm` function is very similar to the `patchm` function. The significant difference is that in `patchesm`, separate patches (delineated by NaNs in the inputs `lat` and `lon`) are separated and plotted as distinct patch objects on the current map axes. The advantage to this is that less memory is required. The disadvantage is that multifaced objects cannot be treated as a single object. For example, the archipelago of the Philippines cannot be treated and handled as a single Handle Graphics object.

### When Patches Are Completely Trimmed Away

Removing graphic objects that fall outside the map frame is called trimming. If, after trimming no polygons remain to be seen within

it, patchesm creates no patches and returns an empty 1-by-0 list of handles. When this occurs, automatic reprojection of the patch data (by changing the projection or any of its parameters) is not possible. In cases where some polygons are completely trimmed away but not others, handles returned for the trimmed polygons will be empty. No polygons or rings that have been totally trimmed away can be reprojected; to plot them again, you will need to call patchesm again with the original data.

## Examples

```
load coast
axesm sinusoid; framem
h = patchesm(lat,long,'b');
```



```
length(h)

ans =
    238
```

## See Also

geoshow, fill13m, fill1m, patchm

## Purpose

Project patch objects on map axes

## Syntax

```
h = patchm(lat,lon,cdata)
h = patchm(lat,lon,cdata,PropertyName,PropertyValue,...)
h = patchm(lat,lon,PropertyName,PropertyValue,...)
h = patchm(lat,lon,z,cdata)
h = patchm(lat,lon,z,cdata, PropertyName,PropertyValue,...)
```

`h = patchm(lat,lon,cdata)` and `h = patchm(lat,lon,cdata,PropertyName,PropertyValue,...)` project and display patch (polygon) objects defined by their vertices given in `lat` and `lon` on the current map axes. `lat` and `lon` must be vectors. The color data, `cdata`, can be any color data designation supported by the standard MATLAB patch function. The object handle or handles, `h`, can be returned.

`h = patchm(lat,lon,PropertyName,PropertyValue,...)` allows any property name/property value pair supported by patch to be assigned to the patchm object.

`h = patchm(lat,lon,z,cdata)` and `h = patchm(lat,lon,z,cdata,PropertyName,PropertyValue,...)` allow the assignment of an altitude, `z`, to each patch object. The default altitude is `z = 0`.

## Remarks

### How patchm Works

This Mapping Toolbox function is very similar to the standard MATLAB patch function. Like its analog, and unlike higher level functions such as `fillm` and `fill3m`, `patchm` adds patch objects to the current map axes regardless of hold state. Except for `XData`, `YData`, and `ZData`, all line properties and styles available through patch are supported by `patchm`.

### When A Patch Is Completely Trimmed Away

Removing graphic objects that fall outside the map frame is called trimming. If, after trimming to the map frame no polygons remain to be seen within it, `patchm` creates no patches and returns an empty 0-by-1 handle. When this occurs, automatic reprojection of the patch data (by changing the projection or any of its parameters) will not be possible. Instead, after changing the projection, call `patchm` again.

**Examples**

```
load coast
axesm sinusoid; framem
h = patchm(lat,long,'b');
```



```
length(h)
```

```
ans =  
    1
```

**See Also**

```
patchesm, fill3m, fillm
```

## Purpose

Project regular data grid on map axes in  $z=0$  plane

## Syntax

```
h = pcolorm(Z)
h = pcolorm(Z,npts)
h = pcolorm(lat,lon,Z)
h = pcolorm(lat,lon,Z,PropertyName,PropertyValue,...)
```

`h = pcolorm(Z)` projects the data grid `Z` on a graticule grid the size of `Z` between the latitude and longitude limits of the current map axes. The handle `h` of the displayed surface can be returned.

`h = pcolorm(Z,npts)` results in a graticule grid defined by `npts`, which is a two-element vector of the form `[latitude-points longitude-points]`. The default `npts` is `[50 100]`.

`h = pcolorm(lat,lon,Z)` allows three other methods of defining the graticule grid. If `lat` and `lon` are matrices, they represent the actual graticule vertices as might be returned by `meshgrat`. If vectors, they are the representative coordinates of the rows and columns, respectively, of such a grid. If they are two-element vectors, they are treated as latitude and longitude limits, and a graticule mesh the size of the default `npts` is calculated.

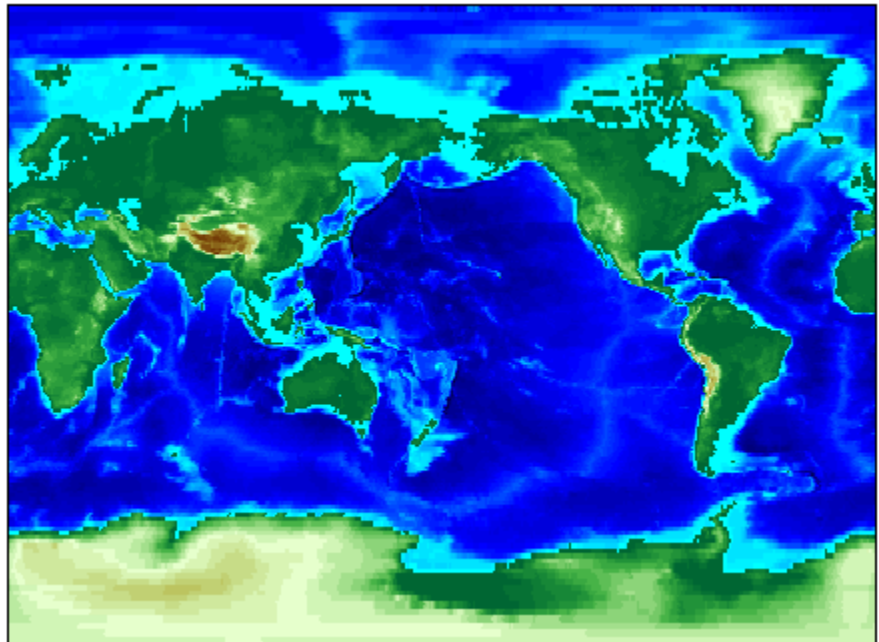
`h = pcolorm(lat,lon,Z,PropertyName,PropertyValue,...)` allows the input of property name/property value pairs to control the surface object properties. Any property supported by the standard MATLAB function `surface` except `XData`, `YData`, and `ZData` can be altered in this manner.

## Remarks

This function warps a data grid to a graticule mesh, which itself is projected according to the map axes property `MapProjection`. The fineness, or resolution, of this grid determines the quality of the projection and the speed of plotting it. There is no hard and fast rule for sufficient graticule resolution, but in general, cylindrical projections need fewer graticule points in the longitudinal direction than do complex curve-generating projections.

**Examples**

```
load topo
axesm miller
pcolorm(topo,[30 30])
demcmap(topo)
tightmap
```

**See Also**

meshgrat, meshm, surfacem, surfm

# pix2latlon

---

## Purpose

Convert pixel coordinates to latitude-longitude coordinates

## Syntax

```
[lat, lon] = pix2latlon(r,row,col)
```

[lat, lon] = pix2latlon(r,row,col) calculates latitude-longitude coordinates lat, lon from pixel coordinates row, col. r is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. row and col are vectors or arrays of matching size. The outputs lat and lon have the same size as row and col.

## Example

```
% Find the lat and lon of the upper left and lower right
% outer corners of a 2-by-2 degree gridded data set.
R = makerefmat(1, 89, 2, 2);
[UL_lat, UL_lon] = pix2latlon(R, .5, .5)

UL_lat =
    88
UL_lon =
    0

[LR_lat, LR_lon] = pix2latlon(R, 90.5, 180.5)

LR_lat =
   268
LR_lon =
   360
```

## See Also

latlon2pix, makerefmat, pix2map



**Purpose**

Convert pixel coordinates to map coordinates

**Syntax**

```
[x,y] = pix2map(R,row,col)
s = pix2map(R,row,col)
[... ] = pix2map(R,p)
```

`[x,y] = pix2map(R,row,col)` calculates map coordinates `x,y` from pixel coordinates `row,col`. `R` is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. `row` and `col` are vectors or arrays of matching size. The outputs `x` and `y` have the same size as `row` and `col`.

`s = pix2map(R,row,col)` combines `X` and `Y` into a single array `s`. If `row` and `col` are column vectors of length `n`, then `s` is an `n`-by-2 matrix and each row (`s(k,:)`) specifies the map coordinates of a single point. Otherwise, `s` has size `[size(row) 2]`, and `s(k1,k2,...,kn,:)` contains the map coordinates of a single point.

`[... ] = pix2map(R,p)` combines `row` and `col` into a single array `p`. If `row` and `col` are column vectors of length `n`, then `p` should be an `n`-by-2 matrix such that each row (`p(k,:)`) specifies the pixel coordinates of a single point. Otherwise, `p` should have size `[size(row) 2]`, and `p(k1,k2,...,kn,:)` should contain the pixel coordinates of a single point.

**Example**

```
% Find the map coordinates for the pixel at (100,50).
R = worldfileread('concord_ortho_w.tfw');
[x,y] = pix2map(R,100,50)

x =
    2.0704950000000000e+005
y =
    9.1290050000000000e+005
```

**See Also**

`makerefmat`, `map2pix`, `pix2latlon`, `worldfileread`

## Purpose

Compute pixel centers for georeferenced image or data grid

## Syntax

```
[x,y] = pixcenters(R, height, width)
[x,y] = pixcenters(r,sizea)
[x,y] = pixcenters(..., 'makegrid')
```

`[x,y] = pixcenters(R, height, width)` returns the spatial coordinates of a spatially-referenced image or regular gridded data set. `R` is the 3-by-2 affine referencing matrix. `height` and `width` are the image dimensions. If `r` does not include a rotation (i.e.,  $r(1,1) = r(2,2) = 0$ ), then `x` is a 1-by-width vector and `y` is a height-by-1 vector. In this case, the spatial coordinates of the pixel in row `row` and column `col` are given by `x(col)`, `y(row)`. Otherwise, `x` and `y` are each a height-by-width matrix such that `x(col,row)`, `y(col,row)` are the coordinates of the pixel with subscripts `(row,col)`.

`[x,y] = pixcenters(r,sizea)` accepts the size vector `sizea = [height, width, ...]` instead of `height` and `width`.

`[x,y] = pixcenters(info)` accepts a scalar struct array with the fields

'RefMatrix'	3-by-2 referencing matrix
'Height'	Scalar number
'Width'	Scalar number

`[x,y] = pixcenters(..., 'makegrid')` returns `x` and `y` as height-by-width matrices even if `r` is irrotational. This syntax can be helpful when you call `pixcenters` from within a function or script.

## Remarks

For more information on referencing matrices, see the `makerefmat` reference page.

`pixcenters` is useful for working with `surf`, `mesh`, or `surface`, and for coordinate transformations.

## Example

```
[Z,R] = arcgridread('MtWashington-ft.grd');
[x,y] = pixcenters(R, size(Z));
h = surf(x,y,Z); axis equal; colormap(demcmap(Z))
```

```
set(h,'EdgeColor','none')
xlabel('x (easting in meters)')
ylabel('y (northing in meters)')
zlabel('elevation in feet')colormap(terrain)
```

**See Also**

`arcgridread`, `makerefmat`, `mapbbox`, `mapoutline`, `pix2map`,  
`worldfileread`

The help for `mapshow` provides an alternative version of the preceding example.

# plabel

---

## **Purpose**

Toggle and control display of parallel labels

## **Syntax**

```
plabel  
plabel('on')  
plabel('off')  
plabel(meridian)  
plabel(MapAxesPropertyName,PropertyValue,...)
```

plabel toggles the visibility of parallel labeling on the current map axes.

plabel('on') sets the visibility of parallel labels to 'on'.

plabel('off') sets the visibility of parallel labels to 'off'.

plabel('reset') resets the displayed parallel labels using the currently defined parallel label properties.

plabel(meridian) sets the value of the PLabelMeridian property of the map axes to the value meridian. This determines the meridian upon which the labels are placed (see axesm). The options for meridian are a scalar longitude or the strings 'east', 'west', or 'prime'.

plabel(*MapAxesPropertyName*,PropertyValue,...) allows paired map axes property names and property values to be passed in. For a complete description of map axes properties, see the axesm reference page.

Parallel label handles can be returned in h if desired.

## **See Also**

axesm, setm, mlabel

**Purpose**

Project 3-D lines and points on map axes

**Syntax**

```
h = plot3m(lat,lon,z)
h = plot3m(lat,lon,linetype)
h = plot3m(lat,lon,PropertyName,PropertyValue,...)
```

`h = plot3m(lat,lon,z)` displays projected line objects on the current map axes. `lat` and `lon` are the latitude and longitude coordinates, respectively, of the line object to be projected. Note that this ordering is conceptually reversed from the MATLAB line function, because the *vertical* ( $y$ ) coordinate comes first. However, the ordering latitude, then longitude, is standard geographic usage. `lat` and `lon` must be the same size, and in the `AngleUnits` of the map axes. `z` is the altitude data associated with each point in `lat` and `lon`. The object handle for the displayed line can be returned in `h`.

The units of `z` are arbitrary, except when using the globe projection. In the case of globe, `z` should have the same units as the radius of the earth or semimajor axis specified in the 'geoid' (reference ellipsoid) property of the map axes. This implies that for a reference ellipsoid vector of [1 0] (a unit sphere), the units of `z` are earth radii.

`h = plot3m(lat,lon,linetype)` allows the specification of the line style, where *linetype* is any string recognized by the MATLAB line function.

`h = plot3m(lat,lon,PropertyName,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB line function except for `XData`, `YData`, and `ZData`.

**Remarks**

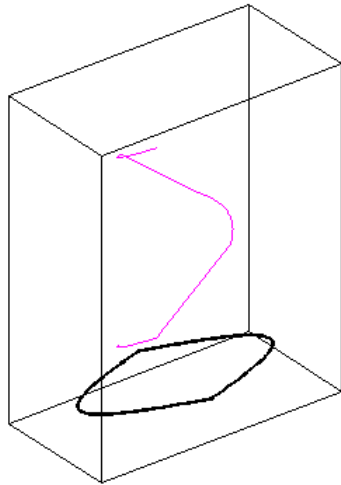
`plot3m` is the mapping equivalent of the MATLAB `plot3` function.

**Example**

```
axesm sinusoid; framem; view(3)
[lats,longs] = interp([45 -45 -45 45 45 -45]',...
                    [-100 -100 100 100 -100 -100]','1);
z = (1:671)'/100;
plot3m(lats,longs,z,'m')
```

# plot3m

---



## See Also

`linem`, `plot3`, `plotm`

**Purpose**

Project 2-D lines and points on map axes

**Syntax**

```
h = plotm(lat,lon)
h = plotm(lat,lon,linetype)
h = plotm(lat,lon,PropertyName,PropertyValue,...)
h = plotm([lat lon],...)
```

`h = plotm(lat,lon)` displays projected line objects on the current map axes. `lat` and `lon` are the latitude and longitude coordinates, respectively, of the line object to be projected. Note that this ordering is conceptually reversed from the MATLAB line function, because the *vertical* ( $y$ ) coordinate comes first. However, the ordering latitude, then longitude, is standard geographic usage. `lat` and `lon` must be the same size, and in the `AngleUnits` of the map axes. The object handle for the displayed line can be returned in `h`.

`h = plotm(lat,lon,linetype)` allows the specification of the line style, where *linetype* is any string recognized by the MATLAB line function.

`h = plotm(lat,lon,PropertyName,PropertyValue,...)` allows the specification of any number of property name/property value pairs for any properties recognized by the MATLAB line function except for `XData`, `YData`, and `ZData`.

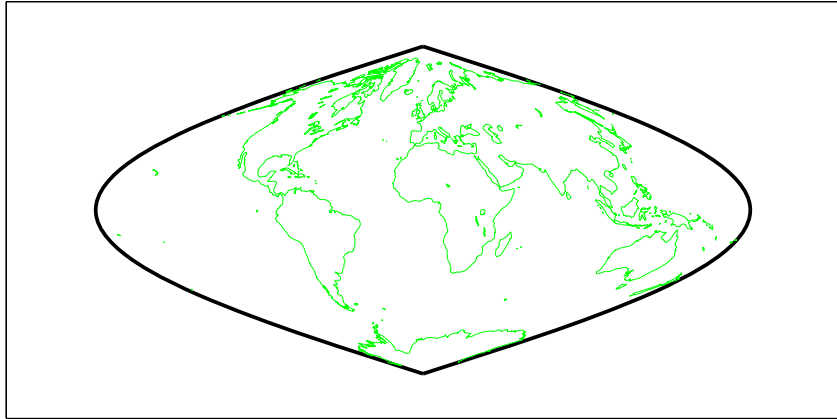
`h = plotm([lat lon],...)` allows the coordinates to be packed into a single two-column matrix.

**Description**

`plotm` is the mapping equivalent of the MATLAB `plot` function.

**Example**

```
load coast
axesm sinusoid; framem
plotm(lat,long,'g')
```



**See Also**

`linem`, `plot`, `plot3m`

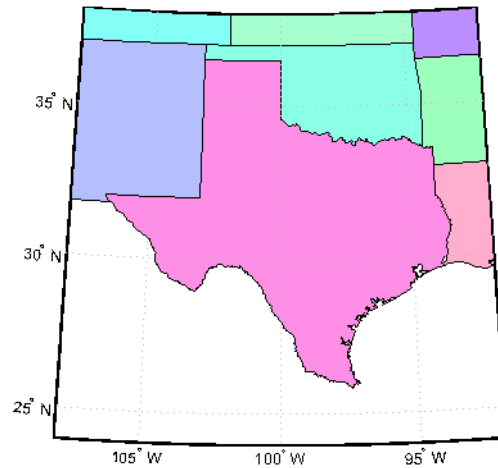


<b>Purpose</b>	Colormaps appropriate to political regions
<b>Syntax</b>	<pre>polcmap polcmap(ncolors) polcmap(ncolors,maxsat) polcmap(ncolors,huelimits,saturationlimits,valuelimits) cmap = polcmap(...)</pre> <p><code>polcmap</code> applies a random muted colormap to the current figure. The size of the colormap is the same as the existing colormap.</p> <p><code>polcmap(ncolors)</code> creates a colormap with the specified number of colors.</p> <p><code>polcmap(ncolors,maxsat)</code> controls the maximum saturation of the colors. Larger maximum saturation values produce brighter, more saturated colors. If omitted, the default is 0.5.</p> <p><code>polcmap(ncolors,huelimits,saturationlimits,valuelimits)</code> controls the colors. Hue, saturation, and value are randomly selected values within the limit vectors. These are two-element vectors of the form [min max]. Valid values range from 0 to 1. As the hue varies from 0 to 1, the resulting color varies from red, through yellow, green, cyan, blue, and magenta, back to red. When the saturation is 0, the colors are unsaturated; they are simply shades of gray. When the saturation is 1, the colors are fully saturated; they contain no white component. As the value varies from 0 to 1, the brightness increases.</p> <p><code>cmap = polcmap(...)</code> returns the colormap without applying it to the figure.</p>
<b>Remarks</b>	You cannot use <code>polcmap</code> to alter the colors of displayed patches drawn by <code>geoshow</code> or <code>mapshow</code> . The patches must have been rendered by <code>displaym</code> . However, you can color patches using <code>polcmap</code> when you call <code>geoshow</code> or <code>mapshow</code> , as shown below.
<b>Example</b>	Draw a map of Texas and surrounding states. Color the patches with a <code>symbolspec</code> constructed using <code>polcmap</code> :

# polcmap

---

```
figure; usamap('texas')
states = shaperead('usastatelo.shp','UseGeoCoords',true);
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))});
geoshow(states, 'DisplayType', 'polygon', ...
    'SymbolSpec', faceColors)
```



Note that the colors you obtain for this example can vary from what you see above because polcmap computes them randomly.

## See Also

demcmap, colormap

**Purpose** Convert polygon contour to counterclockwise vertex ordering

**Syntax** `[x2, y2] = poly2ccw(x1, y1)`  
`[x2, y2] = poly2ccw(x1, y1)` arranges the vertices in the polygonal contour `(x1, y1)` in counterclockwise order, returning the result in `x2` and `y2`. If `x1` and `y1` can contain multiple contours, represented either as NaN-separated vectors or as cell arrays, then each contour is converted to clockwise ordering. `x2` and `y2` have the same format (NaN-separated vectors or cell arrays) as `x1` and `y1`.

**Example** Convert a clockwise-ordered square to counterclockwise ordering.

```
x1 = [0 0 1 1 0];
y1 = [0 1 1 0 0];
ispolycw(x1, y1)

ans =
     1

[x2, y2] = poly2ccw(x1, y1);
ispolycw(x2, y2)

ans =
     0
```

**See also** `ispolycw`, `poly2cw`, `polybool`

**Purpose** Convert polygon contour to clockwise vertex ordering

**Syntax** `[x2, y2] = poly2cw(x1, y1)`  
`[x2, y2] = poly2cw(x1, y1)` arranges the vertices in the polygonal contour `(x1, y1)` in clockwise order, returning the result in `x2` and `y2`. If `x1` and `y1` can contain multiple contours, represented either as NaN-separated vectors or as cell arrays, then each contour is converted to clockwise ordering. `x2` and `y2` have the same format (NaN-separated vectors or cell arrays) as `x1` and `y1`.

**Example** Convert a counterclockwise-ordered square to clockwise ordering.

```
x1 = [0 1 1 0 0];  
y1 = [0 0 1 1 0];  
ispolycw(x1, y1)  
  
ans =  
    0  
  
[x2, y2] = poly2cw(x1, y1);  
ispolycw(x2, y2)  
  
ans =  
    1
```

**See also** `ispolycw`, `poly2ccw`, `polybool`

**Purpose**

Convert polygonal region to patch faces and vertices

**Syntax**

```
[F, V] = poly2fv(x, y)
```

`[F, V] = poly2fv(x, y)` converts the polygonal region represented by the contours `(x, y)` into a faces matrix, `F`, and a vertices matrix, `V`, that can be used with the `patch` function to display the region. The contour vertices can be represented either in NaN-separated vector format or cell array format.

Individual contours in `x` and `y` are assumed to be external contours if their vertices are arranged in clockwise order; otherwise they are assumed to be internal contours. Use `poly2cw` or `poly2ccw`, if necessary, to achieve the desired vertex ordering.

**Example**

Display a rectangular region with two holes using a single patch object.

```
% External contour, rectangle, clockwise ordered.
x1 = [0 0 6 6 0];
y1 = [0 3 3 0 0];

% First hole contour, square, counterclockwise ordered.
x2 = [1 2 2 1 1];
y2 = [1 1 2 2 1];

% Second hole contour, triangle, counterclockwise ordered.
x3 = [4 5 4 4];
y3 = [1 1 2 1];

% Compute face and vertex matrices.
[f, v] = poly2fv({x1, x2, x3}, {y1, y2, y3});

% Display the patch.
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
      'EdgeColor', 'none');
axis off, axis equal
```

## poly2fv

---

See the documentation for `polybool` for additional examples illustrating `poly2fv`.

**See also** `ispolycw`, `patch`, `poly2cw`, `poly2ccw`, `polybool`

**Purpose**

Set operations on polygonal regions

**Syntax**

```
[x,y] = polybool(flag,x1,y1,x2,y2)
```

`[x,y] = polybool(flag,x1,y1,x2,y2)` performs the polygon set operation identified by `flag`. A valid flag string is any one of the following alternatives:

- Region intersection: 'intersection', 'and', '&'
- Region union: 'union', 'or', '|', '+', 'plus'
- Region subtraction: 'subtraction', 'minus', '-'
- Region exclusive or: 'exclusiveor', 'xor'

The polygon inputs are NaN-delimited vectors, or cell arrays containing individual polygonal contours. The result is output using the same format as the input.

`polybool` assumes that individual contours whose vertices are clockwise ordered are external contours, and that contours whose vertices are counterclockwise ordered are internal contours. You can use `poly2cw` to convert a polygonal contour to clockwise ordering.

**Limitations**

Polygons processed via `polybool` are assumed to be in a Cartesian coordinate system. Therefore, geographic data that encompasses a pole cannot be used directly. Use `flatearthpoly` to convert polygons that contain a pole to Cartesian coordinates.

**Examples****Example 1**

Set operations on two overlapping circular regions:

```
theta = linspace(0, 2*pi, 100);
x1 = cos(theta) - 0.5;
y1 = -sin(theta);    % -sin(theta) to make a clockwise contour
x2 = x1 + 1;
y2 = y1;
[xa, ya] = polybool('union', x1, y1, x2, y2);
```

```
[xb, yb] = polybool('intersection', x1, y1, x2, y2);
[xc, yc] = polybool('xor', x1, y1, x2, y2);
[xd, yd] = polybool('subtraction', x1, y1, x2, y2);

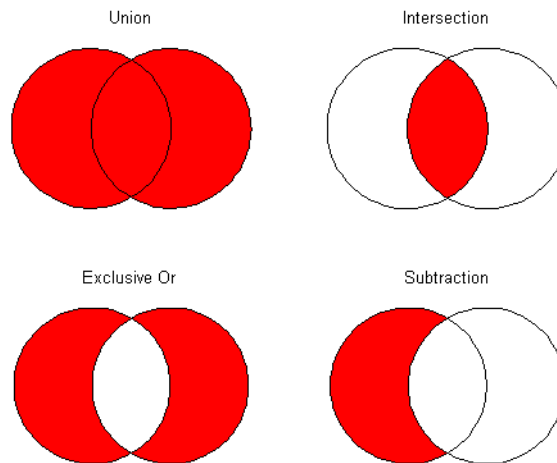
subplot(2, 2, 1)
patch(xa, ya, 1, 'FaceColor', 'r')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Union')

subplot(2, 2, 2)
patch(xb, yb, 1, 'FaceColor', 'r')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Intersection')

subplot(2, 2, 3)
% The output of the exclusive-or operation consists of disjoint
% regions. It can be plotted as a single patch object using the
% face-vertex form. Use poly2fv to convert a polygonal region
% to face-vertex form.
[f, v] = poly2fv(xc, yc);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
'EdgeColor', 'none')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Exclusive Or')

subplot(2, 2, 4)
patch(xd, yd, 1, 'FaceColor', 'r')
axis equal, axis off, hold on
plot(x1, y1, x2, y2, 'Color', 'k')
title('Subtraction')
```





## Example 2

Set operations on regions with holes

```
Ax = {[1 1 6 6 1], [2 5 5 2 2], [2 5 5 2 2]};
Ay = {[1 6 6 1 1], [2 2 3 3 2], [4 4 5 5 4]};
subplot(2, 3, 1)
[f, v] = poly2fv(Ax, Ay);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
      'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Ax), plot(Ax{k}, Ay{k}, 'Color', 'k'), end
title('A')
```

```
Bx = {[0 0 7 7 0], [1 3 3 1 1], [4 6 6 4 4]};
By = {[0 7 7 0 0], [1 1 6 6 1], [1 1 6 6 1]};
subplot(2, 3, 4);
[f, v] = poly2fv(Bx, By);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
      'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Bx), plot(Bx{k}, By{k}, 'Color', 'k'), end
```

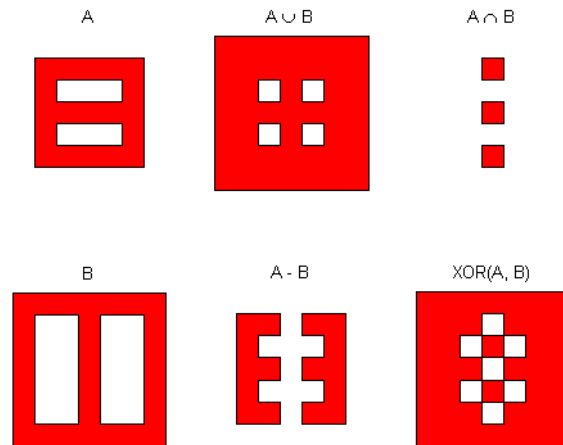
```
title('B')

subplot(2, 3, 2)
[Cx, Cy] = polybool('union', Ax, Ay, Bx, By);
[f, v] = poly2fv(Cx, Cy);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
      'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Cx), plot(Cx{k}, Cy{k}, 'Color', 'k'), end
title('A \cup B')

subplot(2, 3, 3)
[Dx, Dy] = polybool('intersection', Ax, Ay, Bx, By);
[f, v] = poly2fv(Dx, Dy);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
      'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Dx), plot(Dx{k}, Dy{k}, 'Color', 'k'), end
title('A \cap B')

subplot(2, 3, 5)
[Ex, Ey] = polybool('subtraction', Ax, Ay, Bx, By);
[f, v] = poly2fv(Ex, Ey);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
      'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Ex), plot(Ex{k}, Ey{k}, 'Color', 'k'), end
title('A - B')

subplot(2, 3, 6)
[Fx, Fy] = polybool('xor', Ax, Ay, Bx, By);
[f, v] = poly2fv(Fx, Fy);
patch('Faces', f, 'Vertices', v, 'FaceColor', 'r', ...
      'EdgeColor', 'none')
axis equal, axis off, axis([0 7 0 7]), hold on
for k = 1:numel(Fx), plot(Fx{k}, Fy{k}, 'Color', 'k'), end
title('XOR(A, B)')
```

**See Also**

[bufferm](#), [flatearthpoly](#), [ispolycw](#), [poly2cw](#), [poly2ccw](#), [poly2fv](#), [polyjoin](#), [polysplit](#)

# polycut

---

**Purpose** Polygon branch cuts for holes

**Syntax** `[lat2,long2] = polycut(lat,long)`  
`[lat2,long2] = polycut(lat,long)` connects the contour and holes of polygons using optimal branch cuts. Polygons are input as NaN-delimited vectors, or as cell arrays containing individual polygons in each element with the outer face separated from the subsequent inner faces by NaNs. Multiple polygons outputs are separated by NaNs.

**See Also** `polybool`, `polysplit`, `polyjoin`

**Purpose**

Convert polygon segments from cell array to vector format

**Syntax**

```
[lat,lon] = polyjoin(latcells,loncells)
```

[lat,lon] = polyjoin(latcells,loncells) converts polygons from cell array format to column vector format. In cell array format, each element of the cell array is a vector that defines a separate polygon.

**Remarks**

A polygon may consist of an outer contour followed by holes separated with NaNs. In vector format, each vector may contain multiple faces separated by NaNs. There is no structural distinction between outer contours and holes in vector format.

**Example**

```
latcells = {[1 2 3]'; 4; [5 6 7 8 NaN 9]'};
loncells = {[9 8 7]'; 6; [5 4 3 2 NaN 1]'};
[lat,lon] = polyjoin(latcells,loncells);
[lat lon]
```

```
ans =
     1     9
     2     8
     3     7
    NaN    NaN
     4     6
    NaN    NaN
     5     5
     6     4
     7     3
     8     2
    NaN    NaN
     9     1
```

**See Also**

polybool, polycut, polysplit

# polymerge

---

## Purpose

Merge line segments with matching endpoints

## Syntax

```
[lat2,lonc2 = polymerge(lat,lon)
[lat2,lonc2 = polymerge(lat,lon,tol)
[lat2,lonc2 = polymerge(lat,lon,tol,outputformat)
```

[lat2,lonc2 = polymerge(lat,lon) combines vector line segments with identical endpoints. polymerge compares the endpoints of all line segments and combines those that match. The line can be input as vectors of latitude and longitude with NaNs delimiting segments. The line can also be input as cell arrays, with each element of a cell array containing a line segment. The resulting line is in the same format as the input.

[lat2,lonc2 = polymerge(lat,lon,tol) combines line segments whose endpoints are separated by less than the circular tolerance. If omitted, tol = 0 is assumed. The tolerance is in the same units as the polygon input.

[lat2,lonc2 = polymerge(lat,lon,tol,outputformat) controls the format of the resulting polygons. If *outputformat* is 'vector', the result is returned as vectors with NaNs separating the segments. If *outputformat* is 'cell', the result is returned as cell arrays containing segments in each element. If omitted, 'vector' is assumed.

## Example

```
lat = [1 2 3 NaN 6 7 8 9 NaN 6 5 4 3 NaN 12 13 14 NaN 9 10 11 12]';
lon = lat;
[lat2,lon2] = polymerge(lat,lon);
[lat2 lon2]
```

```
ans =
    14    14
    13    13
    12    12
    12    12
    11    11
    10    10
     9     9
```

9	9
8	8
7	7
6	6
6	6
5	5
4	4
3	3
3	3
2	2
1	1

**See Also**      polybool, polyjoin, polysplit

# polysplit

---

**Purpose** Extract segments of NaN-delimited polygon vectors to cell arrays

**Syntax** `[latcells,loncells] = polysplit(lat,lon)`  
`[latcells,loncells] = polysplit(lat,lon)` returns the NaN-delimited segments of the vectors `lat` and `lon` as N-by-1 cell arrays with one polygon segment per cell. `lat` and `lon` must be the same size and have identically-placed NaNs. The polygon segments are column vectors if `lat` and `lon` are column vectors, and row vectors otherwise.

**Example**

```
lat = [1 2 3 NaN 4 NaN 5 6 7 8 9]';  
lon = [9 8 7 NaN 6 NaN 5 4 3 2 1]';  
[latcells,loncells] = polysplit(lat,lon);  
[latcells loncells]  
  
ans =  
    [3x1 double]    [3x1 double]  
    [         4]    [         6]  
    [5x1 double]    [5x1 double]
```

**See Also** `isshapemultipart`, `polybool`, `polycut`, `polyjoin`



**Purpose**

Compute line or polygon intersection points

**Syntax**

```
[xi,yi] = polyxpoly(x1,y1,x2,y2)
```

```
[xi,yi] = polyxpoly(...,'unique')
```

```
[xi,yi,ii] = polyxpoly(...)
```

`[xi,yi] = polyxpoly(x1,y1,x2,y2)` returns the intersection points of two sets of lines and/or polygons.

`[xi,yi] = polyxpoly(...,'unique')` returns only unique intersections.

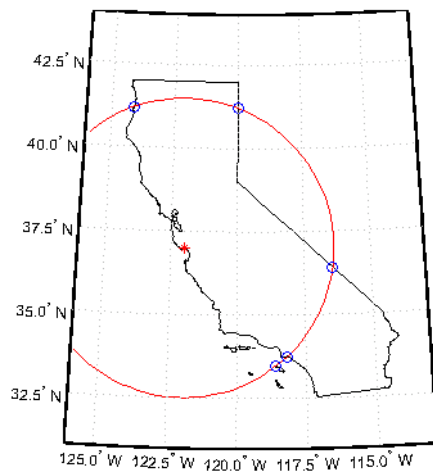
`[xi,yi,ii] = polyxpoly(...)` also returns a two-column index of line segment numbers corresponding to the intersection points.

**Example**

```
california = shaperead('usastatehi',...
    'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'California'),'Name'});
usamap('california')
geoshow(california, 'FaceColor', 'none')

lat0 = 37; lon0 = -122; rad = 500;
[latc, lonc] = scircle1(lat0, lon0, km2deg(rad));
plotm(lat0, lon0, 'r*')
plotm(latc, lonc, 'r')

[lat, lon] = reducem(california.Lat', california.Lon');
[loni, lati] = polyxpoly(lon, lat, lonc, latc);
plotm(lati, loni, 'bo')
```

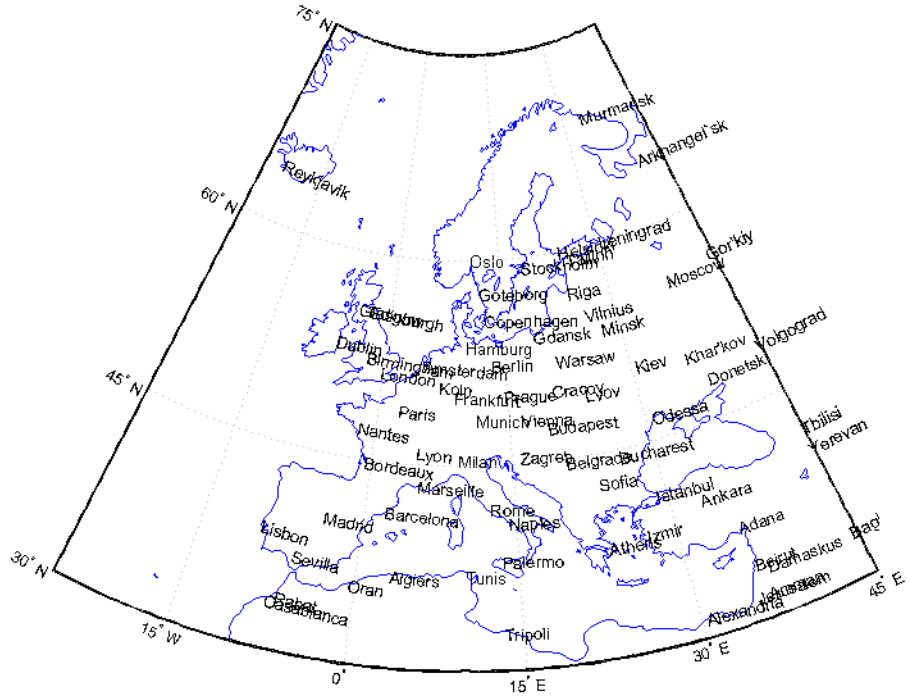


## See Also

crossfix, gcxgc, gcxsc, navfix, rhxrh, scxsc

<b>Purpose</b>	View map at printed size
<b>Syntax</b>	
<b>Background</b>	The appearance of a map onscreen can differ from the final printed output. This results from the difference in the size and shape of the figure window and the area the figure occupies on the printed page. A map that appears readable on screen might be cluttered when the printed output is smaller. Likewise, the relative position of multiple axes can appear different when printed. This function resizes the figure to the printed size.
<b>Remarks</b>	previewmap changes the size of the current figure to match the printed output. If the resulting figure size exceeds the screen size, the figure is enlarged as much as possible.
<b>Examples</b>	Is the text small enough to avoid overlapping in a map of Europe? <pre>figure worldmap europe land=shaperead('landareas.shp','UseGeoCoords',true); geoshow([land.Lat],[land.Lon]) m=gcm; latlim = m.maplatlimit; lonlim = m.maplonlimit; BoundingBox = [lonlim(1) latlim(1);lonlim(2) latlim(2)]; cities=shaperead('worldcities.shp', ...     'BoundingBox',BoundingBox,'UseGeoCoords',true); for index=1:numel(cities)     h=textm(cities(index).Lat, cities(index).Lon, ...         cities(index).Name);     trimcart(h)     rotatetext(h) end orient landscape tightmap</pre>

axis off  
previewmap



## Limitations

The figure cannot be made larger than the screen.

## See Also

pagesetupd1g, paperscale, axescale

**Purpose**

Project displayed map graphics object

**Syntax**

```
project(h)
project(h, 'xy')
project(h, 'yx')
```

`project(h)` takes unprojected objects with handles `h` that are displayed on map axes and projects them. For example, `project` takes a line created on a map axes with the `plot` function and projects it as though it had been created with the `plotm` function. This can be useful if a standard MATLAB function was accidentally executed. The map structure of the existing map axes determines the specifics of the projection. If `h` is the handle of the map axes, then all the children of `h` are projected. Do not attempt this if any children of `h` have already been projected!

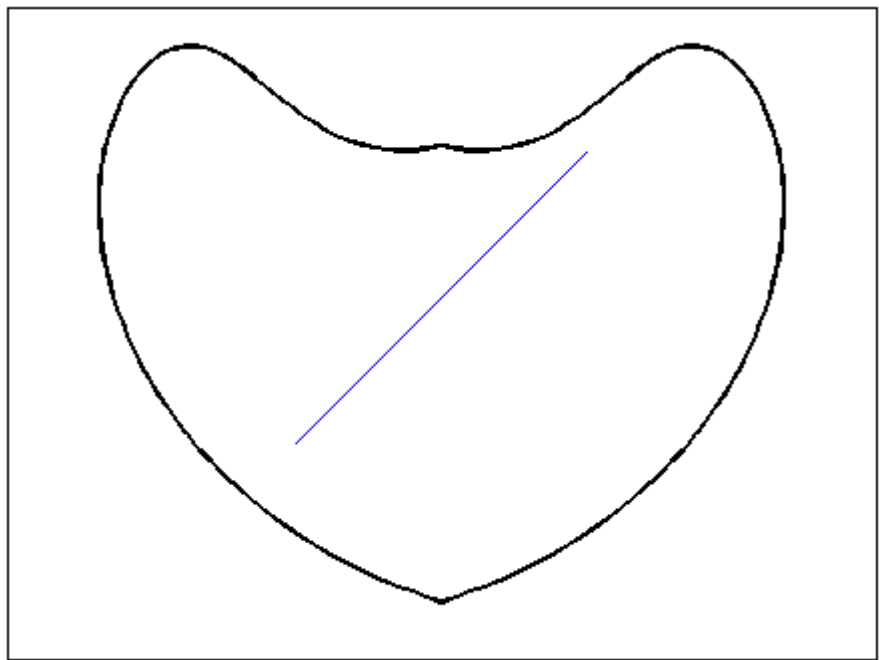
`project(h, 'xy')` specifies that the `XData` of the unprojected objects corresponds to longitudes and the `YData` to latitudes. This is the default assumption.

`project(h, 'yx')` specifies that the `XData` of the unprojected objects corresponds to latitudes and the `YData` to longitudes.

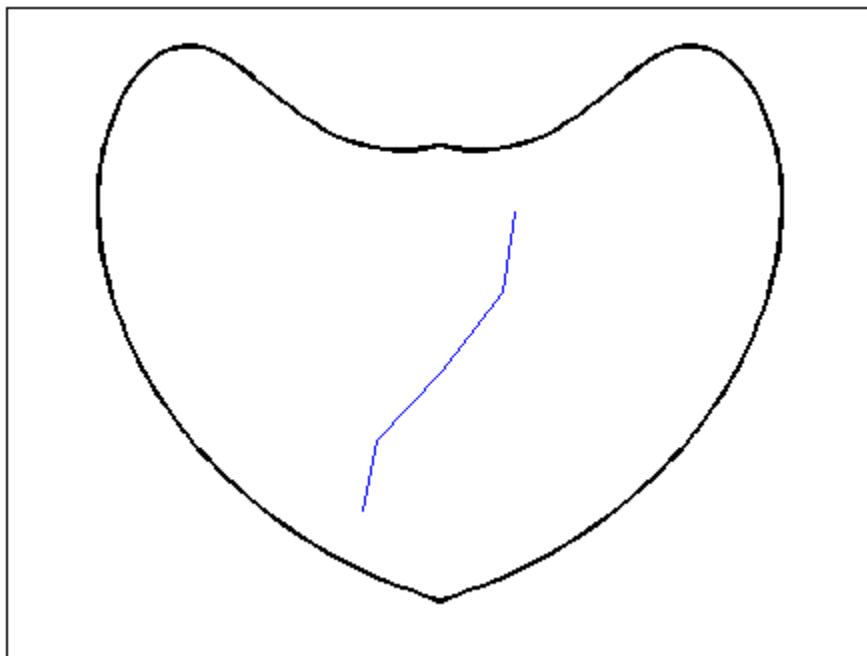
**Example**

Create an axes, plot a line, then project it:

```
axesm('bonne','AngleUnits','radians');framem;
h = plot([-1 -.5 0 .5 1],[-1 -.5 0 .5 1]);
```



project(h)



The line is straight in  $x$ - $y$  space, but when converted to a projected map object, it bends with the projection.

**See Also**

`linem`, `patchm`, `surfacem`, `textm`

**Purpose** Forward map projection using PROJ.4 map projection library

**Syntax** `[x, y] = projfwd(proj, lat, lon)`  
`[x, y] = projfwd(proj, lat, lon)` returns the x and y map coordinates from the forward projection transformation. `proj` is a structure defining the map projection. `proj` can be an `mstruct` or a `GeoTIFF` info structure. `lat` and `lon` are arrays of the latitude and longitude coordinates.

For a complete list of `GeoTIFF` info and map projection structures that you can use with `projfwd`, see the reference page for `projlist`.

**Example** **Overlay the boundary of Massachusetts on an orthophoto of Boston**

Read vector data for state boundary of Massachusetts (in latitude and longitude):

```
S = shaperead('usastatehi', 'UseGeoCoords', true, ...  
            'Selector',{@(name) strcmpi(name,'Massachusetts'), 'Name'});
```

Obtain the projection structure for the orthophoto and project the state boundary vectors to it (Massachusetts State Plane coordinate system, U.S. Survey Feet):

```
proj = geotiffinfo('boston.tif');  
lat = [S.Lat];  
lon = [S.Lon];  
[x, y] = projfwd(proj, lat, lon);
```

Read and display the 'boston.tif' orthophoto image:

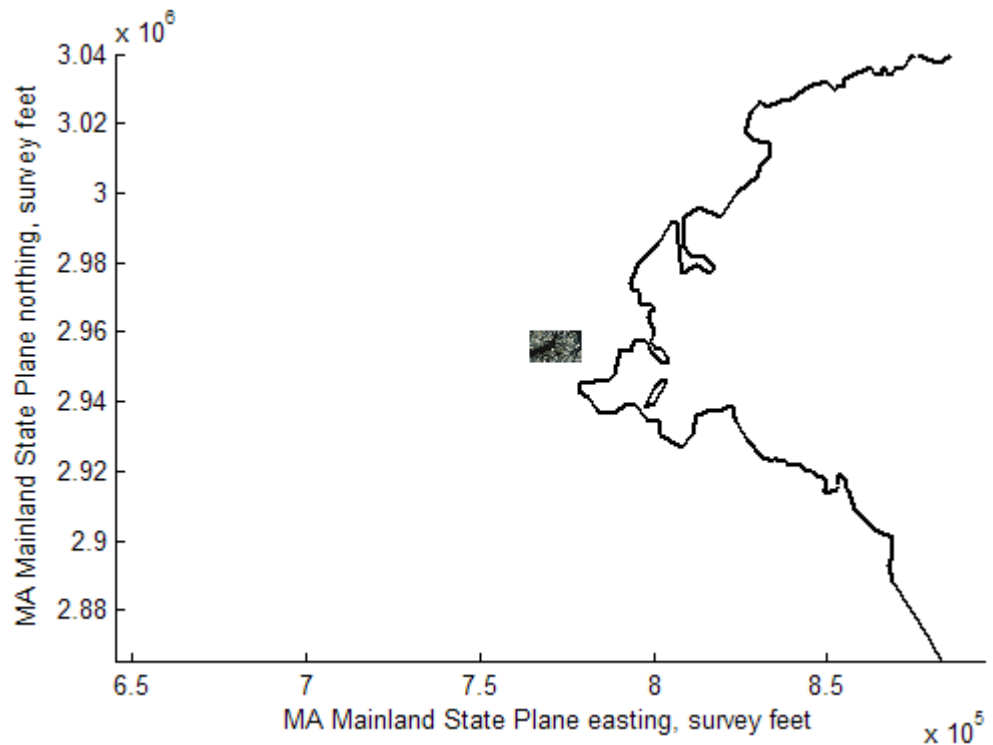
```
[RGB, R, bbox] = geotiffread('boston.tif');  
figure  
mapshow(RGB, R)  
xlabel('MA Mainland State Plane easting, survey feet')  
ylabel('MA Mainland State Plane northing, survey feet')
```



Overlay the state boundary and set map limits to show a little more detail:

```
hold on
mapshow(gca, x, y, 'Color', 'black', 'LineWidth', 2.0)
set(gca, 'XLim', [ 645000,  895000], ...
        'YLim', [2865000, 3040000]);
```

boston.tif image copyright © GeoEye, all rights reserved.



## See Also

geotiffinfo, mfwdtran, minvtran, projinv, projlist

## Purpose

Inverse map projection using PROJ.4 map projection library

## Syntax

```
[lat, lon] = projinv(proj, x, y)
```

[lat, lon] = projinv(proj, x, y) returns the latitude and longitude values from the inverse projection transformation. proj is a structure defining the map projection. proj can be a map projection mstruct or a GeoTIFF info structure. x and y are *x-y* map coordinate arrays. For a complete list of GeoTIFF info and map projection structures that you can use with projinv, see the reference page for projlist.

## Example

### Display Boston Orthophoto on a Mercator projection

- 1 Import the Boston roads from the shapefile and obtain the projection structure from the 'boston.tif' orthophoto:

```
roads = shaperead('boston_roads.shp');  
proj = geotiffinfo('boston.tif');
```

- 2 Convert the road coordinates to the projection's length unit. As shown by the UOMLength field of the projection structure, the units of length in the projected coordinate system is US Survey Feet. Coordinates in the roads shapefile are in meters:

```
proj.UOMLength
```

```
ans =  
US survey foot
```

```
x = [roads.X] * unitsratio('survey feet','meter');  
y = [roads.Y] * unitsratio('survey feet','meter');
```

```
% Now convert the scaled coordinates of the roads to latitude and lon  
[roadsLat, roadsLon] = projinv(proj, x, y);
```

- 3 Read the boston\_ovr.jpg image and worldfile:

```
RGB = imread('boston_ovr.jpg');
```

```
R = worldfileread(getworldfilename('boston_ovr.jpg'));
```

- 4 Read state boundary vectors for Massachusetts from the `usastatehi` shapefile using a selector to eliminate other states:

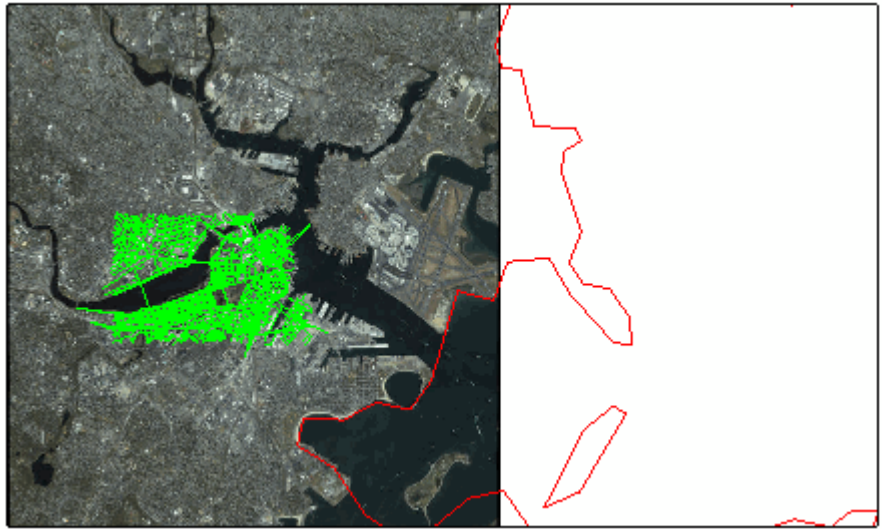
```
S = shaperead('usastatehi', 'UseGeoCoords', true, ...  
    'Selector',{@(name) strcmpi(name,'Massachusetts')}, 'Name');
```

- 5 Open a figure with a Mercator projection and display the state boundary, image, and roads:

```
figure  
axesm('mercator')  
  
geoshow(S.Lat, S.Lon, 'Color','red')  
geoshow(RGB, R)  
geoshow(roadsLat, roadsLon, 'Color', 'green')
```

- 6 Set the map boundary to the image's northern, western, and southern limits, and the eastern limit of the state boundary within the image latitude bounding box:

```
[lon, lat] = mapoutline(R, size(RGB(:,:,1)));  
ltvals = find((S.Lat>=min(lat(:))) & (S.Lat<=max(lat(:))));  
setm(gca,'maplonlimit',[min(lon(:)) max(S.Lon(ltvals))], ...  
    'maplatlimit',[min(lat(:)) max(lat(:))])  
tightmap
```



boston\_ovr.jpg image copyright © GeoEye, all rights reserved.

## See Also

[geotiffinfo](#), [mfwdtran](#), [minvtran](#), [projfwd](#), [projlist](#)

**Purpose**

Map projections supported by projfwd and projinv

**Syntax**

```
projlist(listmode)  
S = projlist(listmode)
```

`projlist(listmode)` displays a table of projection names, IDs, and availability. *listmode* is a string with value 'mapprojection', 'geotiff', 'geotiff2mstruct', or 'all'. The default value is 'mapprojection'.

`S = projlist(listmode)` returns a structure array containing projection names, IDs, and availability. The output of `projlist` for each *listmode* is described below:

- **mapprojection** — Lists the map projection IDs that are available for use with `projfwd` and `projinv`. The output structure contains the fields
  - **Name** — Projection name
  - **MapProjection** — Projection ID string
- **geotiff** — Lists the GeoTIFF projection IDs that are available for use with `projfwd` and `projinv`. The output structure contains the fields
  - **GeoTIFF** — GeoTIFF projection ID string.
  - **Available**— Logical array with values 1 or 0
- **geotiff2mstruct** — Lists the GeoTIFF projection IDs that are available for use with `geotiff2mstruct`. The output structure contains the fields
  - **GeoTIFF** — GeoTIFF projection ID string
  - **MapProjection** — Projection ID string
- **all** — Lists the map and GeoTIFF projection IDs that are available for use with `projfwd` and `projinv`. The output structure contains the fields
  - **GeoTIFF** — GeoTIFF projection ID string

# projlist

---

- MapProjection — Projection ID string
- info — Logical array with values 1 or 0
- mstruct — Logical array with values 1 or 0

## Remarks

projfwd and projinv can be used to process certain forward or inverse map projections. These functions are implemented in C using the PROJ.4 library. projlist provides a convenient list of the projections that can be used with projfwd or projinv. Because projfwd and projinv accept either a map projection structure (mstruct) or a GeoTIFF info structure, projlist provides separate lists for each case. It can also list the projections for which a GeoTIFF info structure can be converted to an mstruct.

## Examples

```
s=projlist

s =
1x19 struct array with fields:
    Name
    MapProjection

s=projlist('geotiff2mstruct')

s =
1x19 struct array with fields:
    GeoTIFF
    MapProjection
```

## See Also

geotiff2mstruct, projfwd, projinv, maplist, maps

**Purpose**

Origin vector to place north pole at specified point

**Syntax**

```
origin = putpole(pole)
origin = putpole(pole,units)
```

`origin = putpole(pole)` returns an origin vector required to transform a coordinate system in such a way as to put the true North Pole at a point specified by the three- (or two-) element vector `pole`. This vector is of the form `[latitude longitude meridian]`, specifying the coordinates in the original system at which the true North Pole is to be placed in the transformed system. The meridian is the longitude upon which the new system is to be centered, which is the new pole longitude if omitted. The output is a three-element vector of the form `[latitude longitude orientation]`, where the latitude and longitude are the coordinates in the untransformed system of the new origin, and the orientation is the azimuth of the true North Pole in the transformed system.

`origin = putpole(pole,units)` allows the specification of the angular units of the origin vector, where *units* is any valid angle units string. The default is 'degrees'.

**Remarks**

When developing transverse or oblique projections, you need transformed coordinate systems. One way to define these systems is to establish the point in the original (untransformed) system that will become the new (transformed) origin.

**Examples**

Pull the North Pole down the 0° meridian by 30° to 60°N. What is the resulting origin vector?

```
origin = putpole([60 0])

origin =
    30.0000         0         0
```

This makes sense: when the pole slid down 30°, the point that was 30° north of the origin slid down to become the origin. Following is a less obvious transformation:

# putpole

---

```
origin = putpole([60 80 0]) % constrain to original central
                             % meridian

origin =
    4.9809         0    29.6217

origin = putpole([60 80 40]) % constrain to arbitrary meridian

origin =
    4.9809    40.0000    29.6217
```

## See Also

neworig, org2pol



**Purpose**

Project 3-D quiver plot on map axes

**Syntax**

```
h = quiver3m(lat,lon,alt,u,v,w)
h = quiver3m(lat,lon,alt,u,v,w,linespec)
h = quiver3m(lat,lon,alt,u,v,w,linespec,'filled')
h = quiver3m(lat,lon,alt,u,v,w,scale)
h = quiver3m(lat,lon,alt,u,v,w,linespec,scale)
h = quiver3m(lat,lon,alt,u,v,w,linespec,scale,'filled')
```

`h = quiver3m(lat,lon,alt,u,v,w)` displays *velocity* vectors with components  $(u,v,w)$  at the geographic points  $(lat,lon)$  and altitude `alt` on a displayed map axes. The inputs `u`, `v`, and `w` determine the direction of the vectors in latitude, longitude, and altitude, respectively. The function automatically determines the length of these vectors to make them as long as possible without overlap. The object handles of the displayed vectors can be returned in `h`.

`h = quiver3m(lat,lon,alt,u,v,w,linespec)` allows the control of the line specification of the displayed vectors with a *linespec* string recognized by the MATLAB line function. If symbols are indicated in *linespec*, they are plotted at the start points of the vectors, i.e., the input points  $(lat,lon,alt)$ .

`h = quiver3m(lat,lon,alt,u,v,w,linespec,'filled')` results in the filling in of any symbols specified by *linespec*.

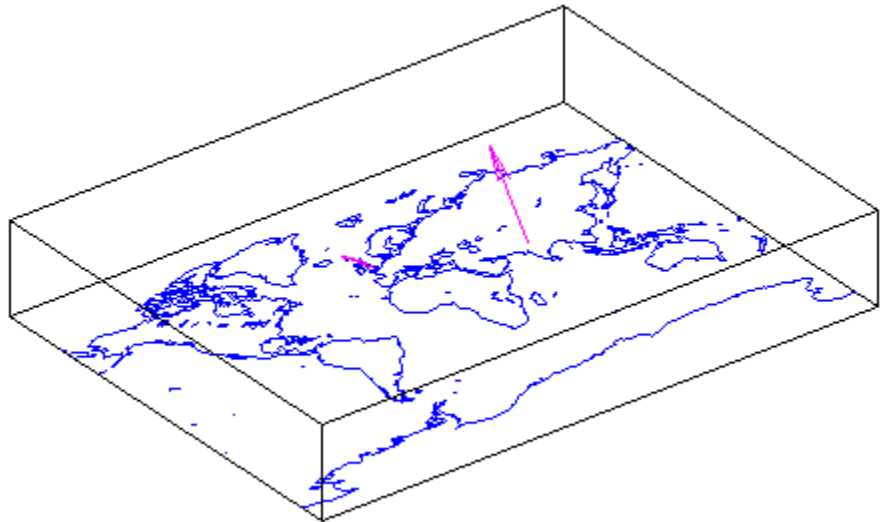
`h = quiver3m(lat,lon,alt,u,v,w,scale)`, `h = quiver3m(lat,lon,alt,u,v,w,linespec,scale)` and `h = quiver3m(lat,lon,alt,u,v,w,linespec,scale,'filled')` alter the automatically calculated vector lengths by multiplying them by the scalar value `scale`. For example, if `scale` is 2, the displayed vectors are twice as long as they would be if `scale` were 1 (the default). When `scale` is set to 0, the automatic scaling is suppressed and the length of the vectors is determined by the inputs. In this case, the vectors are plotted from  $(lat,lon,alt)$  to  $(lat+u,lon+v,alt+w)$ .

**Examples**

Plot 3-D quiver vectors from London (51.5°N,0°) and New Delhi (29°N,77.5°E), both at an altitude of 0. Suppress the automatic scaling. Terminate both vectors at an altitude of 1; the London vector should

terminate 100° southward and 70° eastward, while the New Delhi vector should terminate 50° northward and 10° eastward.

```
load coast
axesm miller; view(3)
plotm(lat,long)
lat0 = [51.5,29]; lon0 = [0 77.5]; alt = [0 0];
u = [-40 50]; v = [-70 10]; w = [1 1];
quiver3m(lat0,lon0,alt,u,v,w,'m')
tightmap
```



## See Also

[quiverm](#), [quiver3](#)

**Purpose**

Project 2-D quiver plot on map axes

**Syntax**

```
h = quiverm(lat,lon,u,v)
h = quiverm(lat,lon,u,v,linespec)
h = quiverm(lat,lon,u,v,linespec,'filled')
h = quiverm(lat,lon,u,v,scale)
h = quiverm(lat,lon,u,v,...linespec,scale,'filled')
```

`h = quiverm(lat,lon,u,v)` displays *velocity* vectors with components  $(u,v)$  at the geographic points  $(lat,lon)$  on displayed map axes. All four inputs should be in the AngleUnits of the map axes. The inputs  $u$  and  $v$  determine the direction of the vectors in latitude and longitude, respectively. The function automatically determines the length of these vectors to make them as long as possible without overlap. The object handles of the displayed vectors can be returned in  $h$ .

`h = quiverm(lat,lon,u,v,linespec)` allows the control of the line specification of the displayed vectors with a *linespec* string recognized by the MATLAB line function. If symbols are indicated in *linespec*, they are plotted at the start points of the vectors, i.e., the input points  $(lat,lon)$ .

`h = quiverm(lat,lon,u,v,linespec,'filled')` results in the filling in of any symbols specified by *linespec*.

`h = quiverm(lat,lon,u,v,scale)` and `h = quiverm(lat,lon,u,v,...linespec,scale,'filled')` alter the automatically calculated vector lengths by multiplying them by the scalar value *scale*. For example, if *scale* is 2, the displayed vectors are twice as long as they would be if *scale* were 1 (the default). When *scale* is set to 0, the automatic scaling is suppressed, and the length of the vectors is determined by the inputs. In this case, the vectors are plotted from  $(lat,lon)$  to  $(lat+u,lon+v)$ .

**Example**

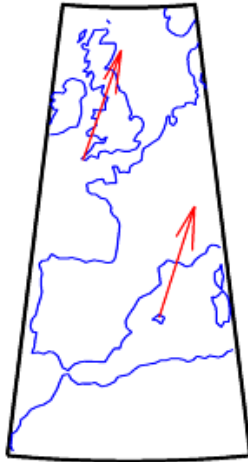
Plot quiver vectors from Land's End (50°N,5.4°W) and Majorca (39.7°N,2.9°E) in a direction corresponding to +5° latitude and +3° longitude. Use automatic scaling.

```
load coast
```

# quiverm

---

```
axesm('eqaconic','MapLatLimit',[30 60],'MapLonLimit',[-10 10])  
framem; plotm(lat,long)  
lat0 = [50 39.7]; lon0 = [-5.4 2.9];  
u = [5 5]; v = [3 3];  
quiverm(lat0,lon0,u,v,'r')
```



## See Also

`quiver3m`, `quiver`

<b>Purpose</b>	Convert angle units from radians to degrees
<b>Syntax</b>	<pre>anglout = rad2deg(anglin)</pre> <p>anglout = rad2deg(anglin) converts angles input in radians to the equivalent measure in degrees.</p>
<b>Remarks</b>	This is both an angle conversion function and a distance conversion function, because arc length can be a measure of distance in either radians or degrees (provided the radius is known).
<b>Example</b>	There are $180^\circ$ in $\pi$ radians: <pre>    anglout = rad2deg(pi)     anglout =         180</pre>
<b>See Also</b>	degrees2dms dms2degrees, deg2rad, nm2km, sm2deg

# rad2dms, rad2dm

---

**Purpose** Convert angles from radians to deg:min or deg:min:sec encoding

---

**Note** The `rad2dm` and `rad2dms` functions are obsolete and error when used. They will be completely removed from the next version of Mapping Toolbox. Instead, call `rad2deg` or multiply input arguments by  $180/\pi$ , and then call `degrees2dm` or `degrees2dms`.

---

**Syntax**

```
anglout = rad2dms(anglin)
angleout = rad2dm(anglin)
```

`anglout = rad2dms(anglin)` converts angles input in radians to the equivalent measure in degrees-minutes-seconds (*dms*) format.

`angleout = rad2dm(anglin)` converts angles input in radians to the equivalent measure in degrees-minutes (*dm*) format. This is the `dms` format, properly rounded to just degrees and minutes.

**Example**

```
rad2dms(1)

ans =
    5717.45

rad2dm(1)

ans =
    5718.00
```

**See Also** `angledim`, `deg2rad`, `dms2degrees`

## Purpose

Convert distance from radians to kilometers, nautical miles, or statute miles

## Syntax

```
km = rad2km(rad)
nm = rad2nm(rad)
sm = rad2sm(rad)
km = rad2km(rad, radius)
nm = rad2nm(rad, radius)
sm = rad2sm(rad, radius)
km = rad2km(rad, sphere)
nm = rad2nm(rad, sphere)
sm = rad2sm(rad, sphere)
```

`km = rad2km(rad)` converts distances from radians to kilometers as measured along a great circle on a sphere with a radius of 6371 km, the mean radius of the Earth.

`nm = rad2nm(rad)` and `sm = rad2sm(rad)` work identically, except that the output units are nautical miles and statute miles, respectively.

`km = rad2km(rad, radius)` converts distances from radians to kilometers as measured along a great circle on a sphere having the specified radius. `radius` must be in units of kilometers.

For `nm = rad2nm(rad, radius)` and `sm = rad2sm(rad, radius)`, make sure your input radius is in the appropriate units.

`km = rad2km(rad, sphere)` converts distances from degrees to kilometers, as measured along a great circle on a sphere approximating an object in the Solar System. `sphere` may be one of the following strings: 'sun', 'moon', 'mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', or 'pluto', and is case-insensitive.

`nm = rad2nm(rad, sphere)` and `sm = rad2sm(rad, sphere)` work identically, except that the output units are nautical miles and statute miles, respectively.

# rad2km, rad2nm, rad2sm

---

## Examples

How long is a trip around the equator in statute miles?

```
sm = rad2sm(2*pi)
```

```
sm =  
2.4874e+04
```

How about on Jupiter?

```
sm = rad2sm(2*pi, 'jupiter')
```

```
sm =  
2.7283e+005
```

## See Also

km2rad, deg2rad, rad2deg, deg2km, km2deg, km2nm, km2sm, deg2nm,  
nm2rad, nm2km, nm2sm, deg2sm, sm2rad, sm2km, sm2nm



**Purpose**

Radii of curvature of ellipsoid

**Syntax**

```
r = rcurve(ellipsoid,lat)
r = rcurve('parallel',ellipsoid,lat)
r = rcurve(ellipsoid,lat,units)
r = rcurve('meridian',ellipsoid,lat,units)
r = rcurve('transverse',ellipsoid,lat,units)
```

`r = rcurve(ellipsoid,lat)` and `r = rcurve('parallel',ellipsoid,lat)` return the parallel radius of curvature at the latitude `lat` for a given elliptical definition, where `ellipsoid` is a two-element ellipsoid vector. This is the radius of the small circle encompassing the ellipsoid at the given latitude. The radius is a distance in units consistent with the semimajor axis, the first element of `ellipsoid`.

`r = rcurve(ellipsoid,lat,units)` specifies the units of the input `lat`, where `units` is any valid angle units string. The default is 'degrees'.

`r = rcurve('meridian',ellipsoid,lat,units)` returns the meridional radius, which is the radius of curvature at the latitude `lat` for the ellipse described by a meridian on the ellipsoid.

`r = rcurve('transverse',ellipsoid,lat,units)` returns the transverse radius, which is the radius of a curve described by the intersection of the ellipsoid with a plane normal to the surface of the ellipsoid at the latitude `lat`.

**Examples**

The radii of curvature of the default ellipsoid at 45°, in kilometers:

```
r = rcurve('transverse',almanac('earth','ellipsoid','km'),...
          45,'degrees')
```

```
r =
    6.3888e+03
```

```
r = rcurve('meridian',almanac('earth','ellipsoid','km'),...
          45,'degrees')
```

```
r =  
  6.3674e+03
```

```
r = rcurve('parallel',almanac('earth','ellipsoid','km'),...  
          45,'degrees')
```

```
r =  
  4.5024e+03
```

### See Also

rsphere

**Purpose**

Read fields or records from fixed-format files

**Syntax**

```
struc = readfields(fname,fstruc)
struc = readfields(fname,fstruc,recordIDs)
struc = readfields(fname,fstruc,fieldIDs)
struc = readfields(fname,fstruc,recordIDs,mformat)
struc = readfields(fname,fstruc,recordIDs,mformat,fid)
struc = readfields(fname,fstruc,recordIDs,mformat,fid,
    'sparse')
```

`struc = readfields(fname,fstruc)` reads all the records from a fixed format file. *fname* is a string containing the name of the file. If it is empty, the file is selected interactively. *fstruc* is a structure defining the format of the file. The contents of *fstruc* are described below. The result is returned in a structure.

`struc = readfields(fname,fstruc,recordIDs)` reads only the records specified in the vector *recordIDs*. For example, *recordIDs* = [1 2 3 4]. All the fields in the selected records are read.

`struc = readfields(fname,fstruc,fieldIDs)` reads only the fields specified in the cell array *fieldIDs*. For example, *fieldIDs* = {1 2 4}. The selected fields are read from all the records. *fieldIDs* can be used in place of *recordIDs* in all calling forms.

`struc = readfields(fname,fstruc,recordIDs,mformat)` opens the file with the specified machine format. *mformat* must be recognized by `fopen`.

`struc = readfields(fname,fstruc,recordIDs,mformat,fid)` reads from a file that is already open. *fid* is the file identifier returned by `fopen`. The records are read starting from the current location in the file.

`struc = readfields(fname,fstruc,recordIDs,mformat,fid,'sparse')` disables error messages when the number of elements read does not agree with the stated format of the file. This is useful for formatted files with empty fields. Use *fid* = [] for files that are not already open. This option is only compatible with reading selected records.

# readfields

---

## Background

Map data is often provided as binary or ASCII files with a fixed format. Writing your own functions to read the data into MATLAB can be difficult and time-consuming, particularly for binary files. This function allows you to read the data by simply specifying the format of the file.

## Examples

Write a binary file and read it.

```
fid = fopen('testbin','wb');
for i = 1:3
    fwrite(fid,['character' num2str(i) ],'char');
    fwrite(fid,i,'int8');
    fwrite(fid,[i i],'int16');
    fwrite(fid,i,'integer*4');
    fwrite(fid,i,'real*8');
end
fclose(fid);

fs(1).length = 10;fs(1).type = 'char';fs(1).name = 'field 1';
fs(2).length = 1;fs(2).type = 'int8'; fs(2).name = 'field 2';
fs(3).length = 2;fs(3).type = 'int16';fs(3).name = 'field 3';
fs(4).length = 1;fs(4).type = 'integer*4';fs(4).name = 'field 4';
fs(5).length = 1;fs(5).type = 'float64'; fs(5).name = 'field 5';

s = readfields('testbin',fs);

s(1)
ans =
    field1: 'character1'
    field2: 1
    field3: [1 1]
    field4: 1
    field5: 1
```

## Limitations

Formatted numbers must stay within the width specified for them. Files must have a size that is an integer multiple of the computed record length. This is potentially a problem for formatted files on DOS platforms that use a carriage return/linefeed line ending everywhere

except the last record. File sizes are not checked when an open file is provided.

## Remarks

The format of the file is described in the input argument `fstruc`. `fstruc` is a structure with one entry for every field in the file. `fstruc` has three required fields: `length`, `name`, and `type`. For fields containing binary data of the type that would be read by `fread`, `length` is the number of elements to be read, `name` is a string containing the field name under which the read data is stored in the output structure, and `type` is a format string recognized by `fread`. Repetition modifiers such as `'40*char'` are *not* supported. Fields with empty field names are omitted from the output.

The following `fstruc` definition is for a file with a 40-character field, a field containing two integers, and a field with a single-precision floating-point number.

```
fstruc(1).length = 40;
fstruc(1).name = 'character Field'; % spaces will be suppressed
filestruc(1).type = 'char';

fstruc(2).length = 2;
fstruc(2).name = 'integer Field'; % spaces will be suppressed
fstruc(2).type = 'int16';

fstruc(3).length = 1;
fstruc(3).name = 'float Field'; % spaces will be suppressed
fstruc(3).type = 'real*4';
```

The `type` can also be a `fscanf` and `sscanf`-style format string of the form `'%nX'`, where `n` is the number of characters within which the formatted data is found, and `X` is the conversion character such as `'g'` or `'d'`. For formatted fields, the `length` entry in `fstruc` is the number of elements, each of which has the width specified in the `type` string. Fortran-style double-precision output such as `'0.0D00'` can be read using a `type` string such as `'%nD'`, where `n` is the number of characters per element. This is an extension to the C-style format strings accepted by `sscanf`. Users unfamiliar with C should note that `'%d'` is preferred

# readfields

---

over `'%i'` for formatted integers. MATLAB follows C in interpreting `'%i'` integers with leading zeros as octal. Line-ending characters in ASCII files must also be counted in the `fstruc` specification. Note that the number of line-ending characters differs across platforms.

A field specification for a formatted field with two integers each six characters wide would be of the form

```
fstruc(4).length = 2;  
fstruc(4).name = 'Elevation Units';  
fstruc(4).type = '%6d'
```

To summarize, `length` is the number of elements for binary numbers, the number of characters for strings, and the number of elements for formatted data.

You can omit fields from all output by providing an empty string for the `fstruc` name field.

## See Also

`grepfields`, `readmtx`, `textread`, `spread`, `d1mread`

**Purpose**

Read Fifth Fundamental Catalog of Stars

**Syntax**

```
struc = readfk5(filename)
```

```
struc = readfk5(filename, struc)
```

`struc = readfk5(filename)` reads the FK5 file and returns the contents in a structure. Each star is an element in the structure, with the different data items stored in appropriately named fields.

`struc = readfk5(filename, struc)` appends the data in the file to the existing structure `struc`.

**Background**

The Fifth Fundamental Catalog of Stars (FK5), Parts I and II, is a compilation of data on more than 4500 stars. The catalog contains positions, errors in positions, proper motions, and characteristics such as magnitudes, spectral types, parallaxes, and radial velocities. There are also cross-references to the identities of stars in other catalogs. It was compiled by researchers at the Astronomisches Rechen-Institut in Heidelberg.

**Remarks**

Positions are given in terms of right ascension and declination. Chapter 8, “Using Map Projections and Coordinate Systems” in the Mapping Toolbox documentation shows how to convert these to latitude and longitude for display by Mapping Toolbox.

The Fifth Fundamental Catalog of Stars (FK5), Parts I and II data and documentation are available over the Internet by anonymous ftp.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

**Examples**

```
FK5 = readfk5('FK5.dat');  
FK5e = readfk5('FK5_ext.dat');  
whos
```

# readfk5

---

Name	Size	Bytes	Class
FK5	1x1535	5042752	struct array
FK5e	1x3117	10226424	struct array

FK5e(1)

ans =

- FK5: 2003
- RAh: 0
- RAm: 5
- RAs: 1.1940
- pmRA: 0.6230
- DEd: 27
- DEm: 40
- DEs: 29.0100
- pmDE: -1.1100
- RAh1950: 0
- RAm1950: 2
- RAs1950: 26.5900
- pmRA1950: 0.6210
- DEd1950: 27
- DEm1950: 23
- DEs1950: 47.4400
- pmDE1950: -1.1100
- EpRA1900: 51.7200
- e\_RAs: 2
- e\_pmRA: 9
- EpDE1900: 46.8200
- e\_DEs: 3.4000
- e\_pmDE: 14
- Vmag: 6.4700
- n\_Vmag: ''
- SpType: 'G5'
- p1x: []
- RV: 12
- AGK3R: '38'
- SRS: ''
- HD: '225292'



DM: 'BD+26 4744'  
GC: '48'

**References** See references [5] and [6] in the Bibliography located at the end of this chapter.

**See Also** dms2degrees, scatterm

# readmtx

---

## Purpose

Read matrix stored in file

## Syntax

```
mtx = readmtx(fname,nrows,ncols,precision)
mtx =
readmtx(fname,nrows,ncols,precision,readrows,readcols)
mtx = readmtx(fname,nrows,ncols,precision,... readrows,
            readcols,mformat)
mtx = readmtx(fname,nrows,ncols,precision,... readrows,
            readcols,mformat,nheadbytes)
mtx = readmtx(fname,nrows,ncols,precision,... readrows,
            readcols,mformat,nheadbytes,nRowHeadBytes)
mtx = readmtx(fname,nrows,ncols,precision,... readrows,
```

```
readcols,mformat,nheadbytes,nRowHeadBytes,nRowTrailBytes)
mtx = readmtx(fname,nrows,ncols,precision,... readrows,
            readcols,mformat,nheadbytes,nRowHeadBytes,
            ... nRowTrailBytes,nFileTrailBytes)
mtx = readmtx(fname,nrows,ncols,precision,... readrows,
            readcols,mformat,nheadbytes,nRowHeadBytes,
            ... nRowTrailBytes,nFileTrailBytes,recordlen)
```

`mtx = readmtx(fname,nrows,ncols,precision)` reads a matrix stored in a file. The file contains only a matrix of numbers with the dimensions *nrows* by *ncols* stored with the specified *precision*. Recognized *precision* strings are described below.

```
mtx =
readmtx(fname,nrows,ncols,precision,readrows,readcols)
```

 reads a subset of the matrix. *readrows* and *readcols* specify which rows and columns are to be read. They can be vectors containing the row or column numbers, or two-element vectors of the form [start end], which are expanded using the colon operator to start:end. To read just two rows or columns, without expansion by the colon operator, provide the indices as a column matrix.

```
mtx = readmtx(fname,nrows,ncols,precision,...
            readrows,readcols,mformat)
```

 specifies the machine format used to write the file. *mformat* can be any string recognized by `fopen`. This

option is used to automatically swap bytes for files written on platforms with a different byte ordering.

```
mtx = readmtx(fname,nrows,ncols,precision,...  
readrows,readcols,mformat,nheadbytes) skips the file header,  
whose length is specified in bytes.
```

```
mtx = readmtx(fname,nrows,ncols,precision,...  
readrows,readcols,mformat,nheadbytes,nRowHeadBytes) also skips  
a header that precedes every row of the matrix. The length of the  
header is specified in bytes.
```

```
mtx = readmtx(fname,nrows,ncols,precision,...  
readrows,readcols,mformat,nheadbytes,nRowHeadBytes,nRowTrailBytes)  
also skips a trailer that follows every row of the matrix. The  
length of the trailer is specified in bytes.
```

```
mtx = readmtx(fname,nrows,ncols,precision,...  
readrows,readcols,mformat,nheadbytes,nRowHeadBytes,...  
nRowTrailBytes,nFileTrailBytes) accounts for the length of data  
following the matrix. The sizes of the components of the matrix are  
used to compute an expected file size, which is compared to the actual  
file size.
```

```
mtx = readmtx(fname,nrows,ncols,precision,...  
readrows,readcols,mformat,nheadbytes,nRowHeadBytes,...  
nRowTrailBytes,nFileTrailBytes,recordlen) overrides the record  
length calculated from the precision and number of columns, and  
instead uses the record length given in bytes. This is used for formatted  
data with extra spaces or line breaks in the matrix.
```

## Background

Map data is often provided as binary or ASCII files with a fixed format. Writing your own functions to read the data into MATLAB can be difficult and time-consuming, particularly for binary files. This function allows you to read the data by simply specifying the format of the file.

## Examples

Write and read a binary matrix file:

```
fid = fopen('binmat','w');
```

# readmtx

---

```
fwrite(fid,1:100,'int16');  
fclose(fid);  
mtx = readmtx('binmat',10,10,'int16')
```

```
mtx =  
    1     2     3     4     5     6     7     8     9    10  
   11    12    13    14    15    16    17    18    19    20  
   21    22    23    24    25    26    27    28    29    30  
   31    32    33    34    35    36    37    38    39    40  
   41    42    43    44    45    46    47    48    49    50  
   51    52    53    54    55    56    57    58    59    60  
   61    62    63    64    65    66    67    68    69    70  
   71    72    73    74    75    76    77    78    79    80  
   81    82    83    84    85    86    87    88    89    90  
   91    92    93    94    95    96    97    98    99   100
```

```
mtx = readmtx('binmat',10,10,'int16',[2 5],3:2:9)
```

```
mtx =  
   13    15    17    19  
   23    25    27    29  
   33    35    37    39  
   43    45    47    49
```

## Limitations

Every row of the matrix must have the same number of elements.

## Remarks

This function reads files that have a general format consisting of a header, a matrix, and a trailer. Each row of the matrix can have a certain number of bytes of extraneous information preceding or following the matrix data.

Both binary and formatted data files can be read. If the file is binary, the precision argument is a format string recognized by `fread`. Repetition modifiers such as `'40*char'` are *not* supported. If the file is formatted, precision is a `fscanf` and `sscanf`-style format string of the form `'%nX'`, where `n` is the number of characters within which the formatted data is found, and `X` is the conversion character such as `'g'`

or 'd'. Fortran-style double-precision output such as '0.0D00' can be read using a precision string such as '%nD', where n is the number of characters per element. This is an extension to the C-style format strings accepted by `sscanf`. Users unfamiliar with C should note that '%d' is preferred over '%i' for formatted integers. MATLAB follows C in interpreting '%i' integers with leading zeros as octal. Formatted files with line endings need to provide the number of trailing bytes per row, which can be 1 for platforms with carriage returns *or* linefeed (Macintosh, UNIX), or 2 for platforms with carriage returns *and* linefeeds (DOS).

**See Also**

`readfields`, `textread`, `spcread`, `dlmread`

## Purpose

Point at specified azimuth, range on sphere or ellipsoid

## Syntax

```
[latout, lonout] = reckon(lat, lon, rng, az)
[latout, lonout] = reckon(lat, lon, rng, az, units)
[latout, lonout] = reckon(lat, lon, rng, az, ellipsoid)
[latout, lonout] = reckon(lat, lon, rng, az, ellipsoid,
    units)
```

```
[latout, lonout] = reckon(track,...)
```

[latout, lonout] = reckon(lat, lon, rng, az), for scalar inputs, calculates a position (latout, lonout) at a given range rng and azimuth az along a great circle from a starting point defined by lat and lon. lat and lon are in degrees. The range is in degrees of arc length on a sphere. The input azimuth is in degrees, measured clockwise from due north. reckon calculates multiple positions when given four non-scalar inputs of matching size.

[latout, lonout] = reckon(lat, lon, rng, az, units), where units is any valid angle units string, specifies the angular units of the inputs and outputs, including rng. The default value is 'degrees'.

[latout, lonout] = reckon(lat, lon, rng, az, ellipsoid) calculates positions along a geodesic on an ellipsoid, as specified by the two-element vector ellipsoid. The range, rng, is in linear distance units matching the units of the semimajor axis of the ellipsoid (the first element of ellipsoid).

[latout, lonout] = reckon(lat, lon, rng, az, ellipsoid, units) calculates positions on the specified ellipsoid with lat, lon, az, latout, and lonout in the specified angle units.

[latout, lonout] = reckon(track,...) calculates positions on great circles (or geodesics) if track is 'gc' and along rhumb lines if track is 'rh'. The default value is 'gc'.

## Examples

What are the coordinates of the point 600 nautical miles northwest of London, UK (51.5°N,0°), in a great circle sense?

```
dist = nm2deg(600) % convert nm distance to degrees
```

```
dist =  
    9.9933  
  
pt1 = reckon(51.5,0,dist,315) % northwest is 315 degrees  
  
pt1 =  
    57.8999  -13.3507
```

Now, where would a plane taking off from London and traveling on a constant northwesterly course for 600 nautical miles end up?

```
pt2 = reckon('rh',51.5,0,dist,315)  
  
pt2 =  
    58.5663  -12.3699
```

How far apart are these points (distance in great circle sense)?

```
separation = distance('gc',pt1,pt2)  
  
separation =  
    0.8430  
  
nmsep = deg2nm(separation) % convert answer to nautical miles  
  
nmsep =  
    50.6156
```

Over 50 nautical miles separate the two points.

## See Also

azimuth, distance, distance, km2deg, dreckon, track, track1, track2

# reducem

---

## Purpose

Reduce density of points in vector data

## Syntax

```
[latout,lonout] = reducem(latin,lonin)
[latout,lonout] = reducem(latin,lonin,tol)
[latout,lonout,cerr] = reducem(...)
[latout,lonout,cerr,tol] = reducem(...)
```

`[latout,lonout] = reducem(latin,lonin)` reduces the number of points in vector map data. In this case the tolerance is computed automatically.

`[latout,lonout] = reducem(latin,lonin,tol)` uses the provided tolerance. The units of the tolerance are degrees of arc on the surface of a sphere.

`[latout,lonout,cerr] = reducem(...)` in addition returns a measure of the error introduced by the simplification. The output `cerr` is the difference in the arc length of the original and reduced data, normalized by the original length.

`[latout,lonout,cerr,tol] = reducem(...)` also returns the tolerance used in the reduction, which is useful when the tolerance is computed automatically.

## Example

Compare the original and reduced outlines of the District of Columbia from the `usastatehi` demo state outline data:

```
dc = shaperead('usastatehi',...
    'UseGeoCoords', true,...
    'Selector',{@(name) ...
        strcmpi(name,'district of columbia'), 'Name'});
lat = extractfield(dc, 'Lat');
lon = extractfield(dc, 'Lon');
[latreduced, lonreduced] = reducem(lat, lon);

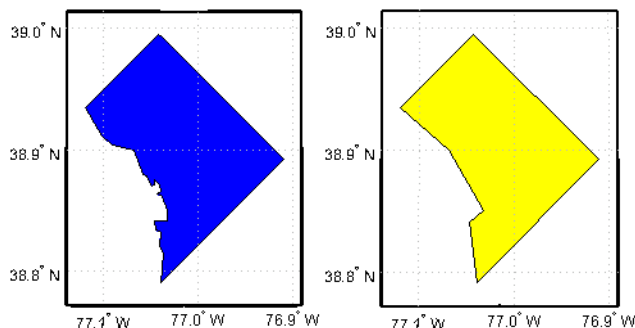
lonlim = dc.BoundingBox(:,1)' + [-0.02 0.02];
latlim = dc.BoundingBox(:,2)' + [-0.02 0.02];

subplot(1,2,1)
```



```
usamap(latlim, lonlim); axis off
geoshow(lat, lon,...
        'DisplayType', 'polygon', 'FaceColor', 'blue')

subplot(1,2,2)
usamap(latlim, lonlim); axis off
geoshow(latreduced, lonreduced,...
        'DisplayType', 'polygon', 'FaceColor', 'yellow')
```



## Remarks

Vector data is reduced using the Douglas-Peucker line simplification algorithm. This method recursively subdivides a polygon until a run of points can be replaced by a straight line segment, with no point in that run deviating from the straight line by more than the tolerance. The distances used to decide on which runs of points to eliminate are computed in a Plate Carrée projection.

Reduced geographic data might not always be appropriate for display. If all intermediate points in a data set are reduced, then lines appearing straight in one projection are incorrectly displayed as straight lines in others.

# reducem

---

## See Also

interp

Interpolate vector data to a specified data separation

resizem

Resize a data grid

**Purpose** Convert referencing matrix to referencing vector

**Syntax** `refvec = refmat2vec(R,s)`

`refvec = refmat2vec(R,s)` converts a referencing matrix, `R`, to the three-element referencing vector `refvec`. `R` is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates. `s` is the size of the array (data grid) that is being referenced. `refvec` is a 1-by-3 referencing vector having elements [cells/degree north-latitude west-longitude] with latitude and longitude limits specified in degrees.

**Example**

```
% Verify the conversion of the geoid referencing vector to a
% referencing matrix.
load geoid;
R = refvec2mat(geoidlegend, size(geoid));
V = refmat2vec(R, size(geoid));
```

**See Also** `makerefmat`, `refvec2mat`

# refvec2mat

---

**Purpose** Convert referencing vector to referencing matrix

**Syntax** `R = refvec2mat(refvec,s)`

`R = refvec2mat(refvec,s)` converts a referencing vector, `refvec`, to the referencing matrix `R`. `refvec` is a 1-by-3 referencing vector having elements [cells/degree north-latitude west-longitude] with latitude and longitude limits specified in degrees. `s` is the size of the array (data grid) that is being referenced. `R` is a 3-by-2 referencing matrix defining a two-dimensional affine transformation from pixel coordinates to spatial coordinates.

**Example**

```
% Convert the geoid referencing vector to a referencing matrix
load geoid;
R = refvec2mat(geoidlegend, size(geoid));
```

**See Also** `makerefmat`, `refmat2vec`

## Purpose

Clean up NaN separators in polygons and lines

## Syntax

```
[xdata, ydata] = removeExtraNaNSeparators(xdata,ydata)
[xdata, ydata, zdata] = removeExtraNaNSeparators(xdata,ydata,
    zdata)
```

[xdata, ydata] = removeExtraNaNSeparators(xdata,ydata)  
removes NaNs from the vectors xdata and ydata, leaving only isolated NaN separators. If present, one or more leading NaNs are removed entirely. If present, a single trailing NaN is preserved. NaNs are removed, but never added, so if the input lacks a trailing NaN, so will the output. xdata and ydata must match in size and have identical NaN locations.

```
[xdata, ydata, zdata] =
removeExtraNaNSeparators(xdata,ydata,zdata)
removes NaNs from the vectors xdata, ydata, and zdata, leaving
only isolated NaN separators and optionally, if consistent with the
input, a single trailing NaN.
```

## Examples

```
xin = [NaN NaN 1:3 NaN 4:5 NaN NaN NaN 6:9 NaN NaN];
yin = xin;
[xout, yout] = removeExtraNaNSeparators(xin, yin);
xout
```

```
xout =
    1  2  3  NaN  4  5 NaN  6  7  8  9  NaN
```

```
xin = [NaN 1:3 NaN NaN 4:5 NaN NaN NaN 6:9]';
yin = xin;
zin = xin;
[xout, yout, zout] = removeExtraNaNSeparators(xin, yin, zin);
xout
```

```
xout =
    1
    2
    3
    NaN
```

# removeExtraNaNSeparators

---

4  
5  
NaN  
6  
7  
8  
9

**Purpose**

Resize regular data grid

**Syntax**

```
newgrid = resizing(Z,m)
newgrid = resizing(Z,[r c])
[newgrid newrefvec] = resizing(Z,m,refvec)
[...] = resizing(...,method)
[...] = resizing(...,method,n)
[...] = resizing(...,method,h)
```

`newgrid = resizing(Z,m)` resizes a regular data grid, `grid`, by a resizing factor `m`. `resizing` uses interpolation to resample to a new sample density/cell size, and returns a new grid that is `m` times the size of `Z`. If `m` is between 0 and 1, the size of `newgrid` is smaller than the size of `Z`. If `m` is greater than 1, the size of `newgrid` is larger. For example, if `m` is 0.5, the number of rows and the number of columns will be halved. By default, `resizing` uses nearest neighbor interpolation.

`newgrid = resizing(Z,[r c])` resizes `Z` to have `r` rows and `c` columns. `r` and `c` must be positive whole numbers.

`[newgrid newrefvec] = resizing(Z,m,refvec)` resizes a regular data grid with a three-element referencing vector, `refvec`, and returns a new grid and its referencing vector, `newrefvec`. In this case `m` must be a scalar resizing factor; it may not be a vector of the form `[r c]`, because using referencing vectors implies that grid cells are squares of latitude and longitude.

`[...] = resizing(...,method)` resizes a regular data grid using one of the following three interpolation methods:

'nearest'	nearest neighbor interpolation (default)
'bilinear'	bilinear interpolation
'bicubic'	bicubic interpolation

If the grid size is being reduced (`m` is less than 1 or `[r c]` is less than the size of the input grid) and `method` is 'bilinear' or 'bicubic', `resizing` applies a lowpass filter before interpolation, to reduce aliasing. The default filter size is 11-by-11.

# resize

---

`[...] = resize(...,method,n)` permits you to specify a specific length for the default filter. `n` is an integer scalar specifying the size of the filter, which is `n`-by-`n`. If `n` is 0 or `method` is 'nearest', `resize` omits the filtering step.

`[...] = resize(...,method,h)` permits you to specify your own filter, `h`, which can be any two-dimensional FIR filter (such as those returned by Image Processing Toolbox functions `ftrans2`, `fwind1`, `fwind2`, or `fsamp2`). If `h` is specified, filtering is applied even when `method` is 'nearest'.

## Example

Double the size of a grid then reduce it using different methods:

```
Z = [1 2; 3 4]
```

```
Z =  
    1  2  
    3  4
```

```
neargrid = resize(Z,2)
```

```
neargrid =  
    1  1  2  2  
    1  1  2  2  
    3  3  4  4  
    3  3  4  4
```

```
bilingrid = resize(Z,2,'bilinear')
```

```
bilingrid =  
    1.0000    1.3333    1.6667    2.0000  
    1.6667    2.0000    2.3333    2.6667  
    2.3333    2.6667    3.0000    3.3333  
    3.0000    3.3333    3.6667    4.0000
```

```
bicubgrid = resize(bilingrid,[3 2],'bicubic')
```



```
bicubgrid =  
    0.7406    1.2994  
    1.6616    2.3462  
    1.9718    2.5306
```

**See Also** [filter2](#) (MATLAB function)

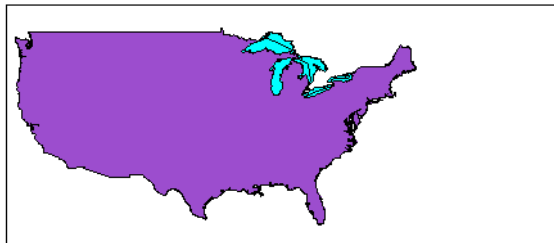
**Purpose** Restack objects within map axes

**Syntax** `restack(h,position)`

`restack(h,position)` changes the stacking position of the object `h` within the axes. `h` can be a handle, a vector of handles to graphics objects, or a name string recognized by `handlem`. Recognized *position* strings are 'top', 'bottom', 'bot', 'up', or 'down'.

**Examples** Restack the great lakes to lie on top of conus:

```
figure; axesm miller
load conus
h = geoshow(gtlakelat, gtlakelon,...
    'DisplayType', 'polygon', 'FaceColor', 'cyan');
geoshow(uslat, uslon,...
    'DisplayType', 'polygon', 'FaceColor', [0.6 0.3 0.8])
% The great lakes were plotted first but need to be on top
% Cast handle to great lakes object to double in call to RESTACK
restack(double(h),'top')
```



**Remarks** This function is the command line equivalent of the stacking buttons in the `mobjects` graphical user interface. The stacking order is the order of the children of the axes.

**See Also** `mobjects`

**Purpose**

Intersection points for pairs of rhumb lines

**Syntax**

```
[newlat,newlong] = rhxrh(lat1,lon1,az1,lat2,lon2,az2)
[newlat,newlon] = rhxrh(lat1,lon1,az1,lat2,lon2,az2,units)
```

[newlat,newlong] = rhxrh(lat1,lon1,az1,lat2,lon2,az2) returns in newlat and newlon the location of the intersection point for each pair of rhumb lines input in *rhumb line notation*. For example, the first line in the pair passes through the point (lat1,lon1) and has a constant azimuth of az1. When the two rhumb lines are identical or do not intersect (conditions that are not, in general, apparent by inspection), two NaNs are returned instead and a warning is displayed. The inputs must be column vectors.

[newlat,newlon] = rhxrh(lat1,lon1,az1,lat2,lon2,az2,units) specifies the units used, where *units* is any valid units string. The default units are 'degrees'.

**Description**

For any pair of rhumb lines, there are three possible intersection conditions: the lines are identical, they intersect once, or they do not intersect at all (except at the poles, where all nonequatorial rhumb lines meet—this is not considered an intersection). rhxrh does not allow multiple rhumb line intersections, although it is possible to construct cases in which such a condition occurs. See the following discussion of Limitations.

*Rhumb line notation* consists of a point on the line and the constant azimuth of the line.

**Examples**

Given a starting point at (10°N,56°W), a plane maintains a constant heading of 35°. Another plane starts at (0°,10°W) and proceeds at a constant heading of 310° (−50°). Where would their two paths cross each other?

```
[newlat,newlong] = rhxrh(10,-56,35,0,-10,310)
```

```
newlat =
    26.9774
```

```
newlong =  
-43.4088
```

## Limitations

Rhumb lines are specifically helpful in navigation because they represent lines of constant heading, whereas great circles have, in general, continuously changing heading. In fact, the Mercator projection was originally designed so that rhumb lines plot as straight lines, which facilitates both manual plotting with a straightedge and numerical calculations using a Cartesian planar representation. When a rhumb line proceeds off the left or right *edge* of this representation at some latitude, it reappears on the other edge at the same latitude and continues on the same slope. For rhumb lines where this occurs—for example, one with a heading of 85°—it is easy to imagine another rhumb line, say one with a heading of 0°, repeatedly intersecting the first. The real-world uses of rhumb lines make this merely an intellectual exercise, however, for in practice it is always clear which *crossing* line segment is relevant. The function `rhxrh` returns at most one intersection, selecting in each case that line segment containing the input starting point for its computation.

## See Also

`gcxgc`, `gcxsc`, `scxsc`, `crossfix`, `polyxpoly`, `navfix`

**Purpose** Construct cell array of workspace variables for `mlayers` tool

**Syntax** `rootlayr`

`rootlayr` allows the `mlayers` tool to be used with workspace variables. It constructs a cell array that contains all the structure variables in the current workspace. This cell array is returned in the variable `ans`, which can then be an input to `mlayers`. If there is an existing variable named `ans`, it is overwritten.

The recommended calling procedure is `rootlayr;mlayers(ans);`

## Examples

`rootlayr` creates a cell array named `ans`, consisting of the three structure variables in the following workspace.

```
whos
  Name           Size           Bytes  Class
  borders        1x1             38390  struct array
  lats           2345x1          18760  double array
  lons           2345x1          18760  double array
  nation         1x1             70224  struct array
  states         1x51            254970  struct array
```

```
rootlayr
ans
ans =
    [1x1 struct]    'borders'
    [1x1 struct]    'nation'
    [1x51 struct]   'states'
```

The function `mlayers(ans)` can now be used to activate the `mlayers` tool for the structures contained in `ans`.

**See Also** `mlayers`

# rotatem

---

## Purpose

Transform vector map data to new origin and orientation

## Syntax

```
[lat1,lon1] = rotatem(lat,lon,origin,'forward')  
[lat1,lon1] = rotatem(lat,lon,origin,'inverse')  
[lat1,lon1] = rotatem(lat,lon,origin,'forward',units)  
[lat1,lon1] = rotatem(lat,lon,origin,'forward',units)
```

[lat1,lon1] = rotatem(lat,lon,origin,'forward') transforms latitude and longitude data (lat and lon) to their new coordinates (lat1 and lon1) in a coordinate system resulting from Euler angle rotations as specified by origin. The input origin is a three- (or two-) element vector having the form [latitude longitude orientation]. The latitude and longitude are the coordinates of the point in the original system, which is the center of the output system. The orientation is the azimuth from the new origin point to the original North Pole in the new system. If origin has only two elements, the orientation is assumed to be 0°. This origin vector might be the output of putpole or newpole.

[lat1,lon1] = rotatem(lat,lon,origin,'inverse') transforms latitude and longitude data (lat and lon) in a coordinate system *that has been transformed* by Euler angle rotations specified by origin to their coordinates (lat1 and lon1) in the coordinate system *from which they were originally transformed*. In a sense, this *undoes* the 'forward' process. Be warned, however, that if data is rotated forward and then inverted, the final data might not be identical to the original. This is because of roundoff and *data collapse* at the original and intermediate singularities (the poles).

[lat1,lon1] = rotatem(lat,lon,origin,'forward',units) and [lat1,lon1] = rotatem(lat,lon,origin,'forward',units) specify the angle units of the data, where *units* is any recognized angle units string. The default is 'radians'. Note that this default is different from that of most functions.

## Description

The rotatem function transforms vector map data to a new coordinate system.

An analytical use of the new data can be realized in conjunction with the newpole function. If a selected point is made the *north pole* of

the new system, then when new vector data is created with `rotatem`, the distance of every data point from this new north pole is its new colatitude ( $90^\circ$  minus latitude). The absolute difference in the great circle azimuths between every pair of points from their new *pole* is the same as the difference in their new longitudes.

## Examples

What are the coordinates of Rio de Janeiro ( $23^\circ\text{S}, 43^\circ\text{W}$ ) in a coordinate system in which New York ( $41^\circ\text{N}, 74^\circ\text{W}$ ) is made the North Pole? Use the `newpole` function to get the origin vector associated with putting New York at the pole:

```
nylat = 41; nylon = -74;
riolat = -23; riolon = -43;
origin = newpole(nylat, nylon);
[riolat1, riolon1] = rotatem(riolat, riolon, origin, ...
                            'forward', 'degrees')

riolat1 =
    19.8247
riolon1 =
   -149.7375
```

What does this mean? For one thing, the colatitude of Rio in this new system is its distance from New York. Compare the distance between the original points and the new colatitude:

```
dist = distance(nylat, nylon, riolat, riolon)

dist =
    70.1753

90-riolat1

ans =
    70.1753
```

## See Also

`neworig`, `newpole`, `org2pol`, `putpole`

# rotatetext

---

**Purpose** Rotate text to projected graticule

**Syntax** `rotatetext`  
`rotatetext(objects)`  
`rotatetext(objects, 'inverse')`

`rotatetext` rotates displayed text objects to account for the curvature of the graticule. The objects are selected interactively from a graphical user interface.

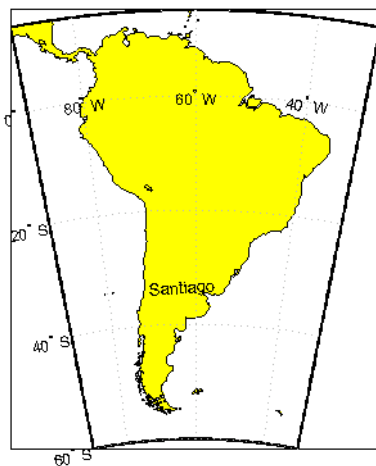
`rotatetext(objects)` rotates the selected objects. `objects` can be a name string recognized by `handlem` or a vector of handles to displayed text objects.

`rotatetext(objects, 'inverse')` removes the rotation added by an earlier use of `rotatetext`. If omitted, `'forward'` is assumed.

**Examples** Add text to a map and rotate the text to the graticule.

```
figure
worldmap('south america')
geoshow('landareas.shp','facecolor','yellow')
cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
Santiago = strmatch('Santiago',{cities(:).Name});
h=textm(cities(Santiago).Lat, cities(Santiago).Lon, ...
        'Santiago');
rotatetext(h)
```



**Remarks**

You can rotate meridian and parallel labels automatically by setting the map axes LabelRotation property to 'on'.

**See Also**

vfdtran, vinvtran

# roundn

---

**Purpose** Round numbers to specified power of 10

**Syntax**

```
outnum = roundn(innum)
outnum = roundn(innum,n)
```

outnum = roundn(innum) rounds the elements of innum to the nearest one-hundredth.

outnum = roundn(innum,n) specifies the power of 10 to which the elements of innum are rounded. For example, if n = 2, round to the nearest hundred (10<sup>2</sup>).

**Examples** Using generated numbers, round them to significant tenths, ones, and tens figures (note that your original numbers could differ):

```
fullfig = 1000*magic(2)/7

fullfig =
  142.8571  428.5714
  571.4286  285.7143

tenths = roundn(fullfig,-1)

tenths =
  142.9000  428.6000
  571.4000  285.7000

units = roundn(fullfig,0)

units =
  143  429
  571  286

tens = roundn(fullfig,1)

tens =
  140  430
    570  290
```

**See Also**

epsm

# rsphere

---

## Purpose

Radii of auxiliary spheres

## Syntax

```
r = rsphere('biaxial',ellipsoid)
r = rsphere('biaxial',ellipsoid,method)
r = rsphere('triaxial',ellipsoid)
r = rsphere('eqavol',ellipsoid)
r = rsphere('authalic',ellipsoid)
r = rsphere('rectifying',ellipsoid)
r = rsphere('curve',ellipsoid,lat,method,units)
r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid)
r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid,units)
```

`r = rsphere('biaxial',ellipsoid)` calculates the radius of a biaxial auxiliary sphere for the ellipsoid specified by the two-element ellipsoid vector `ellipsoid`. The output, `r`, is the radius of this sphere in units consistent with the semimajor axis, that is, the first element of `ellipsoid`. The biaxial radius is calculated by averaging the semimajor and semiminor axes of the ellipsoid, giving each equal weight.

`r = rsphere('biaxial',ellipsoid,method)` specifies the averaging method. If the string `method` is 'mean' (the default), an arithmetic mean is used. If `method` is 'norm', a geometric mean is used.

`r = rsphere('triaxial',ellipsoid)` results in a triaxial radius, which is calculated by averaging the ellipsoidal axes while giving double weight to the semimajor axis to reflect its role in two of the ellipsoid's three dimensions.

`r = rsphere('eqavol',ellipsoid)` returns the radius of a sphere with a volume equal to that of the ellipsoid.

`r = rsphere('authalic',ellipsoid)` returns the radius of a sphere with a surface area equal to that of the ellipsoid.

`r = rsphere('rectifying',ellipsoid)` returns the radius of a sphere with meridional distances equal to those of the ellipsoid.

`r = rsphere('curve',ellipsoid,lat,method,units)` returns a radius that is the result of averaging the meridional and transverse radii of curvature at the specified latitude, `lat`. The units of the input

lat can be specified by the valid angle units string *units*. The default units are 'degrees', the default averaging method is 'mean', and the default latitude is 45°.

```
r = rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid) and r
= rsphere('euler',lat1,lon1,lat2,lon2,ellipsoid,units)
calculate a radius using Euler's Theorem. This calculation requires
the specification of an arc that is defined by its endpoints, (lat1,lon1)
and (lat2,lon2).
```

## Description

The `rsphere` function calculates the radii of auxiliary spheres for the ellipsoid. An auxiliary sphere is a sphere that shares certain desired characteristics with the ellipsoid.

## Examples

Different criteria result in different spheres:

```
r = rsphere('biaxial',almanac('earth','ellipsoid','km'))
```

```
r =
  6.3674e+03
```

```
r = rsphere('triaxial',almanac('earth','ellipsoid','km'))
```

```
r =
  6.3710e+03
```

```
r = rsphere('curve',almanac('earth','ellipsoid','km'))
```

```
r =
  6.3781e+03
```

## See Also

`rcurve`

## Purpose

Read 2-minute terrain/bathymetry from Smith and Sandwell

## Syntax

```
[latgrat, longrat, z] = satbath
[latgrat, longrat, z] = satbath(scalefactor)
[latgrat, longrat, z] = satbath(scalefactor, latlim, lonlim)
[latgrat, longrat, z] = satbath(scalefactor, latlim, lonlim,
    gsize)
```

`[latgrat, longrat, z] = satbath` reads the global topography file for the entire world (`topo_8.2.img`), returning every 50<sup>th</sup> point. The result is returned as a geolocated data grid. If you use a different version of the global topography file, you need to rename it to “`topo_8.2.img`”. If the file is not found on the MATLAB path, a dialog opens to request the file.

`[latgrat, longrat, z] = satbath(scalefactor)` returns the data for the entire world, subsampled by the integer `scalefactor`. A `scalefactor` of 10 returns every 10th point. The matrix at full resolution has 6336 by 10800 points.

`[latgrat, longrat, z] = satbath(scalefactor, latlim, lonlim)` returns data for the specified region. The returned data extends slightly beyond the requested area. If omitted, the entire area covered by the data file is returned. The limits are two-element vectors in units of degrees, with `latlim` in the range `[-90 90]` and `lonlim` in the range `[-180 180]`.

`[latgrat, longrat, z] = satbath(scalefactor, latlim, lonlim, gsize)` controls the size of the graticule matrices. `gsize` is a two-element vector containing the number of rows and columns desired. If omitted, a graticule the size of the data grid is returned.

## Background

This is a global bathymetric model derived from ship soundings and satellite altimetry by W.H.F. Smith and D.T. Sandwell. The model was developed by iteratively adjusting gravity anomaly data from Geosat and ERS-1 against historical track line soundings. This technique takes advantage of the fact that gravity mirrors the large variations in the ocean floor as small variations in the height of the ocean's surface. The computational procedure uses the ship track line data to

calibrate the scaling between the observed surface undulations and the inferred bathymetry. Land elevations are reduced-resolution versions of GTOPO30 data.

## Remarks

Land elevations are given in meters above mean sea level. The data is stored in a Mercator projection grid. As a result, spatial resolution varies with latitude. The grid spacing is 2 minutes (about 4 kilometers) at the equator.

This data is available over the Internet, but subject to copyright. The data file is binary, and should be transferred with no line-ending conversion or byte swapping. This function carries out any byte swapping that might be required. The data requires about 133 MB uncompressed.

The data and documentation are available over the Internet via http and anonymous ftp. Download the latest version of file `topo_x.2.img`, where `x` is the version number, and rename it `topo_8.w.img` for compatibility with the `satbath` function.

`satbath` returns a geolocated data grid rather than a regular data grid and a referencing vector or matrix. This is because the data is in a Mercator projection, with columns evenly spaced in longitude, but with decreasing spacing for rows at higher latitudes. Referencing vectors and matrices assume that the number of cells per degrees of latitude and longitude are both constant across a data grid.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

Read the data for the Falklands Islands (Islas Malvinas) at full resolution.

```
[latgrat,longrat,mat] = satbath(1,[-55 -50],[-65 -55]);  
whos
```

# satbath

---

Name	Size	Bytes	Class
latgrat	247x301	594776	double array
longrat	247x301	594776	double array
mat	247x301	594776	double array

**See Also**      tbase, gtopo30, egm96geoid



**Purpose**

Add or modify graphic scale on map axes

**Syntax**

```
scaleruler
scaleruler on
scaleruler off
scaleruler(property,value,...)
h = scaleruler(...)
```

`scaleruler` toggles the display of a graphic scale. If no graphic scale is currently displayed in the current map axes, one is added. If any graphic scales are currently displayed, they are removed.

`scaleruler on` adds a graphic scale to the current map axes. Multiple graphic scales can be added to the same map axes.

`scaleruler off` removes any currently displayed graphic scales.

`scaleruler(property,value,...)` adds a graphic scale and sets the properties to the values specified. You can display a list of graphic scale properties using the command `setm(h)`, where `h` is the handle to a graphic scale object. The current values for a displayed graphic scale object can be retrieved using `getm`. The properties of a displayed graphic scale object can be modified using `setm`.

`h = scaleruler(...)` returns the `hggroup` handle to the graphic scale object.

**Background**

Cartographers often add graphic elements to the map to indicate its scale. Perhaps the most commonly used is the graphic scale, a ruler-like object that shows distances on the ground at the correct size for the projection.

**Examples**

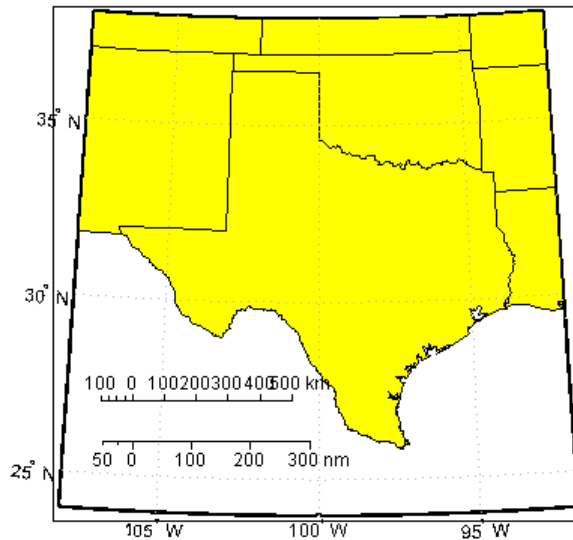
Create a map, add a graphic scale with the default settings, and shift it up a bit. Add a second scale showing nautical miles, and change the tick marks and direction.

```
figure
usamap('Texas')
geoshow('usastatelo.shp', 'FaceColor', [0.9 0.9 0])
```

# scaleruler

---

```
scaleruler on
setm(handlem('scaleruler1'),'YLoc',.5)
scaleruler('units','nm')
setm(handlem('scaleruler2'), ...
    'YLoc', .48, ...
    'MajorTick', 0:100:300,...
    'MinorTick', 0:25:50, ...
    'TickDir', 'down', ...
    'MajorTickLength', km2nm(25),...
    'MinorTickLength', km2nm(12.5))
```



## Remarks

You can reposition graphic scale objects by dragging them with the mouse. You can also change their positions by modifying the XLoc and YLoc properties using setm.

Modifying the properties of the graphic scale results in the replacement of the original object (dragging a scaleruler, however, does not replace it). For this reason, handles to the graphic scale object will change. Use handlem('scaleruler') to get a list of the current handles to all

graphic scale objects. Use `handlem('scalerulerN')`, where N is an integer, to get the handle to a particular graphic scale. Use `namem` to see the names of existing graphic scale objects. The name of a graphic scale object is also stored in the read-only 'Children' property, which is accessed using `getm`.

Use `scaleruler off`, `clmo scaleruler`, or `clmo scalerulerN` to remove the scale rulers. You can also remove a graphic scale object with `delete(h)`, or `delete(handlem('scalerulerN'))`, where N is the corresponding integer.

## Object Properties

### Properties That Control Appearance

#### Color

`ColorSpec {no default}`

*Color of the displayed graphic scale* — Controls the color of the graphic scale lines and text. You can specify a color using a vector of RGB values or one of the MATLAB predefined names. By default, the graphic scale is displayed in black ([0 0 0]).

#### FontAngle

`{normal} | italic | oblique`

*Angle of the graphic scale label text* — Controls the appearance of the graphic scale text components. Use any font angle string recognized by MATLAB.

#### FontName

`courier | {helvetica} | symbol | times`

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

#### FontSize

`scalar in units specified in FontUnits {9}`

# scaleruler

---

*Font size* — Specifies the font size to use for all displayed graphic scale labels, in units specified by the `FontUnits` property. The default point size is 9.

`FontUnits`

inches | centimeters | normalized | {points} | pixels

*Units used to interpret the `FontSize` property* — When set to `normalized`, the toolbox interprets the value of `FontSize` as a fraction of the height of the axes. For example, a `normalized` `FontSize` of 0.16 sets the text characters to a font whose height is one-tenth of the axes' height. The default units, points, are equal to 1/72 of an inch.

`FontWeight`

light | {normal} | demi | bold

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

`Label`

string

*Label text for the graphic scale* — Contains a string used to label the graphic scale. The text is displayed centered on the scale. The label is often used to indicate the scale of the map, for example “1:50,000,000.”

`LineWidth`

scalar {0.5}

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

`MajorTick`

vector

*Graphic scale major tick locations* — Sets the major tick locations for the graphic scale. The default values are chosen to give a reasonably sized scale. You can specify the locations of the tick marks by providing a vector of locations. These are usually equally spaced values as generated by `start:step:end`. The values are distances in the units of the `Units` property.

`MajorTickLabel`  
Cell array of strings

*Graphic scale major tick labels* — Sets the text labels associated with the major tick locations. By default, the labels are identical to the major tick locations. You can override these by providing a cell array of strings. There must be as many strings as tick locations.

`MajorTickLength`  
scalar

*Length of the major tick lines* — Controls the length of the major tick lines. The length is a distance in the units of the `Units` property.

`MinorTick`  
vector

*Graphic scale minor tick locations* — Sets the minor tick locations for the graphic scale. The default values are chosen to give a reasonably sized scale. You can specify the locations of the tick marks by providing a vector of locations. These are usually equally spaced values as generated by `start:step:end`. The values are distances in the units of the `Units` property.

`MinorTickLabel`  
strings

*Graphic scale minor tick labels* — Sets the text labels associated with the minor tick locations. By default, the label is identical to the last minor tick location. You can override this by providing a string label.

MinorTickLength  
scalar

*Length of the minor tick lines* — Controls the length of the minor tick lines. The length is a distance in the units of the Units property.

RulerStyle  
{ruler} | lines | patches

*Style of the graphic scale* — Selects among three different kinds of graphic scale displays. The default ruler style looks like the MATLAB *x*-axis. The lines style has three horizontal lines across the tick marks. This type of graphic scale is often used on maps from the U.S. Geological Survey. The patches style has alternating black and white rectangles in place of lines and tick marks.

TickDir  
{up} | down

*Direction of the tick marks and text* — Controls the direction in which the tick marks and text labels are drawn. In the default up direction, the tick marks and text labels are placed above the baseline, which is placed at the location given in the XLoc property. In the down position, the tick marks and labels are drawn below the baseline.

TickMode  
{auto} | manual

*Tick locations mode* — Controls whether the tick locations and labels are computed automatically or are user-specified. Explicitly setting the tick labels or locations results in a 'manual' tick mode. Setting any of the tick labels or locations to an empty matrix resets the tick mode to 'auto'. Setting the tick mode to 'auto' clears any explicitly specified tick locations and labels, which are then replaced by default values.

XLoc

scalar

*X-location of the graphic scale* — Controls the horizontal location of the graphic scale within the axes. The location is specified in the axes Cartesian projected coordinates. Use `showaxes` to make the Cartesian grid labels visible. You can also move the graphic scale by dragging the baseline with the mouse.

YLoc

scalar

*Y-location of the graphic scale* — Controls the vertical location of the graphic scale within the axes. The location is specified in the axes Cartesian projected coordinates. Use `showaxes` to make the Cartesian grid labels visible. You can also move the graphic scale by dragging the baseline with the mouse.

### Properties That Control Scaling

Azimuth

scalar

*Azimuth of scale computation* — The scale of a map varies, within the projection, with geographic location and azimuth. This property controls the azimuth along which the scaling between geographic and projected coordinates is computed. The azimuth is given in the current angle units of the map axes. The default azimuth is 0.

Lat

scalar

*Latitude of scale computation* — The scale of a map varies, within the projection, with geographic location and azimuth. This property controls the geographic location at which the scaling between geographic and projected coordinates is computed. The

latitude is given in the current angle units of the map axes. The default location is the center of the displayed map.

Long  
scalar

*Longitude of scale computation* — The scale of a map varies, within the projection, with geographic location and azimuth. This property controls the geographic location at which the scaling between geographic and projected coordinates is computed. The longitude is given in the current angle units of the map axes. The default location is the center of the displayed map.

Radius  
almanac body or scalar

*Planetary radius* — The radius property controls the scaling between angular and surface distances. If radius is a string, then it is evaluated as an almanac body to determine the spherical radius. If numerical, it is the radius of the desired sphere in the same units as the Units property. The default is 'earth'.

Units  
(valid distance unit strings)

*Surface distance units* — Defines the distance units displayed in the graphic scale. Units can be any distance unit string recognized by `unitsratio`. The distance string is also used in the last graphic scale text label.

## Other Properties

Children  
(read-only)

*Name string of graphic scale elements* — Contains the tag string assigned to the graphic elements that compose the graphic scale. All elements of the graphic scale have hidden handles except the baseline. You do not normally need to access the elements directly.



**See Also** distance, surfdist, axesscale, paperscale, distortcalc, mdistort

# scatterm

---

## Purpose

Project point markers with variable color and area

## Syntax

```
scatterm(lat,lon,s,c)
scatterm(lat,lon)
scatterm(lat,lon,s)
scatterm(...,m)
scatterm(...,'filled')
h = scatterm(...)
```

`scatterm(lat,lon,s,c)` displays colored circles at the locations specified by the vectors `lat` and `lon` (which must be the same size). The area of each marker is determined by the values in the vector `s` (in points<sup>2</sup>) and the colors of each marker are based on the values in `c`. `s` can be a scalar, in which case all the markers are drawn the same size, or a vector the same length as `lat` and `lon`.

When `c` is a vector the same length as `lat` and `lon`, the values in `c` are linearly mapped to the colors in the current colormap. When `c` is a length(`lat`)-by-3 matrix, the values in `c` specify the colors of the markers as RGB values. `c` can also be a color string.

`scatterm(lat,lon)` draws the markers in the default size and color.

`scatterm(lat,lon,s)` draws the markers with a single color.

`scatterm(...,m)` uses the marker `m` instead of 'o'.

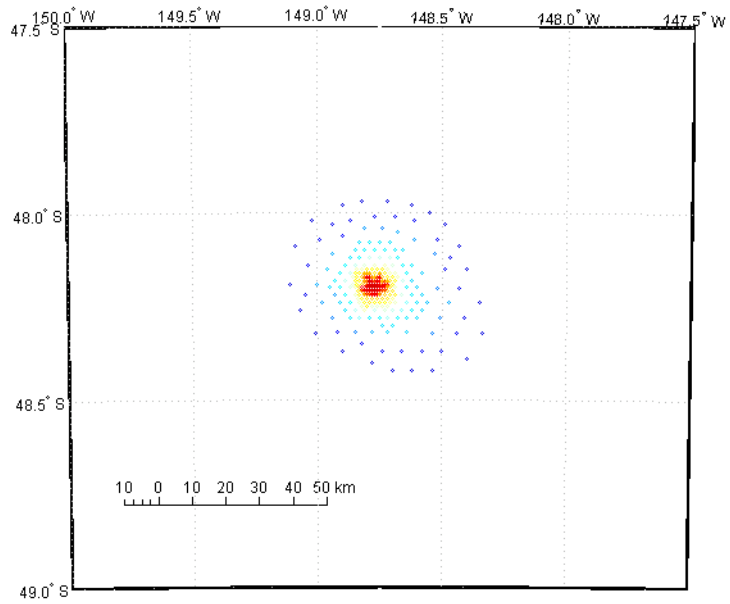
`scatterm(...,'filled')` fills the markers.

`h = scatterm(...)` returns handles of patches created.

## Examples

Plot the seamount data provided with MATLAB as symbols with the color proportional to the height.

```
load seamount
worldmap([-49 -47.5],[-150 -147.5])
scatterm(y,x,5,z)
scaleruler
set(gca,'Visible','off')
```



**See Also**

stem3m

# scircle1

---

## Purpose

Small circles from center, range, and azimuth

## Syntax

```
[latc,lonc] = scircle1(lat,lon,rng)
[latc,lonc] = scircle1(lat,lon,rng,az)
[latc,lonc] = scircle1(lat,lon,rng,az,units)
[latc,lonc] = scircle1(lat,lon,rng,az,ellipsoid,units)
[latc,lonc] = scircle1(lat,lon,rng,az,ellipsoid,units,npts)
[latc,lonc] = scircle1(track,lat,lon,rng...)
pts = scircle1(lat,lon,rng)
```

[latc,lonc] = scircle1(lat,lon,rng) returns the coordinates of points along small circles centered at the points provided in `lat` and `lon` with radii given in `rng`. These radii must in this case be given in the same angle units as the center points ('degrees'). The coordinates for multiple small circles are stored in separate columns of `latc` and `lonc`.

[latc,lonc] = scircle1(lat,lon,rng,az) specifies the arc section of the small circle for which points are returned. The input `az` is a one- or two-column vector. When `az` has a single column, points are returned for the arc segment from 0° azimuth clockwise to the positive entries in `az` (counterclockwise for negative entries). When `az` has two columns, the returned points correspond to arc segments from the first-column entry clockwise to the second-column entry. When `az` is empty or not provided, points for the entire small circle are returned.

[latc,lonc] = scircle1(lat,lon,rng,az,units) specifies the units for the inputs and outputs, where *units* is any valid angle units string. The default value is 'degrees'.

[latc,lonc] = scircle1(lat,lon,rng,az,ellipsoid,units) specifies the elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. The default ellipsoid model is the sphere, which is sufficient for most applications. When a `ellipsoid` is input, the range inputs in `rng` must be in the units of the ellipsoid semimajor axis, rather than in the angle units specified by *units*.

[latc,lonc] = scircle1(lat,lon,rng,az,ellipsoid,units,npts) specifies the number of output points, `npts`, returned per small circle. The default value of `npts` is 100.

`[latc,lonc] = scircle1(track,lat,lon,rng...)` specifies the logic with which ranges are calculated. If the string `track` is `'gc'` (the default), great circle distance is used. If `track` is `'rh'`, rhumb line distance is used.

`pts = scircle1(lat,lon,rng)` returns the points in a two-column output `pts`.

## Background

A small circle is the locus of all points an equal surface distance from a given center. For true small circles, this distance is always calculated in a great circle sense; however, the `scircle1` function allows a locus to be calculated using distances in a rhumb line sense as well. An example of a small circle is *all points exactly 100 miles from the Washington Monument*. Parallels on the globe are all small circles. Great circles are a subset of small circles, specifically those with a radius of  $90^\circ$  or its angular equivalent, so all meridians on the globe are small circles as well.

*Small circle notation* consists of a center point and a radius in units of angular arc length.

## Examples

Create and plot a small circle centered at  $(0^\circ,0^\circ)$  with a radius of  $10^\circ$ :

```
axesm('mercator','MapLatLimit',[30 -30],'MapLonLimit',[-30 30]);
[latc,longc] = scircle1(0,0,10);
plotm(latc,longc,'g')
```

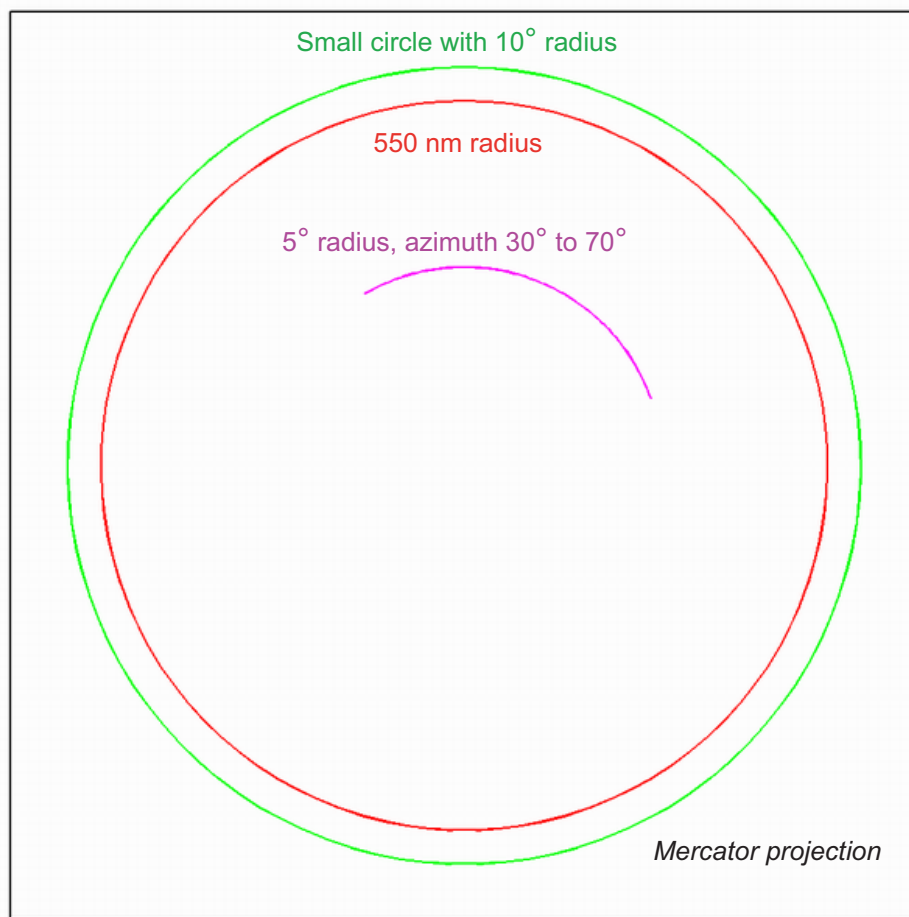
If the desired radius is known in some nonangular distance unit, use the radius returned by the `almanac` function as the ellipsoid to set the range units (use an empty azimuth entry to indicate a full circle):

```
earthradius = almanac('earth','radius','nm');
[latc,longc] = scircle1(0,0,550,[],earthradius);
plotm(latc,longc,'r')
```

For just an arc of the circle, enter an azimuth range:

```
[latc,longc] = scircle1(0,0,5,[-30 70]);
plotm(latc,longc,'m')
```

# scircle1



## See Also

scircle2, track, scircleg, trackg, track1, track2

**Purpose**

Small circles from center and perimeter

**Syntax**

```
[latc,lonc] = scircle2(lat1,lon1,lat2,lon2)
[latc,lonc] = scircle2(lat1,lon1,lat2,lon2,units)
[latc,lonc] = scircle2(lat1,lon1,lat2,lon2,ellipsoid)
[latc,lonc] = scircle2(lat1,lon1,lat2,lon2,ellipsoid,units,
    npts)
[latc,lonc] = scircle2(track,lat1,lon1,lat2,lon2...)
pts = scircle2(lat1,lon1,lat2,lon2)
```

`[latc,lonc] = scircle2(lat1,lon1,lat2,lon2)` returns the coordinates of points along small circles centered at the points provided in `lat1` and `lon1`, which pass through the points provided in `lat2` and `lon2`. The coordinates of multiple small circles are stored in separate columns of `latc` and `lonc`.

`[latc,lonc] = scircle2(lat1,lon1,lat2,lon2,units)` specifies the units for the inputs and outputs, where *units* is any valid angle units string. The default value is 'degrees'.

`[latc,lonc] = scircle2(lat1,lon1,lat2,lon2,ellipsoid)` specifies the elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. The default ellipsoid model is the sphere, which is sufficient for most applications.

`[latc,lonc] = scircle2(lat1,lon1,lat2,lon2,ellipsoid,units,npts)` specifies the number of output points, `npts`, returned per small circle. The default value of `npts` is 100.

`[latc,lonc] = scircle2(track,lat1,lon1,lat2,lon2...)` specifies the logic with which ranges are calculated. If the string `track` is 'gc' (the default), great circle distance is used. If `track` is 'rh', rhumb line distance is used.

`pts = scircle2(lat1,lon1,lat2,lon2)` returns the points in a two-column output `pts`.

**Background**

A small circle is the locus of all points an equal surface distance from a given center. For true small circles, this distance is always calculated in

## scircle2

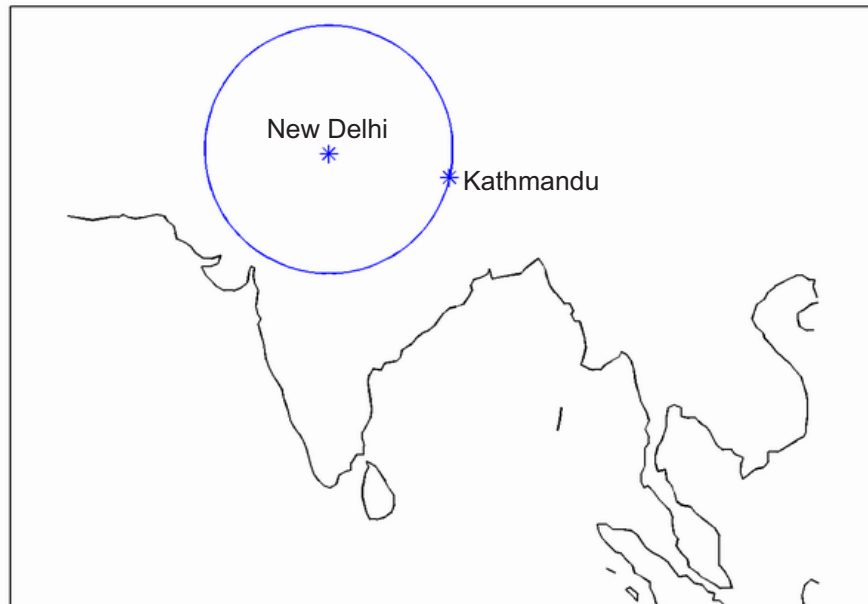
---

a great circle sense; however, the `scircle2` function allows a locus to be calculated using distances in a rhumb line sense as well. An example of a small circle is *all points exactly 100 miles from the Washington Monument*.

### Examples

Plot the locus of all points the same distance from New Delhi as Kathmandu:

```
axesm('mercator','MapLatlimit',[0 40],'MapLonLimit',[60 110]);  
load coast  
plotm(lat,long,'k'); % For reference  
lat1 = 29; lon1 = 77.5; % New Delhi  
lat2 = 27.6; lon2 = 85.5; % Kathmandu  
plotm([lat1 lat2],[lon1 lon2],'b*') % Plot the cities  
[latc,lonc] = scircle2(lat1,lon1,lat2,lon2);  
plotm(latc,lonc,'b')
```





**See Also**      `scircle1, track, track1, track2`

# scircleg

---

## Purpose

Small circle defined via mouse input

## Syntax

```
h = scircleg(ncirc)
h = scircleg(ncirc,npts)
h = scircleg(ncirc,linestyle)
h = scircleg(ncirc,PropertyName,PropertyValue,...)
[lat,lon] = scircleg(ncirc,npts,...)
h = scircleg(track,ncirc,...)
```

`h = scircleg(ncirc)` brings forward the current map axes and waits for the user to make  $(2 * \text{ncirc})$  mouse clicks. The output `h` is a vector of handles for the `ncirc` small circles, which are then displayed.

`h = scircleg(ncirc,npts)` specifies the number of plotting points to be used for each small circle. `npts` is 100 by default.

`h = scircleg(ncirc,linestyle)` specifies the line style for the displayed small circles, where *linestyle* is any line style string recognized by the standard MATLAB function `line`.

`h = scircleg(ncirc,PropertyName,PropertyValue,...)` allows property name/property value pairs to be set, where *PropertyName* and *PropertyValue* are recognized by the `line` function.

`[lat,lon] = scircleg(ncirc,npts,...)` returns the coordinates of the plotted points rather than the handles of the small circles. Successive circles are stored in separate columns of `lat` and `lon`.

`h = scircleg(track,ncirc,...)` specifies the logic with which ranges are calculated. If the string `track` is 'gc' (the default), great circle distance is used. If `track` is 'rh', rhumb line distance is used.

## Description

This function is used to define small circles for display using mouse clicks. For each circle, two clicks are required: one to mark the center of the circle and one to mark any point on the circle itself, thereby defining the radius.

## Background

A small circle is the locus of all points an equal surface distance from a given center. For true small circles, this distance is always calculated

in a great circle sense; however, the `scircleg` function allows a locus to be calculated using distances in a rhumb line sense as well. You can modify the circle after creation by **shift**+clicking it. The circle is then in edit mode, during which you can change the size and position by dragging control points, or by entering values into a control panel. **Shift**+clicking again exits edit mode.

## See Also

`scircle1`, `scircle2`

**Purpose**

Intersection points for pairs of small circles

**Syntax**

```
[newlat,newlon] = scxsc(lat1,lon1,range1,lat2,lon2,range2)
[newlat,newlon]=scxsc(lat1,lon1,range1,lat2,lon2,range2,
    units)
```

[newlat,newlon] = scxsc(lat1,lon1,range1,lat2,lon2,range2) returns in newlat and newlon the locations of the points of intersection of two small circles in *small circle notation*. For example, the first small circle in a pair would be centered on the point (lat1,lon1) with a radius of range1 (in angle units). The inputs must be column vectors. If the circles do not intersect, or are identical, two NaNs are returned and a warning is displayed. If the two circles are tangent, the single intersection point is returned twice.

[newlat,newlon]=scxsc(lat1,lon1,range1,lat2,lon2,range2,units) specifies the angle units used for all inputs, where *units* is any valid angle units string. The default units are 'degrees'.

**Description**

For any pair of small circles, there are four possible intersection conditions: the circles are identical, they do not intersect, they are tangent to each other and hence they intersect once, or they intersect twice.

*Small circle notation* consists of a center point and a radius in units of angular arc length.

**Examples**

Given a small circle centered at (10°S,170°W) with a radius of 20° (~1200 nautical miles), where does it intersect with a small circle centered at (3°N, 179°E), with a radius of 15° (~900 nautical miles)?

```
[newlat,newlong] = scxsc(-10,-170,20,3,179,15)

newlat =
    -8.8368    9.8526
newlong =
    169.7578 -167.5637
```

Note that in this example, the two small circles cross the date line.

**Remarks**

Great circles are a subset of small circles—a great circle is just a small circle with a radius of  $90^\circ$ . This provides two methods of notation for defining great circles. *Great circle notation* consists of a point on the circle and an azimuth at that point. *Small circle notation* for a great circle consists of a center point and a radius of  $90^\circ$  (or its equivalent in radians).

**See Also**

gc2sc, gcxgc, gcxsc, rhxrh, crossfix, polyxpoly

# sdtsemread

---

**Purpose** Read data from SDTS raster/DEM data set

**Syntax** `[Z, R] = sdtsemread(filename)`  
`[Z, R] = sdtsemread(filename)` reads data from an SDTS DEM data set. `Z` is a matrix containing the elevation values. `R` is a referencing matrix (see `makerefmat`). NaNs are assigned to elements of `Z` corresponding to null data values or fill data values in the cell module.  
`filename` can be the name of the SDTS catalog directory file (`*CATD.DDF`) or the name of any of the other files in the data set. `filename` can include the directory name; otherwise `filename` is searched for in the current directory and the MATLAB path. If any of the files specified in the catalog directory are missing, `sdtsemread` fails.

**Remarks** Elevation values can be imported with `sdtsemread` from DEMs that use the SPRE Raster Profile (in use since January, 2001) as well as from older SDTS DEM data sets. Under this profile, elevations can be encoded either as 32-bit floating-point numbers (when their units are “decimal meters”), or as 16-bit integers (when units are “feet” or “meters”). The output class from `sdtsemread` for both types of elevation encoding is `double`.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

**Example** `[Z, R] = sdtsemread('9129CATD.ddf');`  
`mapshow(Z,R, 'DisplayType', 'contour')`

**See Also** `arcgridread`, `makerefmat`, `mapshow`, `sdtsemread`

**Purpose** Information about SDTS data set

**Syntax** `info = sdtsinfo(filename)`

`info = sdtsinfo(filename)` returns a structure whose fields contain information about the contents of a SDTS data set.

`filename` is a string that specifies the name of the SDTS catalog directory file, such as `7783CATD.DDF`. The filename can also include the directory name. If `filename` does not include the directory, then it must be in the current directory or in a directory on the MATLAB path. If `sdtsinfo` cannot find the SDTS catalog file, it returns an error.

If any of the other files in the data set as specified by the catalog file is missing, a warning message is returned. Subsequent calls to read data from the file might also fail.

## Field Descriptions

The `info` structure contains the following fields:

Filename	String containing the name of the catalog directory file of the SDTS transfer set
Title	String containing the name of the data set
ProfileID	String containing the Profile Identifier, e.g., 'SRPE: SDTS RASTER PROFILE and EXTENSIONS'
ProfileVersion	String containing the Profile Version Identifier, e.g., 'VER 1.1 1998 01'
MapDate	String specifying the date associated with the cartographic information contained in the data set
DataCreationDate	String specifying the creation date of the data set
HorizontalDatum	String representing the horizontal datum to which the data is referenced

# sdtsinfo

---

MapRefSystem	String describing the projection and reference system used: 'GEO', 'SPCS', 'UTM', 'UPS', or ' '
ZoneNumber	Scalar value representing the zone number
XResolution	Scalar value representing the X component of the horizontal coordinate resolution
YResolution	Scalar value representing the Y component of the horizontal coordinate resolution
NumberOfRows	Scalar value representing the number of rows of the DEM
NumberOfCols	Scalar value representing the number of columns of the DEM
HorizontalUnits	String specifying the units used for the horizontal coordinate values
VerticalUnits	String specifying the units used for the vertical coordinate values
MinElevation	Scalar value of the minimum elevation value for the data set
MaxElevation	Scalar value of the maximum elevation value for the data set

## Example

```
info = sdtsinfo('9129CATD.DDF');
```

## See Also

`sdtsdemread`, `makereformat`



## Purpose

Convert time from seconds to hrs:min or hrs:min:sec encoding

---

**Note** The `sec2hm` and `sec2hms` functions are obsolete and error when used. They will be completely removed from the next version of Mapping Toolbox.

---

## Syntax

```
timeout = sec2hms(timein)
timeout = sec2hm(timein)
```

`timeout = sec2hms(timein)` converts times input in seconds to the equivalent measure in the hours-minutes-seconds (*hms*) format.

`timeout = sec2hm(timein)` converts times input in seconds to the equivalent measure in the hours-minutes (*hm*) format. This is the *hms* format, properly rounded to just hours and minutes.

## Example

```
sec2hms(3661)

ans =
    101.01

sec2hm(3661)

ans =
    101.00
```

# sec2hr

---

**Purpose** Convert time from seconds to hours

---

**Note** The sec2hr function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox.

---

**Syntax** `timeout = sec2hr(timein)`  
`timeout = sec2hr(timein)` converts times input in seconds to the equivalent measure in hours.

**Example**

```
sec2hr(1000)  
  
ans =  
    0.2778
```

---

<b>Purpose</b>	Sector of small circle defined via mouse input
<b>Syntax</b>	<p>sectorg</p> <p>sectorg prompts the user to indicate by two successive mouse clicks two points that define the center and radius of a small circle arc. By default, the angular width of the sector is 60°. The sector is constructed using the vector defined by the mouse clicks as the reference azimuth (defined to run through the center of the sector).</p> <p>Once a sector has been drawn, <b>Shift</b>+clicking on it displays four control points (center point, arc resize, radial resize, and rotation controls), and the associated <b>Sector</b> control window. You can graphically interact with sectors as follows:</p> <ul style="list-style-type: none"><li>• To translate the circle, click and drag the center (o) control.</li><li>• To change the arc size, click and drag the resize control (square).</li><li>• To change the radial size of the sector, click and drag the radial control (down triangle).</li><li>• To rotate the arc, click and drag the rotation control (x).</li></ul> <p>You can also modify a selected sector by entering the appropriate values in the <b>Sector</b> control window and then pressing <b>Enter</b> or clicking the <b>Close</b> button. Display of the control panel is toggled by <b>Shift</b>+clicking the sector. If you select multiple sectors, a separate <b>Sector</b> control window will appear for each one.</p>
<b>Remarks</b>	<b>Sector</b> control windows are superimposed at the same location. A valid map axes must exist prior to running this function.
<b>See also</b>	scircleg, trackg

# setltn

---

## Purpose

Convert data grid rows and columns to latitude-longitude

## Syntax

```
[lat,lon] = setltn(Z,refvec,row,col)
```

```
mat = setltn(Z,refvec,row,col)
```

```
[lat,lon,badindx] = setltn(Z,refvec,row,col)
```

`[lat,lon] = setltn(Z,refvec,row,col)` returns the latitude and longitudes associated with the input row and column coordinates of the input grid `Z` geolocated by three-element referencing vector `refvec`.

`mat = setltn(Z,refvec,row,col)` returns the coordinates in a single two-column matrix of the form `[latitude longitude]`.

`[lat,lon,badindx] = setltn(Z,refvec,row,col)` returns the indices of the elements of the row and col vectors that lie outside the input grid. The outputs `lat` and `lon` always ignore these points; the third output accounts for them.

## Examples

Find the coordinates of row 45, column 65 of `topo`:

```
load topo
```

```
[lat,lon,badindx] = setltn(topo,topolegend,45,65)
```

```
lat =
```

```
    -45.5000
```

```
lon =
```

```
    64.5000
```

```
badindx =
```

```
    []      % Empty because the point is valid
```

## See Also

`ltn2val`, `pix2latlon`, `setpostn`

**Purpose**

Set properties of map axes and graphics objects

**Syntax**

```
setm(axishndl,PropertyName,PropertyValue,...)
setm(texthndl,'MapPosition',position)
setm(surfhndl,'Graticule',lat,lon,alt)
setm(surfhndl,'MeshGrat',npts,alt)
```

`setm(axishndl,PropertyName,PropertyValue,...)` sets the properties of the map axes specified by its handle to the given values. The map properties must be recognized by `axesm`.

`setm(texthndl,'MapPosition',position)` alters the position of the projected text object specified by its handle to the [latitude longitude] or the [latitude longitude altitude] specified by the position vector.

`setm(surfhndl,'Graticule',lat,lon,alt)` alters the graticule of the projected surface object specified by its handle. The graticule is specified by the latitude and longitude matrices, specifying locations of the graticule vertices. The altitude can be specified by a scalar, or by a matrix providing a value for each vertex.

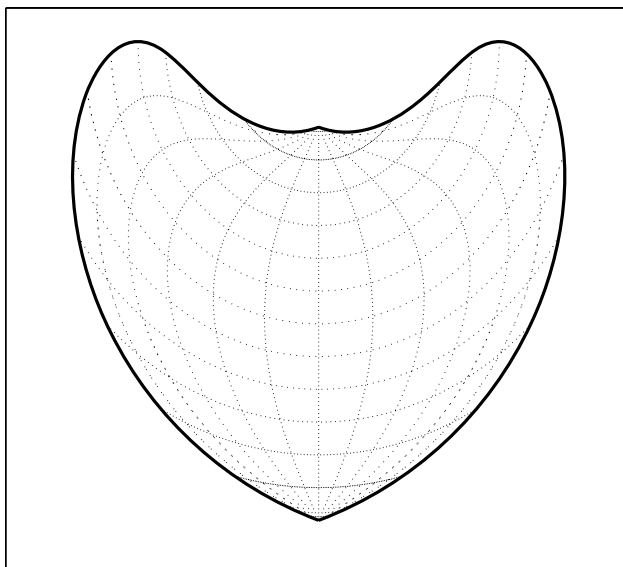
`setm(surfhndl,'MeshGrat',npts,alt)` alters the mesh graticule of projected surface objects displayed using the `meshm` function. In this case, the two-element vector `npts` specifies the graticule size in the manner described under `meshm`. The altitude can be a scalar or a matrix with a size corresponding to `npts`.

**Examples**

Display a map axes and alter it:

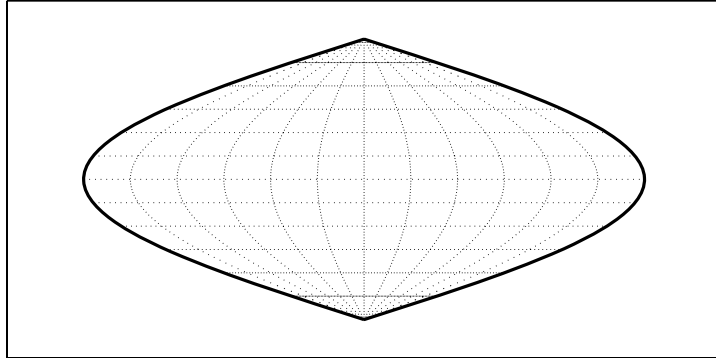
```
axesm('bonne','Frame','on','Grid','on')
```

The standard Bonne projection has a standard parallel at 30°N.



Setting this standard parallel to  $0^\circ$  results in a Sinusoidal projection:

```
setm(gca, 'MapParallels', 0)
```



**See Also** `axesm`, `getm`

# setpostn

---

## Purpose

Convert latitude-longitude to data grid rows and columns

## Syntax

```
[row,col] = setpostn(Z,refvec,lat,long)
```

```
indx = setpostn(Z,refvec,lat,long)
```

```
[row,col,badindx] = setpostn(Z,refvec,lat,long)
```

`[row,col] = setpostn(Z,refvec,lat,long)` returns the row and column indices of the input regular data grid `Z` with a three-element referencing vector `refvec` for the points specified by the vectors `lat` and `long`. All angles are in degrees.

`indx = setpostn(Z,refvec,lat,long)` returns the single-value indices of `map(:)`.

`[row,col,badindx] = setpostn(Z,refvec,lat,long)` also returns the indices of `lat` and `long` corresponding to points outside `Z`. These points are always ignored in `row` and `col`.

## Examples

What are the matrix coordinates in `topo` of Denver, Colorado, at (39.7°N,105°W)?

```
load topo
[row,col] = setpostn(topo,topolegend,39.7,105)
```

```
row =
    130
col =
    105
```

## See Also

`latlon2pix`, `latln2val`, `setlatln`



**Purpose**

Construct cdata and colormap for shaded relief

**Syntax**

```
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap)
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev])
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmap1)
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmap1,
    clim)
```

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap)` constructs the colormap and color indices to allow a surface to be displayed in colored shaded relief. The colors are proportional to the magnitude of Z, but modified by shades of gray based on the surface normals to simulate surface lighting. This representation allows both large and small-scale differences to be seen. X, Y, and Z define the surface. cmap is the colormap used to create the new shaded colormap cimap. cindx is a matrix of color indices to cimap, based on the elevation and surface normal of the Z matrix element. clim contains the color axis limits.

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev])` places the light at the specified azimuth and elevation. By default, the direction of the light is East (90° azimuth) at an elevation of 45°.

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmap1)` chooses the number of grays to give a cimap of length cmap1. By default, the number of grayscales is chosen to keep the shaded colormap below 256. If the vector of azimuth and elevation is empty, the default locations are used.

`[cindx,cimap,clim] = shaderel(X,Y,Z,cmap,[azim elev],cmap1,clim)` uses the color limits to index Z into cmap.

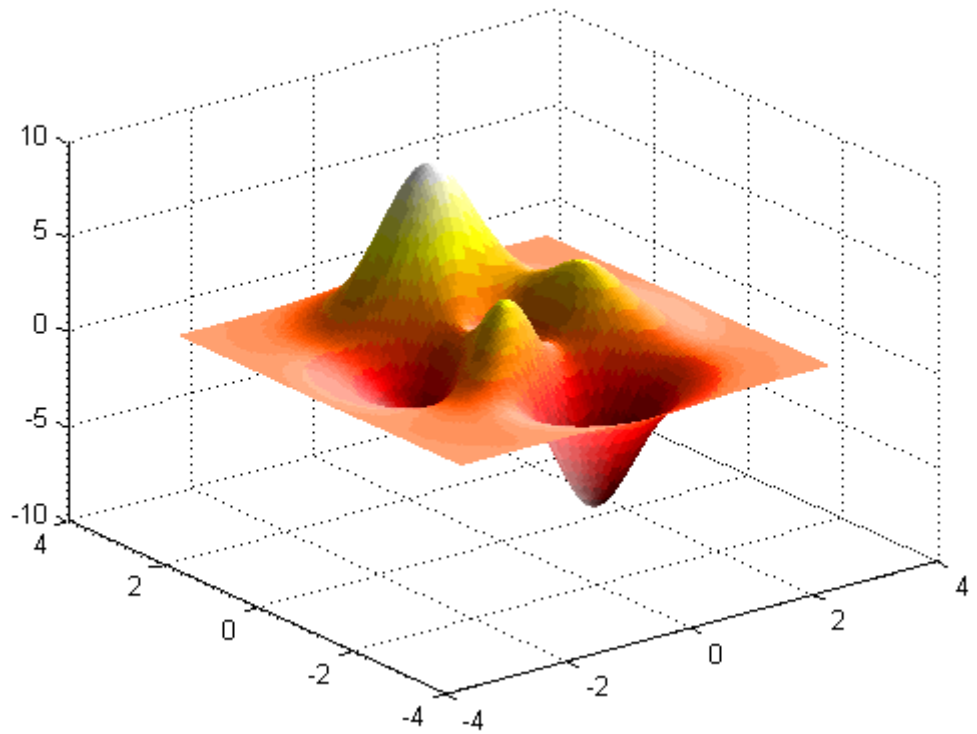
**Remarks**

This function effectively multiplies two colormaps, one with color based on elevation, the other with a grayscale based on the slope of the surface, to create a new color map. This produces an effect similar to using a light on a surface, but with all of the visible colors actually in the colormap. Lighting calculations are performed on the unprojected data.

## Examples

Display the peaks surface with a shaded colormap:

```
[X,Y,Z] = peaks(100);  
cmap = hot(16);  
[cindx,cimap,clim] = shaderel(X,Y,Z,cmap);  
surf(X,Y,Z,cindx); colormap(cimap); caxis(clim)  
shading flat
```



## See Also

[caxis](#), [colormap](#), [light](#), [meshlstrm](#), [surf](#), [surflstrm](#)

**Purpose** Information about shapefile

**Syntax** `info = shapeinfo(filename)`

`info = shapeinfo(filename)` returns a structure, `info`, whose fields contain information about the contents of a shapefile.

The shapefile format was defined by the Environmental Systems Research Institute (ESRI) to store nontopological vector geometry and attribute information for spatial features. A shapefile consists of a main file, an index file, and an xBASE table. All three files have the same base name and are distinguished by the extensions `.SHP`, `.SHX`, and `.DBF`, respectively (e.g., given the base name 'roads' the shapefile filenames would be 'roads.SHP', 'roads.SHX', and 'roads.DBF').

`filename` can be the base name or the full name of any one of the component files. `shapeinfo` reads all three files as long as they exist in the same directory and have valid file extensions. If the main file (with extension `.SHP`) is missing, `shapeinfo` returns an error. If either of the other files is missing, `shapeinfo` returns a warning.

## Field Descriptions

The `info` structure contains the following fields:

Filename	Char array containing the names of the files that were read
ShapeType	String containing the shape type
BoundingBox	Numerical array of size 2-by-N that specifies the minimum (row 1) and maximum (row 2) values for each dimension of the spatial data in the shapefile
Attributes	Structure array of size 1-by-numAttributes that describes the attributes of the data
NumFeatures	The number of spatial features in the shapefile

The `Attributes` structure contains these fields:

# shapeinfo

---

- **Name** — String containing the attribute name as given in the xBASE table
- **Type** — String specifying the MATLAB class of the attribute data returned by `shaperead`. The following attribute (xBASE) types are supported: Numeric, Floating, Character, and Date.

---

**Note** `shapeinfo` cannot tell you what coordinate system data in a shapefile use. Coordinates can be either planar (x, y) or geographic (lat, lon) and have a variety of units. Because shapefiles do not document coordinate systems, `shapeinfo` cannot tell you what map projection coordinate data may be in or what the projection's parameters are. You need to obtain this information from your shapefile vendor. This information can be critical to the proper display of shapefile vector data, because by default `shaperead` will generate geostructs that describe map coordinates (x,y values) unless you specify that the shapefile contains geographic coordinates using the optional `'UseGeoCoords', true` parameter/value pair.

---

## See Also

`shaperead`

**Purpose**

Read vector features and attributes from shapefile

**Syntax**

```
s = shaperead(filename)
[s, a] = shaperead(filename)
[s, a] = shaperead(filename, param1, val1, param2, val2, ...)
```

`s = shaperead(filename)` returns an N-by-1 Version 2 geographic data structure (geostruct2) array, `S`, containing an element for each nonnull spatial feature in the shapefile. `S` combines feature coordinates/geometry and attribute values. The extension of `filename` can be `.shp`, `.dbf` or `.shx`, or be omitted (see Remarks, below).

`[s, a] = shaperead(filename)` returns an N-by-1 geostruct2 array, `s`, and a parallel N-by-1 attribute structure array, `a`. Each array contains an element for each nonnull spatial feature in the shapefile. The attributes are read from the file `filename.dbf` (see Remarks, below).

`[s, a] = shaperead(filename, param1, val1, param2, val2, ...)` returns a subset of the shapefile contents in `s` or `[s, a]`, as determined by the parameters 'RecordNumbers', 'BoundingBox', 'Selector', or 'Attributes'. In addition, the parameter 'UseGeoCoords' can be used in cases where you know that the *x*- and *y*-coordinates in the shapefile actually represent longitude and latitude.

**Remarks**

The shapefile format was defined by the Environmental Systems Research Institute (ESRI) to store nontopological vector geometry and attribute information for spatial features. A shapefile consists of a main file, an index file, and an xBASE table. All three files have the same base name and are distinguished by the extensions `.shp`, `.shx`, and `.dbf`, respectively (e.g., given the base name 'concord\_roads' the shapefile filenames would be 'concord\_roads.shp', 'concord\_roads.shx', and 'concord\_roads.dbf').

`filename` can be the base name or the full name of any one of the component files. `shaperead` reads all three files as long as they exist in the same directory and have valid file extensions. If the main file (with extension `.SHP`) is missing, `shaperead` returns an error. If either of the other files is missing, `shaperead` returns a warning.

## Supported Shape Types

shaperead supports the ordinary 2-D shape types: 'Point', 'Multipoint', 'PolyLine', and 'Polygon'. ('Null Shape' features can also be present in a Point, Multipoint, PolyLine, or Polygon shapefile, but are ignored.) shaperead does *not* support any 3-D or “measured” shape types: 'PointZ', 'PointM', 'MultipointZ', 'MultipointM', 'PolyLineZ', 'PolyLineM', 'PolygonZ', 'PolylineM', or 'Multipatch'.

## Output Structure

The fields in the output structure arrays `s` and `a` depend on (1) the type of shape contained in the file and (2) the names and types of the attributes included in the file:

Field Name	Field Contents	Comment
<i>Geometry</i>	Shape type string	'Point', 'Multipoint', 'PolyLine', or 'Polygon'
BoundingBox	[minX minY; maxX maxY]	Omitted for shape type 'Point'
X or Lon	Coordinate vector	NaN separators used in multipart PolyLine and Polygon shapes
Y or Lat	Coordinate vector	NaN separators used in multipart PolyLine and Polygon shapes
attr1	Value of first attribute	Included in output <code>s</code> if output <code>a</code> is omitted
attr2	Value of second attribute	Included in output <code>s</code> if output <code>a</code> is omitted
...	...	...

The names of the attribute fields (listed above as `Attr1`, `Attr2`, ...) are determined at run-time from the `xBASE` table (with extension

.DBF) and/or optional, user-specified parameters. There can be many attribute fields, or none at all.

## Field Descriptions

- **Geometry** — String with one of the following values: 'Point', 'MultiPoint', 'Line', or 'Polygon'. (Note that these match the standard shapefile types, except that for shape type 'Polyline' the value of the *Geometry* field is simply 'Line'.)
- **BoundingBox** — Contains a 2-by-2 numerical array specifying the minimum and maximum feature coordinate values in each dimension ( $\min([x, y])$ ;  $\max([x, y])$ , where  $x$  and  $y$  are N-by-1 and contain the combined coordinates of all parts of the feature). In the absence of documentation or metadata for the geodata in a shapefile, you can use this information to decide what kind of coordinate system (map or geographic) the shapes have.
- **X and Y / Lon and Lat (Coordinate vector)** — 1-by-N arrays of class double. For 'Point' shapes, they are 1-by-1. In the case of multipart 'Polyline' and 'Polygon' shapes, NaNs are added to separate the lines or polygon rings. In addition, one terminating NaN is added to support horizontal concatenation of the coordinate data from multiple shapes.
- **Attribute fields** — Attribute names, types, and values are defined within a given shapefile. The following four types are supported: Numeric, Floating, Character, and Date. shaperead skips over other attribute types. The field names in the output shape structure are taken directly from the shapefile if they contain no spaces or other illegal characters and there is no duplication of field names (e.g., an attribute named 'BoundingBox', 'PointData', etc., or two attributes with the same names).

Otherwise, the following “name mangling” is applied: Illegal characters are replaced by '\_'. If the first character in the attribute name is illegal, a leading 'Z' is added. Numerals are appended if needed to avoid duplicate names. The attribute values for a feature

are taken from the shapefile and stored as doubles or character arrays:

Attribute Type in Shapefile	MATLAB Storage
Numeric	double (scalar)
Float	double (scalar)
Character	char array
Date	char array

## Parameter-Value Options

By default, shaperead returns an entry for every nonnull feature and creates a field for every attribute. Use the first three parameters below (RecordNumbers, BoundingBox, and Selector) to be selective about which features to read. Use the fourth parameter (Attributes) to control the attributes to keep. Use the fifth (UseGeoCoords) to control the output field names.

Name	Description of Value	Purpose
RecordNumbers	Integer-valued vector, class double	Screen out features whose record numbers are not listed.
BoundingBox	2-by-(2, 3, or 4) array, class double	Screen out features whose bounding boxes fail to intersect the selected box. No trimming of features that partially intersect the box is performed.



Name	Description of Value	Purpose
Selector	Cell array containing a function handle and one or more attribute names. Function must return a logical scalar.	Screen out features for which the function, when applied to the corresponding attribute values, returns false.
Attributes	Cell array of attribute names	Omit attributes that are not listed. Use {} to omit all attributes. Also sets the order of attributes in the structure array.
UseGeoCoords	Scalar logical	If true, replace X and Y field names with Lon and Lat, respectively. Defaults to false.

---

**Note** If you do not know whether a shapefile uses latitude and longitude or map (planar) coordinates (i.e. contains unprojected or projected geodata), you can read it (or use `shapeinfo`) to obtain the `BoundingBox`; the ranges of coordinates may be sufficient information for you to decide what kind of coordinates you have. In some cases you may need to reread the shapefile with or without the `'UseGeoCoords'`, true argument (it defaults to false), depending on whether you believe the coordinates are geographic latitude and longitude or map x and y, respectively. If you do not specify `UseGeoCoords`, the `geostruct` returned by `shaperead` will contain X and Y fields rather than Lon and Lat fields. The `geoshow` function can sense such situations; it issues a warning and calls `mapshow` to plot `geostructs` having X and Y fields. The `mapshow` function refuses to draw `geostructs` that have Lon and Lat fields.

---

## Examples

### Example 1

Read the entire concord\_roads.shp shapefile, including the attributes, in concord\_roads.dbf:

```
S = shaperead('concord_roads.shp');
% Restrict output based on a bounding box and read only two
% of the feature attributes.
bbox = [2.08 9.11; 2.09 9.12] * 1e5;
S = shaperead('concord_roads', 'BoundingBox', bbox, ...
    'Attributes', {'STREETNAME', 'CLASS'});

% Select the class 4 and higher road segments that are at least 200
% meters in length. Note the use of an anonymous function in the
% selector.
S = shaperead('concord_roads.shp', 'Selector', ...
    {@(v1,v2) (v1 >= 4) && (v2 >= 200), 'CLASS', 'LENGTH'});
N = hist([S.CLASS], 1:7)
hist([S.LENGTH])
```

### Example 2

Read worldwide city names and locations in latitude and longitude. Note presence of 'Lat' and 'Lon' fields:

```
S = shaperead('worldcities.shp', 'UseGeoCoords', true)

S =
318x1 struct array with fields:
    Geometry
    Lon
    Lat
    Name
```

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

**See Also**      `shapeinfo`, `updategeostruct`

# shapewrite

---

## Purpose

Write geographic data structure to shapefile

## Syntax

```
shapewrite(S, filename)
shapewrite(S, filename, 'DbfSpec', dbfspec)
```

`shapewrite(S, filename)` writes a Version 2 geographic data structure (a `geostruct2`) to disk in shapefile format. `S` must be a valid `geostruct2` with specific restrictions on its attribute fields:

- Each attribute field value must be either a real, finite, scalar double or a character string.
- The type of a given attribute must be consistent across all features.

`filename` must be a character string specifying the output file name and location. If an extension is included, it must be `'.shp'` or `'.SHP'`.

`shapewrite` creates three output files,

- [basename `'.shp'`]
- [basename `'.shx'`]
- [basename `'.dbf'`]

where `basename` is `filename` without its extension.

If a given attribute is integer-valued for all features, then it is written to the [basename `'.dbf'`] file as an integer. If an attribute is noninteger for any feature, then it is written as a fixed point decimal value with six digits to the right of the decimal place.

`shapewrite(S, filename, 'DbfSpec', dbfspec)` writes a shapefile in which the content and layout of the DBF file is controlled by a DBF specification, indicated here by the parameter value `dbfspec`. A DBF specification is a scalar MATLAB structure with one field for each feature attribute to be included in the output shapefile. To include an attribute in the output, make sure to provide a field in `dbfspec` with a field name identical to the attribute name (the corresponding field

name in S), and assign to that field a scalar structure with the following four fields:

- `FieldName` — The field name to be used in the file
- `FieldType` — The field type to be used in the file ('N' or 'C')
- `FieldLength` — The field length in the file, in bytes
- `FieldDecimalCount` — For numeric fields, the number of digits to the right of the decimal place

When a DBF spec is provided, a given attribute will be included in the output file only if it matches the name of a field in the spec.

The easiest way to construct a DBF spec is to call `makedbfspec`, then modify the output to remove attributes or change the `FieldName`, `FieldLength`, or `FieldDecimalCount` for one or more attributes. See the help for `makedbfspec` for more details and an example.

## Remarks

Geostruct attribute names that are longer than 11 characters are truncated to 11 characters in copying as DBF field names in order to adhere to dBASE (.dbf) file specifications. Consider shortening long field names in your geostruct before calling `shapewrite` to make field names in the DBF file more readable and to avoid introducing duplicate names as a result of truncation.

## Example

Derive a shapefile from `concord_roads.shp` in which roads of CLASS 5 and greater are omitted. Note the use of the 'Selector' option in `shaperead`, together with an anonymous function, to read only the main roads from the original shapefile.

```
shapeinfo('concord_roads') % 609 features

ans =
    Filename: [3x67 char]
    ShapeType: 'PolyLine'
    BoundingBox: [2x2 double]
    NumFeatures: 609
```

# shapewrite

---

```
Attributes: [5x1 struct]

S = shaperead('concord_roads', 'Selector', ...
    {@(roadclass) roadclass < 4, 'CLASS'});
shapewrite(S, 'main_concord_roads.shp')
shapeinfo('main_concord_roads') % 107 features

ans =
    Filename: [3x24 char]
    ShapeType: 'PolyLine'
    BoundingBox: [2x2 double]
    NumFeatures: 107
    Attributes: [5x1 struct]
```

## See Also

makedbfspec, shapeinfo, shaperead, updategeestruct

**Purpose** Toggle display of map coordinate axes

**Syntax**

```
showaxes  
showaxes('on')  
showaxes('off')  
showaxes('hide')  
showaxes('show')  
showaxes('reset')  
showaxes(color)  
showaxes(color)
```

`showaxes` toggles the visibility of the axes between the 'on' and 'off' conditions.

`showaxes('on')` sets the color of the axes (the `XColor` and `YColor` properties) to black.

`showaxes('off')` sets the color of the axes (the `XColor` and `YColor` properties) to the background color, effectively making them invisible. This is the default condition for map axes.

`showaxes('hide')` sets the `Visible` property of the axes to 'on'.

`showaxes('show')` sets the `Visible` property of the axes to 'off'.

`showaxes('reset')` sets the axes properties to the default map axes settings.

`showaxes(color)` sets the color of the axes (the `XColor` and `YColor` properties) to the color specified by any valid color string.

`showaxes(color)` sets the color of the axes (the `XColor` and `YColor` properties) to the color specified by the input RGB triple.

**See Also** `axesm`

# showm

---

## **Purpose**

Specify graphic objects to display on map axes

## **Syntax**

```
showm  
showm(handle)  
showm(object)
```

showm brings up a dialog box for selecting the objects to show (set their `Visible` property to 'on').

showm(handle) shows the objects specified by a vector of handles.

showm(*object*) shows those objects specified by the *object* string, which can be any string recognized by the `handlem` function.

## **See Also**

clma, clmo, handlem, hidem, namem, tagm



**Purpose**

Row and column dimensions needed for regular data grid

**Syntax**

```
[r,c] = sizen(latlim,lonlim,scale)
rc = sizen(latlim,lonlim,scale)
[r,c,refvec] = sizen(latlim,lonlim,scale)
```

`[r,c] = sizen(latlim,lonlim,scale)` returns the required size for a regular data grid lying between the latitude and longitude limits specified by the two-element input vectors `latlim` and `lonlim`, which are of the form `[south-limit north-limit]` and `[west-limit east-limit]`, respectively. The `scale` is the desired cells-per-degree measure of the desired data grid.

`rc = sizen(latlim,lonlim,scale)` returns the size of the matrix in one two-element vector.

`[r,c,refvec] = sizen(latlim,lonlim,scale)` also returns the three-element referencing vector geolocating the desired regular data grid.

**Examples**

How large a matrix would be required for a map of the world at a scale of 25 matrix cells per degree? (That's  $25 \times 25 = 625$  cells per "square" degree.)

```
[r,c] = sizen([90,-90],[-180,180],25)
```

```
r =
    4500
c =
    9000
```

Bear in mind for memory purposes —  $9000 \times 4500 = 4.05 \times 10^7$  entries!

**See Also**

`findm`, `limitm`, `nanm`, `onem`, `spzerom`, `zerom`

# smoothlong

---

**Purpose** Remove discontinuities in longitude data

---

**Note** The `smoothlong` function is obsolete and has been replaced by `unwrapMultipart`, which requires input to be in radians. When working in degrees, use `rad2deg(unwrapMultipart(deg2rad(lon)))`.

---

## Syntax

`ang = smoothlong(angin)`

`ang = smoothlong(angin, angleunits)`

`ang = smoothlong(angin)` removes discontinuities in longitude data. The resulting angles can cover more than one revolution.

`ang = smoothlong(angin, angleunits)` uses the units defined by the input string *angleunits*. If omitted, default units of 'degrees' are assumed. Valid *angleunits* are:

- 'degrees' — decimal degrees
- 'radians'

## See Also

`unwrapMultipart`

**Purpose**

Read columns of data from ASCII text file

**Syntax**

```
mat = spread
```

```
mat = spread(filename)
```

```
mat = spread(cols)
```

`mat = spread` reads an ASCII file of space-delimited data in two columns and returns the data in a matrix, `mat`. The file is selected by dialog box.

`mat = spread(filename)` specifies the file from which to read by its name, given as the string *filename*.

`mat = spread(cols)` specifies the number of columns of space-delimited data in the file with the integer *cols*. The default value of *cols* is 2.

**Remarks**

The `spread` function is similar to the standard MATLAB function `dlmread`. `spread`, however, is much faster at reading large data sets of the type common for geographic purposes.

**See Also**

`nanclip`

**Purpose** Construct sparse regular data grid of 0s

**Syntax** `[Z,refvec] = spzerom(latlim,lonlim,scale)`  
`[Z,refvec] = spzerom(latlim,lonlim,scale)` returns a sparse regular data grid consisting entirely of 0s and a three-element referencing vector for the returned Z. The two-element vectors `latlim` and `lonlim` define the latitude and longitude limits of the geographic region. They should be of the form `[south north]` and `[west east]`, respectively. The scalar `scale` specifies the number of rows and columns per degree of latitude and longitude.

**Examples** `[Z,refvec] = spzerom([46,51],[-79,-75],1)`

```
Z =  
    All zero sparse: 5-by-4  
refvec =  
     1     51    -79
```

**See Also** `limitm`, `nanm`, `onem`, `sizem`, `zerom`

**Purpose**

Standard distance for geographic points

**Syntax**

```
dist = stdist(lat,lon)
dist = stdist(lat,lon,units)
dist = stdist(lat,lon,ellipsoid)
dist = stdist(lat,lon,ellipsoid,units,method)
```

`dist = stdist(lat,lon)` returns a row vector of the latitude and longitude geographic standard distance for the data points specified by the columns of `lat` and `lon`.

`dist = stdist(lat,lon,units)` indicates the angular units of the data. When the standard angle string `units` is omitted, 'degrees' is assumed. Output measurements are in terms of these `units` (as arc length distance).

`dist = stdist(lat,lon,ellipsoid)` specifies the elliptical definition of the Earth to be used with the two-element `ellipsoid` vector. The default ellipsoid model is a spherical Earth, which is sufficient for most applications. Output measurements are in terms of the distance units of the `ellipsoid` vector.

`dist = stdist(lat,lon,ellipsoid,units,method)` specifies the method of calculating the standard distance of the data. The default, 'linear', is simply the average great circle distance of the data points from the centroid. Using 'quadratic' results in the square root of the average of the squared distances, and 'cubic' results in the cube root of the average of the cubed distances.

**Background**

The function `stdm` provides independent standard deviations in latitude and longitude of data points. `stdist` provides a means of examining data scatter that does not separate these components. The result is a *standard distance*, which can be interpreted as a measure of the scatter in the great circle distance of the data points from the centroid as returned by `meanm`.

# stdist

---

## Description

The output distance can be thought of as the radius of a circle centered on the geographic mean position, which gives a measure of the spread of the data.

## Examples

Create latitude and longitude lists using the `worldcities` data set and obtain standard distance deviation for group (compare with the example for `stdm`):

```
cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
Paris = strmatch('Paris',{cities(:).Name});
London = strmatch('London',{cities(:).Name});
Rome = strmatch('Rome',{cities(:).Name});
Madrid = strmatch('Madrid',{cities(:).Name});
Berlin = strmatch('Berlin',{cities(:).Name});
Athens = strmatch('Athens',{cities(:).Name});
lat = [cities(Paris).Lat cities(London).Lat...
       cities(Rome).Lat cities(Madrid).Lat...
       cities(Berlin).Lat cities(Athens).Lat]
lon = [cities(Paris).Lon cities(London).Lon...
       cities(Rome).Lon cities(Madrid).Lon...
       cities(Berlin).Lon cities(Athens).Lon]

dist =stdist(lat,lon)

lat =
    48.8708    51.5188    41.9260    40.4312    52.4257    38.0164
lon =
     2.4131    -0.1300    12.4951    -3.6788    13.0802    23.5183
dist =
     8.1827
```

## See Also

`meanm`, `stdm`

## Purpose

Standard deviation for geographic points

## Syntax

```
[latdev,londev] = stdm(lat,lon)
[latdev,londev] = stdm(lat,lon,ellipsoid)
[latdev,londev] = stdm(lat,lon,units)
```

[latdev,londev] = stdm(lat,lon) returns row vectors of the latitude and longitude geographic standard deviations for the data points specified by the columns of lat and lon.

[latdev,londev] = stdm(lat,lon,ellipsoid) specifies the elliptical definition of the Earth to be used with the two-element ellipsoid vector. The default ellipsoid model is a spherical Earth, which is sufficient for most applications. Output measurements are in terms of the distance units of the ellipsoid vector.

[latdev,londev] = stdm(lat,lon,units) indicates the angular units of the data. When the standard angle string *units* is omitted, 'degrees' is assumed. Output measurements are in terms of these *units* (as arc length distance).

If a single output argument is used, then geodevs = [latdev longdev]. This is particularly useful if the original lat and lon inputs are column vectors.

## Background

Determining the deviations of geographic data in latitude and longitude is more complicated than simple sum-of-squares deviations from the data averages. For latitude deviation, a straightforward angular standard deviation calculation is performed from the *geographic mean* as calculated by meanm. For longitudes, a similar calculation is performed based on data *departure* rather than on angular deviation. See “Geographic Statistics” on page 9-2 in the *Mapping Toolbox User’s Guide*.

## Examples

Create latitude and longitude lists using the worldcities data set and obtain standard distance deviation for group (compare with the example for stdist):

```
cities = shaperead('worldcities.shp', 'UseGeoCoords', true);
```

```
Paris = strmatch('Paris',{cities(:).Name});
London = strmatch('London',{cities(:).Name});
Rome = strmatch('Rome',{cities(:).Name});
Madrid = strmatch('Madrid',{cities(:).Name});
Berlin = strmatch('Berlin',{cities(:).Name});
Athens = strmatch('Athens',{cities(:).Name});
lat = [cities(Paris).Lat cities(London).Lat...
       cities(Rome).Lat cities(Madrid).Lat...
       cities(Berlin).Lat cities(Athens).Lat]
lon = [cities(Paris).Lon cities(London).Lon...
       cities(Rome).Lon cities(Madrid).Lon...
       cities(Berlin).Lon cities(Athens).Lon]
[latstd,lonstd]=stdm(lat,lon)

lat =
    48.8708    51.5188    41.9260    40.4312    52.4257    38.0164
lon =
     2.4131    -0.1300    12.4951    -3.6788    13.0802    23.5183
latstd =
     2.7640
lonstd =
    68.7772
```

## See Also

departure, filterm, hista, histr, meanm, stdist



**Purpose**

Project stem plot map on map axes

**Syntax**

```
h = stem3m(lat,lon,z)
h = stem3m(lat,lon,z,LineStyle)
h = stem3m(lat,lon,z,PropertyName,PropertyValue,...)
```

`h = stem3m(lat,lon,z)` displays a stem plot on the current map axes. Stems are located at the points (lat,lon) and extend from an altitude of 0 to the values of z. The coordinate inputs should be in the same AngleUnits as the map axes. It is important to note that the selection of z-values will greatly affect the 3-D look of the plot. Regardless of AngleUnits, the x and y limits of the map axes are at most  $-\pi$  to  $+\pi$  and  $-\pi/2$  to  $+\pi/2$ , respectively. This means that for most purposes, appropriate z values would be on the order of 1 to 3, not 10 to 30. The axes DataAspectRatio property can be used to adjust the appearance of the graphic. The handles of the displayed stem lines can be returned in h.

`h = stem3m(lat,lon,z,LineStyle)` allows the style of the stem plot's lines to be specified with any string *LineStyle* recognized by the MATLAB line function.

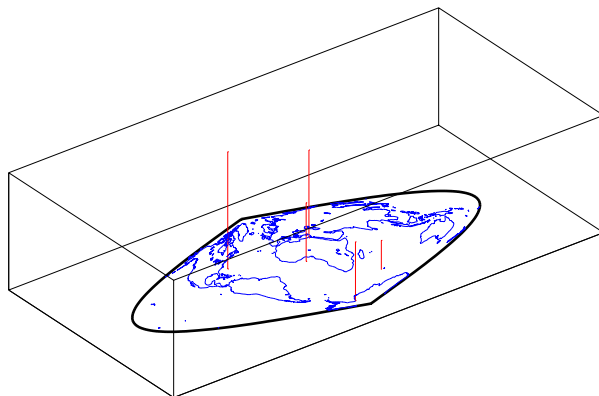
`h = stem3m(lat,lon,z,PropertyName,PropertyValue,...)` allows any property/value pair recognized by the MATLAB line function to be specified for the stems.

**Description**

A stem plot displays data as lines extending normal to the xy-plane, in this case, on a map.

**Examples**

```
load coast
axesm sinusoid; view(3)
h = framem; set(h,'zdata',zeros(size(lat)))
plotm(lat,long)
ptlat = [0 30 30 -50 -78]';
ptlon = [0 30 -70 65 -35]';
ptz = [1 1.5 2 .5 1]';
stem3m(ptlat,ptlon,ptz,'r-')
```



**See Also**

scatterm

**Purpose** Convert strings to angles in degrees

**Syntax** `angles = str2angle(strings)`  
`angles = str2angle(strings)` converts strings containing latitudes and/or longitudes, expressed in one of four different formats of degree-minutes-seconds, to numeric angles in units of degrees.

Format Description	Example
Degree Symbol, Single/Double Quotes	'123 30' '00"W'
'd', 'm', 's' Separators	'123d30m00sW'
Minus Signs as Separators	'123-30-00W'
"Packed DMS"	'1233000W'

Input must conform closely to the examples provided; in particular, the seconds field must be included, even if it is not significant. Except in Packed DMS format, the seconds field can contain a fractional component. Sign characters are not supported; terminate each string with 'N' for positive latitude, 'S' for negative latitude, 'E' for positive longitude, or 'W' for negative longitude. `strings` is string or a cell array of strings. For backward compatibility, `strings` can also be a character matrix. If more than one angle is represented, `strings` can either contain homogeneous or heterogeneous formatting (see example). `angles` is a column vector of class `double`.

```
Example
strs = {'23 30' '00"N', '23-30-00S', '123d30m00sE', '1233000W'}
strs =
    '23 30' '00"N'    '23-30-00S'    '123d30m00sE'    '1233000W'

str2angle(strs)

ans =
    23.5
```

# str2angle

---

```
-23.5  
123.5  
-123.5
```

```
strs = strvcat(strs{:})
```

```
strs =  
23 30'00"N  
23-30-00S  
123d30m00sE  
1233000W
```

```
str2angle(strs)
```

```
ans =  
23.5  
-23.5  
123.5  
-123.5
```

## See Also

[angl2str](#)

## Purpose

Project and add geolocated data grid to current map axes

## Syntax

```
surfacem(lat,lon,Z)
surfacem(lat,lon,Z,alt)
surfacem(Z)
surfacem(Z,gratsize)
surfacem(lat,lon,Z,prop1,val1,prop2,val2,...)
surfacem(lat,lon,Z,alt,prop1,val1,prop2,val2,...)
h = surfacem(...)
```

`surfacem(lat,lon,Z)` projects a data grid on the current map axes, constructing a MATLAB graphics surface object. The surface lies flat in the  $Z = 0$  plane with its `CData` property set to 'grid'. `lat` and `lon` always contain latitude and longitude values, but they may have various sizes and shapes. The sizes and shapes of `lat` and `lon` affect their interpretation, and also determine whether the default `FaceColor` property of the surface is 'flat' or 'texturemap'. The four options for `lat` and `lon` are

- 2-D arrays (matrices) having the same size as `Z`: `lat` and `lon` are treated as geolocation arrays specifying the precise location of each vertex. `FaceColor` is 'flat'.
- 2-D arrays having a different size than `Z`: `lat` and `lon` define a “graticule” mesh that may be either larger or smaller than `Z`. `lat` and `lon` must match each other in size. `FaceColor` is 'texturemap'. If the `lat` and `lon` arrays have less detail than `Z`, the curves in the graticule will become straightened. Using a coarse graticule can conserve display memory. Conversely, using a finer graticule than `Z` can help a relatively coarse data grid conform to a curved graticule.
- Vectors having more than two elements: The elements of `lat` are repeated to form a graticule mesh with size equal to `numel(lat)-by-numel(lon)`. `FaceColor` is 'flat' if the graticule mesh matches `Z` in size, and 'texturemap' otherwise.
- Vectors having exactly two elements each `lat` and `lon` represent the limits of a regular data grid and are expanded into a 50-by-100

# surfacem

---

graticule. FaceColor is 'texturemap' except when Z is precisely 50-by-100.

surfacem(lat,lon,Z,alt) sets the ZData property of the surface to alt, resulting in a 3-D surface. lat and lon can have any of the shapes/sizes listed above, but must expand into a mesh that matches alt in size. CData is set to Z. FaceColor is 'texturemap' unless Z matches alt in size.

surfacem(Z) constructs a 50-by-100 graticule mesh matching the latitude and longitude limits of the current map axes. The surface lies in the  $z = 0$  plane and FaceColor is 'texturemap' except when Z is precisely 50-by-100.

surfacem(Z,gratsize) constructs a graticule mesh with size equal to gratsize (a two-element vector) and matching the latitude and longitude limits of the current map axes. If the size of Z equals gratsize, a 3-D surface results and FaceColor is 'flat'. Otherwise, the surface lies in the  $z = 0$  plane, and FaceColor is 'texturemap'.

surfacem(lat,lon,Z,prop1,val1,prop2,val2,...) and surfacem(lat,lon,Z,alt,prop1,val1,prop2,val2,...) apply additional MATLAB graphics properties to the surface, via property/value pairs. Any property accepted by the surface function may be specified, except for XData, YData, and ZData.

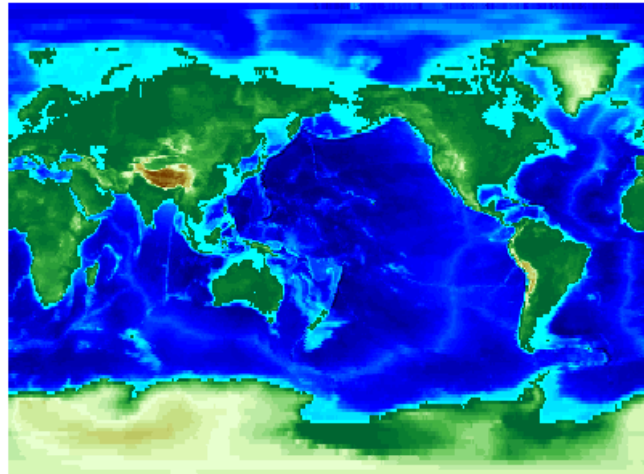
h = surfacem(...) returns a handle to the surface object.

## Remarks

Unlike meshm and surfm, surfacem always adds a surface to the current axes, regardless of hold state.

## Example

```
load topo
axesm miller
surfacem(topo,[30 30])
demcmap(topo)
```



**See Also** `geoshow`, `meshm`, `pcolorm`, `surfm`

# surflm

---

## Purpose

3-D shaded surface with lighting on map axes

## Syntax

```
h = surflm(Z)
h = surflm(Z,s)
h = surflm(lat,lon,Z)
h = surflm(lat,lon,Z,s,k)
```

`h = surflm(Z)` displays the regular data grid `Z` projected to a graticule grid the size of `Z` in accordance with the current map axes `MapProjection` property. It is displayed with a default light source. The handle `h` of the displayed surface object can be returned.

`h = surflm(Z,s)` specifies the direction of the light source. `s` is a two- or three-element vector that specifies the direction from the surface map to the light source. `s=[sx sy sz]` or `s=[azimuth elevation]`. The default `s` is 45° counterclockwise from the current view direction.

`h = surflm(lat,lon,Z)` allows you to specify your graticule. `lat` and `lon` can be vectors with elements corresponding to `Z` rows and columns, respectively, or they can be matrices the size of `Z`. The resulting graticule is the size of `Z`.

`h = surflm(lat,lon,Z,s,k)` specifies the reflectance constant. `k` is a four-element vector defining the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient. `k = [ka,kd,ks,shine]` and defaults to `[.55 .6 .4 10]`.

## Description

`surflm` is like `surf` except that it shades the monochrome map surface with a light source, and the only allowed graticule is the size of the data matrix.

## Examples

To see this, type the following. The graticule is the size of `topo` (180 x 360) and is rendered in 3-D, so it might take a while. It is also memory intensive:

```
load topo
axesm miller
surflm(topo)
```



**See Also**

surfm

**Purpose** 3-D lighted shaded relief of geolocated data grid

**Syntax**

```
surfsrlm(lat, long, Z)
surfsrlm(lat, long, Z, [azim elev])
surfsrlm(lat, long, Z, [azim elev], cmap)
surfsrlm(lat, long, Z, [azim elev], cmap, clim)
h = surfsrlm(...)
```

`surfsrlm(lat, long, Z)` displays the geolocated data grid, colored according to elevation and surface slopes. The current axes must have a valid map projection definition.

`surfsrlm(lat, long, Z, [azim elev])` displays the geolocated data grid with the light coming from the specified azimuth and elevation. Lighting is applied before the data is projected. Angles are in degrees, with the azimuth measured clockwise from North, and elevation up from the zero plane of the surface. By default, the direction of the light source is east (90° azimuth) at an elevation of 45°.

`surfsrlm(lat, long, Z, [azim elev], cmap)` displays the geolocated data grid using the provided colormap. The number of grayscales is chosen to keep the size of the shaded colormap below 256. By default, the colormap is constructed from 16 colors and 16 grays. If the vector of azimuth and elevation is empty, the default locations are used.

`surfsrlm(lat, long, Z, [azim elev], cmap, clim)` uses the provided color axis limits, which are, by default, automatically computed from the data.

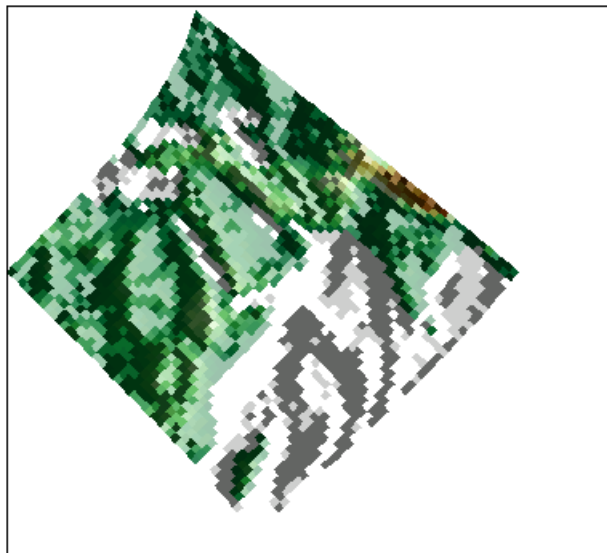
`h = surfsrlm(...)` returns the handle to the surface drawn.

**Remarks** This function effectively multiplies two colormaps, one with color based on elevation, the other with a grayscale based on the slope of the surface, to create a new colormap. This produces an effect similar to using a light on a surface, but with all of the visible colors actually in the colormap. Lighting calculations are performed on the unprojected data.

**Examples**

Create a new colormap using demcmap with white colors for the sea and default colors for land. Use this colormap for the lighted shaded relief map of the Middle East region:

```
load mapmtx
[cmap,clim] = demcmap(map1,[],[1 1 1],[]);
axesm loximuth
surflsrm(lt1,lg1,map1,[],cmap,clim)
```

**See Also**

meshlsrm, meshm, pcolor, shadere1, surfacem, surf1m, surfm

## Purpose

Project geolocated data grid on map axes

## Syntax

```
h = surfm(Z)
h = surfm(Z,npts)
h = surfm(lat,lon,Z)
h = surfm(lat,lon,Z,alt)
h = surfm(lat,lon,Z,PropertyName,PropertyValue,...)
```

`h = surfm(Z)` projects the regular data grid `Z` on a graticule grid the size of `Z` between the latitude and longitude limits of the current map axes. The handle `h` of the displayed surface can be returned.

`h = surfm(Z,npts)` results in a graticule grid defined by `npts`, which is a two-element vector of the form [latitude-points longitude-points].

`h = surfm(lat,lon,Z)` allows three other methods of defining the graticule grid. If `lat` and `lon` are matrices, they represent the actual graticule vertices as might be returned by `meshgrat`. If vectors, they are the representative coordinates of the rows and columns, respectively, of such a grid. If they are two-element vectors, they are treated as latitude and longitude limits, and a graticule mesh of size(`Z`) is calculated.

`h = surfm(lat,lon,Z,alt)` sets the `z`-axis altitude of the graticule mesh. `alt` must be the same size as `lat`. If no `alt` is supplied, the mesh is plotted at `z = 0`, unless `lat` is the same size as `Z`, in which case `zdata = Z`, and a 3-D topological map results. Since the default graticule is the size of `Z`, the default condition for `surfm` is to create the topographic map.

`h = surfm(lat,lon,Z,PropertyName,PropertyValue,...)` allows the input of property name/property value pairs to control the surface object properties. Any property supported by the standard MATLAB function `surface` except `XData`, `YData`, and `ZData` can be altered in this manner.

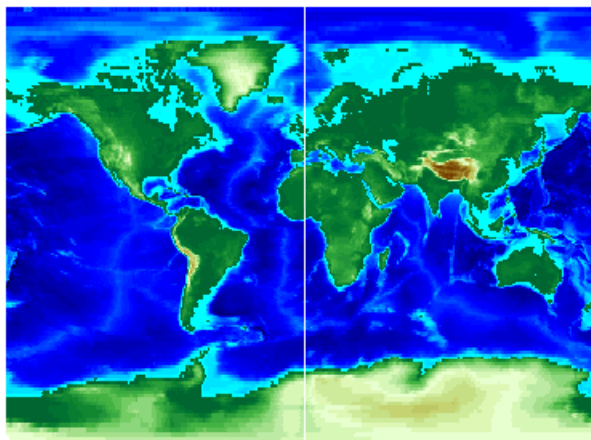
## Description

This function warps a data grid to a graticule mesh, which itself is projected according to the map axes property `MapProjection`. The fineness, or resolution, of this grid determines the quality of the

projection and the speed of plotting it. There is no hard and fast rule for sufficient graticule resolution, but in general, cylindrical projections need very few graticule points in the longitudinal direction, while complex curve-generating projections require more.

## Examples

```
load topo
axesm miller
[meshlat,meshlon] = meshgrat(topo,topolegend,[90 180]);
surfm(meshlat,meshlon,topo)
demcmap(topo)
```



## See Also

meshgrat, meshm, pcolorm, surfacem

# symbolm

---

## Purpose

Project point markers with variable size

## Syntax

```
symbolm(lat,lon,z,'MarkerType')  
symbolm(lat,lon,z,'MarkerType','PropertyName',PropertyValue,  
    ...)  
h = symbolm(...)
```

`symbolm(lat,lon,z,'MarkerType')` constructs a thematic map where the symbol size of each data point (`lat`, `lon`) is proportional to its weighting factor (`z`). The point corresponding to `min(z)` is drawn at the default marker size, and all other points are plotted with proportionally larger markers. The `MarkerType` string is a `LineStyle` string specifying a marker and optionally a color.

`symbolm(lat,lon,z,'MarkerType','PropertyName',PropertyValue,...)` applies the line properties to all the symbols drawn.

`h = symbolm(...)` returns a vector of handles to the projected symbols. Each symbol is projected as an individual line object.

## See also

`stem3m`, `plotm`, `plot`

**Purpose** Set Tag property of map graphics object

**Syntax** tagm(hndl,tagstr)

tagm(hndl,tagstr) sets the Tag property of each object designated in the vector of handles hndl to the associated string (row) of the matrix of strings tagstr.

This property is recognized by the `namem` and `handlem` functions.

**Examples** Normally, a plotted line has a name of 'line':

```
axesm miller
lats = [3 2 1 1 2 3]; longs = [7 8 9 7 8 9];
h=plotm(lats,longs);

untagged = namem(h)
untagged =
line
```

The tagm function can rename it:

```
tagm(h,'testpath');
tagged = namem(h)
tagged =
testpath
```

**See Also** clma, clmo, handlem, hidem, namem, showm

# tbase

---

**Purpose** Read 5-minute global terrain elevations from TerrainBase

**Syntax**

```
[Z,refvec] = tbase(scalefactor)
[Z,refvec] = tbase(scalefactor,latlim,lonlim)
```

[Z,refvec] = tbase(scalefactor) reads the data for the entire world, reducing the resolution of the data by the specified scale factor. The result is returned as a regular data grid and an associated three-element referencing vector.

[Z,refvec] = tbase(scalefactor,latlim,lonlim) reads the data for the part of the world within the latitude and longitude limits. The limits must be two-element vectors in units of degrees.

**Background** TerrainBase is a global model of terrain and bathymetry on a regular 5-minute grid (approximately 10 km resolution). It is a compilation of the best available public domain data from almost 20 different sources, including the DCW-DEM and ETOPO5. The model is currently under development and will be updated as new data sources become available. The data set was created by the National Geophysical Data Center and World Data Center-A for Solid Earth Geophysics in Boulder, Colorado.

**Remarks** Elevations and depths are given in meters above or below mean sea level.

The tbase.bin file is available on CD-ROM from

```
NOAA/NGDC
Mail Code E/GC3
325 Broadway
Boulder, CO 80303
USA
Tel: (303) 497-6338
Fax: (303) 497-6513
```

The data and documentation are available over the Internet via http and anonymous ftp.



**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

No byte-swapping or line-ending conversion is required.

**Examples**

Read every 10th point in the data set:

```
[Z,refvec] = tbase(10);
whos

      Name              Size           Bytes  Class

      Z                 216x432         746496  double array
      refvec            1x3              24      double array

limitm(Z,refvec)

ans =
    -90    90     0    360
```

Read data for Korea and Japan at the full resolution:

```
scalefactor = 1; latlim = [30 45]; lonlim = [115 145];
[Z,refvec] = tbase(scalefactor,latlim,lonlim);
whos datagrid

      Name      Size           Bytes  Class

      Z         180x360         518400  double array
```

**See Also**

gtopo30, etopo, usgsdem

# textm

---

**Purpose** Project text annotation on map axes

**Syntax**

```
textm(lat,lon,string)
textm(lat,lon,z,string)
textm(lat,lon,z,string,PropertyName,PropertyValue,...)
h = textm(...)
```

`textm(lat,lon,string)` projects the text in `string` onto the current map axes at the locations specified by the `lat` and `lon`. The units of `lat` and `lon` must match the 'angleunits' property of the map axes. If `lat` and `lon` contain multiple elements, `textm` places a text object at each location. In this case `string` may be a cell array of strings with the same number of elements as `lat` and `lon`. (For backward compatibility, `string` may also be a 2-D character array such that `size(string,1)` matches `numel(lat)`).

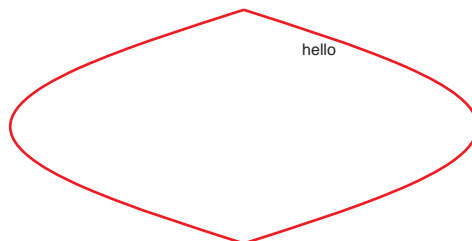
`textm(lat,lon,z,string)` draws the text at the altitude(s) specified in `z`, which must be the same size as `lat` and `lon`. The default altitude is 0.

`textm(lat,lon,z,string,PropertyName,PropertyValue,...)` sets the text object properties. All properties supported by the MATLAB text function are supported by `textm`.

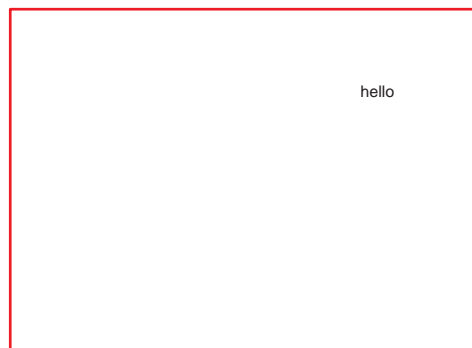
`h = textm(...)` returns the handles to the text objects drawn.

**Example** The feature of `textm` that distinguishes it from the standard MATLAB text function is that the text object is projected appropriately. Type the following:

```
axesm sinusoid
framem('FEdgeColor','red')
textm(60,90,'hello')
```



```
figure; axesm miller  
framem('FEdgeColor','red')  
textm(60,90,'hello')
```



The string 'hello' is placed at the same geographic point, but it appears to have moved relative to the axes because of the different

## textm

---

projections. If you change the projection using the `setm` function, the text moves as necessary. Use `text` to fix text objects in the axes independent of projection.

### See Also

`axesm`, `text` (MATLAB function)

**Purpose**

Read TIGER/Line data

**Syntax**

```
[CL,PR,SR,RR,H,AL,PL] = tgrline(filename)
```

```
[CL,PR,SR,RR,H,AL,PL] = tgrline(filename,year)
```

```
[CL,PR,SR,RR,H,AL,PL] = tgrline(filename,year,countyname)
```

[CL,PR,SR,RR,H,AL,PL] = tgrline(*filename*) reads a set of 1994 TIGER/Line files which share the same filename, but different extensions. The results are returned in a set of geographic data structures (geostruct1s) tagged with feature names and containing:

- county boundaries (CL)
- primary roads (PR)
- secondary roads (SR)
- railroads (RR)
- hydrography (H)
- area landmarks (AL)
- point landmarks (PL)

[CL,PR,SR,RR,H,AL,PL] = tgrline(*filename*,*year*) reads the TIGER line files in the format from that year. The layout of TIGER/Line files is updated periodically and filename extensions may change from year to year. Valid years are 1990, 1992, 1994, 1995, 1999, 2000, 2002, 2003, and 2004.

[CL,PR,SR,RR,H,AL,PL] = tgrline(*filename*,*year*,*countyname*) uses the string *countyname* to tag the county data.

The United States Census Bureau distributes TIGER/Line data over the Internet and via CD-ROM or DVD.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

TIGER and TIGER/Line are registered trademarks of the Census Bureau.

## Background

TIGER/Line files contain vector map data used to support mapping for the U.S. Census Bureau. TIGER is an acronym for Topographically Integrated Geographic Encoding and Referencing. These files contain data for political boundaries, including states, counties, Indian reservations, and census tracts, as well as roads, railroads, hydrography, and landmarks. In addition to the geographically referenced information, the files also contain data to determine the address of an object. The data covers the United States of America and its territories or administrative units: Puerto Rico, the Virgin Islands of the United States, American Samoa, Guam, the Commonwealth of the Northern Mariana Islands, the Republic of Palau, the other Pacific entities that were part of the Trust Territory of the Pacific Islands (the Republic of the Marshall Islands and the Federated States of Micronesia), and the Midway Islands. The most common application of this data is to commercial CD-ROM road atlases.

## Remarks

This function reads only a subset of the data in the TIGER/Line files. For example, the function does not return local roads, zip codes, or census tract numbers.

## Examples

Read from the data for Washington, D.C.:

```
[CL,PR,SR,RR,H,AL,PL] = tgrline('TGR11001',1994,'Wash,DC');
```

## See Also

tigermif, tigerp, shaperead

**Purpose** Read TIGER MIF (MapInfo Interchange Format) thinned boundary files

---

**Note** tigermif is obsolete; use shaperead to read TIGER data in shapefile format

---

## Syntax

```
tigermif(namesstruc)  
tigermif(namesstruc, filename)  
tigermif(namesstruc, filename, pstruc)  
tigermif(namesstruc, filename, pstruc, tstruc)  
tigermif(namesstruc, filename, pstruc, tstruc, getcodes)  
pstruc = tigermif(...)  
[pstruc, tstruc] = tigermif(...)
```

tigermif is obsolete and may be removed in the future. Download the newer shapefile versions of the thinned cartographic boundary files and use shaperead instead.

tigermif(namesstruc) reads a TIGER thinned boundary file in the MIF format. The user selects the file interactively, but must provide the structure containing the names (as returned by the fipsname function). The patch data is returned in a Mapping Toolbox geographic data structure.

tigermif(namesstruc, *filename*) reads the MIF file named in the string *filename*. The filename is provided with the .MIF extension. If the file is not found, a dialog box is activated to allow the user to select a file interactively.

tigermif(namesstruc, *filename*, pstruc) appends the patch data to the existing structure, pstruc.

tigermif(namesstruc, *filename*, pstruc, tstruc) appends the data in the file to the existing patch and text geographic data structures, pstruc and tstruc. The text structure contains labels for the patches. This form is used with two output arguments. The arguments for the existing structures can be set to empty matrices if none are available.

# tigermif

---

`tigermif(namesstruc, filename, pstruc, tstruc, getcodes)` returns only the data matching the scalar or vector of numeric FIPS codes.

`pstruc = tigermif(...)` saves the returned patch data in `pstruc`.

`[pstruc, tstruc] = tigermif(...)` saves the returned patch data in `pstruc` and text labels in `tstruc`. Both are geographic data structures.

## Background

TIGER Thinned Boundary files are lower resolution extracts from the U.S. Census Bureau's detailed TIGER/Line database. U.S. state and county boundaries are available in the MapInfo Interchange Format (MIF).

## Remarks

TIGER data files are available over the Internet, although MIF-formatted ones are not very prevalent. You will have better luck finding TIGER data in shapefile format.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

Read the names file (contains the names of U.S. states and territories):

```
namestruc = fipsname('st_name.dat')

namestruc =
1x57 struct array with fields:
    name
    id
```

Read the file containing Hawaii's thinned state boundaries and text labels into a Mapping Toolbox geographic data structure:

```
[ps,ts] = tigermif(namestruc,'ST15.MIF')

ps =
```



```

        lat: [1585x1 double]
        long: [1585x1 double]
        type: 'patch'
    otherproperty: {}
        tag: 'Hawaii'
    altitude: []
ts =
        lat: 21.1343
        long: -157.9524
        type: 'text'
        tag: 'maptext'
    otherproperty: {1x2 cell}
    string: {1x1 cell}
    altitude: []

```

Read the file containing Alaska's thinned state boundaries, and append it to the Hawaii data:

```

[ps,ts] = tigermif(namestruc,'ST02.MIF',ps,ts)

ps =
1x2 struct array with fields:
    lat
    long
    type
    otherproperty
    tag
    altitude
ts =
1x2 struct array with fields:
    lat
    long
    type
    tag
    otherproperty
    string
    altitude

```

# tigermif

---

Get the state boundaries and text labels for part of New England. The FIPS codes for Connecticut, Massachusetts, and Rhode Island are 9, 25, and 44, respectively:

```
[ps,ts] = tigermif(namestruc,'ST_LOW48.MIF',[],[],[9 25 44])
```

```
ps =  
1x3 struct array with fields:  
    lat  
    long  
    type  
    otherproperty  
    tag  
    altitude  
ts =  
1x3 struct array with fields:  
    lat  
    long  
    type  
    tag  
    otherproperty  
    string  
    altitude
```

## See Also

dcwdata, fipsname, shaperead, tgrline, tigerp

**Purpose**

Read TIGER p and pa (ArcInfo format) thinned boundary files

---

**Note** tigerp is obsolete; use shaperead to read TIGER data in shapefile format

---

**Syntax**

```
tigerp(namesstruc)
tigerp(namesstruc, filename)
tigerp(namesstruc, filename, pstruc)
tigerp(namesstruc, filename, pstruc, tstruc)
tigerp(namesstruc, filename, pstruc, tstruc, getcodes)
pstruc = tigerp(...)
[pstruc, tstruc] = tigerp(...)
```

tigerp is obsolete and may be removed in the future. Download the newer shapefile versions of the thinned cartographic boundary files and use shaperead instead.

tigerp(namesstruc) reads a TIGER thinned boundary file in the ArcInfo format. The user selects the file interactively, but must provide the structure containing the names (as returned by the fipsname function). The patch data is returned in a Mapping Toolbox geographic data structure.

tigerp(namesstruc, filename) reads the ArcInfo file named in the string *filename*. The filename is provided without the '\_p' or '\_pa' extension.

tigerp(namesstruc, filename, pstruc) appends the patch data to the existing structure, pstruc.

tigerp(namesstruc, filename, pstruc, tstruc) appends the data in the file to the existing patch and text geographic data structures, pstruc and tstruc. The text structure contains labels for the patches. This form is used with two output arguments. The arguments for the existing structures can be set to empty matrices if none are available.

tigerp(namesstruc, filename, pstruc, tstruc, getcodes) returns only the data matching the scalar or vector of numeric FIPS codes.

```
pstruc = tigerp(...) saves the returned patch data in pstruc.  
[pstruc,tstruc] = tigerp(...) saves the returned patch data in  
pstruc and text labels in tstruc. Both are geographic data structures.
```

## Background

TIGER Thinned Boundary files are lower resolution extracts from the U.S. Census Bureau's more detailed TIGER/Line database. State, county, minor civil division, census tract/block numbering area, American Indian reservation/Alaska native village statistical area, Alaska native regional corporation, urbanized areas, metropolitan areas, and congressional district boundaries are available in the ArcInfo format.

## Remarks

Coordinate values are based on Clarke's spheroid of 1866 and the North American Datum, 1927 (NAD27).

## Examples

Read the names file with the names of all counties in the U.S. and territories. This file is in FIPS format:

```
namestruc = fipsname('co_name.dat')  
  
namestruc =  
1x3248 struct array with fields:  
    name  
    id
```

Read the file containing Alaska's thinned county boundaries into a Mapping Toolbox geographic data structure:

```
[ps,ts] = tigerp(namestruc,'co_02_p.dat')  
  
ps =  
1x26 struct array with fields:  
    lat  
    long  
    type  
    otherproperty  
    altitude
```

```
    tag
ts =
1x26 struct array with fields:
    lat
    long
    type
    tag
    otherproperty
    altitude
    string
```

Read only the Aleutians East and West:

```
[ps,ts] = tigerp(namestruc,'co_02_p.dat',[],[],[2013 2016])

ps =
1x2 struct array with fields:
    lat
    long
    type
    otherproperty
    altitude
    tag
ts =
1x2 struct array with fields:
    lat
    long
    type
    tag
    otherproperty
    altitude
    string
```

## See Also

dcwdata, fipsname, shaperead, tgrline, tigermif

# tightmap

---

**Purpose** Remove white space around map

**Syntax** `tightmap`

`tightmap` sets the MATLAB axis limits to be tight around the map in the current axes. This eliminates or reduces the white border between the map frame and the axes box. Use `axis auto` to undo `tightmap`.

**Examples** Display a map of Africa. Notice the white space between the map frame and the edge of the axes box.

```
axesm('miller','maplatlim',[-40 40],'maplonlim',[-20 60])
framem; gridm; mlabel; plabel
load coast
plotm(lat, long)
```

Now use `tightmap` to reduce the wasted space:

```
tightmap
```

**Limitations** The axis limits are fixed. If a change in the projection parameters changes the size or position of the map display within the projected coordinate system, execute `tightmap` again. Also note that `tightmap` needs to be re-applied following any call to `setm` that causes projected map objects to be re-projected.

**See Also** `panzoom`, `zoom`, `paperscale`, `axesscale`, `previewmap`

**Purpose** Format time strings

---

**Note** The `time2str` function is obsolete and errors when used. It will be removed from the next version of Mapping Toolbox.

---

## Syntax

```
str = time2str(timein)
str = time2str(timein,clock)
str = time2str(timein,clock,format)
str = time2str(timein,clock,format,units)
str = time2str(timein,clock,format,digits)
```

`str = time2str(timein)` converts a numerical vector of times to a string matrix. The output string matrix is useful for the display of times.

`str = time2str(timein,clock)` uses the specified `clock` input to construct the string matrix. Allowable `clock` strings are '24' (default) for a 24-hour clock, '12' for a 12-hour clock, and 'nav' for a navigational hour clock.

`str = time2str(timein,clock,format)` uses the specified `format` input to construct the string matrix. Allowable for `format` strings are 'hms', for hours, minutes, and seconds, and 'hm' (default), for hours and minutes.

`str = time2str(timein,clock,format,units)` defines the units in which the input times are supplied. Any valid time `units` string can be entered. If omitted, 'hours' is assumed.

`str = time2str(timein,clock,format,digits)` indicates the power of ten to be included for the seconds (if `format` = 'hms') or minutes (if `format` = 'hm'). The default value is 0, so nothing is returned to the right of the decimal ( $10^0$  is the ones column). For example, if `digits` = -2, seconds are returned down to the hundredths column.

## Description

The purpose of this function is to make time-valued variables into strings suitable for map display.

## Examples

13 hours, 56 minutes, 44 seconds in hms format is 1356.44.

```
time = 1356.44;
str = time2str(time,'12','hms','hms')

str =
1:56:44 PM
```

For hm format, appropriate rounding occurs:

```
str = time2str(time,'12','hm','hms')
str =
1:57 PM
```

The 24-hour and navigational representations are

```
str = time2str(time,'24','hms','hms')

str =
13:56:44

str = time2str(time,'nav','hms','hms')

str =
1356'''
```

Navigational times are four digits; if seconds are included, they are rounded to the nearest 15 seconds, which are represented by tick marks (0 = none, 15 = ', 30 = '' , 45 = ''' ).

Consider the hms format time 1356.4456 for rounding purposes:

```
str = time2str(1356.4456,'12','hms','hms',-2) % hundredths

str =
1:56:44.56 PM

str = time2str(1356.4456,'12','hms','hms',-1) % tenths
```



```
str =  
1:56:44.6 PM
```

# timedim

---

**Purpose** Convert time units or encodings

---

**Note** The `timedim` function is obsolete and errors when used. It will be completely removed from the next version of Mapping Toolbox.

---

**Syntax**

```
timeout = timedim(timein,from,to)
```

`timeout = timedim(timein,from,to)` returns the value of the input time `timein`, which is in units specified by the valid time units string `from`, in the desired units given by the valid time units string `to`. Valid time units strings are

'hours' or 'hr'	for decimal hours
'seconds' or 'sec'	for seconds
'hms'	for hours-minutes-seconds
'hm'	for hours-minutes

**Examples** Convert from hours to seconds:

```
timedim(2.56,'hours','seconds')
```

```
ans =  
    9216
```

What is the difference between `hms` and `hm` (best displayed in *bank format*)?

```
format bank  
timedim(2.56,'hours','hms')
```

```
ans =  
    233.36
```

```
timedim(2.56,'hours','hm')
```

```
ans =
```

234.00

The hm answer is the hms answer correctly rounded to whole minutes (that is, rounded based on 60 seconds per minute, not 100).

**See Also**

angledim

# timezone

---

**Purpose** Time zone based on longitude

**Syntax**

```
[zd,zltr,zone] = timezone(long)
[zd,zltr,zone] = timezone(long,units)
```

[zd,zltr,zone] = timezone(long) returns an integer zone description, zd, an alphabetical string zone indicator, zltr, and a string, zone, with the complete zone description and alphabetical zone indicator corresponding to the input longitude long.

[zd,zltr,zone] = timezone(long,units) specifies the angular units with a standard angle *units* string. The default value is 'degrees'. Valid *units* are:

- 'degrees' — decimal degrees
- 'radians'

**Examples**

Given that it is locally 1330 (1:30 p.m.) at a longitude of 75°W, determine GMT:

```
[zd,zltr,zone] = timezone(-75,'degrees')
```

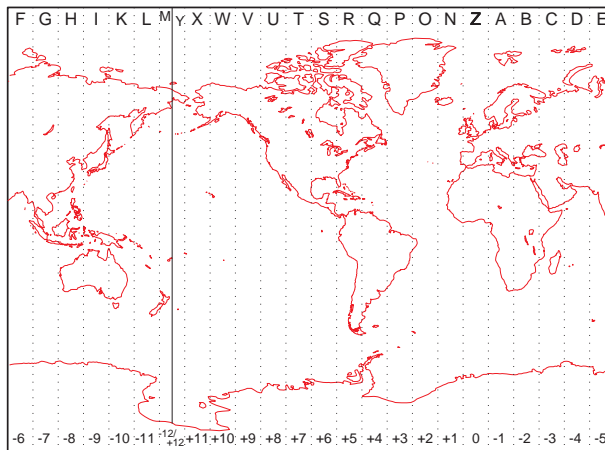
```
zd =
     5
zltr =
    R
zone =
    +5 R
```

Greenwich Mean Time (GMT) is 1330 plus five hours, or 1830 (6:30 p.m.).

**Background**

Time is determined by the position of the Sun relative to the prime meridian, the zero longitude line running through Greenwich, England. When this meridian lies directly below the Sun, it is noon GMT. For local times elsewhere, the Earth is divided into 15° longitude bands, each centered on a central meridian. When a central meridian lies directly

below the Sun, Local Mean Time (LMT) in that zone is noon. The zone description is an integer that when added to LMT gives GMT. For notational convenience, each zone is also given an alphabetical indicator. The indicator at Greenwich is *Z*, so GMT is often called *ZULU time*.



Note that there are actually 25 time zones, because the zone centered on the International Date Line (180° E/W) is split into two: “+12 Y” and “-12 M.”

**Limitations**

National and local governments set their own time zone boundaries for political or geographic convenience. The `timezone` function does not account for statutory deviations from the meridian-based system.

## Purpose

Project Tissot indicatrices on map axes

## Syntax

```
h = tissot
h = tissot(spec)
h = tissot(spec,linestyle)
h = tissot(linestyle)
h = tissot(spec,PropertyName,PropertyValue,...)
h = tissot(linestyle,PropertyName,PropertyValue,...)
```

`h = tissot` plots the default Tissot diagram, as described above, on the current map axes and returns handles for the displayed indicatrices.

`h = tissot(spec)` allows you to specify plotting parameters of the displayed Tissot diagram as described above.

`h = tissot(spec,linestyle)` and `h = tissot(linestyle)` specify any *linestyle* string recognized by the standard MATLAB function `line` to set the line style of the Tissot indicatrices.

`h = tissot(spec,PropertyName,PropertyValue,...)` and `h = tissot(linestyle,PropertyName,PropertyValue,...)` allow the specification of any property and value recognized by the `line` function.

## Background

Tissot indicatrices are plotting symbols that are useful for understanding the various distortions of a given map projection. The indicatrices are circles of identical true radius on the Earth's surface. When plotted on a map projection, they indicate whether the projection has certain features. If the plotted indicatrices all enclose the same area, the projection is equal area (for example, a Sinusoidal projection would have this feature). If they all remain circular, then conformality is indicated (a Mercator projection has this property). Distortions in meridional or parallel distance are exhibited by flattened or stretched indicatrices. Many projections will show very even, circular indicatrices in some regions, often near the center, and wildly distorted indicatrices in others, such as near the edges. The Tissot diagram is therefore very useful in analyzing the appropriateness of a projection to a given purpose or region. Chapter 12, "Map Projections — By Category" of this guide includes Tissot diagrams for every projection on a global scale.

## Description

The general layout of the Tissot diagram is defined by the specification vector `spec`.

```
spec = [Radius]
spec = [Latint,Longint]
spec = [Latint,Longint,Radius]
spec = [Latint,Longint,Radius,Points]
```

`Radius` is the small circle radius of each indicatrix circle. If entered, it should be in the same units as the map axes `Geoid`. The default radius is 1/10th the radius of the sphere.

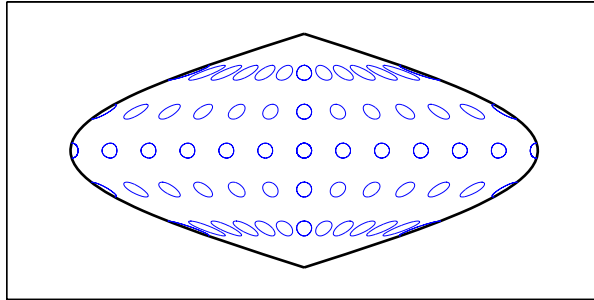
`Latint` is the latitude interval between indicatrix circle centers. If entered it should be in the map axes `AngleUnits`. The default value is one circle every 30° of latitude (that is, 0°, +/-30°, etc.).

`Longint` is the longitude interval between indicatrix circle centers. If entered it should be in the map axes `AngleUnits`. The default value is one circle every 30° of longitude (that is, 0°, +/-30°, etc.).

`Points` is the number of plotting points per circle. The default is 100 points.

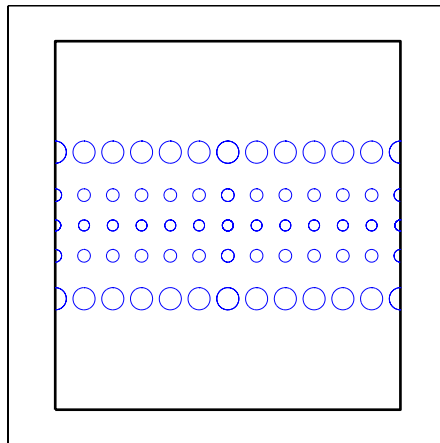
## Examples

```
axesm sinusoid; framem
tissot
```



The Sinusoidal projection is equal area.

```
setm(gca, 'MapProjection', 'Mercator')
```



The Mercator projection is conformal.



**See Also**

`mdistort`, `distortcalc`

See Chapter 12, “Map Projections — By Category”

# toDegrees

---

**Purpose** Convert angles to degrees

**Syntax**

```
[angle1InDegrees, angle2InDegrees,  
 ...] = toDegrees(fromUnits, angle1, angle2, ...)
```

[angle1InDegrees, angle2InDegrees, ...] =  
toDegrees(*fromUnits*, angle1, angle2, ...) converts  
angle1, angle2, ... to degrees from the specified angle units.  
*fromUnits* can be either 'degrees' or 'radians' and may  
be abbreviated. The inputs angle1, angle2, ... and their  
corresponding outputs are numeric arrays of various sizes, with  
size(angleNinDegrees) matching size(angleN).

**See Also** fromDegrees, fromRadians, rad2deg, toRadians

**Purpose** Convert angles to radians

**Syntax** `[angle1InRadians, angle2InRadians, ...] = toRadians(fromUnits, angle1, angle2, ...)`

`[angle1InRadians, angle2InRadians, ...] = toRadians(fromUnits, angle1, angle2, ...)` converts `angle1`, `angle2`, ... to radians from the specified angle units. *fromUnits* can be either 'degrees' or 'radians' and may be abbreviated. The inputs `angle1`, `angle2`, ... and their corresponding outputs are numeric arrays of various sizes, with `size(angleNinRadians)` matching `size(angleN)`.

**See Also** `deg2rad`, `fromDegrees`, `fromRadians`, `toDegrees`

## Purpose

Track segments to connect navigational waypoints

## Syntax

```
[latrkr,lonrkr] = track(waypts)
[latrkr,lonrkr] = track(waypts,units)
[latrkr,lonrkr] = track(lat,lon)
[latrkr,lonrkr] = track(lat,lon,ellipsoid)
[latrkr,lonrkr] = track(lat,lon,ellipsoid,units,npts)
[latrkr,lonrkr] = track(method,lat,...)
trkpts = track(lat,lon...)
```

[latrkr,lonrkr] = track(waypts) returns points in latrkr and lonrkr along a track between the waypoints provided in navigational track format in the two-column matrix waypts. The outputs are column vectors in which successive segments are delineated with NaNs.

[latrkr,lonrkr] = track(waypts,units) specifies the units of the inputs and outputs, where *units* is any valid angle unit string. The default is 'degrees'.

[latrkr,lonrkr] = track(lat,lon) allows the user to input the waypoints in two vectors, lat and lon.

[latrkr,lonrkr] = track(lat,lon,ellipsoid) specifies the elliptical definition of the Earth with a two-element ellipsoid model vector ellipsoid. The default ellipsoid is a spherical Earth, which is sufficient for most applications.

[latrkr,lonrkr] = track(lat,lon,ellipsoid,units,npts) establishes how many intermediate points are to be calculated for every track segment. By default, npts is 30.

[latrkr,lonrkr] = track(method,lat,...) establishes the logic to be used to determine the intermediate points along the track between waypoints. Because this is a navigationally motivated function, the default method is 'rh', which results in rhumb line logic. Great circle logic can be specified with 'gc'.

trkpts = track(lat,lon...) compresses the output into one two-column matrix, trkpts, in which the first column represents latitudes and the second column, longitudes.

## Examples

The track function is useful for generating data in order to display tracks. Lieutenant Sextant is the navigator of the USS Neversail. He is charged with plotting a track to take Neversail from the Straits of Gibraltar to Port Said, Egypt, the northern end of the Suez Canal. He has picked appropriate waypoints and now would like to display the track for his captain's approval.

First, display a chart of the Mediterranean Sea:

```
load coast
axesm('mercator','MapLatLimit',[28 47],'MapLonLimit',[-10 37]...
      'Grid','on','Frame','on','MeridianLabel','on','ParallelLabel','on')
geoshow(lat,long,'DisplayType','line','color','b')
```

These are the waypoints Lt. Sextant has selected:

```
waypoints = [36,-5; 36,-2; 38,5; 38,11; 35,13; 33,30; 31.5,32]
```

```
waypoints =
 36.0000   -5.0000
 36.0000   -2.0000
 38.0000    5.0000
 38.0000   11.0000
 35.0000   13.0000
 33.0000   30.0000
 31.5000   32.0000
```

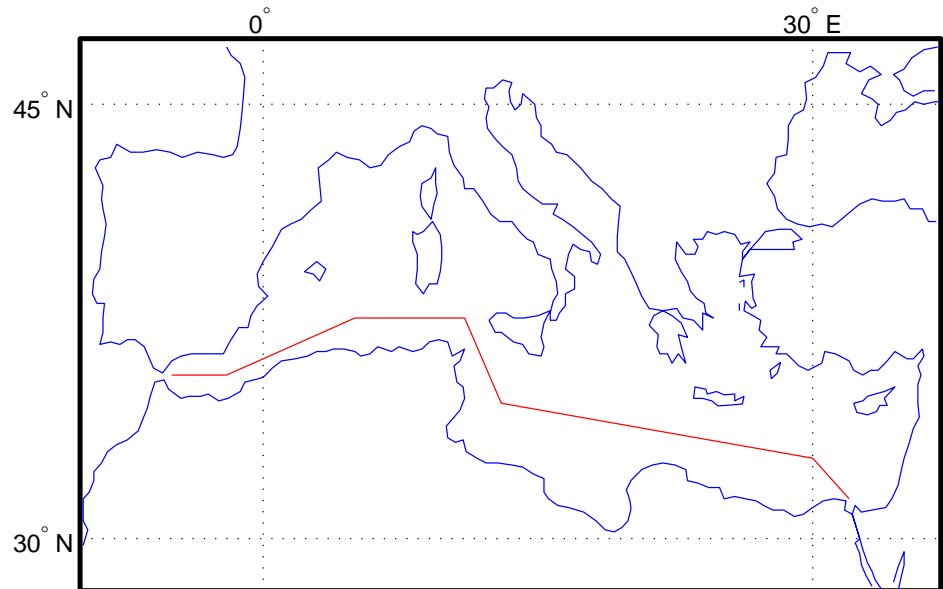
Now display the track:

```
[lptrk,lptrk] = track('rh',waypoints,'degrees');
geoshow(lptrk,lptrk,'DisplayType','line','color','r')
```

With a display this clear, the captain gladly approves the plan.

# track

---



## See Also

`dreckon`, `gcwaypts`, `legs`, `navfix`

**Purpose**

Geographic tracks from starting point, azimuth, and range

**Syntax**

```
[latrkr,lontrk] = track1(lat,lon,az)
[latrkr,lontrk] = track1(track,lat,lon,az)
[latrkr,lontrk] = track1(track,lat,lon,az,units)
[latrkr,lontrk] = track1(track,lat,lon,az,rng)
[latrkr,lontrk] = track1(track,lat,lon,az,rng,ellipsoid,
    units)
[latrkr,lontrk] = track1(track,lat,lon,az,rng,ellipsoid,
    units,npts)
pts = track1(lat,lon,az,...)
```

[latrkr,lontrk] = track1(lat,lon,az) returns, in latrkr and lontrk, points along a complete (great circle) track passing through the point specified by lat and lon with an initial azimuth at that point of az. When the inputs are column vectors, the successive tracks are stored in separate columns of latrkr and lontrk.

[latrkr,lontrk] = track1(track,lat,lon,az) allows the specification of the track logic to be employed. A string *track* of 'gc' is the default, resulting in a great circle track. A *track* of 'rh' results in a complete rhumb line track.

[latrkr,lontrk] = track1(track,lat,lon,az,units) specifies the units of the inputs and outputs, where *units* is any valid angle unit string. The default is 'degrees'.

[latrkr,lontrk] = track1(track,lat,lon,az,rng) specifies the range of the track. *rng* is a one- or two-column matrix. If *rng* has one column, the track extends from the point (lat,lon) at an azimuth of az for a distance *rng* if *rng* is positive, or at an azimuth  $az+180^\circ$  (or its angular equivalent) for a distance of  $\text{abs}(\text{rng})$  if *rng* is negative. If *rng* has two columns, the endpoints are defined as above. In this case, the segment extends from the point associated with the first column of *rng* to the point associated with the second column. *rng* is in *units* (unless a *ellipsoid* is input). When no *rng* is provided, or *rng* is empty, a *complete* track is returned.

```
[latrkr, lontrk] =  
track1(track, lat, lon, az, rng, ellipsoid, units) specifies the  
elliptical definition of the Earth with a two-element ellipsoid model  
vector ellipsoid. The default ellipsoid is a spherical Earth, which is  
sufficient for most applications. If used, the units of the semimajor  
axis of the ellipsoid vector define the units for the rng input,  
overriding units for this purpose.
```

```
[latrkr, lontrk] =  
track1(track, lat, lon, az, rng, ellipsoid, units, npts) specifies the  
number of points, npts, per output track. npts is 100 by default.
```

```
pts = track1(lat, lon, az, ...) combines the outputs into a single  
two-column matrix, pts.
```

## Background

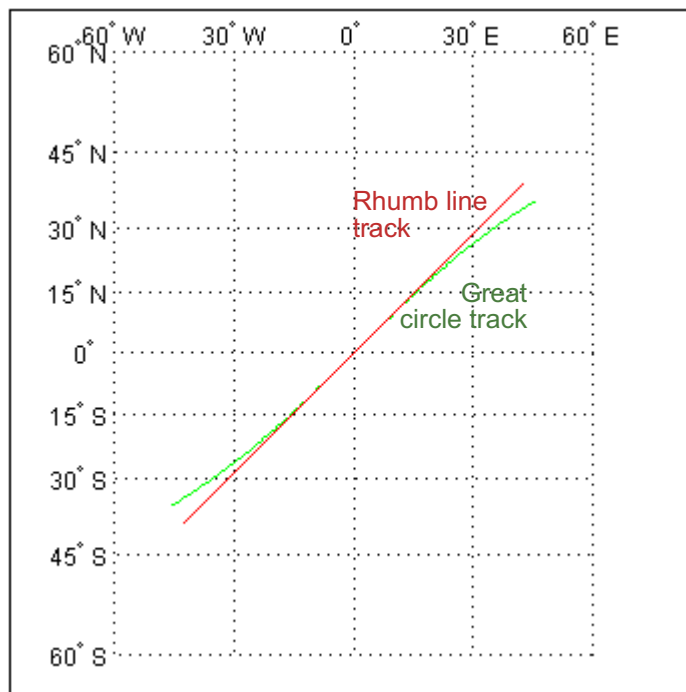
A path along the surface of the Earth connecting two points is a *track*. Two types of track lines are of interest geographically, great circles and rhumb lines. Great circles represent the shortest possible path between two points. Rhumb lines are paths with constant angular headings. They are not, in general, the shortest path between two points.

Full great circles bisect the Earth; the *ends* of the track meet to form a complete circle. Rhumb lines with true east or west azimuths are parallels; the ends also meet to form a complete circle. All other rhumb lines terminate at the poles; their ends do not meet.

## Examples

```
axesm('mercator', 'MapLatLimit', [-60 60], 'MapLonLimit', [-60 60])  
[latrkgc, lontrkgc] = track1(0, 0, 45, [-55 55]);  
plotm(latrkgc, lontrkgc, 'g')  
[latrkrh, lontrkrh] = track1('rh', 0, 0, 45, [-55 55]);  
plotm(latrkrh, lontrkrh, 'r')
```



**See Also**

azimuth, distance, reckon, scircle1, scircle2, track, track2, trackg

## Purpose

Geographic tracks from starting and ending points

## Syntax

```
[latrkr,lonrkr] = track2(lat1,lon1,lat2,lon2)
[latrkr,lonrkr] = track2(track,lat1,lon1,lat2,lon2)
[latrkr,lonrkr] = track2(track,lat1,lon1,lat2,lon2,units)
[latrkr,lonrkr] =
track2(track,lat1,lon1,lat2,lon2,ellipsoid,
        units)
[latrkr,lonrkr] =
track2(lat1,lon1,lat2,lon2,ellipsoid,units,
        npts)
pts = track2(lat1,lon1,lat2,lon2,...)
```

[latrkr,lonrkr] = track2(lat1,lon1,lat2,lon2) returns, in latrkr and lonrkr, points along a (great circle) track between the points specified by lat1 with lon1 and lat2 and lon2. When the inputs are column vectors, the successive tracks are stored in separate columns of latrkr and lonrkr.

[latrkr,lonrkr] = track2(track,lat1,lon1,lat2,lon2) allows the specification of the track logic to be employed. A string *track* of 'gc' is the default, resulting in a great circle track. A *track* of 'rh' results in a rhumb line track.

[latrkr,lonrkr] = track2(track,lat1,lon1,lat2,lon2,units) specifies the units of the inputs and outputs, where *units* is any valid angle unit string. The default is 'degrees'.

[latrkr,lonrkr] = track2(track,lat1,lon1,lat2,lon2,ellipsoid,units) specifies the elliptical definition of the Earth with a two-element ellipsoid model vector ellipsoid. The default ellipsoid is a spherical Earth, which is sufficient for most applications.

[latrkr,lonrkr] = track2(lat1,lon1,lat2,lon2,ellipsoid,units,npts) specifies the number of points, npts, per output track. npts is 100 by default.

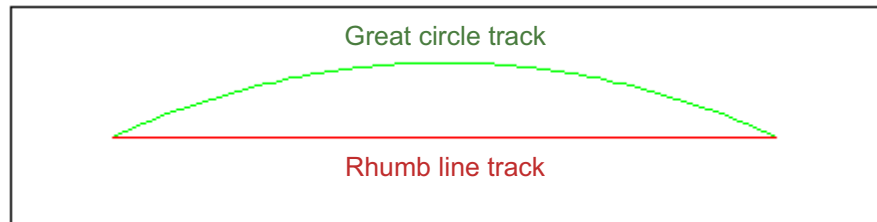
pts = track2(lat1,lon1,lat2,lon2,...) combines the outputs into a single two-column matrix, pts.

## Background

A path along the surface of the Earth connecting two points is a *track*. Two types of track lines are of interest geographically, great circles and rhumb lines. Great circles represent the shortest possible path between two points. Rhumb lines are paths with constant angular headings. They are not, in general, the shortest path between two points.

## Example

```
axesm('mercator','MapLatLimit',[30 50],'MapLonLimit',[-40 40])  
[lattrkgc,lontrkgc] = track2(40,-35,40,35);  
[lattrkrh,lontrkrh] = track2('rh',40,-35,40,35);  
plotm(lattrkgc,lontrkgc,'g')  
plotm(lattrkrh,lontrkrh,'r')
```



## See Also

azimuth, distance, reckon, scircle1, scircle2, track, track1, trackg

## Purpose

Great circle or rhumb line defined via mouse input

## Syntax

```
h = trackg(ntrax)
h = trackg(ntrax,npts)
h = trackg(ntrax,linestyle)
h = trackg(ntrax,PropertyName,PropertyValue,...)
[lat,lon] = trackg(ntrax,npts,...)
h = trackg(track,ntrax,...)
```

`h = trackg(ntrax)` brings forward the current map axes and waits for the user to make  $(2 \times ntrax)$  mouse clicks. The output `h` is a vector of handles for the `ntrax` track segments, which are then displayed.

`h = trackg(ntrax,npts)` specifies the number of plotting points to be used for each track segment. `npts` is 100 by default.

`h = trackg(ntrax,linestyle)` specifies the line style for the displayed track segments, where *linestyle* is any line style string recognized by the standard MATLAB function `line`.

`h = trackg(ntrax,PropertyName,PropertyValue,...)` allows property name/property value pairs to be set, where *PropertyName* and *PropertyValue* are recognized by the `line` function.

`[lat,lon] = trackg(ntrax,npts,...)` returns the coordinates of the plotted points rather than the handles of the track segments. Successive segments are stored in separate columns of `lat` and `lon`.

`h = trackg(track,ntrax,...)` specifies the logic with which tracks are calculated. If the string *track* is 'gc' (the default), a great circle path is used. If *track* is 'rh', rhumb line logic is used.

## Description

This function is used to define great circles or rhumb lines for display using mouse clicks. For each track, two clicks are required, one for each endpoint of the desired track segment. You can modify the track after creation by **Shift**+clicking it. The track is then in edit mode, during which you can change the length and position by dragging control points, or by entering values into a control panel. **Shift**+clicking again exits edit mode.

**See Also**      track1, track2, scircleg

# trimcart

---

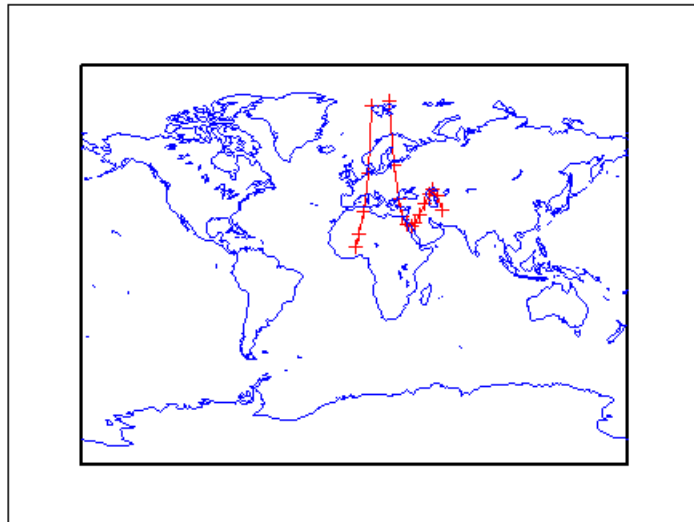
**Purpose** Trim graphic objects to map frame

**Syntax** `trimcart(h)`

`trimcart(h)` clips the graphic objects to the map frame. `h` can be a handle or a vector of handles to graphics objects. `h` can also be any object name recognized by `handlem`. `trimcart` clips lines, surfaces, and text objects.

**Examples**

```
figure; axesm('miller')
framem
[x, y] = humps(0:.05:1);
h = plot(x, y/25, 'r+-');
load coast
geoshow(lat, long)
trimcart(h)
```



**Limitations** `trimcart` does not trim patch objects.

**See Also**      `handlem, makemapped`

# trimdata

---

**Purpose** Trim map data exceeding projection limits

**Syntax** `[ymat,xmat,trimpts] = trimdata(ymat,ylim,xmat,xlim,'object')`

`[ymat,xmat,trimpts] = trimdata(ymat,ylim,xmat,xlim,'object')` identifies points in map data that exceed projection limits. The projection limits are defined by the lower and upper inputs. The particular object to be trimmed is identified by the 'object' input.

Allowable object strings are

- 'surface' for trimming graticules
- 'light' for trimming lights,
- 'line' for trimming lines
- 'patch' for trimming patches
- 'text' for trimming text object location points
- 'none' to skip all trimming operations

**See Also** `clipdata`, `undotrim`, `undoclip`



**Purpose**

Remove object clips introduced by clipdata

**Syntax**

```
[lat,long] = undoclip(lat,long,clippts,'object')
```

[lat,long] = undoclip(lat,long,clippts,'object') removes the object clips introduced by clipdata. This function is necessary to properly invert projected data from the Cartesian space to the original latitude and longitude data points.

The input variable, clippts, must be constructed by the function clipdata.

**Description**

Allowable object strings are

- 'surface' for trimming graticules
- 'light' for trimming lights
- 'line' for trimming lines
- 'patch' for trimming patches
- 'text' for trimming text object location points
- 'none' to skip all trimming operations

**See Also**

clipdata, trimdata, undotrim

# undotrim

---

**Purpose** Remove object trims introduced by trimdata

**Syntax** `[ymat,xmat] = undotrim(ymat,xmat,trimpts,'object')`  
`[ymat,xmat] = undotrim(ymat,xmat,trimpts,'object')` removes the object trims introduced by trimdata. This function is necessary to properly invert projected data from the Cartesian space to the original latitude and longitude data points.

The input variable, trimpts, must be constructed by the function trimdata.

**Description** Allowable object strings are

- 'surface' for trimming graticules
- 'light' for trimming lights
- 'line' for trimming lines
- 'patch' for trimming patches
- 'text' for trimming text object location points
- 'none' to skip all trimming operations

**See Also** clipdata, trimdata, undoclip

**Purpose** Unit conversion factors

**Syntax** `ratio = unitsratio(to, from)`  
`ratio = unitsratio(to, from)` returns the number of to units per one from unit. For example, `unitsratio('cm', 'm')` returns 100 because there are 100 centimeters per meter. `unitsratio` makes it easy to convert from one system of units to another. Specifically, if `x` is in units from and

$$y = \text{unitsratio}(\text{to}, \text{from}) * x$$

then `Y` is in units `to`.

`to` and `from` can be any strings from the second column of one of the following tables (both must come from the same table). `to` and `from` are case insensitive and can be either singular or plural.

## Units of Length

`unitsratio` recognizes the following identifiers for converting units of length:

Unit Name	String(s)
Meter	'm', 'meter(s)', 'metre(s)'
Centimeter	'cm', 'centimeter(s)', 'centimetre(s)'
Millimeter	'mm', 'millimeter(s)', 'millimetre(s)'
Micron	'micron(s)'
Kilometer	'km', 'kilometer(s)', 'kilometre(s)'
Nautical mile	'nm', 'nautical mile(s)'
International foot	'ft', 'international ft', 'foot', 'international foot', 'feet', 'international feet'
Inch	'in', 'inch', 'inches'
Yard	'yd', 'yard(s)'

# unitsratio

---

Unit Name	String(s)
international mile	'mi', 'mile(s)', 'international mile(s)'
U.S. survey foot	'sf', 'survey ft', 'U.S. survey ft', 'survey foot', 'U.S. survey foot', 'survey feet', 'U.S. survey feet'
U.S. survey mile (statute mile)	'sm', 'survey mile(s)', 'statute mile(s)', 'U.S. survey mile(s)'

## Units of Angle

unitsratio recognizes the following identifiers for converting units of angle:

Unit Name	String(s)
radian	'rad', 'radian(s)'
degree	'deg', 'degree(s)'

## Examples

```
% Approximate mean earth radius in meters
radiusInMeters = 6371000
% Conversion factor
feetPerMeter = unitsratio('feet', 'meter')
% Radius in (international) feet:
radiusInFeet = feetPerMeter * radiusInMeters
% The following prints a true statement for valid TO, FROM pairs:
to = 'feet';
from = 'mile';
sprintf('There are %g %s per %s.', unitsratio(to,from), to, from)
% The following prints a true statement for valid TO, FROM pairs:
to = 'degrees';
from = 'radian';
sprintf('One %s is %g %s.', from, unitsratio(to,from), to)
```

**Purpose** Check spatiotemporal unit strings and abbreviations

---

**Note** The unitstr function is obsolete and will be removed in a future release. The syntax `str = unitstr(str, 'times')` has already been removed.

---

## Syntax

```
unitstr
str = unitstr(str0, 'angles')
str = unitstr(str0, 'distances')
```

unitstr, with no arguments, displays a list of strings and abbreviations, recognized by certain Mapping Toolbox functions, for units of angle and length/distance.

`str = unitstr(str0, 'angles')` checks for valid angle unit strings or abbreviations. If a valid string or abbreviation is found, it is converted to a standardized, preset string. 'angles' can be abbreviated.

`str = unitstr(str0, 'distances')` checks for valid length unit strings or abbreviations. If a valid string or abbreviation is found, it is converted to a standardized, preset string. 'distances' can be abbreviated. Note that input strings 'miles' and 'mi' are converted to 'statutemiles'; there is no way to specify international miles in the unitstr function.

## Examples

This function recognizes and standardizes certain abbreviations:

```
str = unitstr('sm', 'distances')

str =
statutemiles
```

And any unique truncation:

```
str = unitstr('ra', 'angles')

str =
```

# unitstr

---

radians

## **See Also**

unitsratio

**Purpose** Unwrap vector of angles with NaN-delimited parts

**Syntax** `unwrapped = unwrapMultipart(p)`

`unwrapped = unwrapMultipart(p)` unwraps a row or column vector of azimuths, longitudes, or phase angles. Input and output units are both radians. If `p` is separated into multiple parts delimited by values of NaN, each part is unwrapped independently. If `p` has only one part, the result is equivalent to `unwrap(p)`. The output is the same size as the input and has NaNs in the same locations.

## Examples **Example 1**

Compare the behavior `unwrapMultipart` to that of `unwrap`. The output of `unwrapMultipart` starts over again at 6.11 following the NaN, unlike the output of `unwrap`. The output of `unwrapMultipart` is equivalent to a concatenation (with NaN-separator) of separate calls to `unwrap`:

```
p1 = [0.17      5.67      4.89      4.10];
p2 = [6.11      1.05      2.27];
unwrap([p1 NaN p2])

ans =
    0.1700   -0.6132   -1.3932   -2.1832     NaN   -0.1732    1.0500    2.2700

unwrapMultipart([p1 NaN p2])

ans =
    0.1700   -0.6132   -1.3932   -2.1832     NaN    6.1100    7.3332    8.5532

[unwrap(p1) NaN unwrap(p2)]

ans =
    0.1700   -0.6132   -1.3932   -2.1832     NaN    6.1100    7.3332    8.5532
```

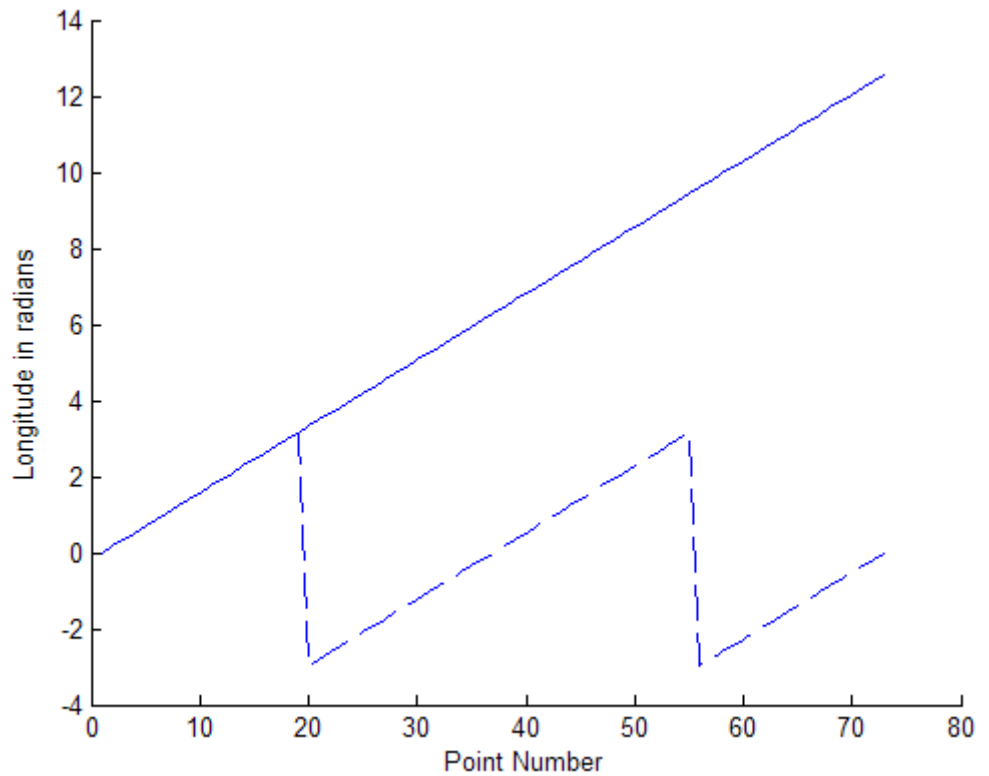
# unwrapMultipart

---

## Example 2

Wrap two revolutions of a sphere to  $\pi$  with `wrapToPi`, and then unwrap it with `unwrapMultipart`:

```
lon = wrapToPi(deg2rad(0:10:720));  
unwrappedlon = unwrapMultipart(lon);  
figure; hold on  
plot(lon, '--')  
plot(unwrappedlon)  
xlabel 'Point Number'  
ylabel 'Longitude in radians'
```





**See Also**

unwrap, wrapTo180, wrapTo360, wrapToPi, wrapTo2Pi

# updategeostruct

---

## Purpose

Convert geographic data structure from Version 1 to Version 2

## Syntax

```
g2 = updategeostruct(g)
s = updategeostruct(g, str)
[s,symbolspec] = updategeostruct(g, ...)
[s,symbolspec] = updategeostruct(g, ..., cmap)
```

`g2 = updategeostruct(g)` accepts a geographic data structure `g`. If `g` is a `geostruct1` for which the 'type' field has value 'line' or 'patch', `updategeostruct` restructures its elements to create a `geostruct2`, `g2`. If `g` is a `geostruct2`, it is copied unaltered to `g2`. `updategeostruct` should not be used for `geostruct1` arrays of type 'text', 'light', 'regular', or 'surface'.

`s = updategeostruct(g, str)` selects only elements whose tag field begins with the string `str` (and whose type field is either 'line' or 'patch'). The selection is case insensitive.

`[s,symbolspec] = updategeostruct(g, ...)` restructures a geographic data structure and determines a `symbolspec` based on the graphic properties specified in the `otherproperty` field for each element of `g` and, if necessary, the `jet` colormap.

`[s,symbolspec] = updategeostruct(g, ..., cmap)` specifies a colormap, `cmap`, to define the colors used in `symbolspec`.

## Remarks

Mapping Toolbox supports two ways of encoding vector features in MATLAB structure arrays. In both cases there is one feature per array element, and in both cases the array elements are called “geographic data structures.” Mapping Toolbox Version 1.3.1 and earlier supported the “Version 1” geographic data structure (called *geostruct1*), in which

- A tag field names an individual feature or object.
- A type field specifies a MATLAB graphics object type ('line', 'patch', 'surface', 'text', or 'light') or has the value 'regular', specifying a regular data grid.
- All coordinates are in latitude-longitude, stored in fields `lat` and `long`.

- An altitude coordinate array extends coordinates to 3-D.
- A string property contains text to be displayed if type is 'text'.
- MATLAB graphics properties are specified explicitly, on a per-feature basis, in an otherproperty field.

The choice of options for the type field reveals that geostruct1 can contain

- Vector geodata (type is 'line' or 'patch')
- Raster geodata (type is 'surface' or 'regular')
- Graphic objects (type is 'text' or 'light')

Beginning with Mapping Toolbox 2.0, geographic data structures can take a more general form (geostruct2)—but only for vector geodata:

- Coordinates can be in either *latitude-longitude* (stored in fields Lat and Lon) or map *x-y* (stored in fields X and Y).
- An optional field, Height or Z, extends coordinates to 3-D.
- A Geometry text field designates the geometric nature of the feature: 'Point', 'Multipoint', 'Line', or 'Polygon' rather than a graphics object type.
- Additional attribute fields, the names and number of which are data-set-specific, describe the nongeometric properties (name, ownership, age, code or identifier, ...).

This is the form of geostruct that shaperead outputs. The Version 2 geographic data structures allow for a greater amount of information to be carried about each vector feature. They also separate the graphics display properties from the fundamental properties of the geographic features themselves.

Instead of being assigned in advance, graphics properties are determined at display time by matching up attribute values against rules provided in a symbolspec. For example, a road class attribute can

# updategeostruct

---

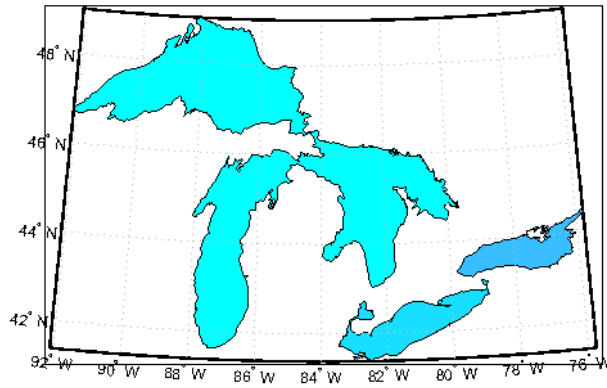
be used to display major highways with a distinctive color and greater line width than secondary roads. The same geographic data structure can be displayed in many different ways, without altering any of its contents, and shapefile data imported from external sources need not be altered to control its graphic display.

Some Version 2 toolbox functions (for example, `mapshow`, `geoshow`, and `mapview`) accept either type of geographic data structure. Other (older) functions (for example, `displaym` and `extractm`) accept only Version 1 geographic data structures. The purpose of `updategeostruct`, which supports the implementation of `mapshow` and `geoshow`, is to restructure Version 1 geographic data structures containing vector geodata, converting them to the newer form.

## Example

Update and display the Great Lakes version 1 geostruct:

```
load greatlakes
cmap = cool(3*numel(greatlakes));
[gtlakes, spec] = updategeostruct(greatlakes, cmap);
lat = extractfield(gtlakes, 'Lat');
lon = extractfield(gtlakes, 'Lon');
lonlim = [min(lon) max(lon)];
latlim = [min(lat) max(lat)];
figure
usamap(latlim, lonlim);
geoshow(gtlakes, 'SymbolSpec', spec)
```



**See Also**

geoshow, makesymbolspec, mapshow, mapview, shaperead

# usamap

---

**Purpose** Construct map axes for United States of America

**Syntax**

```
usamap state
usamap(state)
usamap 'conus'
usamap('conus')
usamap
usamap(latlim, lonlim)
usamap(Z, refvec)
h = usamap(...)
h = usamap('all')
h = usamap('allequal')
```

`usamap state` or `usamap(state)` constructs an empty map axes with a Lambert Conformal Conic projection and map limits covering a U.S. state or group of states specified by input `state`. `state` may be a string or a cell array of strings, where each string contains the name of a state or 'District of Columbia'. Alternatively, `state` may be a standard two-letter U.S. Postal Service abbreviation. The map axes is created in the current axes and the axis limits are set tight around the map frame.

`usamap 'conus'` or `usamap('conus')` constructs an empty map axes for the conterminous 48 states (i.e. excluding Alaska and Hawaii).

`usamap` with no arguments asks you to choose from a menu of state names plus 'District of Columbia', 'conus', 'all', and 'allequal'.

`usamap(latlim, lonlim)` constructs an empty Lambert Conformal map axes for a region of the U.S. defined by its latitude and longitude limits in degrees. `latlim` and `lonlim` are two-element vectors of the form `[southern_limit northern_limit]` and `[western_limit eastern_limit]`, respectively.

`usamap(Z, refvec)` derives the map limits from the extent of a regular data grid with 1-by-3 referencing vector `refvec`.

`h = usamap(...)` returns the handle of the map axes.

`h = usamap('all')` constructs three empty axes, inset within a single figure, for the conterminous states, Alaska, and Hawaii, respectively, using projection parameters suggested by the U.S. Geological Survey. The handles for the three map axes are returned in `h`. `h(1)` is for the conterminous states, `h(2)` is for Alaska, and `h(3)` is for Hawaii.

`h = usamap('allequal')` constructs the map axes with Alaska and Hawaii at the same scale as the conterminous states.

## Remarks

`usamap` uses `tightmap` set the axis limits tight around the map. If you change the projection, or just want more white space around the map frame, use `tightmap` again or `axis auto`.

`axes(h(n))`, where `n = 1, 2, or 3`, makes the desired axes current.

`set(h, 'Visible', 'on')` makes the axes visible.

`set(h, 'ButtonDownFcn', 'selectmoveresize')` allows interactive repositioning of the axes. `set(h, 'ButtonDownFcn', 'uimaptbx')` restores the Mapping Toolbox interfaces.

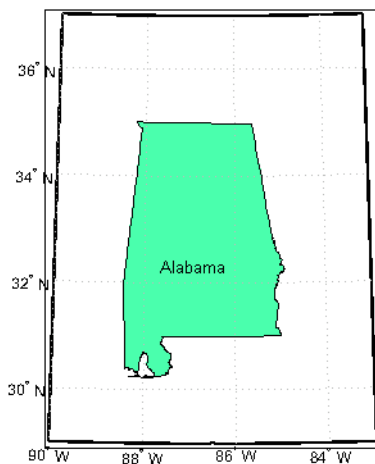
`axesscale(h(1))` resizes the axes containing Alaska and Hawaii to the same scale as the conterminous states.

## Examples

### Example 1

Make a map of Alabama only:

```
usamap('Alabama')
alabamahi = shaperead('usastatehi', 'UseGeoCoords', true,...
    'Selector',{@(name) strcmpi(name,'Alabama'), 'Name'});
geoshow(alabamahi, 'FaceColor', [0.3 1.0, 0.675])
textm(alabamahi.LabelLat, alabamahi.LabelLon, alabamahi.Name,...
    'HorizontalAlignment', 'center')
```

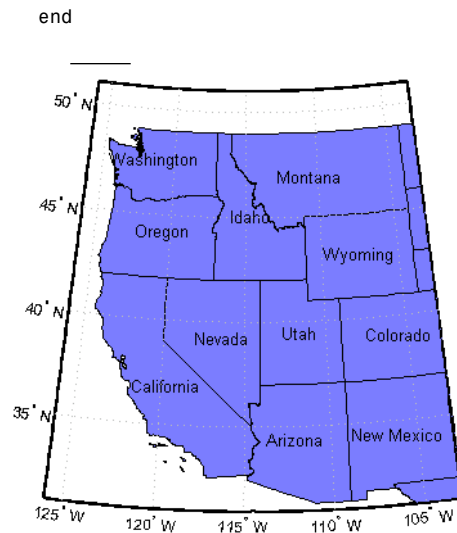


## Example 2

Map a region extending from California to Montana:

```
figure; ax = usamap({'CA','MT'});
set(ax, 'Visible', 'off')
latlim = getm(ax, 'MapLatLimit');
lonlim = getm(ax, 'MapLonLimit');
states = shaperead('usastatehi',...
    'UseGeoCoords', true, 'BoundingBox', [lonlim', latlim']);
geoshow(ax, states, 'FaceColor', [0.5 0.5 1])
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
```

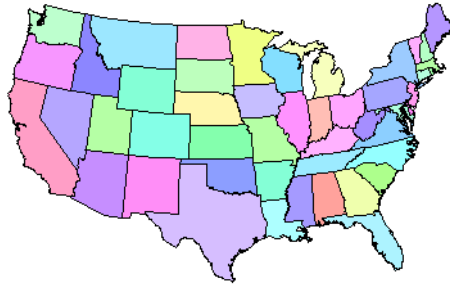




### Example 3

Map the Conterminous United States with a different fill color for each state:

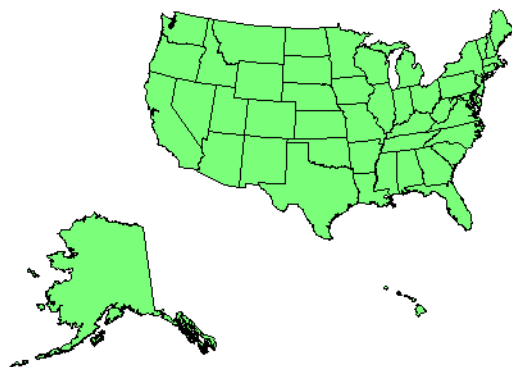
```
figure; ax = usamap('conus');
states = shaperead('usastatelo', 'UseGeoCoords', true, ...
    'Selector', ...
    {@(name) ~any(strcmp(name,{'Alaska','Hawaii'})), 'Name'});
faceColors = makesymbolspec('Polygon', ...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))}); %NOTE - colors are random
geoshow(ax, states, 'DisplayType', 'polygon', ...
    'SymbolSpec', faceColors)
framem off; gridm off; mlabel off; plabel off
```



## Example 4

Map of the USA with separate axes for Alaska and Hawaii:

```
figure; ax = usamap('allequal');
set(ax, 'Visible', 'off')
states = shaperead('usastatelo', 'UseGeoCoords', true);
names = {states.Name};
indexHawaii = strmatch('Hawaii',names);
indexAlaska = strmatch('Alaska',names);
indexConus = 1:numel(states);
indexConus(indexHawaii) = [];
indexConus(indexAlaska) = [];
stateColor = [0.5 1 0.5];
geoshow(ax(1), states(indexConus), 'FaceColor', stateColor)
geoshow(ax(2), states(indexAlaska), 'FaceColor', stateColor)
geoshow(ax(3), states(indexHawaii), 'FaceColor', stateColor)
for k = 1:3
    setm(ax(k), 'Frame', 'off', 'Grid', 'off',...
        'ParallelLabel', 'off', 'MeridianLabel', 'off')
end
```



**See also**

`axesm`, `axesscale`, `geoshow`, `paperscale`, `selectmoversize`,  
`tightmap`, `worldmap`

# usgs24kdem

---

## Purpose

Read USGS 7.5-minute (30-m or 10-m) Digital Elevation Models

## Syntax

```
[lat,lon,Z] = usgs24kdem
[lat,lon,Z] = usgs24kdem(filename)
[lat,lon,Z] = usgs24kdem(filename,samplefactor)
[lat,lon,Z] =
usgs24kdem(filename,samplefactor,latlim,lonlim)
[lat,lon,Z] = ...usgs24kdem(filename,samplefactor,latlim,
lonlim,gsize)
[lat,lon,Z,header,profile] = usgs24kdem(...)
```

[lat,lon,Z] = usgs24kdem reads a USGS 1:24,000 digital elevation map (DEM) file in standard format. The file is selected interactively. The entire file is read and subsampled by a factor of 5. A geolocated data grid is returned with a latitude array, lat, longitude array, lon, and elevation array, Z. Horizontal units are in degrees, vertical units may vary. The 1:24,000 series of DEMs are stored as a grid of elevations spaced either at 10 or 30 meters apart. The number of points in a file will vary with the geographic location.

[lat,lon,Z] = usgs24kdem(filename) reads the USGS DEM specified by filename and returns the result as a geolocated data grid.

[lat,lon,Z] = usgs24kdem(filename,samplefactor) reads a subset of the DEM data from filename. samplefactor is a scalar integer, which when equal to 1 reads the data at its full resolution. When samplefactor is an integer n greater than one, every nth point is read. If samplefactor is omitted or empty, it defaults to 5.

[lat,lon,Z] = usgs24kdem(filename,samplefactor,latlim,lonlim) reads a subset of the elevation data from filename. The limits of the desired data are specified as two-element vectors of latitude, latlim, and longitude, lonlim, in degrees. The elements of latlim and lonlim must be in ascending order. The data may extend somewhat outside the requested area. If limits are omitted, data for the entire area covered by the DEM file is returned.

```
[lat,lon,Z] =
...usgs24kdem(filename,samplefactor,latlim,lonlim,gsize)
```

specifies the graticule size in `gsize`. `gsize` is a two-element vector specifying the number of rows and columns in the latitude and longitude coordinated grid. If omitted, a graticule the same size as the geolocated data grid is returned. Use empty matrices for `latlim` and `lonlim` to specify the coordinated grid size without specifying the geographic limits.

`[lat, lon, Z, header, profile] = usgs24kdem(...)` also returns the contents of the header and raw profiles of the DEM file. The header structure contains descriptions of the data from the file header. The profile structure is the raw profile data from which the geolocated data grid is constructed.

## Background

The U.S. Geological Survey has created a series of digital elevation models based on their paper 1:24,000 scale maps. The grid spacing for these elevations models is either 10 or 30 meters on a Universal Transverse Mercator grid. Each file covers a 7.5 minute quadrangle. The map and data series are available for much of the conterminous United States, Hawaii, and Puerto Rico. The data has been released in a number of formats. This function reads the data in the “standard” file format.

## Example

Use the archived San Francisco South 24K DEM file `sanfranciscos.dem.gz`, which is provided in the Mapping Toolbox `mapdemos` directory.

- 1 Gunzip the demo file to a temporary directory:

```
filenames = gunzip('sanfranciscos.dem.gz', tempdir);
demFilename = filenames{1};
```

- 2 Read every other point of the 1:24,000 DEM file.

```
[lat, lon,Z,header,profile] = usgs24kdem(demFilename,2);
```

- 3 Delete the temporary gunzipped file.

```
delete(demFilename);
```

- 4** As no negative elevations exist, move all points at sea level to -1 to color them blue:

```
Z(Z==0) = -1;
```

- 5** Compute the latitude and longitude limits for the DEM:

```
latlim = [min(lat(:)) max(lat(:))]
```

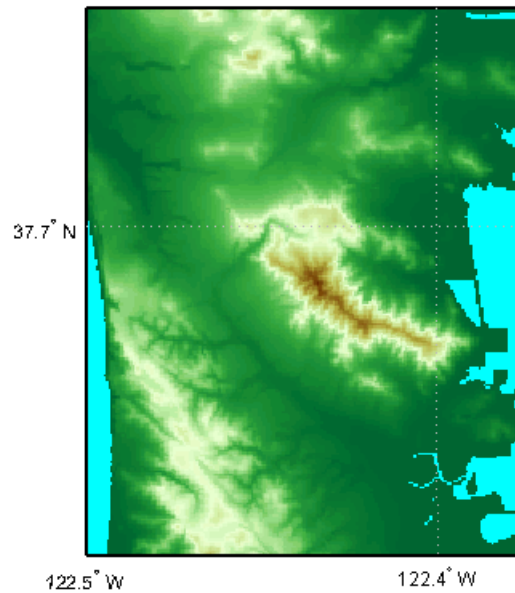
```
latlim =  
    37.6249    37.7504
```

```
lonlim = [min(lon(:)) max(lon(:))]
```

```
lonlim =  
   -122.5008   -122.3740
```

- 6** Display the DEM values:

```
figure  
usamap(latlim, lonlim)  
geoshow(lat, lon, Z, 'DisplayType','surface')  
demcmap(Z)  
daspectm('m',1)
```



### 1 Examine the metadata in the header:

```
header
```

```
header =
```

```
QuadrangleName: 'SAN FRANCISCO SOUTH, CA  
BIG BASIN DEM'  
TextualInfo: 'WMC CTOG'  
Filler: ''  
ProcessCode: ''  
Filler2: ''  
SectionalIndicator: ''  
MCoriginCode: ''  
DEMlevelCode: 2  
ElevationPatternCode: 'regular'  
PlanimetricReferenceSystemCode: 'UTM'
```

```
Zone: 10
ProjectionParameters: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
HorizontalUnits: 'meters'
ElevationUnits: 'feet'
NsidesToBoundingBox: 4
BoundingBox: [1x8 double]
MinMaxElevations: [0 1314]
RotationAngle: 0
AccuracyCode: 'accuracy information in record C'
XYZresolutions: [30 30 1]
NrowsCols: [1 371]
MaxPcontourInt: NaN
SourceMaxCintUnits: NaN
SmallestPrimary: NaN
SourceMinCintUnits: NaN
DataSourceDate: NaN
DataInspRevDate: NaN
InspRevFlag: ''
DataValidationFlag: NaN
SuspectVoidFlag: NaN
VerticalDatum: NaN
HorizontalDatum: NaN
DataEdition: NaN
PercentVoid: NaN
```

## Remarks

This function reads USGS DEM files stored in the UTM projection. The function unprojects the grid back to latitude and longitude. Use `usgsdem` for data stored in geographic grids.

The number of points in a file varies with the geographic location. Unlike the USGS DEM products, which use an equal-angle grid, the UTM projection grid DEMs cannot simply be concatenated to cover larger areas. There can be data gaps between DEMs.

You can obtain the data files from the U.S. Geological Survey and from commercial vendors. Other agencies have made some local area data available online. The DEM files are ASCII files, and can be transferred as text. Line-ending conversion is not necessarily required.



**See Also**

demdataui, dted, gtopo30, tbase, etopo, usgsdem, usgsdems

## Purpose

Read USGS 1-degree (3-arc-second) Digital Elevation Model

## Syntax

```
[Z,refvec] = usgsdem(filename,scalefactor)
```

```
[Z,refvec] = usgsdem(filename,scalefactor,latlim,lonlim)
```

`[Z,refvec] = usgsdem(filename,scalefactor)` reads the specified file and returns the data in a regular data grid along with referencing vector `refvec`, a 1-by-3 vector having elements [cells/degree north-latitude west-longitude] with latitude and longitude limits specified in degrees. The data can be read at full resolution (`scalefactor = 1`), or can be downsampled by the `scalefactor`. A `scalefactor` of 3 returns every third point, giving 1/3 of the full resolution.

`[Z,refvec] = usgsdem(filename,scalefactor,latlim,lonlim)` reads data within the latitude and longitude limits. These limits are two-element vectors with the minimum and maximum values specified in units of degrees.

## Background

The U.S. Geological Survey has made available a set of digital elevation maps of 1-degree quadrangles covering the contiguous United States, Hawaii, and limited portions of Alaska. The data is on a regular grid with a spacing of 30 arc-seconds (or about 100-meter resolution). 1-degree DEMs are also referred to as *3-arc-second* or *1:250,000 scale* DEM data.

The data is derived from the U.S. Defense Mapping Agency's DTED-1 digital elevation model, which itself was derived from cartographic and photographic sources. The cartographic sources were maps from the 7.5-minute through 1-degree series (1:24,000 scale through 1:250,000 scale).

## Remarks

The grid for the digital elevation maps is based on the 1984 World Geodetic System (WGS84). Older DEMs were based on WGS72. Elevations are in meters relative to National Geodetic Vertical Datum of 1929 (NGVD 29) in the continental U.S. and local mean sea level in Hawaii.

The absolute horizontal accuracy of the DEMs is 130 meters, while the absolute vertical accuracy is  $\pm 30$  meters. The relative horizontal and vertical accuracy is not specified, but is probably much better than the absolute accuracy.

These DEMs have a grid spacing of 3 arc-seconds in both the latitude and longitude directions. The exception is DEM data in Alaska, where latitudes between 50 and 70 degrees North have grid spacings of 6 arc-seconds, and latitudes greater than 70 degrees North have grid spacings of 9 arc-seconds.

Statistical data in the files is not returned.

You can obtain the data files from the U.S. Geological Survey and from commercial vendors. Other agencies have made some local area data available online.

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

Read every fifth point in the file containing part of Rhode Island and Cape Cod:

```
[Z,refvec] = usgsdem('providence-e',5);
```

Read the elevation data for Martha's Vineyard at full resolution:

```
[Z,refvec] = usgsdem('providence-e',1,...
[41.2952 41.4826],[-70.8429 -70.4392]);
whos Z
```

Name	Size	Bytes	Class
Z	226x485	876880	double array

## See Also

usgs24kdem, gtopo30, etopo, tbase, usgsdems

# usgsdems

---

<b>Purpose</b>	USGS 1-degree (3-arc-sec) DEM filenames for latitude-longitude quadrangle
<b>Syntax</b>	<pre>[fname,qname] = usgsdems(latlim,lonlim)</pre> <p>[fname,qname] = usgsdems(latlim,lonlim) returns cell arrays of the DEM filenames and quadrangle names covering the geographic region. The region is specified by scalar latitude and longitude points or two-element vectors of latitude and longitude limits in units of degrees.</p>
<b>Background</b>	The U.S. Geological Survey has made available a set of digital elevation maps of 1-degree quadrangles covering the contiguous United States, Hawaii, and limited portions of Alaska. These are referred to as <i>1-degree, 3-arc second</i> or <i>1:250,000 scale</i> DEMs. Because the filenames of these 1 degree data sets are taken from the names of cities or features in the quadrangle, determining the files needed to cover a particular region generally requires consulting an index map or other reference. This function takes the place of such a reference by returning the filenames for a given geographic region.
<b>Remarks</b>	This function only returns filenames for the contiguous United States.
<b>Examples</b>	Which files are needed to map part of New England? <pre>usgsdems([41 44], [-72 -69])</pre> <pre>ans =     'providence-w'     'providence-e'     'chatham-w'     'boston-w'     'boston-e'     'portland-w'     'portland-e'     'bath-w'</pre>
<b>See Also</b>	usgsdem

**Purpose**

Select ellipsoids for given UTM zone

**Syntax**

```
ellipsoid = utmgeoid,  
ellipsoid = utmgeoid(zone)  
[ellipsoid,ellipsoidstr] = utmgeoid(...)
```

`ellipsoid = utmgeoid`, without any arguments, opens the `utmzoneui` interface for selecting a UTM zone. This zone is then used to return the recommended ellipsoid definitions for that particular zone.

`ellipsoid = utmgeoid(zone)` uses the input `zone` to return the recommended ellipsoid definitions.

`[ellipsoid,ellipsoidstr] = utmgeoid(...)` returns the ellipsoid string used by the `almanac` function.

**Background**

The Universal Transverse Mercator (UTM) system of projections tiles the world into quadrangles called zones. Each zone has different projection parameters and commonly used ellipsoidal models of the Earth. This function returns a list of ellipsoid models commonly used in a zone.

**Examples**

```
zone = utmzone(0,100) % degrees  
  
zone =  
47N  
  
[ellipsoid,names] = utmgeoid(zone)  
  
ellipsoid =  
    6377.3    0.081473  
    6377.4    0.081697  
names =  
everest  
bessel
```

**See Also**

`utmzone`

# utmzone

---

**Purpose** Select UTM zone given latitude and longitude

**Syntax**

```
zone = utmzone
zone = utmzone(lat, long)
zone = utmzone(mat),
[latlim, lonlim] = utmzone(zone),
lim = utmzone(zone)
```

`zone = utmzone` selects a Universal Transverse Mercator (UTM) zone with a graphical user interface. The zone designation is returned as a string.

`zone = utmzone(lat, long)` returns the UTM zone containing the geographic coordinates. If `lat` and `long` are vectors, the zone containing the geographic mean of the data set is returned. The geographic coordinates must be in units of degrees.

`zone = utmzone(mat)`, where `mat` is of the form `[lat long]`.

`[latlim, lonlim] = utmzone(zone)`, where `zone` is a valid UTM zone designation, returns the geographic limits of the zone. Valid UTM zones designations are numbers, or numbers followed by a single letter. For example, '31' or '31N'. The returned limits are in units of degrees.

`lim = utmzone(zone)` returns the limits in a single vector output.

**Background** The Universal Transverse Mercator (UTM) system of projections tiles the world into quadrangles called zones. This function can be used to identify which zone is used for a geographic area and, conversely, what geographic limits apply to a UTM zone.

**Examples**

```
[latlim, lonlim] = utmzone('12F')

latlim =
    -56    -48
lonlim =
    -114   -108

utmzone(latlim, lonlim)
```

ans =  
12F

**Limitations**

The UTM zone system is based on a regular division of the globe, with the exception of a few zones in northern Europe. `utmzone` does not account for these deviations.

**See Also**

`utmgeoid`

## Purpose

Convert latitude-longitude vectors to regular data grid

## Syntax

```
[Z, refvec] = vec2mtx(lat, lon, density)
[Z, refvec] = vec2mtx(lat, lon, density, latlim, lonlim)
[Z, refvec] = vec2mtx(lat, lon, Z1, refvec1)
[Z, refvec] = vec2mtx(..., 'filled'),
```

`[Z, refvec] = vec2mtx(lat, lon, density)` creates a regular data grid from vector data, placing ones in grid cells intersected by a vector and zeroes elsewhere. `refvec` is the three-element referencing vector geolocating the computed grid. `lat` and `lon` are vectors of equal length containing geographic locations in units of degrees. `density` indicates the number of grid cells per unit of latitude and longitude (a value of 10 indicates 10 cells per degree, for example), and must be scalar-valued.

`[Z, refvec] = vec2mtx(lat, lon, density, latlim, lonlim)` uses the two-element vectors `latlim` and `lonlim` to define the latitude and longitude limits of the grid. If omitted, the limits are computed automatically.

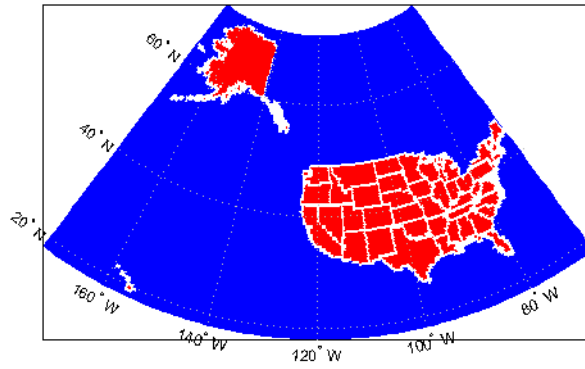
`[Z, refvec] = vec2mtx(lat, lon, Z1, refvec1)` uses a preexisting data grid (`Z1` with referencing vector `refvec1`) to define the limits and density of the output grid.

`[Z, refvec] = vec2mtx(..., 'filled')`, where `lat` and `lon` form one or more closed polygons (with NaN-separators), fills the area outside the polygons with the value two instead of the value zero.

## Example

```
states = shaperead('usastatelo', 'UseGeoCoords', true);
lat = [states.Lat];
lon = [states.Lon];
[Z, refvec] = vec2mtx(lat, lon, 5, 'filled');
figure; worldmap(Z, refvec);
meshm(Z, refvec)
colormap(flag(3))
```



**Limitations**

The *vec2mtx* function does not fill properly if the vector data extends beyond a pole.

**See Also**

`1t1n2val`, `imbedm`, `encodem`, `interp`

**Purpose** Direction angle in map plane from azimuth on ellipsoid

**Syntax**

```
th = vfdtran(lat,lon,az)
th = vfdtran(mstruct,lat,lon,az)
[th,len] = vfdtran(...)
```

`th = vfdtran(lat,lon,az)` transforms the azimuth angle at specified latitude and longitude points on the sphere into the projection space. The map projection currently displayed is used to define the projection space. The input angles must be in the same units as specified by the current map projection. The inputs can be scalars or matrices of the equal size. The angle in the projection space is defined as positive counterclockwise from the  $x$ -axis.

`th = vfdtran(mstruct,lat,lon,az)` uses the map projection defined by the input `mstruct` to compute the map projection.

`[th,len] = vfdtran(...)` also returns the vector length in the projected coordinate system. A value of 1 indicates no scale distortion.

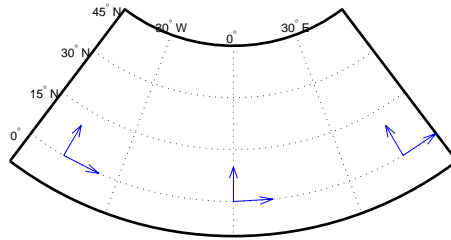
**Background**

The direction of north is easy to define on the three-dimensional sphere, but more difficult on a two-dimensional map. For cylindrical projections in the normal aspect, north is always in the positive  $y$ -direction. For conic projections, north can be to the left or right of the  $y$ -axis. This function transforms any azimuth angle on the sphere to the corresponding angle in the projected paper coordinates.

**Examples**

Sample calculations:

```
axesm('eqdconic','maplatlim',[-10 45],'maplonlim',[-55 55])
gridm; framem; mlabel; plabel
quiverm([0 0 0],[-45 0 45],[0 0 0],[10 10 10],0)
quiverm([0 0 0],[-45 0 45],[10 10 10],[0 0 0],0)
```



```
vfwdtran([0 0 0],[-45 0 45],[0 0 0])
```

```
ans =
    59.614         90    120.39
```

```
vfwdtran([0 0 0],[-45 0 45],[90 90 90])
```

```
ans =
   -30.385    0.0001931    30.386
```

### Limitations

This transformation is limited to the region specified by the frame limits in the current map definition.

### Remarks

The geographic azimuth angle is measured clockwise from north. The projection space angle is measured counterclockwise from the  $x$ -axis.

This function uses a finite difference technique. The geographic coordinates are perturbed slightly in different directions and projected.

# vfwdtran

---

A small amount of error is introduced by numerical computation of derivatives and the variation of map distortion parameters.

## **See Also**

`vinvtran`, `mfwdtran`, `minvtran`, `defaultm`

**Purpose**

Areas visible from point on terrain elevation grid

**Syntax**

```
[visZ,visrefvec] = viewshed(Z,refvec,lat1,lon1)
viewshed(Z,refvec,lat1,lon1,oalt)
viewshed(Z,refvec,lat1,lon1,oalt,talt)
viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt)
viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt,taltopt)
viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt,taltopt,
... actualradius)
viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt,taltopt,
... actualradius,effectiveradius)
```

`[visZ,visrefvec] = viewshed(Z,refvec,lat1,lon1)` computes areas visible from a point on a digital elevation grid. `Z` is a regular data grid containing elevations in units of meters. `refvec` is a three-element referencing vector that geolocates the input data grid. The observer location is provided as scalar latitude and longitude in units of degrees. The resulting `visZ` contains 1s at the surface locations visible from the observer location, and 0s where the line of sight is obscured by terrain.

`viewshed(Z,refvec,lat1,lon1,oalt)` places the observer at the specified altitude in meters above the surface. This is equivalent to putting the observer on a tower. If omitted, the observer is assumed to be on the surface.

`viewshed(Z,refvec,lat1,lon1,oalt,talt)` checks for visibility of target points a specified distance above the terrain. This is equivalent to putting the target points on towers that do not obstruct the view. If omitted, the target points are assumed to be on the surface.

`viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt)` controls whether the observer is at a relative or absolute altitude. If the observer altitude option is 'AGL', the observer altitude `oalt` is in meters above ground level. If `oaltopt` is 'MSL', `oalt` is interpreted as altitude above zero, or mean sea level. If omitted, 'AGL' is assumed.

`viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt,taltopt)` controls whether the target points are at a relative or absolute altitude. If the target altitude option is 'AGL', the target altitude `talt` is in

# viewshed

---

meters above ground level. If `taltopt` is 'MSL', `talt` is interpreted as altitude above zero, or mean sea level. If omitted, 'AGL' is assumed.

`viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt,taltopt,... actualradius)` does the visibility calculation on a sphere with the specified radius. If omitted, the radius of the earth in meters is assumed. The altitudes, the elevations, and the radius should be in the same units. This calling form is most useful for computations on bodies other than the Earth.

`viewshed(Z,refvec,lat1,lon1,oalt,talt,oaltopt,taltopt,... actualradius,effectiveradius)` assumes a larger radius for propagation of the line of sight. This can account for the curvature of the signal path due to refraction in the atmosphere. For example, radio propagation in the atmosphere is commonly treated as straight line propagation on a sphere with  $4/3$  the radius of the Earth. In that case the last two arguments would be `R` and  $4/3 * R_e$ , where `R` is the radius of the earth. Use `Inf` for flat Earth `viewshed` calculations. The altitudes, the elevations, and the radii should be in the same units.

## Example

Compute visibility for a point on the peaks map. Add the detailed information for the line of sight calculation between two points from `los2`.

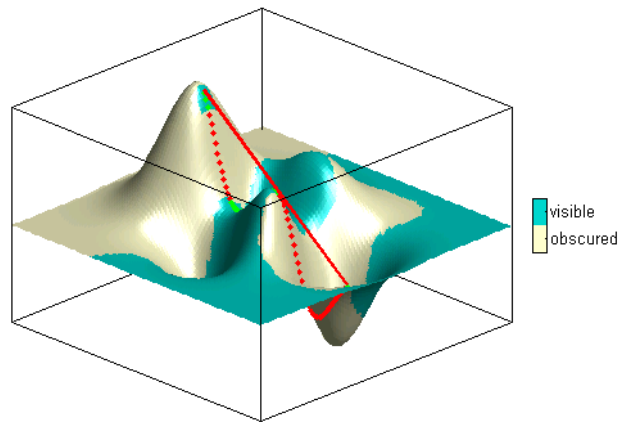
```
Z = 500*peaks(100);
refvec = [ 1000 0 0];
[lat1,lon1,lat2,lon2]=deal(-0.027,0.05,-0.093,0.042);
[visZ,vismaprefvec] = viewshed(Z,refvec,lat1,lon1,100);
[vis,visprofile,dist,z,lattrk,lontrk] = ...
    los2(Z,refvec,lat1,lon1,lat2,lon2,100);
axesm('globe','geoid',almanac('earth','sphere','meters'))
meshm(visZ,vismaprefvec,size(Z),Z);
axis tight
camposm(-10,-10,1e6);
camupm(0,0)
colormap(flipud(summer(2)));
brighten(0.75);
shading interp
```

```

camlight
h = lcolorbar({'obscured','visible'});
set(h,'Position',[.875 .45 .02 .1])

plot3m(lattrk([1;end]),lontrk([1; end]), ...
    z([1; end])+[100; 0],'r','linewidth',2)
plotm(lattrk(~visprofile),lontrk(~visprofile), ...
    z(~visprofile),'r.','markersize',10)
plotm(lattrk(visprofile),lontrk(visprofile), ...
    z(visprofile),'g.','markersize',10)

```



Compute the surface areas visible by radar from an aircraft 3000 meters above the Yellow Sea. Assume that radio wave propagation in the atmosphere can be modeled as straight lines on a  $4/3$  radius Earth. Display the visible areas as blue and the obscured areas as red. Drape the visibility colors on an elevation map, and use lighting to bring out the surface topography. The aircraft's radar can see out a certain radius on the surface of the ocean, but some ocean areas are shadowed by the island of Jeju-Do. Also some mountain valleys closer than the ocean horizon are obscured, while some mountain tops further away are visible.

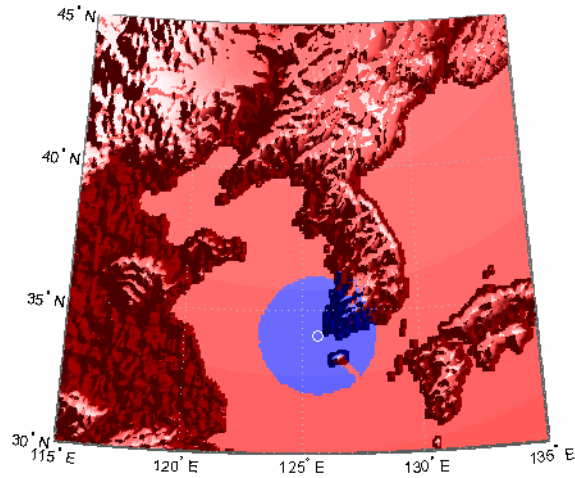
```
load korea
```

# viewshed

---

```
map(map<0) = -1;
figure
worldmap(map,refvec)
da = daspect;
pba = pbaspect;
da(3) = 7.5*pba(3)/da(3);
daspect(da);
demcmap(map)
camlight(90,5);
camlight(0,5);
lighting phong
material([0.25 0.8 0])
lat = 34.0931; lon = 125.6578;
altobs = 3000; alttarg = 0;
plotm(lat,lon,'wo')
Re = almanac('earth','radius','m');
[vmap,vmapl] = viewshed( ...
    map,refvec,lat,lon,altobs,alttarg, ...
    'MSL','AGL',Re,4/3*Re);
meshm(vmap,vmapl,size(map),map)
caxis auto; colormap([1 0 0; 0 0 1])
lighting phong; material metal
axis off
```



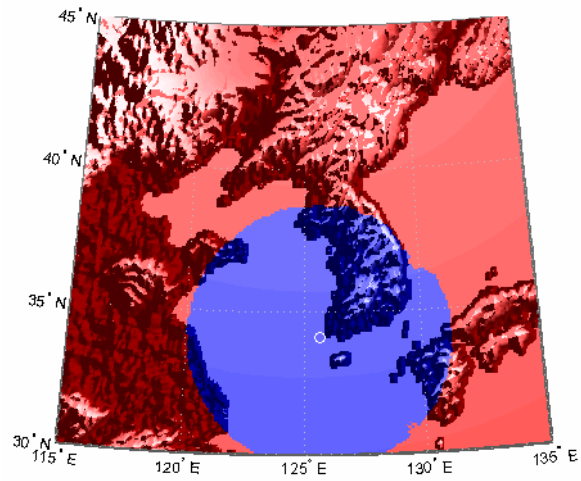


Over what area can the radar plane flying at an altitude of 3000 meters have line-of-sight to other aircraft flying at 5000 meters? Now the area is much larger. Some edges of the area are reduced by shadowing from Jeju-Do and the mountains on the Korean peninsula.

```
[vmap,vmap1] = viewshed(map,refvec,lat,lon,3000,5000,...
                        'MSL','MSL',Re,4/3*Re);
clmo surface
meshm(vmap,vmap1,size(map),map)
material metal
lighting phong
```

# viewshed

---



**See Also**

los2

**Purpose**

Azimuth on ellipsoid from direction angle in map plane

**Syntax**

```
az = vinvtran(x,y,th)
az = vinvtran(mstruct,x,y,th)
[az,len] = vinvtran(...)
```

`az = vinvtran(x,y,th)` transforms an angle in the projection space at the point specified by `x` and `y` into an azimuth angle in geographic coordinates. The map projection currently displayed is used to define the projection space. The input angles must be in the same units as specified by the current map projection. The inputs can be scalars or matrices of equal size. The angle in the projection space angle `th` is defined as positive counterclockwise from the  $x$ -axis.

`az = vinvtran(mstruct,x,y,th)` uses the map projection defined by the input `struct` to compute the map projection.

`[az,len] = vinvtran(...)` also returns the vector length in the geographic coordinate system. A value of 1 indicates no scale distortion for that angle.

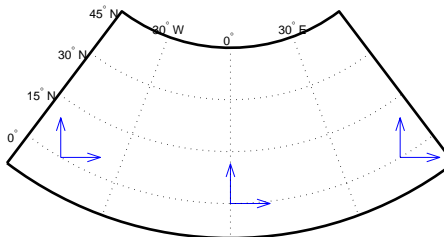
**Background**

While vectors along the  $y$ -axis always point to north in a cylindrical projection in the normal aspect, they can point east or west of north on conics, azimuthals, and other projections. This function computes the geographic azimuth for angles in the projected space.

**Examples**

Sample calculations:

```
axesm('eqdconic','maplatlim',[-10 45],'maplonlim',[-55 55])
gridm; framem; mlabel; plabel
[x,y] = mofdtran([0 0 0],[-45 0 45]);
quiver(x,y,[.2 .2 .2],[0 0 0],0)
quiver(x,y,[0 0 0],[.2 .2 .2],0)
```



```
vinvtran(x,y,[ 0 0 0])
```

```
ans =  
    57.345    90.338   124.98
```

```
vinvtran(x,y,[ 90 90 90])
```

```
ans =  
    331.99         0    28.008
```

## Limitations

This transformation is limited to the region specified by the frame limits in the current map definition.

## Remarks

The geographic azimuth angle is measured clockwise from north. The projection space angle is measured counterclockwise from the  $x$ -axis.

This function uses a finite difference technique. The geographic coordinates are perturbed slightly in different directions and projected.

A small amount of error is introduced by numerical computation of derivatives and the variation of map distortion parameters.

**See Also**

vfwdtran, mfwdtran, minvtran, defaultm

# vmap0data

---

## Purpose

Read selected data from Vector Map Level 0

## Syntax

```
struct = vmap0data(library,latlim,lonlim,theme,topolevel)  
struct = vmap0data(devicename,library,...)  
[struct1, struct2,...] = vmap0data(...,{topolevel1,  
    topolevel2,...})
```

`struct = vmap0data(library,latlim,lonlim,theme,topolevel)` reads the data for the specified theme and topology level directly from the VMAP0 CD-ROM. There are four CDs, one for each of the libraries: 'NOAMER' (North America), 'SASAUS' (Southern Asia and Australia), 'EURNASIA' (Europe and Northern Asia), and 'SOAMAFR' (South America and Africa). The desired *theme* is specified by a two-letter code string. A list of valid codes is displayed when an invalid code, such as '?', is entered. *topolevel* defines the type of data returned. It is a string containing 'patch', 'line', 'point', or 'text'. The region of interest can be given as a point latitude and longitude or as a region with two-element vectors of latitude and longitude limits. The units of latitude and longitude are degrees. The data covering the requested region is returned, but will include data extending to the edges of the tiles. The result is returned as a Mapping Toolbox geographic data structure.

`struct = vmap0data(devicename,library,...)` specifies the logical device name of the CD-ROM for computers that do not automatically name the mounted disk.

```
[struct1, struct2,...] =  
vmap0data(...,{topolevel1,topolevel2,...})
```

reads several topology levels. The levels must be specified as a cell array with the entries 'patch', 'line', 'point', or 'text'. Entering {'all'} for the topology level argument is equivalent to {'patch', 'line', 'point', 'text'}. Upon output, the data structures are returned in the output arguments by topology level in the same order as they were requested.

## Background

The Vector Map (VMAP) Level 0 database represents the third edition of the *Digital Chart of the World*. The second edition was a limited

release item published in 1995. The product is dual named to show its lineage to the original DCW, published in 1992, while positioning the revised product within a broader emerging family of VMAP products. VMAP Level 0 is a comprehensive 1:1,000,000 scale vector base map of the world. It consists of cartographic, attribute, and textual data stored on compact disc read-only memory (CD-ROM). The primary source for the database is the Operational Navigation Chart (ONC) series of the U. S. National Geospatial Intelligence Agency (NGA), formerly the National Imagery and Mapping Agency (NIMA), and before that, the Defense Mapping Agency (DMA). This is the largest scale unclassified map series in existence that provides consistent, continuous global coverage of essential base map features. The database contains more than 1,900 MB of vector data and is organized into 10 thematic layers. The data includes major road and rail networks, major hydrologic drainage systems, major utility networks (cross-country pipelines and communication lines), all major airports, elevation contours (1000 foot (ft), with 500 ft and 250 ft supplemental contours), coastlines, international boundaries, and populated places. The database can be accessed directly from the four optical CD-ROMs that store the database or can be transferred to magnetic media.

## Remarks

Latitudes and longitudes use WGS84 as a horizontal datum. Elevations and depths are in meters above mean sea level.

Some VMAP0 themes do not contain all topology levels. In those cases, empty matrices are returned.

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

The major differences between VMAP0 and the DCW are the elimination of the gazette layer, addition of bathymetric data, and updated political boundaries.

Vector Map Level 0, created in the 1990s, is still probably the most detailed global database of vector map data available to the public. VMAP0 CD-ROMs are available from through the U.S. Geological Survey (USGS):

USGS Information Services (Map and Book Sales)  
Box 25286  
Denver Federal Center  
Denver, CO 80225  
Telephone: (303) 202-4700  
Fax: (303) 202-4693

---

**Note** For details on locating map data for download over the Internet, see the following documentation at the MathWorks Web site: <http://www.mathworks.com/support/tech-notes/2100/2101.html>.

---

## Examples

The *devicename* is platform dependent. On an MS-DOS based operating system it would be something like 'd:', depending on the logical device code assigned to the CD-ROM drive. On a UNIX operating system, the CD-ROM might be mounted as '\cdrom', '\CDROM', '\cdrom1', or something similar. Check your computer's documentation for the right *devicename*.

```
s = vmap0data(devicename, 'NOAMER', 41, -69, '?', 'patch');
```

```
??? Error using ==> vmap0data  
Theme not present in library NOAMER
```

Valid theme identifiers are:

```
libref : Library Reference  
tileref: Tile Reference  
bnd    : Boundaries  
dq     : Data Quality  
elev   : Elevation  
hydro  : Hydrography  
ind    : Industry  
phys   : Physiography  
pop    : Population  
trans  : Transportation  
util   : Utilities
```



```
veg      : Vegetation

BNDpatch = vmap0data(devicename, 'NOAMER', ...
                  [41 44], [-72 -69], 'bnd', 'patch')
BNDpatch =
1x169 struct array with fields:
    type
    otherproperty
    altitude
    lat
    long
    tag
```

Here are other examples:

```
[TRtext, TRline] = vmap0data(devicename, 'SASAUS', ...
    [-48 -34], [164 180], 'trans', {'text', 'line'});

[BNDpatch, BNDline, BNDpoint, BNDtext] = vmap0data(devicename, ...
    'EURNASIA', -48 , 164, 'bnd', {'all'});
```

## See Also

vmap0read, vmap0rhead, geoshow, extractm, mlayers

# vmap0read

---

## Purpose

Read Vector Map Level 0 file

## Syntax

```
vmap0read  
vmap0read(filepath,filename)  
vmap0read(filepath,filename,recordIDs)  
vmap0read(filepath,filename,recordIDs,field,varlen)  
struc = vmap0read(...)  
[struc,field] = vmap0read(...)  
[struc,field,varlen] = vmap0read(...)  
[struc,field,varlen,description] = vmap0read(...)  
[struc,field,varlen,description,  
  narrativefield] = vmap0read(...)
```

vmap0read reads a VMAP0 file. The user selects the file interactively.

vmap0read(*filepath*,*filename*) reads the specified file. The combination [*filepath filename*] must form a valid complete filename.

vmap0read(*filepath*,*filename*,recordIDs) reads selected records or fields from the file. If recordIDs is a scalar or a vector of integers, the function returns the selected records. If recordIDs is a cell array of integers, all records of the associated fields are returned. vmap0read(*filepath*,*filename*,recordIDs,*field*,*varlen*)

uses previously read field and variable-length record information to skip parsing the file header (see below).

struc = vmap0read(...) returns the file contents in a structure.

[struc,*field*] = vmap0read(...) returns the file contents and a structure describing the format of the file.

[struc,*field*,*varlen*] = vmap0read(...) also returns a vector describing which fields have variable-length records.

[struc,*field*,*varlen*,*description*] = vmap0read(...) also returns a string describing the contents of the file.

[struc,*field*,*varlen*,*description*,*narrativefield*] = vmap0read(...) also returns the name of the narrative file for the current file.

## Background

The Vector Map Level 0 (VMAPO) uses binary files in a variety of formats. This function determines the format of the file and returns the contents in a structure. The field names of this structure are the same as the field names in the VMAPO file.

## Remarks

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

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

## Examples

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

```
s = vmap0read('VMAP/VMAPLVO/NOAMER/', 'GRT')

s =
        id: 1
      data_type: 'GEO'
        units: 'M'
  ellipsoid_name: 'WGS 84'
ellipsoid_detail: 'A=6378137 B=6356752 Meters'
  vert_datum_name: 'MEAN SEA LEVEL'
  vert_datum_code: '015'
  sound_datum_name: 'N/A'
  sound_datum_code: 'N/A'
    geo_datum_name: 'WGS 84'
    geo_datum_code: 'WGE'
  projection_name: 'Dec. Deg. (unproj.)'

s = vmap0read('VMAP/VMAPLVO/NOAMER/TRANS/', 'INT.VDT')

s =
34x1 struct array with fields:
    id
```

# vmap0read

---

```
table
attribute
value
description

s(1)

ans =
      id: 1
      table: 'aerofacp.pft'
      attribute: 'use'
      value: 8
      description: 'Military'
s = vmap0read('VMAP/VMAPLV0/NOAMER/TRANS/', 'AEROFACP.PFT', 1)

s =
      id: 1
      f_code: 'GB005'
      iko: 'BGTL'
      nam: 'THULE AIR BASE'
      na3: 'GL52085'
      use: 8
      zv3: 77
      tile_id: 10
      end_id: 1

s = vmap0read('VMAP/VMAPLV0/NOAMER/TRANS/', 'AEROFACP.PFT', {1,2})

s =
1x4424 struct array with fields:
      id
      f_code
```

## See Also

vmap0data, vmap0rhead

<b>Purpose</b>	Read Vector Map Level 0 file headers
<b>Syntax</b>	<pre>vmap0rhead vmap0rhead(filepath,filename) vmap0rhead(filepath,filename,fid) vmap0rhead(...), str = vmap0rhead(...)</pre> <p>vmap0rhead allows the user to select the header file interactively.</p> <p>vmap0rhead(filepath,filename) reads from the specified file. The combination [filepath filename] must form a valid complete filename.</p> <p>vmap0rhead(filepath,filename,fid) reads from the already open file associated with fid.</p> <p>vmap0rhead(...), with no output arguments, displays the formatted header information on the screen.</p> <p>str = vmap0rhead(...) returns a string containing the VMAP0 header.</p>
<b>Background</b>	The Vector Map Level 0 (VMAP0) uses header strings in most files to document the contents and format of that file. This function reads the header string and displays a formatted version in the Command Window, or returns it as a string.
<b>Remarks</b>	<p>This function reads all VMAP0 files except index files (files with names ending in 'X'), thematic index files (files with names ending in 'TI') and spatial index files (files with names ending in 'SI').</p> <p>File separators are platform dependent. The <i>filepath</i> input must use appropriate file separators, which you can determine using the MATLAB filesep function.</p>
<b>Examples</b>	<p>The following example uses UNIX file separators and pathname:</p> <pre>s = vmap0rhead('VMAP/VMAPLV0/NOAMER/' , 'GRT')</pre>

# vmap0rhead

---

```
s =
L;Geographic Reference Table;-;id=I,1,P,Row
Identifier,-,-,-:data_type=T,3,N,Data
Type,-,-,-:units=T,3,N,Units of Measure Code for
Library,-,-,-:ellipsoid_name=T,15,N,Ellipsoid,-,-,-:ellipsoid
_detail=T,50,N,Ellipsoid
Details,-,-,-:vert_datum_name=T,15,N,Datum Vertical
Reference,-,-,-:vert_datum_code=T,3,N,Vertical Datum
Code,-,-,-:sound_datum_name=T,15,N,Sounding
Datum,-,-,-:sound_datum_code=T,3,N,Sounding Datum
Code,-,-,-:geo_datum_name=T,15,N,Datum Geodetic
Name,-,-,-:geo_datum_code=T,3,N,Datum Geodetic
Code,-,-,-:projection_name=T,20,N,Projection Name,-,-,-,;

vmap0rhead('VMAP/VMAPLV0/NOAMER/TRANS/','AEROFACP.PFT')
L
Airport Point Feature Table
aerofacp.doc
id=I,1,P,Row Identifier,-,-,-,
f_code=T,5,N,FACC Feature Code,char.vdt,-,-,
iko=T,4,N,ICAO Designator,char.vdt,-,-,
nam=T,*N,Name,char.vdt,-,-,
na3=T,*N,Name,char.vdt,-,-,
use=S,1,N,Usage,int.vdt,-,-,
zv3=S,1,N,Airfield/Aerodrome Elevation (meters),int.vdt,-,-,
tile_id=S,1,N,Tile Reference ID,-,tile1_id.pti,-,
end_id=I,1,N,Entity Node Primitive ID,-,end1_id.pti,-,
```

## See Also

vmap0data, vmap0read

## Purpose

Wrap longitudes to values west of specified meridian

---

**Note** The `westof` function is obsolete and will be removed in a future release of Mapping Toolbox. Replace it with the following calls, which are also more efficient:

```
westof(lon,meridian,'degrees') ==> meridian-mod(meridian-lon,360)
```

```
westof(lon,meridian,'radians') ==> meridian-mod(meridian-lon,2*pi)
```

---

## Syntax

```
lonWrapped = westof(lon,meridian)
```

```
lonWrapped = westof(lon,meridian,angleunits)
```

`lonWrapped = westof(lon,meridian)` wraps angles in `lon` to values in the interval `(meridian-360 meridian]`. `lon` is a scalar longitude or vector of longitude values. All inputs and outputs are in degrees.

`lonWrapped = westof(lon,meridian,angleunits)` specifies the input and output units with the string *angleunits*. *angleunits* can be either 'degrees' or 'radians'. It may be abbreviated and is case-insensitive. If *angleunits* is 'radians', the input is wrapped to the interval `(meridian-2*pi meridian]`.

# worldfileread

---

**Purpose** Read worldfile and return referencing matrix

**Syntax** `R = worldfileread(worldfilename)`  
`R = worldfileread(worldfilename)` reads the worldfile data from `worldfilename` and constructs the referencing matrix `R`.  
`R` is a 3-by-2 affine transformation matrix that is used in `pix2map` and `map2pix` to transform pixel row and column coordinates to/from map/model coordinates according to  $[x \ y] = [\text{row} \ \text{col} \ 1] * R$ .

**Example**

```
R = worldfileread('concord_ortho_w.tfw')  
  
R =      1.0e+005 *  
          0 -0.0000100000000000  
0.0000100000000000      0  
2.0699950000000000      9.1300050000000001
```

**See Also** `getworldfilename`, `makerefmat`, `pix2map`, `map2pix`, `worldfilewrite`



**Purpose** Construct worldfile from referencing matrix

**Syntax** `worldfilewrite(R, worldfilename)`  
`worldfilewrite(R, worldfilename)` calculates the worldfile entries corresponding to referencing matrix R and writes them into the file `worldfilename`.

R is a 3-by-2 affine transformation matrix that is used in `pix2map` and `map2pix` to transform pixel row and column coordinates to/from map/model coordinates according to  $[x \ y] = [row \ col \ 1] * R$ .

**Example**

```
R = worldfileread('concord_ortho_w.tfw');  
worldfilewrite(R, 'concord_ortho_w_test.tfw');
```

constructs the referencing matrix R from `concord_ortho_w.tfw`, then reconstructs a copy of the worldfile from R.

**See Also** `getworldfilename`, `pix2map`, `map2pix`, `worldfileread`

# worldmap

---

**Purpose** Construct map axes for given region of world

**Syntax**

```
worldmap region
worldmap(region)
worldmap
worldmap(latlim, lonlim)
worldmap(Z, R)
h = worldmap(...)
```

`worldmap region` or `worldmap(region)` sets up an empty map axes with projection and limits suitable to the part of the world specified in `region`. `region` can be a string or a cell array of strings. Permissible strings include names of continents, countries, and islands as well as 'World', 'North Pole', 'South Pole', and 'Pacific'.

`worldmap` with no arguments presents a menu from which you can select the name of a single continent, country, island, or region.

`worldmap(latlim, lonlim)` allows you to define a custom geographic region in terms of its latitude and longitude limits in degrees. `latlim` and `lonlim` are two-element vectors of the form [`southern_limit` `northern_limit`] and [`western_limit` `eastern_limit`], respectively.

`worldmap(Z, R)` derives the map limits from the extent of a regular data grid or georeferenced image `Z`, with 3-by-2 referencing matrix or 1-by-3 referencing vector `R`.

`h = worldmap(...)` returns the handle of the map axes.

For cylindrical projections, `worldmap` uses `tightmap` set the axis limits tight around the map. If you change the projection, or just want more white space around the map frame, use `tightmap` again or `axis auto`.

## Examples

### Example 1

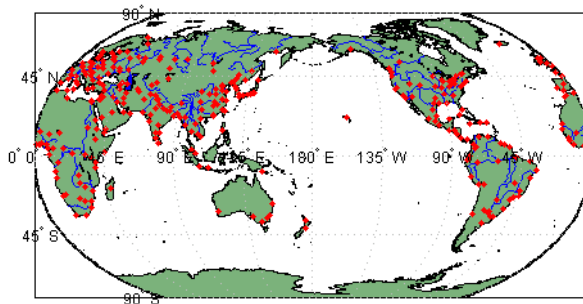
Set up a world map and draw coarse coastlines:

```
worldmap('World')
load coast
plotm(lat, long)
```

## Example 2

Set up worldmap with land areas, major lakes and rivers, and cities and populated places:

```
ax = worldmap('World');
setm(ax, 'Origin', [0 180 0])
land = shaperead('landareas', 'UseGeoCoords', true);
geoshow(ax, land, 'FaceColor', [0.5 0.7 0.5])
lakes = shaperead('worldlakes', 'UseGeoCoords', true);
geoshow(lakes, 'FaceColor', 'blue')
rivers = shaperead('worldrivers', 'UseGeoCoords', true);
geoshow(rivers, 'Color', 'blue')
cities = shaperead('worldcities', 'UseGeoCoords', true);
geoshow(cities, 'Marker', '.', 'Color', 'red')
```



## Example 3

Draw a map of Antarctica:

```
worldmap('antarctica')
antarctica = shaperead('landareas', 'UseGeoCoords', true,...
    'Selector',{@(name) strcmp(name,'Antarctica'), 'Name'});
patchm(antarctica.Lat, antarctica.Lon, [0.5 1 0.5])
```



## Example 4

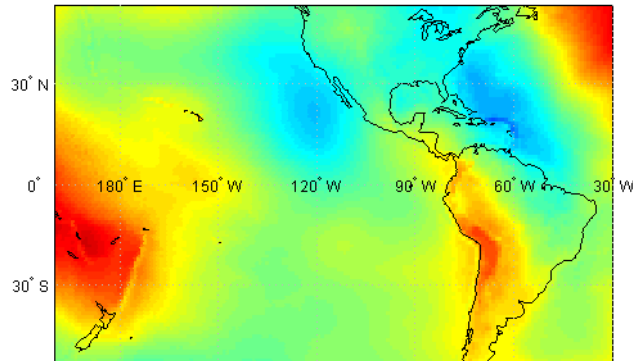
Draw a map of Africa and India with major cities and populated places:

```
worldmap({'Africa','India'})
land = shaperead('landareas.shp','UseGeoCoords',true);
geoshow(land,'FaceColor',[0.15 0.5 0.15])
cities = shaperead('worldcities','UseGeoCoords',true);
```

## Example 5

Make a map of the geoid over South America and the central Pacific:

```
worldmap([-50 50],[160 -30])
load geoid
geoshow(geoid,geoidrefvec,'DisplayType','texturemap');
load coast
geoshow(lat,long)
```



### Example 6

Draw a map of terrain elevations in Korea:

```
load korea
h = worldmap(map, refvec);
set(h, 'Visible', 'off')
geoshow(h, map, refvec, 'DisplayType', 'texturemap')
colormap(demcmap(map))
```

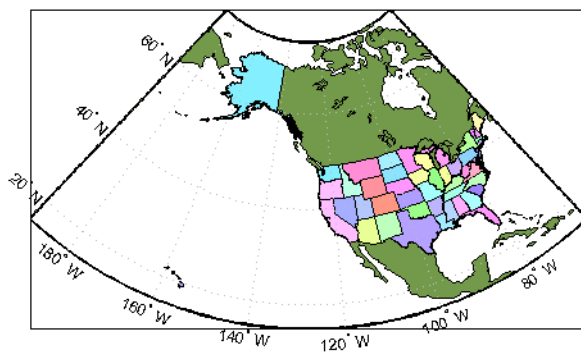
### Example 7

Make a map of the United States of America, coloring state polygons:

```
ax = worldmap('USA');
load coast
geoshow(ax, lat, long,...
'DisplayType', 'polygon', 'FaceColor', [.45 .60 .30])
states = shaperead('usastatelo', 'UseGeoCoords', true);
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor', ...
    polcmap(numel(states))}); % NOTE - colors are random
geoshow(ax, states, 'DisplayType', 'polygon', ...
'SymbolSpec', faceColors)
```

# worldmap

---



## See Also

`axesm`, `framem`, `geoshow`, `gridm`, `mlabel`, `plabel`, `tightmap`, `usamap`

**Purpose** Wrap angle in degrees to [-180 180]

**Syntax** lonWrapped = wrapTo180(lon)  
lonWrapped = wrapTo180(lon) wraps angles in lon, in degrees, to the interval [-180 180] such that 180 maps to 180 and -180 maps to -180. (In general, odd, positive multiples of 180 map to 180 and odd, negative multiples of 180 map to -180.)

**See Also** wrapTo360, wrapTo2Pi, wrapToPi

# wrapTo360

---

**Purpose** Wrap angle in degrees to [0 360]

**Syntax** lonWrapped = wrapTo360(lon)

lonWrapped = wrapTo360(lon) wraps angles in lon, in degrees, to the interval [0 360] such that 0 maps to 0 and 360 maps to 360. (In general, positive multiples of 360 map to 360 and negative multiples of 360 map to zero.)

**See Also** wrapTo180, wrapTo2Pi, wrapToPi



**Purpose** Wrap angle in radians to  $[0, 2\pi]$

**Syntax** `lambdaWrapped = wrapTo2Pi(lambda)`  
`lambdaWrapped = wrapTo2Pi(lambda)` wraps angles in `lambda`, in radians, to the interval  $[0, 2\pi]$  such that 0 maps to 0 and  $2\pi$  maps to  $2\pi$ . (In general, positive multiples of  $2\pi$  map to  $2\pi$  and negative multiples of  $2\pi$  map to zero.)

**See Also** `wrapTo180`, `wrapTo360`, `wrapToPi`

# wrapToPi

---

**Purpose** Wrap angle in radians to  $[-\pi, \pi]$

**Syntax** `lambdaWrapped = wrapToPi(lambda)`  
`lambdaWrapped = wrapToPi(lambda)` wraps angles in `lambda`, in radians, to the interval  $[-\pi, \pi]$  such that  $\pi$  maps to  $\pi$  and  $-\pi$  maps to  $-\pi$ . In general, odd, positive multiples of  $\pi$  map to  $\pi$  and odd, negative multiples of  $\pi$  map to  $-\pi$ .

**See Also** `wrapTo180`, `wrapTo360`, `wrapTo2Pi`

**Purpose**

Adjust  $z$ -plane of displayed map objects

**Syntax**

```
zdatam
zdatam(hndl)
zdatam('str')
zdatam(hndl,zdata)
zdatam('str',zdata)
```

`zdatam` displays a GUI for selecting an object from the current axes and modifying its `ZData` property.

`zdatam(hndl)` and `zdatam('str')` display a GUI to modify the `ZData` of the object(s) specified by the input. `str` is any string recognized by `handlem`.

`zdatam(hndl,zdata)` alters the  $z$ -plane position of displayed map objects designated by the MATLAB graphics handle `hndl`. The  $z$ -plane position may be the `Z` position in the case of text objects, or the `ZData` property in the case of other graphic objects. The function behaves as follows:

- If `hndl` is an `hggroup` handle, the `ZData` property of the children in the `hggroup` are altered.
- If the handle is scalar, then `ZData` can be either a scalar ( $z$ -plane definition), or a matrix of appropriate dimension for the displayed object.
- If `hndl` is a vector, then `ZData` can be a scalar or a vector of the same dimension as `hndl`.
- If `ZData` is a scalar, then all objects in `hndl` are drawn on the `ZData`  $z$ -plane.
- If `ZData` is a vector, then each object in `hndl` is drawn on the plane defined by the corresponding `ZData` element.
- If `ZData` is omitted, then a modal dialog box prompts for the `ZData` entry.

# zdatam

---

`zdatam('str', zdata)` identifies the objects by the input `str`, where `str` is any string recognized by `handlem`, and uses `zdata` as described above to update their `ZData` property.

## **Description**

This function adjusts the  $z$ -plane position of selected graphics objects. It accomplishes this by setting the objects' `ZData` properties to the appropriate values.

## **See Also**

`handlem`, `setm`

**Purpose** Wrap longitudes to [0 360] degree interval

---

**Note** The zero22pi function has been replaced by wrapTo360 and wrapTo2Pi.

---

## Syntax

```
newlon = zero22pi(lon)
newlon = zero22pi(lon,angleunits)
```

newlon = zero22pi(lon) *wraps* the input angle lon in degrees to the 0 to 360 degree range.

newlon = zero22pi(lon,angleunits) works in the units defined by the string *angleunits*, which can be either 'degrees' or 'radians'. *angleunits* can be abbreviated and is case-insensitive.

## Examples

```
zero22pi(567.5)

ans =
    207.5

zero22pi(-567.5)

ans =
    152.5

zero22pi(-7.5,'radian')

ans =
    5.0664
```

**See Also** wrapTo2Pi, wrapTo360

# zerom

---

**Purpose** Construct regular data grid of 0s

**Syntax** `[Z,refvec] = zerom(latlim,lonlim,scale)`

`[Z,refvec] = zerom(latlim,lonlim,scale)` returns a full regular data grid consisting entirely of 0s and a three-element referencing vector for the returned Z. The two-element vectors `latlim` and `lonlim` define the latitude and longitude limits of the geographic region. They should be of the form `[south north]` and `[west east]`, respectively. The scalar `scale` specifies the number of rows and columns per degree of latitude and longitude.

**Examples** `[Z,refvec] = zerom([46,51],[-79,-75],1)`

```
Z =
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
    0    0    0    0
refvec =
    1    51   -79
```

**See Also** `limitm`, `nanm`, `onem`, `sizem`, `spzerom`

**Purpose** Define map axes and modify map projection and display properties

### Activation

Command Line	Maptool	Map Display
axesm axesmui c = axesmui(...)	<b>Display &gt; Projection</b>	extend-click map display

### Description

axesm activates a Projection Control dialog box, which allows map projection definition and property modification. If no map is currently defined, axesm creates a map axes with the Robinson projection as the default.

axesmui activates the Projection Control box for the current map axes.

c is an optional output argument that indicates whether the Projection Control dialog box was closed by the cancel button. c = 1 if the cancel button is pushed. Otherwise, c = 0.

Extend-clicking a map display brings up the Projection Control dialog box for that map axes.

## Controls

The screenshot shows the 'Projection Control' dialog box. The 'Map Projection' dropdown is set to 'Pcyl: Mollweide'. The 'Geoid' section has '1.00' and '0' in the input boxes, and 'unit sphere' in the dropdown. 'Angle Units' is set to 'degrees'. The 'Map Limits' and 'Frame Limits' sections have Latitude from -90 to 90 and Longitude from -180 to 180. 'Map Origin' has 'Lat and Long' at (0, 0) and 'Orientation' at 0. 'Cartesian Origin' has 'False E and N' at (0, 0) and 'Scalefactor' at 1. 'Parallels' is set to 40.7333. 'Aspect' is set to 'normal'. The bottom buttons are 'Frame', 'Grid', 'Labels', 'Fill in', 'Reset', 'Apply', 'Help', and 'Cancel'.

The **Map Projection** pull-down menu is used to select a map projection. The projections are listed by type, and each is preceded by a four-letter type indicator:

Cyln = Cylindrical  
Pcyl = Pseudocylindrical  
Coni = Conic  
Poly = Polyconic  
Pcon = Pseudoconic  
Azim = Azimuthal  
Mazi = Modified Azimuthal  
Pazi = Pseudoazimuthal

The **Zone** button and edit box are used to specify the UTM or UPS zone. For non-UTM and UPS projections, the two are disabled.

The **Geoid** edit boxes and pull-down menu are used to specify the geoid. Units must be in meters for the UTM and UPS projections, since this is the standard unit for the two projections. For non-UTM and UPS



projections, the geoid unit can be anything, bearing in mind that the resulting projected data will be in the same units as the geoid.

The **Angle Units** pull-down menu is used to specify the angle units used on the map projection. All angle entries corresponding to the current map projection must be entered in these units. Current angle entries are automatically updated when new angle units are selected.

The **Map Limits** edit boxes are used to specify the extent of the map data in geographic coordinates. The **Latitude** edit boxes contain the southern and northern limits of the map. The **Longitude** edit boxes contain the western and eastern limits of the map. The map limits establish the extent of the meridian and parallel grid lines, regardless of the display settings (see grid settings). Map limits are always in geographic coordinates, regardless of the map origin and orientation setting. In the normal aspect, the map display is trimmed to the minimum of the map and frame limits.

The **Frame Limits** edit boxes are used to specify the location of the map frame, measured from the center of the map projection in the base coordinate system. The **Latitude** edit boxes contain the southern and northern frame edge locations. The **Longitude** edit boxes contain the western and eastern frame edge locations. Displayed map data are trimmed at the frame limits. For azimuthal map projections, the latitude limits should be set to `inf` and the desired trim distance from the map origin. In the normal aspect, the map display is trimmed to the minimum of the map and frame limits.

The **Map Origin** edit boxes are used to specify the origin and aspect angle of the map projection. The **Lat** and **Long** boxes specify the map origin in geographic coordinates. This is the point that is placed in the center of the projection. If either box is left blank, 0 degrees is used. The **Orientation** box specifies the azimuth angle of the North Pole relative to the map origin. Azimuth is measured clockwise from the top of the projection. If the **Orientation** box is disabled, then the selected map projection requires a fixed orientation. See the *Mapping Toolbox User's Guide* for a complete description of the map origin.

The **Cartesian Origin** edit boxes are used to specify the  $x$ - $y$  offset, along with a desired scale factor of the map projection. The **False E and N** boxes specify the false easting and northing in Cartesian coordinates. These must be in the same units as the geoid. The **Scalefactor** box specifies the scale factor used in the map projection calculations.

The **Parallels** edit boxes specify the standard parallels of the selected map projection. A particular map projection may have one or two standard parallels. If the edit boxes are disabled, then the selected projection has no standard parallels or the standard parallels are fixed.

The **Aspect** pull-down menu is used to select a normal or transverse display aspect. When the aspect is normal, *north* (on the base projection) is up, and the map is displayed in a *portrait* setting. In a transverse aspect, north (in the base projection) is to the right, and the map is displayed in a *landscape* setting. This property does not control the map projection aspect. The projection aspect is determined by the map Origin property).

The **Frame** button brings up the Map Frame Properties dialog box, which allows the map frame settings to be modified.

The **Grid** button brings up the Map Grid Properties dialog box, which allows the map grid settings to be modified.

The **Labels** button brings up the Map Label Properties dialog box, which allows the parallel and meridian label settings to be modified.

The **Fill in** button is used to compute projection and display settings based on any currently specified map parameters. Only settings that are left blank are affected when this button is pushed.

The **Reset** button is used to reset the default projection properties and display settings of the current map. Default display settings include frame, grid, and label properties set to 'off'.

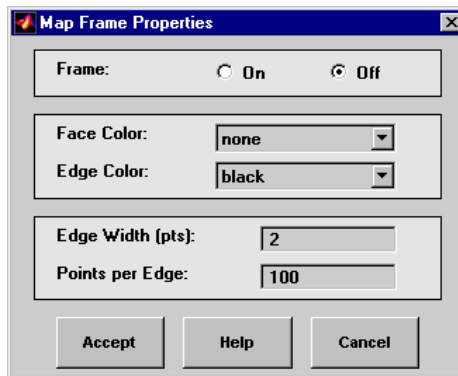
The **Apply** button is used to apply the projection and display settings to the current map, which results in the map being reprojected.

The **Help** button is used to bring up online help text for each control on the Projection Control dialog box.

The **Cancel** button disregards any modified projection or display settings and closes the Projection Control dialog box.

### Map Frame Properties Dialog Box

This dialog box allows modification of the map frame settings. It is accessed via the **Frame** button on the Projection Control dialog box.



The **Frame** selection buttons determine whether the map frame is visible.

The **Face Color** pull-down menu is used to select the background color of the map frame. Selecting none results in a transparent frame background, i.e., the same as the axes color. Selecting custom allows a custom RGB triple to be defined for the background color.

The **Edge Color** pull-down menu is used to select the color of the frame edge. Selecting none hides the frame edge. Selecting custom allows a custom RGB triple to be defined for the edge color.

The **Edge Width** edit box is used to enter the line width of the frame edge, in points.

The **Points per Edge** edit box is used to enter the number of points used to display each edge of the map frame.

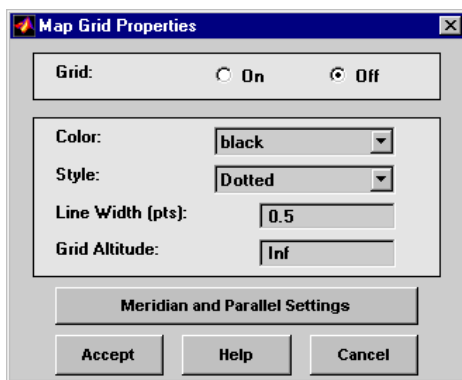
The **Accept** button accepts any modifications made to the map frame properties and returns to the Projection Control dialog box. Changes

are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the map frame properties and returns to the Projection Control dialog box.

## Map Grid Properties Dialog Box

This dialog box allows modification of the map frame settings. It is accessed via the **Grid** button on the Projection Control dialog box.



The Grid selection buttons determine whether the map grid is visible.

The **Color** pull-down menu is used to select the color of the map grid lines. Selecting custom allows a custom RGB triple to be defined for the grid line color.

The **Style** pull-down menu is used to select the line style of the map grid lines.

The **Line Width** edit box is used to enter the width of the map grid lines, in points.

The **Grid Altitude** edit box is used to enter  $z$ -axis location of the map grid. This property can be used to place some mapped objects above or below the map grid. The default map grid altitude is `inf`, which places the grid above all other mapped objects.

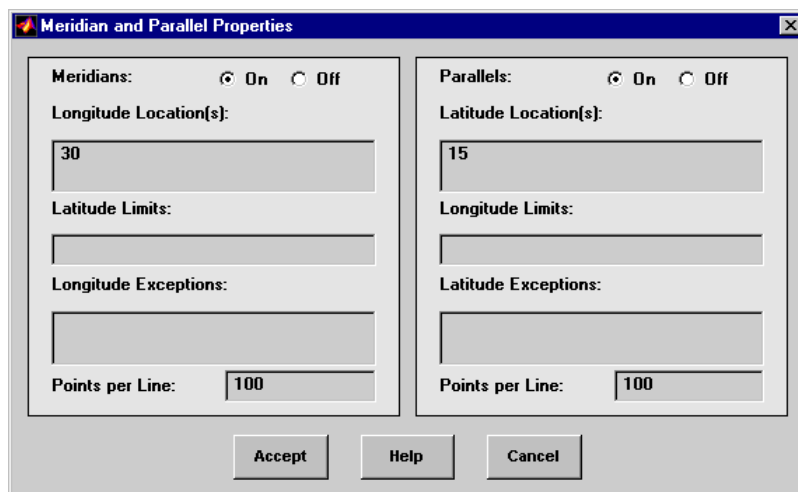
The **Meridian and Parallel Settings** button brings up the **Meridian and Parallel Properties** dialog box, which allows the properties of the meridian and parallel grid lines to be modified.

The **Accept** button accepts any modifications made to the map grid properties and returns to the Projection Control dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the map grid properties and returns to the Projection Control dialog box.

### Meridian and Parallel Properties Dialog Box

This dialog box is used to modify the settings for meridian and parallel grid lines. It is accessed via the **Meridian and Parallel Settings** button on the Map Grid Properties dialog box.



The **Meridians** selection buttons determine whether the meridian grid lines are visible when the map grid is turned on.

The **Longitude Location(s)** edit box is used to specify which meridians are to be displayed if the meridian lines are turned on. If a scalar

interval value is entered, meridian lines are displayed at that interval, starting from the Prime Meridian and proceeding in east and west directions. If a vector of values is entered, meridian lines are displayed at locations given by each element of the vector.

The **Latitude Limits** edit box is used to specify the latitude limits beyond which meridian lines do not extend. If this property is left empty, all meridian lines extend to the map latitude limits (specified by the Latitude Map Limits entry on the **Projection Control** dialog box). This entry must be a two-element vector enclosed in brackets.

The **Longitude Exceptions** edit box is used to enter specific meridians of the displayed grid that are to extend beyond the latitude limits, to the map limits. This entry is a vector of longitude values.

The **Parallels** selection buttons determine whether the parallel grid lines are visible when the map grid is turned on.

The **Latitude Location(s)** edit box is used to specify which parallels are to be displayed if the parallel lines are turned on. If a scalar interval value is entered, parallel lines are displayed at that interval, starting from the Equator and proceeding in north and south directions. If a vector of values is entered, parallel lines are displayed at locations given by each element of the vector.

The **Longitude Limits** edit box is used to specify the longitude limits beyond which parallel lines do not extend. If this property is left empty, all parallel lines extend to the map longitude limits (specified by the Longitude Map Limits entry on the **Projection Control** dialog box). This entry must be a two-element vector enclosed in brackets.

The **Latitude Exceptions** edit box is used to enter specific parallels of the displayed grid that are to extend beyond the longitude limits, to the map limits. This entry is a vector of latitude values.

The **Points per Line** edit boxes are used to enter the number of points used to plot each meridian and each parallel grid line. The default value is 100 points.

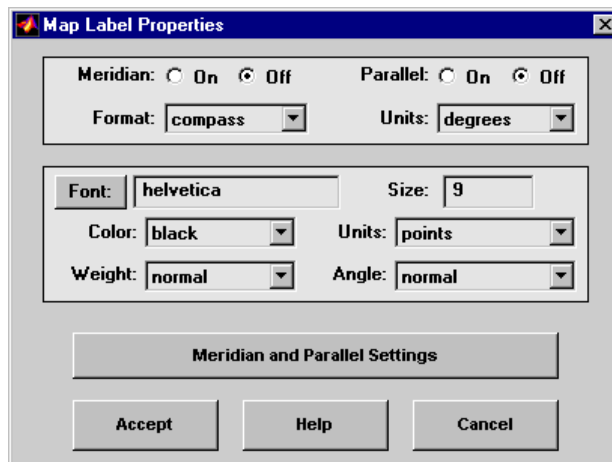
The **Accept** button accepts any modifications that have been made to the meridian and parallel grid line properties and return to the Map

Grid Properties dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the meridian and parallel grid lines and returns to the Map Grid Properties dialog box.

### Map Label Properties Dialog Box

This dialog box is used to modify the settings of the meridian and parallel labels. It is accessed via the **Label** button on the Projection Control dialog box.



The **Meridian** and **Parallel** selection buttons determine whether the meridian and parallel labels are visible.

The **Format** pull-down menu is used to specify the format of the grid labels. If compass is selected, meridian labels are appended with E for east and W for west, and parallel labels are appended with N for north and S for south. If signed is chosen, meridian labels are prefixed with + for east and - for west, and parallel labels are prefixed with + for north and - for south. If none is selected, western meridian labels and southern parallel labels are prefixed by -, but no symbol precedes eastern meridian labels and northern parallel labels.

The label **Units** pull-down menu is used to specify the angle units used to display the parallel and meridian labels. These units, used for display purposes only, need not be the same as the angle units of the map projection.

The **Font** edit box is used to specify the character font used to display the parallel and meridian labels. If the font specified does not exist on the computer, the default of Helvetica is used. Pressing the **Font** button previews the selected font.

The font **Size** edit box is used to enter an integer value that specifies the font size of the parallel and meridian labels. This value must be in the units specified by the font **Units** pull-down menu.

The font **Color** pull-down menu is used to select the color of the parallel and meridian labels. Selecting custom allows a custom RGB triple to be defined for the labels.

The font **Weight** pull-down menu is used to specify the character weight of the parallel and meridian labels.

The font **Units** pull-down menu is used to specify the units used to interpret the font size entry. When set to normalized, the value entered in the **Size** edit box is interpreted as a fraction of the height of the axes. For example, a normalized font size of 0.1 sets the label text to a height of one tenth of the axes height.

The font **Angle** pull-down menu is used to select the character slant of the parallel and meridian labels. normal specifies nonitalic font. italic and oblique specify italic font.

The **Meridian and Parallel Settings** button brings up the Meridian and Parallel Label Properties dialog box, which allows modification of properties specific to the meridian and parallel grid labels.

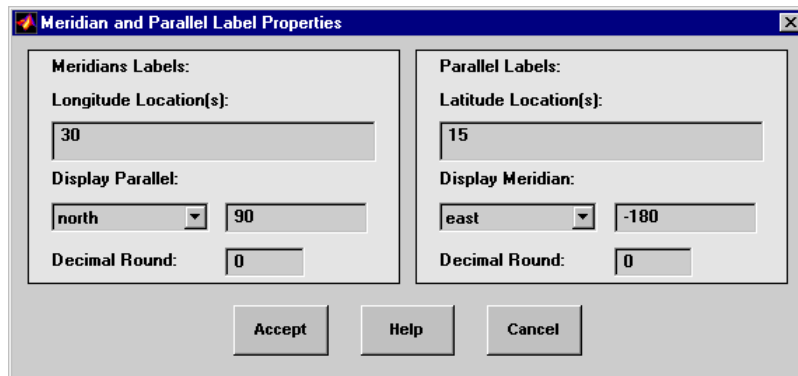
The **Accept** button accepts any modifications that have been made to the map label properties and returns to the Projection Control dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.



The **Cancel** button disregards any modifications to the map labels and returns to the **Projection Control** dialog box.

### Meridian and Parallel Label Properties Dialog Box

This dialog box is used to modify properties specific to the meridian and parallel grid labels. It is accessed via the **Meridian and Parallel Settings** button on the Map Label Properties dialog box.



The **Longitude Location(s)** edit box is used to specify which meridians are to be labeled. Meridian labels need not coincide with displayed meridian grid lines. If a scalar interval value is entered, labels are displayed at that interval, starting from the Prime Meridian and proceeding in east and west directions. If a vector of values is entered, labels are displayed at longitude locations given by each element of the vector.

The **Display Parallel** pull-down menu and edit box are used to specify the latitude location of the meridian labels. If a scalar latitude value is provided in the edit box, the meridian labels are placed at that parallel. Alternatively, the pull-down menu can be used to select a latitude location. If north is chosen, meridian labels are placed at the maximum map latitude limit. If south is chosen, meridian labels are placed at the minimum map latitude limit.

The **Latitude Location(s)** edit box is used to specify which parallels are to be labeled. Parallel labels need not coincide with displayed parallel grid lines. If a scalar interval value is entered, labels are displayed at that interval, starting from the Equator and proceeding in north and south directions. If a vector of values is entered, labels are displayed at latitude locations given by each element of the vector.

The **Display Meridian** pull-down menu and edit box are used to specify the longitude location of the parallel labels. If a scalar longitude value is provided in the edit box, the parallel labels are placed at that meridian. Alternatively, the pull-down menu can be used to specify a longitude location. If east is chosen, parallel labels are placed at the maximum map longitude limit. If west is chosen, parallel labels are placed at the minimum map longitude limit.

The **Decimal Round** edit boxes are used to specify the power of ten to which the meridian and parallel labels are rounded. For example, a value of -1 results in labels displayed to the tenths decimal place.

The **Accept** button accepts any modifications that have been made to the meridian and parallel label properties and return to the Map Label Properties dialog box. Changes are applied to the current map only when the **Apply** button on the **Projection Control** dialog box is pushed.

The **Cancel** button disregards any modifications to the meridian and parallel labels and returns to the Map Label Properties dialog box.

The **Map Geoid** edit box is used to specify the geoid (ellipsoid) definition for the current map axes. The geoid is defined by a two-element vector of the form [semimajor-axis eccentricity]. Eccentricity must be a value between 0 and 1, but not equal to 1. A nonzero eccentricity represents an ellipsoid. The default geoid is a sphere with radius 1, represented as [1 0]. If a scalar entry is provided, it is assumed to be the radius of a sphere.

The **Accept** button accepts any modifications that have been made to the map geoid and return to the Projection Control dialog box. Changes are applied to the current map only when the **Apply** button on the Projection Control dialog box is pushed.

The **Cancel** button disregards any modifications to the map geoid and returns to the Projection Control dialog box.

## See Also

axesm

# clmo-ui

---

**Purpose** GUI to clear mapped objects

**Activation**

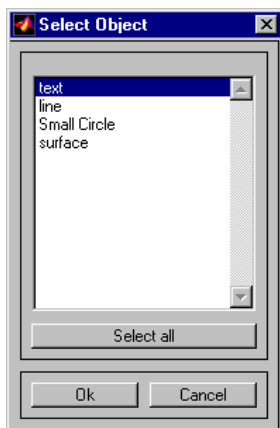
Command Line	Maptool
c.lmo	<b>Tools &gt; Delete &gt; Object</b>

**Description**

c.lmo brings up a Select Object dialog box for selecting mapped objects to delete.

**Controls**

The scroll box is used to select the desired objects from the list of mapped objects.



Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button deletes the selected objects from the map. Pushing the **Cancel** button aborts the operation.

**See Also**

c.lmo

**Purpose** Add colormap menu to figure window

**Activation**

**Command Line**

```
clrmenu
clrmenu(h)
```

**Description**

clrmenu adds a colormap menu to the current figure.  
 clrmenu(h) adds a colormap menu to the figure specified by the handle h.

**Controls**

The following choices are included on the colormap menu:

**Gray, Hsv, Hot, Pink, Cool, Bone, Jet, Copper, Spring, Summer, Autumn, Winter, Flag,** and **Prism** generate colormaps.

**Rand** is a random colormap.

**Brighten** increases the brightness.

**Darken** decreases the brightness.

**Flipud** inverts the order of the colormap entries.

**Fliplr** interchanges the red and blue components.

**Permute** permutes the colormap: red > blue, blue > green, green > red.

**Spin** spins the colormap.

**Define** allows a workspace variable to be specified for the colormap.

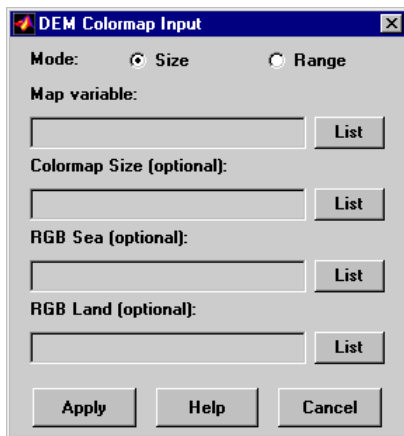
**Remember** stores the current colormap.

**Restore** reverts to the stored colormap (initially, the stored colormap is the colormap in use when clrmenu is invoked).

**Refresh** redraws the current figure window.

**Digital Elevation** activates the DEM Color Map Input dialog box. Use it to specify a colormap for a digital elevation map, and then apply the

colormap to the current figure. The number of land and sea colors in the colormap is appropriate for the maximum elevations and depths of the data grid. The dialog box is shown and described below:



The **Mode** selection buttons are used to specify whether the length of the colormap is specified or whether the altitude range increment assigned to each color is specified.

The **Map variable** edit box is used to specify the data grid containing the elevation data.

The **Color Map Size** edit box is used in Size mode. This entry defines the length of the colormap. If omitted, a default length of 64 is used. This entry must be a scalar value.

The **Altitude Range** edit box is used in Range mode. This entry defines the altitude range increment assigned to each color. If omitted, a default increment of 100 is used. This entry must be a scalar value.

The **RGB Sea** edit box is used to define colors for data with negative values. The actual sea colors of the generated colormap are interpolated from this matrix. This entry can be a matrix of any length (n by 3). The colormap matrix of the current figure can be used by entering the string 'window' in this box. The demcmap function provides default sea colors, which are used if this entry is left blank.

The **RGB Land** edit box is used to define colors for data with positive values. The actual land colors of the generated colormap are interpolated from this matrix. This entry can be a matrix of any length (n by 3). The colormap matrix of the current figure can be used by entering the string 'window' in this box. The demcmap function provides default sea colors, which are used if this entry is left blank.

Pressing the **Apply** button accepts the input data, creates the colormap, and assigns it to the current figure.

Pressing the **Cancel** button disregards any input data and closes the DEM Color Map Input dialog box.

**See Also**

colorm, demcmap

# colorm

**Purpose** Create index map colormaps

## Activation

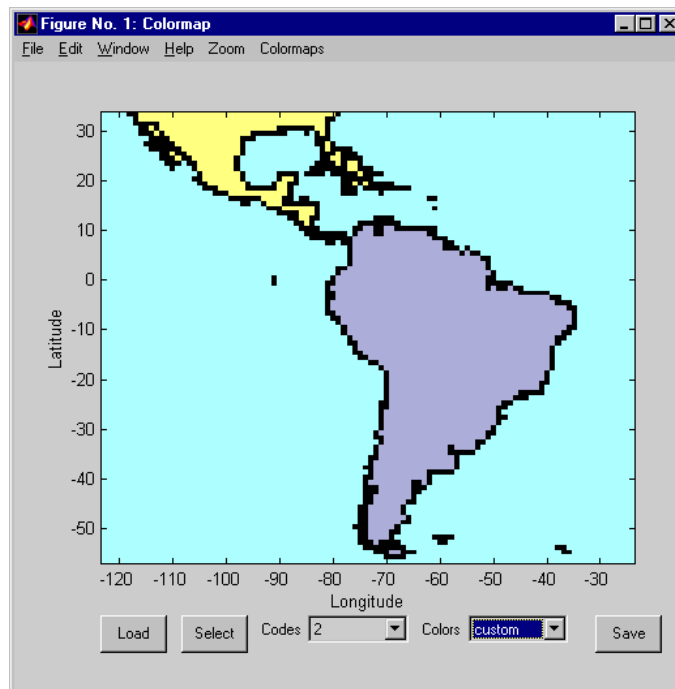
### Command Line

```
colorm(datagrid,refvec)
```

## Description

`colorm(datagrid,refvec)` displays the data grid in a new figure window and allows a colormap to be edited and saved to a new variable. `datagrid` and `refvec` are the data grid and the referencing vector of the surface. `map` must have positive index values into the colormap.

## Controls





The **colorm** tool displays the surface map data in a new figure window with the current colormap. **Zoom** and **Colormaps** menus are activated for that figure.

The **Zoom On/Off** menu toggles the panzoom box on and off. The box can be moved by clicking the new location or by dragging the box to the new location. The box size can be increased or decreased by dragging a corner of the box. Pressing the Return key or double-clicking in the center of the box zooms in.

The **Colormaps** menu provided a variety of colormap options that can be applied to the map. See `clrmenu` in this guide for a description of the **Colormaps** menu options.

The **Load** button activates a dialog box, used to specify a colormap variable to be applied to the displayed surface map. This colormap can then be edited and saved.

The **Select** button activates the mouse cursor and allows a point on the map to be selected. The value of that point then appears in the **Codes** pull-down menu. The color of the selected point appears in the **Color** pull-down menu and can then be edited.

The **Codes** pull-down menu is used to select a particular value in the data grid. The color associated with that value then appears in the **Color** pull-down menu and can be edited.

The **Color** pull-down menu is used to select a particular color to assign to the value currently displayed in the Codes pull-down menu. A custom color can be defined by selecting the custom option. This brings up a custom color interface with which an RGB triple can be selected.

The **Save** button is used to save the modified colormap to the workspace. A dialog box appears in which the colormap variable name is entered.

## See Also

`encodem`, `getseeds`, `maptrim`, `panzoom`, `seedm`

# demdataui

---

**Purpose** UI for selecting digital elevation data

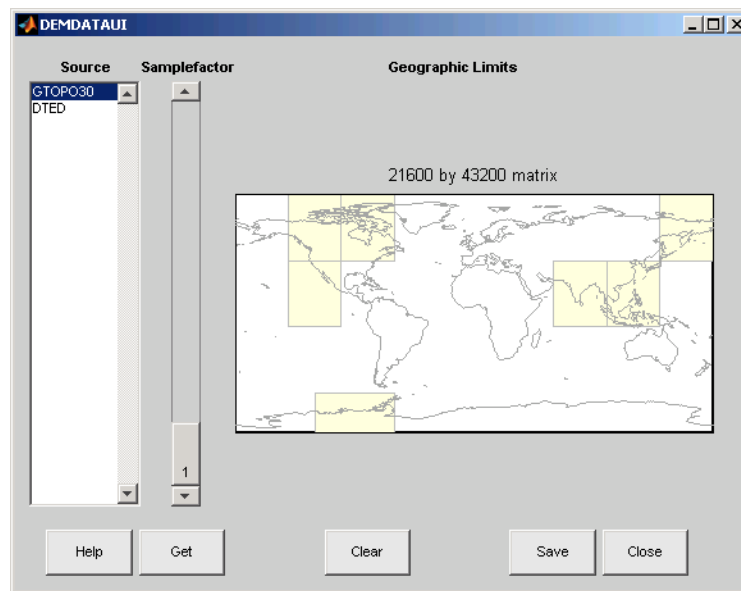
**Activation** demdataui

**Description** demdataui is a graphical user interface to extract digital elevation map data from a number of external data files.

The demdataui panel lets you read data from a variety of high-resolution digital elevation maps (DEMs). These DEMs range in resolution from about 10 kilometers to 100 meters or less. The data files are available over the Internet at no cost, or (in some cases) on CD-ROMs for varying fees. demdataui reads ETOPO5, TerrainBase, GTOPO30, GLOBE, satellite bathymetry, and DTED data. See the links under See Also for more information on these data sets. demdataui looks for these geospatial data files on the MATLAB path and, for some operating systems, on CD-ROM disks.

You use the list to select the source of data and the map to select the region of interest. When you click the **Get** button, data is extracted and displayed on the map. Use the **Save** button to save the data in a MAT-file or to the base workspace for later display. The **Close** button closes the window.

## Controls



### The Map

The map controls the geographic extent of the data to be extracted. demdataui extracts data for areas currently visible on the map. Use the mouse to zoom in or out to the area of interest. See [zoom](#) for more on zooming.

Some data sources divide the world up into tiles. When extracting, data is concatenated across all visible tiles. The map shows the tiles in light yellow with light gray edges. When data resolution is high, extracting data for large area can take much time and memory. An approximate count of the number of points is shown above the map. Use the **Samplefactor** slider to reduce the amount of data.

### The List

The list controls the source of data to be extracted. Click a name to see the geographic coverage in light yellow. The sources list shows the data sources found when demdataui started.

demdataui searches for data files on the MATLAB path. On some computers, demdataui also checks for data files on the root level of letter drives. demdataui looks for the following data: etopo5: new\_etopo5.bin or etopo5.northern.bat and etopo5.southern.bat files. tbase: tbase.bin file. satbath: topo\_6.2.img file. gtopo30: a directory that contains subdirectories with the data files. For example, demdataui would detect gtopo30 data if a directory on the path contained the directories E060S10 and E100S10, each of which holds the uncompressed data files. globedem: a directory that contains data files and in the subdirectory "/esri/hdr" the "\*.hdr" header files. dted: a directory that has a subdirectory named DTED. The contents of the DTED directory are more subdirectories organized by longitude and, below that, the DTED data files for each latitude tile. See the help for functions with the data source names for more on the data attributes and internet locations.

## The Samplefactor Slider

The **Sample Factor** slider allows you to reduce the density of the data. A sample factor of 2 returns every second point. The current sample factor is shown on the slider.

## The Get Button

The **Get** button reads the currently selected data and displays it on the map. Press the standard interrupt key combination for your platform to interrupt the process.

## The Clear Button

The **Clear** button removes any previously read data from the map.

## The Save Button

The **Save** button saves the currently displayed data to a MAT-file or the base workspace. If you choose to save to a file, you will be prompted for a file name and location. If you choose to save to the base workspace, you can choose the variable name under which the data will be stored. The results are stored as a geographic data structure. Use `load` and `displaym` to redisplay the data from a file on a map axes. To display the data in the base workspace, use `displaym`. To gain

access to the data matrices, subscript into the structure (for example, `datagrid = demdata(1).map; refvec = demdata(1).maplegend`). Use `worldmap` to create easy displays of the elevation data (for example, `worldmap(datagrid,refvec)`). Use `meshm` to add regular data grids to existing displays, or `surfm` or a similar function for geolocated data grids (for example, `meshm(datagrid,refvec)` or `surfm(latgrat,longrat,z)`).

## The Close Button

The **Close** button closes the `demdataui` panel.

## See Also

`etopo`, `tbase`, `gtopo30`, `globedem`, `dted`, `satbath`, `vmap0ui`

# handlem-ui

---

**Purpose** GUI for handles of specified mapped objects

**Activation**

**Command Line**

h = handlem

h = handlem('prompt')

**Description**

h = handlem brings up a Select Object dialog box, which lists all currently displayed objects. Objects can be selected and their handles returned.

h = handlem('prompt') brings up a Specify Object dialog box, which allows greater control of object selection.

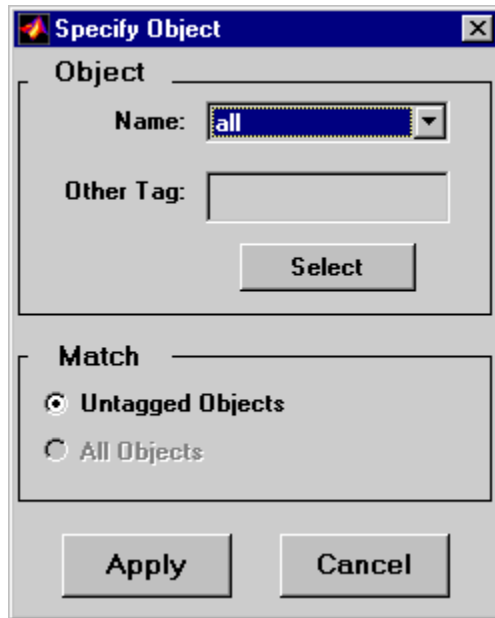
**Controls**

Select Object Dialog Box



The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button returns the object handles in the variable `h`. Pushing the **Cancel** button aborts the operation.

Specify Object Dialog Box



The **Object** Controls are used to select an object type or tag. The **Name** pull-down menu is used to select from a list of predefined object strings. The **Other Tag** edit box is used to specify an object tag not listed in the **Name** pull-down menu. Pushing the **Select** button brings up the Select Object dialog box, which shows only the currently displayed objects for selection.

The **Match** Controls are used when a Handle Graphics object type (image, line, surface, patch, or text) is specified. The **Untagged Objects** selection button is used to return the handles of only those objects with empty tag properties. The **All Objects** selection button is

# handlem-ui

---

used to return all object handles of the specified type, regardless of whether they are tagged.

Pushing the **Apply** button returns the handles of the specified objects. Pushing the **Cancel** button aborts the operation.

## See Also

handlem



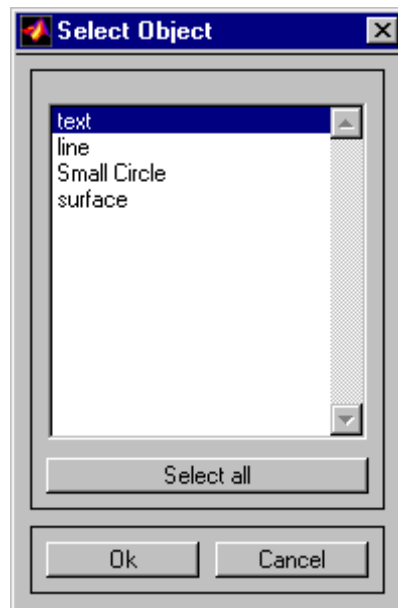
**Purpose** Hide specified mapped objects

**Activation**

Command Line	Maptool
hidem	Tools > Hide > Object

**Description**

hidem brings up a Select Object dialog box for selecting mapped objects to hide (Visible property set to 'off').

**Controls**

The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button changes the Visible property of the selected objects to 'off'. Pushing the **Cancel** button aborts the operation without changing any properties of the selected objects.

**See Also**

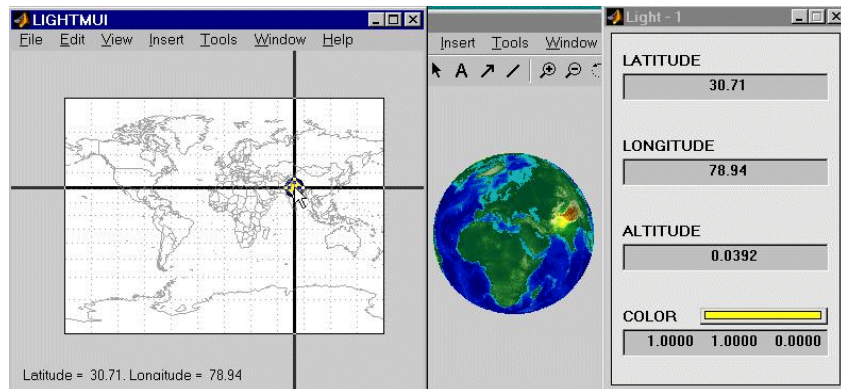
hidem

# lightmui

**Purpose** Control position of lights on globe or 3-D map

**Syntax** `lightmui(hax)`

**Description** `lightmui(hax)` creates a GUI to control the position of lights on a globe or 3-D map in map axes `hax`. You can control the position of lights by clicking and dragging the icon or by dialog boxes. Right-click the appropriate icon in the GUI to invoke the corresponding dialog box. You can change the light color by entering the RGB components manually or by clicking the pushbutton.



**See Also** `lightm`

**Purpose** Add menu activated tools to map figure

**Activation****Command Line**

```
maptool(PropertyName,PropertyValue)
```

```
maptool(ProjectionFile,...)
```

```
h = maptool(...)
```

**Description**

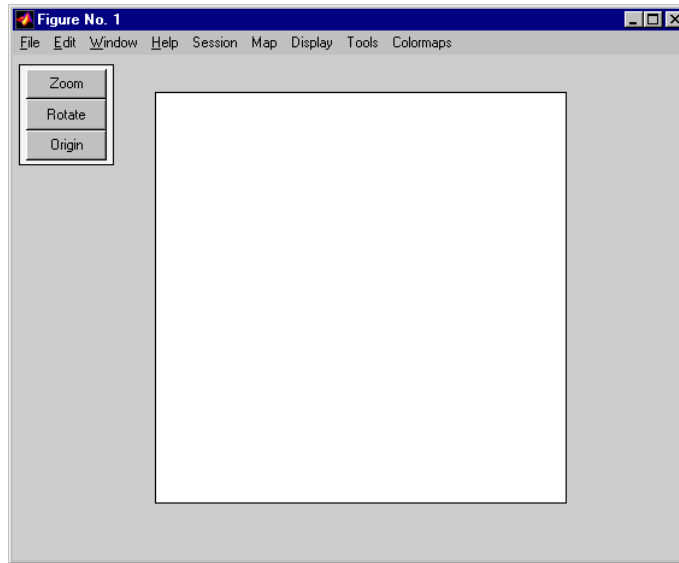
maptool creates a figure window with a map axes and activates the Projection Control dialog box for defining map projection and display properties. The figure window features a special menu bar that provides access to most of Mapping Toolbox GUIs.

maptool(*PropertyName*,*PropertyValue*,...) creates a figure window with a map axes defined by the supplied map properties. The MapProjection property must be the first input pair. maptool supports the same map properties as axesm.

maptool(*ProjectionFile*,*PropertyName*, *PropertyValue*,...) allows for the omission of the MapProjection property name. *ProjectionFile* must be the identifying string of an available map projection.

h = maptool(...) returns a two-element vector containing the handle of the maptool figure window and the handle of the map axes.

## Controls



### Session Menu

The **Load** option is used to load workspace data. Select from the workspace names provided, or use the **Specify Workspace** option to enter a different workspace.

The **Layers** option is used to load a map layers workspace and activate the `mlayers` tool. Select from the workspace names provided, or use the **Other** option to enter a different workspace. Choosing **Workspace** loads all structure variables in the current workspace.

The **Renderer** option is used to set the renderer for the maptool figure window. The Figure Renderer dialog box is activated when this option is selected.

The **Variables** option is used to view the current workspace variables.

The **Command** option brings up the Workspace Commands dialog box for entering commands to operate on the current workspace.

The **Clear** option is used to clear variables and functions from memory.

## Map Menu

The **Lines** option activates the Line Map Input dialog box for projecting two- and three-dimensional line objects onto the map axes.

The **Patches** option activates the Patch Map Input dialog box for projecting patch objects onto the map axes.

The **Regular Surfaces** option activates the Mesh Map Input dialog box for projecting a regular data grid onto a graticule projected onto the map axes.

The **General Surfaces** option activates the Surface Map Input dialog box for projecting a geolocated data grid onto the map axes.

The **Comet** option activates the Comet Map Input dialog box for a projecting two- or three-dimensional comet plot onto the map axes.

The **Contours** option activates the Contour Map Input dialog box for projecting a two- or three-dimensional contour plot onto the map axes.

The **Quiver 2D** option activates the Quiver Map Input dialog box for projecting a two-dimensional quiver plot onto the map axes.

The **Quiver 3D** option activates the Quiver3 Map Input dialog box for projecting a three-dimensional quiver plot onto the map axes.

The **Stem** option activates the Stem Map Input dialog box for projecting a stem plot onto the map axes.

The **Scatter** option activates the Scatter Map Input dialog box for projecting a scatter plot onto the map axes.

The **Text** option activates the Text Map Input dialog box for projecting text objects onto the map axes.

The **Light** option activates the Light Map Input dialog box for projecting light objects onto the map axes.

## Display Menu

The **Projection** option activates the Projection Control dialog box for editing map projection properties and map display settings.

The **Graticule** option is used to view and edit the graticule size for surface maps.

The **Legend** option is used to display a contour map legend.

The **Frame** option is used to toggle the map frame on and off.

The **Grid** option is used to toggle the map grid on and off.

The **Meridian Labels** option is used to toggle the meridian grid labels on and off.

The **Parallel Labels** option is used to toggle the parallel grid labels on and off.

The **Tracks** option activates the Define Tracks input box for calculating and displaying Great Circle and Rhumb Line tracks on the map axes.

The **Small Circles** option activates the Define Small Circles input box for calculating and displaying small circles on the map axes.

The **Surface Distances** option activates the Surface Distance dialog box for distance, azimuth, and reckoning calculations.

## Tools Menu

The **Hide** option is used to hide the mouse tool buttons.

The **Off** option is used to turn off the current mouse tool.

The **Zoom Tool** option is used to toggle Panzoom (panzoom) mode on and off. It is used for zooming in on a two-dimensional map display.

The **Set Limits** option is used to define the zoom out limits to the current settings on the axes.

The **Full View** option is used to zoom out to the current axes limit settings.

The **Rotate** option is used to toggle Rotate 3-D (rotate3d) mode on and off. Rotate 3-D mode is used to interactively rotate the view of a three-dimensional plot.

The **Origin** option is used to toggle Origin (originui) mode on and off. Origin mode is used to interactively modify the map origin.

The **2D View** option is used to set the default two-dimensional view (azimuth=0, elevation=90).

The **Objects** option activates the Object Sets dialog box, which allows for property manipulation of objects displayed on the map axes.

The **Edit** option activates the MATLAB Property Editor to manipulate properties of a plotted object. Choose from the **Current Object** or **Last Object** options, or choose the **Object** option to activate the Select Object dialog box.

The **Show** option is used to set the `Visible` property of mapped objects to 'on'. The **All** option shows all currently mapped objects. The **Object** option activates the Select Object dialog box.

The **Hide** option is used to set the `Visible` property of mapped objects to 'off'. Choose from the **All** or **Map** options, or choose the **Object** option to activate the Select Object dialog box.

The **Delete** option is used to clear the selected objects. The **All** option clears the current map, frame, and grid lines. The map definition is left in the axes definition. The **Map** option clears the current map, deleting objects plotted on the map but leaving the frame and grid lines displayed. The **Object** option activates the Select Object dialog box.

The **Axes** option is used to manipulate the MATLAB Cartesian axes. The **Show** option shows this axes, the **Hide** option hides this axes, and the **Color** option allows for custom color selection for this axes.

## Colormaps Menu

The **Colormaps** menu allows for manipulation of the colormap for the current figure. See the `clrmenu` reference page for details on the **Colormaps** menu options.

The **Zoom** button toggles Zoom mode on and off. Zoom mode is used for zooming in on a two-dimensional map display.

The **Rotate** button toggles Rotate 3-D mode on and off. Rotate 3-D mode is used to interactively rotate the view of a three-dimensional plot.

The **Origin** button toggles Origin mode on and off. Origin mode is used to interactively modify the map origin.

## See Also

`axesm`



**Purpose** Interactively trim and convert map data from vector to raster format

**Activation****Command Line**

```
maptrim(lat,lon)
maptrim(lat,lon,linespec)
maptrim(datagrid,refvec)
maptrim(datagrid,refvec,PropertyName,PropertyValue,...)
```

**Description**

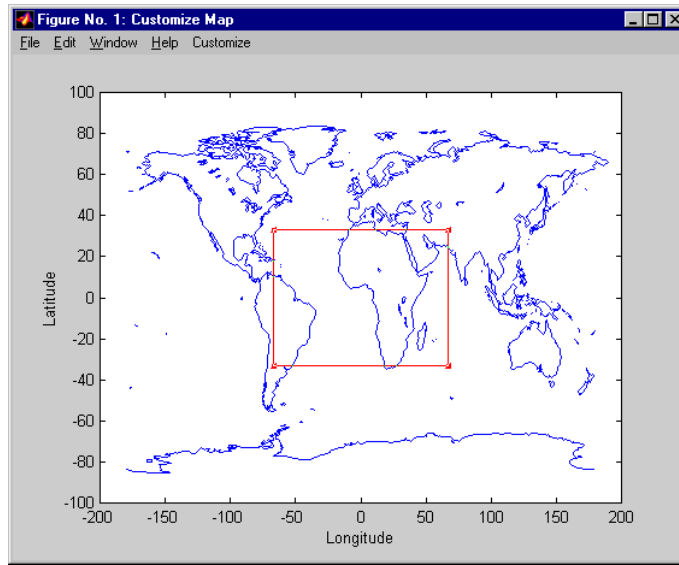
`maptrim(lat,lon)` displays the supplied map data in a new figure window and allows a region of the map to be selected and saved in the workspace. `lat` and `lon` must be vector map data. The output can be line, patch, or regular surface (matrix) data. If patch map output is selected, the inputs `lat` and `lon` must originally be patch map data.

`maptrim(lat,lon,linespec)` displays the supplied map data using the `linespec` string.

`maptrim(datagrid,refvec)` displays data grid data in a new figure window and allows a subset of this map to be selected and saved. The output is regular surface data.

`maptrim(datagrid,refvec,PropertyName,PropertyValue)` displays the data grid using the surface properties provided. The object `Tag`, `EdgeColor`, and `UserData` properties cannot be set.

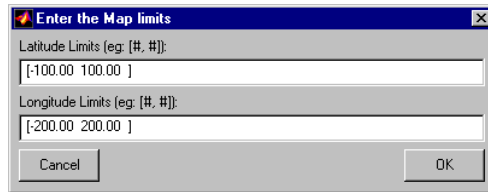
## Controls



The `maptrim` tool displays the supplied map data in a new figure window and activates a **Customize** menu for that figure. The **Customize** menu has three menu options: **Zoom On/Off**, **Limits**, and **Save As**.

The **Zoom On/Off** menu option toggles the panzoom box on and off. The box can be moved by clicking the new location or by dragging the box to the new location. The box size can be increased or decreased by dragging a corner of the box. Pressing the Return key or double-clicking in the center of the box zooms in.

The **Limits** menu option activates the Enter Map Limits dialog box, which is used to enter the latitude and longitude limits of the desired map subset. These entries are two-element vectors, enclosed in brackets. Pressing the **OK** button zooms in to the new limits. Pressing the **Cancel** button disregards the new limits and returns to the map display.



The **Save As** menu option is used to specify the variable names in which to save the map data subset. To save line and patch data, enter the new latitude and longitude variable names, along with the map resolution. For surface data, enter the new map and referencing vector variable names, along with the scale of the map. Latitude and longitude limits are optional.

## See Also

maptrim1, maptrim, maptrims, panzoom

# mayers

---

**Purpose** Interactive display of Version 1 geographic data structures

**Activation**

Command Line	Maptool
<code>mayers(workspace)</code>	<b>Session &gt; Layers</b>
<code>mayers(workspace,h)</code>	
<code>mayers(cellarray)</code>	
<code>mayers(cellarray,h)</code>	

**Description**

The `mayers` tool activates a dialog box for the specified geographic data structure `workspace`, which enables display and manipulation of the map objects that it comprises.

`mayers(workspace)` associates the geographic data structures, which in this context are also called map layers, in the `workspace` MAT-file with the current map axes. The geographic data structure variables are accessible only through the `mayers` tool, and not through the base workspace. `workspace` must be a string.

`mayers(workspace,h)` assigns the layers in `workspace` to the map axes indicated by the handle `h`.

`mayers(cellarray)` associates the layers specified by `cellarray` with the current map axes. `cellarray` must be of size `n` by 2. Each row of `cellarray` represents a map layer. The first column of `cellarray` contains the layer structure, and the second column contains the name of the layer structure. Such a cell array can be generated from data in the current workspace with the function `rootlayr`. In this case, the calling sequence would be `rootlayr; mayers(ans)`.

`mayers(cellarray,h)` assigns the layers specified by `cellarray` to the map axes specified by the handle `h`.

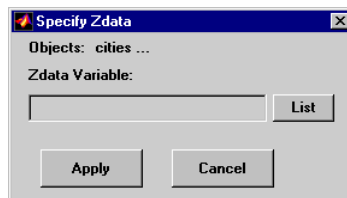
## Controls



The scrollable list box displays all of the map layers currently associated with the map axes. An asterisk next to the layer name indicates that the layer is currently visible. An h next to the layer name indicates a layer that is plotted, but currently hidden.

The **Plot** button plots the selected map layer. Once the selected layer is plotted, the button toggles between **Hide** and **Show**, to turn the Visible property of the plotted objects to 'off' and 'on', respectively.

The **Zdata** button activates the Specify Zdata dialog box, which is used to enter the workspace variable containing the ZData for the selected map layer. Pressing the **List** button produces a list of all current workspace variables, from which the ZData variable can be selected. This entry can also be a scalar.



The **Highlight** button is used to toggle the selected map layer between highlighted and normal display.

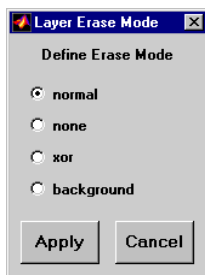
The **Members** button brings up a list of members of the selected map layer. Members of a layer are defined by their Tag property.

# mlayers

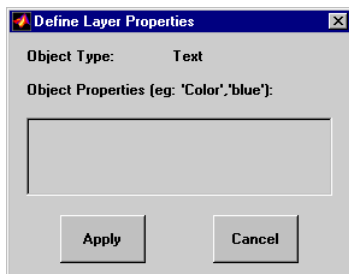
---

The **Delete** button deletes the selected map layer from the map.

The **Emode** button activates the Layer Erase Mode dialog box, which is used to specify the erase mode of the selected map layer.



The **Property** button activates the Define Layer Properties dialog box, which is used to specify or change properties of all objects in the selected map layer. String entries must be enclosed in single quotes.



The **Purge** button deletes the selected map layer from the mlayers tool. Selecting **Yes** from the Confirm Purge dialog box deletes the map layer from both the mlayers tool and the map display. Selecting **Data Only** from the Confirm Purge dialog box deletes the map layer from the mlayers tool, while retaining the plotted object on the map display.

## See Also

objects, rootlayr

**Purpose** Manipulate object sets displayed on map axes

## Activation

Command Line	Maptool
objects	Tools > Objects
objects(h)	

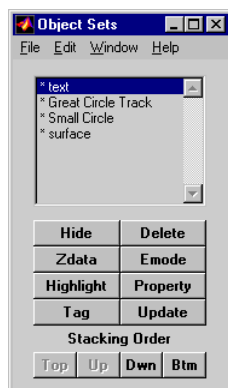
## Description

An object set is defined as all objects with identical tags. If no tags are supplied, object sets are defined by object type.

objects allows manipulation of the object sets on the current map axes.

objects(h) allows manipulation of the objects set on the map axes specified by the handle h.

## Controls



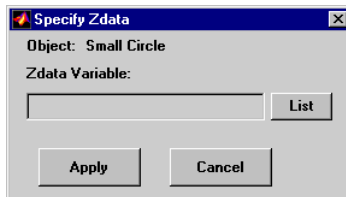
The scrollable list box displays all of the object sets associated with the map axes. An asterisk next to an object set name indicates that the object set is currently visible. An h next to an object set name indicates an object set that is plotted, but currently hidden. The order shown in the list indicates the stacking order of objects within the same plane.

# mobjects

---

The **Hide/Show** button toggles the `Visible` property of the selected object set to 'off' and 'on', respectively, depending on the current `Visible` status.

The **Zdata** button activates the Specify Zdata dialog box, which is used to enter the workspace variable containing the ZData. The ZData property is used to specify the plane in which the selected object set is drawn. Pressing the **List** button produces a list of all current workspace variables, from which the ZData variable can be selected. Alternatively, a scalar value can be entered instead of a variable.

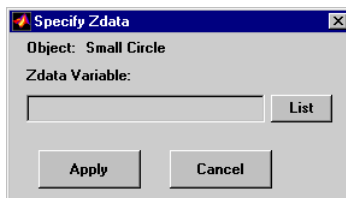


The **Highlight** button highlights all objects belonging to the selected object set.

The **Tag** button brings up an Edit Tag dialog box, which allows the tag of all members of the selected object set to be modified.

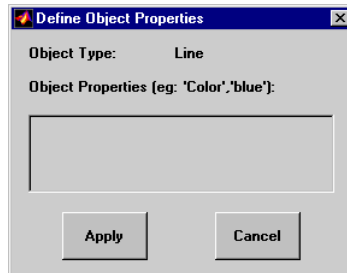
The **Delete** button clears all objects belonging to the selected object set from the map. The cleared object set remains associated with the map axes.

The **Emode** button activates the Object Erase Mode dialog box, which is used to specify the erase mode of the selected object set.





The **Property** button activates the Define Object Properties dialog box, which is used to specify additional properties of all objects in the selected object set. String entries must be enclosed in single quotes.



The **Update** button updates the list box display with current objects sets.

The **Stacking Order** buttons are used to modify the drawing order of the selected object set in relation to other plotted object sets in the same plane. Objects drawn first appear at the bottom of the stack, and objects drawn last appear at the top of the stack. The **Top** button places the selected object set above all other object sets in its plane. The **Up** and **Dwn** buttons move the selected object set up and down one place in the stacking order, respectively. The **Btm** button places the selected object set below all other object sets in its plane. Note that the ZData property overrides stacking order, i.e., if an object is at the top of the stacking order for its plane, it can still be covered by an object drawn in a higher plane.

## See Also

mlayers

# originui

---

**Purpose** Interactively modify map origin

**Activation**

Command Line	Maptool
originui	<b>Tools &gt; Origin (menu) &gt; Origin(button)</b>
originui on	
originui off	

**Description**

originui provides a tool to modify the origin of a displayed map projection. A marker (dot) is displayed where the origin is currently located. This dot can be moved and the map reprojected with the identified point as the new origin.

originui automatically toggles the current axes into a mode where only actions recognized by originui are executed. Upon exit of this mode, all prior ButtonDown functions are restored to the current axes and its children.

originui on activates origin tool. originui off e-activates the tool. originui will toggle between these two states.

**Controls**

**Keystrokes**

originui recognizes the following keystrokes. **Enter** (or **Return**) will reproject the map with the identified origin and remain in the originui mode. **Delete** and **Escape** will exit the origin mode (same as originui off). **N,S,E,W** keys move the marker North, South, East or West by 10.0 degrees for each keystroke. **n,s,e,w** keys move the marker in the respective directions by 1 degree per keystroke.

**Mouse Actions**

originui recognizes the following mouse actions when the cursor is on the origin marker.

- Single-click and hold moves the origin marker. Double-click the marker reprojects the map with the specified map origin and remains in the origin mode (same as originui **Return**).
- Extended-click moves the marker along the Cartesian X or Y direction only (depending on the direction of greatest movement).
- Alternate-click exits the origin tool (same as originui off).

## Macintosh Key Mapping

- Extend-click: **Shift**+click mouse button
- Alternate-click: **Option**+click mouse button

## MS-Windows Key Mapping

- Extend-click: **Shift**+click left button or both buttons
- Alternate-click: **Ctrl**+click left button or right button

## X-Windows Key Mapping

- Extend-click: **Shift**+click left button or middle button
- Alternate-click: **Ctrl**+click left button or right button

## See Also

axesm, setm

# panzoom

---

**Purpose** Pan and zoom on map axes

**Activation**

Command Line	Maptool
panzoom	<b>Tools &gt; Zoom Tool (menu) &gt; Zoom (button)</b>
panzoom on	
panzoom off	
panzoom setlimits	
panzoom out	
panzoom fullview	

**Description**

panzoom toggles the pan and zoom tool on and off.

panzoom on activates the pan and zoom tool.

panzoom off deactivates the pan and zoom tool.

panzoom setlimits sets the zoom out limits to the current settings on the map axes.

panzoom out zooms out to the current map axes limit settings.

panzoom fullview resets the axes to their full view range and resets the pan and zoom tool with these settings.

The pan and zoom tool provides an interactive means of defining zoom limits on a two-dimensional map display. A box that can be resized and moved appears on the map display and is used to define the zoom area. The box cannot be moved beyond the current axes limits.

**Controls**

**Mouse Interaction**

With the cursor inside the zoom box, a single-click and drag moves the box. The zoom box can be resized by dragging the corners of the box. A double-click in the center of the box zooms in to the current boundaries of the box. A single-click outside the zoom box moves the box to that

location. An extend-click inside or outside of the zoom box zooms out by a factor of two. Alternate-click exits the pan and zoom tool.

### **Keyboard Interaction**

The following keyboard interaction is enabled if the figure containing the map axes is made the active window.

Pressing the **Return** key sets the axes to the current zoom box and remains in pan and zoom mode. The **Enter** key sets the axes to the current zoom box and exits pan and zoom mode. Pressing the **Esc** or **Delete** keys exits pan and zoom mode.

### **See Also**

zoom

# parallelui

---

**Purpose** Interactively modify map parallels

**Activation**

Command Line	Maptool
<code>parallelui</code>	<b>Tools &gt; Parallels (menu)</b>
<code>parallelui on</code>	
<code>parallelui off</code>	

**Description**

`parallelui` toggles the parallel tool on and off.

`parallelui on` activates the parallel tool

`parallelui off` deactivates the parallel tool

The `parallelui` GUI provides a tool to modify the standard parallels of a displayed map projection. One or two red lines are displayed where the standard parallels are currently located. The parallel lines can be dragged to new locations, and the map reprojected with the locations of the parallel lines as the new standard parallels.

**Controls**

Mouse Interaction

A single-click-and-drag moves the parallel lines. A double-click on one of the standard parallels reprojects the map using the new parallel locations.

**See Also**

`axesm`, `setm`

## Purpose

GUIs to edit properties of mapped objects

## Activation

map display: Alternate-click mapped object (for Click-and-Drag Property Editor)

In plot edit mode, double-click mapped object (to obtain MATLAB Property Editor; click the **More Properties...** button to open the Property Inspector)

maptool: **Tools > Edit Plot** menu item (for MATLAB Property Editor)

## Description

Alternate (e.g., **Ctrl**+clicking a mapped object activates a property editor, which allows modification of some basic properties of the object through simple mouse clicks and drags. The objects supported by this editor are map axes, lines, text, patches, and surfaces, and the properties supported for each object type are shown below.

In plot edit mode, double-clicking a mapped object activates the MATLAB Property Editor for that object. From the Property Editor you can launch the Property Inspector, a GUI that lists the properties and values of the selected object and allows you to modify them.

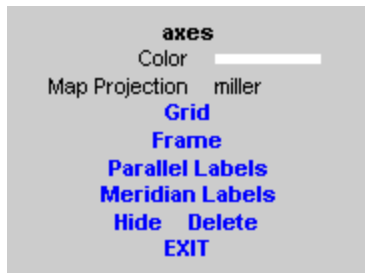
## Controls

### Click-and-Drag Property Editor

The Click-and-Drag editor lists object properties and values. The object tag appears at the top of the editor. Property names and values that appear in blue are toggles. For example, clicking **Frame** in the axes editor toggles the value of the Frame property between 'on' and 'off'.

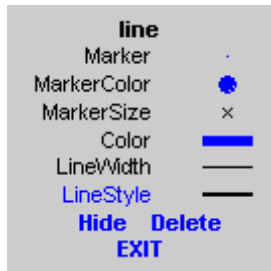
# property editors

---



## Click-and-Drag Editor for a map axes

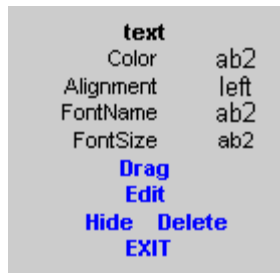
Property values that appear on the right side of the editor box are modified by clicking and dragging. For example, to change the MarkerColor property of a line object, click and hold the dot next to **MarkerColor**, and drag the cursor until the dot appears in the desired color.



## Click-and-Drag Editor for a line object

The **Drag** control in the text editor is used to reposition the text string. In drag mode, use the mouse to move the text to a new location, and click to reposition the text. The **Edit** control in the text editor activates a **Text Edit** window, which is used to modify text.





## Click-and-Drag Editor for a text object

The **Marker** property name in the patch editor is used to toggle the marker on and off. The property value to the right of **Marker** can be modified by clicking and dragging until the desired marker symbol appears.



## Click-and-Drag Editor for a patch object

The **Graticule** control on the surface editor activates a Graticule Mesh dialog box, which is used to alter the size of the graticule.

To move the property editor around the figure window, hold down the **Shift** key while dragging the editor box. Alternate-clicking the background of the property editor closes the **Click-and-Drag** editing session.

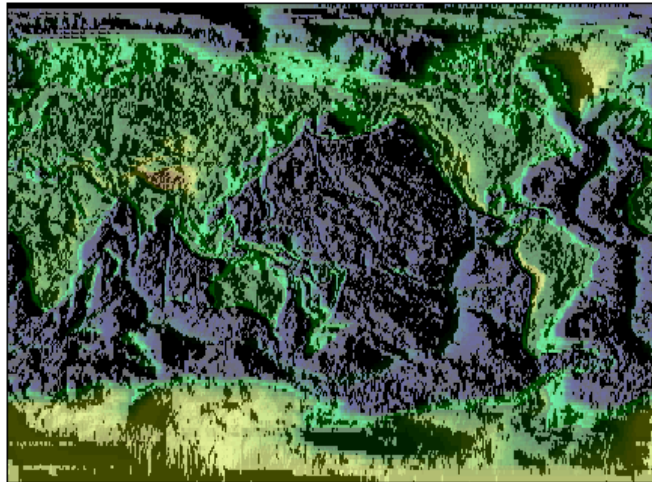
## Guide Property Editor

The MATLAB Property Inspector (the `inspect` function) allows you to view and modify property values for most properties of the selected

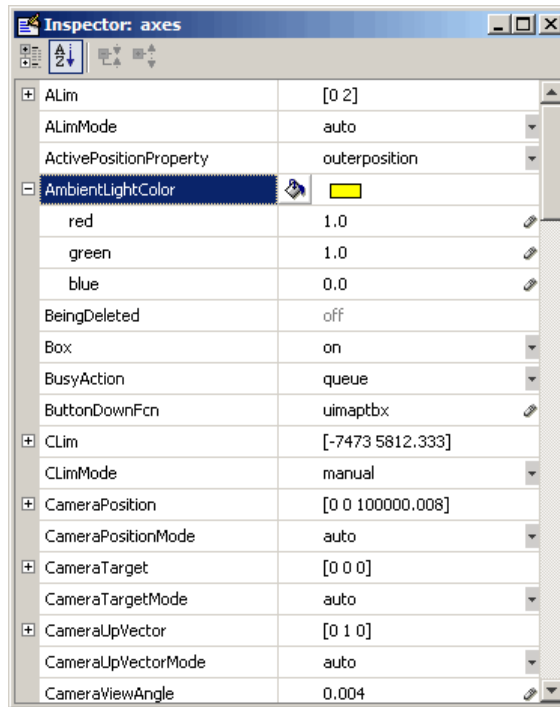
# property editors

---

object. Use it to expand and collapse the hierarchy of objects, showing an object's parents and children. A plus sign (+) before a property indicates that it can be expanded to show its components, for example the axes AmbientLightColor applied to the surface object displayed below. A minus sign (-) before an object indicates an object can be collapsed to hide its components. To activate the Object Browser, check the **Show Object Browser** check box. The **Property List** shows all the property names of the selected object and their current values. To activate the **Property List**, check the **Show Property List** check box. To change a property value, use the edit boxes above the Property List. Pressing the **Close** button closes the Guide Property Editor and applies the property modifications to the object.



**A lit surface object in a map axes**



## Property Inspector view of axes object

**See Also** `propedit`, `inspect`, `uimaptbx`

**Purpose** GUI to interactively perform data queries

## Activation

### Command Line

```
qrydata(cellarray)
qrydata(titlestr,cellarray)
qrydata(h,cellarray)
qrydata(h,titlestr,cellarray)
qrydata(...,cellarray1,cellarray2,...)
```

## Description

A data query is used to obtain the data corresponding to a particular (x,y) or (lat,lon) point on a standard or map axes.

`qrydata(cellarray)` activates a data query dialog box for interactive queries of the data set specified by `cellarray` (described below).

`qrydata` can be used on a standard axes or a map axes. (x,y) or (lat,lon) coordinates are entered in the dialog box, and the data corresponding to these coordinates is then displayed.

`qrydata(titlestr,cellarray)` uses the string `titlestr` as the title of the query dialog box.

`qrydata(h,cellarray)` and `qrydata(h,titlestr,cellarray)` associate the data queries with the axes specified by the handle `h`, which in turn allows the input coordinates to be specified by clicking the axes.

The input `cellarray` is used to define the data set and the query. The first cell must contain the string used to label the data display line. The second cell must contain the type of query operation, either a predefined operation or a valid user-defined function name. This input must be a string. The predefined query operations are 'matrix', 'vector', 'mapmatrix', and 'mapvector'.

The 'matrix' query uses the MATLAB `interp2` function to find the value of the matrix `Z` at the input (x,y) point. The format of the `cellarray` input for this query is:

{ 'label', 'matrix', X, Y, Z, *method* }. X and Y are matrices specifying the points at which the data Z is given. The rows and columns of X and Y must be monotonic. *method* is an optional argument that specifies the interpolation method. Possible *method* strings are 'nearest', 'linear', or 'cubic'. The default is 'nearest'.

The 'vector' query uses the MATLAB `interp2` function to find the value of the matrix Z at the input (x,y) point, then uses that value as an index to a data vector. The value of the data vector at that index is returned by the query. The format of cellarray for this type of query is: { 'label', 'vector', X, Y, Z, vector }. X and Y are matrices specifying the points at which the data Z is given. The rows and columns of X and Y must be monotonic. vector is the data vector.

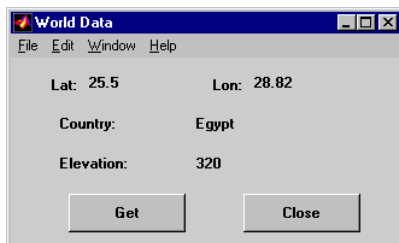
The 'mapmatrix' query interpolates to find the value of the map at the input (lat,lon) point. The format of cellarray for this query is: { 'label', 'mapmatrix', datagrid, refvec, *method* }. datagrid and refvec are the data grid and the corresponding referencing vector. *method* is an optional argument that specifies the interpolation method. Possible *method* strings are 'nearest', 'linear', or 'cubic'. The default is 'nearest'.

The 'mapvector' query interpolates to find the value of the map at the input (lat,lon) point, then uses that value as an index to a data vector. The value of the vector at that index is returned by the query. The format of cellarray for this type of query is { 'label', 'mapvector', datagrid, refvec, vector }. datagrid and refvec are the data grid and the corresponding referencing vector. vector is the data vector.

User-defined query operations allow for functional operations using the input (x,y) or (lat,lon) coordinates. The format of cellarray for this type of query is { 'label', *function*, other arguments... } where the other arguments are the remaining elements of cellarray as in the four predefined operations above. *function* is a user-created function and must refer to an M-file of the form `z = fcn(x,y,other_arguments...)`.

`qrydata(...,cellarray1,cellarray2,...)` is used to input multiple cell arrays. This allows more than one data query to be performed on a given point.

## Controls



### Sample data query dialog box

If an axes handle `h` is not provided, or if the axes specified by `h` is not a map axes, the currently selected point is labeled as **Xloc** and **Yloc** at the top of the query dialog box. If `h` is a map axes, the current point is labeled as **Lat** and **Lon**. Displayed below the current point are the results from the queries, each labeled as specified by the 'label' input arguments.

The **Get** button appears if an axes handle `h` is provided. Pressing this button activates a mouse cursor, which is used to select the desired point by clicking the axes. Once a point is selected, the queries are performed and the results are displayed.

The **Process** button appears if the handle `h` is not provided. In this case, the  $(x, y)$  coordinates of the desired point are entered into the edit boxes. Pressing the **Process** button performs the data queries and displays the results.

Pressing the **Close** button closes the query dialog box.

## Examples

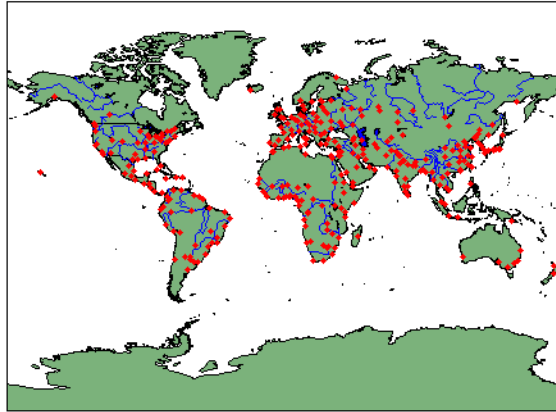
This example illustrates use of a user-defined query to display city names for map points specified by a mouse click. The query is evaluated by a user-supplied M-file called `qrytest.m`, described below:

```
axesm miller
land = shaperead('landareas', 'UseGeoCoords', true);
```

```
geoshow(land, 'FaceColor', [0.5 0.7 0.5])
lakes = shaperead('worldlakes', 'UseGeoCoords', true);
geoshow(lakes, 'FaceColor', 'blue')
rivers = shaperead('worldrivers', 'UseGeoCoords', true);
geoshow(rivers, 'Color', 'blue')
cities = shaperead('worldcities', 'UseGeoCoords', true);
geoshow(cities, 'Marker', '.', 'Color', 'red')
tightmap
lat = [cities.Lat]';
lon = [cities.Lon]';
mat = strvcat(cities.Name);
qrydata(gca, 'City Data', {'City', 'qrytest', lat, lon, mat})
```

Create the M-file qrytest on your path, and in it put the following code:

```
function cityname = qrytest(lt, lg, lat, lon, mat)
% function QRYTEST returns city name for mouse click
% QRYTEST will find the closest city (min radius) from
% the mouse click, within an angle of 5 degrees.
%
latdiff = lt-lat;
londiff = lg-lon;
rad = sqrt(latdiff.^2+londiff.^2);
[minrad,index] = min(rad);
if minrad > 5
    index = [];
end
switch length(index)
    case 0, cityname = 'No city located near click';
    case 1, cityname = mat(index,:);
end
```



Clicking the mouse over a city marker displays the name of the selected city. Clicking the mouse in an area away from any city markers displays the string 'No city located near click'.

## See Also

`interp2`



**Purpose** GUI to display small circles on map axes

---

**Note** scirclui is obsolete. Use scircleg instead.

---

**Activation**

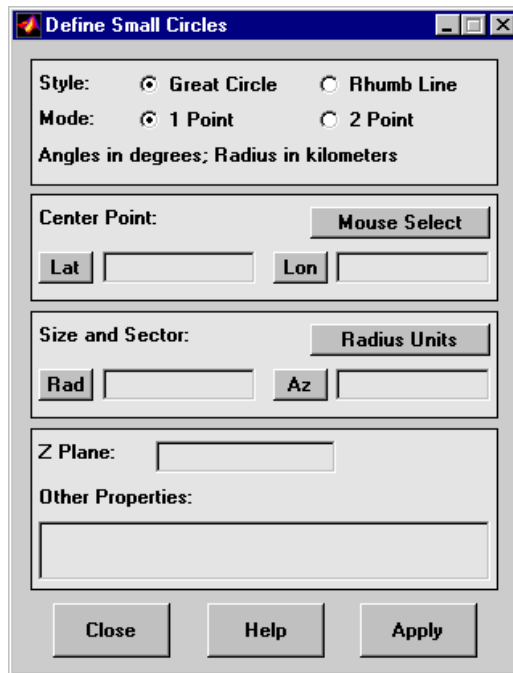
Command Line	Maptool
scirclui	<b>Display mall Circles</b>
scirclui(h)	

**Description**

scirclui activates the Define Small Circles dialog box for adding small circles to the current map axes.

scirclui(h) activates the Define Small Circles dialog box for adding small circles to the map axes specified by the axes handle h.

## Controls



The image shows a Windows-style dialog box titled "Define Small Circles". It has a blue title bar with standard window controls. The dialog is divided into several sections:

- Style:** Two radio buttons: "Great Circle" (selected) and "Rhumb Line".
- Mode:** Two radio buttons: "1 Point" (selected) and "2 Point".
- Angles in degrees; Radius in kilometers:** A text label.
- Center Point:** A "Mouse Select" button and two text boxes labeled "Lat" and "Lon".
- Size and Sector:** A "Radius Units" button and two text boxes labeled "Rad" and "Az".
- Z Plane:** A text box.
- Other Properties:** A large empty text box.
- Buttons:** "Close", "Help", and "Apply" buttons at the bottom.

### Define Small Circles dialog box for one-point mode

The **Style** selection buttons are used to specify whether the circle radius is a constant great circle distance or a constant rhumb line distance.

The **Mode** selection buttons are used to specify whether one point or two points are to be used in defining the small circle. If one-point mode is selected, a center point, radius, and azimuth are the required inputs. If two-point mode is selected, a center point, and perimeter point on the circle are the required inputs.

The **Center Point** controls are used in both one-point and two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the center point of the small circle to be displayed. These values must be in degrees. To display more than one small circle, a vector of values can be entered, enclosed in brackets in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for

easier entry of long vectors. The **Mouse Select** button is used to select a center point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

The **Circle Point** controls are used only in two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of a point on the perimeter of the small circle to be displayed. These values must be in degrees. To display more than one small circle, a vector of values can be entered, enclosed in brackets in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for easier entry of long vectors. The **Mouse Select** button is used to select a perimeter point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

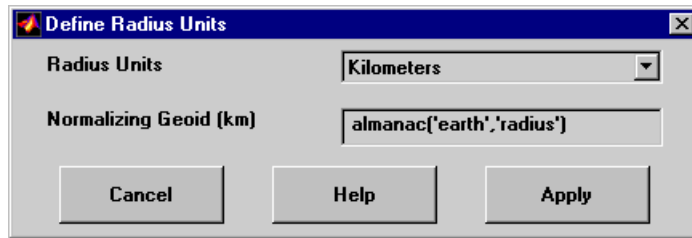
The **Size and Sector** controls are used only in one-point mode. The **Radius Units** button brings up a Define Radius Units dialog box, which allows for modification of the small circle radius units and the normalizing geoid. The **Rad** edit box is used to enter the radius of the small circle in the proper units. The **Arc** edit box is used to specify the sector azimuth, measured in degrees, clockwise from due north. If the entry is omitted, a complete small circle is drawn. When entering radius and arc data for more than one small circle, vectors of values, enclosed in brackets, are entered in each edit box. Pushing the **Rad** or **Arc** button brings up an expanded edit box for that entry, which is useful for entering long vectors.

The **Z Plane** edit box is used to enter a scalar value that specifies the plane in which to display the small circles.

The **Other Properties** edit box is used to specify additional properties of the small circles to be projected, such as 'Color', 'b'. String entries must be enclosed in quotes.

Pressing the **Apply** button accepts the input data and displays the small circles on the current map axes.

Pressing the **Cancel** button disregards any input data and closes the Define Small Circles dialog box.



### Define Radius Units Dialog Box

This dialog box, available only in one-point mode, allows for modification of the small circle radius units and the normalizing geoid.

The **Radius Units** pull-down menu is used to select the units of the small circle radius. The unit selected is displayed near the top of the Define Small Circles dialog box, and all latitude and longitude entries must be entered in these units. Users must also be sure to specify the normalizing geoid in the same units. If radians are selected, it is assumed the radius entry is a multiple of the radius used to display the current map, as defined by the map geoid property.

The **Normalizing Geoid** edit box is used to modify the radius used to normalize the small circle radius to a radian value, which is necessary for proper calculations and map display. This entry must be in the same units as the small circle radius. If the small circle radius units are in radians, then the normalizing geoid must be the same as the geoid used for the current map axes.

Pressing the **Cancel** button disregards any modifications and closes the Define Radius Units dialog box.

Pressing the **Apply** button accepts any modifications and returns to the Define Small Circles dialog box.

### See Also

scircle1, scircle2

**Purpose** GUI to fill data grids with seeded values

**Activation**

**Command Line**

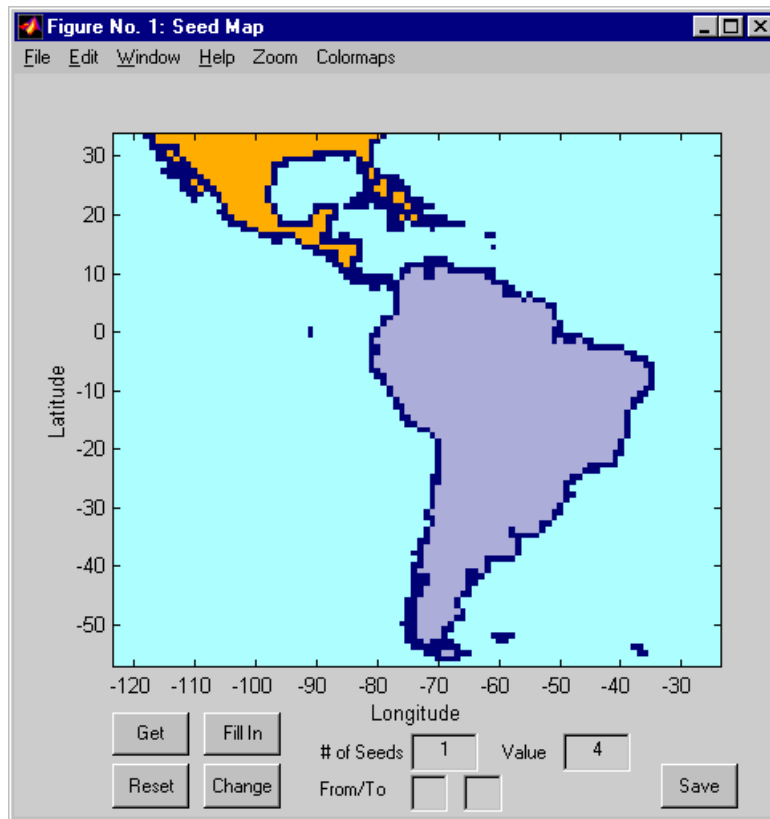
```
seedm(datagrid,refvec)
```

**Description**

Encoding is the process of filling in specific values in regions of a data grid up to specified boundaries, which are indicated by entries of 1 in the variable map. Encoding entire regions at one time allows indexed maps to be created quickly.

`seedm(datagrid,refvec)` displays the surface map in a new figure window and allows for seeds to be specified and the encoded map generated. The encoded map can then be saved to the workspace. `map` is the data grid and must consist of positive integer index values. `refvec` is the referencing vector of the surface.

## Controls



The **Zoom On/Off** menu toggles the zoom box on and off. The box can be moved by clicking the new location or by dragging the box to the new location. The box size can be increased or decreased by dragging a corner of the box. Pressing the **Return** key or double-clicking in the center of the box zooms in to the box limits.

The **Colormaps** menu provides a variety of colormap options that can be applied to the map. See `clrmenu` in this guide for a description of the **Colormaps** menu options.

The **Get** button allows mouse selection of points on the map to which seeds are assigned. The number of points to be selected is entered in

the **# of Seeds** edit box. The value of the seed is entered in the **Value** edit box. This seed value is assigned to each point selected with the mouse. The **Get** button is pressed to begin mouse selection. After all the points have been selected, the **Fill In** button is pressed to perform the encoding operation. The region containing the seed point is filled in with the seed value. The **Reset** button is used to disregard all points selected with the mouse before the **Fill In** button is pressed.

Alternatively, specific map values can be globally replaced by using the **From/To** edit boxes. The value to be replaced is entered in the first edit box, and the new value is entered in the second edit box. Pressing the **Change** button replaces all instances of the **From** value to the **To** value in the map.

---

**Note** Values of 1 represent boundaries and should not be changed.

---

The **Save** button is used to save the encoded map to the workspace. A dialog box appears in which the map variable name is entered.

## See Also

colorm, encodem, getseeds, maptrim

# showm-ui

---

**Purpose** Show specified mapped objects

**Activation**

Command Line	Maptool
showm	<b>Tools &gt; Show &gt; Object</b>

**Description**

showm brings up a Select Object dialog box for selecting mapped objects to show (Visible property set to 'on').

**Controls**



The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button changes the Visible property of the selected objects to 'on'. Pushing the **Cancel** button aborts the operation without changing any properties of the selected objects.

**See Also**

showm



**Purpose** Interactive distance, azimuth, and reckoning calculations

**Activation**

Command Line	Maptool
surfdist	<b>Display &gt; Surface &gt; Distances</b>
surfdist(h)	
surfdist([])	

**Description**

surfdist activates the Surface Distance dialog box for the current axes only if the axes has a proper map definition. Otherwise, the Surface Distance dialog box is activated, but is not associated with any axes.

surfdist(h) activates the Surface Distance dialog box for the axes specified by the handle h. The axes must be a map axes.

surfdist([]) activates the Surface Distance dialog box and does not associate it with any axes, regardless of whether the current axes has a valid map definition.

## Controls

Surface Distance

Style:  Great Circle  Rhumb Line

Mode:  1 Point  2 Point

Show Track

Angles in degrees: Range in kilometers

Starting Point:

Lat:  Lon:

Ending Point:

Lat:  Lon:

Direction:

Az:  Rng:

The **Style** selection buttons are used to specify whether a great circle or rhumb line is used to calculate the surface distance. When all other entries are provided, selecting a style updates the surface distance calculation.

The **Mode** selection buttons are used to specify whether one point or two points are to be used in defining the track distance. If one-point mode is selected, a starting point, azimuth, and range are the required inputs, and the ending point is computed. If two-point mode is selected, starting and ending points of the track are required, and the azimuth and distance along this track are then computed.

The **Show Track** check box is used to indicate whether the track is shown on the associated map display. The track is deleted when the Surface Distance dialog box is closed, or when the **Show Track** check box is unchecked and the surface distance calculations are recomputed.

The **Starting Point** controls are used for both one-point and two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the starting point of the track. These values must be in degrees. Only one starting point can be entered. The **Mouse Select** button is used to select a starting point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

The **Ending Point** controls are enabled only for two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the ending point of the track. These values must be in degrees. Only one ending point can be entered. The **Mouse Select** button is used to select an ending point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection. During one-point mode, the Ending Point controls are disabled, but the ending point that results from the surface distance calculation is displayed.

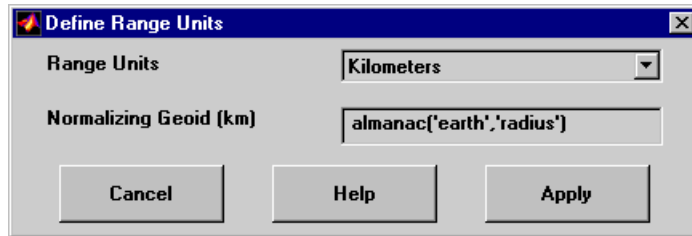
The **Direction** controls are enabled only for one-point mode. The **Range Units** button brings up a Define Range Units dialog box which allows for modification of the range units and the normalizing geoid. The **Az** edit box is used to enter the azimuth, which sets the initial direction of the track from the starting point. Azimuth is measured in degrees clockwise from due north. The **Rng** edit box is used to specify the reckoning range of the track, in the proper units. The azimuth and reckoning range, along with the starting point, are used to compute the ending point of the track in one-point mode. During two-point mode, the **Direction** controls are disabled, but the azimuth and range values resulting from the surface distance calculation are displayed.

Pressing the **Close** button disregards any input data, deletes any surface distance tracks that have been plotted, and closes the Surface Distance dialog box.

Pressing the **Compute** button accepts the input data and computes the specified distances.

## Define Range Units Dialog Box

This dialog box, available only for one-point mode, allows for modification of the range units and the normalizing geoid.



The **Range Units** pull-down menu is used to select the units of the reckoning range. The unit selected is displayed near the top of the Surface Distance dialog box, and all latitude and longitude entries must be entered in these units. Users must also be sure to specify the normalizing geoid in the same units. If radians are selected, it is assumed the range entry is a multiple of the radius of the normalizing geoid. In this case, the normalizing geoid must be the same as the geoid used to display the current map.

The **Normalizing Geoid** edit box is used modify the radius used to normalize range entries to radian values, which is necessary for proper calculations and map display. This entry must be in the same units as the range units. If the range units are in radians, then the normalizing geoid must be the same as the geoid used for the current map axes.

Pressing the **Cancel** button disregards any modifications and closes the Define Range Units dialog box.

Pressing the **Apply** button accepts any modifications and returns to the Surface Distance dialog box.

**Purpose** GUI to edit tag property of mapped object

**Activation****Command Line**

```
tagm
```

```
tagm(h)
```

**Description**

tagm brings up a Select Object dialog box for selecting mapped objects and changing their Tag property. Upon selecting the objects, the Edit Tag dialog box is activated, in which the new tag is entered.

tagm(h) activates the Edit Tag dialog box for the objects specified by the handle h.

**Controls****Select Object Dialog Box**

The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **Ok** button activates the Edit Tag dialog box. Pushing the **Cancel** button aborts the operation without changing any properties of the selected objects.



## **Edit Tag Dialog Box**

The new tag string is entered in the edit box. Pressing the **Apply** button changes the Tag property of all selected objects to the new tag string. Pressing the **Cancel** button closes the Edit Tag dialog box without changing the Tag property of the selected objects.

## **See Also**

tagm

**Purpose** GUI to display great circles and rhumb lines on map axes

---

**Note** trackui is obsolete. Use trackg instead.

---

**Activation**

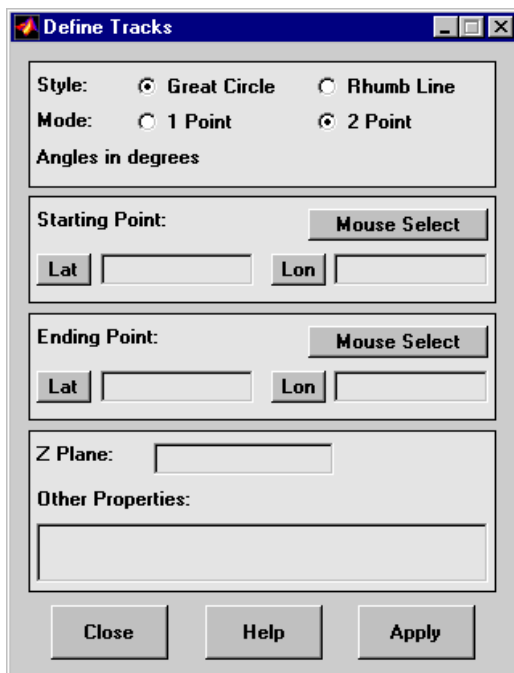
Command Line	Maptool
trackui	<b>Display &gt; Tracks</b>
trackui(h)	

**Description**

trackui activates the Define Tracks dialog box for adding great circle or rhumb line tracks to the current map axes.

trackui(h) activates the Define Tracks dialog box for adding great circle or rhumb line tracks to the map axes specified by the axes handle h.

## Controls



### Define Tracks dialog box for two-point mode

The **Style** selection buttons are used to specify whether a great circle or rhumb line track is displayed.

The **Mode** selection buttons are used to specify whether one point or two points are to be used in defining the track. If one-point mode is selected, a starting point, azimuth, and range are the required inputs. If two-point mode is selected, starting and ending points are required.

The **Starting Point** controls are used for both one-point and two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the starting point of the track to be displayed. These values must be in degrees. To display more than one track, a vector of values can be entered, enclosed in brackets in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for easier entry of long vectors. The **Mouse Select** button is used to select a starting point



by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

The **Ending Point** controls are used only for two-point mode. The **Lat** and **Lon** edit boxes are used to enter the latitude and longitude of the ending point of the track to be displayed. These values must be in degrees. To display more than one track, a vector of values can be entered, enclosed in brackets, in each edit box. Pushing the **Lat** or **Lon** button brings up an expanded edit box for easier entry of long vectors. The **Mouse Select** button is used to select an ending point by clicking the displayed map. The coordinates of the selected point then appear in the **Lat** and **Lon** edit boxes and can be modified. The coordinates appear in degrees, regardless of the angle units defined for the current map projection.

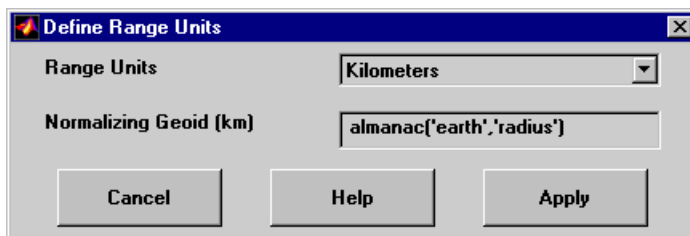
The **Direction** controls are used only for one-point mode. The **Range Units** button brings up a Define Range Units dialog box, which allows for modification of the range units and the normalizing geoid. The **Az** edit box is used to enter the azimuth, which sets the initial direction of the track from the starting point. Azimuth is measured in degrees clockwise from due north. The **Rng** edit box is used to specify the range of the track, in the proper units. If the range entry is omitted, a complete track is drawn. When inputting azimuth and range data for more than one track, vectors of values, enclosed in brackets, are entered in each edit box. Pushing the **Az** or **Rng** button brings up an expanded edit box for that entry, which is useful for entering long vectors.

The **Z Plane** edit box is used to enter a scalar value that specifies the plane in which to display the tracks.

The **Other Properties** edit box is used to specify additional properties of the tracks to be projected, such as 'Color', 'b'. String entries must be enclosed in quotes.

Pressing the **Apply** button accepts the input data and displays the tracks on the current map axes.

Pressing the **Cancel** button disregards any input data and closes the Define Tracks dialog box.



### Define Range Units Dialog Box

This dialog box, available only for one-point mode, allows for modification of the range units and the normalizing geoid.

The **Range Units** pull-down menu is used to select the units of the track range. The unit selected is displayed near the top of the Define Tracks dialog box, and all latitude and longitude entries must be entered in these units. Users must also be sure to specify the normalizing geoid in the same units. If radians are selected, it is assumed the range entry is a multiple of the radius used to display the current map.

The **Normalizing Geoid** edit box is used to modify the radius used to normalize range entries to radian values, which is necessary for proper calculations and map display. This entry must be in the same units as the range units. If the range units are in radians, then the normalizing geoid must be the same as the geoid used for the current map axes.

Pressing the **Cancel** button disregards any modifications and closes the Define Range Units dialog box.

Pressing the **Apply** button accepts any modifications and returns to the Define Tracks dialog box.

### See Also

track1, track2

**Purpose** Handle buttondown callbacks for mapped objects

**Activation** set the ButtonDownFcn property to 'uimaptbx'

**Description** uimaptbx processes mouse events for mapped objects. uimaptbx can be assigned to an object by setting the ButtonDownFcn to 'uimaptbx'. This is the default setting for all objects created with Mapping Toolbox.

If uimaptbx is assigned to an object, the following mouse events are recognized: A single-click and hold on an object displays the object tag. If no tag is assigned, the object type is displayed. A double-click on an object activates the MATLAB Property Editor. An extend-click on an object activates the Projection Control dialog box, which allows the map projection and display properties to be edited. An alternate-click on an object allows basic properties to be edited using simple mouse clicks and drags.

Definitions of extend-click and alternate-click on various platforms are as follows:

For MS-Windows: Extend-click – **Shift**+click left button or both buttons

Alternate-click – **Ctrl**+click left button or right button

For X-Windows: Extend-click – **Shift**+click left button or middle button

Alternate-click – **Ctrl**+ click left button or right button

**See Also** axesm, axesmui, property editors

# utmzoneui

---

**Purpose** Choose or identify UTM zone by clicking map

**Activation**

Command Line
utmzoneui
utmzoneui(InitZone)

**Description**

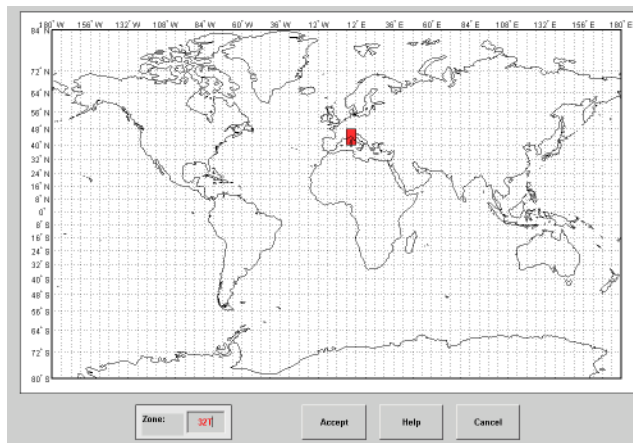
zone = utmzoneui will create an interface for choosing a UTM zone on a world display map. It allows for clicking an area for its appropriate zone, or entering a valid zone to identify the zone on the map.

zone = utmzoneui(InitZone) will initialize the displayed zone to the zone string given in InitZone.

To interactively pick a UTM zone, activate the interface, and then click any rectangular zone on the world map to display its UTM zone. The selected zone is highlighted in red and its designation is displayed in the **Zone** edit field. Alternatively, type a valid UTM designation in the **Zone** edit field to select and see the location of a zone. Valid zone designations consist of an integer from 1 to 60 followed by a letter from C to X.

Typing only the numeric portion of a zone designation will highlight a column of cells. Clicking **Accept** returns a that UTM column designation. You cannot return a letter (row designation) in such a manner, however.

## Controls



## Remarks

The syntax of `utmzoneui` is similar to that of `utmzone`. If `utmzone` is called with no arguments, the `utmzoneui` interface is displayed for you to select a zone. Note that `utmzone` can return latitude-longitude coordinates of a specified zone, but that `utmzoneui` only returns zone names.

## See Also

<code>ups</code>	Universal Polar Stereographic (UPS) Projection.
<code>utm</code>	Universal Transverse Mercator (UTM) Projection.
<code>utmgeoid</code>	Select ellipsoid for a given UTM zone.
<code>utmzone</code>	Select a UTM zone.

# vmap0ui

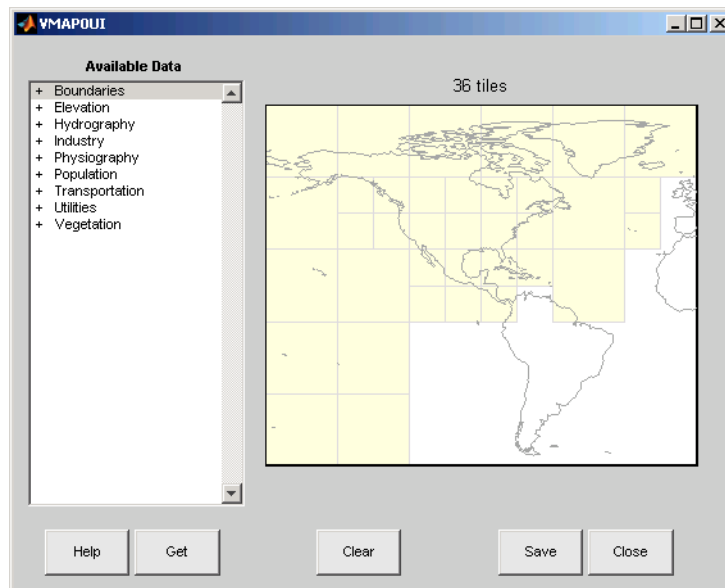
**Purpose** UI for selecting data from Vector Map Level 0

**Description** `vmap0ui(dirname)` launches a graphical user interface for interactively selecting and importing data from a Vector Map Level 0 (VMAP0) data base. Use the string `dirname` to specify the directory containing the data base. For more on using `vmap0ui`, click the **Help** button after the interface appears.

`vmap0ui(devicename)` or `vmap0ui devicename` uses the logical device (volume) name specified in string `devicename` to locate CD-ROM drive containing the VMAP0 CD-ROM. Under the Windows operating system it could be 'F:', 'G:', or some other letter. Under Macintosh OS X it should be '/Volumes/VMAP'. Under other UNIX systems it could be '/cdrom/'.

`vmap0ui` can be used on Windows without any arguments. In this case it attempts to automatically detect a drive containing a VMAP0 CD-ROM. If `vmap0ui` fails to locate the CD-ROM device, then specify it explicitly.

## Controls



The vmap0ui screen lets you read data from the Vector Map Level 0 (VMAPO). The VMAPO is the most detailed world map database available to the public.

You use the list to select the type of data and the map to select the region of interest. When you click the **Get** button, data is extracted and displayed on the map. Use the **Save** button to save the data in a MAT-file or to the base workspace for later display. The **Close** button closes the window.

### The Map

The **Map** controls the geographic extent of the data to be extracted. vmap0ui extracts data for areas currently visible on the map. Use the mouse to zoom in or out to the area of interest. Type `help zoom` for more on zooming.

The VMAPO divides the world into tiles of about 5-by-5 degrees. When extracting, data is returned for all visible tiles, including those parts of the tile that are outside the current view. The map shows the VMAPO tiles in light yellow with light gray edges. The data density is high, so extracting data for a large number of tiles can take much time and memory. A count of the number of visible tiles is above the map.

### The List

The **List** controls the type of data to be extracted. The tree structure of the list reflects the structure of the VMAPO database. Upon starting vmap0ui, the list shows the major categories of VMAP data, called themes. Themes are subdivided into features, which consist of data of common graphic types (patch, line, point, or text) or cultural types (airport, roads, railroads). Double-click a theme to see the associated features. Features can have properties and values, for example, a railroad tracks property, with values single or multiple. Double-click a feature to see the associated properties and values. Double-clicking an open theme or feature closes it. When a theme is selected, vmap0ui gets all the associated features. When a feature is selected, vmap0ui gets all of that feature's data. When properties and values are selected, vmap0ui gets the data for any of the properties and values that match (that is, the union operation).

## The Get Button

The **Get** button reads the currently selected VMAP0 data and displays it on the map. Use the **Cancel** button on the progress bar to interrupt the process. For a quicker response, press the standard interrupt key combination for your platform.

## The Clear Button

The **Clear** button removes any previously read data from the map.

## The Save Button

The **Save** button saves the currently displayed VMAP0 data to a MAT-file or the base workspace. If you choose to save to a file, you are prompted for a filename and location. If you choose to save to the base workspace, you are notified of the variable names that will be overwritten. The results are stored as geographic data structures with variable names based on theme and feature names. Use `load` and `displaym` to redisplay the data from a file on a map axes. You can also use the `mlayers` GUI to read and display the data from a file. To display the data in the base workspace, use `displaym`. To display all the geographic data structures, use `rootlayer; displaym(ans)`. To display all of the geographic data structures using the `mlayers` GUI, type `rootlayer; mlayers(ans)`.

## The Close Button

The **Close** button closes the `vmap0ui` panel.

## Examples

- 1 Launch `vmap0ui` and automatically detect a CD-ROM on Microsoft Windows:

```
vmap0ui
```

- 2 Launch `vmap0ui` on Macintosh OS X (need to specify volume name):

```
vmap0ui('Volumes/VMAP')
```

## See also

`displaym`, `extractm`, `mlayers`, `vmap0data`



**Purpose** GUI to adjust  $z$ -plane of mapped objects

**Activation****Command Line**

```
zdatam  
zdatam(h)  
zdatam(str)
```

**Description**

zdatam brings up a Select Object dialog box for selecting mapped objects and adjusting their ZData property. Upon selecting the objects, the Specify Zdata dialog box is activated, in which the new ZData variable is entered. Note that not all mapped objects have the ZData property (for example text objects).

zdatam(h) activates the Specify Zdata dialog box for the objects specified by the handle h.

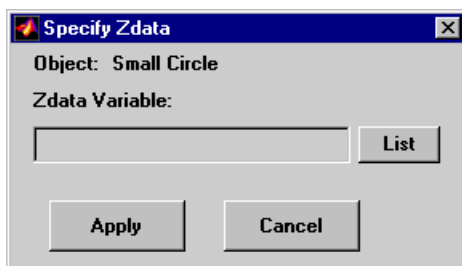
zdatam(str) activates the Specify Zdata dialog box for the objects identified by str, where str is any string recognized by handlem.

## Controls



### Select Object Dialog Box

The scroll box is used to select the desired objects from the list of mapped objects. Pushing the **Select all** button highlights all objects in the scroll box for selection. Pushing the **OK** button activates another Specify Zdata dialog box. Pushing the **Cancel** button aborts the operation without changing any properties of the selected objects.



### Specify ZData Dialog Box

The **Zdata Variable** edit box is used to specify the name of the ZData variable. Pressing the **List** button produces a list of all current workspace variables, from which the ZData variable can be selected. A scalar value or a valid MATLAB expression can also be entered.

Pressing the **Apply** button changes the ZData property of all selected objects to the new values. Pressing the **Cancel** button closes the Specify ZData dialog box without changing the ZData property of the selected objects.

## See Also

zdatam



# Map Projections — By Category

---

Cylindrical Projections (p. 12-2)	Map projections developed on a cylinder
Pseudocylindrical Projections (p. 12-2)	Variants of map projections developed on cylinders
Conic Projections (p. 12-4)	Map projections developed on a cone
Polyconic and Pseudoconic Projections (p. 12-4)	Map projections developed on a family of cones (polyconic) and conic variants
Azimuthal, Pseudoazimuthal, and Modified Azimuthal Projections (p. 12-4)	Map projections that preserve azimuths from a central point and their variants
UTM and UPS Systems (p. 12-5)	Constructing Universal Transverse Mercator and Universal Polar Stereographic maps
3-D Globe Display (p. 12-5)	Visualizing maps on a sphere

See Chapter 8, “Using Map Projections and Coordinate Systems” for a general discussion of map projections, and “Summary and Guide to Projections” on page 8-64 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
millier	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	Briesemeiste's 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	Weichel Equal Area Projection

## UTM and UPS Systems

ups	Universal Polar Stereographic (UPS) system
utm	Universal Transverse Mercator (UTM) system

## 3-D Globe Display

globe	Earth as sphere in 3-D graphics
-------	---------------------------------



# Map Projections — Alphabetical List

---

# Aitoff Projection

---

**Classification** Modified Azimuthal

**Syntax** aitoff

**Graticule** Meridians: Central meridian is a straight line half the length of the Equator. Other meridians are complex curves, equally spaced along the Equator, and concave toward the central meridian.

Parallels: Equator is straight. Other parallels are complex curves, equally spaced along the central meridian, and concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator and central meridian.

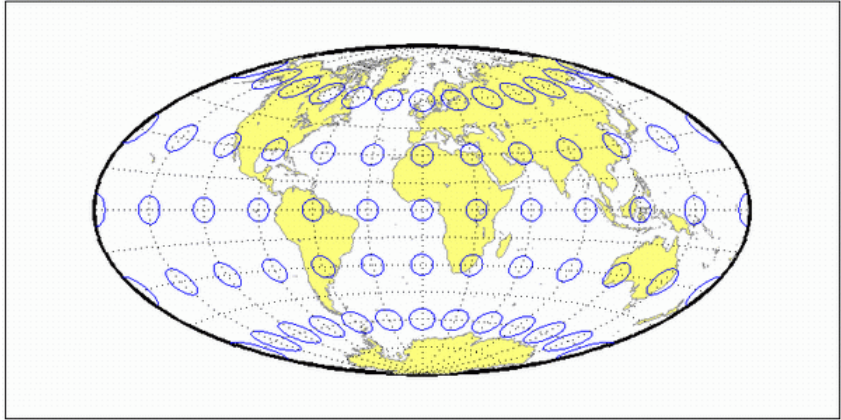
**Features** This projection is neither conformal nor equal area. The only point free of distortion is the center point. Distortion of shape and area are moderate throughout. This projection has less angular distortion on the outer meridians near the poles than pseudoazimuthal projections

**Parallels** There is no standard parallel for this projection.

**Remarks** This projection was created by David Aitoff in 1889. It is a modification of the Equidistant Azimuthal projection. The Aitoff projection inspired the similar Hammer projection, which is equal area.

**Example**

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



# Albers Equal-Area Conic Projection

---

**Classification** Conic

**Syntax** eqaconic  
eqaconic

**Graticule** Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the point of convergence. Spacing of parallels decreases away from the central latitudes.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

**Features** This is an equal-area projection. Scale is true along the one or two selected standard parallels. Scale is constant along any parallel; the scale factor of a meridian at any given point is the reciprocal of that along the parallel to preserve equal-area. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is neither conformal nor equidistant.

**Parallels** The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane and a Lambert Azimuthal Equal-Area projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Equal-Area Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Equal-Area Cylindrical projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

# Albers Equal-Area Conic Projection

## Remarks

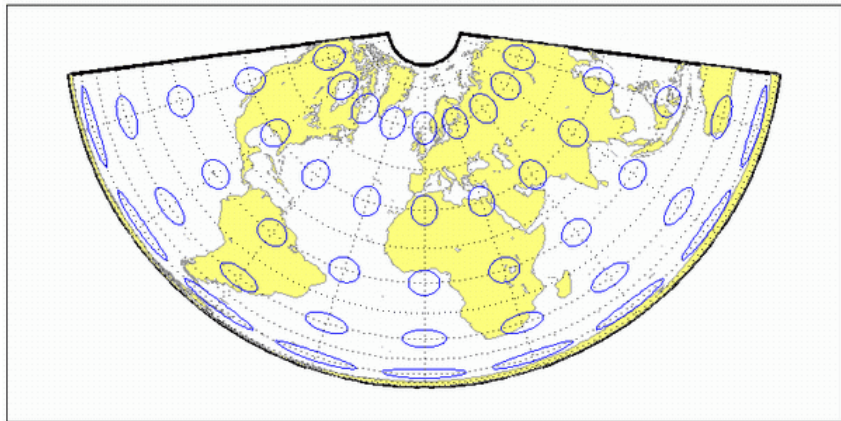
This projection was presented by Heinrich Christian Albers in 1805.

## 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



# Albers Equal-Area Conic Projection – Standard

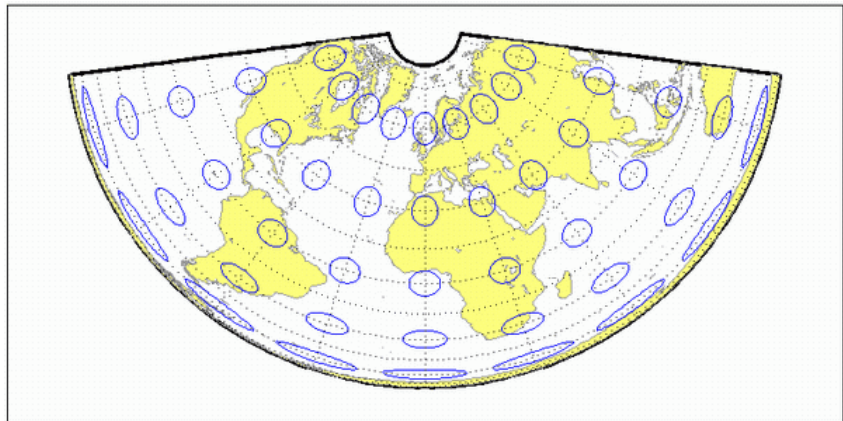
parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

## 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;
```



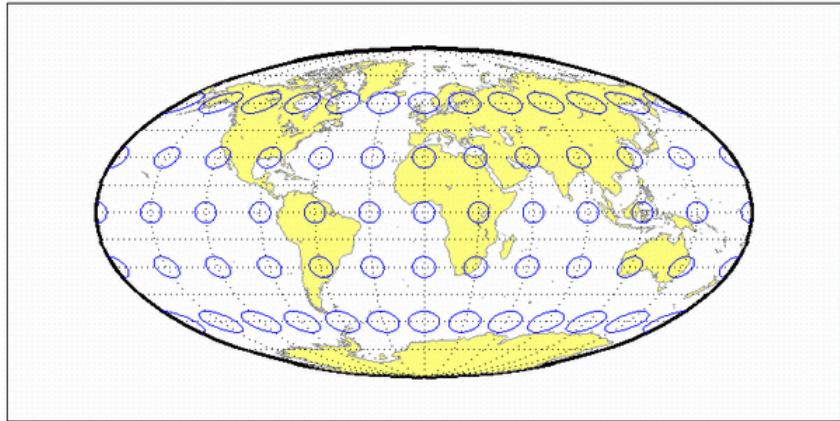
## See also

`eqaconic`

# Apianus II Projection

---

<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	apianus
<b>Graticule</b>	Meridians: Equally spaced elliptical curves converging at the poles. Parallels: Equally spaced straight lines. Poles: Points. Symmetry: About the Equator and central meridian.
<b>Features</b>	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.
<b>Parallels</b>	There is no standard parallel for this projection.
<b>Remarks</b>	This projection was first described in 1524 by Peter Apian (or Bienewitz).
<b>Limitations</b>	This projection is available only on the sphere.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('apianus','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>



# 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 50° 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 50°.

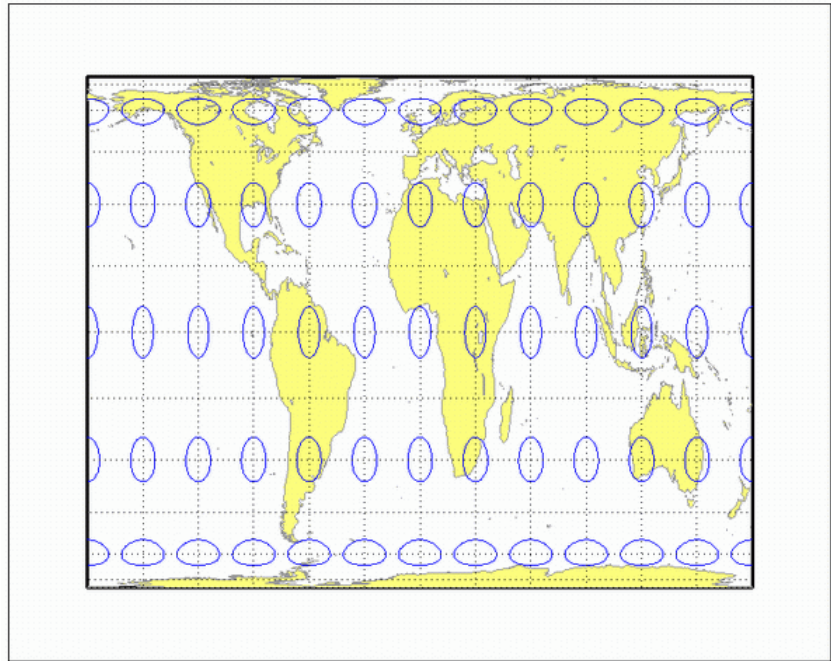
**Remarks** The Balthasart Cylindrical projection was presented in 1935 and is a special form of the Equal-Area Cylindrical projection secant at 50°N and S.

**Example**

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

# Balthasart Cylindrical Projection

---



# 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 30° 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 30°.

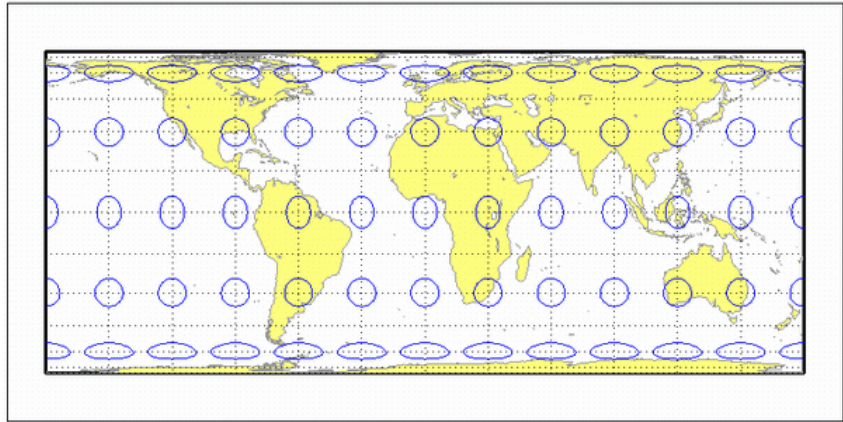
**Remarks** This projection is named for Walter Behrmann, who presented it in 1910 and is a special form of the Equal-Area Cylindrical projection secant at 30°N and S.

**Example**

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

# Behrmann Cylindrical Projection

---



# 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 30° 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 30°.

**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 30°N and S.

**Limitations** This projection is available only on the sphere.

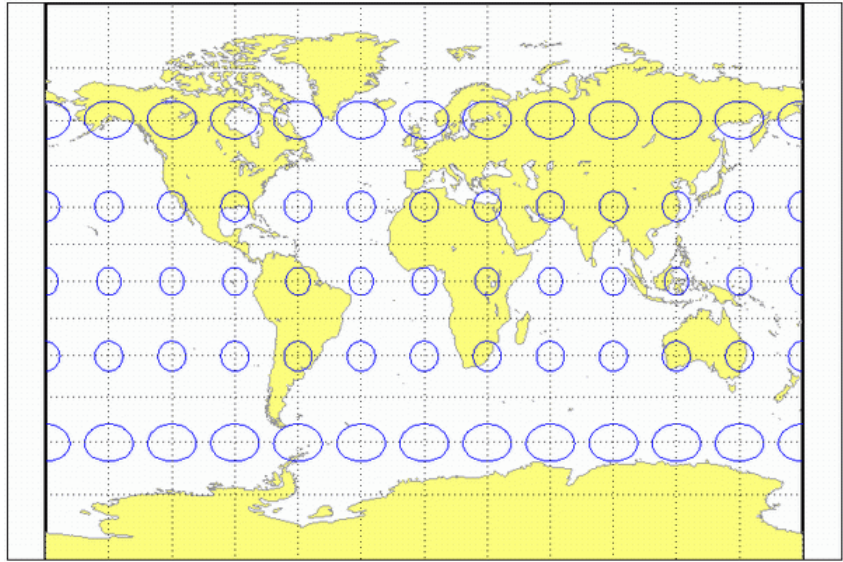
**Example**

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



# Bolshoi Sovietskii Atlas Mira Projection

---



# 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.

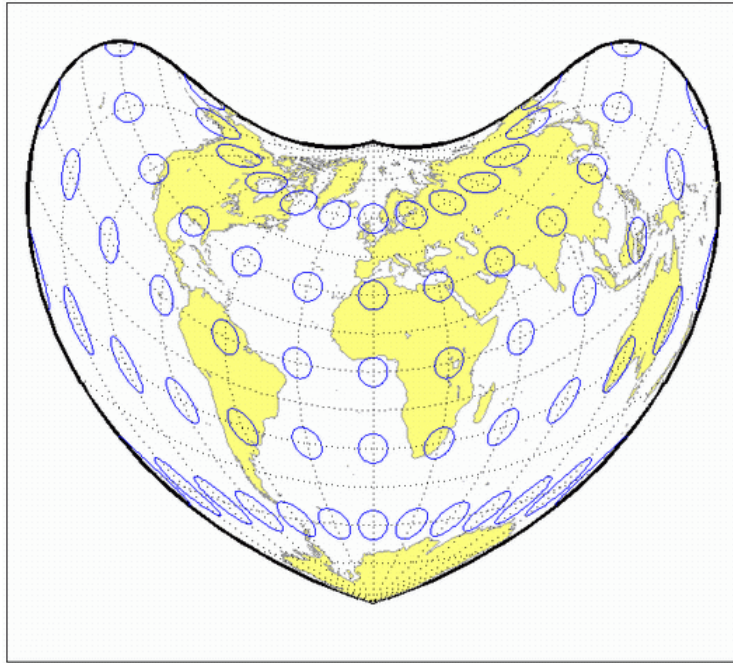
**Features** This is an equal-area projection. The curvature of the standard parallel is identical to that on a cone tangent at that latitude. The central meridian and the central parallel are free of distortion. This projection is not conformal.

**Parallels** This projection has one standard parallel, which is 30°N 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°.

**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.

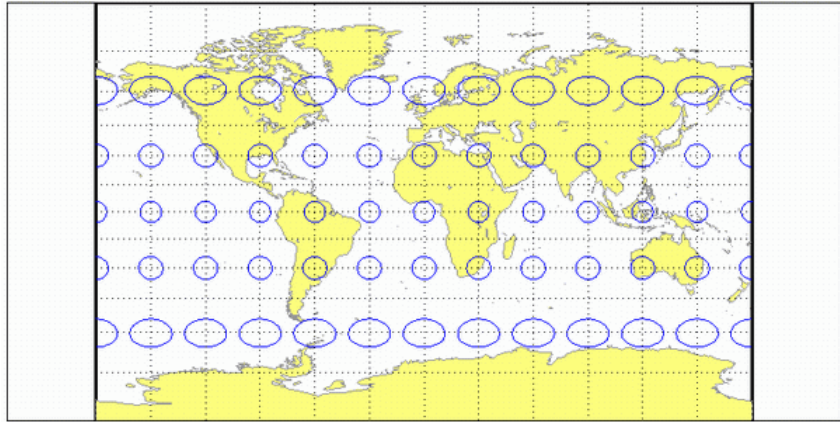
**Limitations** This projection is available only on the sphere.

**Example**

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

# Braun Perspective Cylindrical Projection

---



# 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.

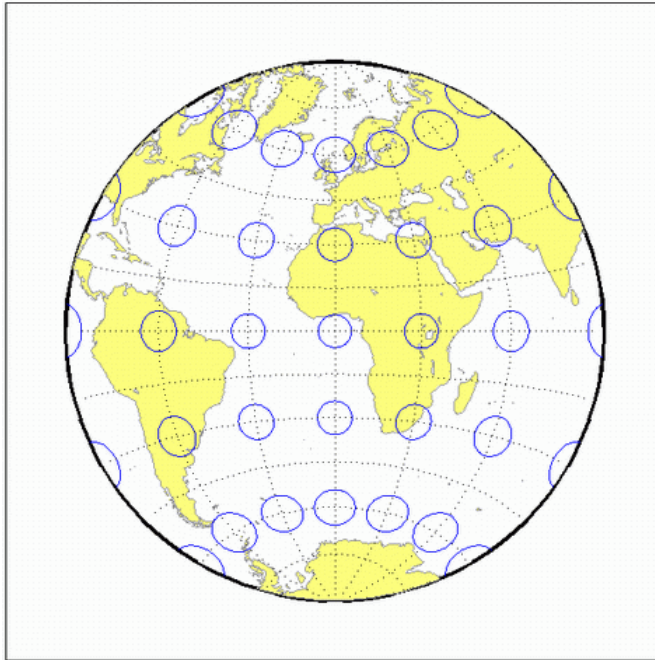
**Limitations** This projection is available only on the sphere.

**Example**

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

# Breusing Harmonic Mean Projection

---



# Briesemeister Projection

---

**Classification** Modified Azimuthal

**Syntax** bries

**Graticule** Meridians: Central meridian is straight. Other meridians are complex curves.

Parallels: Complex curves.

Poles: Points.

Symmetry: About the central meridian.

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

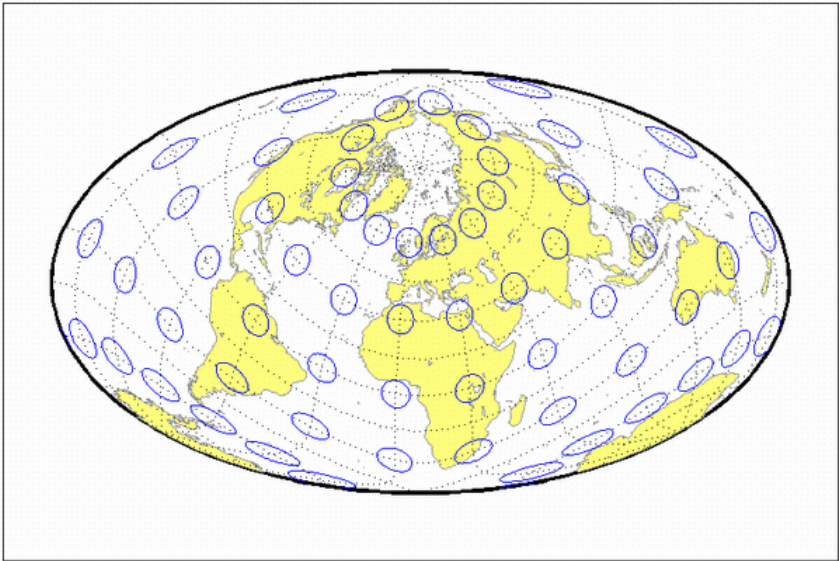
**Parallels** There is no standard parallel for this projection.

**Remarks** This projection was presented by William Briesemeister in 1953. It is an oblique Hammer projection with an axis ratio of 1.75 to 1, instead of 2 to 1.

**Example**

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





# Cassini Cylindrical Projection

---

**Classification** Cylindrical

**Syntax** `cassini`

**Graticule** Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if  $90^\circ$  from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

**Features** This is a projection onto a cylinder tangent at the central meridian. Distortion of both shape and area are functions of distance from the central meridian. Scale is true along the central meridian and along any straight line perpendicular to the central meridian (i.e., it is equidistant).

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

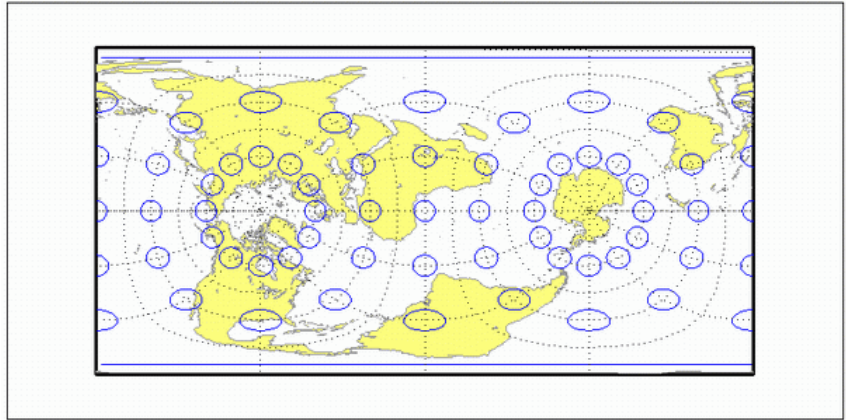
**Remarks** This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.

**Example**

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

# Cassini Cylindrical Projection

---



**See also**

`cassinistd`

# Cassini Cylindrical Projection – Standard

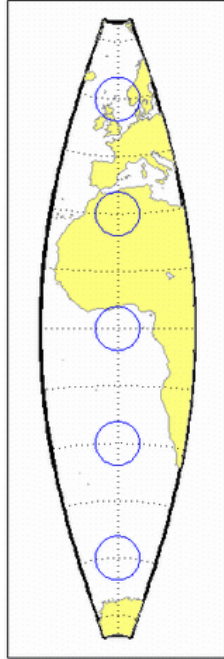
---

<b>Syntax</b>	<code>cassinistd</code>
<b>Graticule</b>	<p>Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).</p> <p>Other Meridians: Straight lines if 90° from central meridian, complex curves concave toward the central meridian otherwise.</p> <p>Parallels: Complex curves concave toward the nearest pole.</p> <p>Poles: Points along the central meridian.</p> <p>Symmetry: About any straight meridian or the Equator.</p>
<b>Features</b>	<p>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).</p>
<b>Parallels</b>	<p>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 <i>of the base projection</i> is by definition fixed at 0°.</p>
<b>Remarks</b>	<p>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.</p> <p><code>cassinistd</code> implements the Cassini projection directly on a sphere or reference ellipsoid, as opposed to using the equidistant cylindrical projection in transverse mode as in function <code>cassini</code>. 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.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('cassinistd', 'Frame', 'on', 'Grid', 'on');</pre>

# Cassini Cylindrical Projection – Standard

---

```
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 0°.

**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.

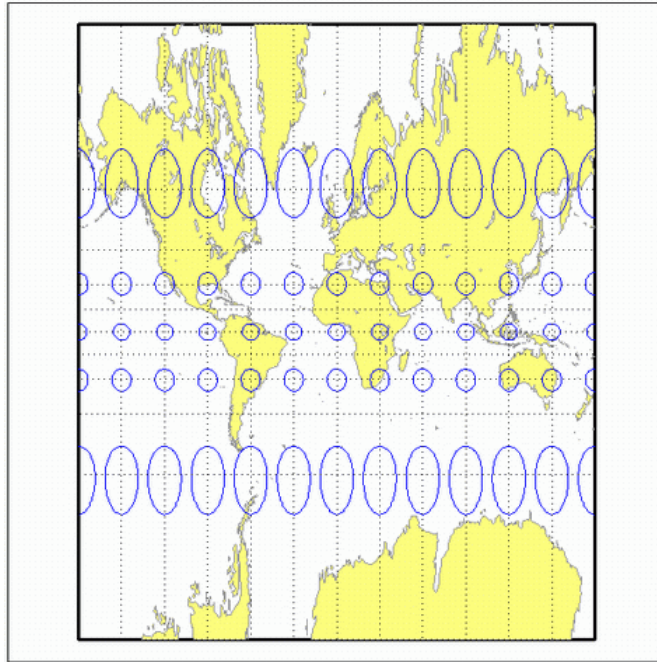
**Limitations** This projection is available only on the sphere. Data at latitudes greater than 75° is trimmed to prevent large values from dominating the display.

**Example**

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

# Central Cylindrical Projection

---



# 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.

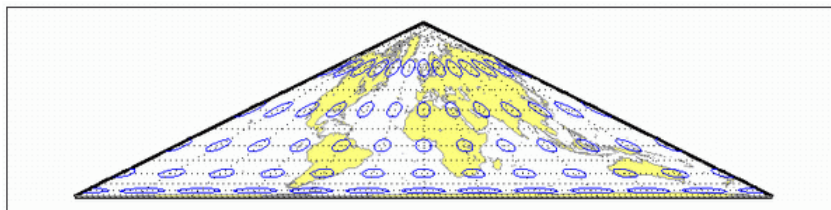
**Features** This is a novelty projection showing a straight-line, equal-area graticule. Scale is true along the 15°51'N parallel, constant along any parallel, and *different* for any pair of parallels. Distortion is severe in many regions, and is only absent at 15°51'N on the central meridian. This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

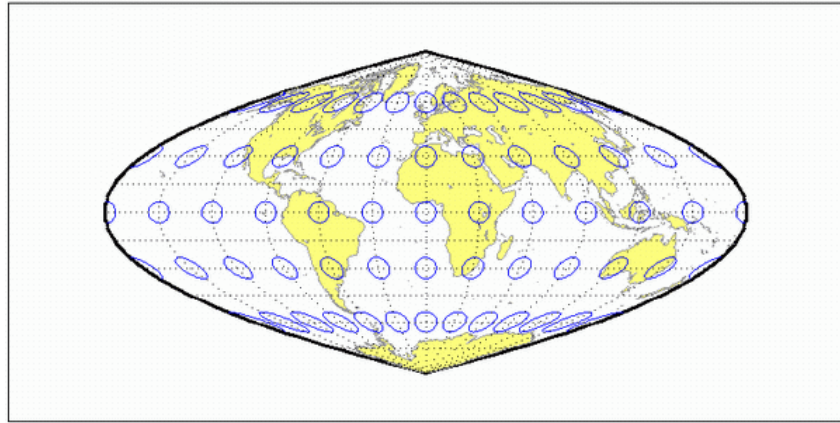




<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	craster
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced parabolas intersecting at the poles and concave toward the central meridian.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes very gradually and is greatest near the Equator.</p> <p>Poles: Points.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along the 36°46' 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 36°46' parallels. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>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 36°46'.</p>
<b>Remarks</b>	<p>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 <math>P_4</math> projection.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('craster', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

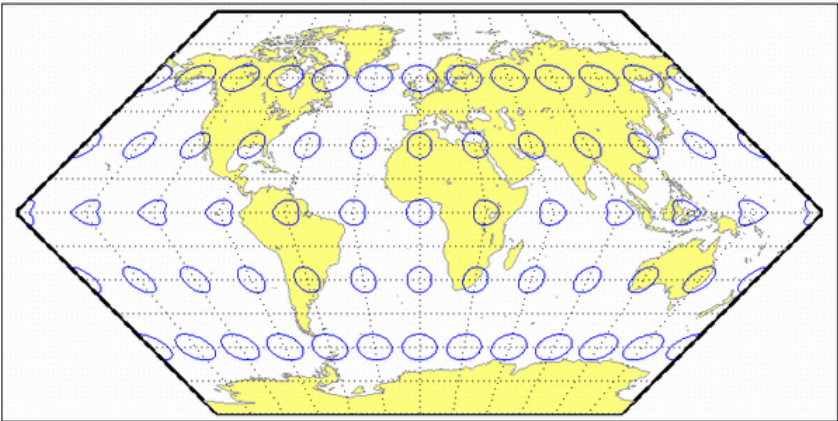
# Craster Parabolic Projection

---



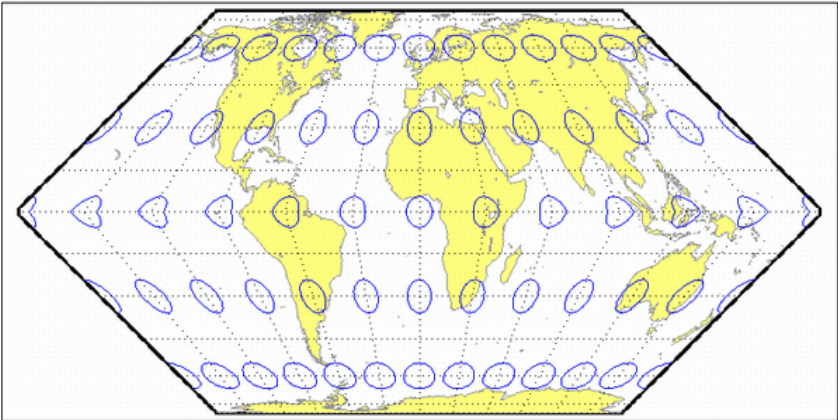
<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	eckert1
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced straight converging lines broken at the Equator.</p> <p>Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.</p> <p>Poles: Lines half as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>Scale is true along the 47°10' 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.</p>
<b>Parallels</b>	<p>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 47°10'.</p>
<b>Remarks</b>	<p>This projection was presented by Max Eckert in 1906.</p>
<b>Limitations</b>	<p>This projection is available only on the sphere.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('eckert1','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Eckert I Projection



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	eckert2
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced straight converging lines broken at the Equator.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.</p> <p>Poles: Lines half as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along the 55°10' 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 55°10'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.</p>
<b>Parallels</b>	<p>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 55°10'.</p>
<b>Remarks</b>	<p>This projection was presented by Max Eckert in 1906.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('eckert2', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

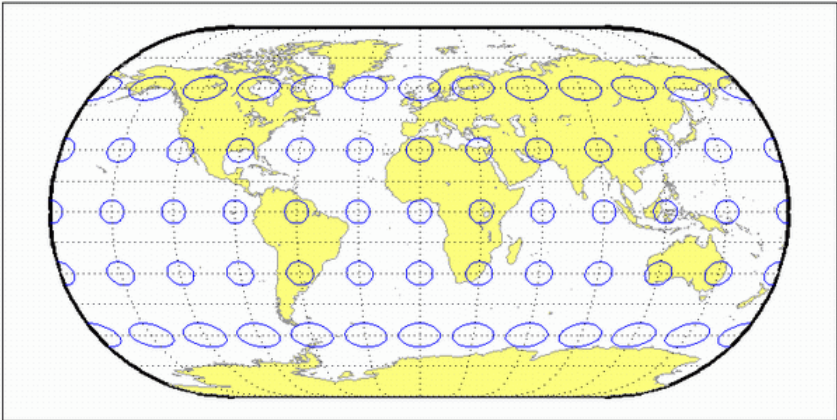
# Eckert II Projection



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	eckert3
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180° east and west of the central meridian, are semicircles.</p> <p>Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.</p> <p>Poles: Lines half as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	Scale is true along the 35°58' 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.
<b>Parallels</b>	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 35°58'.
<b>Remarks</b>	This projection was presented by Max Eckert in 1906.
<b>Limitations</b>	This projection is available only on the sphere.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('eckert3','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Eckert III Projection

---

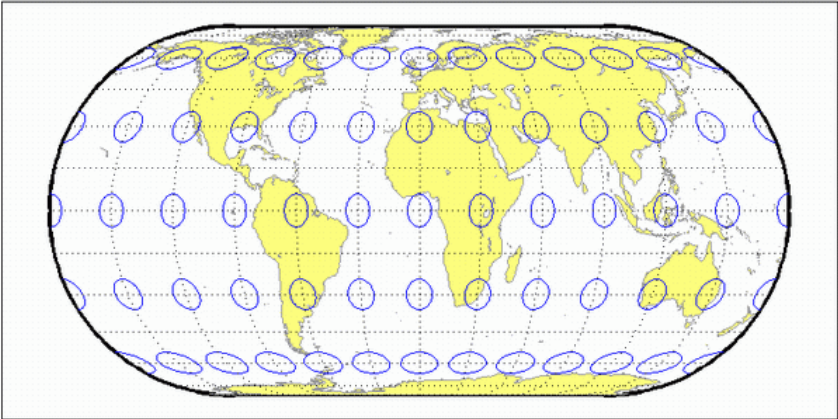




<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	eckert4
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180° east and west of the central meridian, are semicircles.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.</p> <p>Poles: Lines half as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along the 40°30' 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 40°30' parallels intersect the central meridian. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>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 40°30'.</p>
<b>Remarks</b>	<p>This projection was presented by Max Eckert in 1906.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('eckert4', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Eckert IV Projection

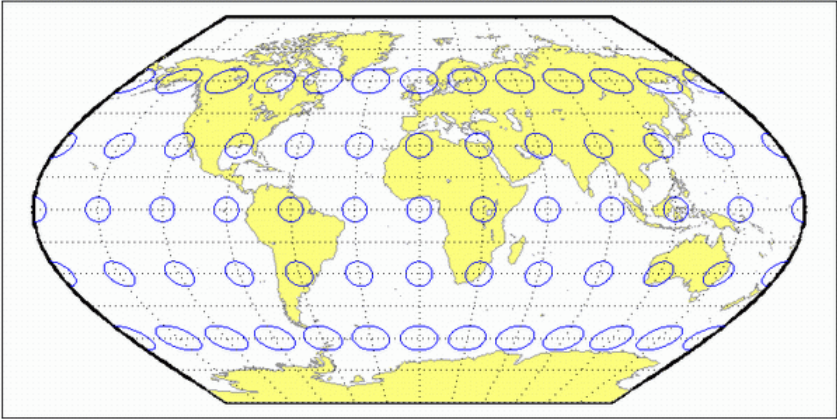
---



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	eckert5 eckert5
<b>Graticule</b>	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.
<b>Features</b>	This projection is an arithmetic average of the $x$ and $y$ coordinates of the Sinusoidal and Plate Carrée projections. Scale is true along latitudes $37^{\circ}55'N$ 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.
<b>Parallels</b>	This projection has one standard parallel, which is by definition fixed at $0^{\circ}$ .
<b>Remarks</b>	This projection was presented by Max Eckert in 1906.
<b>Limitations</b>	This projection is available only on the sphere.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('eckert5','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Eckert V Projection

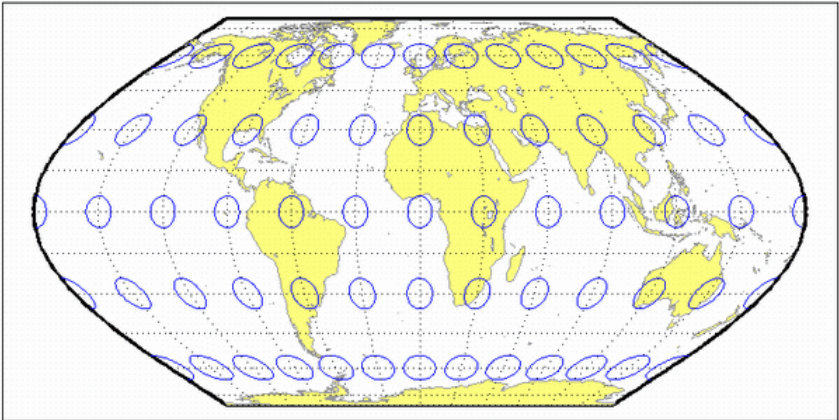
---



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	eckert6
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.</p> <p>Poles: Lines half as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along the 49°16' 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 49°16' parallels intersect the central meridian. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>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 49°16'.</p>
<b>Remarks</b>	<p>This projection was presented by Max Eckert in 1906.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('eckert6', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Eckert VI Projection

---



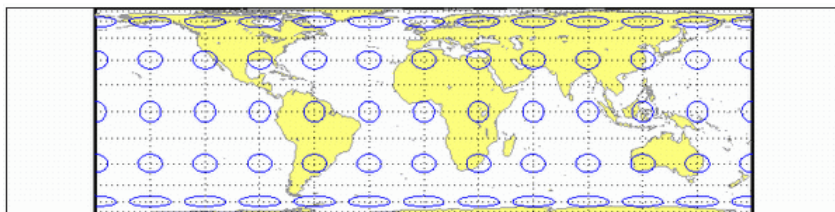
# Equal-Area Cylindrical Projection

---

<b>Classification</b>	Cylindrical
<b>Syntax</b>	eqacylin
<b>Graticule</b>	<p>Meridians: Equally spaced straight parallel lines.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.</p> <p>Poles: Straight lines equal in length to the Equator.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>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° (the Lambert variation).</p>
<b>Remarks</b>	<p>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.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm('eqacylin', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Equal-Area Cylindrical Projection

---



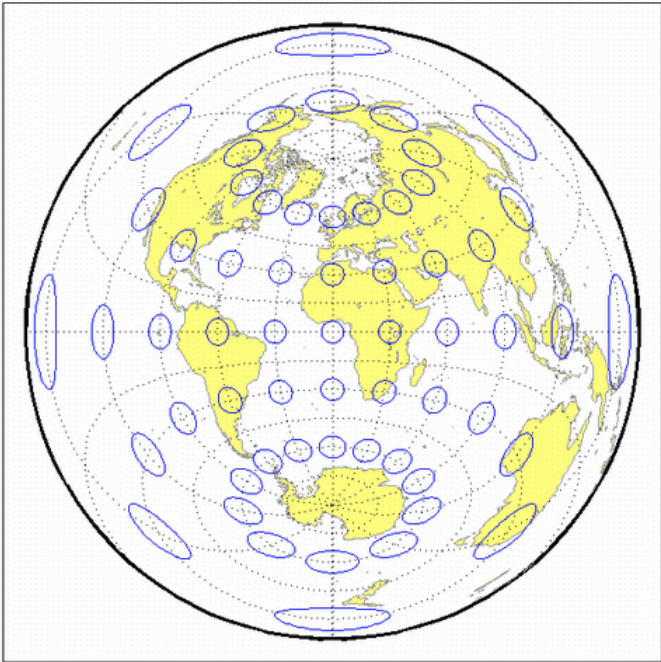


# Equidistant Azimuthal Projection

---

<b>Classification</b>	Azimuthal
<b>Syntax</b>	eqdazim
<b>Graticule</b>	<p>The graticule described is for the polar aspect.</p> <p>Meridians: Equally spaced straight lines intersecting at a central pole. The angles between them are the true angles.</p> <p>Parallels: Equally spaced circles, centered on the central pole. The entire Earth may be shown.</p> <p>Poles: Central pole is a point. The opposite pole is a bounding circle with a radius twice that of the Equator.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>There are no standard parallels for azimuthal projections.</p>
<b>Remarks</b>	<p>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.</p>
<b>Limitations</b>	<p>This projection is available only on the sphere.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('eqdazim','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Equidistant Azimuthal Projection



<b>Classification</b>	Conic
<b>Syntax</b>	eqdconic
<b>Graticule</b>	<p>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.</p> <p>Parallels: Equally spaced concentric circular arcs centered on the point of meridional convergence.</p> <p>Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>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].</p>
<b>Remarks</b>	<p>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.</p>

# Equidistant Conic Projection

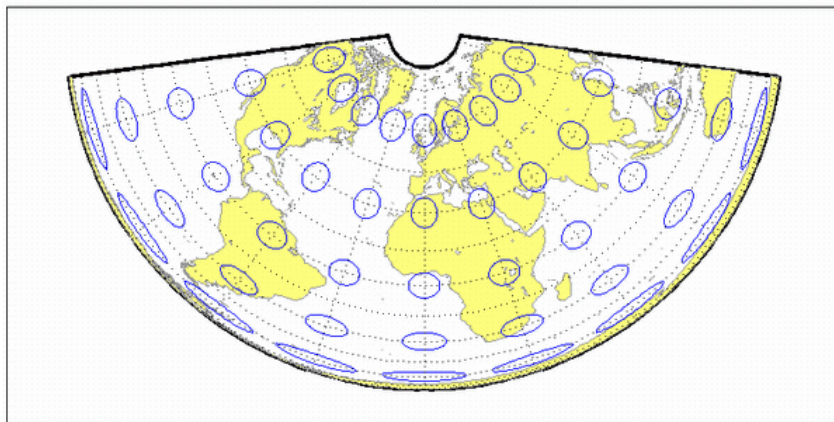
---

## 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`

# Equidistant Conic Projection – Standard

---

## Syntax

`eqdconicstd`

## Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Equally spaced concentric circular arcs centered on the point of meridional 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

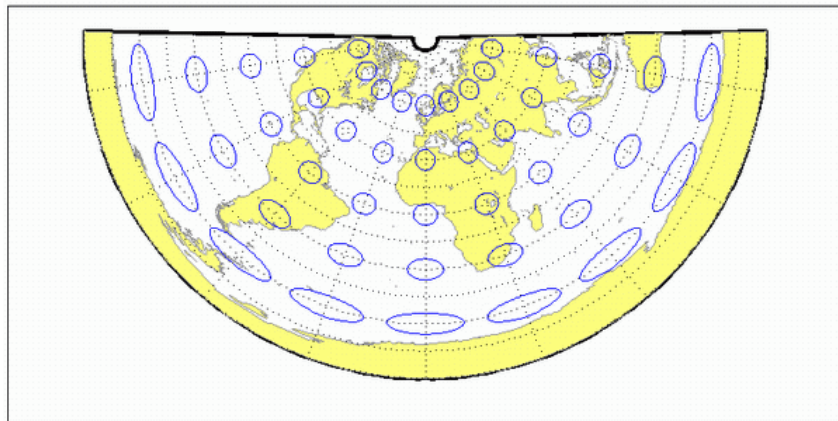
In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.

## 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

# Equidistant Cylindrical Projection

---

**Classification** Cylindrical

**Syntax** eqdcylin

**Graticule** Meridians: Equally spaced straight parallel lines more than half as long as the Equator.  
Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.  
Poles: Straight lines equal in length to the Equator.  
Symmetry: About any meridian or the Equator.

**Features** This is a projection onto a cylinder secant at the standard parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the standard parallels and is constant along any parallel and along the parallel of opposite sign.

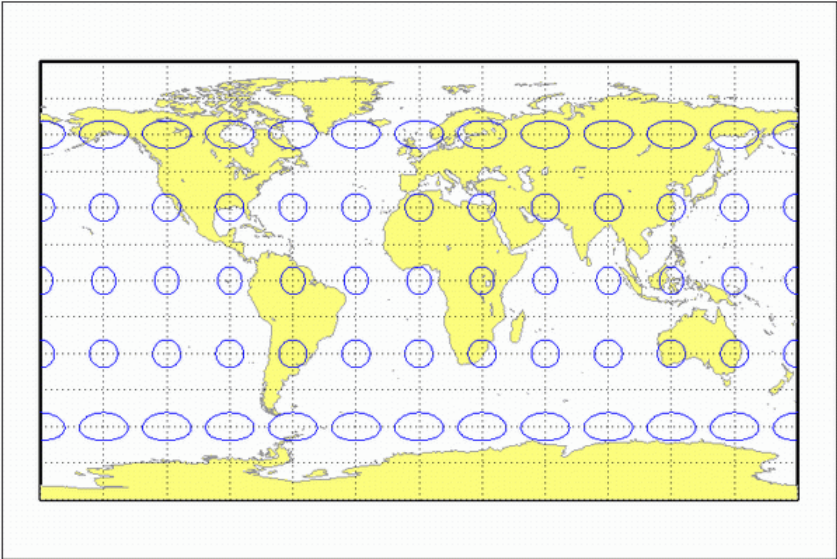
**Parallels** For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude can be chosen; the default is arbitrarily set to 30°.

**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 0°, and the Gall Isographic, with standard parallels at 45°N 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

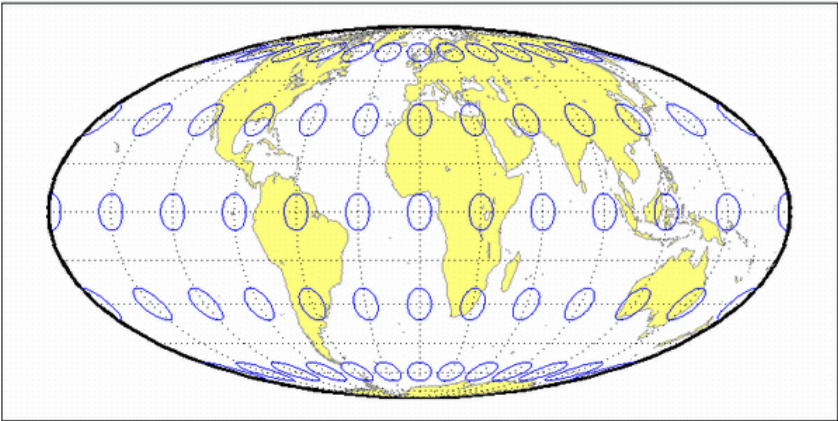




<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	fournier
<b>Graticule</b>	Meridians: Equally spaced elliptical curves converging at the poles. Parallels: Straight lines. Poles: Points. Symmetry: About the Equator and central meridian.
<b>Features</b>	This projection is equal-area. Scale is constant along any parallel or pair of parallels equidistant from the Equator. This projection is neither equidistant nor conformal.
<b>Parallels</b>	There is no standard parallel for this projection.
<b>Remarks</b>	This projection was first described in 1643 by Georges Fournier. This is actually his second projection, the Fournier II.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm('fournier', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Fournier Projection

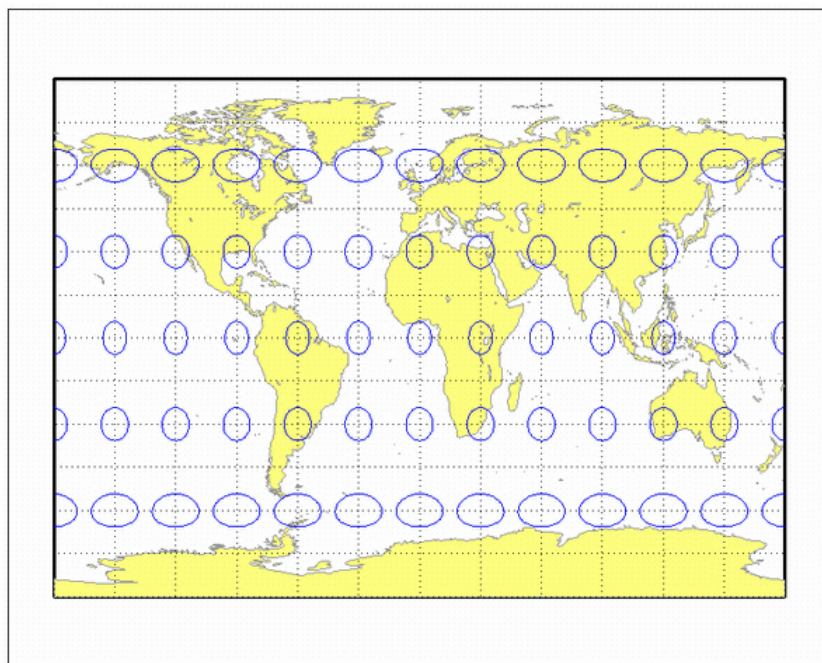
---



<b>Classification</b>	Cylindrical
<b>Syntax</b>	<code>giso</code>
<b>Graticule</b>	<p>Meridians: Equally spaced straight parallel lines more than half as long as the Equator.</p> <p>Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.</p> <p>Poles: Straight lines equal in length to the Equator.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>This is a projection onto a cylinder secant at the 45° parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the two standard parallels, and is constant along any parallel and along the parallel of opposite sign.</p>
<b>Parallels</b>	<p>For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45°.</p>
<b>Remarks</b>	<p>This projection is a specific case of the Equidistant Cylindrical projection, with standard parallels at 45°N and S.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('giso','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Gall Isographic Projection

---



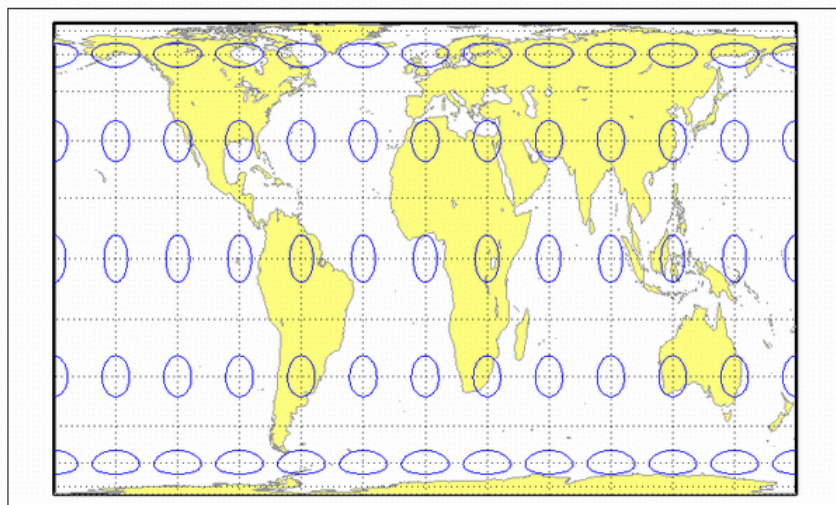
# Gall Orthographic Projection

---

<b>Classification</b>	Cylindrical
<b>Syntax</b>	gortho
<b>Graticule</b>	<p>Meridians: Equally spaced straight parallel lines.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.</p> <p>Poles: Straight lines equal in length to the Equator.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>This is an orthographic projection onto a cylinder secant at the 45° 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.</p>
<b>Parallels</b>	<p>For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45°.</p>
<b>Remarks</b>	<p>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 45°N and S. This projection is also known as the Peters projection.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('gortho', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Gall Orthographic Projection

---

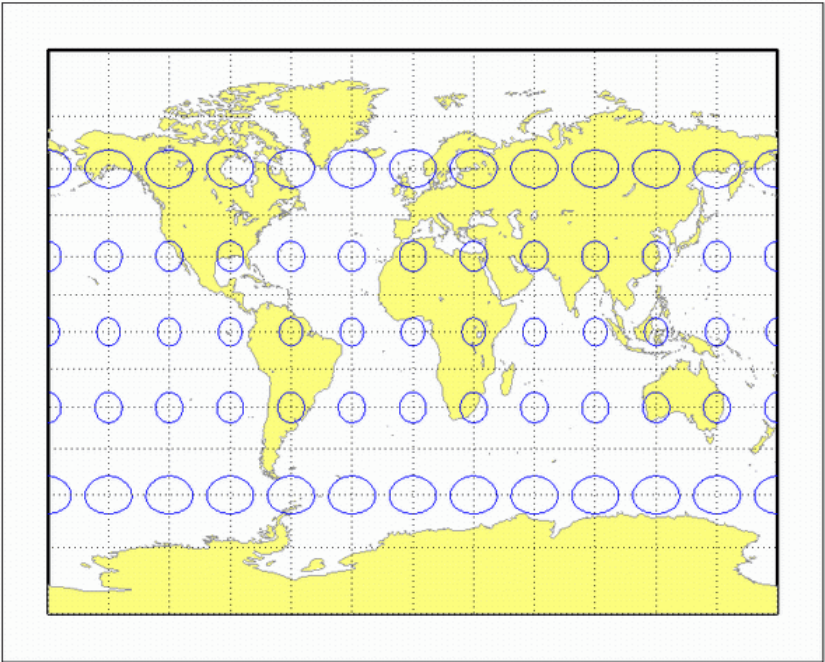


# Gall Stereographic Projection

---

<b>Classification</b>	Cylindrical
<b>Syntax</b>	gstereo
<b>Graticule</b>	<p>Meridians: Equally spaced straight parallel lines 0.77 as long as the Equator.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.</p> <p>Poles: Straight lines equal in length to the Equator.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>This is a perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at the 45° 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.</p>
<b>Parallels</b>	<p>For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45°.</p>
<b>Remarks</b>	<p>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 45°N and S.</p>
<b>Limitations</b>	<p>This projection is available only on the sphere.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('gstereo','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Gall Stereographic Projection



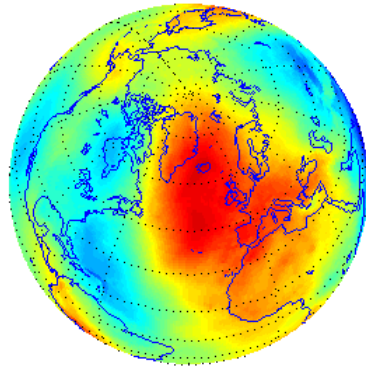


---

<b>Classification</b>	Spherical
<b>Syntax</b>	<code>globe</code>
<b>Graticule</b>	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.
<b>Features</b>	In the three-dimensional sense, <code>globe</code> 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'.
<b>Parallels</b>	The globe requires no standard parallels.
<b>Remarks</b>	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.
<b>Example</b>	<pre>% 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)</pre>

# Globe

---

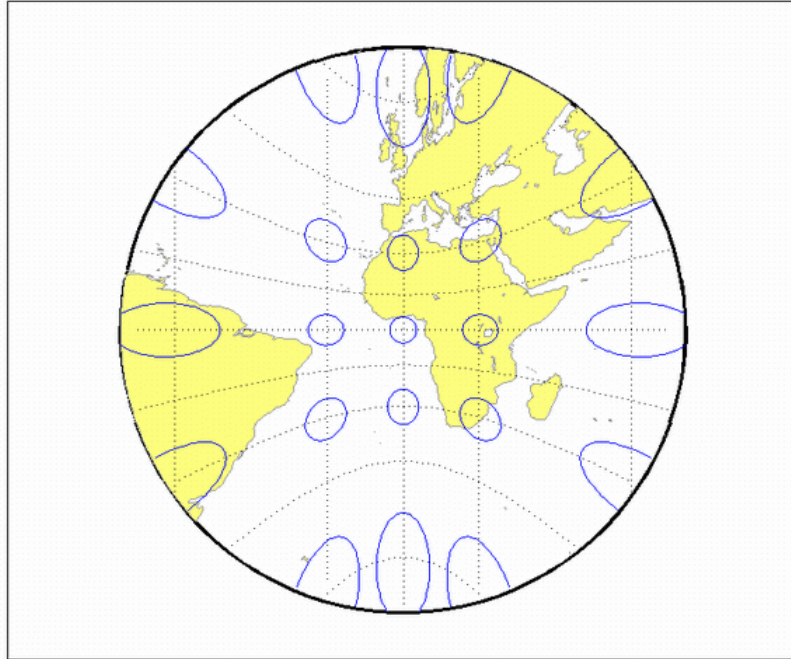


<b>Classification</b>	Azimuthal
<b>Syntax</b>	gnomonic
<b>Graticule</b>	<p>The graticule described is for a polar aspect.</p> <p>Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.</p> <p>Parallels: Unequally spaced circles centered on the central pole. Spacing increases rapidly away from this pole. The Equator and the opposite hemisphere cannot be shown</p> <p>Pole: The central pole is a point; the other pole is not shown.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	There are no standard parallels for azimuthal projections.
<b>Remarks</b>	<p>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 Gnostic or Central projection.</p>
<b>Limitations</b>	<p>This projection is available only on the sphere. Data greater than 65° distant from the center point is trimmed.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('gnomic', 'Frame', 'on', 'Grid', 'on');</pre>

# Gnomonic Projection

---

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);  
tissot;
```



# Goode Homolosine Projection

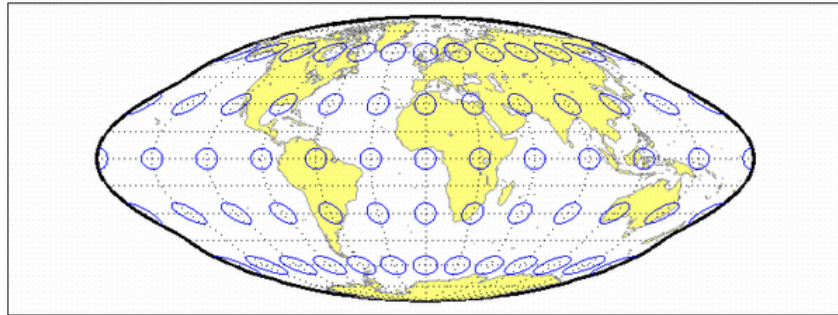
---

<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	goode
<b>Graticule</b>	<p>Central Meridian: Straight line 0.44 as long as the Equator.</p> <p>Other Meridians: Equally spaced sinusoidal curves between the 40°44'11.8" parallels and elliptical arcs elsewhere, all concave toward the central meridian. The result is a slight, visible bend in the meridians at 40°44'11.8" N and S.</p> <p>Parallels: Straight parallel lines, perpendicular to the central meridian. Equally spaced between the 40°44'11.8" parallels, with gradually decreasing spacing outside these parallels.</p> <p>Poles: Points.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along all parallels and the central meridian between 40°44'11.8" 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 40°44'11.8" N and S, and to that of the Mollweide projection elsewhere. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>This projection has one standard parallel, which is by definition fixed at 0°.</p>
<b>Remarks</b>	<p>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.</p>
<b>Limitations</b>	<p>This projection is available in an uninterrupted form only.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('goode', 'Frame', 'on', 'Grid', 'on');</pre>

# Goode Homolosine Projection

---

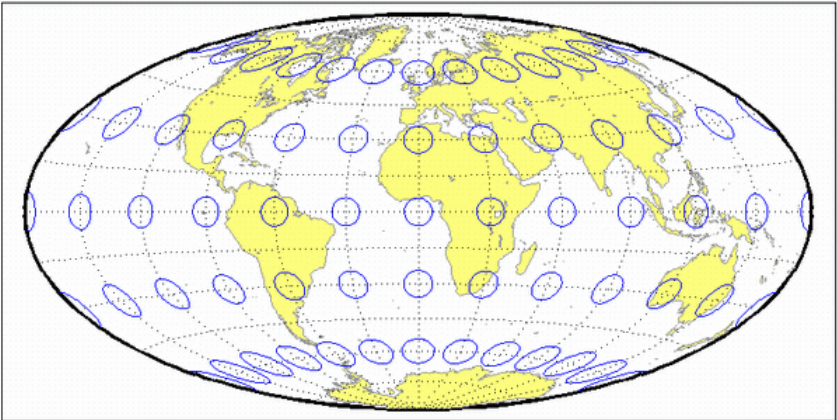
```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);  
tissot;
```



<b>Classification</b>	Modified Azimuthal
<b>Syntax</b>	hammer
<b>Graticule</b>	<p>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.</p> <p>Parallels: Equator is straight. Other parallels are complex curves, equally spaced along the central meridian, and concave toward the nearest pole.</p> <p>Poles: Points.</p> <p>Symmetry: About the Equator and central meridian.</p>
<b>Features</b>	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
<b>Parallels</b>	There is no standard parallel for this projection.
<b>Remarks</b>	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.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('hammer','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Hammer Projection

---





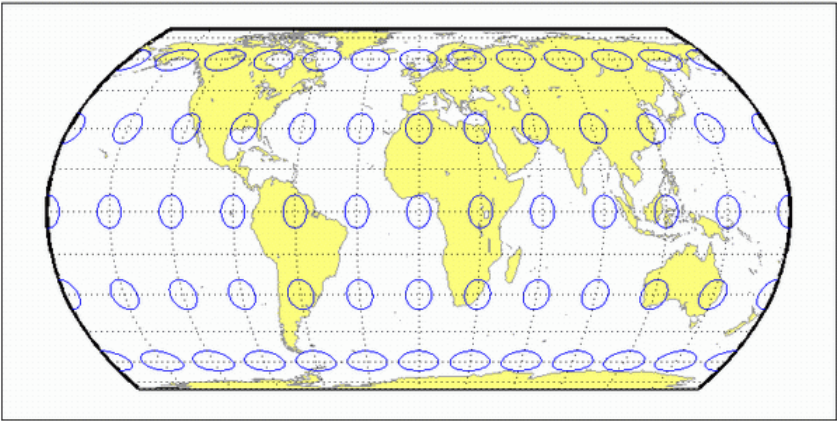
# Hatano Asymmetrical Equal-Area Projection

---

<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	hatano
<b>Graticule</b>	<p>Central Meridian: Straight line 0.48 as long as the Equator.</p> <p>Other Meridians: Equally spaced elliptical arcs concave toward the central meridian. The eccentricity of each ellipse changes at the Equator.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is not symmetrical about the Equator.</p> <p>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.</p> <p>Symmetry: About the central meridian but <i>not</i> the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along 40°42'N and 38°27'S, and is constant along any parallel but generally <i>not</i> between pairs of parallels equidistant from the Equator. It is free of distortion only along the central meridian at 40°42'N and 38°27'S. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>Because of the asymmetrical nature of this projection, two standard parallels must be specified. The standard parallels are by definition fixed at 40°42'N and 38°27'S.</p>
<b>Remarks</b>	<p>This projection was presented by Masataka Hatano in 1972.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('hatano', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Hatano Asymmetrical Equal-Area Projection

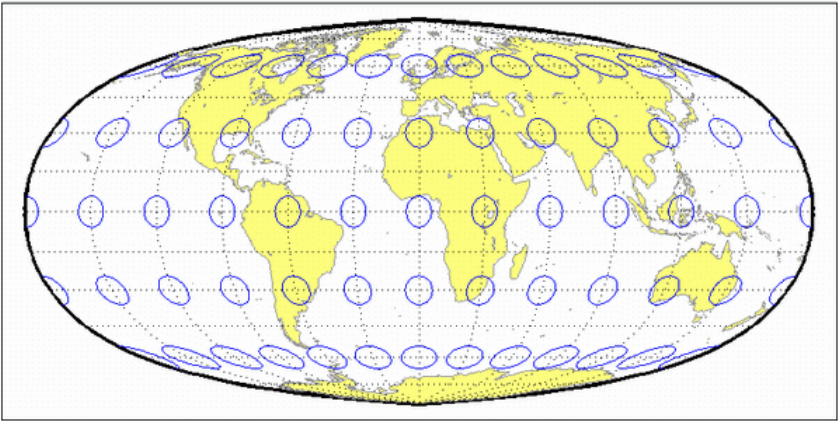
---



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	kavrsky5
<b>Graticule</b>	<p>Meridians: Complex curves converging at the poles. A sine function is used for <math>y</math>, but the meridians are not sine curves.</p> <p>Parallels: Unequally spaced straight lines.</p> <p>Poles: Points.</p> <p>Symmetry: About the Equator and the central meridian.</p>
<b>Features</b>	This is an equal-area projection. Scale is true along the fixed standard parallels at $35^\circ$ , and 0.9 true along the Equator. This projection is neither conformal nor equidistant.
<b>Parallels</b>	The fixed standard parallels are at $35^\circ$ .
<b>Remarks</b>	This projection was described by V. V. Kavraisky in 1933.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm('kavrsky5', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Kavraisky V Projection

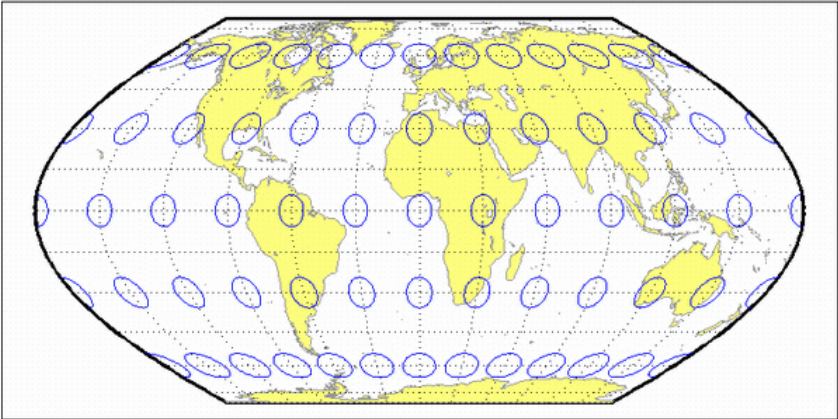
---



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	kavrsky6
<b>Graticule</b>	Central Meridian: Straight line half the length of the Equator. Meridians: Sine curves (60° segments). Parallels: Unequally spaced straight lines. Poles: Straight lines half the length of the Equator. Symmetry: About the Equator and the central meridian.
<b>Features</b>	This is an equal-area projection. Scale is constant along any parallel or pair of equidistant parallels. This projection is neither conformal nor equidistant.
<b>Parallels</b>	There are no standard parallels for this projection.
<b>Remarks</b>	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.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('kavrsky6', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Kavraisky VI Projection

---



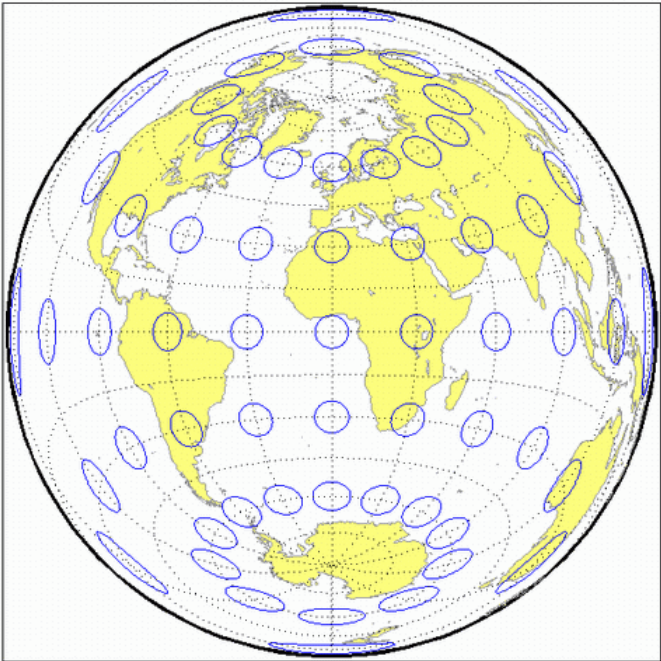
# Lambert Azimuthal Equal-Area Projection

---

<b>Classification</b>	Azimuthal
<b>Syntax</b>	eqaazim
<b>Graticule</b>	<p>The graticule described is for a polar aspect.</p> <p>Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.</p> <p>Parallels: Unequally spaced circles centered on the central pole. The entire Earth can be shown. Spacing decreases away from the central pole.</p> <p>Pole: The central pole is a point; the other pole is a bounding circle with 1.41 the radius of the Equator.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p>This nonperspective projection is equal-area. Only the center point is free of distortion, but distortion is moderate within 90° 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.</p>
<b>Parallels</b>	There are no standard parallels for azimuthal projections.
<b>Remarks</b>	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.
<b>Limitations</b>	Data greater than 160° distant from the center point is trimmed.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('eqaazim','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Lambert Azimuthal Equal-Area Projection

---





# Lambert Conformal Conic Projection

---

<b>Classification</b>	Conic
<b>Syntax</b>	lambert
<b>Graticule</b>	<p>Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.</p> <p>Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.</p> <p>Poles: The pole nearest a standard parallel is a point, the other cannot be shown.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>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].</p>
<b>Remarks</b>	<p>This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.</p>

# Lambert Conformal Conic Projection

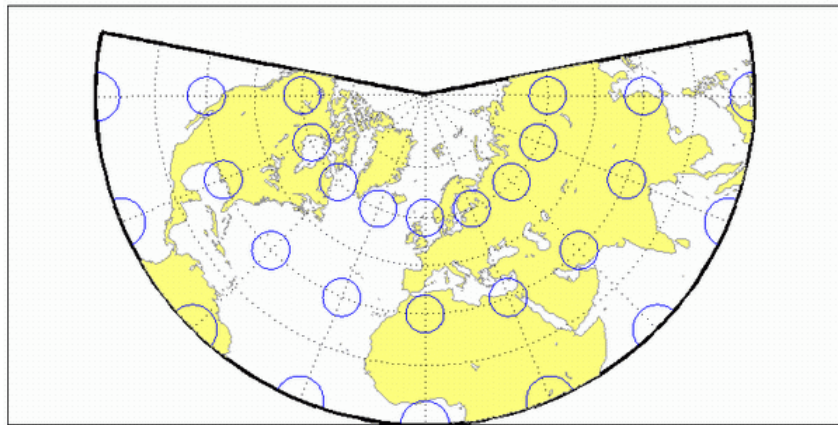
---

## 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`

# Lambert Conformal Conic Projection – Standard

---

<b>Classification</b>	Conic
<b>Syntax</b>	<code>lambertstd</code>
<b>Graticule</b>	<p>Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.</p> <p>Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.</p> <p>Poles: The pole nearest a standard parallel is a point, the other cannot be shown.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p><code>lambertstd</code> implements the Lambert Conformal Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See <code>lambert</code> for an alternative implementation based on rotating the authalic sphere.</p> <p>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.</p>
<b>Parallels</b>	<p>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].</p>

# Lambert Conformal Conic Projection – Standard

---

## Remarks

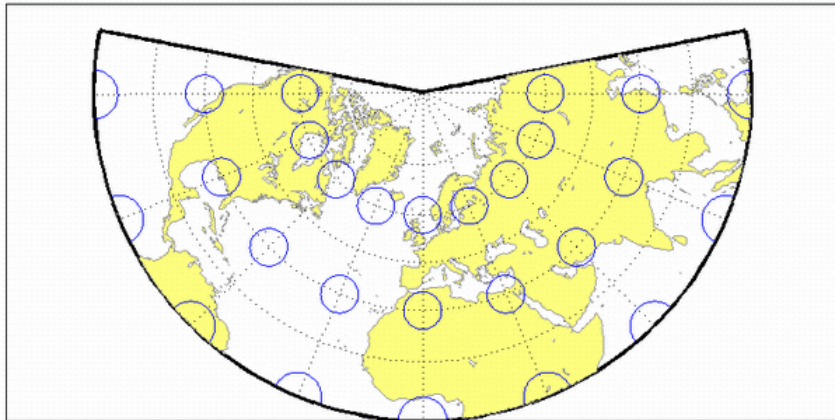
This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.

## 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](#)

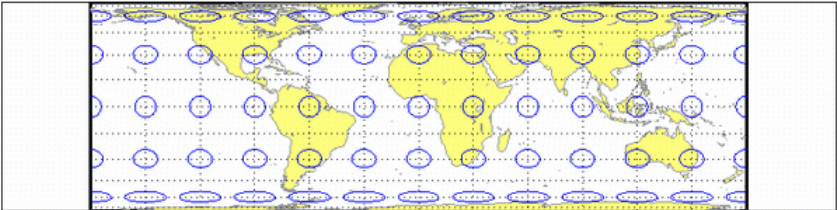
# Lambert Equal-Area Cylindrical Projection

---

<b>Classification</b>	Cylindrical
<b>Syntax</b>	<code>lambcyln</code>
<b>Grtaticule</b>	<p>Meridians: Equally spaced straight parallel lines 0.32 as long as the Equator.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.</p> <p>Poles: Straight lines equal in length to the Equator.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>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 0°.</p>
<b>Remarks</b>	<p>This projection is named for Johann Heinrich Lambert and is a special form of the Equal-Area Cylindrical projection tangent at the Equator.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('lambcyn','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Lambert Equal-Area Cylindrical Projection

---



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	loximuth
<b>Graticule</b>	<p>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 40°.</p> <p>Other Meridians: Complex curves intersecting at the poles and concave toward the central meridian.</p> <p>Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.</p> <p>Poles: Points.</p> <p>Symmetry: About the central meridian. Symmetry about the Equator only when it is the central latitude.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>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 0°.</p>

# Loximuthal Projection

---

## Remarks

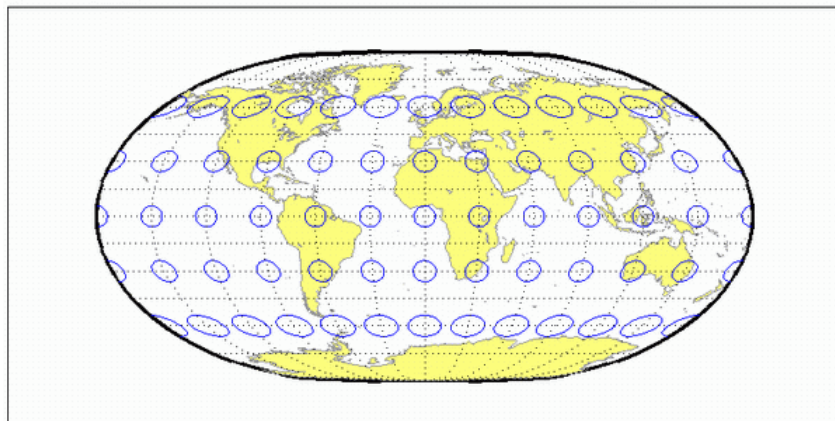
This projection was presented by Karl Siemon in 1935 and independently by Waldo R. Tobler in 1966. The Bordone Oval projection of 1520 was very similar to the Equator-centered Loximuthal.

## 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;
```





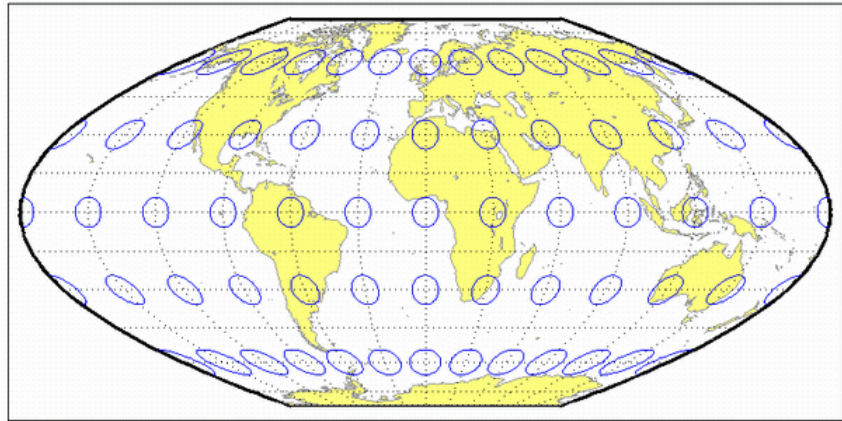
# McBryde-Thomas Flat-Polar Parabolic Projection

---

<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	flatplrp
<b>Graticule</b>	<p>Central Meridian: Straight line 0.48 as long as the Equator.</p> <p>Other Meridians: Equally spaced parabolic curves concave toward the central meridian.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.</p> <p>Poles: Lines one-third as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along the 45°30' 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 45°30' parallels. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>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 45°30'.</p>
<b>Remarks</b>	<p>This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('flatplrp', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# McBryde-Thomas Flat-Polar Parabolic Projection

---



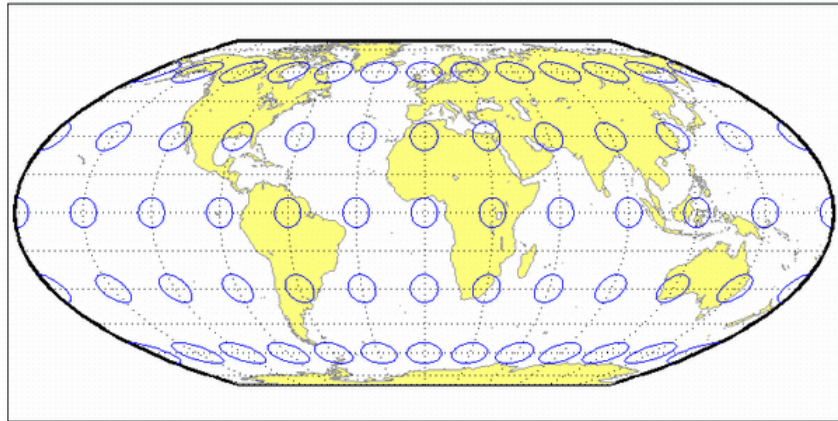
# McBryde-Thomas Flat-Polar Quartic Projection

---

<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	flatplr <sub>q</sub>
<b>Graticule</b>	<p>Central Meridian: Straight line 0.45 as long as the Equator.</p> <p>Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.</p> <p>Poles: Lines one-third as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along the 33°45' 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 33°45' parallels. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>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 33°45'.</p>
<b>Remarks</b>	<p>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.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm('flatplr<sub>q</sub>', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# McBryde-Thomas Flat-Polar Quartic Projection

---



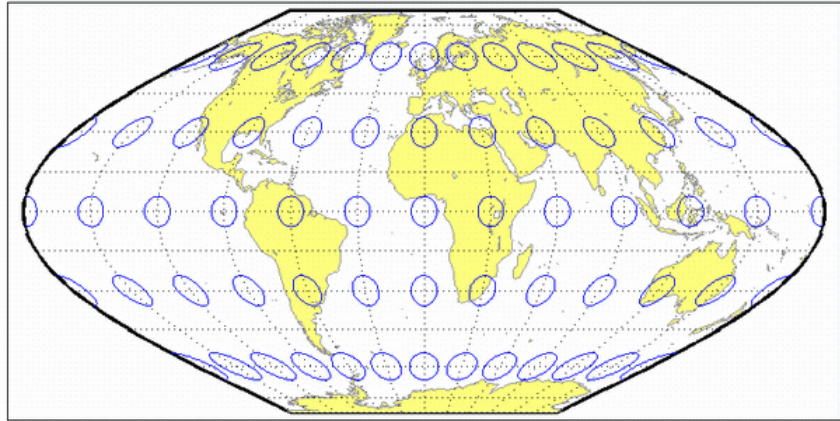
# McBryde-Thomas Flat-Polar Sinusoidal Projection

---

<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	flatplrs
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.</p> <p>Poles: Lines one-third as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This projection is equal-area. Scale is true along the 55°51' 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 55°51' parallels. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>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 55°51'.</p>
<b>Remarks</b>	<p>This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('flatplrs', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# McBryde-Thomas Flat-Polar Sinusoidal Projection

---



<b>Classification</b>	Cylindrical
<b>Syntax</b>	mercator
<b>Graticule</b>	<p>Meridians: Equally spaced straight parallel lines.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.</p> <p>Poles: Cannot be shown.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>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.</p> <p>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.</p>
<b>Parallels</b>	<p>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 86° may be chosen; the default is arbitrarily set to 0°.</p>
<b>Remarks</b>	<p>The Mercator projection is named for Gerardus Mercator, who presented it <i>for navigation</i> 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,</p>

# Mercator Projection

---

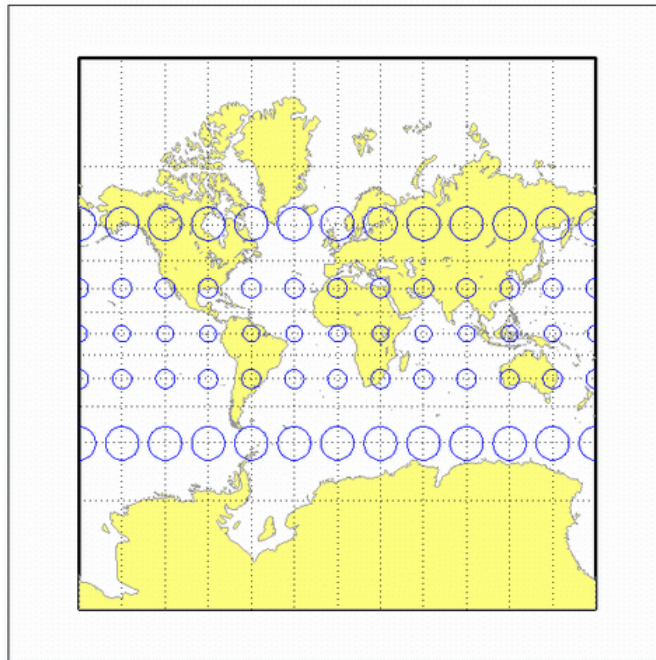
called the Wright projection, after Edward Wright, who developed the mathematics behind the projection in 1599.

## Limitations

Data at latitudes greater than  $86^\circ$  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;
```





<b>Classification</b>	Cylindrical
<b>Syntax</b>	milller
<b>Graticule</b>	<p>Meridians: Equally spaced straight parallel lines 0.73 as long as the Equator.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles, less rapidly than that of the Mercator projection.</p> <p>Poles: Straight lines equal in length to the Equator.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>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.</p> <p>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.</p>
<b>Parallels</b>	<p>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 0°.</p>
<b>Remarks</b>	<p>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.</p>

# Miller Cylindrical Projection

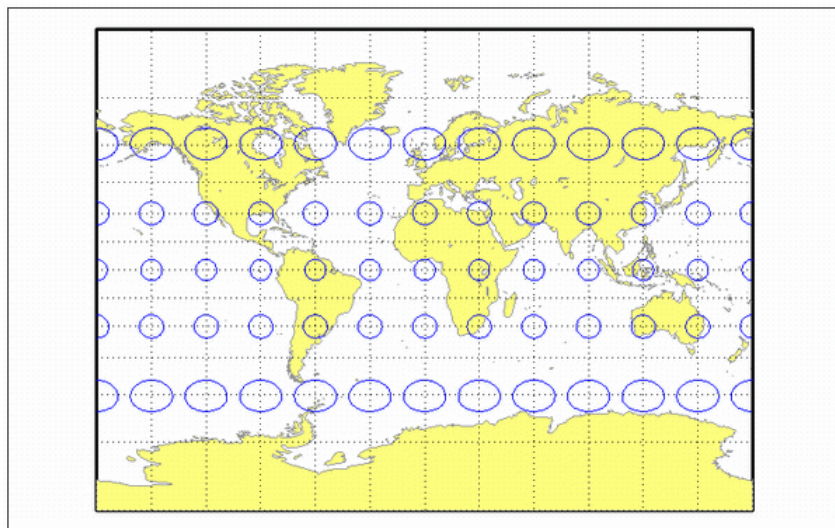
---

## Limitations

This projection is available only for the sphere.

## Example

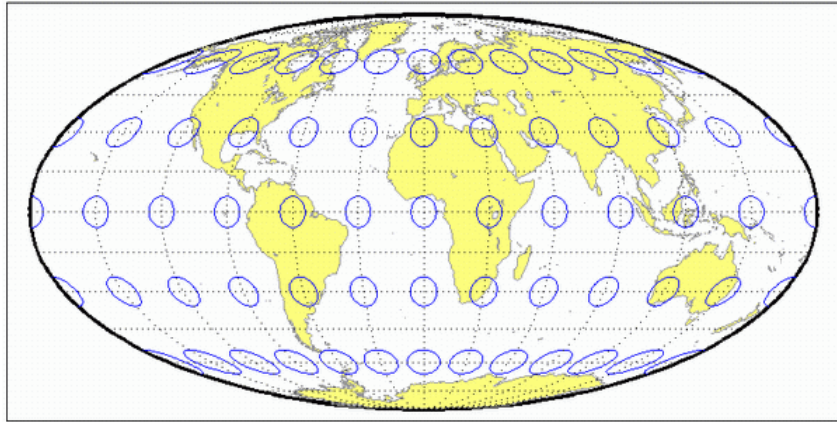
```
landareas = shaperead('landareas.shp','UseGeoCoords',true);  
axesm('miller','Frame','on','Grid','on');  
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);  
tissot;
```



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	mollweid
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Meridians 90° 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.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator, but the spacing changes gradually.</p> <p>Poles: Points.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>This is an equal-area projection. Scale is true along the 40°44' 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 40°44' parallels intersect the central meridian. This projection is not conformal or equidistant.</p>
<b>Parallels</b>	<p>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 40°44'.</p>
<b>Remarks</b>	<p>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.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('mollweid', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Mollweide Projection

---



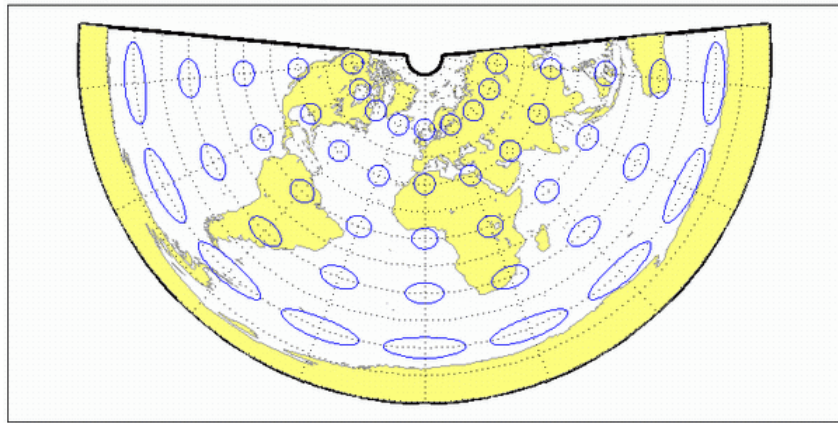
# Murdoch I Conic Projection

---

<b>Classification</b>	Conic
<b>Syntax</b>	murdoch1
<b>Graticule</b>	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.
<b>Features</b>	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.
<b>Parallels</b>	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].
<b>Remarks</b>	Described by Patrick Murdoch in 1758.
<b>Limitations</b>	This projection is available only for the sphere. Longitude data greater than 135° east or west of the central meridian is trimmed.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('murdoch1','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Murdoch I Conic Projection

---



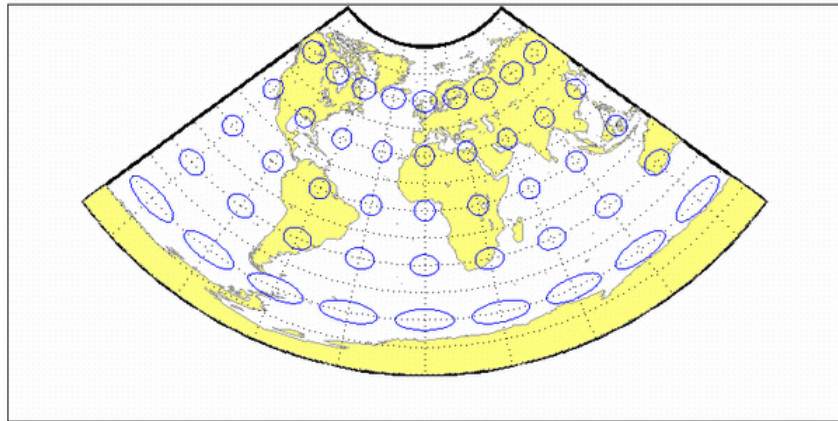
# Murdoch III Minimum Error Conic Projection

---

<b>Classification</b>	Conic
<b>Syntax</b>	murdoch3
<b>Graticule</b>	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.
<b>Features</b>	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.
<b>Parallels</b>	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].
<b>Remarks</b>	Described by Patrick Murdoch in 1758, with errors corrected by Everett in 1904.
<b>Limitations</b>	This projection is available only for the sphere. Longitude data greater than 135° east or west of the central meridian is trimmed.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('murdoch3','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Murdoch III Minimum Error Conic Projection

---





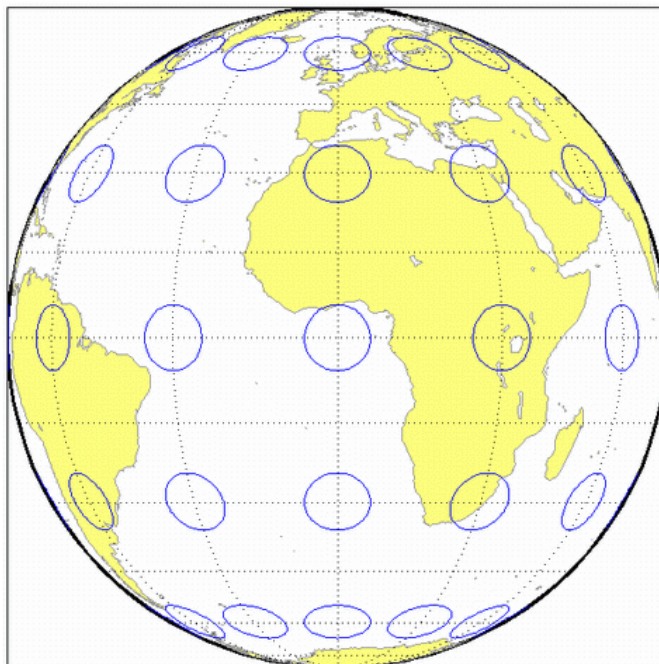
<b>Classification</b>	Azimuthal
<b>Syntax</b>	ortho
<b>Graticule</b>	<p>The graticule described is for a polar aspect.</p> <p>Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.</p> <p>Parallels: Unequally spaced circles centered on the central pole. Spacing decreases away from this pole. The opposite hemisphere cannot be shown.</p> <p>Pole: The central pole is a point; the other pole is not shown.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	There are no standard parallels for azimuthal projections.
<b>Remarks</b>	This projection appears to have been developed by the Egyptians and Greeks by the second century B.C.
<b>Limitations</b>	This projection is available only for the sphere. Data greater than 89° distant from the center point is trimmed.

# Orthographic Projection

---

## Example

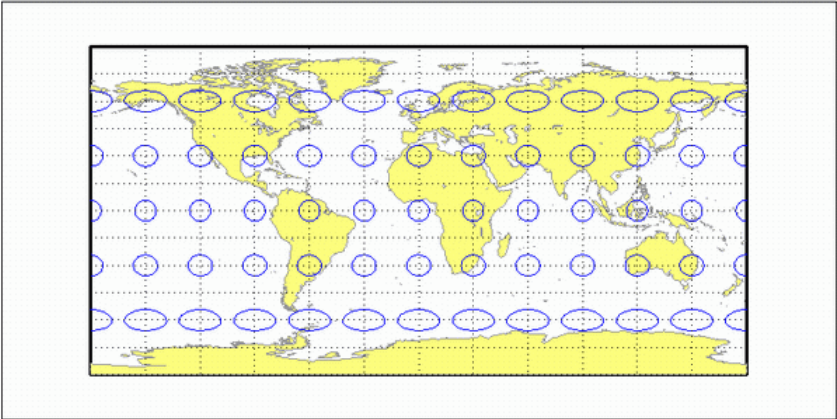
```
landareas = shaperead('landareas.shp','UseGeoCoords',true);  
axesm('ortho','Frame','on','Grid','on');  
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);  
tissot;
```



<b>Classification</b>	Cylindrical
<b>Syntax</b>	pcarree
<b>Graticule</b>	<p>Meridians: Equally spaced straight parallel lines half as long as the Equator.</p> <p>Parallels: Equally spaced straight parallel lines, perpendicular to and having the same spacing as the meridians.</p> <p>Poles: Straight lines equal in length to the Equator.</p> <p>Symmetry: About any meridian or the Equator.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>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 0°.</p>
<b>Remarks</b>	<p>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 Eratosthenes, 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.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('pcarree','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Plate Carrée Projection

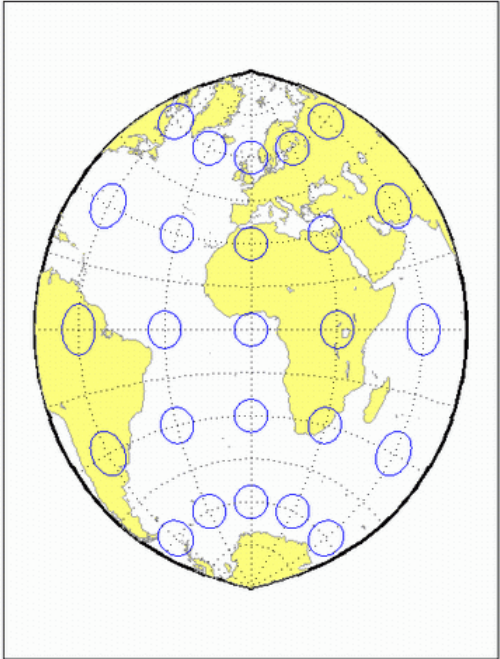
---



<b>Classification</b>	Polyconic
<b>Syntax</b>	polycon
<b>Graticule</b>	<p>Central Meridian: A straight line.</p> <p>Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.</p> <p>Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.</p> <p>Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.</p> <p>Symmetry: About the Equator or the central meridian.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>By definition, this projection has no standard parallels, since every parallel is a <i>standard parallel</i>.</p>
<b>Remarks</b>	<p>This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.</p>
<b>Limitations</b>	<p>Longitude data greater than 75° east or west of the central meridian is trimmed.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm ('polycon', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);</pre>

# Polyconic Projection

tissot;



**See Also**

polyconstd

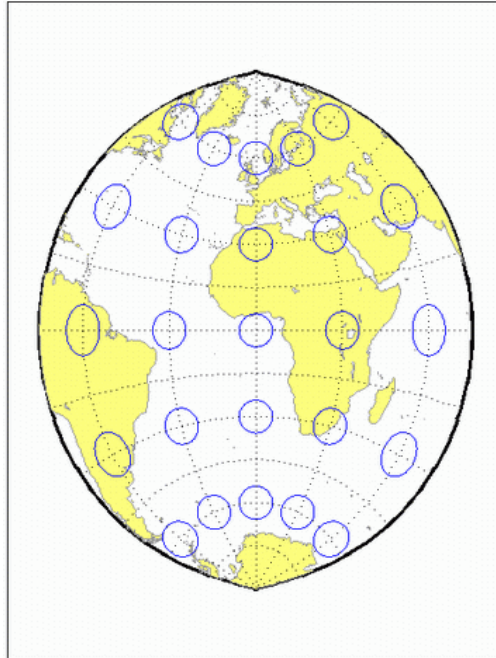
<b>Classification</b>	Polyconic
<b>Syntax</b>	<code>polyconstd</code>
<b>Graticule</b>	<p>Central Meridian: A straight line.</p> <p>Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.</p> <p>Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.</p> <p>Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.</p> <p>Symmetry: About the Equator or the central meridian.</p>
<b>Features</b>	<p><code>polyconstd</code> implements the Polyconic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See <code>polycon</code> for an alternative implementation based on rotating the rectifying sphere.</p> <p>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.</p>
<b>Parallels</b>	By definition, this projection has no standard parallels, since every parallel is a <i>standard parallel</i> .
<b>Remarks</b>	This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.
<b>Limitations</b>	Longitude data greater than 75° east or west of the central meridian is trimmed.

# Polyconic Projection – Standard

---

## Example

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



## See Also

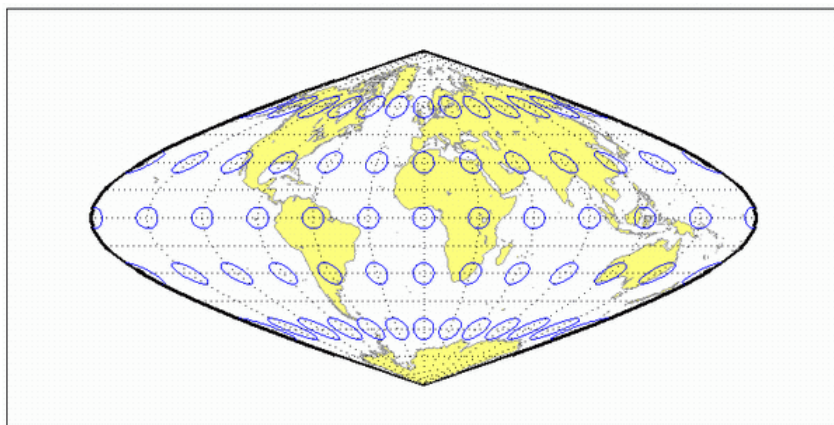
[polycon](#)



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	putnins5
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced portions of hyperbolas intersecting at the poles and concave toward the central meridian.</p> <p>Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.</p> <p>Poles: Points.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>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'.</p>
<b>Remarks</b>	<p>This projection was presented by Reinholds V. Putnins in 1934.</p>
<b>Limitations</b>	<p>This projection is available only for the sphere.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('putnins5','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Putnins P5 Projection

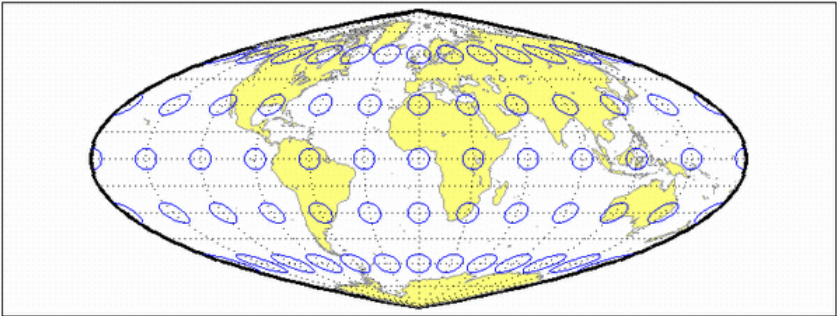
---



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	quartic
<b>Gromatic</b>	<p>Central Meridian: Straight line 0.45 as long as the Equator.</p> <p>Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.</p> <p>Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes gradually and is greatest near the Equator.</p> <p>Poles: Points.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>This projection has one standard parallel, which is by definition fixed at 0°.</p>
<b>Remarks</b>	<p>This projection was presented by Karl Siemon in 1937 and independently by Oscar Sherman Adams in 1945.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('quartic', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Quartic Authalic Projection

---

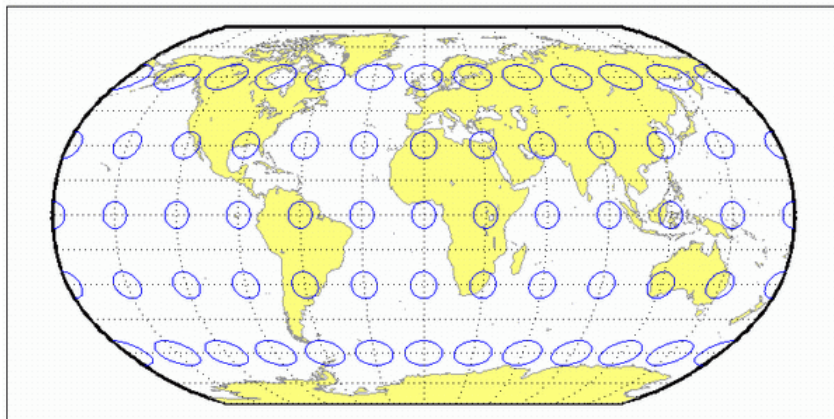


<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	robinson
<b>Graticule</b>	<p>Central Meridian: Straight line 0.51 as long as the Equator.</p> <p>Other Meridians: Equally spaced, resemble elliptical arcs and are concave toward the central meridian.</p> <p>Parallels: Straight parallel lines, perpendicular to the central meridian. Spacing is equal between the 38° parallels, decreasing outside these limits.</p> <p>Poles: Lines 0.53 as long as the Equator.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>Scale is true along the 38° 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 45° of the center and along the Equator. This projection is not equal-area, conformal, or equidistant; however, it is considered to <i>look right</i> 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.</p>
<b>Parallels</b>	<p>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 38°.</p>
<b>Remarks</b>	<p>This projection was presented by Arthur H. Robinson in 1963, and is also called the Orthophanic projection, which means <i>right appearing</i>.</p>
<b>Limitations</b>	<p>This projection is available only for the sphere.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('robinson', 'Frame', 'on', 'Grid', 'on');</pre>

# Robinson Projection

---

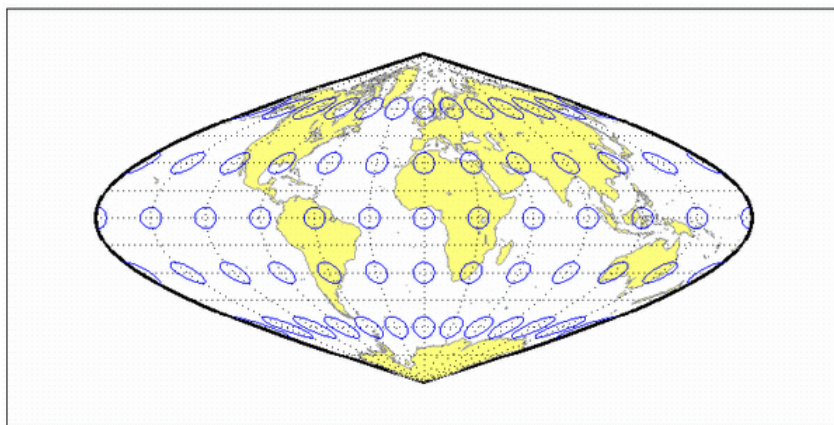
```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);  
tissot;
```



<b>Classification</b>	Pseudocylindrical
<b>Syntax</b>	sinusoid
<b>Graticule</b>	<p>Central Meridian: Straight line half as long as the Equator.</p> <p>Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.</p> <p>Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.</p> <p>Poles: Points.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>This projection has one standard parallel, which is by definition fixed at 0°.</p>
<b>Remarks</b>	<p>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.</p>
<b>Example</b>	<pre>landareas = shaperead('landareas.shp', 'UseGeoCoords', true); axesm ('sinusoid', 'Frame', 'on', 'Grid', 'on'); geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]); tissot;</pre>

# Sinusoidal Projection

---

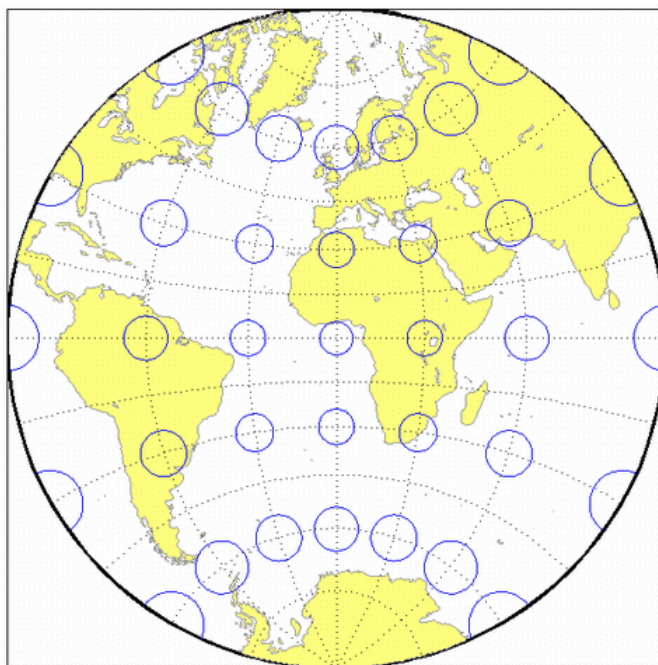




<b>Classification</b>	Azimuthal
<b>Syntax</b>	stereo
<b>Graticule</b>	<p>The graticule described is for a polar aspect.</p> <p>Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.</p> <p>Parallels: Unequally spaced circles centered on the central pole. Spacing increases gradually away from this pole. The opposite hemisphere cannot be shown</p> <p>Pole: The central pole is a point; the other pole is not shown.</p> <p>Symmetry: About any meridian.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	There are no standard parallels for azimuthal projections.
<b>Remarks</b>	The polar aspect of this projection appears to have been developed by the Egyptians and Greeks by the second century B.C.
<b>Limitations</b>	Data greater than 90° distant from the center point is trimmed.
<b>Example</b>	<pre>landareas = shaperead('landareas.shp','UseGeoCoords',true); axesm('stereo','Frame','on','Grid','on'); geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]); tissot;</pre>

# Stereographic Projection

---



# 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.

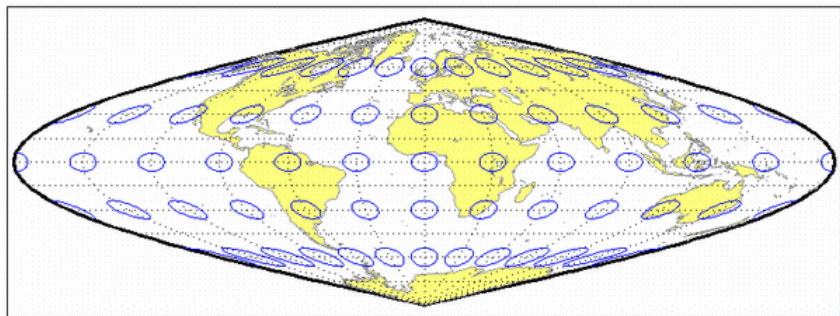
**Parallels** There are no standard parallels for this projection.

**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 nearly correct.

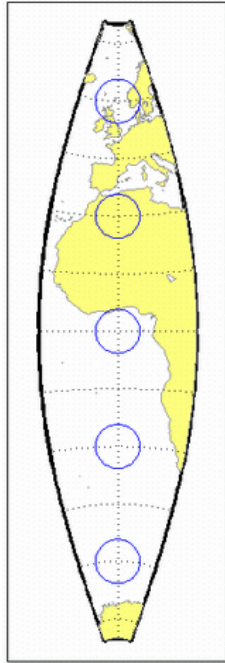
**Remarks** The uniformity of scale along its central meridian makes Transverse Mercator an excellent choice for mapping areas that are elongated north-to-south. Its best known application is the definition of Universal Transverse Mercator (UTM) coordinates. Each UTM zone spans only 6 degrees of longitude, 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;
```

# Transverse Mercator Projection

---



# 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 37°24' 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 37°24'.

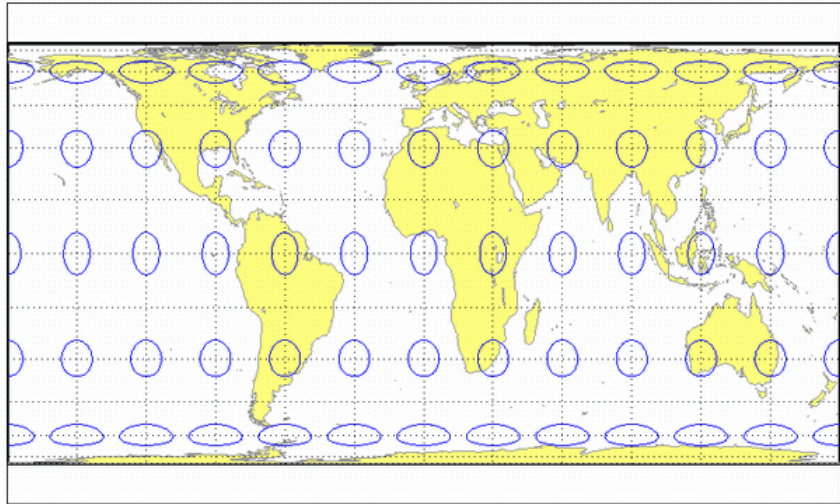
**Remarks** This projection is named for Trystan Edwards, who presented it in 1953. It is a special form of the Equal-Area Cylindrical projection secant at 37°24'N and S.

**Example**

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

# Trystan Edwards Cylindrical Projection

---



# 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 84° N to 90° N, and "South" for latitudes 80° S to 90° 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

---

<b>Classification</b>	Cylindrical
<b>Syntax</b>	utm
<b>Graticule</b>	<p>Meridians: Complex curves concave toward the central meridian.</p> <p>Parallels: Complex curves concave toward the nearest pole.</p> <p>Poles: Not shown.</p> <p>Symmetry: About the central meridian or the Equator.</p>
<b>Features</b>	<p>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.</p>
<b>Parallels</b>	<p>There are no standard parallels for this projection. There are two lines of zero distortion by virtue of the scale factor.</p>
<b>Remarks</b>	<p>The UTM system divides the world between 80° S and 84° 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.</p> <p>The UTM system was introduced in the 1940s by the U.S. Army. It is widely used in topographic and military mapping.</p>

# 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”).

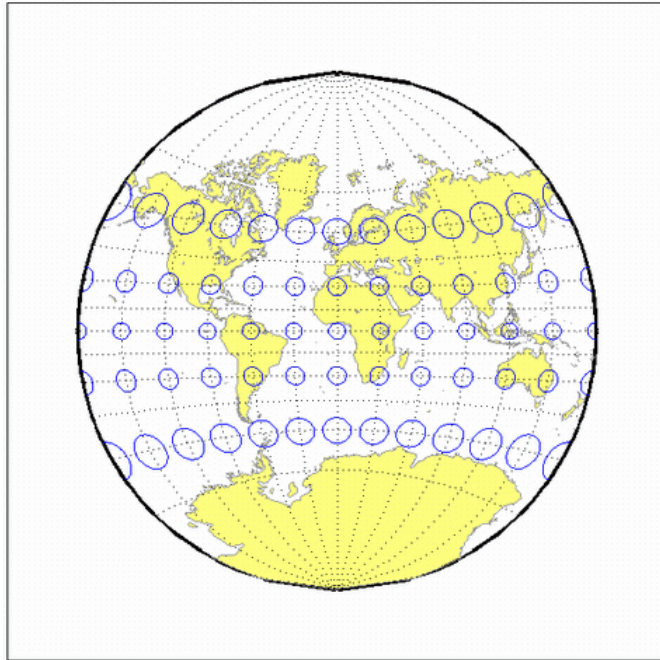
**Limitations** This projection is available only for the sphere.

**Example**

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('vgrint1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```

# Van der Grinten I Projection

---



# 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.

**Parallels** The standard parallel contains the observation altitude above the surface in the same units as the geoid semi-major axis.

**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.

**Limitations** This projection is available only for the sphere. Data more distant than the limit of visibility is trimmed.

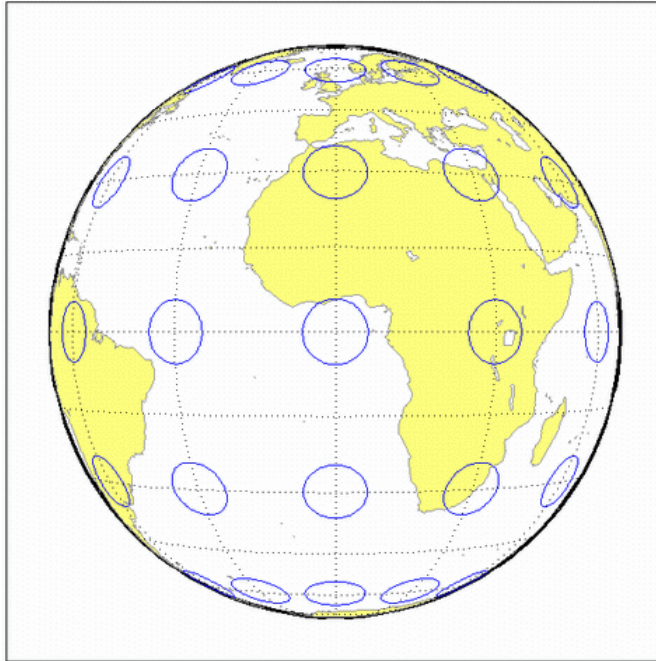
**Example**

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

# Vertical Perspective Azimuthal Projection

---

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 103°55' 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 42°59' 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 42°59' 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 42°59'.

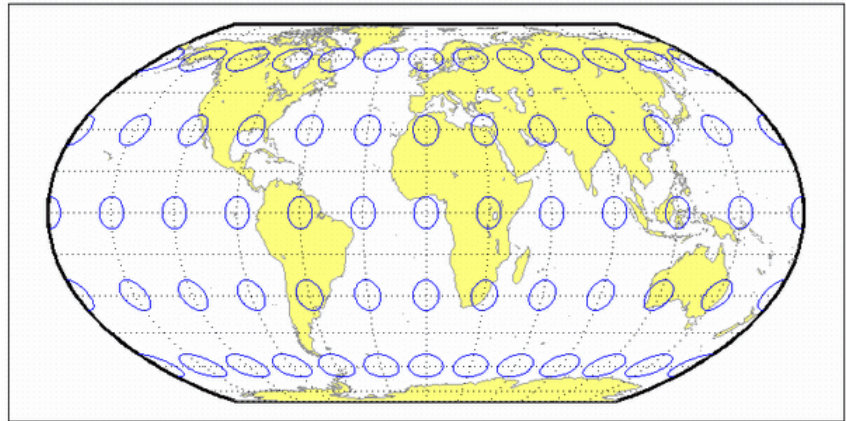
**Remarks** This projection was presented by Karlheinz Wagner in 1932.

**Example**

```
landareas = shaperead('landareas.shp', 'UseGeoCoords', true);  
axism ('wagner4', 'Frame', 'on', 'Grid', 'on');  
geoshow(landareas, 'FaceColor', [1 1 .5], 'EdgeColor', [.6 .6 .6]);  
tissot;
```

# Wagner IV Projection

---



# 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*.

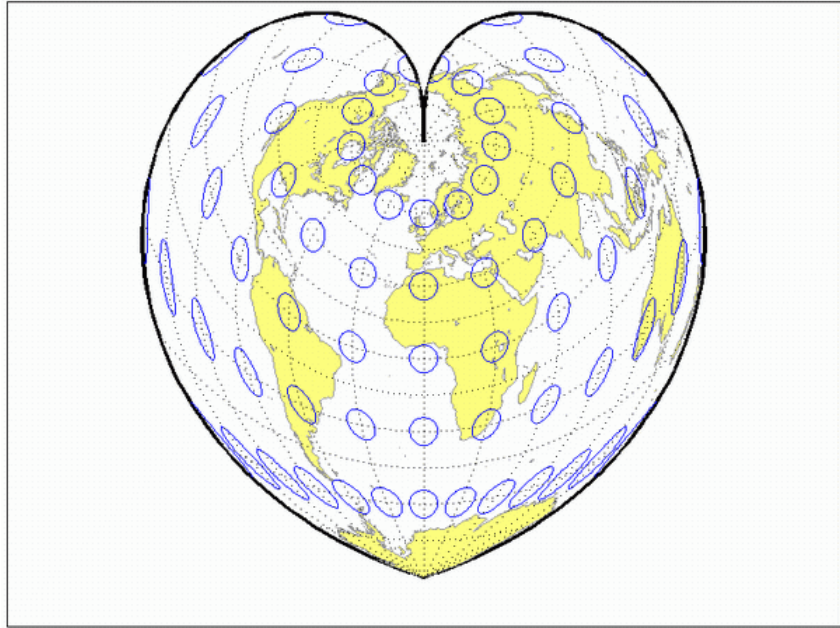
**Parallels** The standard parallel for this projection is set to 90° N.

**Remarks** This projection was developed by Johannes Stabius (Stab) about 1500 and was promoted by Johannes Werner in 1514. It is also called the Stab-Werner projection.

**Example**

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





# Wetch Cylindrical Projection

---

**Classification** Cylindrical

**Syntax** wetch

**Graticule** Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if  $90^\circ$  from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

**Features** This is a perspective projection from the center of the Earth onto a cylinder tangent to the central meridian. It is not equal-area, equidistant, or conformal. Scale is true along the central meridian and constant between two points equidistant in  $x$  and  $y$  from the central meridian. There is no distortion along the central meridian, but it increases rapidly away from the central meridian in the  $y$ -direction.

**Parallels** For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, which is the transverse aspect of the Central Cylindrical, the standard parallel *of the base projection* is by definition fixed at  $0^\circ$ .

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

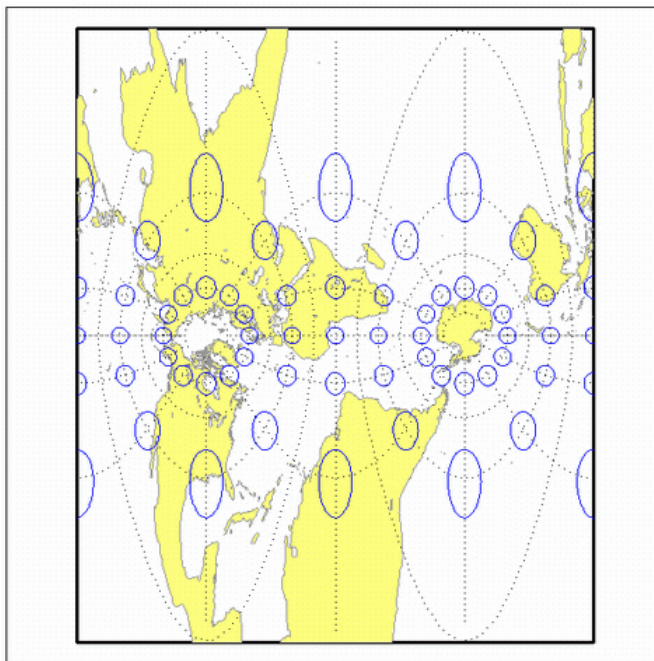
**Limitations** This projection is available only for the sphere. To prevent large  $y$ -values from dominating the display, data at  $y$ -values that would correspond to latitudes of greater than  $75^\circ$  in the normal aspect of the Central Cylindrical projection is trimmed.

**Example**

```
landareas = shaperead('landareas.shp', 'UseGeoCoords', true);  
axesm ('wetch', 'Frame', 'on', 'Grid', 'on');
```

# Wetck Cylindrical Projection

```
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.

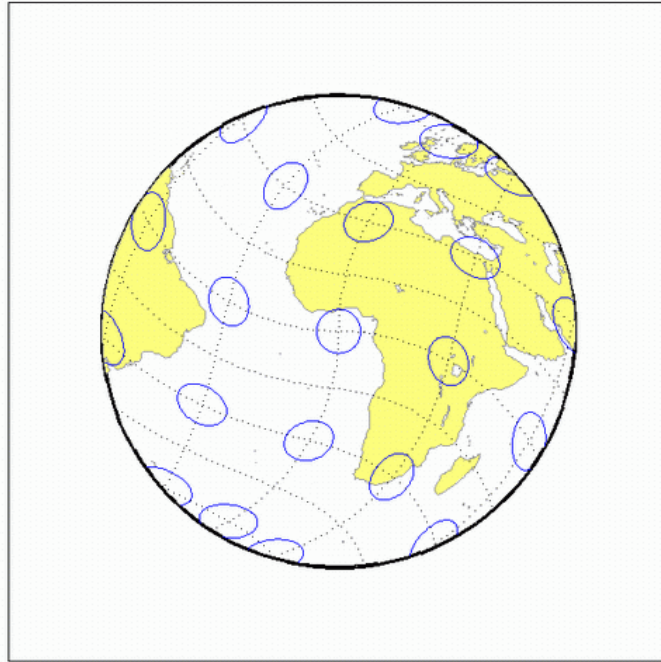
**Parallels** There are no standard parallels for azimuthal projections.

**Remarks** This projection was presented by H. Wiechel in 1879.

**Limitations** Data greater than 65° 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 arithmetical 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.

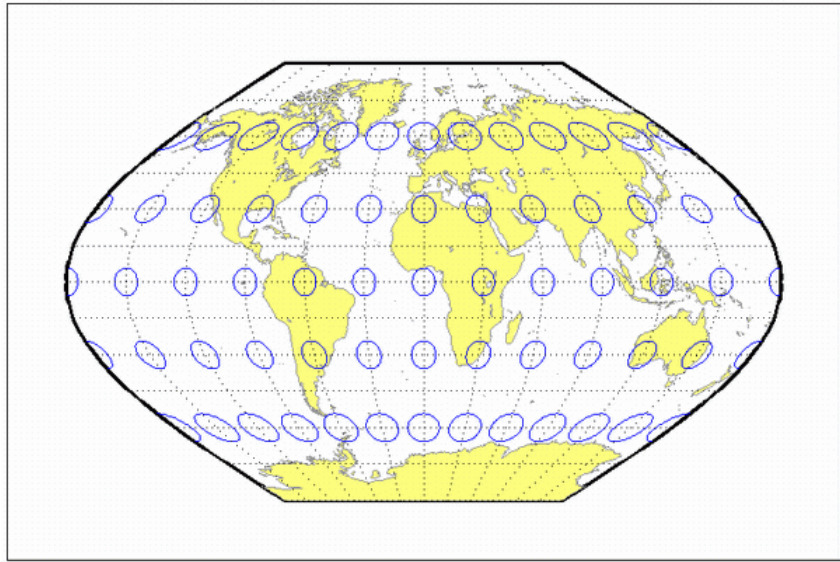
**Parallels** For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. Any latitude may be chosen; the default is set to  $50^{\circ}28'$ .

**Remarks** This projection was developed by Oswald Winkel in 1914. Its limiting form is the Eckert V when a standard parallel of  $0^{\circ}$  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

---



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, 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. In Mapping Toolbox, data grids can be implicit (regular) or explicit (irregular), 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 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.

**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  $ddd^{\circ} mm'$ . 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. Mapping Toolbox can represent DM angles as column vectors, [degrees minutes].

**DMS**

Degrees-minutes-seconds angle notation of the form  $ddd^{\circ} mm' ss''$ . 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. Mapping Toolbox can represent DMS angles as column vectors, [degrees minutes seconds].

**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

$$\text{ellipsvec} = [\text{semimajor-axis eccentricity}]$$

**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**

In Mapping Toolbox, a MATLAB structure array with one element per vector geographic feature. It includes a Geometry or type field, at least two coordinate array fields (X and Y, Lat and Lon, or lat and long), 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**

See **Geographic data structure** on page Glossary-8.

**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. In Mapping Toolbox, Tissot indicatrices can be displayed using the `tissot` command, and indicatrices for all supported projections are provided. See Chapter 13, “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. Mapping Toolbox does not include interrupted projections, 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 ( $\text{topo} > 0$ ). 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 display functions in Mapping Toolbox 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**

In Mapping Toolbox, a projected rectangle or quadrangle enclosing a geographic data displayed on 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.

**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 projections supported by Mapping Toolbox, 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.

**Pseudocylindrical 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. In Mapping Toolbox, great circles, including the Equator and all meridians, are treated as special, limiting cases of small circles. Mapping Toolbox generalizes 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 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, such geodata is often represented by two vectors, one with latitudes, another with longitudes. Segments can be demarcated by inserting NaNs in both vectors. Often the pair of coordinate vectors constitute field values in a geographic data structure array.

**Viewshed**

The portion of a surface that is visible from a given point on or above it; derived from the concept of a watershed.

**Waypoints**

Points through which a trip, track, or transit passes, usually corresponding to course or speed changes.

**WGS 72 (World Geodetic System 1972)**

An Earth-centered datum, used as a definition of DMA (now NGA) DEMs. The WGS 72 datum was the result of an extensive effort extending over approximately three years to collect selected satellite,

surface gravity, and astrogeodetic data available throughout 1972. This data was combined using a unified WGS solution (a large-scale least squares adjustment).

**WGS 84 (World Geodetic System 1984)**

The WGS 84 was developed as a replacement for the WGS 72 by the military mapping community as a result of new and more accurate instrumentation and a more comprehensive control network of ground stations. The newly developed satellite radar altimeter was used to deduce geoid heights from oceanic regions between 70° north and south latitude. Geoid heights were also deduced from ground-based Doppler and ground-based laser satellite-tracking data, as well as surface gravity data. The ellipsoid associated with WGS 84 is GRS 80.

**World file**

A small text file used to georeference different raster image formats, developed to incorporate imagery into ESRI's ArcView software.

**Zenithal projection**

*See* Azimuthal projection.





# Bibliography

---

- 1** Snyder, J.P., *Map Projections — A Working Manual*, U.S. Geological Survey Professional Paper 1395, Washington, D.C., 1987.
- 2** Maling, D.H., *Coordinate Systems and Map Projections*, 2nd Edition, Pergamon Press, New York, NY, 1992.
- 3** Snyder, J.P., and Voxland, P.M., *An Album of Map Projections*, U.S. Geological Survey Professional Paper 1453, Washington, D.C., 1994.
- 4** Snyder, J.P., *Flattening the Earth — 2000 Years of Map Projections*, University of Chicago Press, Chicago, IL, 1993.
- 5** U.S. National Geospatial Intelligence Agency, “Military Specification: Digital Chart of the World (DCW)”, MIL-D-89009, 13 April 1992.



# Examples

---

Use this list to find examples in the documentation.

## **Your First Maps**

“See the World” on page 1-4

“Tour Boston with the Map Viewer” on page 1-9

## **Vector Geodata**

“A Look at Vector Data” on page 2-4

## **Raster Geodata**

“A Look at Raster Data” on page 2-8

## **Combining Vector and Raster Geodata**

“Viewing Raster and Vector Data on the Same Map” on page 2-11

## **Understanding Vector Data**

“Selecting Data to Read with the shaperead Function” on page 2-21

## **Understanding Raster Data**

“Constructing a Global Data Grid” on page 2-29

“Computing Map Limits from Reference Vectors” on page 2-30

“Geographic Interpretation of Matrix Elements” on page 2-32

“The Geography of Gridded Geodata” on page 2-33

“Accessing Data Grid Elements” on page 2-34

“Using a Mask to Recode a Data Grid” on page 2-36

“Precomputing the Size of a Data Grid” on page 2-37

## Geolocated Data Grids

“Geolocated Grid Format” on page 2-38

“Transforming Geolocated to Regular Grids” on page 2-45

## Exporting Vector Geodata

“Generating A Single Placemark” on page 2-52

“Placemarks from Addresses” on page 2-54

“Exporting Road Names to Placemarks” on page 2-56

“Setting Up Tables for the Road Placemarks” on page 2-57

## Creating and Viewing Maps

“Using worldmap” on page 4-5

“Using usamap” on page 4-7

“Accessing and Manipulating Map Axes Properties” on page 4-13

“Switching Between Projections” on page 4-18

“Changing Map Projections when Using geoshow” on page 4-25

“Placing Geographic and Nongeographic Objects in a Map Axes” on page 4-28

“Displaying Vector Maps as Lines” on page 4-43

“Displaying Vector Maps as Lines or Patches” on page 4-46

“Fitting Gridded Data to the Graticule” on page 4-54

“Using Raster Data to Create 3-D Displays” on page 4-57

“Defining Small Circles and Tracks Interactively” on page 4-63

“Determining and Manipulating Object Names” on page 4-67

## Making Three-Dimensional Maps

“Using dteds, usgsdems, and gtopo30s to Identify DEM Files” on page 5-5

“Mapping a Single DTED File with the DTED Function” on page 5-6

“Mapping Multiple DTED Files with the DTED Function” on page 5-9

“Extracting DEM Data with demdataui” on page 5-13

“Computing Line of Sight with los2” on page 5-19

“Lighting a Terrain Map Constructed from a DTED File” on page 5-22

“Lighting a Global Terrain Map with lightm and lightmui” on page 5-25

“Creating Monochrome Shaded Relief Maps Using surfm” on page 5-29

“Coloring Shaded Relief Maps and Viewing Them in 3-D” on page 5-34

“Colored 3-D Relief Maps Illuminated with Light Objects” on page 5-36

“Draping via Converting a Regular Grid to a Geolocated Data Grid” on page 5-43

“Draping a Geolocated Grid on Regular Data Grid via Texture Mapping” on page 5-46

## **Making Three-dimensional Maps**

“Draping Geoid Heights over Topography” on page 5-40

“The Globe Display Compared with the Orthographic Projection” on page 5-50

“Using Opacity and Transparency in Globe Displays” on page 5-52

“Over-the-Horizon 3-D Views Using Camera Positioning Functions” on page 5-55

“Displaying a Rotating Globe” on page 5-57

## **Customizing and Printing Maps**

“Inset Maps” on page 6-2

“Graphic Scales” on page 6-8

“North Arrows” on page 6-14

“Choropleth Maps” on page 6-18

“Scatter Maps” on page 6-26

## **Using Cartesian MATLAB Display Functions**

“Example 1: Triangulating Data Points” on page 6-28

“Example 2: Constructing Quiver Maps” on page 6-30

## Using Colormaps and Colorbars

“Colormap for Terrain Data” on page 6-34

“Contour Colormaps” on page 6-37

“Colormaps for Political Maps” on page 6-39

## Vector Data Manipulation

“Extracting and Joining Polygons or Line Segments” on page 7-2

“Linking Line Segments into Polygons” on page 7-4

“Interpolating Vectors to Achieve a Minimum Point Density” on page 7-6

“Interpolating Coordinates at Specific Locations” on page 7-7

“Overlaying Polygons with the polybool Function” on page 7-13

“Removing Discontinuities from a Small Circle” on page 7-17

“Generating a Buffer Around a Compound Polygon” on page 7-20

“Trimming Vectors to Form Lines and Polygons” on page 7-22

“Using `reducem` to Simplify Lines” on page 7-26

## Raster Data Manipulation

“Creating Data Grids from Vector Data” on page 7-31

“Using a GUI to Rasterize Polygons” on page 7-36

“Generating “Blank” Logical Grids” on page 7-39

“Obtaining the Area Occupied by a Logical Grid Variable” on page 7-41

“Using the `mapprofile` Function” on page 7-42

“Computing Gradient Data from a Regular Data Grid” on page 7-46

## **Projections and Transformations**

- “Pseudocylindrical Map Projections” on page 8-7
- “Exploring Projection Aspect” on page 8-13
- “Determining Projection Parameters” on page 8-20
- “Visualizing Projection Distortions via Tissot Indicatrices” on page 8-28
- “Visualizing Projection Distortions via Isolines” on page 8-30
- “Using distortcalc to Determine Map Projection Geometric Distortions” on page 8-32
- “Retrieving Projected Coordinates from a Figure” on page 8-38
- “Using mfwdtran with a Geographic Data Structure” on page 8-40
- “Recovering Geodetic Coordinates with minvtran” on page 8-43
- “Obtaining Angular Directions in a Projection Space” on page 8-44
- “Reorienting Vector Data with rotatem” on page 8-47



**A**

accuracy of map computations 11-183

Adams, O. S.

    Craster projection 13-31

    Quartic Authalic projection 13-113

Airy Minimum Error Azimuthal projection 13-20

Airy, George

    Airy Minimum Error Azimuthal  
    projection 13-20

aitoff 13-2

Aitoff projection 13-2

    and Equidistant Azimuthal projection 13-2

    and Hammer projection 13-2

Aitoff, David

    Aitoff projection 13-2

Albers Equal-Area Conic projection 13-4

    and Behrmann Cylindrical projection 13-4  
    13-6

    and Lambert projections 13-4 13-6

Albers Equal-Area Conic standard  
projection 13-6

Albers, Heinrich Christian

    Albers Equal-Area Conic projection 13-5

almanac 11-2

    examples of 3-46

American Geographical Society 13-95

American Polyconic projection 13-107

American Polyconic standard projection 13-109

angl2str 11-6

angle conversion

    degrees to dm or dms 11-127 to 11-128

    degrees to rads 11-126

    dm or dms to degrees 11-149 11-151

    radians to degrees 11-507

    various units 11-8

angle conversions

    summary of 3-26

angle units

    convention for navigation functions 9-12

    in geospatial data 3-18

angledim 11-8

angles

    converting degrees to radians 11-126

    converting from degrees 11-209

    converting radians to degrees 11-507

    converting to degrees 11-648

    converting to radians 11-649

    converting various units 11-8

    converting with dgrees2dm 11-127

    converting with dgrees2dms 11-128

    converting with dm2degrees 11-149

    converting with dms2degrees 11-151

    normalizing to  $-\pi$ - $\pi$  11-448

    normalizing to  $0$ - $2\pi$  11-731

    radians conversion 11-210

    unwrapping 11-669

annotation

    north arrows 11-443

antipodal points

    locating on globe 11-9

antipode 11-9

Apian, Peter 13-8

apianus 13-8

Apianus II projection 13-8

arcgridread 11-12

areaint 11-13

    example of 7-11

areamat 11-16

    using 7-41

areaquad 11-19

    using 3-44

ASCII file

    converting delimiters to NaNs 11-433

ASCII geodata

    reading space-delimited 11-601

attribute spec

    definition Glossary-1

attribute specification

    for KML formatting 11-338

    . *See* makeattribspec

- auxiliary sphere
    - calculating radius 11-546
  - avhrrgoode 11-22
  - avhrrlambert 11-26
  - axes
    - map. *See* map axes
  - axes, Cartesian. *See* Cartesian axes
  - axes2ecc 11-30
    - using 3-6
  - axesm 11-31
    - map frame and 4-31
    - map grid 4-38
  - axesm GUI 11-733
  - axesmui 11-733
  - axesscale 11-49
    - using 6-2
  - azimuth 11-52
    - between track waypoints 11-323
    - calculating 11-52
    - calculating with GUI 11-799
    - defined 3-42
    - example of 3-42
    - finding cross fix position 11-99
    - in projected coordinates 6-32
  - azimuthal projection 8-9
- B**
- Babinet projection 13-97
  - Balthasart Cylindrical projection 13-10
    - and Equal-Area Cylindrical projection 13-10
  - balthsrt 13-10
  - Bartholomew, John
    - Nordic projection 13-69
  - base projection 8-17
  - bearing. *See* azimuth
  - behrmann 13-12
  - Behrmann Cylindrical projection 13-12
    - and Equal-Area Cylindrical projection 13-12
  - Behrmann, Walter
    - Behrmann Cylindrical projection 13-12
  - Bienewitz, Peter
    - Apianus projection 13-8
  - Bolshoi Sovietskii Atlas Mira projection 13-14
  - bonne 13-16
  - Bonne projection 13-16
    - and Sinusoidal projection 13-16
    - and Werner projection 13-16
  - Bonne, Rigobert
    - Bonne projection 13-16
  - Bordone Oval projection 13-86
  - braun 13-18
  - Braun
    - Braun Perspective Cylindrical projection 13-18
  - Braun Perspective Cylindrical projection 13-18
    - and BSAM projection 13-18
    - and Gall Stereographic projection 13-18
  - Breusing Harmonic Mean projection 13-20
    - and Stereographic projection 13-20
  - Breusing, F. A. Arthur
    - Breusing projection 13-20
  - bries 13-22
  - Briesemeister projection 13-22
    - and Hammer projection 13-22
  - Briesemeister, William
    - Briesemeister projection 13-22
  - bsam 13-14
  - BSAM projection 13-14
    - and Braun Perspective Cylindrical projection 13-14
  - buffer zone
    - defined 7-19
  - bufferm 11-54
    - example of 7-19
- C**
- camposm 11-56
  - camtargm 11-58

- camupm 11-60
- cart2grn 11-62
- Cartesian axes
  - displaying 11-597
- Cartesian coordinates
  - conversion to geographic 11-62
- Cartesian plots
  - Mapping Toolbox and 6-28
- cassini 13-24
- Cassini Cylindrical projection 13-24
  - and Plate Carrée projection 13-24
- Cassini Cylindrical standard projection 13-26
  - and Plate Carrée projection 13-26
- Cassini de Thury, César François
  - Plate Carrée projection 13-24
- Cassini projection 13-105
- cassinistd 13-26
- ccylin 13-28
- Central Cylindrical projection 13-28
  - and Mercator projection 13-28
  - and Wetch projection 13-28
- Central projection 13-65
- Ch'ien Lo-Chih 13-93
- changem
  - example 2-36
- choropleth maps 6-18
- circcirc 11-64
- circles. *See* great circles. *See* small circles
- clabelm 11-65
- clegendm 11-67
- clipdata 11-70
- clma 11-71
- clmo 11-72
  - GUI 11-746
- clrmenu 11-747
- cmapui 11-74
- coast MAT-file 2-5
- collig 13-30
- Collignon projection 13-30
- Collignon, \x83 douard
  - Collignon projection 13-30
- colorbar 6-37
  - labeled 6-43
- colorbars
  - nominal 6-44
- colorm 11-750
- colormaps
  - annotating 6-43
  - digital elevation maps 6-34
  - manipulation with clrmenu GUI 11-747
  - political data 6-39
  - regular data grids 11-750
  - shaded relief map 11-583
  - surface contour maps 6-37
  - terrain elevations 11-130
- colorui 11-75
- combinations
  - enumerating 11-76
- combntns 11-76
- comet3m 11-78
- cometm 11-79
  - description 6-24
- conic projections
  - developed 8-8
  - equidistant standard formulation 13-51
  - spherical equidistant 13-49
- Conical Orthomorphic projection 13-79
  - standard formulation 13-81
- contour maps
  - adding legend 11-67
  - creating 2-D 11-90
  - creating 3-D 11-80
  - labeling 11-65
- contour3m 11-80
- contourcmap 11-84
  - example 6-37
- contourfm 11-86
- contourm 11-90
- conventions
  - longitude ranges 3-12

- conversion
    - ASCII file delimiters 11-433
    - Cartesian to geographic coordinates 11-62
    - distance from degrees 11-124
    - distance to degrees 11-310
    - distance to radians 11-312
    - distance to string 11-136
    - distance units from radians 11-509
    - ellipsoid axes to eccentricity 11-30
    - ellipsoid eccentricity to flattening 11-168
    - ellipsoid eccentricity to  $n$ 
      - representation 11-169
    - ellipsoid flattening to eccentricity 11-204
    - ellipsoid  $n$  representation to eccentricity 11-431
    - equal-area to geographic coordinates 11-184
      - from degrees 11-209
      - from radians 11-210
    - geographic to equal-area coordinates 11-269
    - great circles to small circles 11-211
    - metric and other distance units 11-314
      - to degrees 11-648
      - to radians 11-649
  - convertlat 11-95
  - coordinate system
    - transformations 11-540
  - coordinate transformations 8-46
    - raster data 8-50
    - vector data 8-47
  - coordinates
    - equal-area conversion 11-184
  - Cossin, Jean
    - Sinusoidal projection 13-117
  - craster 13-31
  - Craster Parabolic projection 13-31
  - Craster, John Evelyn Edmund
    - Craster projection 13-31
  - creating ones data grids 11-450
  - cross fix positions 11-99
  - crossfix 11-99
  - current point from map axes 11-216
  - cylindrical projections
    - developed 8-6
- D**
- daspectm 11-104
  - data grids 2-7
    - coloring 6-34
    - constructing graticule mesh 11-415
    - conversion from geographic coordinates 11-582
    - conversion to geographic coordinates 11-578
    - defined 2-7
    - displaying 4-53
    - encoding geographic regions 11-181
    - gradientdata grids
      - slopedata grids:aspectdigital elevation maps:gradientdigital elevation maps:slopedigital elevation maps:aspect 7-45
    - graticules 4-54
    - logical maps 7-39
    - NaNs 11-435
    - ones 11-450
    - projecting on graticule 11-460
    - projecting on plots 11-611
    - projecting with lighting 11-614
    - resizing 11-533
    - sparse zeros 11-602
    - zeros 11-732
    - See also* geolocated data grids; regular data grids
  - data reduction
    - vector geodata 7-25
  - dateline
    - cutting map at 7-17
  - dcwdata 11-106
  - dcwgaz 11-110
  - dcwrhead 11-117

- de l'Isle, Nicolas
    - Equidistant Conic projection 13-49 13-52
  - dead reckoning 11-157
    - calculating positions 9-33
    - example 9-31
    - rules of 9-33
  - Deetz, Charles H.
    - Craster projection 13-31
  - defaultm 11-119
  - deg2km 11-124
    - example 3-41
  - deg2nm 11-124
    - example 3-25
  - deg2rad 11-126
  - deg2sm 11-124
  - degrees-minutes-seconds
    - representing 3-20
  - degrees2dm 11-127
  - degrees2dms 11-128
  - delaunay 6-29
  - demcmap 11-130
    - example 6-34
  - demdataui 11-752
    - example 5-13
  - DEMs. *See* digital elevation maps
  - departure 9-5 11-132
    - between meridians 11-132
  - Digital Chart of the World (DCW)
    - reading gazette 11-110
    - reading headers 11-117
    - reading selected data 11-106
  - digital elevation maps 6-34
    - colormap for 6-34
    - colormaps 11-130
    - description 2-7
    - line of sight in 5-19
    - reading data interactively 5-13
    - texture mapping color data onto 5-40
  - displaying
    - surfaces 11-460
  - displaym 11-134
  - dist2str 11-136
  - distance 11-138
    - converting between units 11-314
    - converting degrees to other units 11-124
    - converting radians to distance units 11-509
    - converting to degrees 11-310
    - converting to radians 11-312
    - converting to string 11-136
    - example 3-41
  - distance conversions
    - summary of 3-26
  - distance units
    - convention for navigation functions 9-12
    - converting between 3-16
  - distances on sphere
    - as angles 3-23
  - distortcalc 11-142
  - DM and DMS notations 3-20
  - dm2degrees 11-149
  - dms2degrees 11-151
  - Douglas-Peucker algorithm 7-25
  - dreckon 11-157
    - in dead reckoning 9-33
  - drift correction 9-36
  - driftcorr 11-160
    - example 9-37
  - driftvel 11-162
    - example 9-38
  - dted 11-163
  - dteds 11-166
- E**
- Earth 11-2
    - default geoid 3-9
    - ellipsoid models 3-9
    - See also* almanac
  - ecc2flat 11-168
  - ecc2n 11-169

- eccentricity 11-30
- Eckert I projection 13-33
- Eckert II projection 13-35
- Eckert III projection 13-37
- Eckert IV projection 13-39
- Eckert V projection 13-41
  - and Plate Carrée projection 13-41
  - and Sinusoidal projection 13-41
- Eckert VI projection 13-43
- Eckert, Max
  - Eckert I projection 13-33
  - Eckert II projection 13-35
  - Eckert III projection 13-37
  - Eckert IV projection 13-39
  - Eckert V projection 13-41
  - Eckert VI projection 13-43
- eckert1 13-33
- eckert2 13-35
- eckert3 13-37
- eckert4 13-39
- eckert5 13-41
- eckert6 13-43
- Edwards, Trystan
  - Trystan Edwards Cylindrical projection 13-124
- egm96geoid 11-172
- Egyptians 13-103
  - and Stereographic projection 13-119
- elevation 11-174
  - defined 3-43
  - measuring 3-42
- elevation maps. *See* digital elevation maps
- ellipse1 11-177
- ellipsoid
  - approximating planetary geoid. *See* almanac
  - as a geoid model 3-4
  - converting parameters 3-6
  - models for Earth 3-9
  - models for planets 3-46
  - radius of curvature 11-511
- ellipsoid parameters
  - converting axes to eccentricity 11-30
  - converting eccentricity to flattening 11-168
  - converting eccentricity to n representation 11-169
  - converting flattening to eccentricity 11-204
  - converting n rerepresentation to eccentricity 11-431
- ellipsoidal distances
  - along meridian 11-413
- ellipsoidal reckoning
  - along meridian 11-414
- Elliptical projection 13-97
- encodem 11-181
- epsm 11-183
- eqa2grn 11-184
  - example 9-10
- eqaazim 13-77
- eqaconic 13-6
- eqaconic projection 13-4
- eqacylin 13-45
- eqdazim 13-47
- eqdconic 13-49
- eqdconicstd 13-51
- eqdcylin 13-53
- Equal-Area Cylindrical projection 13-45
  - and Balthasart Cylindrical projection 13-45
  - and Behrmann Cylindrical projection 13-45
  - and Gall Orthographic projection 13-45
  - and Lambert Equal-Area Cylindrical projection 13-45
  - and Trystan Edwards Cylindrical projection 13-45
- Equidistant Azimuthal projection 13-47
  - and Postel projection 13-47
  - and Zenithal projection 13-47
- equidistant conic projection 13-49
- Equidistant Conic projection
  - and Equidistant Azimuthal projection 13-49
  - 13-51

- and Equidistant Cylindrical projection 13-49  
13-51
- and Plate Carrée projection 13-49 13-51
- equidistant conic standard projection 13-51
- Equidistant Cylindrical projection 13-53
  - and Die Rechteckige Plattkarte 13-53
  - and Equirectangular projection 13-53
  - and Gall Isographic projection 13-53
  - and Plate Carrée projection 13-53
  - and Projection of Marinus 13-53
  - and Rectangular projection 13-53
- Equirectangular projection 13-53
- Erastosthenes 13-105
- etopo5 11-190
- ETOPO5 model 11-190
- Etzlaub, Erhard 13-93
- Everett 13-101
- extractfield 11-192
- extractm 11-194

## F

- Fifth Fundamental Catalog of Stars 11-517
- fill3m 11-196
- fillm 11-198
  - usage 4-51
- filterm 11-200
  - example 7-24
- findm 11-201
  - example 2-35
- fipsname 11-203
- fixing. *See* navigational fixing
- Flat-Polar Quartic projection 13-89
- flat2ecc 11-204
- flatearthpoly 11-205
  - example 7-17
- flatplr 13-87
- flatplr 13-89
- flatplrs 13-91
- fournier 13-55
  - Fournier II projection 13-55
  - Fournier projection 13-55
  - Fournier, Georges
    - Fournier II projection 13-55
  - frame. *See* map frame
  - framem 11-208
    - map frame and 4-31
  - fromDegrees 11-209
  - fromRadians 11-210

## G

- Gall Isographic projection 13-57
  - and Equidistant Cylindrical projection 13-57
- Gall Orthographic projection 13-59
  - and Equal-Area Cylindrical projection 13-59
  - and Peters projection 13-59
- Gall projection 13-61
- Gall Stereographic projection 13-61
  - and Braun Perspective Cylindrical  
projection 13-61
- Gall, James
  - Gall Orthographic projection 13-59
  - Gall Stereographic projection 13-61
- gc2sc 11-211
- gcm 11-213
- gcpmap 11-216
- gcwaypts 11-218
  - example 9-27
- gcxgc 11-220
- gcxsc 11-222
  - and scxsc 7-9
- geodata. *See* geospatial data
- geographic coordinates
  - conversion from data grid 11-578
  - conversion to data grid 11-582
  - conversion to equal-area 11-269
  - selection with mouse 11-299
- geographic data structure
  - creating input to mlayers 11-539

- defined 2-16
- displaying 11-134
- extracting data 11-194
- interacting with objects 11-770
- Version 1 2-19
- Version 2 2-17
- geographic mean 9-2
- geographic points
  - standard deviation 11-605
  - standard distance 11-603
- geographic standard deviation 9-4
- geographic statistics
  - calculating geographic mean 9-2
  - calculating geographic standard deviation 9-4
  - equal-area coordinate system 9-9
  - equirectangular binning 9-7
  - histograms 9-7
- geoid
  - availability for planets 3-46
  - converting ellipsoid parameters 3-6
  - defined 3-2
  - ellipsoid approximation 3-4
  - ellipsoid models for Earth 3-9
  - importance of in mapping 5-40
- geoid vector
  - for planets. *See* almanac
  - . *See* ellipsoid vector
- geoloc2grid 11-227
- geolocated data grids
  - displaying 4-53
  - displaying image and surface coloring 4-57
  - displaying light shading 5-29
  - displaying shaded relief 5-33
  - format 2-38
  - geographic interpretation 2-41
  - projecting 11-460
  - projecting on plots 11-611
  - projecting shaded relief 11-616
  - projecting surfaces 11-618
  - projecting with lighting 11-614
  - transforming to regular 2-45
- geoshow 11-229
- geospatial data
  - combining vector and raster 2-10
  - elevation grids 2-7
  - locating on Internet 1-29
  - raster 2-7
  - types of 2-2
  - uncompressing and compressing 2-61
  - vector 2-4
- geospatial data access
  - DCW data 11-106
  - DCW gazette 11-110
  - DCW headers 11-117
  - ETOPO5 model 11-190
  - Fifth Fundamental Catalog of Stars 11-517
  - from and to Internet 2-46
  - shapefiles 11-585 11-587
  - TIGER ArcInfo files 11-633
  - TIGER FIPS name files 11-203
  - TIGER MIF files 11-629
  - TIGER/Line data 11-627
  - USGS 1-degree DEM data 11-688
  - USGS 7.5-minute DEM data 11-682
  - USGS DEM filenames 11-690
  - via* Internet 1-29
- geospatial data formats
  - reading and writing 2-46
- geostruct1 2-19
- geostruct2 2-17
- geotiff2mstruct 11-243
- geotiffinfo 11-245
- geotiffread 11-252
- getm 11-255
  - example 4-13
  - graphic scales 6-8
- getseeds 11-256
- getworldfilename 11-257
- giso 13-57



- globe 13-63
  - globe display 13-63
    - and Orthographic projection 13-63
  - Globe display
    - label rotation and 5-52
    - using 5-49
  - globedem 11-258
  - globedems 11-261
  - gnomonic 13-65
  - Gnomonic projection 13-65
  - goode 13-67
  - Goode Homolosine projection 13-67
    - and Mollweide projection 13-67
    - and Sinusoidal projection 13-67
  - Goode, J. Paul
    - Goode Homolosine projection 13-67
  - Google KML file format
    - writing to 11-315
  - gortho 13-59
  - gradientm 11-262
    - example 7-45
  - graphic scales 6-8
  - graticule
    - as grid container 2-42
    - choosing resolution 4-54
    - defined 4-54
  - graticule mesh 11-415
  - great circle track
    - calculating from one point 11-653
    - calculating from two points 11-656
    - displaying 11-805
  - great circles
    - approximating tracks with rhumb lines 9-27
    - calculating points of 3-39
    - converting to small circles 11-211
    - defined 3-32
    - interactive 4-63
    - intersection 11-220
    - intersection with small circles 11-222
  - Great Soviet World Atlas 13-14
  - Greeks 13-103
    - and Stereographic projection 13-119
  - grepfields 11-264
  - grid2image 11-268
  - gridm 11-267
  - grids. *See* geolocated data grid. *See* map grid.
    - See* regular data grid
  - grn2eqa 11-269
    - discussion 9-9
  - gshhs 11-271
  - gstereo 13-61
  - gtextm 11-277
  - gtopo30 11-278
  - gtopo30s 11-282
  - GUIDE property editor 11-781
- ## H
- hammer 13-69
  - Hammer projection 13-69
    - and Briesemeister projection 13-69
    - and Lambert Azimuthal Equal Area projection 13-69
  - Hammer-Aitoff projection 13-69
  - handlem 11-283
    - example 4-68
  - handlem GUI 11-756
  - Hassler, Ferdinand Rudolph
    - Polyconic projection 13-107 13-109
  - hatano 13-71
  - Hatano Asymmetrical Equal-Area projection 13-71
  - Hatano, Masataka
    - Hatano Asymmetrical Equal-Area projection 13-71
  - hidem 11-286
    - example of 4-68
  - hidem GUI 11-759
  - hista 11-287
  - histograms

- equal area geographic 11-287
- equirectangular geographic 11-289
- geographic 9-7
- histr 11-289
  - example 9-7
- Homolographic projection 13-97
- Homolosine projection 13-67
- Hondius, Jodocus
  - Sinusoidal projection 13-117
- hypsometric tints 6-34

**I**

- imbedm 11-296
- ind2rgb8 11-298
- inputm 11-299
  - example 4-61
  - waypoint definition with 9-29
- inset maps
  - controlling scale 6-2
  - creating 6-2
- interplat and interp1 7-7
- interplon 7-7
- interplon and interp1 7-7
- interp1 11-300
  - interpolating vector data with 7-5
- interpolation
  - along a path 7-42
  - latitude and longitude 7-5
  - latitudes example 7-7
  - longitudes 7-7
- intersection
  - great circles 11-220
  - great circles and small circles 11-222
  - object sets 11-99
  - rhumb lines 11-537
  - small circles 11-570
- intrplat 11-301
- intrplon 11-303
- inverse projection. *See* map projections

- ismap 11-305
- ismapped 11-306
- ispolycw 11-307

**J**

- Jupiter. *See* almanac

**K**

- Kavraisky V projection 13-73
- Kavraisky VI projection 13-75
- Kavraisky, V. V.
  - Kavraisky V projection 13-73
  - Kavraisky VI projection 13-75
- kavrsky5 13-73
- kavrsky6 13-75
- km2deg 11-310
- km2nm 11-314
- km2rad 11-312
- km2sm 11-314
- KML files
  - exporting 2-51
  - specifying attributes for 11-338
- KML placemarks
  - from geographic points 2-52
  - from geostructs 2-57
- kmlwrite 11-315
- kmlwrite, using 2-51
- korea DEM 4-57

**L**

- La Carte Parall\e logrammatique 13-53
- lambcy1n 13-83
- lambert 13-79 13-81
- Lambert Azimuthal Equal-Area projection 13-77
- Lambert Conformal Conic projection 13-79 13-81
  - and Mercator projection 13-79 13-81
  - and Stereographic projection 13-79 13-81

- Lambert Conformal Conic standard projection 13-81
  - Lambert Equal-Area Azimuthal projection 13-20
  - Lambert Equal-Area Cylindrical projection 13-83
    - and Equal-Area Cylindrical projection 13-83
  - Lambert, Johann Heinrich 13-77
    - and Lambert Conformal Conic projection 13-79 13-82
    - and Lambert Equal-Area Cylindrical projection 13-83
    - Equal-Area Cylindrical projection 13-45
  - latitude
    - defined 3-11
  - latitude and longitude 3-11
    - finding corresponding time zone 11-642
    - finding for map entries 11-201
    - interpolation 7-5
    - See also* map frame, setting limits; map limits
  - latitudes and longitudes
    - string formatting 3-28
  - latlon2pix 11-321
  - lcolorbar 11-322
    - example 6-43
  - legs 11-323
    - course and distance of 9-30
    - in navigation 9-12
  - legs example 9-30
  - length units
    - choosing 3-16
  - light objects 11-325
    - lightmui 5-22
  - lightm 11-325
    - map light objects 5-36
  - limitm
    - example 2-30
  - line objects 11-329
    - displaying 4-43
    - displaying on maps in 2-D 11-469
    - displaying on maps in 3-D 11-467
  - line simplification 7-25
  - linecirc 11-328
  - linem 11-329
  - logical maps
    - defined 7-39
  - longitude
    - defined 3-12
    - ranges 3-12
  - longitude wrapping
    - to [-180 180] 11-725
    - to [-pi pi] 11-728
    - to [0 360] 11-726
    - to [0 pi] 11-727
  - longitudes
    - unwrapping with NaNs 11-669
  - Lorgna projection 13-77
  - los2 11-331
    - example 5-19
  - loximuth 13-85
  - Loximuthal projection 13-85
  - loxodromes. *See* rhumb lines
  - ltln2val 11-335
    - example 2-35
- ## M
- majaxis 11-337
  - makattribspec 11-338
  - makemapped 11-347
    - and mapped objects 6-30
  - makerefmat 11-349
  - makesymbolspec 11-355
    - setting patch colors 6-8
  - map
    - definition 2-2
    - deleting 11-71
    - precision 11-183
  - map axes
    - accessing default property values 4-16
    - accessing properties 4-13

- Cartesian data and 4-28
- changing projection of 4-29
- defining map projection with GUI 11-733
- defining map projections 11-31
- example of properties 4-14
- inset maps 6-2
- modifying properties 11-579
- resetting to default properties 4-22
- retrieving map structure 11-213
- retrieving properties 11-255
- setting properties 4-13
- setting properties with axesm 11-31
- setting properties with GUI 11-733
- testing 11-305
- use of userdata 4-3
- map data
  - querying with GUI 11-786
  - . *See* raster geodata. *See* vector geodata
- map display
  - 3-D globes 13-63
  - light objects 11-325
  - lighted surfaces 11-614
  - patches with fill13m 11-196
  - patches with fill1m 11-198
  - patches with patchesm 11-456
  - patches with patchm 11-458
  - surfaces with meshm 11-419
  - surfaces with surfacem 11-611
  - surfaces with surfm 11-618
  - text 11-277
  - text objects 11-624
- map frame
  - adjusting for a new projection 4-18
  - controlling appearance 4-36
  - defined 4-31
  - displaying 11-208
  - full-world 4-31
  - modifying properties 11-579
  - resetting altitude 4-37
  - setting limits 4-31
  - setting properties 11-31 11-208
  - setting properties with GUI 11-733
  - trimming objects to 6-30
- map grid
  - controlling appearance 4-38
  - defined 4-38
  - displaying 4-38 11-267
  - modifying properties 11-579
  - resetting altitude 4-39
  - setting properties 11-31
  - setting properties with gridm 11-267
  - setting properties with GUI 11-733
- map grid labels
  - alternate 11-429
  - displaying meridians 11-428
  - displaying parallels 11-466
  - modifying properties 11-579
  - setting properties with axesm 11-31
- map layers 11-770
- map legend
  - deprecated term 2-28
  - . *See* referencing vector
- map limits
  - adjusting for a new projection 4-18
  - setting 4-36
- map objects
  - mobjects GUI 4-66
- map origin 8-11
  - computing from new pole 11-442
  - computing new 11-501
  - See also* orientation vectors
- map projection
  - defining with GUI 11-733
  - identification strings 11-372
  - inverse 11-425
  - names 11-372
- map projections
  - 2-D vs. 3-D 5-49
  - area 8-5
  - azimuthal 8-9

- base 8-17
- changing 11-579
- changing with geoshow 4-25
- choosing 8-64
- classifying distortion 8-4
- computations 8-38
- conformality 8-4
- conic 8-8
- cylindrical 8-6
- defined 8-3
- defining 11-31
- developable surface 8-4
- distance 8-4
- equidistance 8-4
- equivalence 8-5
- forward 11-421
- general properties 3-29
- planar 11-421
- polyconic 8-8
- projecting objects 11-491
- pseudocylindrical projection
  - examples 8-7
- shape 8-4
- switching with setm 4-18
- table of properties 8-64
- vectors 8-44
- visualizing distortions 8-28
- map scale
  - between axes 6-2
  - when printing 6-45
- map text
  - placement via mouse 11-277
  - projecting 11-624
- map viewer
  - using 1-9
- map2pix 11-360
- mapbbox 11-361
- maplist 11-362
- mapoutline 11-364
- mapped objects
  - converting from standard objects 6-30
  - manipulating by name 4-66
  - reprojecting 4-23
  - trimming to map frame 6-30
- Mapping Toolbox
  - help for 1-26
- mapprofile 11-367
  - example 7-42
- maps 11-372
  - printing 6-45
- mapshow 11-374
- maptool 11-761
- maptrim GUI 11-767
- maptriml 11-390
  - discussion 7-22
- maptrimp 11-391
  - discussion 7-22
- maptrims 11-394
- mapview 11-396
  - example 1-9
- Marinus of Tyre 13-105
  - Equidistant Cylindrical projection 13-53
- Mars. *See* almanac
- MATLAB graphics
  - on projected maps 6-28
- matrix geodata. *See* raster geodata
- matrix maps. *See* raster geodata
- McBryde, F. Webster
  - and McBryde-Thomas Flat-Polar Parabolic projection 13-87
  - and McBryde-Thomas Flat-Polar Quartic projection 13-89
  - and McBryde-Thomas Flat-Polar Sinusoidal projection 13-91
- McBryde-Thomas Flat-Polar Parabolic projection 13-87
- McBryde-Thomas Flat-Polar Quartic projection 13-89
- McBryde-Thomas Flat-Polar Sinusoidal projection 13-91

- mdistort 11-406
  - mean geographic location 11-411
    - example 9-2
  - meanm 11-411
    - example 9-4
  - mercator 13-93
  - Mercator Equal-Area projection 13-117
  - Mercator projection 13-93
    - bearings on 9-13
    - in navigational tracking 9-29
    - transverse aspect 8-17
  - Mercator, Gerardus 13-93
    - Equidistant Conic projection 13-49 13-52
  - Mercury. *See* almanac
  - meridian labels 11-428
    - alternate 11-429
  - meridianarc 11-413
  - meridianfwd 11-414
  - MeridianLabel
    - use of 4-41
  - meridians
    - controlling display 4-38
    - defined 3-12
    - distance along 11-413
    - reckoning position along 11-414
  - mesh. *See* graticule mesh
  - meshgrat 11-415
    - 3-D example 4-57
    - example 2-44
    - use of 4-56
  - meshlsrm 11-417
    - coloring and shading terrain maps 5-33
  - meshm 11-419
  - mfwdtran 11-421
  - miller 13-95
  - Miller Cylindrical projection 13-95
    - and Mercator projection 13-95
  - Miller, Osborn Maitland 13-95
  - minaxis 11-424
    - example 3-6
  - minvtran 11-425
  - mlabel 11-428
  - mlabelzero22pi 11-429
  - mlayers 11-770
  - MLineException
    - usage 4-40
  - MLineLimit
    - usage 4-40
  - mobjects 11-773
  - modsine 13-121
  - mollweid 13-97
  - Mollweide projection 13-97
    - and Goode Homolosine projection 13-97
    - and Sinusoidal projection 13-97
  - Mollweide, Carl B. 13-97
  - Moon. *See* almanac
  - mouse interactions
    - defining small circles 11-568
    - processing button-down callbacks 11-809
    - selection of geographic coordinates 11-299
    - text on maps 11-277
    - with displayed maps 4-61
  - Murdoch I Conic projection 13-99
  - Murdoch III Minimum Error Conic projection 13-101
  - Murdoch, Patrick
    - and Murdoch I Conic projection 13-99
    - and Murdoch III Minimum Error projection 13-101
  - murdoch1 13-99
  - murdoch3 13-101
- ## N
- n2ecc 11-431
  - namem 11-432
    - example 4-67
  - nanclip 11-433
  - nanm 11-435
    - data grid construction 7-40

- NaNs
    - in data grids 11-435
  - National Geographic Society
    - and Robinson projection 13-115
  - navfix 11-436
    - example 9-18
  - navigation
    - calculating dead reckoning positions 9-33
    - calculating waypoints 9-27
    - connecting waypoints 9-29
    - course and distance legs 9-30
    - distance conventions 9-12
    - fixing position 9-13
    - functions for 9-11
    - retrieving time zone for longitude 9-38
    - units and conventions 9-12
  - navigational conventions
    - distance, speed, and angles 9-12
  - navigational fixing
    - example 9-18
    - navfix 11-436
    - position 9-13
  - navigational tracks
    - calculating segments between waypoints 11-650
    - connecting waypoints 9-29
    - displaying 9-29
    - format 9-12
  - Neptune. *See* almanac
  - neworig 11-439
    - example 8-50
  - newpole 11-442
    - example 8-48
  - nm2deg 11-310
  - nm2km 11-314
  - nm2rad 11-312
  - nm2sm 11-314
  - normal aspect 8-11
  - north arrows 6-14
  - northarrow 11-443
  - np2pi 11-448
- O**
- objects
    - assigning tags 11-621
    - assigning tags with GUI 11-803
    - deleting 11-72
    - deleting with GUI 11-746
    - displaying 11-598
    - displaying with GUI 11-798
    - editing properties of 11-781
    - hiding 11-286
    - hiding with GUI 11-759
    - interacting with GUI 11-773
    - modifying zdata 11-729
    - modifying zdata with GUI 11-815
    - projecting to map axes 11-491
    - repackaging vector 7-2
    - retrieving handle 11-283
    - retrieving handle with GUI 11-756
    - retrieving name 11-432
    - testing if mapped 11-306
  - oblique aspect 8-12
  - onem 11-450
    - example 7-39
  - Ordinary Polyconic projection 13-107
  - org2pol 11-451
  - orientation
    - projection 8-11
  - orientation vectors 8-11
  - origin
    - interactive modification 11-776
    - transformation 11-439
  - origin property. *See* projection aspect
  - origin vectors. *See* orientation vectors
  - originui 11-776
  - ortho 13-103
  - Orthographic projection 13-103
  - Orthophanic projection 13-115

**P**

- panzoom 6-45
- panzoom GUI 11-778
- paperscale 11-453
  - example 6-45
- parallel labels 11-466
- Parallellabel
  - use of 4-41
- parallels
  - controlling display 4-38
  - defined 3-11
- parallelui 11-780
- patch 11-458
- patch drawing functions
  - differences between 4-51
- patch maps
  - functions for 4-51
- patch objects
  - displaying 4-46
  - filling 11-196
  - filling 2-D 11-198
  - filling 2-D and 3-D 11-458
  - filling separate 11-456
- patchesm 11-456
  - usage 4-51
- patchm
  - usage 4-51
- pcarree 13-105
- pcolorm 11-460
- Peters projection 13-59
- piloting. *See* navigational fixing
- pix2latlon 11-462
- pix2map 11-463
- pixcenters 11-464
- plabel 11-466
- placemarks
  - from addresses 2-54
  - HTML tables for 2-57
- planetary data 11-2
- Plate Carrée projection 13-105
- plot3m 11-467
- plotm 11-469
  - example 4-44
- Pluto. *See* almanac
- polcmap 11-471
  - example 6-39
- pole transformations 11-451
- poltical maps
  - coloring 6-39
- poly2ccw 11-473
- poly2cw 11-474
- poly2fv 11-475
- polybool 11-477
  - cutting across dateline 7-17
  - example 7-12
- polycon 13-107
- polyconic projection
  - developed 8-8
- Polyconic projection 13-107
- Polyconic standard projection 13-109
- polyconstd 13-109
- polycut 11-482
- polygon
  - buffer zones 7-19
  - displaying as line object 4-43
  - eliminating date line crossing 7-17
  - extracting segments 7-2
  - intersection points 7-10
  - set operations 7-12
  - surface area 7-11
- polygon maps
  - functions for 4-51
- polygon surface area 11-13
- polygons
  - displaying as patch objects 4-46
  - extracting segments 7-2
  - set operatons using polybool 7-12
- polyjoin 11-483
  - example 7-3
- polymerge 11-484



- example 7-4
- polysplit 11-486
  - example 7-2
- polyxpoly 11-487
  - and date line 7-17
  - example 7-10
- positions
  - dead reckoning 11-157
  - reckoning 11-524
- Postel, Guillaume
  - Equidistant Azimuthal projection 13-47
- previewmap 11-489
- printing maps 6-45
- project 11-491
  - example 6-29
- projection. *See* map projections
- projection aspect
  - normal 8-11
  - oblique 8-12
  - skew-oblique 8-16
  - transverse 8-12
- Projection of Marinus 13-53
- projections. *See* map projections
- projfwd 11-494
- projinv 11-496
- projlist 11-499
- property editors 11-781
- Ptolemy, Claudius
  - Bonne projection 13-16
  - Equidistant Conic projection 13-49 13-52
- Putnins
  - P4 and Craster projections 13-31
- Putnins P4 projection 13-31
- Putnins P5 projection 13-111
- Putnins, Reinholds V. 13-111
- putnins5 13-111
- putpole 11-501

**Q**

- qrydata 11-786
- quadrangle surface area 11-19
- quartic 13-113
- Quartic Authalic projection 13-113
- querying map data 11-786
- quiver 6-31
- quiver3m 11-503
- quiverm 11-505
  - description 6-24

**R**

- rad2deg 11-507
- rad2km 11-509
- rad2nm 11-509
- rad2sm 11-509
- radius of auxiliary sphere 11-546
- radius of curvature 11-511
- radius of planets 3-46 11-2
  - See also* almanac
- Rand McNally
  - and Robinson projection 13-115
- range
  - angles 11-731
  - finding cross fix position 11-99
- raster geodata 11-533
  - defined 2-7
  - displaying as lighted shaded relief 11-616
  - displaying as mesh 11-419
  - displaying as shaded relief 11-417
  - displaying as surface 11-618
  - georeferencing 2-27
  - representing 2-27
  - resizing 11-533
  - trimming 11-394
  - trimming with GUI 11-767
  - See also* data grids
- raster maps. *See* raster geodata
- rcurve 11-511

- readfields 11-513
  - readfk5 11-517
  - readmtx 11-520
  - reckon 11-524
    - example 3-39
  - reckoning 11-524
    - distances with GUI 11-799
    - position ahead 3-38
  - Rectangular projection 13-53
  - reducem 11-526
  - referencing matrix
    - defined 2-27
    - for image 1-24
  - referencing vector
    - defined 2-27
    - refmat variable 2-27
  - refmat2vec 11-529
  - refvec2mat 11-530
  - regular data grids 2-29
    - accessing elements 2-34
    - calculating required matrix size 11-599
    - creating colormap 11-750
    - defined 2-29
    - determining limits 2-30
    - determining size with scaling 2-37
    - displaying 4-53
    - displaying image and surface coloring 4-57
    - displaying shaded relief 5-33
    - encoding 11-296
    - encoding regions 11-795
    - geographic interpretation 2-32
    - global 2-29
    - precomputing size 2-37
    - projecting shaded relief 11-417
    - projecting with meshm 11-419
    - recoding 2-36
    - retrieving values 11-335
    - seeds for encoding 11-256
    - surface area 11-16
    - transforming to new coordinate system map
      - origin 11-439
      - trimming 11-394
    - See also* geolocated data grids
  - reprojecting maps
    - limitations on 4-23
  - resizem 11-533
  - restack 11-536
  - rhumb line track
    - calculating from one point 11-653
    - calculating from two points 11-656
    - displaying 11-805
  - rhumb lines
    - approximating great circle tracks with 9-27
    - calculating points 3-39
    - defined 3-32
  - rhumb lines intersection 11-537
  - rhxrh 11-537
    - and scxsc 7-9
  - robinson 13-115
  - Robinson projection 13-115
  - Robinson, Arthur H.
    - Robinson projection 13-115
  - rootlayr 11-539
  - rotatem 11-540
    - example 8-47
  - rotatetext 11-542
  - rounding 11-544
  - roundn 11-544
  - rsphere 11-546
  - Ruysch, Johannes
    - Equidistant Conic projection 13-49 13-52
- S**
- Sanson-Flamsteed projection 13-117
  - satbath 11-548
  - Saturn. *See* almanac
  - scale
    - between axes 6-2

- printing maps to 6-45
- scaleruler 11-551
  - example 6-8
- scatterm 11-560
  - description 6-24
  - proportional symbol maps 9-8
- scircle1 11-562
  - example 3-34
- scircle2 11-565
  - example 3-34
- scircleg 11-568
  - example 4-63
- scirclui 11-791
- scxsc 11-570
  - and gscxsc 7-9
- sdtsemread 11-572
- sdtinfo 11-573
- sectorg 11-577
- seedm 11-795
- selectors
  - with shapefile data 2-21
- semimajor axis 11-337
- semiminor axis 11-424
- setltn 11-578
  - example 2-32
- setm 11-579
  - example 4-13
  - graphic scales 6-8
  - map frame 4-31
  - map grid 4-38
- setpostn 11-582
  - example 2-33
- shaded relief map
  - constructing cdata 11-583
  - constructing colormap 11-583
  - geolocated data grids 11-616
- shaded relief maps 5-33
  - regular data grids 11-417
- shaderel 11-583
- shapefiles
  - information from 11-585
  - reading with shaperead 11-587
  - writing with shapewrite 11-594
- shapeinfo 11-585
- shaperead 11-587
  - data selectors 2-21
- shapewrite 11-594
- showaxes 11-597
- showm 11-598
  - example 4-68
- showm GUI 11-798
- Siemon, Karl 13-86
  - Quartic Authalic projection 13-113
- simple conic projection 13-49
- Simple Cylindrical projection 13-105
- simplification of map data 7-25
- sinusoid 13-117
- Sinusoidal projection 13-117
- size 11-599
  - example 2-37
- skew-oblique aspect 8-16
- sllipsoidal conic projection 13-51
- sm2deg 11-310
- sm2km 11-314
- sm2nm 11-314
- sm2rad 11-312
- small circles
  - calculating from center and perimeter
    - point 11-565
  - calculating from center and radius 11-562
  - defined 3-33
  - defining with mouse 11-568
  - displaying 11-791
  - interactive 4-63
  - intersection 11-570
  - intersection with great circles 11-222
- spatial errors
  - in maps 8-28
- spread 11-601
- specifying attributes

- for KML output 11-338
  - speed units
    - format for navigation functions 9-12
  - spzrom 11-602
    - and zrom 7-40
  - Stab-Werner projection 13-134
  - Stabius, Johannes
    - Werner projection 13-134
  - standard deviation of geographic data 9-4
  - standard deviation of geographic points 11-605
  - standard distance of geographic points 11-603
  - stdist 11-603
    - defined 9-6
  - stdm 11-605
    - defined 9-4
  - stem plot
    - example 6-24
  - stem3m 11-607
    - description 6-24
  - stereo 13-119
  - Stereographic projection 13-119
  - str2angle 11-609
  - Sun. *See* almanac
  - surface area
    - accessing from almanac 3-47
    - measuring polygons 7-11
    - planets. *See* almanac
    - polygon 11-13
    - quadrangle 11-19
    - regular data grids 11-16
  - surface aspect
    - defined 7-45
  - surface distance
    - along a parallel 11-132
    - between track waypoints 11-323
    - between two points 11-138
    - calculating with GUI 11-799
  - surface gradient
    - defined 7-45
  - surface objects
    - constructing graticule mesh 11-415
    - displaying 4-53
    - projecting lighted 11-614
    - projecting on graticule 11-460
    - projecting with meshm 11-419
    - projecting with surfacem 11-611
    - projecting with surfm 11-618
  - surface slope
    - defined 7-45
  - surfacem 11-611
  - surfdist 11-799
  - surflm 11-614
    - lighting terrain maps 5-29
  - surflsrm 11-616
    - coloring and shading terrain maps 5-33
  - surfm 11-618
  - Sylvanus, Bernardus
    - Bonne projection 13-16
  - symbol plot
    - example 6-26
  - symbol specification. *See* symbolspecs
  - symbolspecs
    - definition Glossary-21
    - example for roads 1-18
    - setting patch colors 6-8
    - with geoshow 4-10
    - with polcmap 4-10
- ## T
- tagm 11-621
  - tagm GUI 11-803
  - tbase 11-622
  - textm 11-624
  - texture mapping
    - onto digital elevation maps 5-40
  - tgrline 11-627
  - Thales
    - Gnomonic projection 13-65
  - thematic maps

- 3-D bar graphs 6-24
  - comet maps 6-24
  - quiver maps 6-24
  - scatter maps 6-24
  - tissot maps 6-24
  - Thomas, Paul D.
    - and McBryde-Thomas Flat-Polar Parabolic projection 13-87
    - and McBryde-Thomas Flat-Polar Quartic projection 13-89
    - and McBryde-Thomas Flat-Polar Sinusoidal projection 13-91
  - TIGER data
    - ArcInfo files 11-633
    - MIF files 11-629
    - reading FIPS name files 11-203
    - TIGER/Line data 11-627
  - tigermif 11-629
  - tigerp 11-633
  - tightmap 11-636
    - printing maps 6-45
  - time zones
    - determining from longitude 11-642
    - for navigation 9-38
    - navigational 9-36
  - timezone 11-642
    - example 9-39
  - tissot 11-644
    - description 6-24
    - example 8-28
  - tissot indicatrices
    - projecting 11-644
  - Tissot Modified Sinusoidal projection 13-121
  - Tissot, N. A.
    - Tissot Modified Sinusoidal projection 13-121
  - Tobler, Waldo R. 13-86
  - toDegrees 11-648
  - topo DEM 2-7
  - topographical maps. *See* digital elevation maps
  - toRadians 11-649
  - track 11-650
    - description 9-29
  - track waypoints
    - azimuth 11-323
    - distance 11-323
  - track1 11-653
    - example 3-39
  - track2 11-656
    - example 4-45
    - vs. track1 3-39
  - trackg 11-658
    - example 4-63
  - tracks. *See* great circles. *See* rhumb lines
  - trackui 11-805
  - tranmerc 13-122
  - transformation of coordinate system 8-46 11-540
    - See also* coordinate transformation
  - transverse aspect 8-12
  - transverse Mercator projection
    - example 8-61
  - Transverse Mercator projection 13-122
    - and UTM 13-122
  - trimcart 11-660
    - and mapped objects 6-30
  - trimdata 11-662
  - trimming data 7-21
  - trimming map data
    - attribute filtering 7-24
  - trisurf 6-29
  - trystan 13-124
  - Trystan Edwards Cylindrical projection 13-124
    - and Equal-Area Cylindrical projection 13-124
  - Tunhuang star chart 13-93
  - two-column ASCII geodata
    - reading 11-601
- U**
- uimaptbx 11-809

- undoclip 11-663
  - undotrim 11-664
  - units
    - testing for valid abbreviations 11-667
    - testing for valid strings 11-667
  - unitsratio 11-665
    - examples 3-17
  - unitstr 11-667
  - Universal Polar Stereographic projection 13-126
    - and UTM 13-126
    - limits 8-52
  - Universal Transverse Mercator system 13-127
    - and Gauss-Kr\ger 13-127
    - and Transverse Mercator projection 13-127
    - military mapping 13-127
  - unprojection
    - geographic data 11-425
  - unwrapMultipart 11-669
  - updategeostruct 11-672
  - ups 13-126
  - UPS projection 13-126
  - Uranus. *See* almanac
  - usamap 11-676
    - using 4-7
  - userdata
    - in map axes 4-3
  - USGS 1-degree DEM data
    - reading files 11-688
  - USGS DEM 7.5-minute data
    - reading files 11-682
  - USGS DEM data
    - returning filenames 11-690
  - usgs24kdem 11-682
  - usgsdem 11-688
  - usgsdems 11-690
  - utm 13-127
  - UTM
    - description 8-52
      - See also* Universal Transverse Mercator system
    - ellipsoid for 8-60
    - system 13-127
    - zone 8-60
  - utmgeoid 11-691
  - utmzone 11-692
- ## V
- Van der Grinten I projection 13-128
  - Van der Grinten, Alphons J.
    - Van der Grinten I projection 13-128
  - vec2mtx 11-694
  - vector data. *See* vector geodata
  - vector geodata
    - calculating intersections 7-8
    - converting to grid 11-767
    - defined 2-4
    - displaying as lines with `linem` 11-329
    - displaying as lines with `plot3m` 11-467
    - displaying as lines with `plotm` 11-469
    - extracting from data structures 11-194
    - filtering 11-200
    - geographic interpolation 7-5
    - mean location 11-411
    - reducing 11-526
    - representing 2-13
    - simplifying/reducing 7-25
    - structures 2-16
    - trimming data to a region 7-21
    - trimming lines 11-390
    - trimming polygons 11-391
    - trimming vector via attributes 7-24
  - vector maps
    - delineation of objects in 2-14
    - displaying as lines 4-43
    - displaying as patches 4-46
    - projected directions 8-44
  - Venus. *See* almanac
  - vertical exaggeration
    - `daspectm` 5-23

- Vertical Perspective Azimuthal projection 13-130
    - and Orthographic projection 13-130
  - vfdwtran 6-32 11-696
    - and direction vectors 8-45
  - vgrint1 13-128
  - viewshed 11-699
    - defined 5-20
    - example 5-20
  - vinvtran 11-705
  - vmap0data 11-708
  - vmap0read 11-712
  - vmap0rhead 11-715
  - vmap0ui 11-812
  - volume of planets 3-47 11-2
    - See also* almanac
  - von Hammer, H. H. Ernst
    - Hammer projection 13-69
  - vperspec 13-130
- W**
- Wagner I projection 13-75
  - Wagner IV projection 13-132
  - Wagner, Karlheinz
    - Wagner I projection 13-75
    - Wagner IV projection 13-132
  - wagner4 13-132
  - waypoints 11-650
    - calculating 9-27
    - calculating on great circle 11-218
    - connecting 9-29
    - in navigation 9-12
    - selecting with mouse 9-29
    - See also* track waypoints
  - werner 13-134
  - Werner projection 13-134
  - Werner, Johannes
    - Werner projection 13-134
  - wetch 13-136
  - Wetch Cylindrical projection 13-136
    - and Central Cylindrical projection 13-136
  - Wetch, J.
    - Wetch Cylindrical projection 13-136
  - wiechel 13-138
  - Wiechel projection 13-138
  - Wiechel, H.
    - Wiechel projection 13-138
  - winkel 13-140
  - Winkel I projection 13-140
    - and Eckert V projection 13-140
    - and Equidistant Cylindrical projection 13-140
    - and Sinusoidal projection 13-140
  - Winkel, Oswald
    - Winkel I projection 13-140
  - worldfileread 11-718
  - worldfiles
    - creating from mapview 1-23
    - reading with worldfileread 1-24
  - worldfilewrite 11-719
  - worldmap 11-720
    - introduction to 1-4
    - using 4-5
  - wrapTo180 11-725
  - wrapTo2Pi 11-727
  - wrapTo360 11-726
  - wrapToPi 11-728
  - Wright projection 13-93
  - Wright, Edward 13-93
- Y**
- Young, A. E.
    - Breusing projection 13-20
- Z**
- zdatam 11-729
    - GUI 11-815
  - Zenithal Equal-Area projection 13-77

Zenithal Equivalent projection 13-77

zero22pi 11-731

zerom 11-732

example 7-40

zeros 11-602

zooming in and out of map displays 11-778