How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Mapping Toolbox™ User's Guide

© COPYRIGHT 1997–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

May 1997	First printing	New for Version 1.0
October 1998	Second printing	Version 1.1
November 2000	Third printing	Version 1.2 (Release 12)
July 2002	Online only	Revised for Version 1.3 (Release 13)
September 2003	Online only	Revised for Version 1.3.1 (Release 13SP1)
January 2004	Online only	Revised for Version 2.0 (Release 13SP1+)
April 2004	Online only	Revised for Version 2.0.1 (Release 13SP1+)
June 2004	Fourth printing	Revised for Version 2.0.2 (Release 14)
October 2004	Online only	Revised for Version 2.0.3 (Release 14SP1)
March 2005	Fifth printing	Revised for Version 2.1 (Release 14SP2)
August 2005	Sixth printing	Minor revision for Version 2.1
September 2005	Online only	Revised for Version 2.2 (Release 14SP3)
March 2006	Online only	Revised for Version 2.3 (Release 2006a)
September 2006	Seventh printing	Revised for Version 2.4 (Release 2006b)
March 2007	Online only	Revised for Version 2.5 (Release 2007a)
September 2007	Eighth printing	Revised for Version 2.6 (Release 2007b)
March 2008	Online only	Revised for Version 2.7 (Release 2008a)
October 2008	Online only	Revised for Version 2.7.1 (Release 2008b)
March 2009	Online only	Revised for Version 2.7.2 (Release 2009a)
September 2009	Online only	Revised for Version 3.0 (Release 2009b)
March 2010	Online only	Revised for Version 3.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.2 (Release 2010b)
April 2011	Online only	Revised for Version 3.3 (Release 2011a)
September 2011	Online only	Revised for Version 3.4 (Release 2011b)
March 2012	Online only	Revised for Version 3.5 (Release 2012a)
September 2012	Online only	Revised for Version 3.6 (Release 2012b)
March 2013	Online only	Revised for Version 3.7 (Release 2013a)
September 2013	Online only	Revised for Version 4.0 (Release 2013b)
March 2014	Online only	Revised for Version 4.0.1 (Release 2014a)
October 2014	Online only	Revised for Version 4.0.2 (Release 2014b)
March 2015	Online only	Revised for Version 4.1 (Release 2015a)
September 2015	Online only	Revised for Version 4.2 (Release 2015b)
March 2016	Online only	Revised for Version 4.3 (Release 2016a)
September 2016	Online only	Revised for Version 4.4 (Release 2016b)
March 2017	Online only	Revised for Version 4.5 (Release 2017a)
September 2017	Online only	Revised for Version 4.5.1 (Release 2017b)
March 2018	Online only	Revised for Version 4.6 (Release 2018a)
September 2018	Online only	Revised for Version 4.7 (Release 2018b)
March 2019	Online only	Revised for Version 4.8 (Release 2019a)
September 2019	Online only	Revised for Version 4.9 (Release 2019b)
March 2020	Online only	Revised for Version 4.10 (Release 2020a)
September 2020	Online only	Revised for Version 5.0 (Release 2020b)

Contents

1 Getting Started

Mapping Toolbox Product Description 1-2

Acknowledgments 1-3

Create Your First World Map 1-4

Tour Boston with the Map Viewer App 1-9
Open the Map Viewer App 1-9

Getting More Help 1-23
Ways to Get Mapping Toolbox Help 1-23

2 Understanding Map Data

What Is a Map? 2-2

What Is Geospatial Data? 2-3

Vector Geodata 2-4

Inspect and Display Vector Map Data 2-5

Raster Geodata 2-7
Digital Elevation Data 2-7
Remotely Sensed Image Data 2-7

Display Shaded Relief Map Using Raster Data 2-8

Combine Vector and Raster Geodata on the Same Map 2-10
Combining Raster Data and Vector Data on the Same Map 2-10

Create and Display Polygons 2-12
Simple Polygon 2-12
Polygons with Holes or Multiple Regions 2-13
Polygons Using Geographic Coordinates 2-15
Filled Region of Polygons Using Geographic Coordinates 2-17

Segments Versus Polygons 2-20

Geographic Data Structures **2-22**
Shapefiles **2-22**
The Contents of Geographic Data Structures **2-22**
Examining a Geographic Data Structure **2-24**
How to Construct Geographic Data Structures **2-25**
Mapping Toolbox Version 1 Display Structures **2-29**

Georeferenced Raster Data **2-30**
Reference Objects **2-30**
Referencing Matrices **2-30**
Referencing Vectors **2-30**

Construct a Global Data Grid **2-32**

Precompute the Size of a Data Grid **2-34**

Geolocated Data Grids **2-35**
Define Geolocated Data Grid **2-35**

Geographic Interpretations of Geolocated Grids **2-39**
Type 1: Values Associated with the Upper Left Grid Coordinate **2-39**
Type 2: Values Centered Within Four Adjacent Coordinates **2-40**
Ordering of Cells **2-41**
Transform Regular to Geolocated Grids **2-41**
Transforming Geolocated to Regular Grids **2-41**

Unprojecting a Digital Elevation Model (DEM) **2-42**

Creating a Half-Resolution Georeferenced Image **2-53**

Georeferencing an Image to an Orthotile Base Layer **2-58**

Find Geospatial Data Online **2-70**

Find Vector Geodata **2-71**

Find Geospatial Raster Data **2-73**
Download Data **2-73**
Use Web Map Service Data **2-74**

Functions that Read and Write Geospatial Data **2-75**

Export Vector Geodata **2-78**

Exporting Vector Data to KML **2-79**

Export KML Files for Viewing in Earth Browsers **2-90**
Generate a Single Placemark Using kmlwritepoint **2-90**
Generate Placemarks from Addresses **2-91**
Export Point Geostructs to Placemarks **2-91**

Select Shapefile Data to Read **2-94**
Example 1: Predicate Function in Separate File **2-94**
Example 2: Predicate as Function Handle **2-95**
Example 3: Predicate as Anonymous Function **2-95**

Example 4: Predicate (Anonymous Function) Defined Within Cell Array . **2-96**
Example 5: Parametrizing the Selector; Predicate as Nested Function . . **2-96**

Exporting Images and Raster Grids to GeoTIFF **2-98**

Converting Coastline Data (GSHHG) to Shapefile Format **2-113**

3 Understanding Geospatial Geometry

The Shape of the Earth **3-2**
Ellipsoid Shape **3-2**
Geoid Shape **3-2**

Reference Spheroids **3-4**
referenceSphere Objects **3-4**
referenceEllipsoid Objects **3-6**
World Geodetic System 1984 . . . . . **3-8**
Ellipsoid Vectors **3-9**
oblateSpheroid Objects **3-10**

Work with Reference Spheroids **3-11**
Map Projections **3-11**
Curves and Areas **3-12**
3-D Coordinate Transformations **3-12**

Latitude and Longitude **3-13**
Plot Latitude and Longitude **3-13**

Relationship Between Points on Sphere **3-15**

Length and Distance Units **3-16**
Choosing Units of Length **3-16**
Converting Units of Length **3-16**

Compute Conversion Ratio Between Units of Length **3-17**

Angle Representations and Angular Units **3-18**
Radians and Degrees **3-18**
Default and Variable Angle Units **3-19**
Degrees, Minutes, and Seconds **3-19**
Converting Angle Units that Vary at Run Time **3-20**

Angles as Binary and Formatted Numbers **3-22**
Formatting Latitudes and Longitudes **3-22**

Convert from Linear Measurements to Spherical Measurements **3-23**

Distances on the Sphere **3-24**
Arc Length as an Angle in the distance and reckon Functions **3-25**
Summary: Available Distance and Angle Conversion Functions **3-25**

Great Circles **3-27**

Rhumb Lines **3-28**

Azimuth **3-29**
Calculate Azimuth **3-29**

Elevation **3-31**

Generate Vector Data for Points Along Great Circle or Rhumb Line Tracks **3-32**

Reckoning **3-34**

Calculate Distance Between Two Points in Geographic Space **3-35**

Small Circles **3-36**

Calculate Vector Data for Points Along a Small Circle **3-37**

Generate Small Circles **3-38**

Measure Area of Spherical Quadrangles **3-40**

Plotting a 3-D Dome as a Mesh Over a Globe **3-41**

Choose a 3-D Coordinate System **3-47**
Earth-Centered Earth-Fixed Coordinates **3-47**
Geodetic Coordinates **3-48**
East-North-Up Coordinates **3-49**
North-East-Down Coordinates **3-49**
Azimuth-Elevation-Range Coordinates **3-50**
Tips **3-51**

Vectors in 3-D Coordinate Systems **3-52**
Tips **3-53**

Find Ellipsoidal Height from Orthometric Height **3-55**
Find Ellipsoidal Height from Orthometric and Geoid Height **3-57**

4 Creating and Viewing Maps

Introduction to Mapping Graphics **4-2**

Continent, Country, Region, and State Maps Made Easy **4-3**

Set Background Colors for Map Displays **4-4**

Create Simple Maps Using worldmap **4-5**

Create Simple Maps Using usamap **4-7**

The Map Axes **4-11**
Tips to Working with Map Axes **4-11**

Access and Change Map Axes Properties **4-13**

Map Limit Properties **4-19**
Specify Map Projection Origin and Frame Limits Automatically **4-20**
Create Cylindrical Projection Using Map Limit Properties **4-23**
Create Conic Projection Using Map Limit Properties **4-25**
Create Southern Hemisphere Conic Projection **4-26**
Create North-Polar Azimuthal Projection **4-27**
Create South-Polar Azimuthal Projection **4-29**
Create Equatorial Azimuthal Projection **4-30**
Create General Azimuthal Projection **4-31**
Create Long Narrow Oblique Mercator Projection **4-32**

Switch Between Projections **4-34**
Change Projection Updating Meridian and Parallel Labels **4-34**
Change Projection Resetting Frame Limits **4-36**

Reprojection of Graphics Objects **4-40**
Auto-Reprojection of Mapped Objects and Its Limitations **4-40**
Reprojectability of Maps Generated Using geoshow **4-41**

Create Maps Using geoshow **4-43**

Creating Maps Using MAPSHOW **4-50**

Change Map Projections Using geoshow **4-68**
Change Map Projection with Vector Data Using geoshow **4-68**
Change Map Projection with Raster Data Using geoshow **4-69**

Use Geographic and Nongeographic Objects in Map Axes **4-72**

The Map Frame **4-75**

Plot Regions of Robinson Frame and Grid Using Map Limits **4-77**

Map and Frame Limits **4-82**

The Map Grid **4-83**
Control Grid Spacing **4-83**
Layer Grids **4-83**
Limit Grid Lines **4-83**
Label Grids **4-84**

Summary of Polygon Display Functions **4-86**

Display Vector Data as Points and Lines **4-87**

Display Vector Maps as Lines or Patches **4-91**

Types of Data Grids and Raster Display Functions **4-98**

Fit Gridded Data to the Graticule **4-99**
Fit Gridded Data to Fine and Coarse Graticules **4-99**

Create 3-D Displays with Raster Data **4-103**

Creating Map Displays with Latitude and Longitude Data **4-106**

Creating Map Displays with Data in Projected Coordinate Reference System **4-116**

Pick Locations Interactively **4-125**

Create an Interactive Map for Selecting Point Features **4-127**

Create Small Circle and Track Annotations on Maps Interactively **4-133**

Interactively Display Text Annotations on a Map **4-135**

Work with Objects by Name **4-136**
Manipulate Displayed Map Objects By Name **4-136**

5 Making Three-Dimensional Maps

Sources of Terrain Data **5-2**
Digital Terrain Elevation Data from NGA **5-2**
Digital Elevation Model Files from USGS **5-2**

Determine and Visualize Visibility Across Terrain **5-3**
Compute Line of Sight **5-3**

Light a Terrain Map of a Region **5-5**

Surface Relief Shading **5-8**
Create Monochrome Shaded Relief Map **5-8**

Colored Surface Shaded Relief **5-13**
Create Colored Shaded Relief Map **5-13**

Relief Mapping with Light Objects **5-17**
Illuminate Color 3-D Relief Maps with Light Objects **5-17**

Drape Data on Elevation Maps **5-24**
Combine Elevation Maps with Other Kinds of Data **5-24**
Drape Data over Terrain with Different Gridding **5-24**

Drape Geoid Heights Over Topography **5-25**

Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid **5-30**

Drape Geolocated Grid on Regular Data Grid via Texture Mapping **5-36**

The Globe Display Compared with the Orthographic Projection **5-39**

Use Opacity and Transparency in Globe Displays **5-45**

Access Basemaps and Terrain for Geographic Globe **5-48**
Use Installed Basemap **5-48**
Download Basemaps **5-48**
Add Custom Basemaps **5-48**
Access Terrain **5-48**
Specify Basemaps and Terrain **5-49**

Create Interactive Basemap Picker **5-50**

Visualize Aircraft Line-of-Sight Over Terrain **5-52**

Visualize UAV Flight Path on 2-D and 3-D Maps **5-61**

6 Customizing and Printing Maps

Inset Maps **6-2**

Graphic Scales **6-9**

North Arrows **6-15**

Thematic Maps **6-18**
Choropleth Maps **6-18**
Stem Maps **6-19**
Contour Maps **6-19**
Scatter Maps **6-20**

Create Choropleth Map of Population Density **6-21**

Contour Colormaps **6-24**

Colormaps for Political Maps **6-27**
Explore Colormaps for Political Maps **6-27**
Labeling Colorbars **6-29**
Editing Colorbars **6-30**

Scale Maps for Printing **6-31**

7 Manipulating Geospatial Data

Extract and Join Polygons or Line Segments **7-2**

Link Line Segments with Common Endpoints into Polygons **7-4**

Geographic Interpolation of Vectors **7-5**

Interpolate Vertices Between Known Data Points **7-7**

Interpolate Coordinates at Specific Locations **7-8**

Vector Intersections **7-9**

Calculate Intersections of Small Circles **7-11**

Calculate Intersection of Rhumb Line Tracks **7-12**

Calculate Intersections of Vector Data **7-13**

Calculate Area of Geographic Polygons **7-15**

Polygon Set Logic **7-16**

Overlay Polygons Using Set Logic **7-17**

Remove Longitude Coordinate Discontinuities at Date Line Crossings **7-22**

Polygon Buffer Zones **7-26**
Generate Buffer Internal to Polygon **7-26**

Trim Vectors to Preserve Polygonal Patches **7-28**

Simplify Vector Coordinate Data **7-31**

Simplify Polygon and Line Data **7-32**

Convert Vector Data to Raster Format **7-37**
Creating Data Grids from Vector Data **7-37**

Rasterize Polygons Interactively **7-42**

Data Grids as Logical Variables **7-44**

Compute Elevation Profile Along Straight Line **7-45**

Compute Gradient, Slope, and Aspect from Regular Data Grid **7-48**

Map Projections and Distortions **8-2**
Use Inverse Projection to Recover Geographic Coordinates **8-2**
Projection Distortions **8-2**

Quantitative Properties of Map Projections **8-4**

The Three Main Families of Map Projections **8-5**
Unwrapping the Sphere to a Plane **8-5**
Cylindrical Projections **8-5**
Conic Projections **8-6**
Azimuthal Projections **8-7**

Projection Aspect **8-9**
The Orientation Vector **8-9**
Control the Map Projection Aspect with an Orientation Vector **8-11**

Projection Parameters **8-16**
Projection Characteristics Maps Can Have **8-16**

Visualize Spatial Error Using Tissot Indicatrices **8-22**
Visualize Projection Distortions using Tissot Indicatrices **8-22**

Visualize Projection Distortions Using Isolines **8-26**

Quantify Map Distortions at Point Locations **8-30**
Use distortcalc to Determine Map Projection Geometric Distortions **8-30**

Rotational Transformations on the Globe **8-34**
Reorient Vector Data with rotatem **8-34**
Reorient Gridded Data **8-36**

Create a UTM Map **8-38**
Create a UTM Map **8-38**

Set UTM Parameters Interactively **8-42**

Work in UTM Without a Displayed Map **8-45**

Use the Transverse Aspect to Map Across UTM Zones **8-47**

Summary and Guide to Projections **8-49**
Cylindrical Projections **8-49**
Pseudocylindrical Projections **8-52**
Conic Projections **8-56**
Pseudoconic Projections **8-57**
Polyconic Projections **8-58**
Azimuthal Projections **8-59**
Pseudoazimuthal Projections **8-61**
Modified Azimuthal Projections **8-61**

Transform Coordinates to a Different Projected CRS **8-62**

Project and Display Raster Data **8-65**
Project Raster Data **8-65**
Unproject Raster Data **8-66**

9 Creating Web Map Service Maps

Basic WMS Terminology **9-2**

Basic Workflow for Creating WMS Maps **9-3**
Workflow Summary **9-3**
Create a Map of Elevation in Europe **9-3**

Search the WMS Database **9-5**
Introduction to the WMS Database **9-5**
Find Temperature Data in the WMS Database **9-5**

Refine Your Search **9-7**
Refine Search by Text **9-7**
Refine Search by Geographic Limits **9-7**

Update Your Layer **9-8**

Retrieve Your Map **9-10**
Map Retrieval Methods **9-10**
Understand Coordinate Reference System Codes **9-10**
Retrieve Your Map with wmsread **9-11**
Use wmsread with Optional Parameters **9-12**
Add a Legend to Your Map **9-12**
Retrieve Your Map with WebMapServer.getMap **9-19**

Modify Your Map Request **9-24**
Set Map Request Geographic Limits and Time **9-24**
Edit Web Map Request URL Manually **9-25**

Overlay Multiple Layers **9-27**
Create Composite Map of Multiple Layers from One Server **9-27**
Combine Layers from One Server with Data from Other Sources **9-28**
Drape Orthoimagery Over DEM **9-29**

Animate Data Layers **9-33**
Create Movie of Terra/MODIS Maps **9-33**
Create Animated GIF File of WMS Maps **9-34**
Animate Time-Lapse Radar Observations **9-36**

Display Animation of Radar Images over GOES Backdrop **9-39**

Retrieve Data from Web Map Server **9-41**
Merge Elevation Data with Rasterized Vector Data **9-42**
Display Merged Elevation and Bathymetry Layer (SRTM30 Plus) **9-44**
Drape WMS Imagery onto Elevation Data **9-46**

Save Your Favorite Servers **9-49**

Explore Other Layers using a Capabilities Document **9-50**

Write WMS Images to a KML File **9-53**

Search for Layers Outside the Database **9-55**

Troubleshoot WMS Servers **9-56**
Connection Errors **9-56**
Wrong Scale **9-57**
Problems with Geographic Limits **9-58**
Problems with Server Changing LayerName **9-58**
Non-EPSG:4326 Coordinate Reference Systems **9-58**
Map Not Returned **9-59**
Unsupported WMS Version **9-60**
Other Unrecoverable Server Errors **9-60**

Troubleshoot Access to the Hosted WMS Database **9-61**

Introduction to Web Map Display **9-62**
Web Map Coordinate Systems **9-64**

Basic Workflow for Displaying Web Maps **9-66**
Workflow Summary **9-66**

Display a Web Map **9-67**

Select a Base Layer Map **9-68**

Specify a Custom Base Layer **9-70**

Specify a WMS Layer as a Base Layer **9-72**

Add an Overlay Layer to the Map **9-74**

Add Line, Polygon, and Marker Overlay Layers to Web Maps **9-76**

Remove Overlay Layers on a Web Map **9-82**

View Multiple Web Maps in a Browser **9-86**

Navigate a Web Map **9-89**

Close a Web Map **9-92**

Annotate a Web Map with Measurement Information **9-93**

Compositing and Animating Web Map Service (WMS) Meteorological Layers **9-97**

Troubleshoot Common Problems with Web Maps **9-112**
Why Does My Web Map Contain Empty Tiles? **9-112**
Why Does My Web Map Lose Detail When I Zoom In? **9-112**

10 Mapping Applications

Geographic Statistics for Point Locations on a Sphere **10-2**
Geographic Means **10-2**
Geographic Standard Deviation **10-3**

Equal-Areas in Geographic Statistics **10-6**
Geographic Histograms **10-6**
Converting to an Equal-Area Coordinate System **10-7**

Navigation **10-9**
What Is Navigation? **10-9**
Conventions for Navigational Functions **10-9**

Fix Position **10-11**
Some Possible Situations **10-11**
Using navfix **10-14**
A Numerical Example of Using navfix **10-16**

Plan the Shortest Path **10-20**

Display Navigational Tracks **10-23**

Dead Reckoning **10-26**

Drift Correction **10-30**

Time Zones **10-32**

11 Map Projections — Alphabetical List

Getting Started

This chapter provides step-by-step examples of basic Mapping Toolbox capabilities and guides you toward examples and documentation that can help answer your questions. For an alphabetical list of functions click on `MATLAB Functions` link at bottom of main Mapping Toolbox page.

- "Mapping Toolbox Product Description" on page 1-2
- "Acknowledgments" on page 1-3
- "Create Your First World Map" on page 1-4
- "Tour Boston with the Map Viewer App" on page 1-9
- "Getting More Help" on page 1-23

Mapping Toolbox Product Description

Analyze and visualize geographic information

Mapping Toolbox provides algorithms and functions for transforming geographic data and creating map displays. You can visualize your data in a geographic context, build map displays from more than 60 map projections, and transform data from a variety of sources into a consistent geographic coordinate system.

Mapping Toolbox supports a complete workflow for managing geographic data. You can import vector and raster data from a wide range of file formats and web map servers. The toolbox lets you process and customize data using trimming, interpolation, resampling, coordinate transformations, and other techniques. Data can be combined with base map layers from multiple sources in a single map display. You can export data in file formats such as shapefile, GeoTIFF, and KML.

Acknowledgments

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

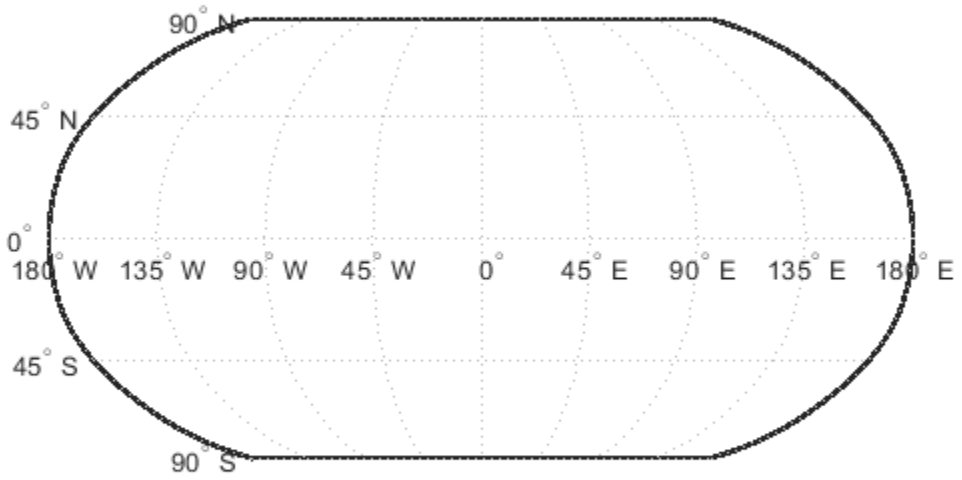
Except where noted, the information contained in example and sample data files (found in *matlabroot*`/examples/map/data` and *matlabroot*`/toolbox/map/mapdata`) is derived from publicly available digital data sets. These data files are provided as a convenience to Mapping Toolbox users. MathWorks® makes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

Create Your First World Map

This example shows how to use the Mapping Toolbox to create a world map. Geospatial data can be voluminous, complex, and difficult to process. Mapping Toolbox functions handle many of the details of loading and displaying geospatial data, and use built-in data structures that facilitate data storage. *Spatial data* refers to data describing location, shape, and spatial relationships. *Geospatial data* is spatial data that is in some way georeferenced, or tied to specific locations on, under, or above the surface of a planet.

Create an empty map axes, ready to hold the data of your choice. The function `worldmap` automatically selects a reasonable choice for your map projection and coordinate limits. To display a world map, the function chose a Robinson projection centered on the prime meridian and the equator (0° latitude, 0° longitude).

```
worldmap world
```



Import low-resolution world coastline data. The coastline data is a set of discrete vertices that, when connected in the order given, approximate the coastlines of continents, major islands, and inland seas. The vertex latitudes and longitudes are stored as vectors in a MAT-file. Load the MAT-file and view the variables in the workspace.

```
load coastlines
whos
```

```
  Name            Size            Bytes  Class     Attributes
```

```
  coastlat      9865x1             78920  double
  coastlon      9865x1             78920  double
```

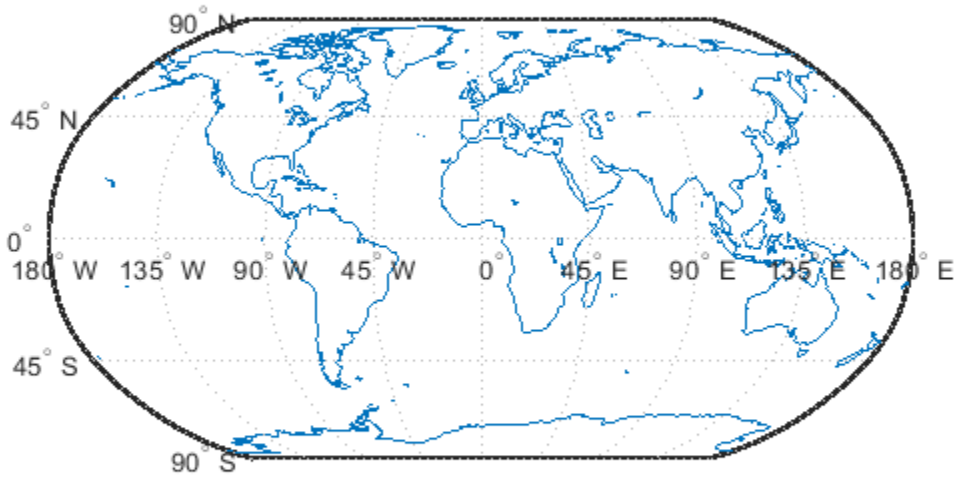
Determine how many separate elements are in the coastline data vectors. Even though there is only one vector of latitudes, `coastlat`, and one vector of longitudes, `coastlon`, each of these vectors contain many distinct polygons, forming the worlds coastlines. These vectors use NaN separators and NaN terminators to divide each vector into multiple parts.

```
[latcells, loncells] = polysplit(coastlat, coastlon);
numel(latcells)
```

```
ans = 241
```

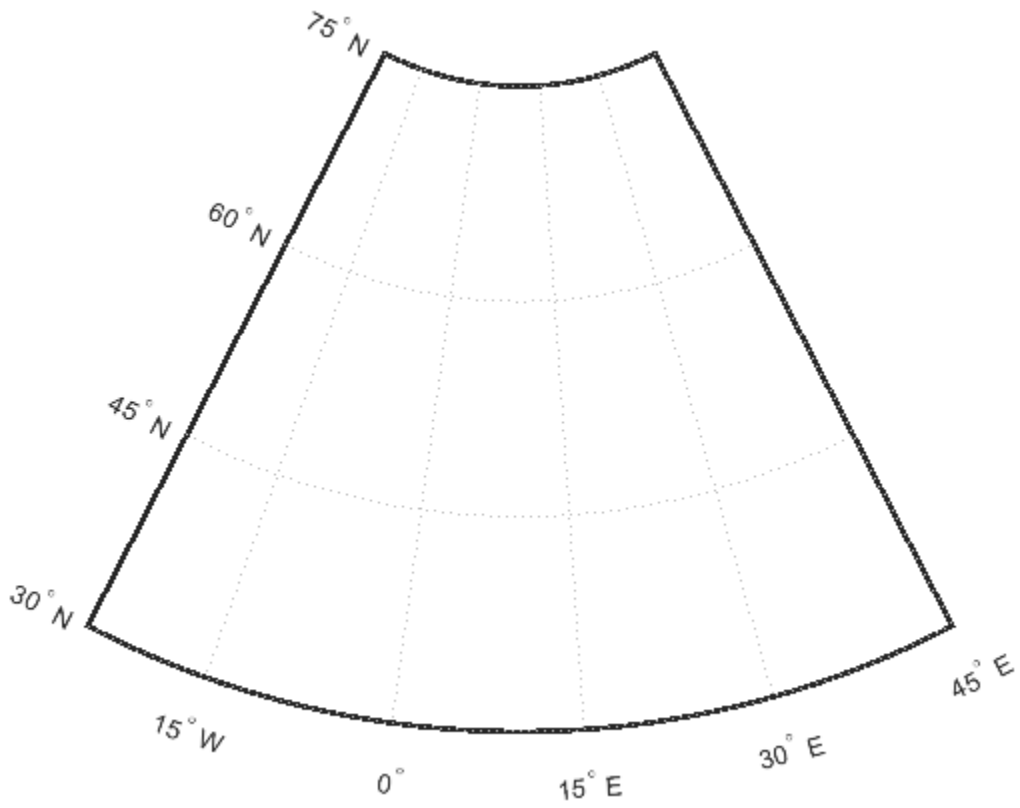
Plot the coastline data on the map axes using the `plotm` function. `plotm` is the geographic equivalent of the MATLAB `plot` function. It accepts coordinates in latitude and longitude, transforms them to x and y via a specified map projection, and displays them in a figure axes. In this example, `worldmap` uses the Robinson projection.

```
plotm(coastlat, coastlon)
```



Create a new map axes for plotting data over Europe. This time, specify a return argument for the `worldmap` function to get a handle to the figure's axes. The axes object on which map data is displayed is called a *map axes*. In addition to the graphics properties common to any MATLAB axes object, a map axes object contains additional properties covering map projection type, projection parameters, map limits, etc. The `getm` and `setm` functions and others allow you to access and modify these properties.

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



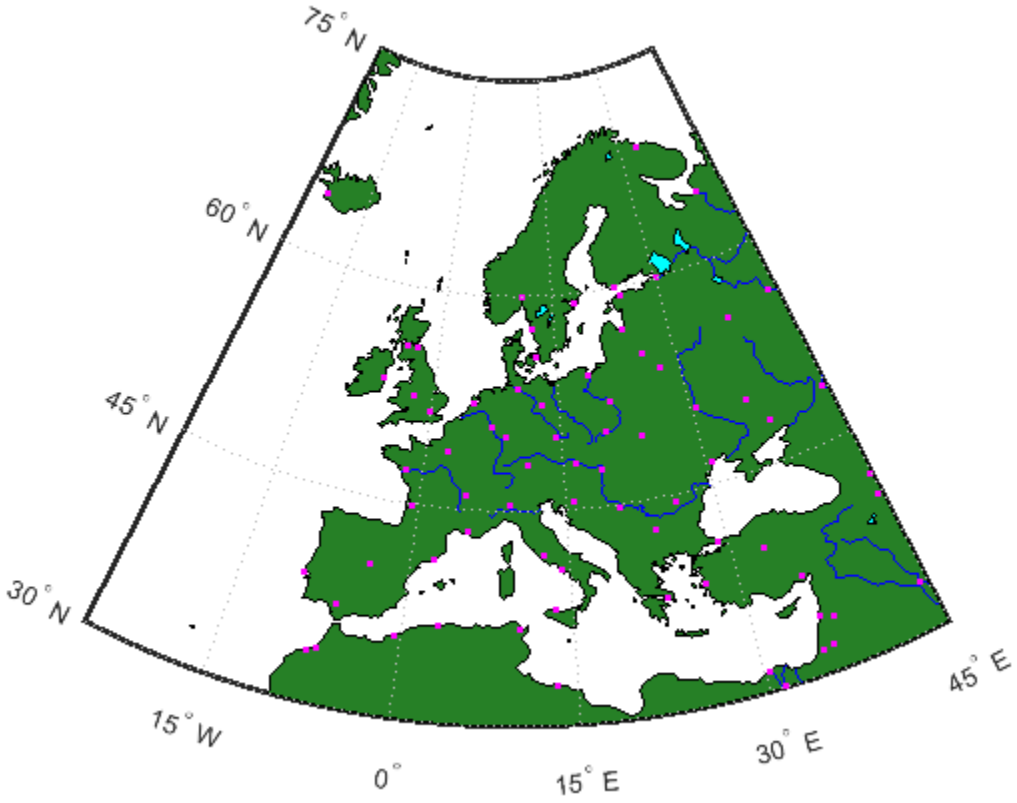
Determine which map projection `worldmap` is using.

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

```
ans =
'eqdconic'
```

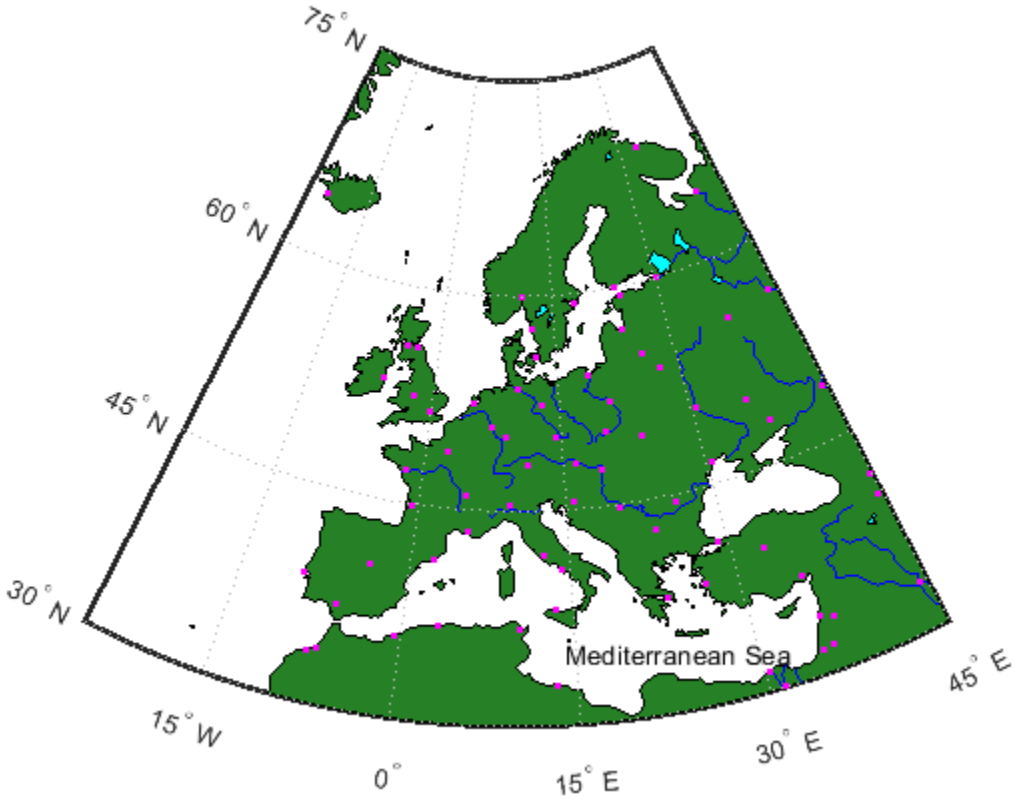
Add data to the map of Europe by using the `geoshow` function to import and display several sample shapefiles. Note how the `geoshow` function can plot data directly from files onto a map axes without first importing it into the workspace. To change the color of the marker, use the `MarkerEdgeColor` property and, for some markers, the `MarkerFaceColor` property.

```
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', 'magenta')
```



Place a label on the map to identify the Mediterranean Sea.

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


75° N
60° N
45° N
30° N
15° W
0°
15° E
30° E
45° E
Mediterranean Sea

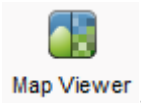
Tour Boston with the Map Viewer App

The Map Viewer app 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
- Import, reorder, symbolize, hide, and delete data layers
- Identify coordinate locations
- List data attributes
- Display selected data attributes as *data tips* (signposts that identify attribute values, such as place names or route numbers)

The following example illustrates these capabilities.

Open the Map Viewer App

1 Open the Map Viewer app. On the **Apps** tab, in the **Image Processing and Computer Vision** section, click **Map Viewer** 

. You can also start the Map Viewer using the `mapview` command. The Map Viewer opens with a blank canvas. (No data is present.)

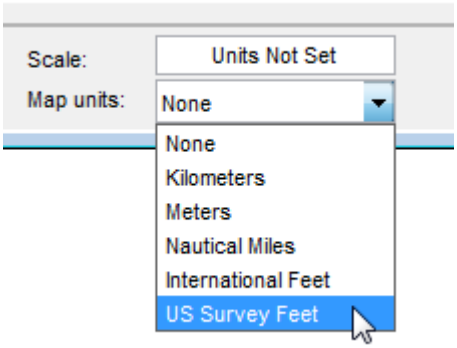
Note that The Map Viewer is designed primarily for working with data sets that refer to a projected map coordinate system (as opposed to a geographic, latitude-longitude system), so the coordinate axes are named *X* and *Y*.

2 Import map data. In the Map Viewer, select the **File** menu and then choose **Import From File**. Navigate to the *`matlabroot`*`/examples/map/data` folder, where *`matlabroot`* represents your MATLAB® installation folder, and open the GeoTIFF file `boston.tif`.

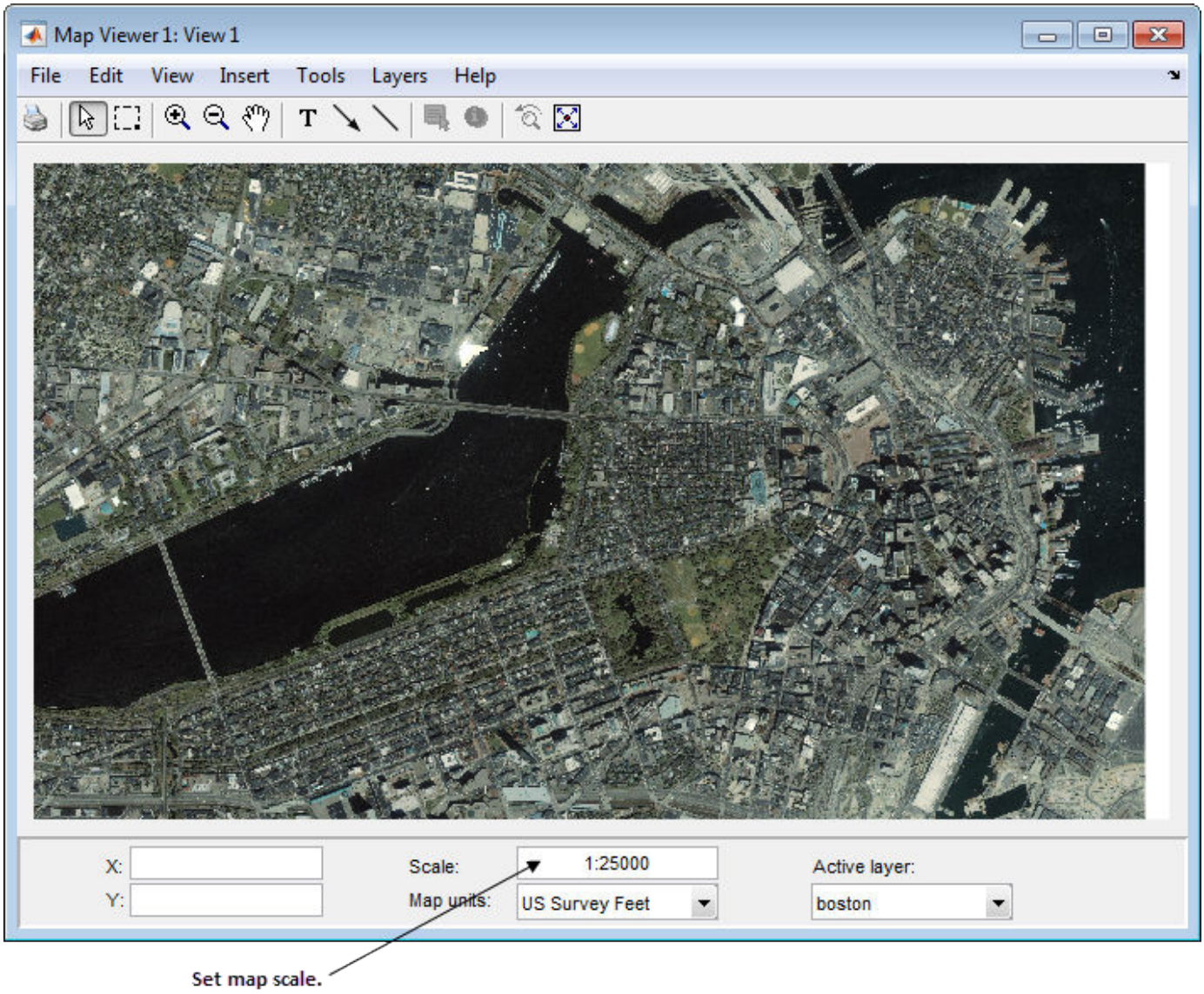
The file opens in the Map Viewer. The image is a visible red, green, and blue composite from a georeferenced IKONOS-2 panchromatic/multispectral product created by GeoEye. Copyright © GeoEye, all rights reserved. For further information about the image, refer to the text files `boston.txt` and `boston_metadata.txt`.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

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

3 Set the map scale in the Map Viewer. To do this, you must first set the map distance units. Click the **Map units** menu at the bottom center and select `US Survey Feet`.



4 Set the map scale. Type `1:25000` in the **Scale** box, which is above the **Map units** menu, and press Enter. The Map Viewer now looks like this.



5 Get the map coordinates for a location on the map, interactively. Place the cursor over a location on the map. The example puts the cursor over the bridge that goes over the pond in Boston Garden. The map coordinates for this location are shown at the lower left as `772,423.18` feet easting (**X**), `2,954,372.40` feet northing (**Y**), in Massachusetts State Plane coordinates.

6 Import a vector data layer. For this example, import a line shapefile that contains data about the streets and highways in the central Boston area.

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

The `shaperead` function returns the data as a geographic data structure.

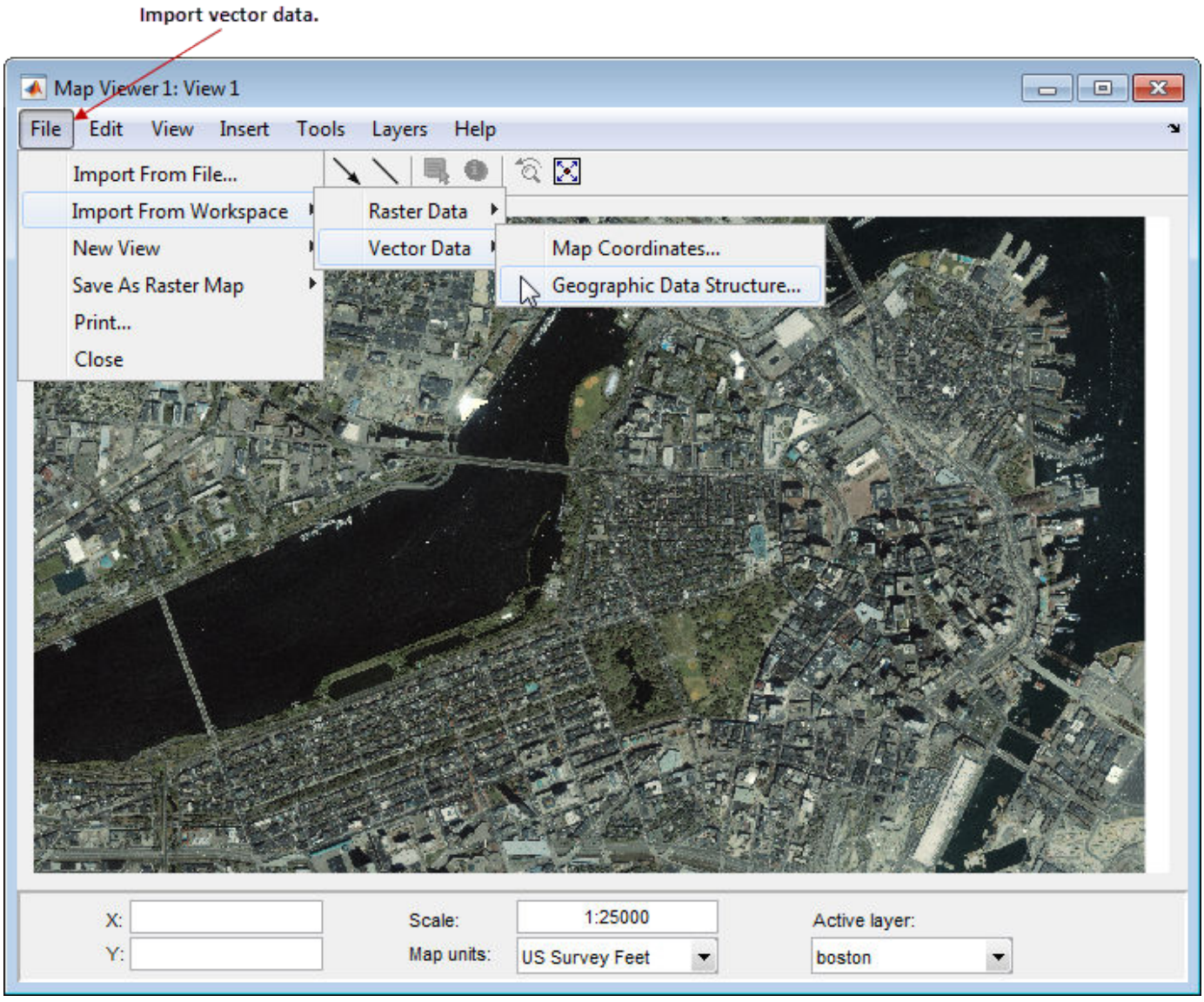
7 Convert the `X` and `Y` coordinate fields of `boston_roads.shp` from meters to U.S. survey feet. As is frequently the case when overlaying geodata, the coordinate system used by `boston_roads.shp` (in units of meters) does not completely agree with the one for the satellite

image, `boston.tif` (in units of feet). If you were to ignore this, the two data sets would be out of registration by a large distance.

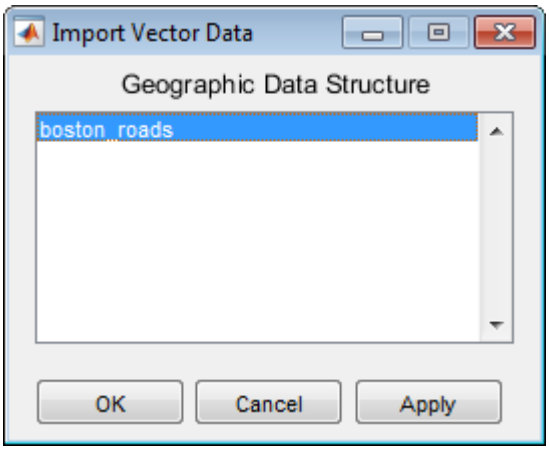
```
surveyFeetPerMeter = unitsratio('survey feet','meter');
for k = 1:numel(boston_roads)
    boston_roads(k).X = surveyFeetPerMeter * boston_roads(k).X;
    boston_roads(k).Y = surveyFeetPerMeter * boston_roads(k).Y;
end
```

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

8 In the Map Viewer **File** menu, select **Import From Workspace > Vector Data > Geographic Data Structure**.

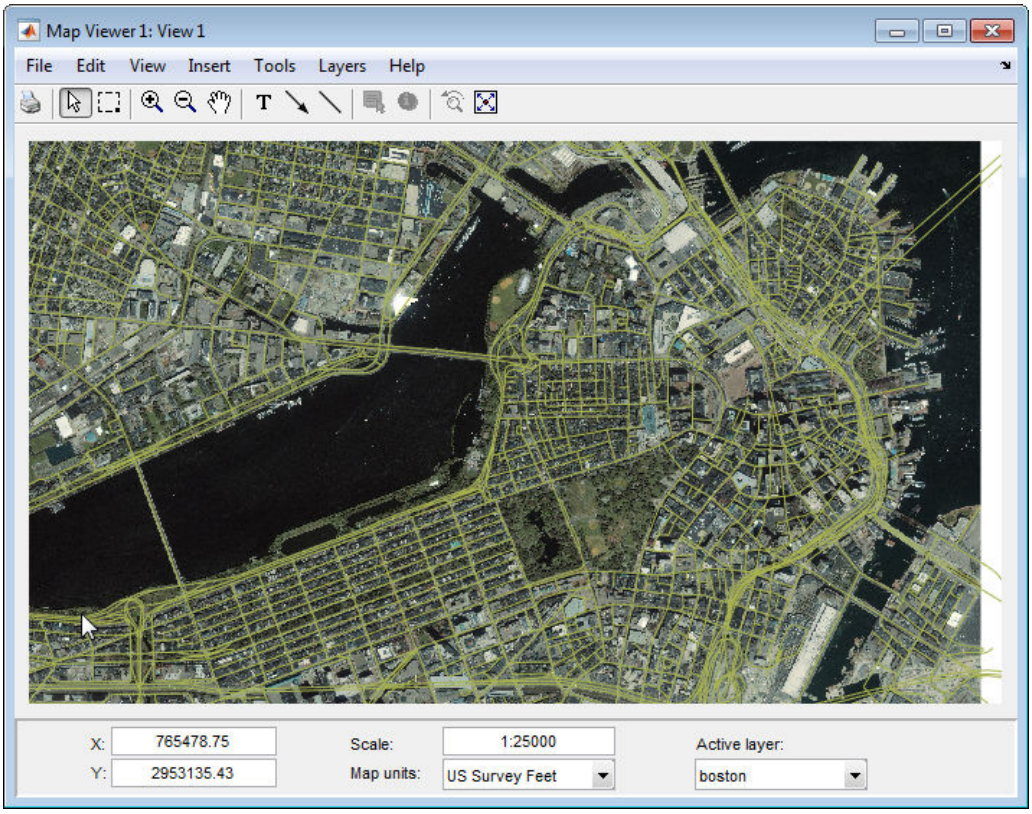


In the Import Vector Data dialog box, select the variable `boston_roads` as the data to import from the workspace, and click **OK**.



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

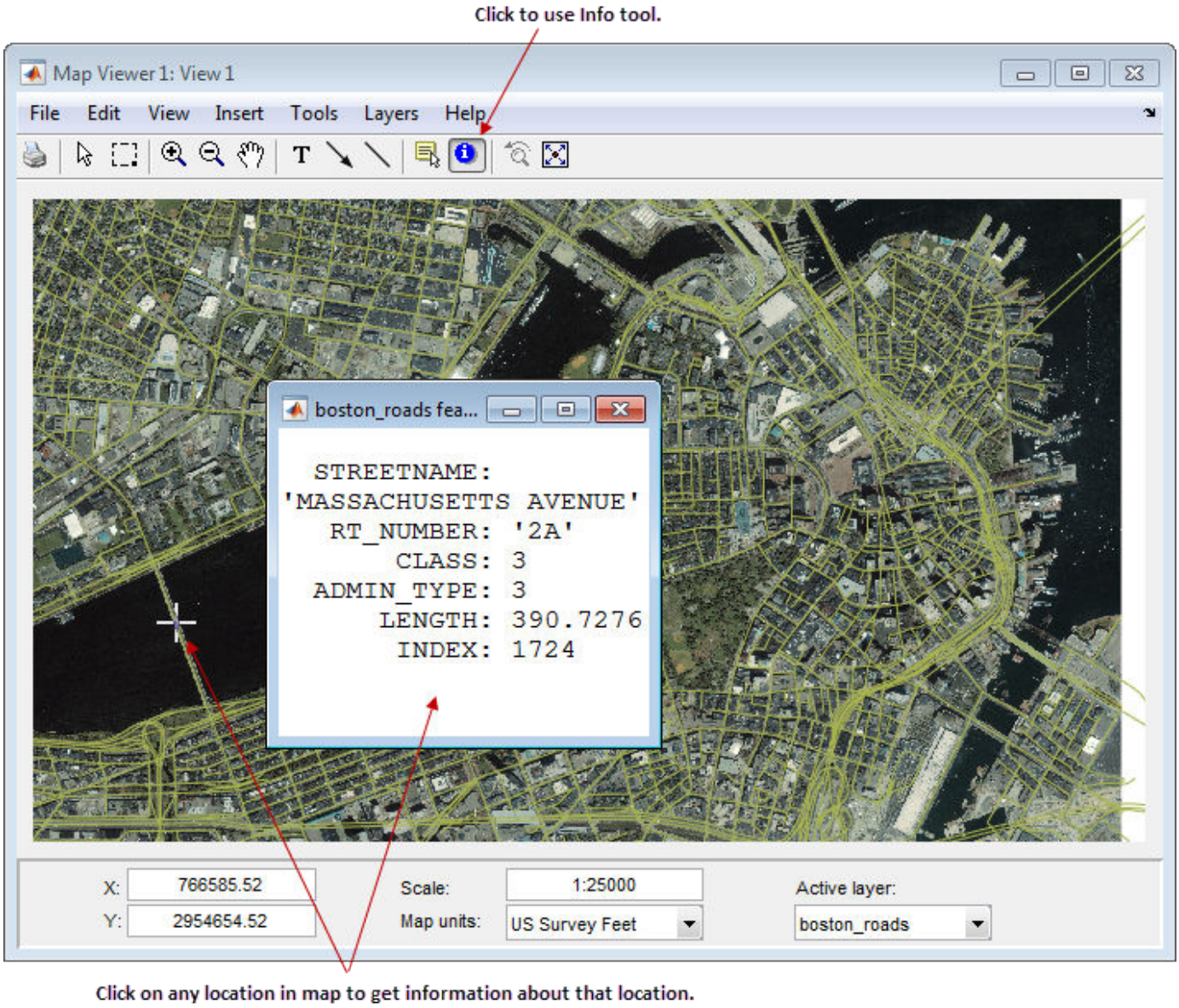
9 After the Map Viewer finishes importing the roads layer, it selects a random color and renders all the shapes with that color as solid lines. The view looks like this.



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

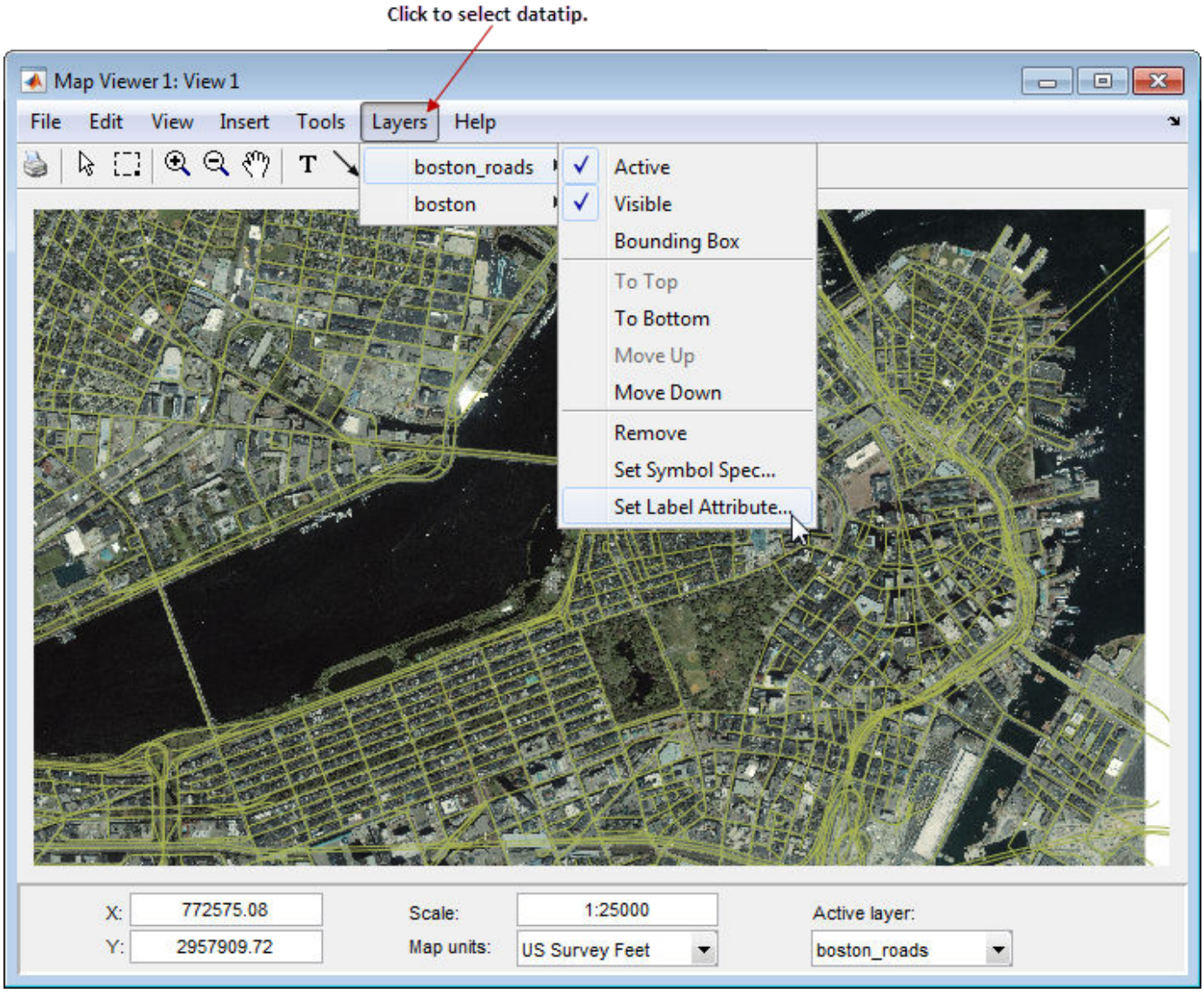
10 Explore the attributes of the vector layer. First, make the vector layer the active layer using the **Active layer** menu at the bottom right. Select `boston_roads`. You can designate any layer to be the active layer; it does not need to be the topmost layer. By default, the first layer imported is active. Changing the active layer has no visual effect on the map. Doing so allows you to query attributes of the layer you select. For example, once you make the vector layer the active layer,

the **Info** tool button near the right end of the toolbar becomes enabled. Select the **Info** tool, the cursor changes to a cross-hairs shape. Click any location on the map to view attributes of the selected object.

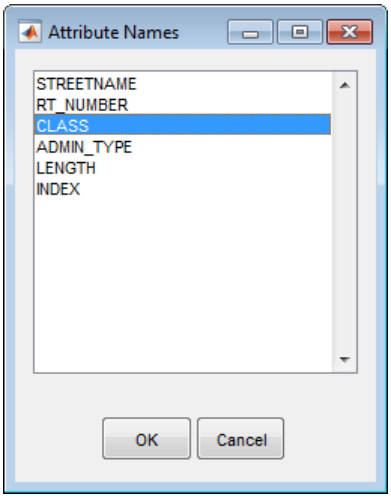


The selected road is Massachusetts Avenue (Route 2A). As the above figure shows, the `boston_roads` vectors have six attributes, including an implicit `INDEX` attribute added by the Map Viewer. Use this tool to explore other roads. Dismiss open Info windows by clicking their close boxes.

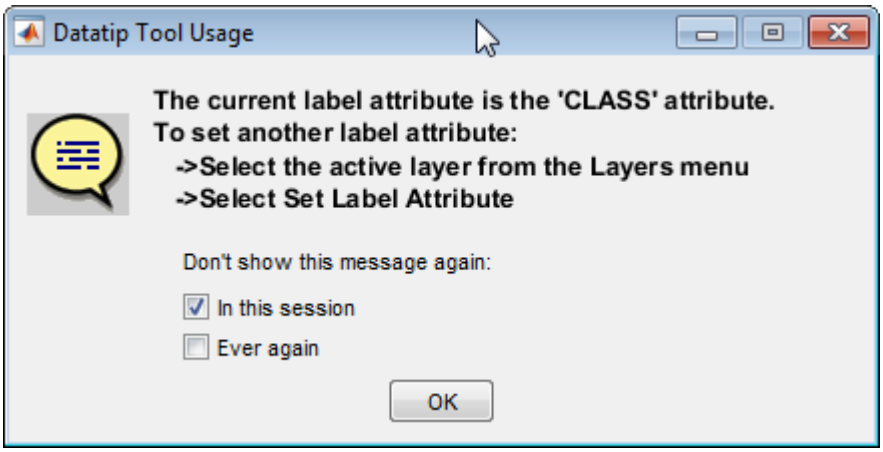
11 Use a data tip to annotate the map with other attribute values. From the **Layers** menu, select **boston_roads > Set Label Attribute**.



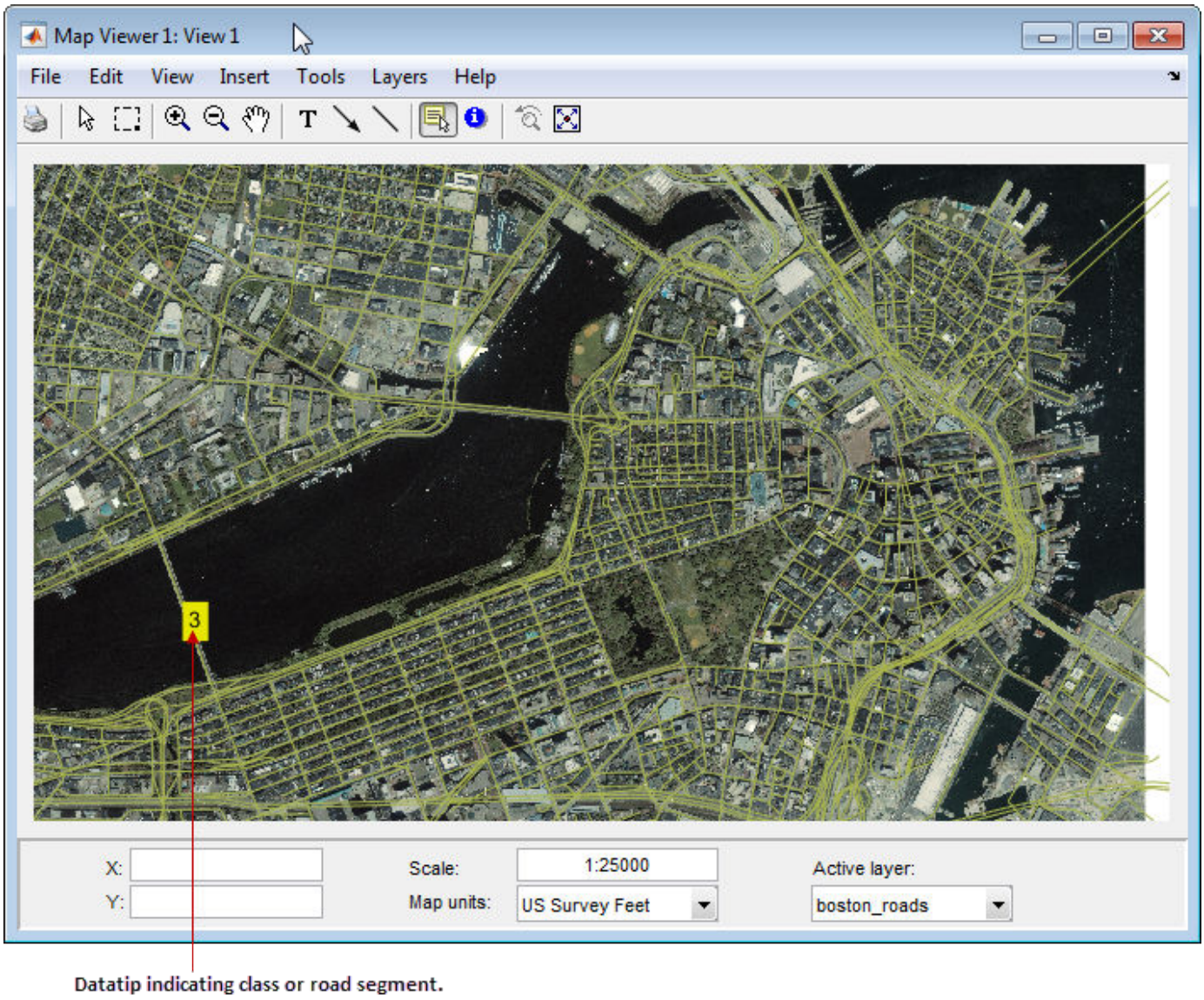
From the list in the Attribute Names dialog, select `CLASS` and click **OK**.



12 From the Tools menu, select the **Datatip** tool. A dialog box appears to remind you how to change attributes. Click **OK** to dismiss the box.



The cursor assumes a cross-hairs (+) shape. Click on a road segment in the map and the data tip tool puts a small marker on the road that contains a numeric identifier that indicates the administrative class. The class of the road crossing the Charles river we explored earlier is of class 3.



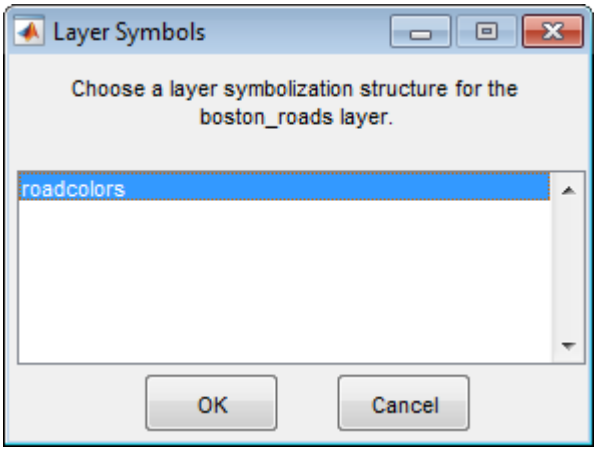
13 You can change how the roads are rendered by identifying an attribute to which to key line symbology. Color roads according to their `CLASS` attribute, which takes on the values 1:6. Do this by creating a symbolspec in the workspace. A symbolspec is a cell array that associates attribute names and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

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

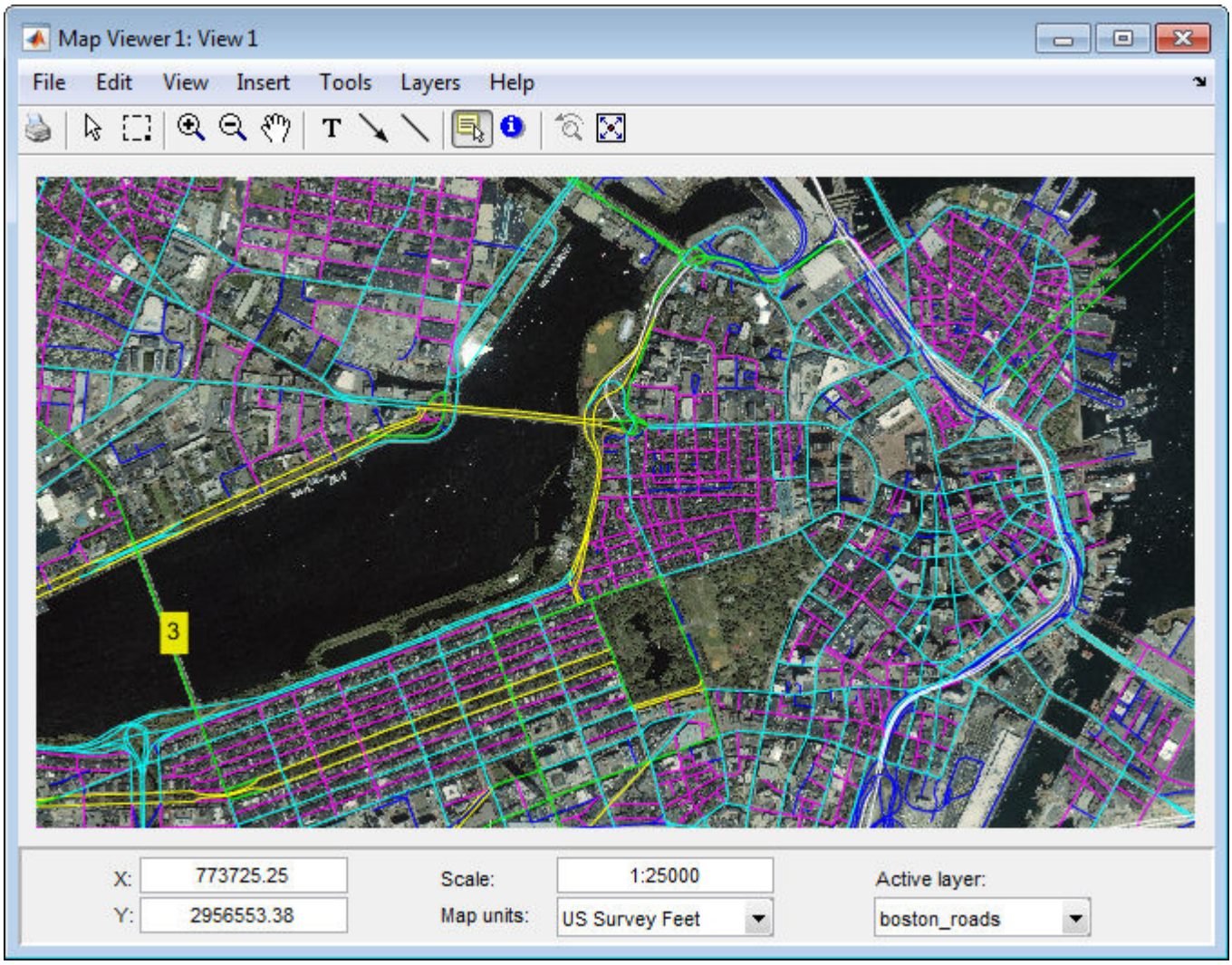
The following output appears:

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

14 The Map Viewer recognizes and imports symbolspecs from the workspace. To apply the one you just created, from the **Layers** menu, select **boston_roads > Set Symbol Spec**. From the Layer Symbols dialog, select the `roadcolors` symbolspec you just created and click **OK**.



After the Map Viewer has read and applied the symbolspec, the map looks like this.



15 Remove the data tips before going on. To dismiss data tips, right-click one of them and select **Delete all data tips** from the menu that appears.

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

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

17 Convert the coordinates of these landmarks to units of survey feet before importing them into Map Viewer. The locations for these landmarks are given in meters.

```
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
```

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

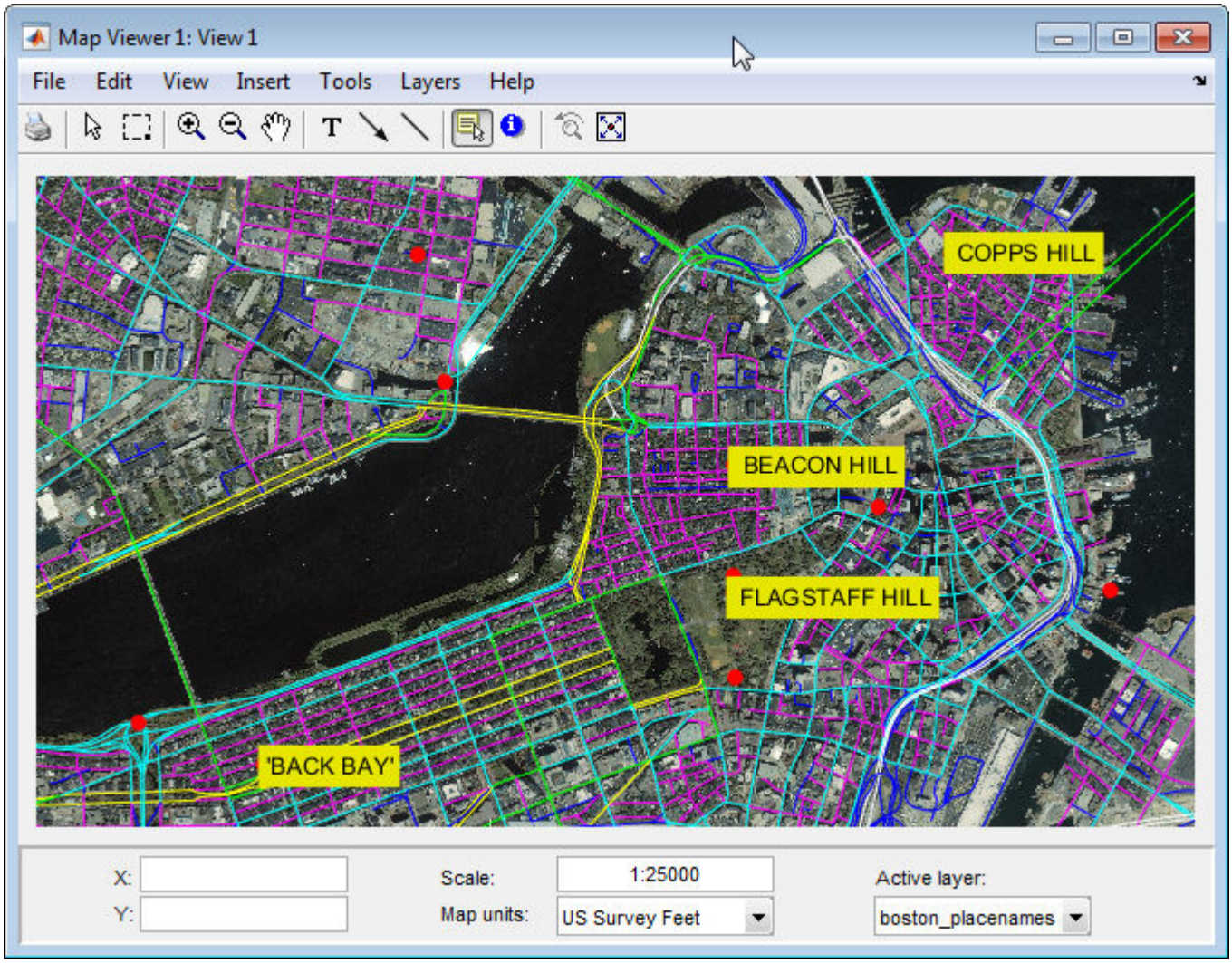
19 The `boston_placenames` markers are symbolized as small `x` markers, but these markers do not show up over the orthophoto. To solve this problem, create a symbolspec for the markers to represent them as red filled circles. At the MATLAB command line, type:

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

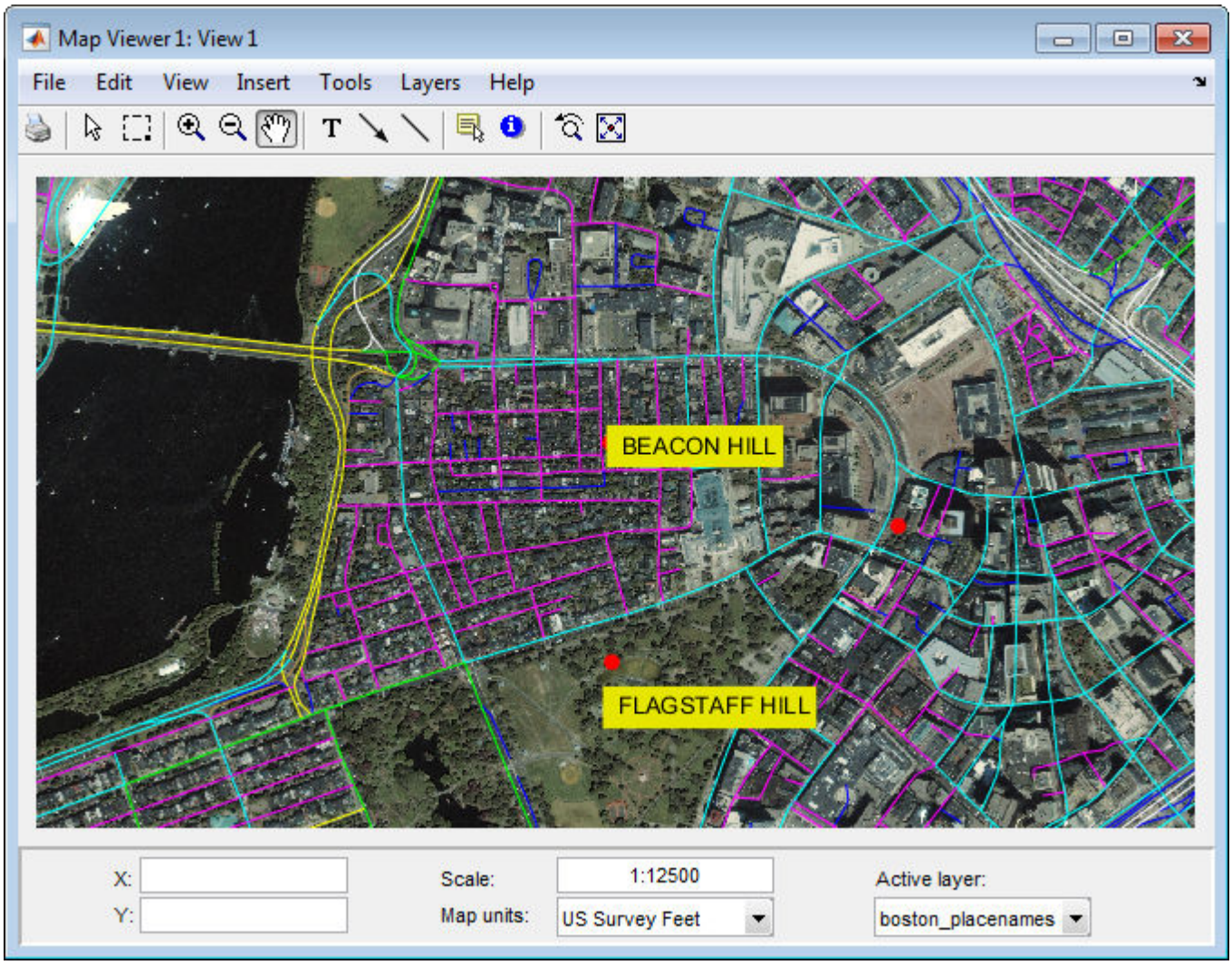
The `Default` keyword causes the specified symbol to be applied to all point objects in a given layer unless specifically overridden by an attribute-coded symbol in the same or a different symbolspec.

20 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.

21 To see the name of a Boston place name, make `boston_placenames` the currently active layer (using the Active layer menu, and then select **Datatip** from the Tools menu. The cursor changes to a cross-hair shape. Click any red circle and the tool places a data tip annotation on the map with the name of the location.



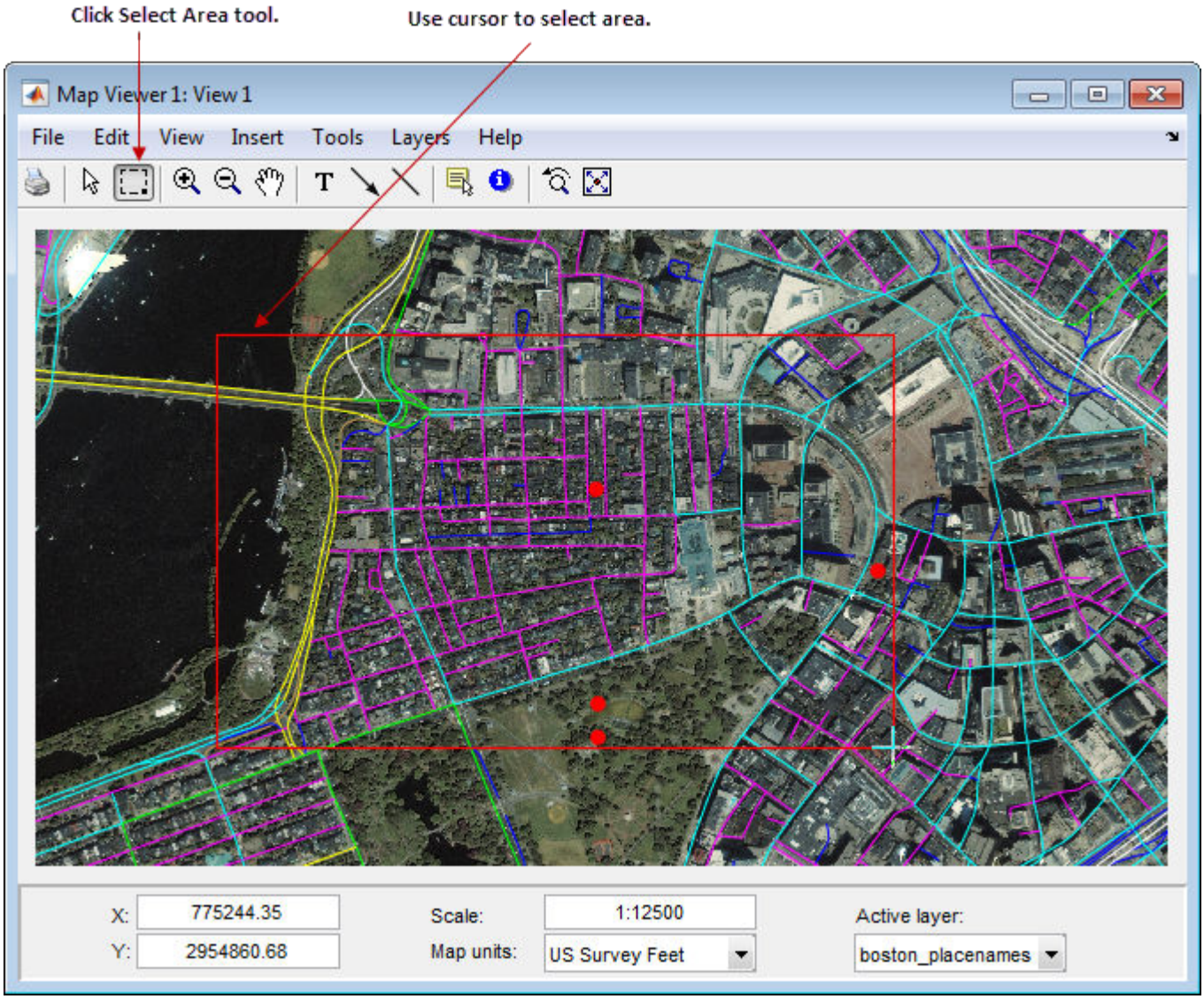
22 Zoom in on Beacon Hill for a closer view of the Massachusetts State House and Boston Common. Select the **Zoom in** tool; move the (magnifier) cursor until the **X** readout is approximately `774,011` and the **Y** readout is roughly `2,955,615`; and click once to enlarge the view. The scale changes to about `1:12,500` and the map appears as below.



23 From the **Tools** menu, choose **Select Annotations** to change from the **Datatip** tool back to the original cursor. Right-click any of the data tips and select **Delete all datatips** from the pop-up context menu. This clears the place names you added to the map.

24 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.

Note The **Select area** tool is not supported in MATLAB Online™. To view a particular region on the map, use the **Zoom in**, **Zoom out**, and **Pan** tools instead.



25 In order to be able to save a file in the next step, change your working folder to a writable folder.

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

In the Export to File dialog, navigate to a folder 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 world file, `central_boston.tfw`, is created along with the TIF.

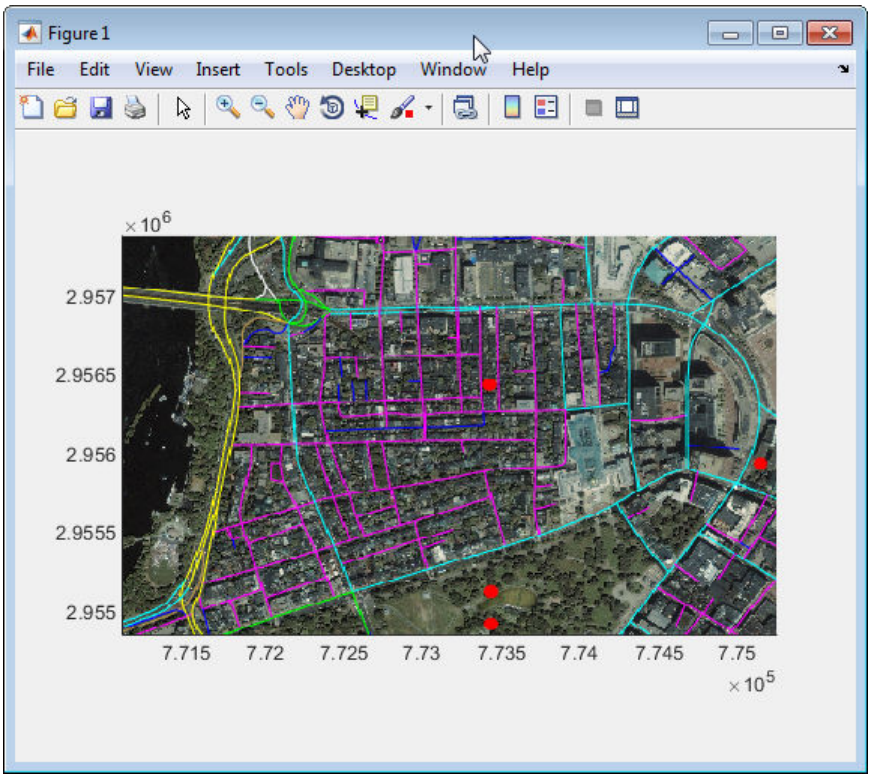
Whenever you save a raster map in this manner, two files are created:

- An image file (*`file`*`.tif`, *`file`*`.png`, or *`file`*`.jpg`)
- An accompanying world file that georeferences the image (*`file`*`.tfw`, *`file`*`.pgw`, or *`file`*`.jgw`)

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

27 Read in the saved image and its colormap with the MATLAB function `imread`, create a reference object for it by reading in `central_boston.tfw` with `worldfileread`, and display the map with `mapshow`:

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



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

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

When you are through with a viewing session, close the Map Viewer using the window's close box or select **Close** from the **File** menu. For more information about the Map Viewer, see the `mapview` reference page.

Getting More Help

Ways to Get Mapping Toolbox Help

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

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

Understanding Map Data

- “What Is a Map?” on page 2-2
- “What Is Geospatial Data?” on page 2-3
- “Vector Geodata” on page 2-4
- “Inspect and Display Vector Map Data” on page 2-5
- “Raster Geodata” on page 2-7
- “Display Shaded Relief Map Using Raster Data” on page 2-8
- “Combine Vector and Raster Geodata on the Same Map” on page 2-10
- “Create and Display Polygons” on page 2-12
- “Segments Versus Polygons” on page 2-20
- “Geographic Data Structures” on page 2-22
- “Georeferenced Raster Data” on page 2-30
- “Construct a Global Data Grid” on page 2-32
- “Precompute the Size of a Data Grid” on page 2-34
- “Geolocated Data Grids” on page 2-35
- “Geographic Interpretations of Geolocated Grids” on page 2-39
- “Unprojecting a Digital Elevation Model (DEM)” on page 2-42
- “Creating a Half-Resolution Georeferenced Image” on page 2-53
- “Georeferencing an Image to an Orthotile Base Layer” on page 2-58
- “Find Geospatial Data Online” on page 2-70
- “Find Vector Geodata” on page 2-71
- “Find Geospatial Raster Data” on page 2-73
- “Functions that Read and Write Geospatial Data” on page 2-75
- “Export Vector Geodata” on page 2-78
- “Exporting Vector Data to KML” on page 2-79
- “Export KML Files for Viewing in Earth Browsers” on page 2-90
- “Select Shapefile Data to Read” on page 2-94
- “Exporting Images and Raster Grids to GeoTIFF” on page 2-98
- “Converting Coastline Data (GSHHG) to Shapefile Format” on page 2-113

What Is a Map?

Mapping Toolbox software manipulates electronic representations of geographic data. It lets you import, create, use, and present geographic data in various forms and to various 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. For more information, see “Vector Geodata” on page 2-4 and “Raster Geodata” on page 2-7.

Vector Geodata

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

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

For an example of vector data, see “Inspect and Display Vector Map Data” on page 2-5. For further information on how Mapping Toolbox software manages map projections, see the Getting Started topic. For details on data structures that the toolbox uses to represent vector geodata, see “Geographic Data Structures” on page 2-22.

Inspect and Display Vector Map Data

This example shows how to display vector map data and examine vector data values.

Load vector data set MAT-file of world coastlines and look at the variables created in the workspace. The variables `coastlat` and `coastlon` are vectors which together form a vector map of the coastlines of the world.

```
load coastlines
whos
```

```
  Name             Size            Bytes  Class     Attributes

  coastlat      9865x1             78920  double
  coastlon      9865x1             78920  double
```

View a map of this vector data. The example presents the map using a Mercator projection. A map projection displays the surface of a sphere (or a spheroid) in a two-dimensional plane. 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.

```
axesm mercator
framem
plotm(coastlat,coastlon)
```

Inspect the first 20 coordinate values of the coastline vector data set.

```
[coastlat(1:20) coastlon(1:20)]

ans = 20×2

  -83.8300 -180.0000
  -84.3300 -178.0000
  -84.5000 -174.0000
  -84.6700 -170.0000
  -84.9200 -166.0000
  -85.4200 -163.0000
  -85.4200 -158.0000
  -85.5800 -152.0000
  -85.3300 -146.0000
  -84.8300 -147.0000
      ⋮
```

To see where these coastline vector points fall on the map, plot them in red. As you might have deduced by looking at the first column of the data, there is only one continent that lies below -80 latitude: Antarctica.

```
plotm(coastlat(1:20),coastlon(1:20),'r')
```

Raster Geodata

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

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

A raster can encode either an average value across a cell or a value sampled (posted) at the center of that cell. While geolocated data grids explicitly indicate which type of values are present (see “Geolocated Data Grids” on page 2-35), external metadata/user knowledge is required to be able to specify whether a regular data grid encodes averages or samples of values. To see an example, view “Display Shaded Relief Map Using Raster Data” on page 2-8.

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, octree, or otherwise.

The `topo60c` MAT-file, which contains global terrain data, is an example of a DEM. In this 180-by-360 matrix, each row represents one degree of latitude, and each column represents one degree of longitude. Each element of this matrix is the average elevation, in meters, for the one-degree-by-one-degree region of the Earth to which its row and column correspond.

Remotely Sensed Image Data

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

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

Display Shaded Relief Map Using Raster Data

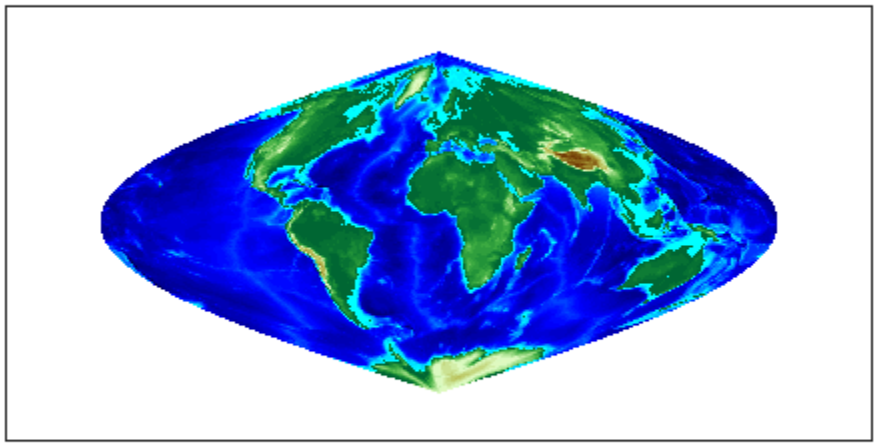
This example shows how to generate a shaded relief map using a raster data set, also known as a data grid. Note that the content, symbolization, and the projection of the map are completely independent. The structure and content of the `topo60c` data grid are the same no matter how you display it, although how it is projected and symbolized can affect its interpretation.

Load elevation raster data and a geographic cells reference object.

```
load topo60c
```

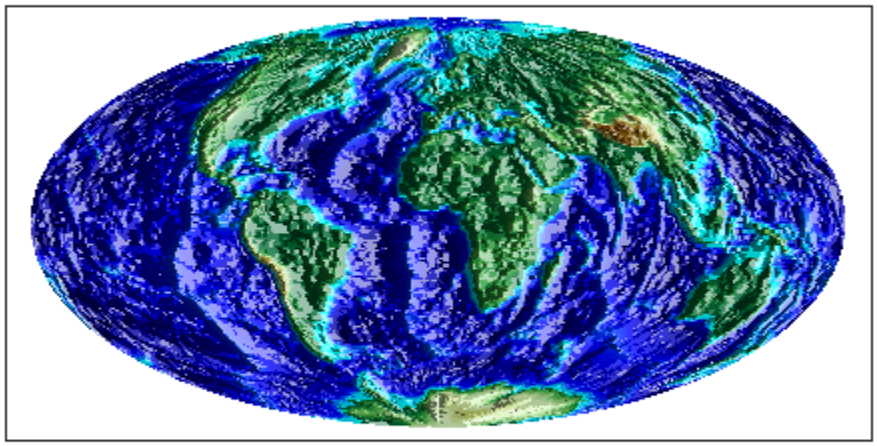
Create a map axes object using a sinusoidal projection. Display the data using a shaded relief map. One way to do this is to use `geoshow` and apply a colormap appropriate for elevation data using `demcmap`.

```
axesm sinusoid
geoshow(topo60c,topo60cR,'DisplayType','texturemap')
demcmap(topo60c)
```



Create a new map axes object using a Hammer projection. Then, display the elevation data as a lighted shaded relief map using `meshlsrm`. By default, the `meshlsrm` function applies a colormap appropriate for elevation data and illuminates the map from the east.

```
figure
axesm hammer
meshlsrm(topo60c,topo60cR)
```



Combine Vector and Raster Geodata on the Same Map

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).

Combining Raster Data and Vector Data on the Same Map

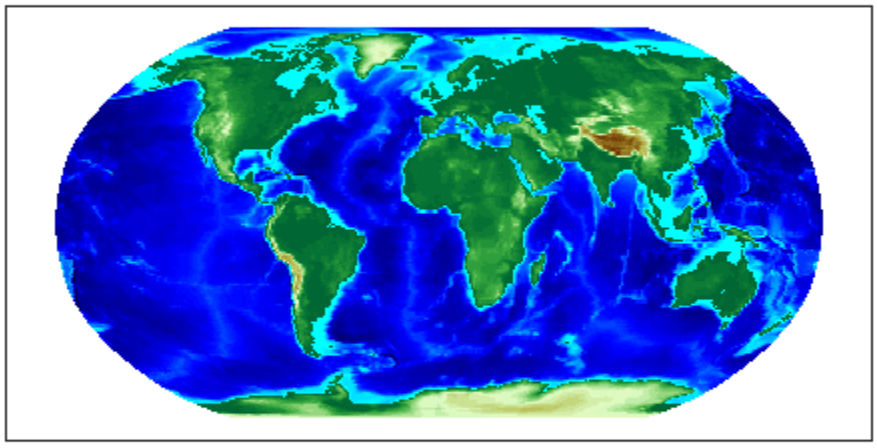
This example shows how to combine raster data and vector data on the same map using the `geoshow` function.

First, load elevation raster data and a geographic cells reference object. Get the coordinates of coastlines as vectors.

```
load topo60c
load coastlines
```

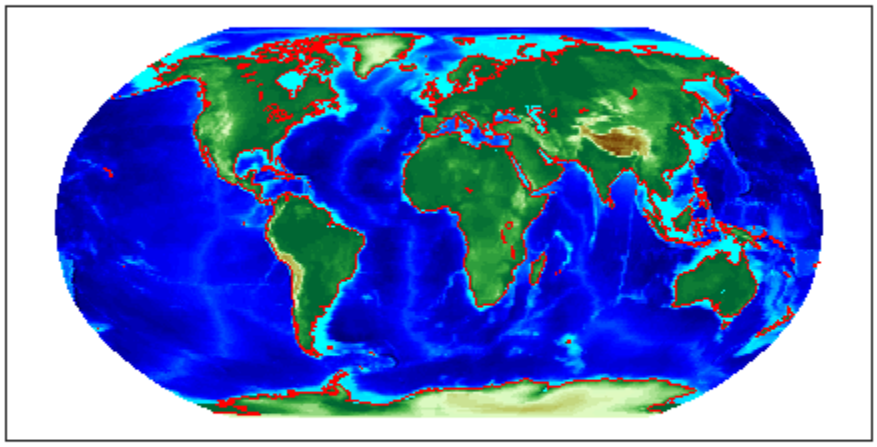
Create a map axes object that uses the Robinson projection. Then, display the raster and apply a colormap appropriate for elevation data.

```
axesm robinson
geoshow(topo60c,topo60cR,'DisplayType','texturemap')
demcmap(topo60c)
```



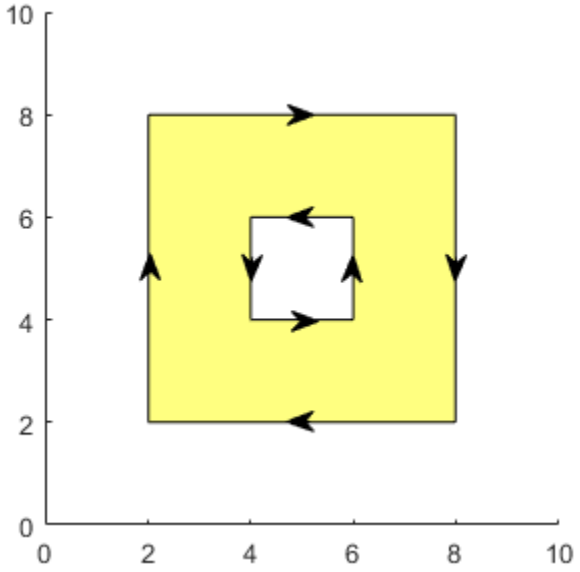
Display the vectors in red on top of the elevation map.

```
geoshow(coastlat,coastlon,'Color','r')
```



Create and Display Polygons

Polygons represent geographic objects that cover area, such as continents, islands, and lakes. They may contain holes or multiple regions. Create a polygon by listing vertices that define its boundaries without intersecting. The order of the vertices determines what parts of the polygon are filled. List external boundaries clockwise and internal boundaries counterclockwise, such that the inside of the polygon is always to the right of the boundary.



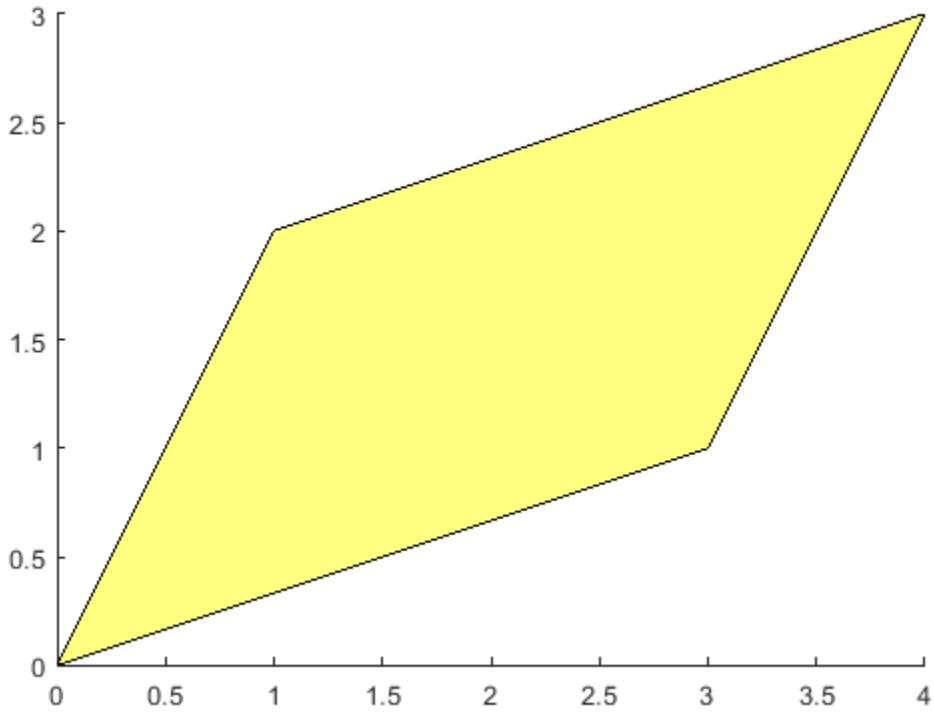
Simple Polygon

Display a simple polygon with one region and no holes. First, list its vertices in a clockwise order. Close the polygon by repeating the first vertex at the end of the list.

```
x1 = [0 3 4 1 0];
y1 = [0 1 3 2 0];
```

Display the vertices as a polygon using the `mapshow` function by specifying `'DisplayType'` as `'polygon'`.

```
mapshow(x1,y1,'DisplayType','polygon')
```



Polygons with Holes or Multiple Regions

Define polygons with multiple regions or holes by separating the boundaries with `NaN` values. List the vertices of external boundaries in a clockwise order and the vertices of internal boundaries in a counterclockwise order.

```
x2 = [0 1 8 6 0 NaN 1 4 2 1 NaN 5 6 7 3 5];
y2 = [0 6 8 2 0 NaN 1 3 5 1 NaN 3 5 7 6 3];
```

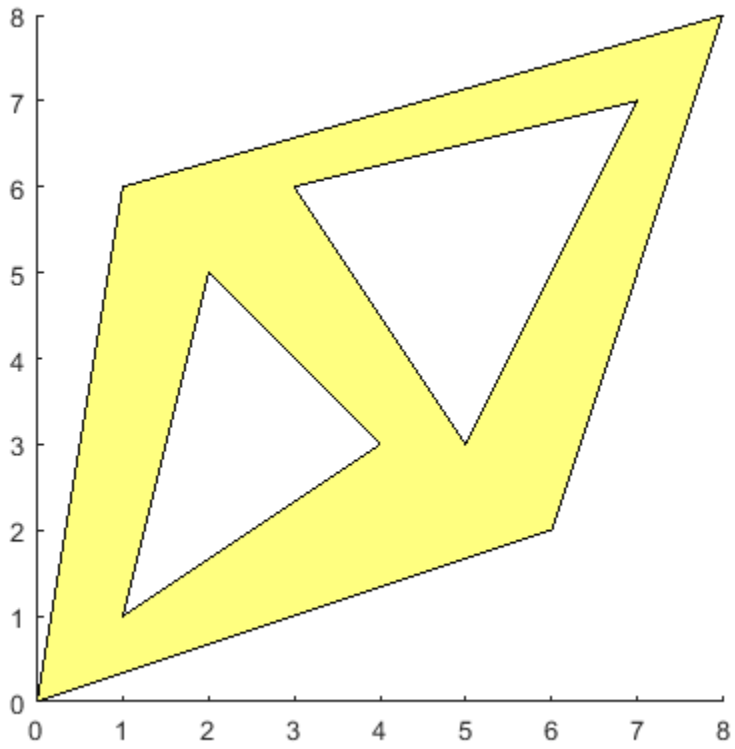
These vectors define a polygon with one external boundary and two internal boundaries. The boundaries are separated using `NaN` values. Verify the vertex order of the boundaries using the `ispolycw` function. The `ispolycw` function returns 1 when the vertices are in a clockwise order.

```
ispolycw(x2,y2)
```

```
ans = 1x3 logical array

   1   0   0
```

Display the polygon. The internal boundaries create holes within the polygon.

```
figure
mapshow(x2,y2,'DisplayType','polygon')
```



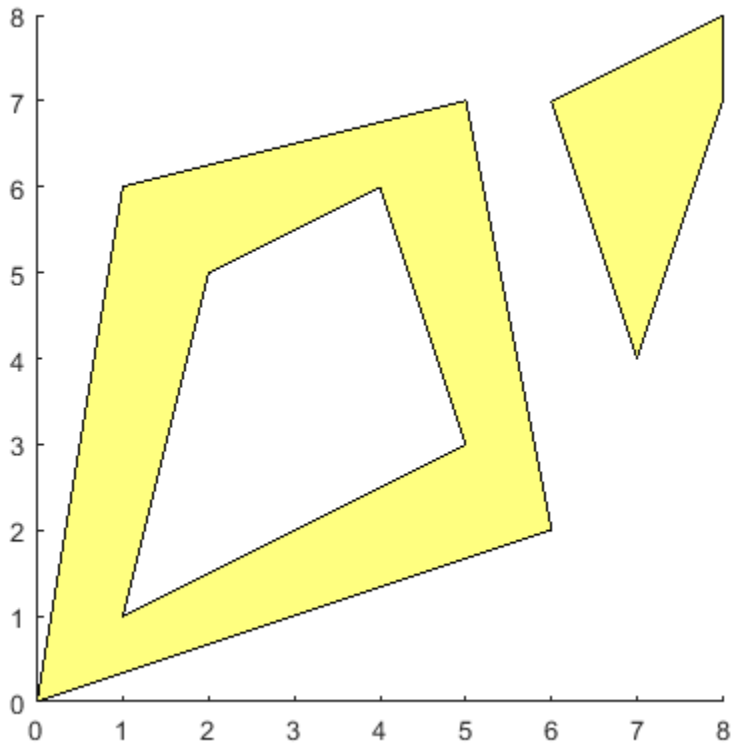
Now, list the vertices for a polygon with two nonintersecting regions. One of the regions has a hole. Verify the vertex order of the boundaries using `ispolycw`.

```
x3 = [0 1 5 6 0 NaN 1 5 4 2 1 NaN 7 6 8 8 7];
y3 = [0 6 7 2 0 NaN 1 3 6 5 1 NaN 4 7 8 7 4];
ispolycw(x3,y3)
```

```
ans = 1x3 logical array

   1   0   1
```

Display the polygon. The external boundaries create two nonintersecting regions and the internal boundary creates a hole.

```
figure
mapshow(x3,y3,'DisplayType','polygon')
```



Polygons Using Geographic Coordinates

In general, you can use geographic coordinates when you define polygons over small regions and call functions such as `ispolycw`. This is true except in cases where the polygon wraps a pole or crosses the Antimeridian.

For example, display the state of Michigan on a map using polygons with geographic coordinates. First, read the vertices of the state boundaries.

```
states = shaperead('usastatehi','UseGeoCoords',true);
michigan = states(22);
lat = michigan.Lat;
lon = michigan.Lon;
```

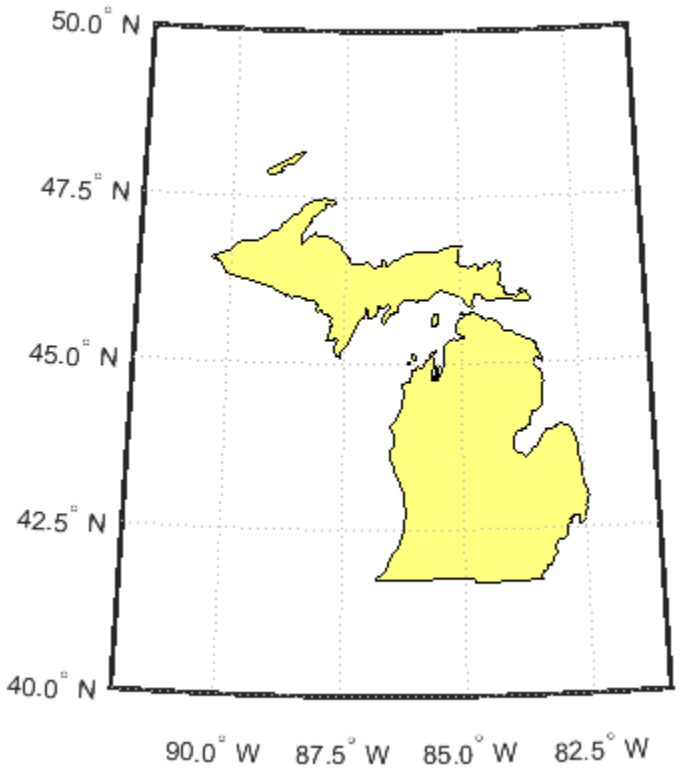
Count the boundaries and verify their vertex order. To use `ispolycw` with geographic coordinates, list the longitude vector as the first argument and the latitude vector as the second argument. The 1-by-6 output array means there are six boundaries. Each element of the array is 1, which means that each boundary is the exterior boundary of its own region.

```
ispolycw(lon,lat)
```

```
ans = 1x6 logical array

   1   1   1   1   1   1
```

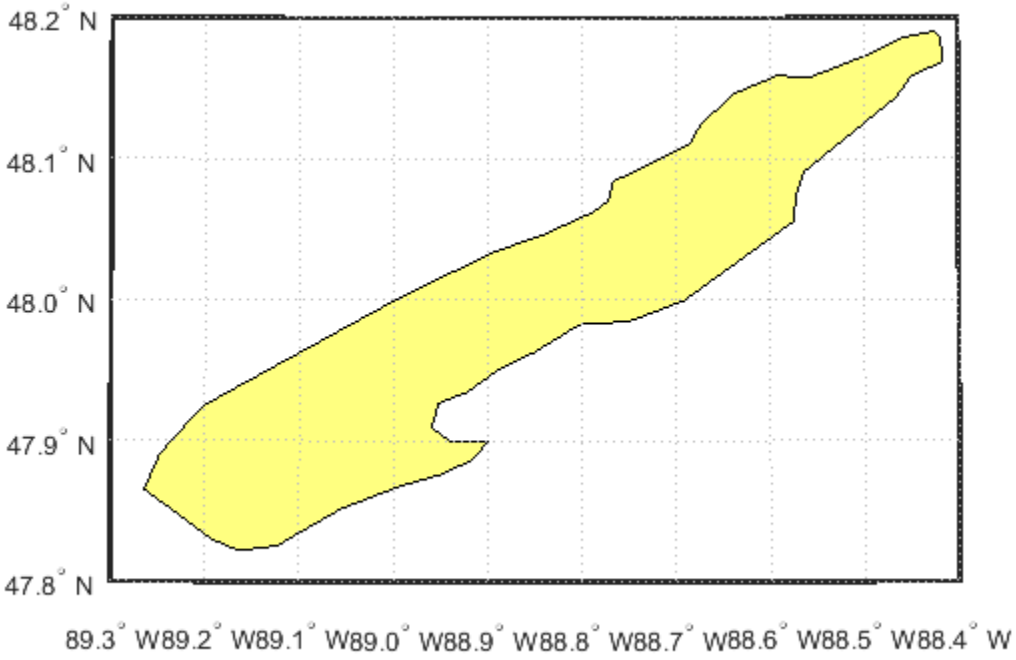
Display the polygon on a map using the `geoshow` function, specifying `'DisplayType'` as `'polygon'`.

```
usamap 'Michigan'
geoshow(lat,lon,'DisplayType','polygon')
```



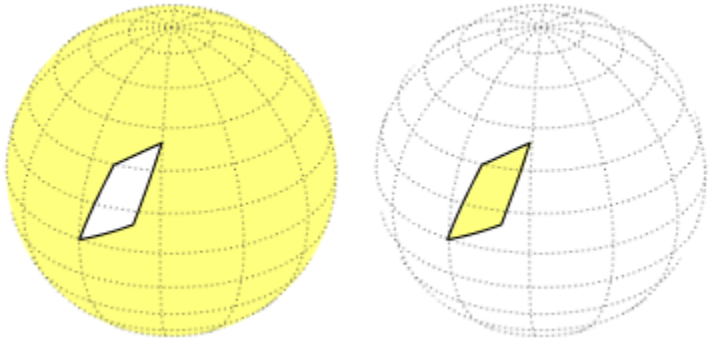
Clip the polygon to the latitude and longitude limits of Isle Royale National Park using the `maptrimp` function. Display the clipped polygon on a new map.

```
latlim = [47.8 48.2];
lonlim = [-89.3 -88.4];
[latT,lonT] = maptrimp(lat,lon,latlim,lonlim);

figure
usamap(latlim,lonlim)
geoshow(latT,lonT,'DisplayType','polygon')
```



Filled Region of Polygons Using Geographic Coordinates

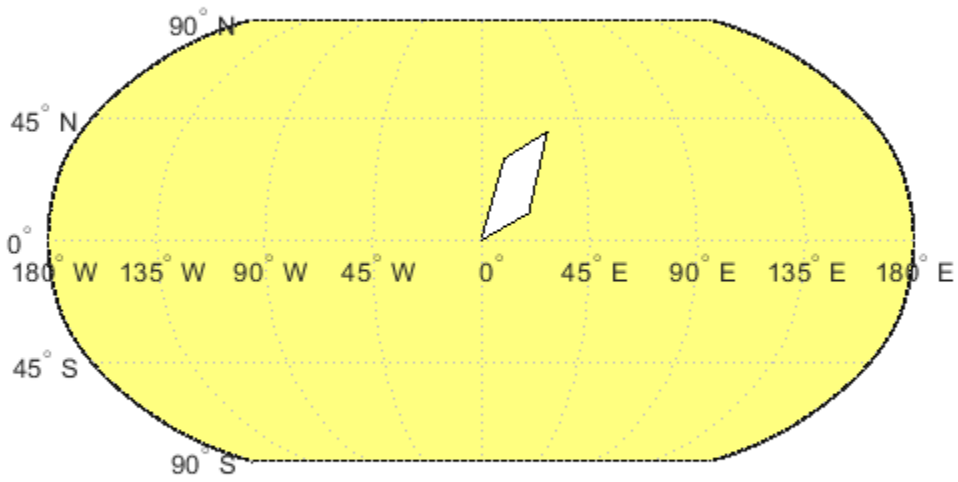
When you display a polygon on the Earth, the boundary divides the Earth into two regions. Both of these regions have finite area, so either could be the inside region of the polygon.



As a result, when you project the vertices of a polygon onto a map using the `geoshow` function, the filled region may be different than you expect. Change which region is filled by reversing the order of the vertices.

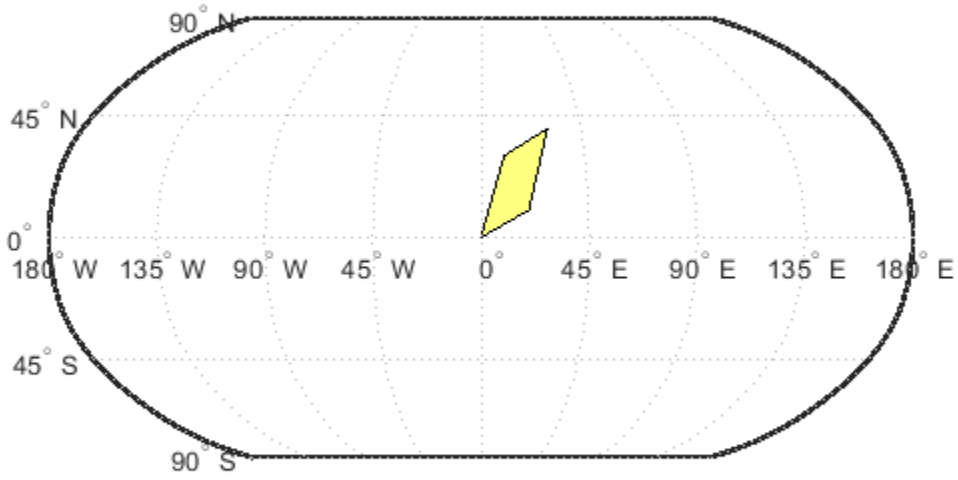
For example, display a small polygon on a world map.

```
lat2 = [0 10 40 30 0];
lon2 = [0 20 30 10 0];

figure
worldmap('world')
geoshow(lat2,lon2,'DisplayType','polygon')
```



The outside region of the polygon is filled. Reverse the order of the vertices by applying the `flip` function to the coordinate vectors. Then, display the polygon again.

```
lat2f = flip(lat2);
lon2f = flip(lon2);

figure
worldmap('world')
geoshow(lat2f,lon2f,'DisplayType','polygon')
```



The inside region of the polygon is filled instead.

See Also

`geoshow` | `ispolycw` | `mapshow` | `polyshape` | `worldmap`

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 that 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 “Link Line Segments with Common Endpoints into Polygons” on page 7-4.

See Also

`geoshow` | `polymerge`

More About

- “Create and Display Polygons” on page 2-12

Geographic Data Structures

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

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

Shape files

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

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

The Contents of Geographic Data Structures

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

Fields in a Geographic Data Structure

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

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

PolyLine and Polygon Shapes

In geographic data structures with `Line` or `Polygon` geometries, individual features can have multiple parts—disconnected line segments and polygon rings. The parts can include counterclockwise inner rings that outline "holes." For an illustration of this, see “Create and Display Polygons” on page 2-12. Each disconnected part is separated from the next by a NaN within the `X` and `Y` (or `Lat` and `Lon`) vectors. You can use the `isShapeMultipart` function to determine if a feature has NaN-separated parts.

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

Mapstructs and Geostructs

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

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

- If the shapefile includes a projection file (`.prj`), use `shapeinfo` to get information about the coordinate reference system. If the `CoordinateReferenceSystem` field of the returned structure is a `projcrs` object, you have map coordinates. If the field is a `geocrs` object, you have geographic coordinates.

- If the shapefile does not include a projection file, use `shapeinfo` to obtain the `BoundingBox`. By looking at the ranges of coordinates, you may be able to tell what kind of coordinates you have.
- Ask your data provider.

The `geoshow` function displays geographic features stored in geostructs, and the `mapshow` function displays geographic features stored in mapstructs. If you try to display a mapstruct with `geoshow`, the function issues a warning and calls `mapshow`. If you try to display a geostruct with `mapshow`, the function projects the coordinates with a Plate Carree projection and issues a warning.

Examining a Geographic Data Structure

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

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

The output appears as follows:

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

The shapefile contains 609 features. In addition to the `Geometry`, `BoundingBox`, and coordinate fields (`X` and `Y`), there are five attribute fields: `STREETNAME`, `RT_NUMBER`, `CLASS`, `ADMIN_TYPE`, and `LENGTH`.

Look at the 10th element:

```
S(10)
```

The output appears as follows:

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

This mapstruct contains `'Line'` features. The tenth line has nine vertices. The values of the first two attributes are character vectors. The second happens to be an empty character vector. The final three attributes are numeric. Across the elements of `S`, `X` and `Y` can have various lengths, but `STREETNAME` and `RT_NUMBER` must always contain character vectors, and `CLASS`, `ADMIN_TYPE` and `LENGTH` must always contain scalar doubles.

In this example, `shaperead` returns an unfiltered mapstruct. If you want to filter out some attributes, see “Select Shapefile Data to Read” on page 2-94 for more information.

How to Construct Geographic Data Structures

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

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

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

Making Point and Line Geostructs

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

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

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

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

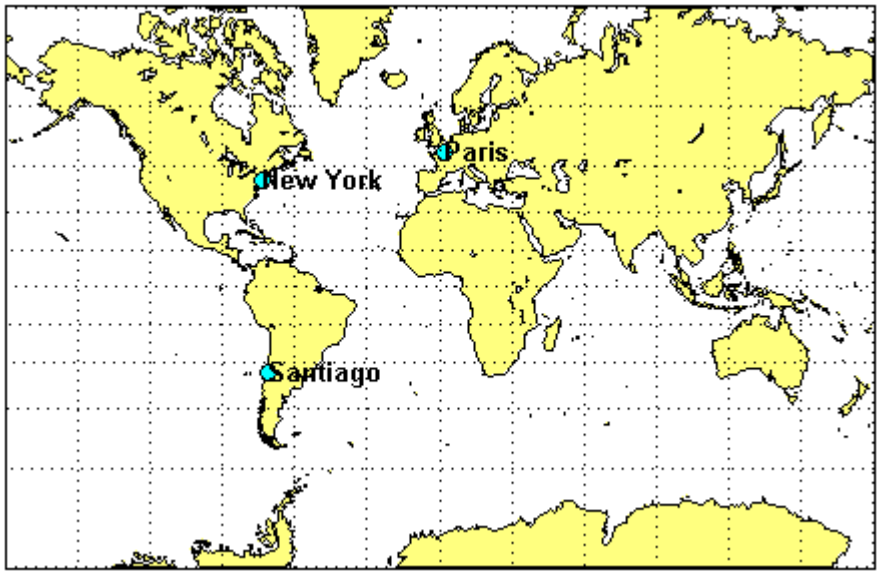
- `Geometry` (in this case `'Point'`)
- `Lat` (for points, this is a scalar double)
- `Lon` (for points, this is a scalar double)

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

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

% Add city names as City fields. You can name optional fields
% anything you like other than Geometry, Lat, Lon, X, or Y.
```

```
Cities(1).Name = 'Paris';
Cities(2).Name = 'Santiago';
Cities(3).Name = 'New York';
% Inspect your completed geostruct and its first member
Cities

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

Cities(1)

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

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

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

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

% Display the city names using data in the geostruct field Name.
% Note that you must treat the Name field as a cell array.
textm([Cities(:).Lat],[Cities(:).Lon],...
    {Cities(:).Name},'FontWeight','bold');
```



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

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

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

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

Tracks(2).Name = 'Santiago-New York';
[Tracks(2).Lat Tracks(2).Lon] = ...
        track2(trackType,latsant,lonsant,latnyc,lonnyc);

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

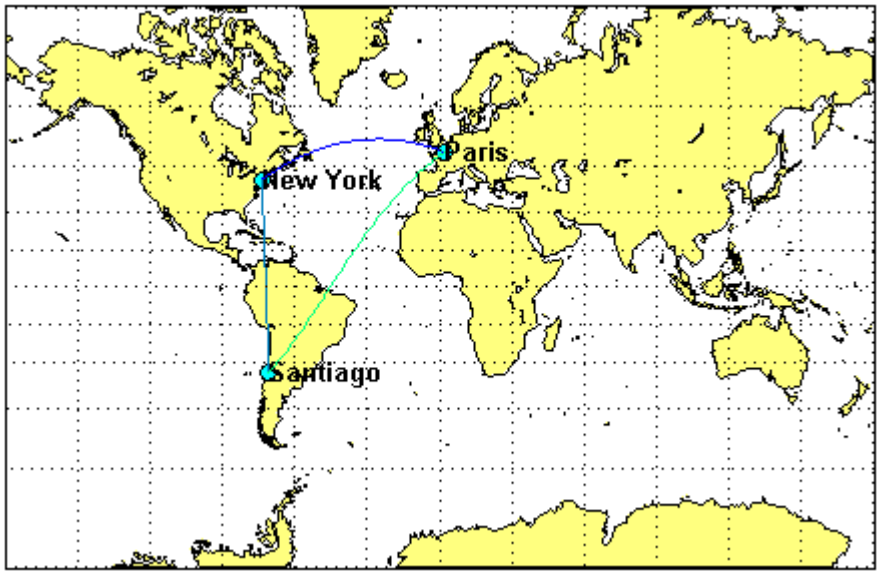
5 Compute lengths of the great circle tracks:

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

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

6 Map the three tracks in the line geostruct:

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

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

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



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

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

Making Polygon Geostructs

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

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

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

- `ispolycw` — True if vertices of polygonal contour are clockwise ordered

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

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

For an example of working with polygon geostructs, see “Converting Coastline Data (GSHHG) to Shapefile Format” on page 2-113.

Mapping Toolbox Version 1 Display Structures

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

See Also

`shapeinfo` | `shaperead`

More About

- “Create and Display Polygons” on page 2-12

Georeferenced Raster Data

Raster geodata consists of georeferenced data grids and images that are stored as matrices or objects in the MATLAB workspace. 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.

All regular data grids require a reference object, matrix, or vector, that specify the placement and resolution of the data set. Geolocated data grids do not require a separate reference object, as they explicitly identify the geographic coordinates of all rows and columns. For details on geolocated grids, see “Geolocated Data Grids” on page 2-35.

Reference Objects

A spatial reference object encapsulates the relationship between a geographic or planar coordinate system and a system of intrinsic coordinates anchored to the columns and rows of a 2-D spatially referenced raster grid or image. A reference object for raster data that is referenced to a geographic latitude-longitude system can be a `GeographicCellsReference` or `GeographicPostingsReference` object. A referencing object for raster data referenced to a planar (projected) map coordinate system can be a `MapCellsReference` or `MapPostingsReference` object. Unlike the older referencing matrix and vector representations (described below), a reference object is self-documenting, providing a rich set of properties to describe both the intrinsic and extrinsic geometry. The use of reference objects is preferred, but some referencing matrix and vector functionality continues to be supported for the purpose of compatibility.

Referencing Matrices

A referencing matrix is a 3-by-2 matrix of doubles that describes the scaling, orientation, and placement of the data grid on the globe. For a given referencing matrix, `R`, one of the following relations holds between rows and columns and coordinates (depending on whether the grid is based on map coordinates or geographic coordinates, respectively):

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

Convert a referencing matrix to a raster reference object using the `refmatToGeoRasterReference` or `refmatToMapRasterReference` functions.

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 may encounter 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]
```

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 object or

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 objects or matrices.

Convert a referencing vector to a geographic raster reference object using the `refvecToGeoRasterReference` function.

Construct 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.

1 Create a 3-by-6 grid:

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

2 Now make a referencing object:

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

Your output appears like this:

```
miniR =

  GeographicCellsReference with properties:

            LatitudeLimits: [-90 90]
           LongitudeLimits: [-180 180]
                RasterSize: [3 6]
      RasterInterpretation: 'cells'
          ColumnsStartFrom: 'south'
             RowsStartFrom: 'west'
       CellExtentInLatitude: 60
      CellExtentInLongitude: 60
     RasterExtentInLatitude: 180
    RasterExtentInLongitude: 360
           XIntrinsicLimits: [0.5 6.5]
           YIntrinsicLimits: [0.5 3.5]
       CoordinateSystemType: 'geographic'
                  AngleUnit: 'degree'
```

3 Set up an equidistant cylindrical map projection:

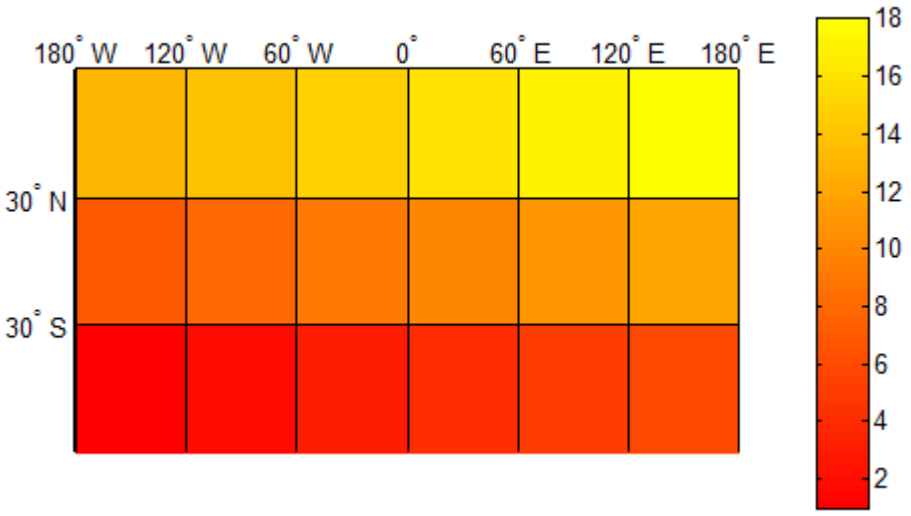
```
figure('Color','white')
ax = axesm('MapProjection', 'eqdcylin');
axis off
setm(ax,'GLineStyle','-', 'Grid','on','Frame','on')
```

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

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

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

```
geoshow(miniZ, miniR, 'DisplayType', 'texturemap');
colormap('autumn')
colorbar
```



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

Precompute the Size of a Data Grid

Before making a large, memory-taxing data grid, you should first determine what its size will be. If you know the latitude and longitude limits of a region, you can calculate the size of the raster by creating a referencing object, for any desired map resolution and scale.

Specify the latitude and longitude limits for the region. This example calculates the size of a map of the continental U.S. at a scale of 10 cells per degree.

```
latlim = [  25  50];
lonlim = [-130 -60];
```

Specify the extent of the data grid using cells per degree.

```
cellsPerDegree = 10;
extent = 1/cellsPerDegree;
```

Construct a referencing object and verify that the size of the raster is reasonable (in this case, 250 by 700 cells).

```
R = georefcells(latlim,lonlim,extent,extent);
R.RasterSize
```

```
ans = 1×2

   250   700
```

Geolocated Data Grids

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

Regular data grids cover 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.

Define Geolocated Data Grid

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, and a matrix giving cell-by-cell longitude coordinates.

Load a MAT-file containing an irregularly shaped geolocated data grid called `mapmtx`.

```
load mapmtx
```

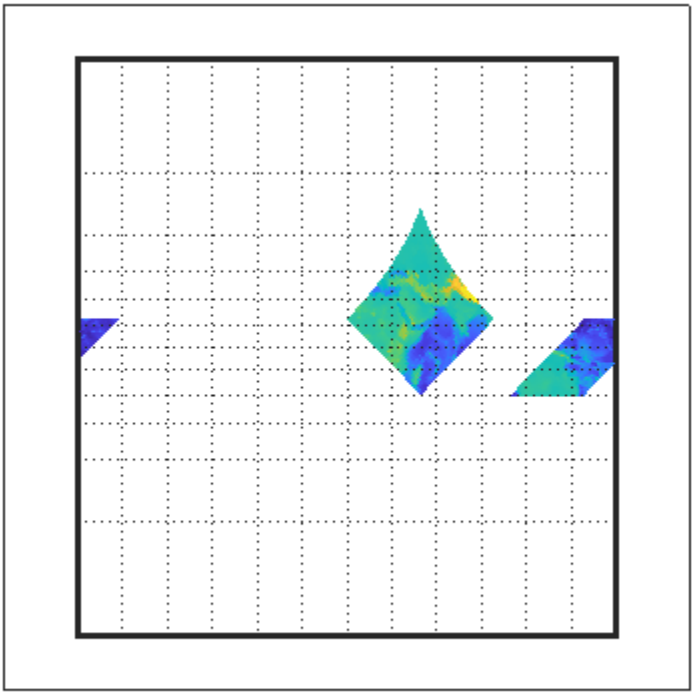
View the variables created from this MAT-file. Two geolocated data grids are in this data set, each requiring three variables. The values contained in `map1` correspond to the latitude and longitude coordinates, respectively, in `lt1` and `lg1`. Notice that all three matrices are the same size. Similarly, `map2`, `lt2`, and `lg2` together form a second geolocated data grid. These data sets were extracted from the `topo60c` data grid. Neither of these maps is regular, because their columns do not run north to south.

```
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
```

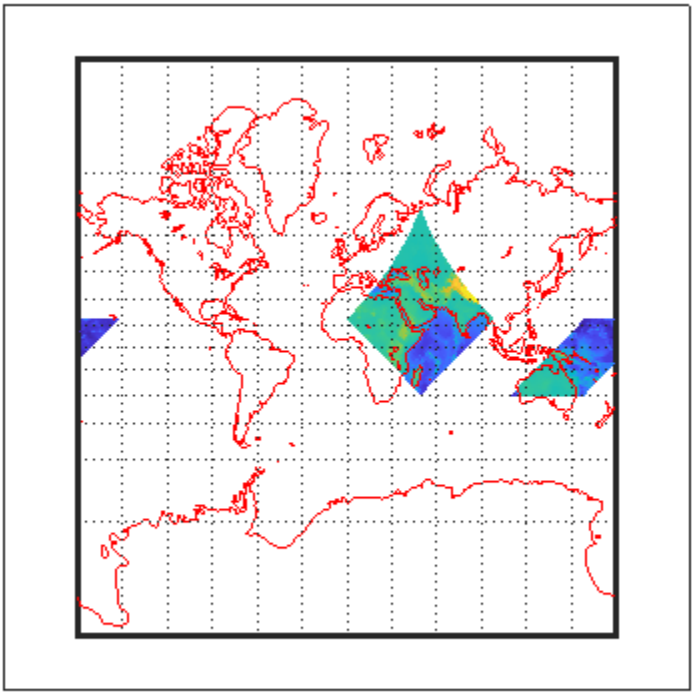
Display the grids one after another to see their geography.

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



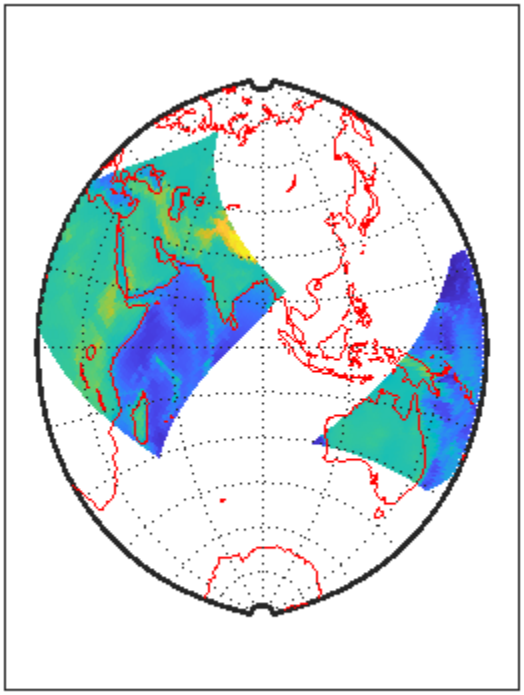
Showing coastlines will help to orient you to these skewed grids. Notice that neither grid is a regular rectangle. One looks like a diamond geographically, the other like a trapezoid. The trapezoid is displayed in two pieces because it crosses the edge of the map. These shapes can be thought of as the 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.

```
load coastlines
plotm(coastlat,coastlon,'r')
```



Now change the view to a polyconic projection with an origin at 0°N, 90°E. As the polyconic projection is limited to a 150° range in longitude, those portions of the maps outside this region are automatically trimmed.

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



Geographic Interpretations of Geolocated Grids

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

- First, a map matrix having the same number of rows and columns as the latitude and longitude coordinate matrices represents the values of the map data at the corresponding geographic points (centers of data cells).
- Next, a map matrix having one fewer row and one fewer column than the geographic coordinate matrices represents the values of the map data within the area formed by the four adjacent latitudes and longitudes.
- Finally, if the latitude and longitude matrices have smaller dimensions than the map matrix, you can interpret them as describing a coarser *graticule*, or mesh of latitude and longitude cells, into which the blocks of map data are warped.

This section discusses the first two interpretations of geolocated data grids. For more information on the use of graticules, see “The Map Grid” on page 4-83.

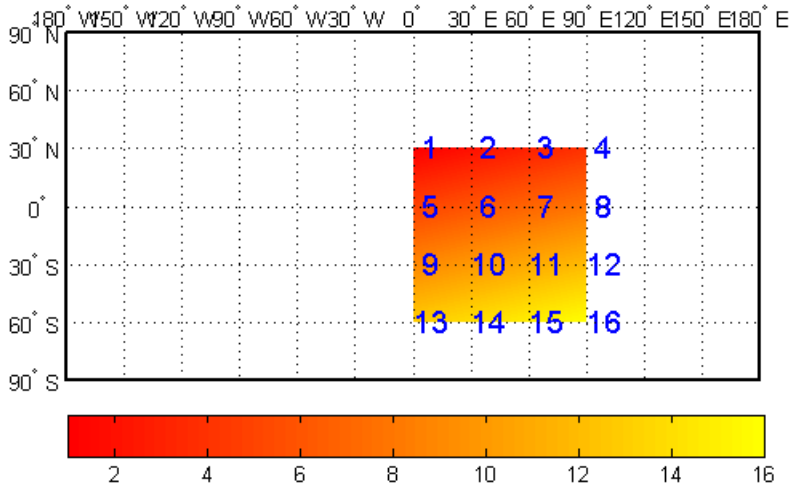
Type 1: Values Associated with the Upper Left Grid Coordinate

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:

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

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

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

Display the geolocated data grid with the values of `map` shown at the associated latitudes and longitudes:

```
figure('Color','white','Colormap',autumn(64))
axesm('pcarree','Grid','on','Frame','on',...
    'PLineLocation',30,'PLabelLocation',30)
mlabel; plabel; axis off; tightmap

h = geoshow(lat,lon,Z,'DisplayType','surface');
set(h,'ZData',zeros(size(Z)))
ht = textm(lat(:),lon(:),num2str(Z(:)), ...
    'Color','blue','FontSize',14);

colorbar('southoutside')
```



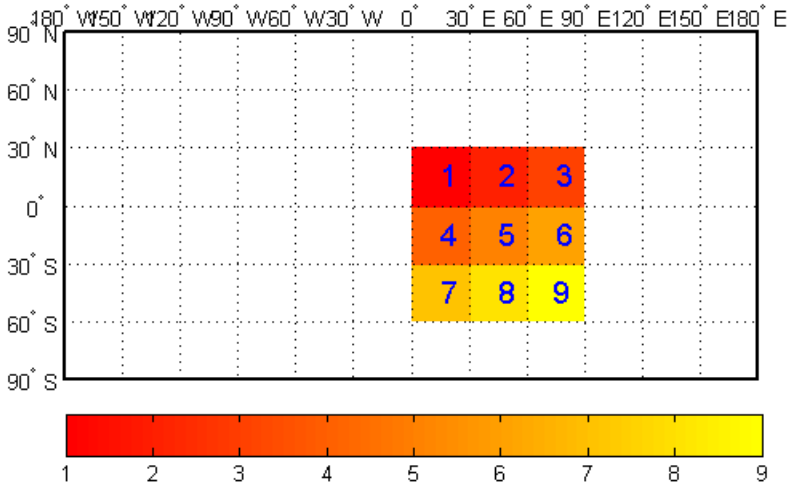
Notice that only 9 of the 16 total cells are displayed. The value displayed for each cell is the value at the upper left corner of that cell, whose coordinates are given by the corresponding `lat` and `lon` elements. By convention, the last row and column of the map matrix are not displayed, although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

Type 2: Values Centered Within Four Adjacent Coordinates

For the second interpretation, consider a 3-by-3 map matrix with the same `lat` and `lon` variables:

```
delete(h)
delete(ht)

Z3by3 = [ ...
 1  2  3; ...
 4  5  6; ...
 7  8  9];

h = geoshow(lat,lon,Z3by3,'DisplayType','texturemap');

tlat = [ ...
   15 15 15; ...
   -15 -15 -15; ...
   -45 -45 -45];

tlon = [ ...
    15 45 75; ...
    15 45 75; ...
    15 45 74];

textm(tlat(:),tlon(:),num2str(Z3by3(:)), ...
    'Color','blue','FontSize',14)
```

Display 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 `lon` matrices are ordered. In a geolocated data grid, the order of values in the two coordinate matrices determines the arrangement of the displayed values.

Transform 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. For example, load elevation raster data and a geographic cells reference object. Then, create a latitude-longitude grid for the data by calling the `meshgrat` function.

```
load topo60c
[lat,lon] = meshgrat(topo60c,topo60cR);
```

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.

Unprojecting a Digital Elevation Model (DEM)

This example shows how to convert a USGS DEM into a regular latitude-longitude grid having comparable spatial resolution. U.S. Geological Survey (USGS) 30-meter Digital Elevation Models (DEMs) are regular grids (raster data) that use the UTM coordinate system. Using such DEMs in applications may require reprojecting and resampling them. You can readily apply the approach show here to projected map coordinate systems other than UTM and to other DEMs and most types of regular data grids.

First, set the output display format to `longG` so that the output displays more decimal places. Get the current output format so that you can restore it at the end of the example.

```
currentFormat = get(0,'format');
format longG
```

Step 1: Import the DEM and its Metadata

This example uses a USGS DEM for a quadrangle 7.5-arc-minutes square located in the White Mountains of New Hampshire, USA. Import the data and a map cells reference object using the `readgeoraster` function. Get additional metadata using the `georasterinfo` function.

```
[Z,R] = readgeoraster('MtWashington-ft.grd','OutputType','double');
info = georasterinfo('MtWashington-ft.grd');
```

Replace missing data with `NaN` values.

```
m = info.MissingDataIndicator;
Z = standardizeMissing(Z,m);
```

Step 2: Get Projection Information

Get information about the projected coordinate reference system by querying the `ProjectedCRS` property of the reference object. The result is a `projcrs` object. Then, get the ellipsoid for the coordinate reference system.

```
p = R.ProjectedCRS;
ellipsoid = p.GeographicCRS.Spheroid
```

```
ellipsoid =

referenceEllipsoid with defining properties:

                Code: 7008
                Name: 'Clarke 1866'
          LengthUnit: 'meter'
        SemimajorAxis: 6378206.4
        SemiminorAxis: 6356583.8
    InverseFlattening: 294.978698213898
         Eccentricity: 0.0822718542230038

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
```

```
    Volume
```

Step 3: Determine which UTM Zone to Use and Construct a Map Axes

From the `Name` property of the `projcrs` object, you can tell that the DEM is gridded in a Universal Transverse Mercator (UTM) coordinate system.

```
p.Name
```

```
ans =

    "UTM Zone 19, Northern Hemisphere"
```

To find the UTM zone, first locate the center of the DEM in UTM coordinates. Then, convert the coordinates to latitude-longitude.

```
[M,N] = size(Z);
xCenterIntrinsic = (1 + N)/2;
yCenterIntrinsic = (1 + M)/2;
[xCenter,yCenter] = intrinsicToWorld(R,xCenterIntrinsic,yCenterIntrinsic);
[latCenter,lonCenter] = projinv(p,xCenter,yCenter)
```

```
latCenter =

          44.3124367104673


lonCenter =

         -71.3126432693478
```

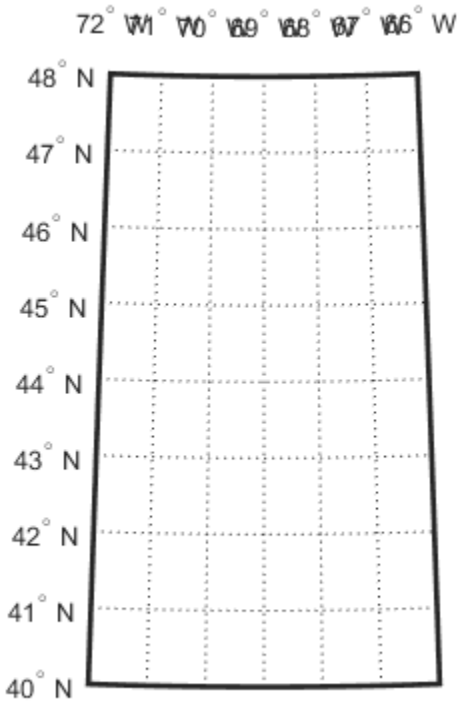
Find the UTM zone for the DEM by using the `utmzone` function.

```
utmZone = utmzone(latCenter,lonCenter)
```

```
utmZone =

    '19T'
```

Use the zone and ellipsoid to create a map axes.

```
figure
axesm('utm','zone',utmZone,'geoid',ellipsoid)
axis off
gridm
mlabel on
plabel on
framem on
```



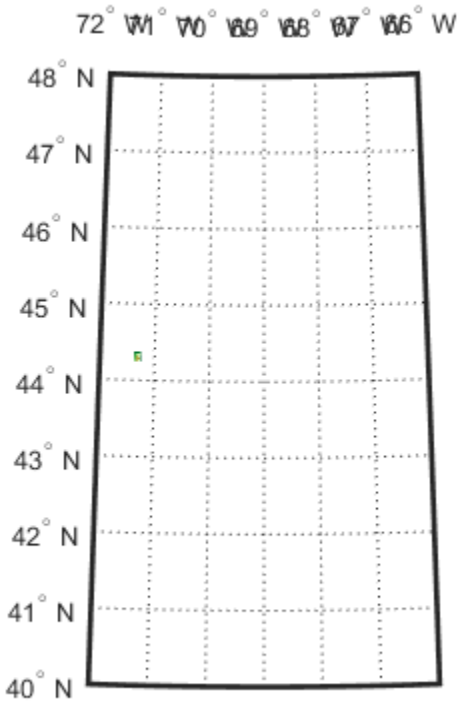
Note: if you can visually place the approximate location of New Hampshire on a world map, then you can confirm this result with the `utmzoneui` GUI.

```
utmzoneui(actualZone)
```

Step 4: Display the Original DEM on the Map Axes

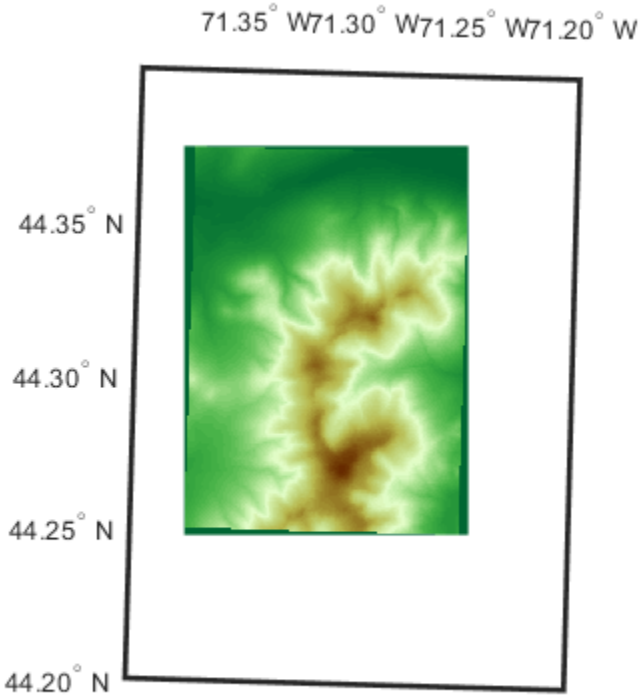
Use `mapshow` (rather than `geoshow` or `meshm`) to display the DEM on the map axes because the data are gridded in map (x-y) coordinates.

```
mapshow(Z,R,'DisplayType','texturemap')
demcmap(Z)
```



The DEM covers such a small part of this map that it may be hard to see (look between 44 and 44 degrees North and 72 and 71 degrees West), because the map limits are set to cover the entire UTM zone. You can reset them (as well as the map grid and label parameters) to get a closer look.

```
setm(gca,'MapLatLimit',[44.2 44.4],'MapLonLimit',[-71.4 -71.2])
setm(gca,'MLabelLocation',0.05,'MLabelRound',-2)
setm(gca,'PLabelLocation',0.05,'PLabelRound',-2)
setm(gca,'PLineLocation',0.025,'MLineLocation',0.025)
```



When it is viewed at this larger scale, narrow wedge-shaped areas of uniform color appear along the edge of the grid. These are places where `Z` contains the value NaN, which indicates the absence of actual data. By default they receive the first color in the color table, which in this case is dark green. These null-data areas arise because although the DEM is gridded in UTM coordinates, its data limits are defined by a latitude-longitude quadrangle. The narrow angle of each wedge corresponds to the non-zero "grid declination" of the UTM coordinate system in this part of the zone. (Lines of constant x run precisely north-south only along the central meridian of the zone. Elsewhere, they follow a slight angle relative to the local meridians.)

Step 5: De ne the Output Latitude-Longitude Grid

The next step is to define a regularly-spaced set of grid points in latitude-longitude that covers the area within the DEM at about the same spatial resolution as the original data set.

First, you need to determine how the latitude changes between rows in the input DEM (i.e., by moving northward by 30 meters).

```
rng = R.CellExtentInWorldY;  % In meters, consistent with p.LengthUnit
latcrad = deg2rad(latCenter);   % latCenter in radians

% Change in latitude, in degrees
dLat = rad2deg(meridianfwd(latcrad,rng,ellipsoid) - latcrad)
```

```
dLat =
```

```
      0.000269984939366415
```

The actual spacing can be rounded slightly to define the grid spacing to be used for the output (latitude-longitude) grid.

```
gridSpacing = 1/4000;   % In other words, 4000 samples per degree
```

To set the extent of the output (latitude-longitude) grid, start by finding the corners of the DEM in UTM map coordinates.

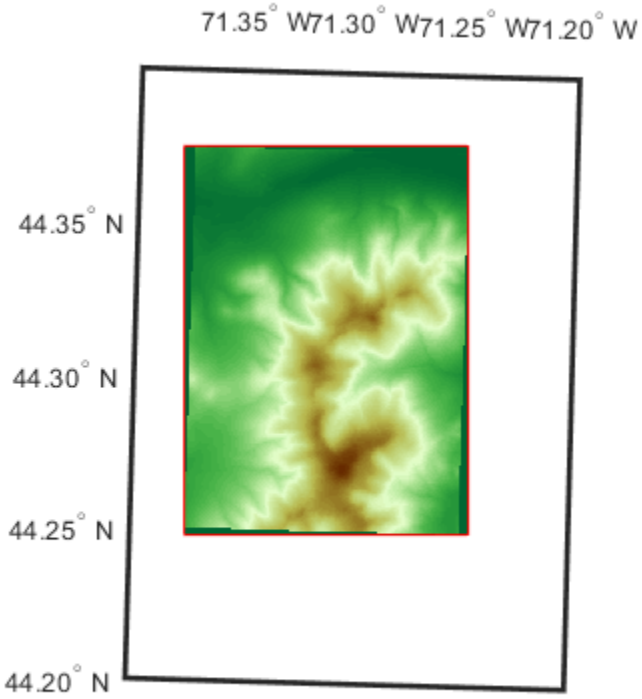
```
xCorners = R.XWorldLimits([1 1 2 2])'
yCorners = R.YWorldLimits([1 2 2 1])'
```

```
xCorners =

      310380
      310380
      320730
      320730


yCorners =

     4901880
     4916040
     4916040
     4901880
```

Then convert the corners to latitude-longitude. Display the latitude-longitude corners on the map (via the UTM projection) to check that the results are reasonable.

```
[latCorners, lonCorners] = projinv(p,xCorners, yCorners)
hCorners = geoshow(latCorners,lonCorners,'DisplayType','polygon',...
    'FaceColor','none','EdgeColor','red');
```

```
latCorners =

          44.2474175605687
          44.3747915486804
          44.3774240601986
          44.2500384686392


lonCorners =

         -71.3749065609587
         -71.3800513603087
         -71.2502438233865
         -71.2453790282992
```



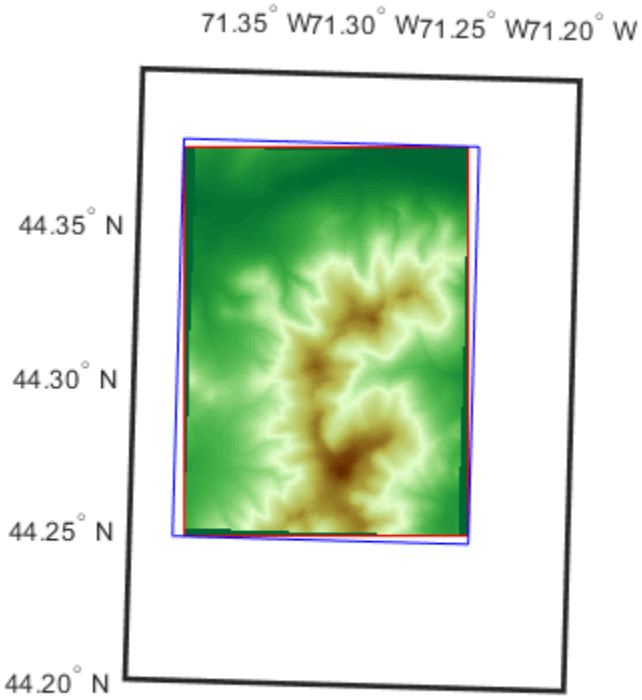
Next, round outward to define an output latitude-longitude quadrangle that fully encloses the DEM and aligns with multiples of the grid spacing.

```
latSouth = gridSpacing * floor(min(latCorners)/gridSpacing)
lonWest  = gridSpacing * floor(min(lonCorners)/gridSpacing)
latNorth = gridSpacing * ceil( max(latCorners)/gridSpacing)
lonEast  = gridSpacing * ceil( max(lonCorners)/gridSpacing)

qlatlim = [latSouth latNorth];
qlonlim = [lonWest lonEast];

dlat = 100*gridSpacing;
dlon = 100*gridSpacing;

[latquad, lonquad] = outlinegeoquad(qlatlim, qlonlim, dlat, dlon);

hquad = geoshow(latquad, lonquad, ...
    'DisplayType','polygon','FaceColor','none','EdgeColor','blue');

snapnow;
```

```
latSouth =

                44.24725
```

```
lonWest =
```

```
                -71.38025

latNorth =

                 44.3775

lonEast =

                -71.24525
```



Finally, construct a geographic raster referencing object for the output grid. It supports transformations between latitude-longitude and the row and column subscripts. In this case, use of a world file matrix, W, enables exact specification of the grid spacing.

```
W = [gridSpacing    0              lonWest + gridSpacing/2; ...
     0              gridSpacing    latSouth + gridSpacing/2]
```

```
W =

                0.00025                     0              -71.380125
                      0               0.00025               44.247375
```

```
nRows = round((latNorth - latSouth) / gridSpacing)
nCols = round(wrapTo360(lonEast - lonWest) / gridSpacing)
```

```
nRows =

   521


nCols =

   540
```

```
Rlatlon = georasterref(W,[nRows nCols],'cells');
Rlatlon.GeographicCRS = p.GeographicCRS
```

```
Rlatlon =

  GeographicCellsReference with properties:

            LatitudeLimits: [44.24725 44.3775]
           LongitudeLimits: [-71.38025 -71.24525]
                RasterSize: [521 540]
      RasterInterpretation: 'cells'
          ColumnsStartFrom: 'south'
             RowsStartFrom: 'west'
      CellExtentInLatitude: 1/4000
     CellExtentInLongitude: 1/4000
    RasterExtentInLatitude: 0.13025
   RasterExtentInLongitude: 0.135
         XIntrinsicLimits: [0.5 540.5]
         YIntrinsicLimits: [0.5 521.5]
     CoordinateSystemType: 'geographic'
            GeographicCRS: [1x1 geocrs]
                AngleUnit: 'degree'
```

`Rlatlon` fully defines the number and location of each cell in the output grid.

Step 6: Map Each Output Grid Point Location to UTM X-Y

Finally, you're ready to make use of the map projection, applying it to the location of each point in the output grid. First compute the latitudes and longitudes of those points, stored in 2-D arrays.

```
[rows,cols] = ndgrid(1:nRows, 1:nCols);
[lat,lon] = intrinsicToGeographic(Rlatlon,cols,rows);
```

Then apply the projection to each latitude-longitude pair, arrays of UTM x-y locations having the same shape and size as the latitude-longitude arrays.

```
[XI,YI] = projfwd(p,lat,lon);
```

At this point, `XI(i,j)` and `YI(i,j)` specify the UTM coordinate of the grid point corresponding to the i-th row and j-th column of the output grid.

Step 7: Resample the Original DEM

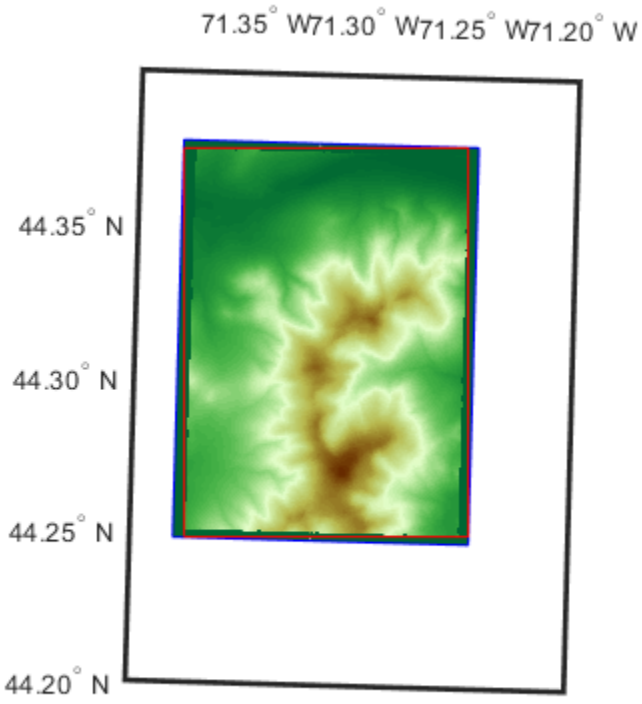
The final step is to use the MATLAB `interp2` function to perform bilinear resampling.

At this stage, the value of projecting from the latitude-longitude grid into the UTM map coordinate system becomes evident: it means that the resampling can take place in the regular X-Y grid, making `interp2` applicable. The reverse approach, unprojecting each (X,Y) point into latitude-longitude, might seem more intuitive but it would result in an irregular array of points to be interpolated -- a much harder task, requiring use of the far more costly `griddata` function or some rough equivalent.

```
[rows,cols] = ndgrid(1:M,1:N);
[X,Y] = intrinsicToWorld(R,cols,rows);
method = 'bilinear';
extrapval = NaN;
Zlatlon = interp2(X,Y,Z,XI,YI,method,extrapval);
```

View the result in the projected axes using `geoshow`, which will re-project it on the fly. Notice that it fills the blue rectangle, which is aligned with lines of latitude and longitude. (In contrast, the red rectangle, which outlines the original DEM, aligns with UTM x and y.) Also notice NaN-filled regions along the edges of the grid. The boundaries of these regions appear slightly jagged, at the level of a single grid spacing, due to round-off effects during interpolation. Move the red quadrilateral and blue quadrangle to the top, to ensure that they are not hidden by the raster display.

```
geoshow(Zlatlon,Rlatlon,'DisplayType','texturemap')
uistack([hCorners hquad],'top')
```



Restore the original output display format.

```
format(currentFormat)
```

Credits

MtWashington-ft.grd (and supporting files):

```
  United States Geological Survey (USGS) 7.5-minute Digital Elevation
  Model (DEM) for the Mt. Washington quadrangle, with elevation in
  meters. http://edc.usgs.gov/products/elevation/dem.html

  For more information, run:

  >> type MtWashington-ft.txt
```

See Also

`demcmap` | `georasterref` | `intrinsicToGeographic` | `intrinsicToWorld` | `refmatToMapRasterReference`

Creating a Half-Resolution Georeferenced Image

This example shows how to create a half-resolution version of a georeferenced TIFF image, using referencing objects and Image Processing Toolbox™ functions `ind2gray` and `imresize`.

Step 1: Import a Georeferenced TIFF Image

Read an indexed-color TIFF image and convert it to grayscale. The size of the image is 2000-by-2000.

```
[X, cmap] = imread('concord_ortho_w.tif');
I_orig = ind2gray(X, cmap);
```

Read the corresponding world file. Each image pixel covers a one-meter square on the map.

```
R_orig = worldfileread('concord_ortho_w.tfw','planar',size(X));
```

Choose a convenient format for displaying the result.

```
currentFormat = get(0,'format');
format short g
R_orig
```

```
R_orig =
  MapCellsReference with properties:

           XWorldLimits: [207000 209000]
           YWorldLimits: [911000 913000]
             RasterSize: [2000 2000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1
      CellExtentInWorldY: 1
    RasterExtentInWorldX: 2000
    RasterExtentInWorldY: 2000
        XIntrinsicLimits: [0.5 2000.5]
        YIntrinsicLimits: [0.5 2000.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
           ProjectedCRS: []
```

Step 2: Resize the Image to Half Its Original Size

Halve the resolution, creating a smaller (1000-by-1000) image.

```
I_half = imresize(I_orig, size(I_orig)/2, 'bicubic');
```

Step 3: Construct a Referencing Object for the Resized Image

The resized image has the same limits as the original, just a smaller size, so copy the referencing object and reset its `RasterSize` property.

```
R_half = R_orig;
R_half.RasterSize = size(I_half)
```

```
R_half =
  MapCellsReference with properties:
```

```
          XWorldLimits: [207000 209000]
          YWorldLimits: [911000 913000]
            RasterSize: [1000 1000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 2
      CellExtentInWorldY: 2
    RasterExtentInWorldX: 2000
    RasterExtentInWorldY: 2000
        XIntrinsicLimits: [0.5 1000.5]
        YIntrinsicLimits: [0.5 1000.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
          ProjectedCRS: []
```

Step 4: Visualize the Results

Display each image in map coordinates, and mark a reference point with a red + in both figures.

```
xlimits = [208000 208250];
ylimits = [911800 911950];

x = 208202.21;
y = 911862.70;

figure
mapshow(I_orig,R_orig)
hold on
plot(x,y,'r+')
xlim(xlimits)
ylim(ylimits)
ax = gca;
ax.TickDir = 'out';
```

```
figure
mapshow(I_half,R_half)
hold on
plot(x,y,'r+')
xlim(xlimits)
ylim(ylimits)
ax = gca;
ax.TickDir = 'out';
```

Graphically, they coincide, even though the same map location corresponds to two different locations in intrinsic coordinates.

```
[xIntrinsic1, yIntrinsic1] = worldToIntrinsic(R_orig, x, y)
```

```
xIntrinsic1 =
       1202.7


yIntrinsic1 =
       1137.8
```

```
[xIntrinsic2, yIntrinsic2] = worldToIntrinsic(R_half, x, y)
```

```
xIntrinsic2 =
        601.6


yIntrinsic2 =
       569.15
```

```
format(currentFormat);
```

Data Set Information

The `concord_ortho_w.tif` and `concord_ortho_w.tfw` files are derived using orthophoto tiles from the Bureau of Geographic Information (MassGIS), Commonwealth of Massachusetts, Executive

Office of Technology and Security Services. For more information about the data sets, use the command `type concord_ortho.txt`. For an updated link to the data provided by MassGIS, see https://www.mass.gov/service-details/massgis-data-layers.

See Also

`MapCellsReference` | `imread` | `imresize` | `worldToIntrinsic` | `worldfileread`

Georeferencing an Image to an Orthotile Base Layer

This example shows how to register an image to an earth coordinate system and create a new "georeferenced" image. It requires Image Processing Toolbox™ in addition to Mapping Toolbox™.

In this example, all georeferenced data are in the same earth coordinate system, the Massachusetts State Plane system (using the North American Datum of 1983 in units of meters). This defines our "map coordinates." The georeferenced data include an orthoimage base layer and a vector road layer.

The data set to be georeferenced is a digital aerial photograph covering parts of the village of West Concord, Massachusetts, collected in early spring, 1997.

Step 1: Render Orthoimage Base Tiles in Map Coordinates

The orthoimage base layer is structured into 4000-by-4000 pixel tiles, with each pixel covering exactly one square meter in map coordinates. Each tile is stored as a TIFF image with a world file. The aerial photograph of West Concord overlaps two tiles in the orthoimage base layer. (For convenience, this example actually works with two 2000-by-2000 sub-tiles extracted from the larger 4000-by-4000 originals. They have the same pixel size, but cover only the area of interest.)

Read the two orthophoto base tiles required to cover the extent of the aerial image.

```
[baseImage1,cmap1] = imread('concord_ortho_w.tif');
[baseImage2,cmap2] = imread('concord_ortho_e.tif');
```

Read the world files for the two tiles

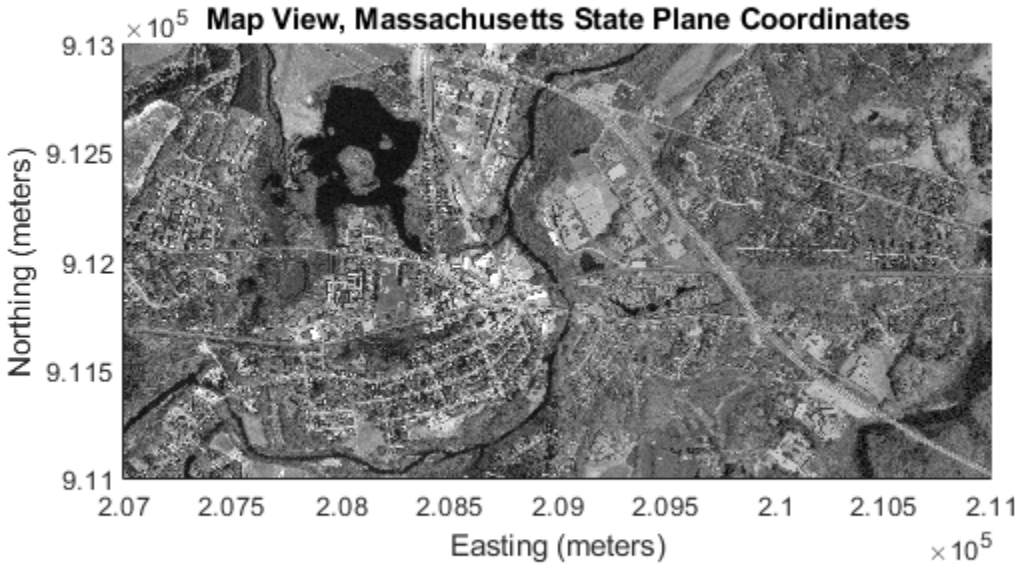
```
currentFormat = get(0,'format');
format short g
R1 = worldfileread('concord_ortho_w.tfw','planar',size(baseImage1))
R2 = worldfileread('concord_ortho_e.tfw','planar',size(baseImage2))
```

```
R1 =

  MapCellsReference with properties:

           XWorldLimits: [207000 209000]
           YWorldLimits: [911000 913000]
             RasterSize: [2000 2000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1
      CellExtentInWorldY: 1
    RasterExtentInWorldX: 2000
    RasterExtentInWorldY: 2000
        XIntrinsicLimits: [0.5 2000.5]
        YIntrinsicLimits: [0.5 2000.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
           ProjectedCRS: []



R2 =
```

```
  MapCellsReference with properties:

            XWorldLimits: [209000 211000]
            YWorldLimits: [911000 913000]
              RasterSize: [2000 2000]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1
      CellExtentInWorldY: 1
    RasterExtentInWorldX: 2000
    RasterExtentInWorldY: 2000
        XIntrinsicLimits: [0.5 2000.5]
        YIntrinsicLimits: [0.5 2000.5]
      TransformationType: 'rectilinear'
    CoordinateSystemType: 'planar'
            ProjectedCRS: []
```

Display the images in their correct spatial positions.

```
mapshow(baseImage1,cmap1,R1)
ax1 = gca;
mapshow(ax1,baseImage2,cmap2,R2)
title('Map View, Massachusetts State Plane Coordinates');
xlabel('Easting (meters)');
ylabel('Northing (meters)');
```



Step 2: Register Aerial Photograph to Map Coordinates

This step uses functions `cpselect`, `cpstruct2pairs`, `fitgeotrans`, and `imwarp`, and method `projective2d/transformPointsForward`, from the Image Processing Toolbox together with map raster reference objects from Mapping Toolbox. Together, they enable georegistration based on control point pairs that relate the aerial photograph to the orthoimage base layer.

Read the aerial photo.

```
inputImage = imread('concord_aerial_sw.jpg');
figure
imshow(inputImage)
title('Unregistered Aerial Photograph')
```

Both orthophoto images are indexed images but `cpselect` only takes grayscale images, so convert them to grayscale.

```
baseGray1 = im2uint8(ind2gray(baseImage1,cmap1));
baseGray2 = im2uint8(ind2gray(baseImage2,cmap2));
```

Downsample the images by a factor of 2, then pick two separate sets of control point pairs: one for points in the aerial image that appear in the first tile, and another for points that appear in the second tile. Save the control point pairs to the base workspace as control point structures named cpstruct1 and cpstruct2.

```
n = 2; % downsample by a factor n
load mapexreg.mat % load some points that were already picked
```

Optionally, edit or add to the pre-picked points using `cpselect`. Note that you need to work separately on the control points for each orthotile.

```
cpselect(inputImage(1:n:end,1:n:end,1),...
         baseGray1(1:n:end,1:n:end),cpstruct1);

cpselect(inputImage(1:n:end,1:n:end,1),...
         baseGray2(1:n:end,1:n:end),cpstruct2);
```

This tool helps you pick pairs of corresponding control points. Control points are landmarks that you can find in both images, like a road intersection, or a natural feature. Three pairs of control points have already been picked for each orthophoto tile. If you want to proceed with these points, go to Step 3: Infer and apply geometric transformation. If you want to add some additional pairs of points, do so by identifying landmarks and clicking on the images. Save control points by choosing the **File** menu, then the **Save Points to Workspace** option. Save the points, overwriting variables `cpstruct1` and `cpstruct2`.

Step 3: Infer and Apply Geometric Transformation

Extract control point pairs from the control point structures.

```
[input1,base1] = cpstruct2pairs(cpstruct1);
[input2,base2] = cpstruct2pairs(cpstruct2);
```

Account for downsampling by factor n.

```
input1 = n*input1 - 1;
input2 = n*input2 - 1;
base1 = n*base1 - 1;
base2 = n*base2 - 1;
```

Transform base image coordinates into map (State Plane) coordinates.

```
[x1, y1] = intrinsicToWorld(R1, base1(:,1), base1(:,2));
[x2, y2] = intrinsicToWorld(R2, base2(:,1), base2(:,2));
```

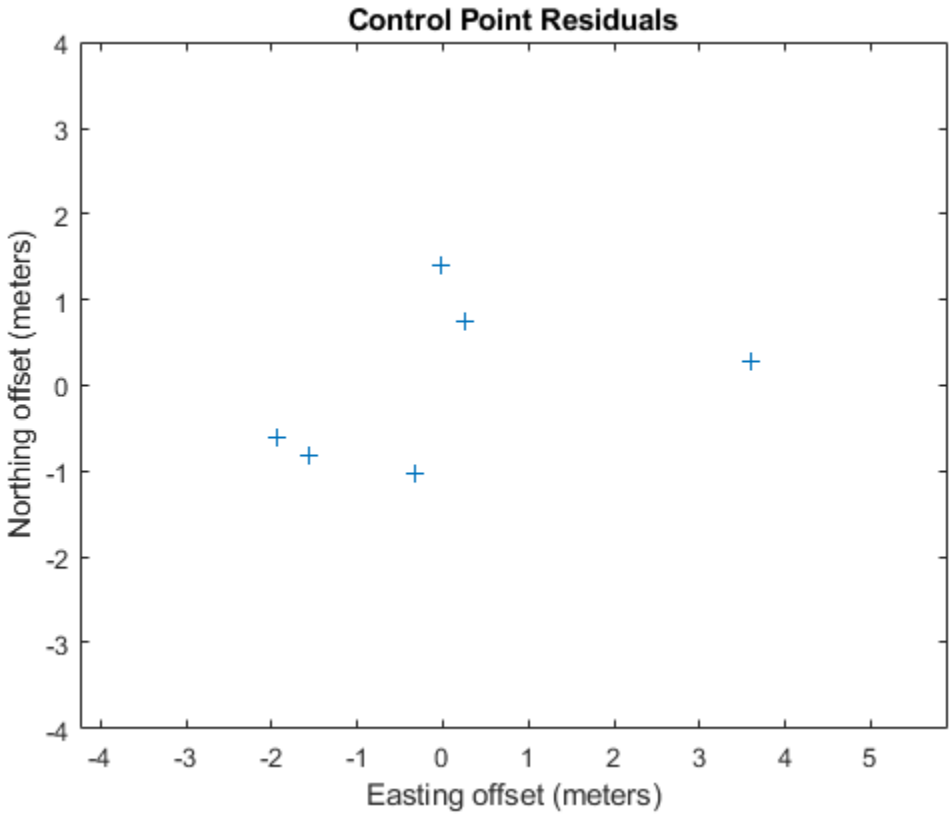
Combine the two sets of control points and infer a projective transformation. (The projective transformation should be a reasonable choice, since the aerial image is from a frame camera and the terrain in this area is fairly gentle.)

```
input = [input1; input2];
spatial = [x1 y1; x2 y2];

tform = fitgeotrans(input,spatial,'projective')
```

```
tform =

  projective2d with properties:

                T: [3x3 double]
    Dimensionality: 2
```

Compute and plot (2D) residuals.

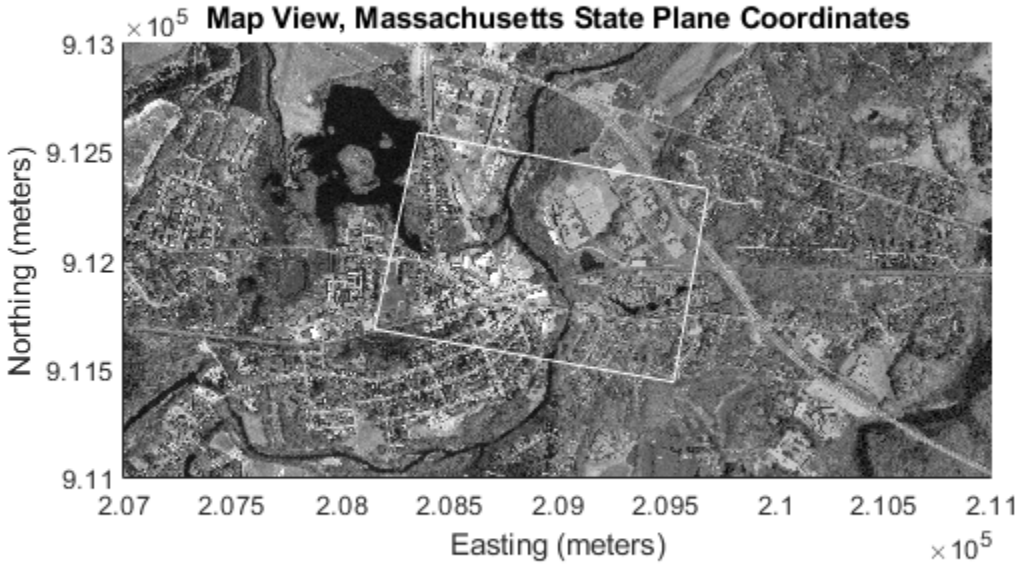
```
residuals = transformPointsForward(tform, input) - spatial;
figure
plot(residuals(:,1),residuals(:,2),'+')
title('Control Point Residuals');
xlabel('Easting offset (meters)');
ylabel('Northing offset (meters)');
xlim([-4 4])
ylim([-4 4])
axis equal
```



Predict corner locations for the registered image, in map coordinates, and connect them to show the image outline.

```
mInput = size(inputImage,1);
nInput = size(inputImage,2);

inputCorners = 0.5 ...
    + [0        0;
       0        mInput;
       nInput   mInput;
       nInput   0;
       0        0];

outputCornersSpatial = transformPointsForward(tform, inputCorners);

outputCornersX = outputCornersSpatial(:,1);
outputCornersY = outputCornersSpatial(:,2);
```

```
figure(ax1.Parent)
line(outputCornersX,outputCornersY,'Color','w')
```



Calculate the average pixel size of the input image (in map units).

```
pixelSize = [hypot( ...
    outputCornersX(2) - outputCornersX(1), ...
    outputCornersY(2) - outputCornersY(1)) / mInput, ...
 hypot( ...
    outputCornersX(4) - outputCornersX(5), ...
    outputCornersY(4) - outputCornersY(5)) / nInput]
```

```
pixelSize =

      0.90963      0.89054
```

Variable `pixelSize` gives a starting point from which to select a width and height for the pixels in our georegistered output image. In principle we could select any size at all for our output pixels. However, if we make them too small we waste memory and computation time, ending up with a big (many rows and columns) blurry image. If we make them too big, we risk aliasing the image as well as needlessly discarding information in the original image. In general we also want our pixels to be square. A reasonable heuristic is to select a pixel size that is slightly larger than `max(pixelSize)` and is a "reasonable" number (i.e., 0.95 or 1.0 rather than 0.9096). Here we chose 1, which means that each pixel in our georegistered image will cover one square meter on the ground. It's "nice" to have georegistered images that align along integer map coordinates for display and analysis.

```
outputPixelSize = 1;
```

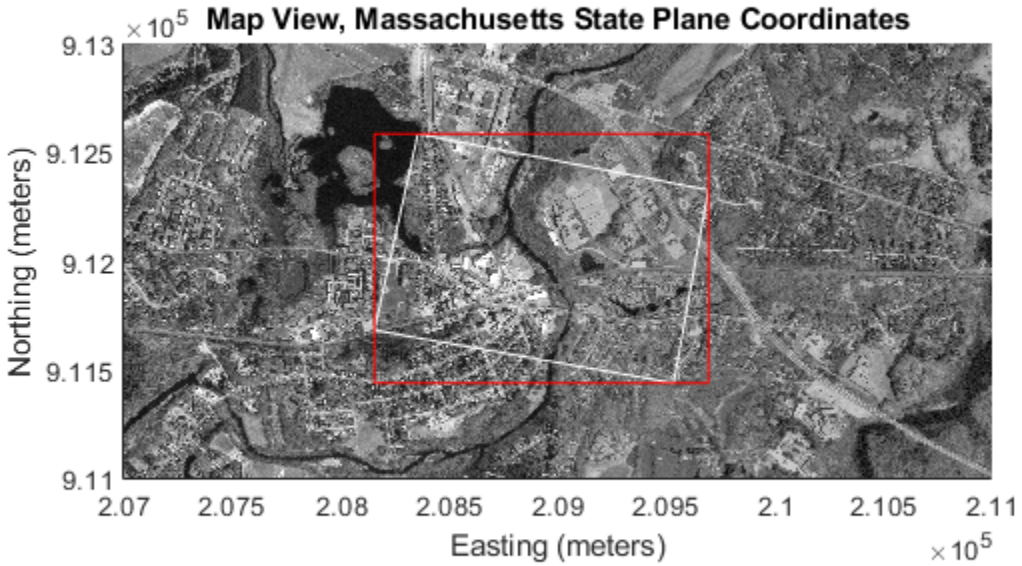
Choose world limits that are integer multiples of the output pixel size.

```
xWorldLimits = outputPixelSize ...
    * [floor(min(outputCornersX) / outputPixelSize), ...
       ceil(max(outputCornersX) / outputPixelSize)]

yWorldLimits = outputPixelSize ...
    * [floor(min(outputCornersY) / outputPixelSize), ...
       ceil(max(outputCornersY) / outputPixelSize)]
```

```
xWorldLimits =

      208154      209693


yWorldLimits =

      911435      912583
```

Display a bounding box for the registered image.

```
line(xWorldLimits([1 1 2 2 1]),yWorldLimits([2 1 1 2 2]),'Color','red')
```



The dimensions of the registered image will be as follows:

```
mOutput = diff(yWorldLimits) / outputPixelSize
nOutput = diff(xWorldLimits) / outputPixelSize
```

```
mOutput =

        1148


nOutput =

        1539
```

Create an Image Processing Toolbox referencing object for the registered image.

```
R = imref2d([mOutput nOutput],xWorldLimits,yWorldLimits)
```

```
R =

  imref2d with properties:

          XWorldLimits: [208154 209693]
          YWorldLimits: [911435 912583]
             ImageSize: [1148 1539]
    PixelExtentInWorldX: 1
    PixelExtentInWorldY: 1
    ImageExtentInWorldX: 1539
    ImageExtentInWorldY: 1148
       XIntrinsicLimits: [0.5 1539.5]
       YIntrinsicLimits: [0.5 1148.5]
```

Create a map raster reference object, which is the Mapping Toolbox counterpart to an Image Processing Toolbox referencing object.

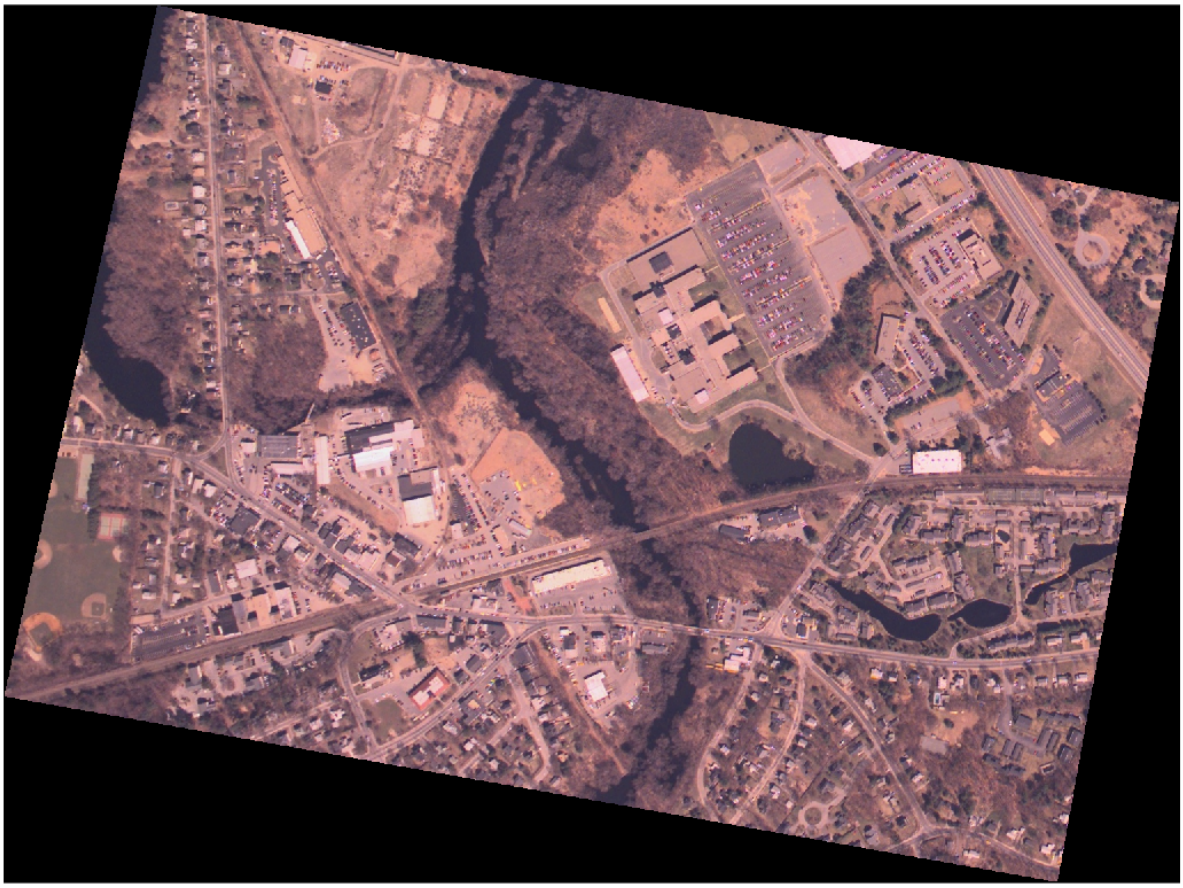
```
Rmap = maprasterref('RasterSize',R.ImageSize, ...
    'XWorldLimits',R.XWorldLimits,'YWorldLimits',R.YWorldLimits, ...
    'ColumnsStartFrom','north')
```

```
Rmap =

  MapCellsReference with properties:

           XWorldLimits: [208154 209693]
           YWorldLimits: [911435 912583]
             RasterSize: [1148 1539]
    RasterInterpretation: 'cells'
        ColumnsStartFrom: 'north'
           RowsStartFrom: 'west'
      CellExtentInWorldX: 1
      CellExtentInWorldY: 1
    RasterExtentInWorldX: 1539
    RasterExtentInWorldY: 1148
        XIntrinsicLimits: [0.5 1539.5]
        YIntrinsicLimits: [0.5 1148.5]
      TransformationType: 'rectilinear'
```

```
    CoordinateSystemType: 'planar'
            ProjectedCRS: []
```

Apply the geometric transformation using `imwarp`. Flip its output so that the columns run from north to south.

```
registered = flipud(imwarp(inputImage, tform, 'OutputView', R));
figure
imshow(registered)

format(currentFormat)
```



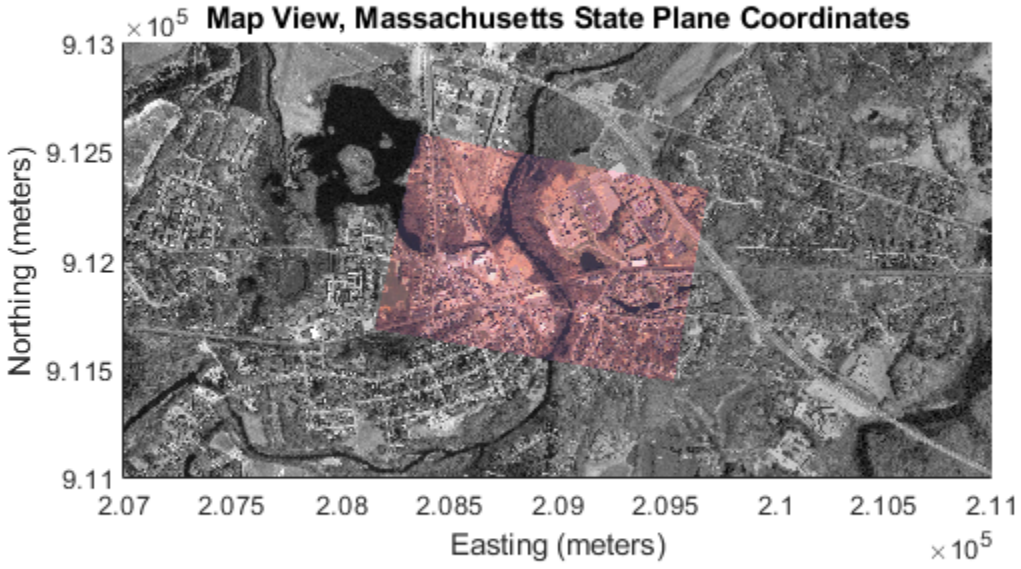
You can write the registered image as a TIFF with a world file, thereby georeferencing it to our map coordinates.

```
imwrite(registered,'concord_aerial_sw_reg.tif');
worldfilewrite(Rmap,getworldfilename('concord_aerial_sw_reg.tif'));
```

Step 4: Display the Registered Image in Map Coordinates

Display the registered image on the same (map coordinate) axes as the orthoimage base tiles. The registered image does not completely fill its bounding box, so it includes null-filled triangles. Create an alpha data mask to make these fill areas render as transparent.

```
alphaData = registered(:,:,1);
alphaData(alphaData~=0) = 255;

figure
mapshow(baseImage1,cmap1,R1)
ax2 = gca;
mapshow(ax2,baseImage2,cmap2,R2)
title('Map View, Massachusetts State Plane Coordinates');
xlabel('Easting (meters)');
ylabel('Northing (meters)');

mInput = mapshow(ax2,registered,Rmap);
mInput.AlphaData = alphaData;
```



You can assess the registration by looking at features that span both the registered image and the orthophoto images.

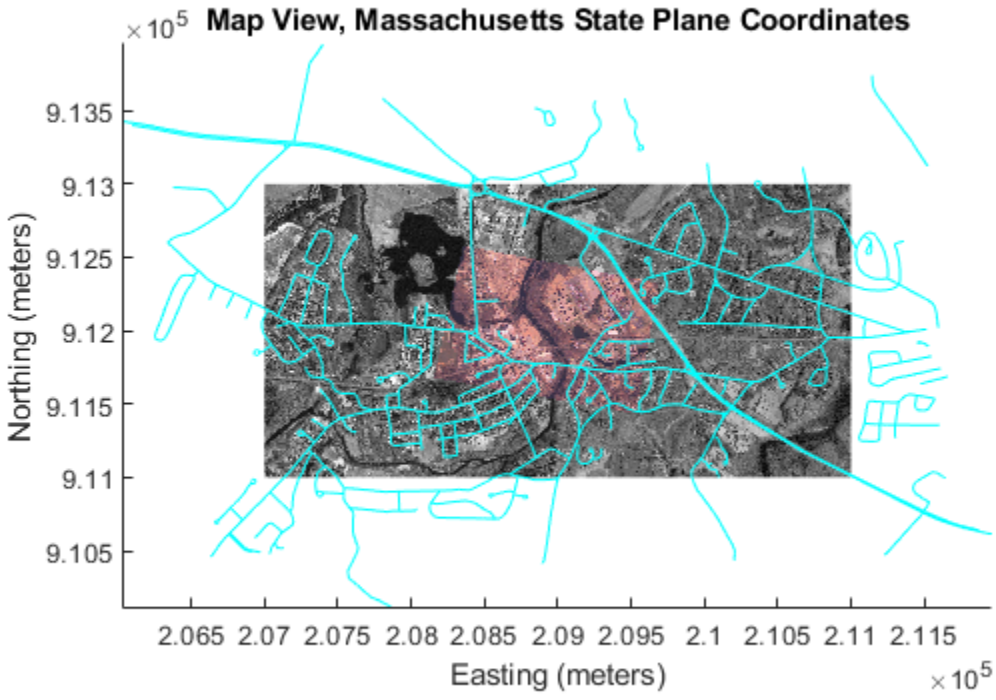
Step 5: Overlay Vector Road Layer (from Shape le)

Use `shapeinfo` and `shaperead` to learn about the attributes of the vector road layer. Render the roads on the same axes and the base tiles and registered image.

```
roadsfile = 'concord_roads.shp';
roadinfo = shapeinfo(roadsfile);
roads = shaperead(roadsfile);
```

Create symbolization based on the CLASS attribute (the type of road). Note that very minor roads have CLASS values equal to 6, so don't display them.

```
roadspec = makesymbolspec('Line',{'CLASS',6,'Visible','off'});

mapshow(ax2,roads,'SymbolSpec',roadspec,'Color','cyan')
```



Observe that the `roads` line up quite well with the roads in the images. Two obvious linear features in the images are not roads but railroads. The linear feature that trends roughly east-west and crosses both base tiles is the Fitchburg Commuter Rail Line of the Massachusetts Bay Transportation Agency. The linear feature that trends roughly northwest-southeast is the former Framingham-Lowell secondary line.

Credits

concord_orthow_w.tif, concord_ortho_e.tif, concord_roads.shp:

```
  Office of Geographic and Environmental Information (MassGIS),
  Commonwealth of Massachusetts  Executive Office of Environmental Affairs
  http://www.state.ma.us/mgis

  For more information, run:
```

```
>> type concord_ortho.txt
>> type concord_roads.txt
```

concord_aerial_sw.jpg

```
Visible color aerial photograph courtesy of mPower3/Emerge.

For more information, run:

>> type concord_aerial_sw.txt
```

See Also

MapCellsReference | cpstruct2pairs | fitgeotrans | im2uint8 | imread | imref2d | intrinsicToWorld | maprasterref | transformPointsForward | worldfileread

Find Geospatial Data Online

Many vector and raster data formats have been developed for storing geospatial data. With Mapping Toolbox you can read geodata files in general purpose formats (e.g., Esri® shapefile, GeoTIFF, and SDTS DEM) that a variety of mapping and image processing applications also read and write. You can also read files that are in a variety of special formats designed to exchange specific sets or kinds of geodata (e.g., GSHHG, VMAP0, DEM, and DTED files). You can order, and in many cases, download such data over the Internet.

Mapping Toolbox provides generalized sample data in the form of data files for the entire Earth and its major regions, as well as some higher resolution files covering small areas. These data sets are frequently used in the code examples provided in the Mapping Toolbox documentation. You can find them in *matlabroot*`/examples/map/data` and *matlabroot*`/toolbox/map/mapdata`. You can list them, along with their metadata, by typing the following at the command line:

```
ls(fullfile(matlabroot,'examples','map','data'))
ls(fullfile(matlabroot,'toolbox','map','mapdata'))
```

In addition, the `worlddatamap` function, available on MATLAB Central, allows you to use `worldmap` to map a region using data from a shapefile or data grid. Examples of `worlddatamap` and world vector data in shapefile format are available under the heading worlddatamap Examples.

For information about a small but useful subset of geodata resources on the Internet, see the following topics:

`Note` MathWorks does not warrant the accuracy, timeliness, or fitness for use of any data set listed in these topics, and makes no endorsement of any data vendor mentioned.

- “Find Vector Geodata” on page 2-71 — Lists URLs from which you can obtain vector (point, line, or polygon) geospatial data sets and data products, such as Esri shape files.
- “Find Geospatial Raster Data” on page 2-73 — Lists URLs from which you can obtain raster (gridded) geospatial data sets and data products, such as Digital Terrain Elevation Data (DTED). This topic also covers raster maps from Web Map Service servers.

`Note` If you are viewing this documentation installed locally (controlled by your Documentation location preference), you should also consult “Find Geospatial Data Online” on page 2-70 on the MathWorks website for possible updates and corrections.

Find Vector Geodata

This table contains some commonly used vector (point, line, or polygon) geospatial data sets that are available over the Internet. The table includes the names of Mapping Toolbox functions that read specific kinds of data. Click any numbered footnote in the right column to access data sets and associated documentation on the Web. Note, however, that Web addresses (URLs) for data can disappear or change, making some of the links unusable.

Note If you are using a Macintosh and the links in this table do not work, open the Mapping Toolbox documentation in a separate Web browser and view this section there. When you open the documentation, search for "Finding Vector Geodata" to find this topic.

Vector Data Set or Data Product	Data Provider	Import Functions	Internet URLs for Documentation and Data (HTTP or FTP)
Canadian provincial and Mexican state boundaries in zipped shapefile format	NOAA	`shapeinfo` `shaperead`	Documentation link: [1] Canada data link: [2] Mexico data link: [3]
Esri shapefiles	Esri and many other sources	`shapeinfo` `shaperead`	Documentation link: [1] For data links, see entries for TIGER, U.S. National Atlas, and U.S. coastlines.
Global Self-Consistent Hierarchical High-Resolution Geography (GSHHG)	NOAA/ NGDC	`gshhs`	Doc and data link: [1] Data (older versions): [2]
TIGER/Line® files	U.S. Census Bureau	`shapeinfo` `shaperead`	Documentation and data link: [1] Data (for 2000, in shapefile format): [2]
TIGER cartographic boundary files	U.S. Census Bureau	`shapeinfo` `shaperead`	Doc and Data link: [1] This site contains downloadable boundary files for census geographies in Esri Ungenerate, E00, and shapefile formats. Use shapefile format when importing to Mapping Toolbox.
U.S. coastlines, historical data	NOAA	`shapeinfo` `shaperead`	Documentation, metadata, and data in shapefile format: [1]
World coastlines from various sources	NGDC/ USGS	`load`	Data link: [1]

Abbreviations for data providing organizations used in the preceding table:

- Esri (Environmental Sciences Research Institute)
- NGDC (National Geophysical Data Center)
- NIST (National Institute of Standards and Technology)
- NOAA (National Oceanic and Atmospheric Administration)

- USGS (U.S. Geological Survey)

See Also

More About

- “Find Geospatial Data Online” on page 2-70

Find Geospatial Raster Data

In this section...

“Download Data” on page 2-73
“Use Web Map Service Data” on page 2-74

Get geospatial raster data over the Internet by downloading it or by accessing the Web Map Service (WMS) database.

Download Data

Find and download geospatial raster data using resources such as the ones in these tables. For information about supported file formats, see `readgeoraster` and `worldfileread`.

Note If you are using a Macintosh and the links on this page do not work, open the Mapping Toolbox documentation in a separate browser and view this section there. You can find this topic by searching for "Find Geospatial Raster Data".

Elevation

Resource	Provider	Examples of Products and Data Sets
EarthExplorer	US Geological Survey (USGS)	DTED, GMTED2010, GTOPO30
The National Map Download Application	USGS	3DEP
Data.gov	US General Services Administration	DTED, 3DEP, GMTED2010
ETOPO1 Global Relief Model	National Oceanic and Atmospheric Administration (NOAA) and National Centers for Environmental Information (NCEI)	ETOPO1 (use GeoTIFF format)
GMTED2010 Viewer	USGS	GMTED2010
The Global Land One-km Base Elevation Project	NOAA and NCEI	GLOBE
Global Topography	Scripps Institution of Oceanography	Smith and Sandwell

Land Cover Classification

Resource	Provider	Examples of Products and Data Sets
EarthExplorer	USGS	GLCC, AVHRR
Digital Coast Data Access Viewer	NOAA Office for Coastal Management	C-CAP Regional Land Cover and Change

Resource	Provider	Examples of Products and Data Sets
Data.gov	US General Services Administration	GLCC, AVHRR

Imagery

Resource	Provider	Examples of Products and Data Sets
EarthExplorer	USGS	Landsat
Digital Coast Data Access Viewer	NOAA Office for Coastal Management	High-Resolution Orthoimagery
Data.gov	US General Services Administration	Landsat

Use Web Map Service Data

Mapping Toolbox includes a built-in database of prequalified Web Map Service (WMS) servers and layers. Search the WMS database for layers using the `wmsfind` function. Read layers from the database using the `wmsread` function.

See Also

`readgeoraster` | `worldfileread`

Functions that Read and Write Geospatial Data

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

Function	Description	Type of Data	Type of Coordinates
`avhrrgoode`	Read data products derived from the Advanced Very High Resolution Radiometer (AVHRR) and stored in the Goode Homosoline projection: Global Land Cover Classification (GLCC) or Normalized Difference Vegetation Index (NDVI)	raster	geolocated
`avhrrlambert`	Read AVHRR GLCC and NDVI data products stored in the Lambert Conformal Conic projection	raster	geolocated
`dcwdata`	Read selected data from the Digital Chart of the World (DCW)	vector	geo
`dcwgaz`	Search for entries in the DCW gazette	vector	geo
`dcwread`	Read a DCW file	vector	geo
`dcwrhead`	Read a DCW file header	properties	geo
`demdataui`	GUI for interactively selecting data from various Digital Elevation Models (DEMs)	raster	geo
`dteds`	List DTED data grid file names for a specified latitude-longitude quadrangle	file names	geo
`egm96geoid`	Read 15-minute gridded geoid heights from the EGM96 geoid model	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
`georasterinfo`	Get information about data files in formats such as Esri Binary Grid, Esri GridFloat, DTED, GeoTIFF, and GPX	raster	map geo
`geotiffinfo`	Get information about GeoTIFF files	properties	map geo

Function	Description	Type of Data	Type of Coordinates
`geotiffwrite`	Write GeoTIFF file	raster	map geo
`getworldfilename`	Derive a world file name from an image file name	file name	geo map
`globedems`	List GLOBE data file names for a specified latitude-longitude quadrangle	file names	geo
`gshhs`	Read Global Self-Consistent Hierarchical High-Resolution Geography (GSHHG) data	vector	geo
`gtopo30s`	List GTOPO30 data file names for a specified latitude-longitude quadrangle	file names	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
`readgeoraster`	Read data in formats such as Esri Binary Grid, Esri GridFloat, DTED, GeoTIFF, and GPX	raster	geo map
`sdtsinfo`	Get information about SDTS data set	properties	geo
`shapeinfo`	Get information about the geometry and attributes of geographic features stored in a shapefile (a set of ".shp", ".shx" and ".dbf" files)	properties	geo map
`shaperead`	Read geographic feature coordinates and associated attributes from a shapefile	vector	geo map
`shapewrite`	Write geospatial data and associated attributes in shapefile format	vector	geo map
`usgsdems`	List USGS digital elevation model (DEM) file names covering a specified latitude-longitude quadrangle	file names	map
`vmap0data`	Extract selected data from the Vector Map Level 0 (VMAP0) CD-ROMs	vector	geo
`vmap0read`	Read a VMAP0 file	vector	geo
`vmap0rhead`	Read VMAP0 file headers	properties	geo
`vmap0ui`	Activate GUI for interactively selecting VMAP0 data	vector	geo
`worldfileread`	Read a world file and return a referencing matrix	georeferencing information	geo
`worldfilewrite`	Export a referencing matrix into an equivalent world file	georeferencing information	geo

The MATLAB environment provides many general file reading and writing functions (for example, `imread`, `imwrite`, `urlread`, and `urlwrite`) which you can use to access geospatial data you want to use with Mapping Toolbox software. For example, you can read a TIFF image with `imread` and its accompanying world file with `worldfileread` to import the image and construct a referencing matrix to georeference it. See the Mapping Toolbox examples “Creating a Half-Resolution Georeferenced Image” on page 2-53 and “Georeferencing an Image to an Orthotile Base Layer” on page 2-58 for examples of how you can do this.

Export Vector Geodata

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

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

For examples of and additional information about reading and writing shapefiles and DBF specs, see the documentation for `shapeinfo`, `shaperead`, `shapewrite`, and `makedbfspec`. The example provided in “How to Construct Geographic Data Structures” on page 2-25 also demonstrates exporting vector data using `shapewrite`. For information about creating KML files, see “Export KML Files for Viewing in Earth Browsers” on page 2-90.

Exporting Vector Data to KML

This example shows how to structure geographic point, line, and polygon vector data and export it to a Keyhole Markup Language (KML) file. KML is an XML-based markup language designed for visualizing geographic data on Web-based maps or "Earth browsers", such as Google Earth™, Google Maps™, NASA WorldWind, and the ESRI® ArcGIS™ Explorer.

The following functions write geographic data to a KML file:

- kmlwritepoint Write geographic points to KML file
- kmlwriteline Write geographic line to KML file
- kmlwritepolygon Write geographic polygon to KML file
- kmlwrite Write geographic data to KML file

De ne an Output Folder for the KML Files

This example creates several KML files and uses the variable `kmlFolder` to denote their location. The value used here is determined by the output of the `tempdir` command, but you could easily customize this.

```
kmlFolder = tempdir;
```

Create a cell array of the KML file names used in this example in order to optionally remove them from your KML output folder when the example ends.

```
kmlFilenames = {};
```

Create a Function Handle to Open an Earth Browser

A KML file can be opened in a variety of "Earth browsers", Web maps, or an editor. You can customize the following anonymous function handle to open a KML file. Executing this function handle launches the Google Earth browser, which must be installed on your computer. You can use the application by assigning the variable `useApplication` to `true` in your workspace or assign it to `true` here.

```
useApplication = exist('useApplication','var') && useApplication;

if useApplication
    if ispc
        % On Windows(R) platforms display the KML file with:
        openKML = @(filename) winopen(filename);
    elseif ismac
        % On Mac platforms display the KML file with:
        cmd = 'open -a Google\ Earth ';
        openKML = @(filename) system([cmd filename]);
    else
        % On Linux platforms display the KML file with:
        cmd = 'googleearth ';
        openKML = @(filename) system([cmd filename]);
    end
else
    % No "Earth browser" is installed on the system.
    openKML = @(filename) disp('');
end
```

Example 1: Write Single Point to KML File

This example writes a single point to a KML file.

Assign latitude and longitude values for Paderborn, Germany.

```
lat = 51.715254;
lon = 8.75213;
```

Use `kmlwritepoint` to write the point to a KML file.

```
filename = fullfile(kmlFolder,'Paderborn.kml');
kmlwritepoint(filename,lat,lon);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 2: Write Single Point to KML File with Icon and Description

This example writes a single point to a KML file. The placemark includes an icon and a description with HTML markup.

Assign latitude and longitude coordinates for a point that locates the headquarters of MathWorks® in Natick, Massachusetts.

```
lat = 42.299827;
lon = -71.350273;
```

Create a description for the placemark. Include HTML tags in the description to add new lines for the address.

```
description = sprintf('%s<br>%s</br><br>%s</br>', ...
   '3 Apple Hill Drive', 'Natick, MA. 01760', ...
   'https://www.mathworks.com');
```

Assign `iconFilename` to a GIF file on the local system's network.

```
iconDir = fullfile(matlabroot,'toolbox','matlab','icons');
iconFilename = fullfile(iconDir,'matlabicon.gif');
```

Assign the name for the placemark.

```
name = 'The MathWorks, Inc.';
```

Use `kmlwritepoint` to write the point and associated data to the KML file.

```
filename = fullfile(kmlFolder,'MathWorks.kml');
kmlwritepoint(filename,lat,lon,'Description',description,'Name',name, ...
   'Icon',iconFilename);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 3: Write Multiple Points to KML File

This example writes the locations of major European cities to a KML file, including the names of the cities, and removes the default description table.

Assign the latitude, longitude bounding box.

```
latlim = [ 30; 75];
lonlim = [-25; 45];
```

Read the data from the `worldcities` shapefile into a geostruct array.

```
cities = shaperead('worldcities.shp','UseGeoCoords',true, ...
    'BoundingBox',[lonlim, latlim]);
```

Convert to a geopoint vector.

```
cities = geopoint(cities);
```

Use `kmlwrite` to write the geopoint vector to a KML file. Assign the name of the placemark to the name of the city. Remove the default description since the data has only one attribute.

```
filename = fullfile(kmlFolder,'European_Cities.kml');
kmlwrite(filename,cities,'Name',cities.Name,'Description',{});
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 4: Write Multiple Points to KML File with Modified Attribute Table

This example writes placemarks at the locations of tsunami (tidal wave) events, reported over several decades and tagged geographically by source location, to a KML file.

Read the data from the tsunamis shapefile.

```
tsunamis = shaperead('tsunamis','UseGeoCoords',true);
```

Convert to a geopoint vector.

```
tsunamis = geopoint(tsunamis);
```

Sort the attributes.

```
tsunamis = tsunamis(:, sort(fieldnames(tsunamis)));
```

Construct an attribute specification.

```
attribspec = makeattribspec(tsunamis);
```

Modify the attribute specification to:

- Display `Max_Height`, `Cause`, `Year`, `Location`, and `Country` attributes
- Rename the `Max_Height` field to `Maximum Height`
- Highlight each attribute label with a bold font
- Set to zero the number of decimal places used to display Year
- We have independent knowledge that the height units are meters, so we will add that to the Height format specifier

```
desiredAttributes = {'Max_Height','Cause','Year','Location','Country'};
allAttributes = fieldnames(attribspec);
attributes = setdiff(allAttributes, desiredAttributes);
attribspec = rmfield(attribspec, attributes);
attribspec.Max_Height.AttributeLabel = '<b>Maximum Height</b>';
attribspec.Max_Height.Format = '%.1f Meters';
attribspec.Cause.AttributeLabel = '<b>Cause</b>';
attribspec.Year.AttributeLabel = '<b>Year</b>';
attribspec.Year.Format = '%.0f';
attribspec.Location.AttributeLabel = '<b>Location</b>';
attribspec.Country.AttributeLabel = '<b>Country</b>';
```

Use `kmlwrite` to write the geopoint vector containing the selected attributes and source locations to a KML file.

```
filename = fullfile(kmlFolder, 'Tsunami_Events.kml');
name = tsunamis.Location;
kmlwrite(filename,tsunamis,'Description',attribspec,'Name',name)
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 5: Write Single Point with a LookAt Virtual Camera to KML File

This example writes a single point with a LookAt virtual camera near Machu Picchu, Peru

Use a geopoint vector to define a LookAt virtual camera.

```
lat = -13.163111;
lon = -72.544945;
lookAt = geopoint(lat,lon);
lookAt.Range = 1500;
lookAt.Heading = 260;
lookAt.Tilt = 67;
```

Use `kmlwritepoint` to write the point location and LookAt information.

```
filename = fullfile(kmlFolder, 'Machu_Picchu.kml');
alt = 2430;
name = 'Machu Picchu';
kmlwritepoint(filename,lat,lon,alt,'LookAt',lookAt,'Name',name);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 6: Write Single Point with a Camera to KML File

This example writes a single point with a camera view of the Washington Monument in Washington D.C to a KML file. The marker is placed at the ground location of the camera.

Construct the camera.

```
camlat = 38.889301;
camlon = -77.039731;
camera = geopoint(camlat,camlon);
camera.Altitude = 500;
camera.Heading = 90;
camera.Tilt = 45;
camera.Roll = 0;
```

Use `kmlwritepoint` to write the point location and Camera information.

```
name = 'Camera ground location';
lat = camera.Latitude;
lon = camera.Longitude;
filename = fullfile(kmlFolder,'WashingtonMonument.kml');
kmlwritepoint(filename,lat,lon,'Camera',camera,'Name',name)
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 7: Write Address Data to KML File

This example writes unstructured address data to a KML file.

Create a cell array containing names of several places of interest in the Boston area.

```
names = {'Boston', ...
    'Massachusetts Institute of Technology', ...
    'Harvard University', ...
    'Fenway Park', ...
    'North End'};
```

Create a cell array containing addresses for the places of interest in the Boston area.

```
addresses = { ...
    'Boston, MA', ...
    '77 Massachusetts Ave, Cambridge, MA 02139', ...
    'Massachusetts Hall, Cambridge MA 02138', ...
    '4 Yawkey Way, Boston, MA', ...
    '134 Salem St, Boston, MA'};
```

Use a Google Maps icon for each of the placemarks.

```
icon = 'http://maps.google.com/mapfiles/kml/paddle/red-circle.png';
```

Use `kmlwrite` to write the cell array of addresses to the KML file.

```
filename = fullfile(kmlFolder, 'Places_of_Interest.kml');
kmlwrite(filename,addresses,'Name',names,'Icon',icon,'IconScale',1.5);
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 8: Write Single Line to KML File

This example writes a single line connecting the top of Mount Washington to the Mount Washington Hotel in Carroll, New Hampshire, to a KML file.

Assign coordinate values for the region of interest.

```
lat_Mount_Washington = 44.270489039;
lon_Mount_Washington = -71.303246453;

lat_Mount_Washington_Hotel = 44.258056;
lon_Mount_Washington_Hotel = -71.440278;

lat = [lat_Mount_Washington lat_Mount_Washington_Hotel];
lon = [lon_Mount_Washington lon_Mount_Washington_Hotel];
```

Set the altitude to 6 feet, for the approximate height of a person.

```
alt = 6 * unitsratio('meters', 'feet');
```

Add a camera viewpoint from the Mount Washington Hotel.

```
clat = lat(2);
clon = lon(2);
camera = geopoint(clat,clon,'Altitude',2,'Tilt',90,'Roll',0,'Heading',90);
```

Use `kmlwriteline` to write the arrays to a KML file.

```
filename = fullfile(kmlFolder, 'Mount_Washington.kml');
name = 'Mount Washington';
kmlwriteline(filename,lat,lon,alt,'Name',name,'Color','k','Width',3, ...
    'Camera',camera,'AltitudeMode','relativeToGround');
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 9: Write GPS Track Log to KML File

This example writes a GPS track log to a KML file.

Read the track log from the GPX file. The data in the track log was obtained from a GPS wristwatch held while gliding over Mount Mansfield in Vermont, USA, on August 28, 2010.

```
track = gpxread('sample_mixed','FeatureType','track');
```

Use `kmlwriteline` to write the track log to a KML file. The elevation values obtained by the GPS are relative to sea level.

```
filename = fullfile(kmlFolder, 'GPS_Track_Log.kml');
lat = track.Latitude;
lon = track.Longitude;
alt = track.Elevation;
name = 'GPS Track Log';
kmlwriteline(filename,lat,lon,alt,'Name',name,'Color','k','Width',2, ...
    'AltitudeMode','relativeToSeaLevel');
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 10: Write Circles to KML File

This example writes circles as lines around London City Airport to a KML file. The example includes a `LookAt` virtual camera.

Assign latitude and longitude values for the center of the feature.

```
lat0 = 51.50487;
lon0 = .05235;
```

Assign `azimuth` to [] to compute a complete small circle. Use the WGS84 ellipsoid.

```
azimuth = [];
spheroid = wgs84Ellipsoid;
```

Compute small circles of 3000, 2000, and 1000 meter radius. Assign a color value of `'blue'`, `'green'`, and `'red'` for each circle. Assign an elevation value of 100 meters (above ground) for each circle. Use a line geoshape vector to contain the data.

```
radius = 3000:-1000:1000;
colors = {'blue','green','red'};
elevation = 100;
circles = geoshape(0,0,'Name','','Color','','Elevation',elevation);
for k = 1:length(radius)
    [lat, lon] = scircle1(lat0,lon0,radius(k),azimuth,spheroid);
    circles(k).Latitude = lat;
    circles(k).Longitude = lon;
    circles(k).Name = [num2str(radius(k)) ' Meters'];
    circles(k).Color = colors{k};
    circles(k).Elevation = elevation;
end
```

Use a geopoint vector to define a LookAt virtual camera with a viewpoint from the east of the airport and aligned with the runway.

```
lat = 51.503169;
lon =  0.105478;
range = 3500;
heading = 270;
tilt =  60;
lookAt = geopoint(lat,lon,'Range',range,'Heading',heading,'Tilt',tilt);
```

Use `kmlwrite` to write the geoshape vector containing the circles and associated data to a KML file.

```
filename = fullfile(kmlFolder,'Small_Circles.kml');
kmlwrite(filename,circles,'AltitudeMode','relativeToGround','Width',2, ...
    'Name',circles.Name,'Color',circles.Color,'LookAt',lookAt);
```

Open the KML file. Using Google Earth, the `LookAt` view point is set when clicking on either one of the `1000 Meters`, `2000 Meters`, or `3000 Meters` strings in the Places list.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 11: Write Circular Polygons to KML File

This example writes circular polygons around London City Airport to a KML file. It includes a `LookAt` virtual camera and uses the same data calculated in step 9.

Change the `Geometry` property value of the geoshape vector to `'polygon'`. The polygons are drawn in the same order as the geoshape vector and are indexed from largest to smallest radii, thus each polygon will be visible in the browser.

```
circles.Geometry = 'polygon';
```

Change the elevation of each polygon.

```
circles.Elevation = 1000:1000:3000;
```

Use a geopoint vector to define a LookAt virtual camera with a viewpoint from the east of the airport, aligned with the runway, and with a view of all three polygons.

```
lat = 51.501587;
lon = 0.066147;
range = 13110;
heading = 270;
tilt = 60;
lookAt = geopoint(lat,lon,'Range',range,'Heading',heading,'Tilt',tilt);
```

Use `kmlwrite` to write the polygon geoshape vector containing the circular polygons and associated data to a KML file. Extrude the polygons to the ground. Set the polygon edge color to black and assign a face alpha value to provide visibility inside the polygon.

```
filename = fullfile(kmlFolder,'Small_Circle_Polygons.kml');
name = circles.Name;
color = circles.Color;
kmlwrite(filename,circles,'AltitudeMode','relativeToGround','Extrude',true, ...
    'Name',name,'FaceColor',color,'EdgeColor','k','FaceAlpha',.6,'LookAt',lookAt);
```

Open the KML file. Using Google Earth, the `LookAt` view point is set when clicking on either one of the `1000 Meters`, `2000 Meters`, or `3000 Meters` strings in the Places list.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 12: Write Polygon Data from Shapefile to KML File

This example writes polygon data from the `usastatelo` shapefile to a KML file. The polygon faces are set with a color appropriate for political regions. The polygon faces are set with an alpha value to provide visibility inside the polygon.

```
states = shaperead('usastatelo','UseGeoCoords',true);
states = geoshape(states);
colors = polcmap(length(states));
name = states.Name;
filename = fullfile(kmlFolder,'usastatelo.kml');
kmlwrite(filename,states,'Name',name,'FaceColor',colors,'FaceAlpha',.6, ...
    'EdgeColor','k')
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 13: Write Polygon Contours to KML File

This example contours a grid in a local coordinate system, returns the contours in a geographic system, and writes the polygon contours to a KML file.

Create a grid in a local system.

```
X = -150000:10000:150000;
Y =  0:10000:300000;
[xmesh, ymesh] = meshgrid(X/50000, (Y - 150000)/50000);
Z = 8 + peaks(xmesh, ymesh);
```

Define a local geodetic origin near Frankfurt, Germany and an ellipsoidal height.

```
lat0 = 50.108;
lon0 = 8.6732;
h0 = 100;
```

Define contour levels.

```
levels = 0:2:18;
```

Contour the grid and return the output in a polygon geoshape vector.

```
[~, contourPolygons] = geocontourxy(X,Y,Z,lat0,lon0,h0,'LevelList',levels);
```

Output the contours to a KML file. Set the faces with an alpha value. Set `CutPolygons` to `false` since the altitude values are not uniform. Clamp the polygons to the ground.

```
colors = parula(length(contourPolygons));
filename = fullfile(kmlFolder,'Contour_Polygons.kml');
```

```
kmlwrite(filename,contourPolygons,'FaceColor',colors,'FaceAlpha',.6, ...
    'EdgeColor','k','CutPolygons',false,'AltitudeMode','clampToGround')
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Example 14: Write Polygon with Inner Ring to KML File

This example constructs a polygon with an inner ring around the Eiffel Tower and writes the polygon to a KML file. The polygon's altitude is set to 500 meters above ground.

```
lat0 = 48.858288;
lon0 = 2.294548;
outerRadius = .02;
innerRadius = .01;
[lat1,lon1] = scircle1(lat0,lon0,outerRadius);
[lat2,lon2] = scircle1(lat0,lon0,innerRadius);
[lon2,lat2] = poly2ccw(lon2,lat2);
lat = [lat1; NaN; lat2];
lon = [lon1; NaN; lon2];
alt = 500;
filename = fullfile(kmlFolder,'EiffelTower.kml');
```

Export the polygon to a KML file. Set the edge color to black, the face color to cyan, and the face alpha value.

```
kmlwritepolygon(filename,lat,lon,alt,'EdgeColor','k','FaceColor','c', ...
    'FaceAlpha',.5)
```

Open the KML file.

```
openKML(filename)
```

Add `filename` to `kmlFilenames`.

```
kmlFilenames{end+1} = filename;
```

Delete Generated KML Files

Optionally, delete the new KML files from your KML output folder.

```
if ~useApplication
    for k = 1:length(kmlFilenames)
        delete(kmlFilenames{k})
    end
end
```

Data Set Information

The data in `worldcities.shp` is from the Digital Chart of the World (DCW) browser layer, published by the U.S. National Geospatial-Intelligence Agency (NGA), formerly the National Imagery and Mapping Agency (NIMA). For more information about the data set, use the command `type worldcities.txt`.

The data in `tsunamis.shp` is from the Global Tsunami Database, U.S. National Geospatial Data Center (NGDC), National Oceanic and Atmospheric Administration (NOAA). For more information about the data set, use the command `type tsunamis.txt`.

The data in `usastatelo.shp` is based on data from the CIA World DataBank II and the U.S. Census Bureau site "State and County QuickFacts". For more information about the data set, use the command `type usastatelo.txt`. For an updated link to the U.S. Census Bureau site "State and County QuickFacts", see https://www.census.gov/quickfacts/fact/table/US/PST045219.

See Also

`kmlwrite` | `kmlwriteline` | `kmlwritepoint` | `kmlwritepolygon`

Export KML Files for Viewing in Earth Browsers

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

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

Generate a Single Placemark Using kmlwritepoint

This example shows how to generate a placemark using `kmlwritepoint` by specifying the latitude and longitude that identifies a location. This example also specifies the icon used for the placemark and the text that appears in the balloon associated with the placemark.

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

This code produces the following KML file.

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
   <Document>
      <name>MathWorks</name>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description>3 Apple Hill Drive<br>Natick, MA. 01760</b>;
                     <br>https://www.mathworks.com</b>;
         </description>
         <name>The MathWorks, Inc.</name>
         <Style>
            <IconStyle>
               <Icon>
                 <href>
                   https://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, "https://www.mathworks.com," was automatically rendered as a hyperlink. The Google Earth service also adds a link called "Directions". `kmlwritepoint` 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.

Generate Placemarks from Addresses

This example shows how to generate a placemark using street addresses or more general addresses such as postal codes, city, state, or country names, instead of latitude and longitude information. If the viewing application is capable of looking up addresses, such placemarks can be displayed in appropriate, although possibly imprecise, locations. (Note that the Google Maps service does not support address-based placemarks.)

When you use addresses, `kmlwrite` creates an `<address>` element for each placemark rather than `<point>` elements containing `<coordinates>` elements. For example, here is code for `kmlwrite` that generates address-based placemarks for three cities in Australia from a cell array:

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

The generated KML file has the following structure and content:

```
<?xml version="1.0" encoding="utf-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
   <Document>
      <name>Australian_Cities</name>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Perth, Australia</name>
         <address>Perth, Australia</address>
      </Placemark>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Melbourne, Australia</name>
         <address>Melbourne, Australia</address>
      </Placemark>
      <Placemark>
         <Snippet maxLines="0"> </Snippet>
         <description> </description>
         <name>Sydney, Australia</name>
         <address>Sydney, Australia</address>
      </Placemark>
   </Document>
</kml>
```

Export Point Geostructs to Placemarks

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

The Mapping Toolbox tsunamis shapefiles contain a database of 162 tsunami (tidal wave) events reported between 1950 and 2006, described as point locations with 21 variables (including 18 attributes). You can type out the metadata file `tsunamis.txt` to see the definitions of all the data fields. The steps below select some of these from the shapefiles and display them as tables in exported KML placemarks.

1 Read the tsunami shapefiles, selecting certain attributes.

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

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

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

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

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

```
tsunamis(1)
```

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

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

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

```
kmlfilename = 'tsunami1.kml';
kmlwritepoint(kmlfilename,tsunamis(1).Lat,tsunamis(1).Lon);
```

3 View the placemarks in an earth browser. For example, you can view KML files with the Google Earth browser, which must be installed on your computer.

For Windows, use the `winopen` function:

```
winopen(filename)
```

For Linux, if the file name is a partial path, use the following commands:

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

For Mac, if the file name is a partial path, use the following commands:

```
cmd = 'open -a Google\ Earth '
fullfilename = fullfile(pwd, filename);
system([cmd fullfilename])
```

4 Customize the placemark contents

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

```
attribspec = makeattribspec(tsunamis)

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

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

```
attribspec.Max_Height.AttributeLabel = '<b>Maximum Height</b>';
attribspec.Max_Height.Format = '%.1f Meters';
attribspec.Cause.AttributeLabel = '<b>Cause</b>';
attribspec.Year.AttributeLabel = '<b>Year</b>';
attribspec.Year.Format = '%.0f';
attribspec.Location.AttributeLabel = '<b>Location</b>';
attribspec.Country.AttributeLabel = '<b>Country</b>';
```

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

5 Customize the placemark icon

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

```
iconname = ...
  'http://maps.google.com/mapfiles/kml/shapes/placemark_circle.png';
kmlwritepoint(kmlfilename,tsunamis(1).Lat,tsunamis(1).Lon, ...
  'Description',attribspec,'Name',{tsunamis(1).Location}, ...
  'Icon',iconname,'IconScale',2);
```

6 Vary placemark size by tsunami height

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

```
% Create vector with log2 exponents of |Max_Height| values
[loghgtx loghgte] = log2([tsunamis.Max_Height]);
% Create a vector replicating the icon URL
iconnames = cellstr(repmat(iconname,numel(tsunamis),1));
kmlwritepoint(kmlfilename,tsunamis(1).Lat,tsunamis(1).Lon,
    'Description',attribspec,...
    'Name',{tsunamis(1).Location},'Icon',iconname,...
    'IconScale',loghgte);
```

Select Shapefile Data to Read

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 character vector indicating the name of an attribute.

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

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

Example 1: Predicate Function in Separate File

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

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

2 You can then call `shaperead` like this:

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

roadselector =
    @roadfilter    'CLASS'    'LENGTH'

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

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

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

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

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

Prior to Version 7 of the Mapping Toolbox software, putting the selector in a file or local function 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 local function thereof rather than in a file of its own.

Example 2: Predicate as Function Handle

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

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

Example 3: Predicate as Anonymous Function

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

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

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

s = shaperead('concord_roads','Selector', ...
```

```
              {roadfilterfcn, 'CLASS', 'LENGTH'})

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

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

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

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

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

Example 5: Parametrizing 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 to utilize workspace variables in as parameters, rather than requiring that the parameters be hard-coded as constants within the predicate function. In the following example, the workspace variables `minimumClass` and `minimumLength` could have been assigned through a variety of computations whose results were unknown until run-time, yet their values can be made available within the predicate as long as it is defined as a nested function. In this example the nested function is wrapped in a file called `constructroadselector.m`, which returns a complete selector: a handle to the predicate (named `nestedroadfilter`) and the two 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
```

Exporting Images and Raster Grids to GeoTIFF

This example shows how to write data referenced to standard geographic and projected coordinate systems to GeoTIFF files, using `geotiffwrite`. The Tagged-Image File Format (TIFF) has emerged as a popular format to store raster data. The GeoTIFF specification defines a set of TIFF tags that describe "Cartographic" information associated with the TIFF raster data. Using these tags, geolocated imagery or raster grids with coordinates referenced to a Geographic Coordinate System (latitude and longitude) or a (planar) Projected Coordinate System can be stored in a GeoTIFF file.

Setup: Define a Data Folder and File Name Utility Function

This example creates several temporary GeoTIFF files and uses the variable `datadir` to denote their location. The value used here is determined by the output of the `tempdir` command, but you could easily customize this. The contents of `datadir` are deleted at the end of the example.

```
datadir = fullfile(tempdir, 'datadir');
if ~exist(datadir, 'dir')
   mkdir(datadir)
end
```

Define an anonymous function to prepend `datadir` to the input file name:

```
datafile = @(filename)fullfile(datadir, filename);
```

Example 1: Write an Image Referenced to Geographic Coordinates

Write an image referenced to WGS84 geographic coordinates to a GeoTIFF file. The original image (boston_ovr.jpg) is stored in JPEG format, with referencing information external to the image file, in the "world file" (boston_ovr.jgw). The image provides a low resolution "overview" of Boston, Massachusetts, and the surrounding area.

Read the image from the JPEG file.

```
basename = 'boston_ovr';
imagefile = [basename '.jpg'];
A1 = imread(imagefile);
```

Obtain a referencing object from the world file.

```
worldfile = getworldfilename(imagefile);
R1 = worldfileread(worldfile,'geographic',size(A1));
```

Write the image to a GeoTIFF file.

```
filename1 = datafile([basename '.tif']);
geotiffwrite(filename1,A1,R1)
```

Return information about the file as a `RasterInfo` object. Note that the value of `CoordinateReferenceSystem` is a geographic coordinate reference system object.

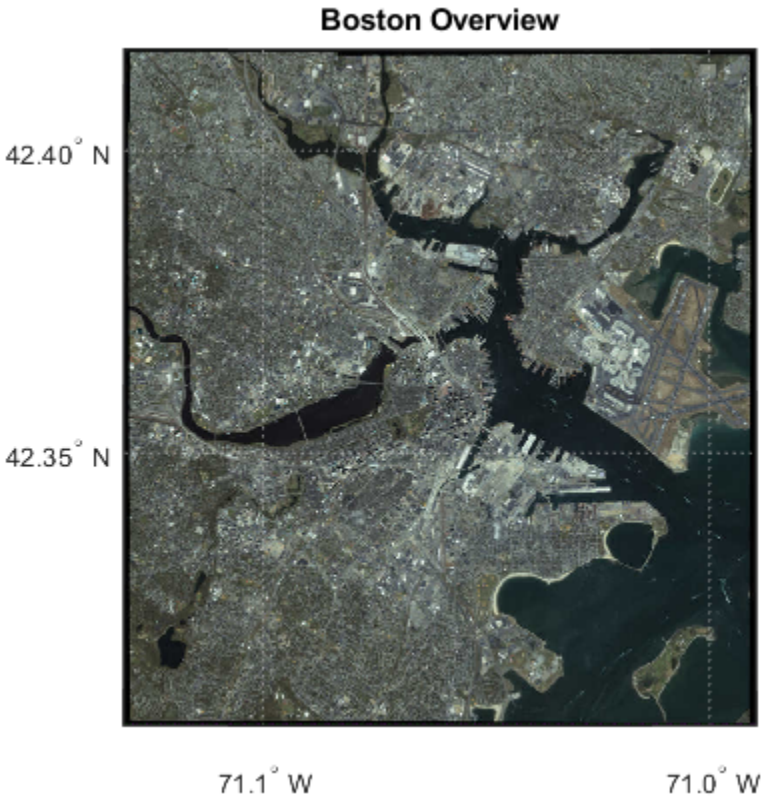
```
info1 = georasterinfo(filename1);
info1.CoordinateReferenceSystem
```

```
ans =

  geocrs with properties:
```

```
           Name: "WGS 84"
          Datum: "World Geodetic System 1984"
      Spheroid: [1×1 referenceEllipsoid]
PrimeMeridian: 0
    AngleUnit: "degree"
```

Re-import the new GeoTIFF file and display the Boston overview image, correctly located, in a map axes.

```
figure
usamap(R1.LatitudeLimits,R1.LongitudeLimits)
setm(gca,'PLabelLocation',0.05,'PlabelRound',-2,'PlineLocation',0.05)
geoshow(filename1)
title('Boston Overview')
```



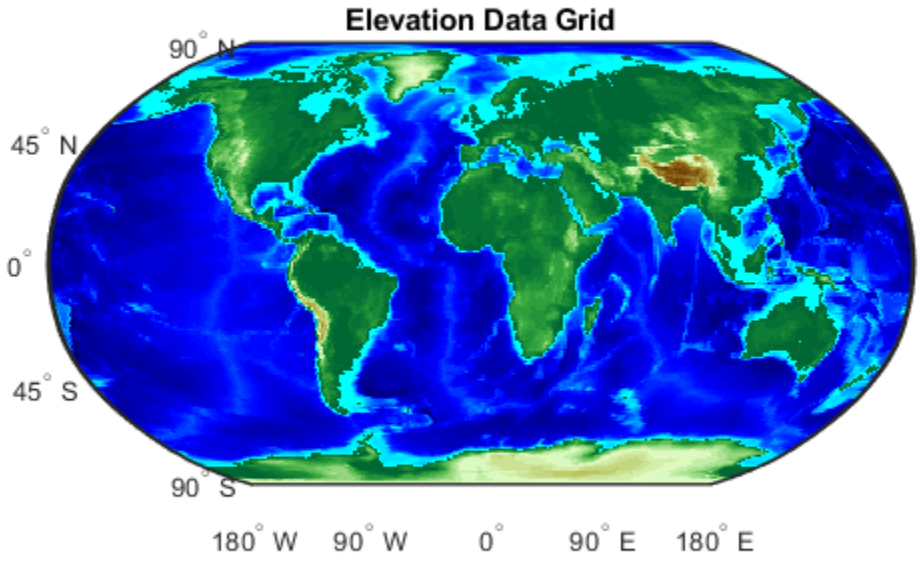
Example 2: Write a Data Grid Referenced to Geographic Coordinates

Load elevation raster data and a geographic cells reference object. Write the data grid to a GeoTIFF file.

```
load topo60c
Z2 = topo60c;
R2 = topo60cR;
filename2 = datafile('topo60c.tif');
geotiffwrite(filename2,Z2,R2)
```

The values in the data grid range from -7473 to 5731. Display the grid as a texture-mapped surface rather than as an intensity image.

```
figure
worldmap world
gridm off
setm(gca,'MLabelParallel',-90,'MLabelLocation',90)
tmap = geoshow(filename2,'DisplayType','texturemap');
demcmap(tmap.CData)
title('Elevation Data Grid')
```



Example 3: Change Data Organization of GeoTIFF Files

When you write data using `geotiffwrite` or read data using `readgeoraster`, the columns of the data grid start from north and the rows start from west. For example, the input data from `topo60c.mat` starts from south, but the output data from `topo60c.tif` starts from north.

```
R2.ColumnsStartFrom
[Z3,R3] = readgeoraster(filename2);
R3.ColumnsStartFrom
```

```
ans =

    'south'


ans =
```

```
    'north'
```

Therefore, the input data and data in the GeoTIFF file is flipped.

```
isequal(Z2,flipud(Z3))
```

```
ans =

  logical

   1
```

If you need the data in your workspace to match again, then flip the Z values and set the referencing object such that the columns start from the south:

```
R3.ColumnsStartFrom = 'south';
Z3 = flipud(Z3);
isequal(Z2,Z3)
```

```
ans =

  logical

   1
```

The data in the GeoTIFF file is encoded with positive scale values. Therefore, when you view the file with ordinary TIFF-viewing software, the northern edge of the data set is at the top. To make the data layout in the output file match the data layout of the input, you can construct a Tiff object and use it to reset some of the tags and the image data.

```
t = Tiff(filename2,'r+');

pixelScale = getTag(t,'ModelPixelScaleTag');
pixelScale(2) = -pixelScale(2);
setTag(t,'ModelPixelScaleTag',pixelScale);

tiepoint = getTag(t,'ModelTiepointTag');
tiepoint(5) = intrinsicToGeographic(R2,0.5,0.5);
setTag(t,'ModelTiepointTag',tiepoint);

setTag(t,'Compression', Tiff.Compression.None)

write(t,Z2);

rewriteDirectory(t)
close(t)
```

Verify the referencing object and data grid from the input data match the output data file. To do this, read the Tiff image and create a reference object. Then, compare the grids.

```
t = Tiff(filename2);
Atiff = read(t);
```

```
close(t)
Rtiff = georefcells(R2.LatitudeLimits,R2.LongitudeLimits,size(Atiff));

isequal(Z2,Atiff)
isequal(R2,Rtiff)
```

```
ans =

  logical

   1


ans =

  logical

   1
```

Example 4: Write an Image Referenced to a Projected Coordinate System

Write the Concord orthophotos to a single GeoTIFF file. The two adjacent (west-to-east) georeferenced grayscale (panchromatic) orthophotos cover part of Concord, Massachusetts, USA. The concord_ortho.txt file indicates that the data are referenced to the Massachusetts Mainland (NAD83) State Plane Projected Coordinate System. Units are meters. This corresponds to the GeoTIFF code number 26986 as noted in the GeoTIFF specification at http://geotiff.maptools.org/spec/geotiff6.html#6.3.3.1 under PCS_NAD83_Massachusetts.

Read the two orthophotos.

```
[X_west,R_west] = readgeoraster('concord_ortho_w.tif');
[X_east,R_east] = readgeoraster('concord_ortho_e.tif');
```

Combine the images and reference objects.

```
X4 = [X_west X_east];
R4 = R_west;
R4.XWorldLimits = [R_west.XWorldLimits(1) R_east.XWorldLimits(2)];
R4.RasterSize = size(X4);
```

Write the data to a GeoTIFF file. Use the code number, 26986, indicating the PCS_NAD83_Massachusetts Projected Coordinate System.

```
coordRefSysCode = 26986;
filename4 = datafile('concord_ortho.tif');
geotiffwrite(filename4,X4,R4,'CoordRefSysCode',coordRefSysCode)
```

Return information about the file as a `RasterInfo` object. Note that the value of `CoordinateReferenceSystem` is a projected coordinate reference system object.

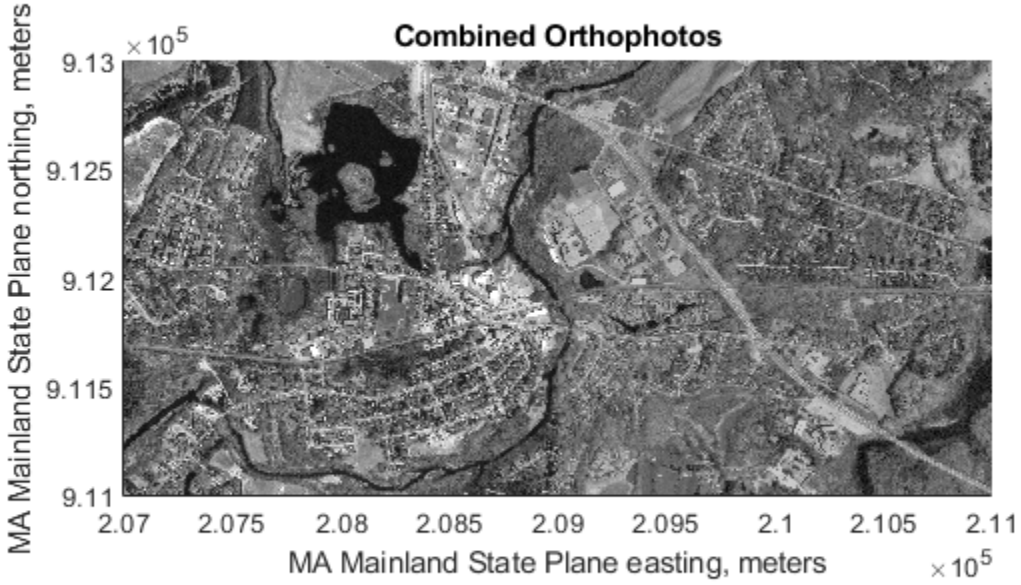
```
info4 = georasterinfo(filename4);
info4.CoordinateReferenceSystem
```

```
ans =
```

```
  projcrs with properties:

                  Name: "NAD83 / Massachusetts Mainland"
         GeographicCRS: [1×1 geocrs]
      ProjectionMethod: "Lambert Conic Conformal (2SP)"
            LengthUnit: "meter"
  ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Display the combined Concord orthophotos.

```
figure
mapshow(filename4)
title('Combined Orthophotos')
xlabel('MA Mainland State Plane easting, meters')
ylabel('MA Mainland State Plane northing, meters')
```



Example 5: Write a Cropped Image from a GeoTIFF File

Write a subset of a GeoTIFF file to a new GeoTIFF file.

Read the RGB image and referencing information from the `boston.tif` GeoTIFF file.

```
[A5,R5] = readgeoraster('boston.tif');
```

Crop the image.

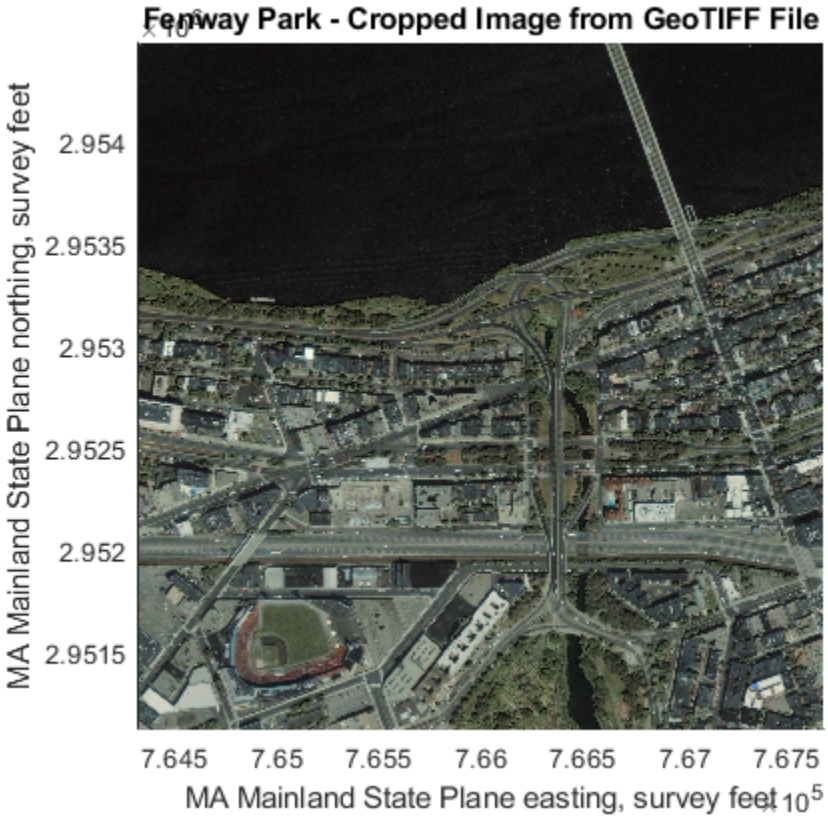
```
xlimits = [ 764318  767677];
ylimits = [2951122 2954482];
[A5crop,R5crop] = mapcrop(A5,R5,xlimits,ylimits);
```

Write the cropped image to a GeoTIFF file. Use the GeoKeyDirectoryTag from the original GeoTIFF file.

```
info5 = geotiffinfo('boston.tif');
filename5 = datafile('boston_subimage.tif');
geotiffwrite(filename5,A5crop,R5crop, ...
   'GeoKeyDirectoryTag',info5.GeoTIFFTags.GeoKeyDirectoryTag)
```

Display the GeoTIFF file containing the cropped image.

```
figure
mapshow(filename5)
title('Fenway Park - Cropped Image from GeoTIFF File')
xlabel('MA Mainland State Plane easting, survey feet')
ylabel('MA Mainland State Plane northing, survey feet')
```



Example 6: Write Elevation Data to GeoTIFF File

Write elevation data for an area around South Boulder Peak in Colorado to a GeoTIFF file.

```
elevFilename = 'n39_w106_3arc_v2.dt1';
```

Read the DEM from the file. To plot the data using `geoshow`, the data must be of type `single` or `double`. Specify the data type for the raster using the `'OutputType'` name-value pair.

```
[Z6,R6] = readgeoraster(elevFilename,'OutputType','double');
```

Create a structure to hold the GeoKeyDirectoryTag information.

```
key = struct( ...
    'GTModelTypeGeoKey',[], ...
    'GTRasterTypeGeoKey',[], ...
    'GeographicTypeGeoKey',[]);
```

Indicate the data is in a geographic coordinate system by specifying the `GTModelTypeGeoKey` field as 2. Indicate that the reference object uses postings (rather than cells) by specifying the `GTRasterTypeGeoKey` field as 2. Indicate the data is referenced to a geographic coordinate reference system by specifying the `GeographicTypeGeoKey` field as 4326.

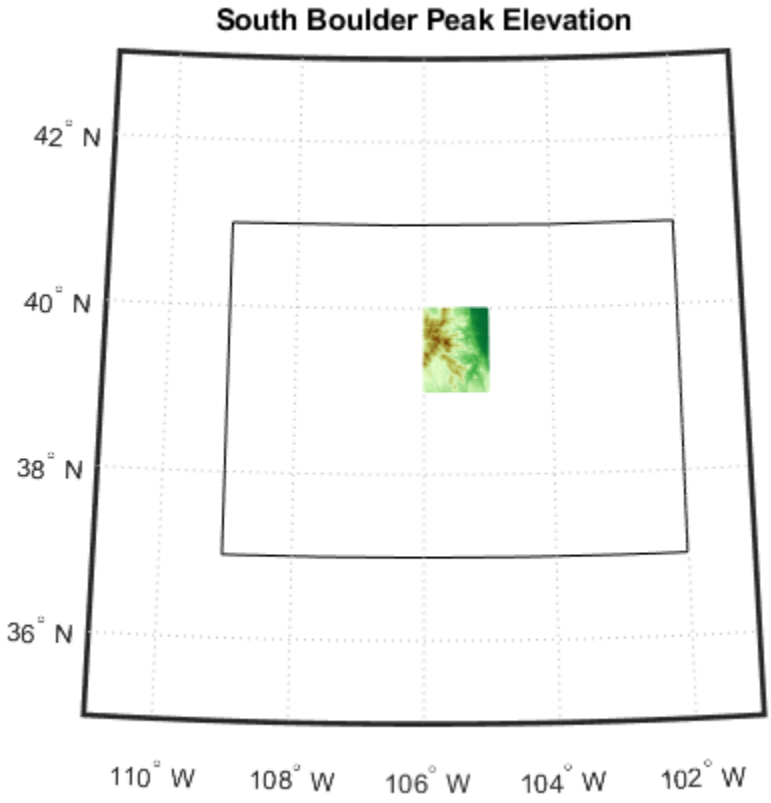
```
key.GTModelTypeGeoKey = 2;
key.GTRasterTypeGeoKey = 2;
key.GeographicTypeGeoKey = 4326;
```

Write the elevation data to a GeoTIFF file.

```
filename6 = datafile('southboulder.tif');
geotiffwrite(filename6,Z6,R6,'GeoKeyDirectoryTag',key)
```

Verify the data has been written to a file by displaying it. First, import vector data that represents the state boundary of Colorado using `shaperead`. Then, display the boundary and GeoTIFF file.

```
S = shaperead('usastatelo','UseGeoCoords',true,'Selector',...
   {@(name) any(strcmp(name,{'Colorado'})),'Name'});
figure
usamap 'Colorado'
hold on
geoshow(S,'FaceColor','none')
g = geoshow(filename6,'DisplayType','mesh');
demcmap(g.ZData)
title('South Boulder Peak Elevation')
```



Example 7: Write Non-Image Data to a TIFF File

If you are working with a data grid that is class double with values that range outside the limits required of a floating point intensity image (0 <= data <= 1), and if you store the data in a TIFF file using `imwrite`, then your data will be truncated to the interval [0,1], scaled, and converted to uint8. Obviously it is possible for some or even all of the information in the original data to be lost. To avoid these problems, and preserve the numeric class and range of your data grid, use `geotiffwrite` to write the data.

Create sample Z data.

```
n = 512;
Z7 = peaks(n);
```

Create a referencing object to reference the rows and columns to X and Y.

```
R7 = maprasterref('RasterSize',[n n],'ColumnsStartFrom','north');
R7.XWorldLimits = R7.XIntrinsicLimits;
R7.YWorldLimits = R7.YIntrinsicLimits;
```

Create a structure to hold the GeoKeyDirectoryTag information. Set the model type to 1 indicating Projected Coordinate System (PCS).

```
key.GTModelTypeGeoKey = 1;
```

Set the raster type to 1 indicating PixelIsArea (cells).

```
key.GTRasterTypeGeoKey = 1;
```

Indicate a user-defined Projected Coordinate System by using a value of 32767.

```
key.ProjectedCSTypeGeoKey = 32767;
```

Write the data to a GeoTIFF file with `geotiffwrite`. For comparison, write a second file using `imwrite`.

```
filename_geotiff = datafile('zdata_geotiff.tif');
filename_tiff = datafile('zdata_tiff.tif');
geotiffwrite(filename_geotiff,Z7,R7,'GeoKeyDirectoryTag',key)
imwrite(Z7, filename_tiff);
```

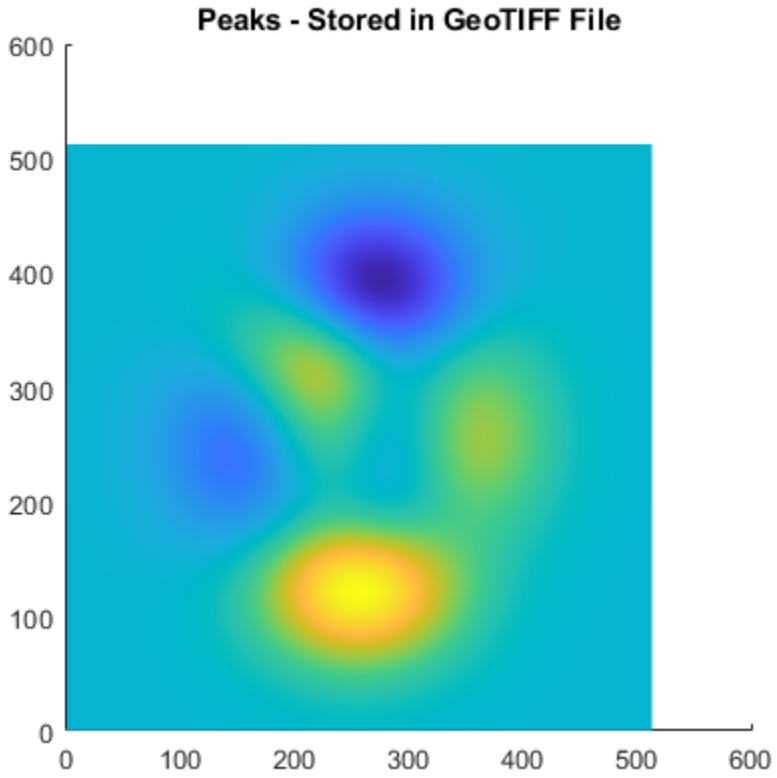
When you read the file using `imread` the data values and class type are preserved. You can see that the data values in the TIFF file are not preserved.

```
geoZ  = imread(filename_geotiff);
tiffZ = imread(filename_tiff);
fprintf('Class type of Z: %s\n', class(Z7))
fprintf('Class type of data in GeoTIFF file: %s\n', class(geoZ))
fprintf('Class type of data in    TIFF file: %s\n', class(tiffZ))
fprintf('Does data in GeoTIFF file equal Z: %d\n', isequal(geoZ, Z7))
fprintf('Does data in    TIFF file equal Z: %d\n', isequal(tiffZ, Z7))
```

```
Class type of Z: double
Class type of data in GeoTIFF file: double
Class type of data in    TIFF file: uint8
Does data in GeoTIFF file equal Z: 1
Does data in    TIFF file equal Z: 0
```

You can view the data grid using `mapshow`.

```
figure
mapshow(filename_geotiff,'DisplayType','texturemap')
title('Peaks - Stored in GeoTIFF File')
```



Example 8: Modify an Existing File While Preserving Meta Information

You may want to modify an existing file, but preserve most, if not all, of the meta information in the TIFF tags. This example converts the RGB image from the `boston.tif` file into an indexed image and writes the new data to an indexed GeoTIFF file. The TIFF meta-information, with the exception of the values of the BitDepth, BitsPerSample, and PhotometricInterpretation tags, is preserved.

Read the image from the `boston.tif` GeoTIFF file.

```
[A8,R8] = readgeoraster('boston.tif');
```

Use the MATLAB function, `rgb2ind`, to convert the RGB image to an indexed image `X` using minimum variance quantization.

```
[X8,cmap] = rgb2ind(A8,65536);
```

Obtain the TIFF tag information using `imfinfo`.

```
info8 = imfinfo('boston.tif');
```

Create a TIFF tags structure to preserve selected information from the `info` structure.

```
tags = struct( ...
    'Compression',  info8.Compression, ...
    'RowsPerStrip', info8.RowsPerStrip, ...
    'XResolution',  info8.XResolution, ...
    'YResolution',  info8.YResolution, ...
    'ImageDescription', info8.ImageDescription, ...
```

```
    'DateTime',    info8.DateTime, ...
    'Copyright',   info8.Copyright, ...
    'Orientation', info8.Orientation);
```

The values for the PlanarConfiguration and ResolutionUnit tags must be numeric rather than string valued, as returned by `imfinfo`. You can set these tags by using the constant properties from the Tiff class. For example, here are the possible values for the PlanarConfiguration constant property:

```
Tiff.PlanarConfiguration
```

```
ans =

  struct with fields:

      Chunky: 1
    Separate: 2
```

Use the string value from the `info` structure to obtain the desired value.

```
tags.PlanarConfiguration = ...
    Tiff.PlanarConfiguration.(info8.PlanarConfiguration);
```

Set the ResolutionUnit value in the same manner.

```
tags.ResolutionUnit = Tiff.ResolutionUnit.(info8.ResolutionUnit);
```

The Software tag is not set in the `boston.tif` file. However, `geotiffwrite` will set the `Software` tag by default. To preserve the information, set the value to the empty string which prevents the tag from being written to the file.

```
tags.Software = '';
```

Copy the GeoTIFF information from `boston.tif`.

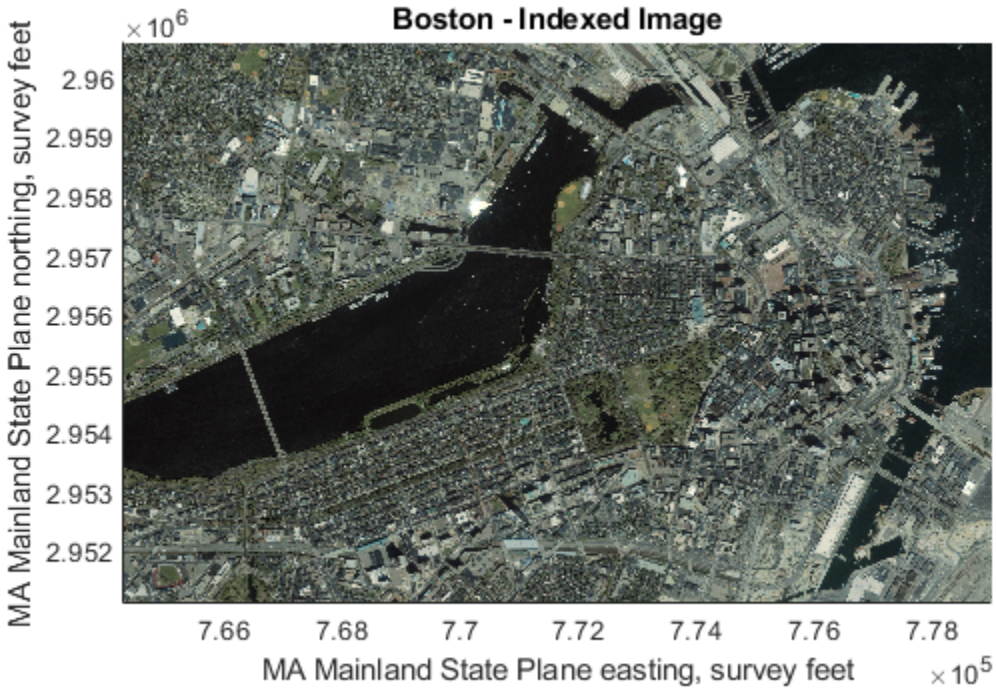
```
geoinfo = geotiffinfo('boston.tif');
key = geoinfo.GeoTIFFTags.GeoKeyDirectoryTag;
```

Write to the GeoTIFF file.

```
filename8 = datafile('boston_indexed.tif');
geotiffwrite(filename8,X8,cmap,R8,'GeoKeyDirectoryTag',key,'TiffTags',tags)
```

View the indexed image.

```
figure
mapshow(filename8)
title('Boston - Indexed Image')
xlabel('MA Mainland State Plane easting, survey feet')
ylabel('MA Mainland State Plane northing, survey feet')
```



Compare the information in the structures that should be equal by printing a table of the values.

```
info_rgb = imfinfo('boston.tif');
info_indexed = imfinfo(filename8);
tagNames = fieldnames(tags);
tagNames(strcmpi('Software', tagNames)) = [];
names = [{'Height' 'Width'}, tagNames'];

spacing = 2;
fieldnameLength = max(cellfun(@length, names)) + spacing;
formatSpec = ['%-' int2str(fieldnameLength) 's'];

fprintf([formatSpec formatSpec formatSpec '\n'], ...
    'Fieldname', 'RGB Information', 'Indexed Information')
fprintf([formatSpec formatSpec formatSpec '\n'], ...
    '---------', '---------------', '--------------------')

for k = 1:length(names)
    fprintf([formatSpec formatSpec formatSpec '\n'], ...
        names{k}, ...
        num2str(info_rgb.(names{k})), ...
        num2str(info_indexed.(names{k})))
end
```

```
Fieldname           RGB Information      Indexed Information
---------           ---------------      --------------------
Height              2881                 2881
Width               4481                 4481
```

```
Compression          Uncompressed         Uncompressed
RowsPerStrip         256                  256
XResolution          300                  300
YResolution          300                  300
ImageDescription     "GeoEye"             "GeoEye"
DateTime             2007:02:23 21:46:13  2007:02:23 21:46:13
Copyright            "(c) GeoEye"         "(c) GeoEye"
Orientation          1                    1
PlanarConfiguration  Chunky               Chunky
ResolutionUnit       Inch                 Inch
```

Compare the information that should be different, since you converted an RGB image to an indexed image, by printing a table of values.

```
names = {'FileSize', 'ColorType', 'BitDepth', ...
    'BitsPerSample', 'PhotometricInterpretation'};

fieldnameLength = max(cellfun(@length, names)) + spacing;
formatSpec = ['%-' int2str(fieldnameLength) 's'];
formatSpec2 = '%-17s';

fprintf(['\n' formatSpec formatSpec2 formatSpec2 '\n'], ...
    'Fieldname', 'RGB Information', 'Indexed Information')
fprintf([formatSpec formatSpec2 formatSpec2 '\n'], ...
    '---------', '---------------', '-------------------')
for k = 1:length(names)
    fprintf([formatSpec formatSpec2 formatSpec2 '\n'], ...
        names{k}, ...
        num2str(info_rgb.(names{k})), ...
        num2str(info_indexed.(names{k})))
end
```

```
Fieldname                 RGB Information  Indexed Information
---------                 ---------------  -------------------
FileSize                  38729900         27925078
ColorType                 truecolor        indexed
BitDepth                  24               16
BitsPerSample             8  8  8          16
PhotometricInterpretation RGB              RGB Palette
```

Cleanup: Remove Data Folder

Remove the temporary folder and data files.

```
rmdir(datadir, 's')
```

Data Set Information

The files `boston.tif` and `boston_ovr.jpg` include materials copyrighted by GeoEye, all rights reserved. GeoEye was merged into the DigitalGlobe corporation on January 29th, 2013. For more information about the data sets, use the commands `type boston.txt` and `type boston_ovr.txt`.

The files `concord_orthow_w.tif` and `concord_ortho_e.tif` are derived using orthophoto tiles from the Bureau of Geographic Information (MassGIS), Commonwealth of Massachusetts, Executive Office of Technology and Security Services. For more information about the data sets, use the command `type concord_ortho.txt`. For an updated link to the data provided by MassGIS, see https://www.mass.gov/service-details/massgis-data-layers.

The DTED file `n39_w106_3arc_v2.dt1` is courtesy of the US Geological Survey.

See Also

`geotiffinfo` | `geotiffwrite` | `getworldfilename` | `worldfileread`

Converting Coastline Data (GSHHG) to Shape le Format

This example shows how to:

- Extract a subset of coastline data from the Global Self-consistent Hierarchical High-resolution Geography (GSHHG) data set
- Manipulate polygon features to add lakes and other interior water bodies as inner polygon rings ("holes")
- Save the modified data set to a shapefile for future use in MATLAB®, or for export to a geographic information system

The Global Self-consistent Hierarchical High-resolution Geography (GSHHG; formerly Global Self-consistent Hierarchical High-resolution Shorelines, or GSHHS) data set, by Paul Wessel and Walter H. F. Smith, provides a consistent set of hierarchically arranged closed polygons. They can be used to construct base maps, or in applications or analyses that involve operations like geographic searches or the statistical properties of coastlines.

Step 1: De ne a Working Folder

This example creates several temporary files and uses the variable `workingFolder` to denote their location. The value used here is determined by the output of the `tempdir` command, but you could easily customize this.

```
workingFolder = tempdir;
```

Step 2: GNU° Unzip and Index the Coarse-Resolution GSHHG Layer

GSHHG is available in wide range of spatial resolutions. This example uses the lowest-resolution data, from the binary file `gshhs_c.b`. A GNU zipped copy of this file is included in the Mapping Toolbox™ data folder, on the MATLAB path.

Use the MATLAB `gunzip` function to decompress `gshhs_c.b.gz` and create the file `gshhs_c.b` in the location indicated by `workingFolder`. Then create an index file, `gshhs_c.i`, in the same folder. In general, having an index file helps to accelerate later calls to the `gshhs` function. Note that when you use the `'createindex'` option, `gshhs` does not extract data.

```
files = gunzip('gshhs_c.b.gz', workingFolder);
filename = files{1};
indexfile = gshhs(filename, 'createindex');
```

Step 3: Import the GSHHG Data for South America

Select data for a specific latitude-longitude quadrangle and import it as a Mapping Toolbox "geostruct" array:

```
latlim = [-60  15];
lonlim = [-90 -30];
S = gshhs(filename, latlim, lonlim);
```

If you have finished extracting data, you can remove the decompressed GSHHS file and the index file.

```
delete(filename)
delete(indexfile)
```

Step 4: Examine the Data Set

Examine the first element of the geostruct array `S`. In addition to the `Lat` and `Lon` coordinate arrays, note the various attribute fields that are present.

```
S(1)
```

```
ans = struct with fields:
           Geometry: 'Polygon'
        BoundingBox: [2x2 double]
                Lat: [1x972 double]
                Lon: [1x972 double]
              South: -53.9004
              North: 71.9942
               West: 191.8947
               East: 325.2054
               Area: 3.7652e+07
              Level: 1
        LevelString: 'land'
          NumPoints: 971
      FormatVersion: 3
             Source: 'WVS'
   CrossesGreenwich: 0
           GSHHS_ID: 1
```

GSHHS comprises four levels of shorelines:

- Level 1 - "Land"
- Level 2 - "Lake"
- Level 3 - "Island in lake"
- Level 4 - "Pond in island in lake"

Check to see which levels the data you've imported includes. The `Level` field contains numerical level numbers.

```
levels = [S.Level];
unique(levels)
```

```
ans = 1×3

     1     2     3
```

The `LevelString` field provides their interpretation. For example,

```
S(104).LevelString
```

```
ans =
'lake'
```

shows that feature 104 is a lake (a Level 2 feature).

In this example, due either to the low resolution or to spatial subsetting, no Level 4 features are present.

Step 5: Extract the Top Two Levels into Separate Geostruct Arrays

This example manipulates the top two levels of the GSHHS hierarchy, inserting each "lake" into the surrounding land mass.

Extract GSHHS Level 1 (exterior coastlines of continents and oceanic islands):

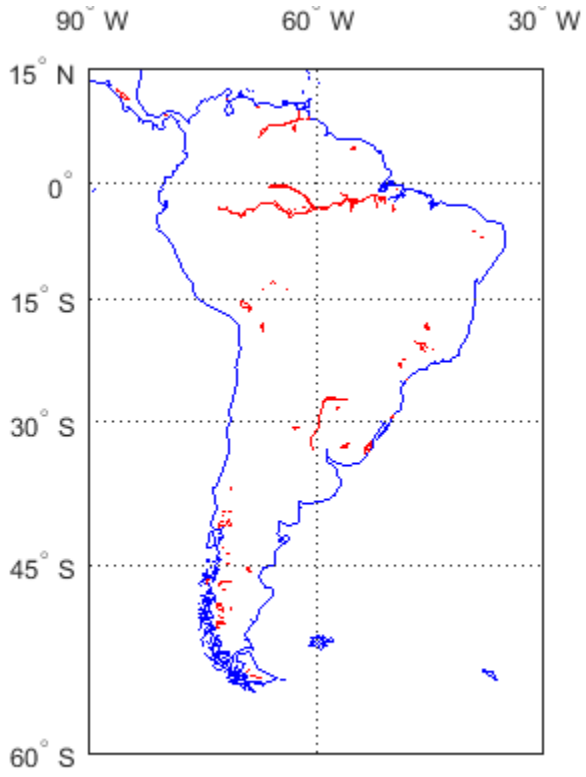
```
L1 = S(levels == 1);
```

Extract Level 2 (coastlines of lakes and seas within Level 1 polygons):

```
L2 = S(levels == 2);
```

To see their spatial relationships, you can map Level 1 edges as blue lines and Level 2 edges as red lines:

```
figure
axesm('mercator', 'MapLatLimit', latlim, 'MapLonLimit', lonlim)
gridm; mlabel; plabel
geoshow([L1.Lat], [L1.Lon], 'Color', 'blue')
geoshow([L2.Lat], [L2.Lon], 'Color', 'red')
tightmap
```



Step 6: Merge Level 2 Polygons into Level 1

Define an anonymous predicate function to detect bounding-box intersections (returning true if a pair of bounding boxes intersect and false otherwise). Inputs A and B are 2-by-2 bounding-box matrices of the form

```
  [min(lon)  min(lat)
   max(lon)  max(lat)].

boxesIntersect = ...
    @(A,B) (~(any(A(2,:) < B(1,:)) || any(B(2,:) < A(1,:))));
```

For convenience in looping over them, copy the Level 1 bounding boxes to a 3-D array:

```
L1boxes = reshape([L1.BoundingBox],[2 2 numel(L1)]);
```

Check each Level 1 - Level 2 pair of features for possible intersection. See if `polybool` returns any output or not, but avoid calling `polybool` unless a bounding box intersection is detected first:

```
for k = 1:numel(L2)
    for j = 1:numel(L1)
        % See if bounding boxes intersect
        if boxesIntersect(L2(k).BoundingBox, L1boxes(:,:,j))
            % See if actual features intersect
            if ~isempty(polybool('intersection', ...
                L2(k).Lon, L2(k).Lat, L1(j).Lon, L1(j).Lat))
                % Reverse level 2 vertex order before merge to
                % correctly orient inner rings
                L1(j).Lon = [L1(j).Lon fliplr(L2(k).Lon) NaN];
                L1(j).Lat = [L1(j).Lat fliplr(L2(k).Lat) NaN];
            end
        end
    end
end
```

Step 7: Save Results in a Shapefile

With a single call to `shapewrite`, you can create a trio of files,

```
gshhs_c_SouthAmerica.shp
gshhs_c_SouthAmerica.shx
gshhs_c_SouthAmerica.dbf
```

in your working folder.

```
shapepath = fullfile(workingFolder,'gshhs_c_SouthAmerica');
shapewrite(L1, shapepath)
```

Step 8: Validate the Shapefile

To validate the results of `shapewrite`, read the new shapefile into the geostruct array `southAmerica`:

```
southAmerica = shaperead(shapepath, 'UseGeoCoords', true)
```

```
southAmerica=79×1 struct array with fields:
    Geometry
    BoundingBox
    Lon
    Lat
    South
    North
    West
    East
    Area
```

```
    Level
    LevelString
    NumPoints
    FormatVersi
    Source
    CrossesGree
    GSHHS_ID
```

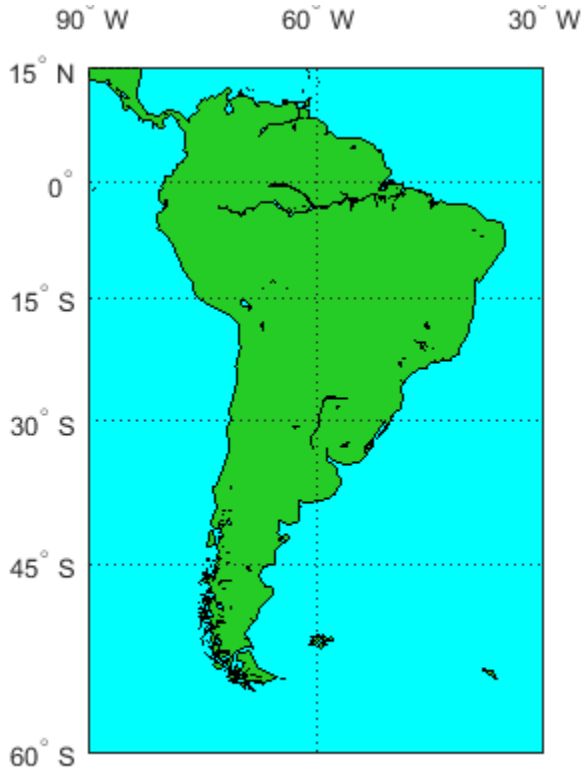
Note that the two longest fieldnames, `'FormatVersion'` and `'CrossesGreenwich'`, have been truncated to 11 characters. This happened during the call to `shapewrite` and is unavoidable because of a rigid 11-character limit in the xBASE tables (.DBF format) used to store attributes in shapefiles. (In general, when writing shapefiles you may want to re-define fieldnames longer than 11 characters in order to avoid or control the effects of automatic truncation.)

Optionally, remove the new shapefiles from your working folder. (This example needs to clean up after itself; in a real application you would probably want to omit this step.)

```
delete([shapepath '.*'])
```

Display the geostruct imported from the new shapefile. Note the various "holes" in the South America polygon indicating lakes and shorelines of other extended bodies of water in the interior of the continent.

```
figure
ax = axesm('mercator', 'MapLatLimit', latlim, 'MapLonLimit', lonlim);
ax.Color = 'cyan';
gridm; mlabel; plabel
geoshow(southAmerica, 'FaceColor', [0.15 0.8 0.15])
tightmap
```



Reference

Wessel, P., and W. H. F. Smith, 1996, A global self-consistent, hierarchical, high-resolution shoreline database, Journal of Geophysical Research, Vol. 101, pp. 8741-8743.

Additional Data

The complete GSHHG data set may be downloaded from the U.S. National Oceanic and Atmospheric Administration (NOAA) web site. Follow the links from

https://www.mathworks.com/help/map/finding-geospatial-data.html

Credits

The GSHHG data file is provided in the Mapping Toolbox courtesy of Dr. Paul Wessel of the University of Hawaii and Dr. Walter H. F. Smith of NOAA.

For more information, run:

```
>> type gshhs_c.txt
```

See Also

`gshhs` | `shaperead` | `shapewrite`

Understanding Geospatial Geometry

- "The Shape of the Earth" on page 3-2
- "Reference Spheroids" on page 3-4
- "Work with Reference Spheroids" on page 3-11
- "Latitude and Longitude" on page 3-13
- "Relationship Between Points on Sphere" on page 3-15
- "Length and Distance Units" on page 3-16
- "Compute Conversion Ratio Between Units of Length" on page 3-17
- "Angle Representations and Angular Units" on page 3-18
- "Angles as Binary and Formatted Numbers" on page 3-22
- "Convert from Linear Measurements to Spherical Measurements" on page 3-23
- "Distances on the Sphere" on page 3-24
- "Great Circles" on page 3-27
- "Rhumb Lines" on page 3-28
- "Azimuth" on page 3-29
- "Elevation" on page 3-31
- "Generate Vector Data for Points Along Great Circle or Rhumb Line Tracks" on page 3-32
- "Reckoning" on page 3-34
- "Calculate Distance Between Two Points in Geographic Space" on page 3-35
- "Small Circles" on page 3-36
- "Calculate Vector Data for Points Along a Small Circle" on page 3-37
- "Generate Small Circles" on page 3-38
- "Measure Area of Spherical Quadrangles" on page 3-40
- "Plotting a 3-D Dome as a Mesh Over a Globe" on page 3-41
- "Choose a 3-D Coordinate System" on page 3-47
- "Vectors in 3-D Coordinate Systems" on page 3-52
- "Find Ellipsoidal Height from Orthometric Height" on page 3-55

The Shape of the Earth

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

Ellipsoid Shape

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

The toolbox includes ellipsoid models that represent the figures of the Sun, Moon, and planets, as well as a set of the most common ellipsoid models of the Earth. For more information, see “Reference Spheroids” on page 3-4.

Geoid Shape

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 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 1 percent of the differences between the semimajor and semiminor ellipsoid axes used to approximate the Earth's shape.

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.

Map the Geoid

Get geoid heights and a geographic postings reference object from the EGM96 geoid model. Load coastline latitude and longitude data.

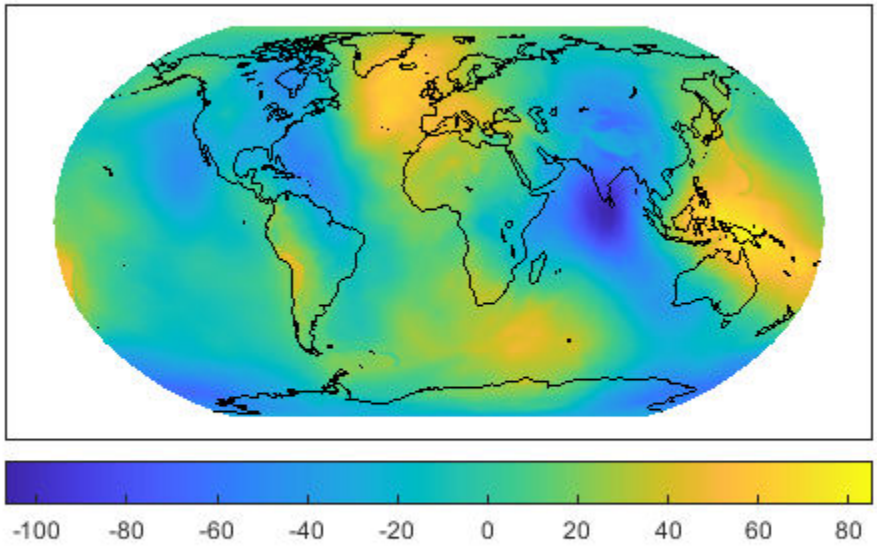
```
[N,R] = egm96geoid;
load coastlines
```

Display the geoid heights as a surface using a Robinson projection. Ensure the coastline data appears over the surface by setting the `'CData'` name-value pair to the geoid heights data and the `'ZData'` name-value pair to a matrix of zeros. Then, display the coastline data.

```
axesm robinson
Z = zeros(R.RasterSize);
geoshow(N,R,'DisplayType','surface','CData',N,'ZData',Z)
geoshow(coastlat,coastlon,'color','k')
```

Display a colorbar below the map.

```
colorbar('southoutside')
```



Reference Spheroids

When the Earth (or another roughly spherical body such as the Moon) is modeled as a sphere having a standard radius, it is called a *reference sphere*. Likewise, when the model is a flattened (oblate) ellipsoid of revolution, with a standard semimajor axis and standard inverse flattening, semiminor axis, or eccentricity, it is called a *reference ellipsoid*. Both models are spheroidal in shape, so each can be considered to be a type of *reference spheroid*. Mapping Toolbox supports several representations for reference spheroids: `referenceSphere`, `referenceEllipsoid`, and `oblateSpheroid` objects, and an older representation, *ellipsoid vector*.

In this section...

“referenceSphere Objects” on page 3-4
“referenceEllipsoid Objects” on page 3-6
“World Geodetic System 1984” on page 3-8
“Ellipsoid Vectors” on page 3-9
“oblateSpheroid Objects” on page 3-10

referenceSphere Objects

When using a strictly spherical model, you should generally use a `referenceSphere` object (although both `referenceEllipsoid` and `oblateSpheroid` can represent a perfect sphere).

By default, `referenceSphere` returns a dimensionless unit sphere:

```
referenceSphere
```

```
ans =

referenceSphere with defining properties:

          Name: 'Unit Sphere'
    LengthUnit: ''
        Radius: 1

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

You can request a specific body by name, and the radius will be in meters by default:

```
earth = referenceSphere('Earth')
```

```
earth =

referenceSphere with defining properties:
```

```
          Name: 'Earth'
    LengthUnit: 'meter'
        Radius: 6371000

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

You can reset the length unit if desired (and the radius is rescaled appropriately) :

```
earth.LengthUnit = 'kilometer'
```

```
earth =

referenceSphere with defining properties:

          Name: 'Earth'
    LengthUnit: 'kilometer'
        Radius: 6371

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

or specify the length unit at the time of construction:

```
referenceSphere('Earth','km')
```

```
ans =

referenceSphere with defining properties:

          Name: 'Earth'
    LengthUnit: 'kilometer'
        Radius: 6371

  and additional properties:

    SemimajorAxis
    SemiminorAxis
    InverseFlattening
    Eccentricity
```

```
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Any length unit supported by `validateLengthUnit` can be used. A variety of abbreviations are supported for most length units, see `validateLengthUnit` for a complete list.

One thing to note about `referenceSphere` is that only the defining properties are displayed, in order to reduce clutter at the command line. (This approach saves a small amount of computation as well.) In particular, don't overlook the dependent `SurfaceArea` and `Volume` properties, even though they are not displayed. The surface area of the spherical earth model, for example, is easily obtained through the `SurfaceArea` property:

```
earth.SurfaceArea
```

```
ans =
   5.1006e+08
```

This result is in square kilometers, because the `LengthUnit` property of the object earth has value `'kilometer'`.

When programming with Mapping Toolbox it may help to be aware that `referenceSphere` actually includes all the geometric properties of `referenceEllipsoid`and `oblateSpheroid` (`SemimajorAxis`, `SemiminorAxis`, `InverseFlattening`, `Eccentricity`, `Flattening`, `ThirdFlattening`, and `MeanRadius`, as well as `SurfaceArea`, and `Volume`). None of these properties can be set on a `referenceSphere`, and some have values that are fixed for all spheres. `Eccentricity` is always `0`, for example. But they provide a flexible environment for programming because any geometric computation that accepts a `referenceEllipsoid` will also run properly given a `referenceSphere`. This is a type of polymorphism in which different classes support common, or strongly overlapping interfaces.

referenceEllipsoid Objects

When using an oblate spheroid to represent the Earth (or another roughly spherical body), you should generally use a `referenceEllipsoid` object. An important exception occurs with certain small-scale map projections, many of which are defined only on the sphere. However, all important projections used for large-scale work, including Transverse Mercator and Lambert Conformal Conic, are defined on the ellipsoid as well as the sphere.

Like `referenceSphere`, `referenceEllipsoid` returns a dimensionless unit sphere by default:

```
referenceEllipsoid
```

```
ans =

referenceEllipsoid with defining properties:

                Code: []
                Name: 'Unit Sphere'
          LengthUnit: ''
        SemimajorAxis: 1
        SemiminorAxis: 1
    InverseFlattening: Inf
         Eccentricity: 0
```

```
  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

More typically, you would request a specific ellipsoid by name, resulting in an object with semimajor and semiminor axes properties in meters. For example, the following returns a `referenceEllipsoid` with `SemimajorAxis` and `InverseFlattening` property settings that match the defining parameters of Geodetic Reference System 1980 (GRS 80).

```
grs80 = referenceEllipsoid('Geodetic Reference System 1980')

grs80 =

referenceEllipsoid with defining properties:

                Code: 7019
                Name: 'Geodetic Reference System 1980'
          LengthUnit: 'meter'
        SemimajorAxis: 6378137
        SemiminorAxis: 6356752.31414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

In general, you should use the reference ellipsoid corresponding to the geodetic datum to which the coordinates of your data are referenced. For instance, the GRS 80 ellipsoid is specified for use with coordinates referenced to the North American Datum of 1983 (NAD 83).

As in the case of `referenceSphere`, you can reset the length unit if desired:

```
grs80.LengthUnit = 'kilometer'

grs80 =

referenceEllipsoid with defining properties:

                Code: 7019
                Name: 'Geodetic Reference System 1980'
          LengthUnit: 'kilometer'
        SemimajorAxis: 6378.137
        SemiminorAxis: 6356.75231414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:
```

```
    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

or specify the length unit at the time of construction:

```
referenceEllipsoid('Geodetic Reference System 1980','km')

ans =

referenceEllipsoid with defining properties:

                Code: 7019
                Name: 'Geodetic Reference System 1980'
          LengthUnit: 'kilometer'
        SemimajorAxis: 6378.137
        SemiminorAxis: 6356.75231414036
    InverseFlattening: 298.257222101
         Eccentricity: 0.0818191910428158

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

Any length unit supported by `validateLengthUnit` can be used.

The command-line display includes four geometric properties: `SemimajorAxis`, `SemiminorAxis`, `InverseFlattening`, and `Eccentricity`. Any pair of these properties, as long as at least one is an axis length, is sufficient to fully define a oblate spheroid; the four properties constitute a mutually dependent set. Parameters `InverseFlattening` and `Eccentricity` as a set are not sufficient to define an ellipsoid because both are dimensionless shape properties. Neither of those parameters provides a length scale, and, furthermore, are mutually dependent: `ecc = sqrt((2 - f) * f)`.

In addition, there are five dependent properties that are not displayed, in order to reduce clutter on the command line: `Flattening`, `ThirdFlattening`, `MeanRadius`, `SurfaceArea`, and `Volume`. `SurfaceArea` and `Volume` work the same way as their `referenceSphere` counterparts. To continue the preceding example, the surface area of the GRS 80 ellipsoid in square kilometers (because `LengthUnit` is `'kilometer'`), is easily obtained as follows:

```
grs80.SurfaceArea

ans =
   5.1007e+08
```

See the `referenceEllipsoid` reference page for definitions of the shape properties, permissible values for the `Name` property, and information on the `Code` property.

World Geodetic System 1984

Due in part to widespread use of the U.S. NAVSTAR Global Positioning System (GPS), which is tied to World Geodetic System 1984 (WGS 84), the WGS 84 reference ellipsoid is often the appropriate

choice. For both convenience and speed (obtained by bypassing a table look-up step), it's a good idea in this case to use the `wgs84Ellipsoid` function, for example,

```
wgs84 = wgs84Ellipsoid;
```

The preceding line is equivalent to:

```
wgs84 = referenceEllipsoid('wgs84');
```

but it is easier to type and faster to run. You can also specify a length unit. `wgs84Ellipsoid(lengthUnit)`, is equivalent to `referenceEllipsoid('wgs84',lengthUnit)`, where `lengthUnit` is any unit value accepted by the `validateLengthUnit` function.

For example, the follow two commands show that the surface area of the WGS 84 ellipsoid is a little over 5 x 10^14 square meters:

```
s = wgs84Ellipsoid
```

```
s =

referenceEllipsoid with defining properties:

                Code: 7030
                Name: 'World Geodetic System 1984'
          LengthUnit: 'meter'
        SemimajorAxis: 6378137
        SemiminorAxis: 6356752.31424518
    InverseFlattening: 298.257223563
         Eccentricity: 0.0818191908426215

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

```
s.SurfaceArea
```

```
ans =

   5.1007e+14
```

Ellipsoid Vectors

An ellipsoid vector is simply a 2-by-1 double of the form: `[semimajor_axis eccentricity]`. Unlike a spheroid object (any instance of `referenceSphere`, `referenceEllipsoid`, or `oblateSpheroid`), an ellipsoid vector is not self-documenting. Ellipsoid vectors are not even self-identifying. You have to know that a given 2-by-1 vector is indeed an ellipsoid vector to make any use of it. This representation does not validate that `semimajor_axis` is real and positive, for example, you have to do such validations for yourself.

Many toolbox functions accept ellipsoid vectors as input, but such functions accept spheroid objects as well and, for the reasons just stated, spheroid objects are recommended over ellipsoid vectors. In case you have written a function of your own that requires an ellipsoid vector as input, or have

received such a function from someone else, note that you can easily convert any spheroid object `s` into an ellipsoid vector as follows:

```
[s.SemimajorAxis s.Eccentricity]
```

This means that you can construct a spheroid object using any of the three class constructors, or the `wgs84Ellipsoid` function, and hand off the result in the form of an ellipsoid vector if necessary.

oblateSpheroid Objects

`oblateSpheroid` is the superclass of `referenceEllipsoid`. An `oblateSpheroid` object is just like a `referenceEllipsoid` object minus its `Code`, `Name`, and `LengthUnit` properties. In fact, the primary role of the `oblateSpheroid` class is to provide the purely geometric properties and behaviors needed by the `referenceEllipsoid` class.

For most purposes, you can simply ignore this distinction, and the `oblateSpheroid` class itself, as a matter of internal software composition. No harm will come about, because a `referenceEllipsoid` object can do anything and be used anywhere that an `oblateSpheroid` can.

However, you can use `oblateSpheroid` directly when dealing with an ellipsoid vector that lacks a specified name or length unit. For example, compute the volume of a ellipsoid with a semimajor axis of 2000 and eccentricity of 0.1, as shown in the following.

```
e = [2000 0.1];
s = oblateSpheroid;
s.SemimajorAxis = e(1);
s.Eccentricity = e(2)
s.Volume
```

```
s =

oblateSpheroid with defining properties:

        SemimajorAxis: 2000
        SemiminorAxis: 1989.97487421324
    InverseFlattening: 199.498743710662
         Eccentricity: 0.1

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume


ans =

   3.3342e+10
```

Of course, since the length unit of `e` is unspecified, the unit of `s.Volume` is likewise unspecified.

Work with Reference Spheroids

Reference spheroids are needed in three main contexts: map projections, curves and areas on the surface of a spheroid, and 3-D computations involving geodetic coordinates.

Map Projections

You can set the value of the `Geoid` property of a new map axes (which is actually a Spheroid property) using any type of reference spheroid representation when constructing the map axes with `axesm`. Except in the case of UTM and UPS, the default value is an ellipsoid vector representing the unit sphere: `[1 0]`. It is also the default value when using the `worldmap` and `usamap` functions.

You can reset the `Geoid` property of an existing map axes to any type of reference spheroid representation by using `setm`. For example, `worldmap` always sets up a projection based on the unit sphere but you can subsequently use `setm` to switch to the spheroid of your choice. To set up a map of North America for use with Geodetic Reference System 1980, for instance, follow `worldmap` with a call to `setm`, like this:

```
ax = worldmap('North America');
setm(ax,'geoid',referenceEllipsoid('grs80'))
```

When projecting or unprojecting data without a map axes, you can set the `geoid` field of a map projection structure (`mstruct`) to any type of reference spheroid representation. Remember to follow all `mstruct` updates with a second call to `defaultm` to ensure that all properties are set to legitimate values. For example, to use the Miller projection with WGS 84 in kilometers, start with:

```
mstruct = defaultm('miller');
mstruct.geoid = wgs84Ellipsoid('km');
mstruct = defaultm(mstruct);
```

You can inspect the `mstruct` to ensure that you are indeed using the WGS 84 ellipsoid:

```
mstruct.geoid
```

```
ans =

referenceEllipsoid with defining properties:

                Code: 7030
                Name: 'World Geodetic System 1984'
          LengthUnit: 'kilometer'
        SemimajorAxis: 6378.137
        SemiminorAxis: 6356.75231424518
    InverseFlattening: 298.257223563
         Eccentricity: 0.0818191908426215

  and additional properties:

    Flattening
    ThirdFlattening
    MeanRadius
    SurfaceArea
    Volume
```

See Map Axes Properties for definitions of the fields found in `mstructs`.

Curves and Areas

Another important context in which reference spheroids appear is the computation of curves and areas on the surface of a sphere or oblate spheroid. The `distance` function, for example, assumes a sphere by default, but accepts a reference spheroid as an optional input. `distance` is used to compute the length of the geodesic or rhumb line arc between a pair of points with given latitudes and longitudes. If a reference spheroid is provided through the `ellipsoid` argument, then the unit used for the arc length output matches the `LengthUnit` property of the spheroid.

Other functions for working with curves and areas that accept reference spheroids include `reckon`, `scircle1`, `scircle2`, `ellipse1`, `track1`, `track2`, and `areaquad`, to name just a few. When using such functions without their `ellipsoid` argument, be sure to check the individual function help if you are unsure about which reference spheroid is assumed by default.

3-D Coordinate Transformations

The third context in which reference spheroids frequently appear is the transformation of geodetic coordinates (latitude, longitude, and height above the ellipsoid) to other coordinate systems. For example, the `geodetic2ecef` function, which converts point locations from a geodetic system to a geocentric (Earth-Centered Earth-Fixed) Cartesian system, requires a reference spheroid object (or an ellipsoid vector) as input. And the `elevation` function, which converts from geodetic to a local spherical system (azimuth, elevation, and slant range) also accepts a reference spheroid object or ellipsoid vector, but uses the GRS 80 ellipsoid by default if none is provided.

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° at the north pole, and negative in the southern hemisphere, reaching a limit of -90° at the south pole. Lines of constant latitude are called `parallels`.

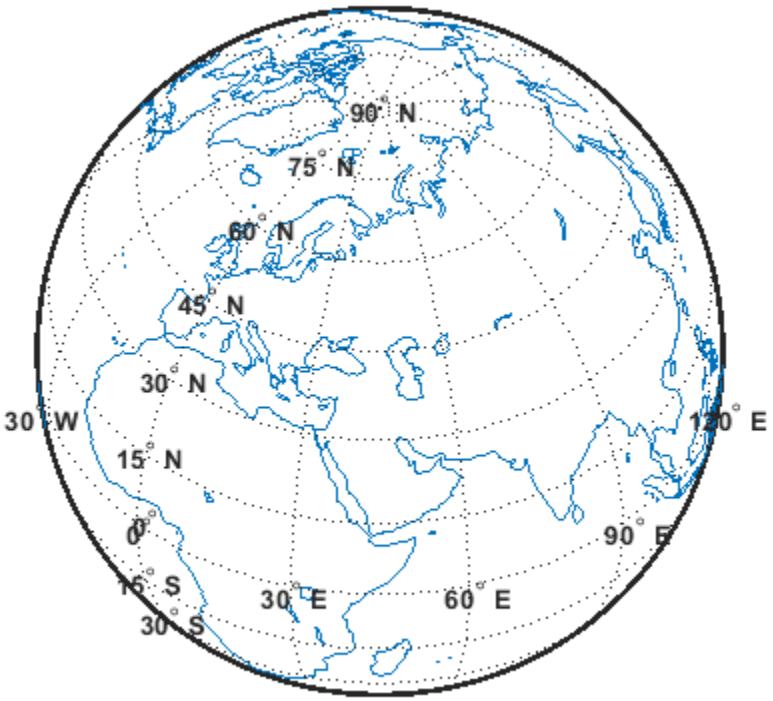
Longitude is the angle at the center of the planet between two planes that align with and intersect along the axis of rotation, perpendicular to the plane of the equator. One plane passes through the surface point in question, and the other plane is the *prime meridian* (0° longitude), which is defined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

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

Plot Latitude and Longitude

This example shows how to plot latitude and longitude.

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


90° N
75° N
60° N
45° N
30° N
15° N
0°
15° S
30° S
30° W
30° E
60° E
90° E
120° E

Relationship Between Points on Sphere

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 “Distances on the Sphere” on page 3-24, you can convert angular distance on a sphere to linear distance. This is different from working on an ellipsoid, where one can only speak of linear distances between points, and to compute them one must specify which reference ellipsoid to use.

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

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

The first three are angular quantities, while the last is a length. Mapping Toolbox functions exist for computing these quantities. For additional examples, see “Navigation” on page 10-9.

There is no single default unit of distance measurement in the toolbox. Navigation functions use nautical miles as a default 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.

Length and Distance Units

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

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

Length units are needed to describe

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

Choosing Units of Length

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

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

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

Converting Units of Length

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

- `unitsratio` computes multiplicative factors for converting between 12 different units of length as well as between degrees and radians. You can use `unitsratio` 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 3-20 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 3-25.

Compute Conversion Ratio Between Units of Length

This example shows how to use the `unitsratio` function to create a conversion factor for many different units of length, such as microns, millimeters, inches, international feet, and U.S. survey feet. The `unitsratio` function also lets you convert angles between degrees and radians. For more information, see `unitsratio`.

Create a conversion factor for inches to centimeters and convert 4 inches into centimeters.

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

```
cmPerInch = 2.5400
```

```
cm = cmPerInch * 4
```

```
cm = 10.1600
```

Create the inverse conversion factor and multiply it by the `cmPerInch` conversion factor.

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

```
inch = 1
```

Angle Representations and Angular Units

In this section...

“Radians and Degrees” on page 3-18
“Default and Variable Angle Units” on page 3-19
“Degrees, Minutes, and Seconds” on page 3-19
“Converting Angle Units that Vary at Run Time” on page 3-20

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 `deg2rad` for this purpose:

```
thetaInRadians = deg2rad(thetaInDegrees)
```

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

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

or by using `rad2deg`,

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

The practice of using these functions has two significant advantages:

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

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

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

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

Degrees, Minutes, and Seconds

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

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

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

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

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

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

```
degrees2dm(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.

The toolbox makes no internal use of DM or DMS representation. The conversion functions `dm2degrees` and `dms2degrees` are provided only as tools for data import. Likewise, `degrees2dm` and `degrees2dms` are only useful for displaying geographic coordinates on maps, publishing coordinate values, and for formatting data to be exported to other applications. Methods for accomplishing this are discussed below, in “Formatting Latitudes and Longitudes” on page 3-22.

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? The toolbox provides a set of utility functions that help you deal with such situations at run time.

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

- `fromDegrees`
- `toDegrees`
- `fromRadians`
- `toRadians`

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

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

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

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

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

```
piInDegrees =

   180
```

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 character vector, it is often helpful to describe that value 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 character vector 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 character vector, as in

```
num2str(degrees2dm(rad2deg(1)))

ans =
57      17.7468
```

Formatting Latitudes and Longitudes

When a DM or DMS representation of an angle is expressed as a character vector, 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 character vectors. 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 3-19 for more details about DM and DMS representation.

Convert from Linear Measurements to Spherical Measurements

This example shows how to 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 or radians.

Convert a degree of arc length at the Earth's equator to nautical miles.

```
nauticalmiles = deg2nm(1)
```

```
nauticalmiles = 60.0405
```

Specify the radius to use in the conversion calculation. The default value assumes the Earth's radius.

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

```
nauticalmiles = 30.3338
```

Return the distance in statute miles rather than nautical miles.

```
deg2sm(1)
```

```
ans = 69.0932
```

Distances on the Sphere

In this section...

"Arc Length as an Angle in the distance and reckon Functions" on page 3-25
"Summary: Available Distance and Angle Conversion Functions" on page 3-25

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 software includes a set of six functions to conveniently convert distances along the surface of the Earth (or another planet) from units of kilometers (km), nautical miles (nm), or 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.

Arc Length 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, `arclen`, is in degrees (along with all the other inputs and outputs):

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

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

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

If an `ellipsoid` argument is provided, however, then `arclen` 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), `arclen` can be considered to be 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 `arclen` will appear to be in radians:

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

Summary: Available Distance and Angle Conversion Functions

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

Functions that Directly Convert Angles, Lengths, and Spherical Distances

Convert	To Degrees	To Radians	To Kilometers	To Nautical Miles	To Statute Miles

Degrees	`toDegrees`
	`fromDegrees`
		`deg2rad`
		`toRadians`
		`fromDegrees`
			`deg2km`	`deg2nm`	`deg2sm`

Radians	`rad2deg`
	`toDegrees`
	`fromRadians`
		`toRadians`
		`fromRadians`
			`rad2km`	`rad2nm`	`rad2sm`

Kilometers	`km2deg`	`km2rad`		`km2nm`	`km2sm`
Nautical Miles	`nm2deg`	`nm2rad`	`nm2km`		`nm2sm`
Statute Miles	`sm2deg`	`sm2rad`	`sm2km`	`sm2nm`	

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

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.

For more information, see “Generate Vector Data for Points Along Great Circle or Rhumb Line Tracks” on page 3-32.

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°. 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. For information about how to calculate points along great circles and rhumb lines, see “Generate Vector Data for Points Along Great Circle or Rhumb Line Tracks” on page 3-32.

Azimuth

Azimuth is the angle a line makes with a meridian, measured clockwise from north. Thus the azimuth of due north is 0°, due east is 90°, due south is 180°, and due west is 270°. You can instruct several Mapping Toolbox functions to compute azimuths for any pair of point locations, either along rhumb lines or along great circles. These will have different results except along cardinal directions. For great circles, the result is the azimuth at the initial point of the pair defining a great circle path. This is because great circle azimuths other than 0°, 90°, 180°, and 270° 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°) mod 360°). For great circles, the back azimuth is generally not the complement, and the difference depends on the distance between the two points.

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

For more an example that uses azimuths, see “Reckoning” on page 3-34

Calculate Azimuth

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

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

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

azgc =
      19.0391
```

For the rhumb line, the constant azimuth is

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

azrh =
      58.8595
```

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

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?

```
[azim, elevang, slantrange] = geodetic2aer( ...
         0, km2deg(10), 10000, 0, 0, 0, referenceEllipsoid('grs 80'))

azim =

    90


elevang =

   44.9005


slantrange =

   1.4156e+04
```

On an ellipsoid, azimuths returned from `geodetic2aer` generally will differ from those returned by `azimuth` and `distance`.

See Also

`geodetic2aer`

Generate Vector Data for Points Along Great Circle or Rhumb Line Tracks

You can generate vector data corresponding to points along great circle or rhumb line tracks using the `track1` and `track2` functions. 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]);
```

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. For more information about rhumb lines, see “Rhumb Lines” on page 3-28. For more information about great circles, see “Great Circles” on page 3-27. More vector path calculations are described in “Navigation” on page 10-9.

Reckoning

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

As an example, an airplane takes off from La Guardia Airport in New York (40.75°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.

See Also

More About

- “Rhumb Lines” on page 3-28
- “Great Circles” on page 3-27

Calculate Distance Between Two Points in Geographic Space

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

Small Circles

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

Small circles are most easily defined by distance from a point. *All points 45 nm (nautical miles) distant from (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 functions also can calculate *loxodromic small circles*, for which distances are measured in a rhumb line sense (along lines of constant azimuth). Do not confuse such figures with true small circles.

To learn how to compute small circles, see “Calculate Vector Data for Points Along a Small Circle” on page 3-37.

Calculate Vector Data for Points Along a Small Circle

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° in distance and between 30° and 120° in azimuth from (67°N,135°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. For more information about small circles, see “Small Circles” on page 3-36.

Generate Small Circles

Generate a true small circle, a loxodromic small circle, and the limiting case of a great circle.

Display the map axes with an orthographic projection.

```
figure;
axesm ortho; gridm on; framem on
setm(gca,'Origin', [45 30 30], 'MLineLimit', [75 -75],...
'MLineException',[0 90 180 270])
```

Define three center points on the sphere.

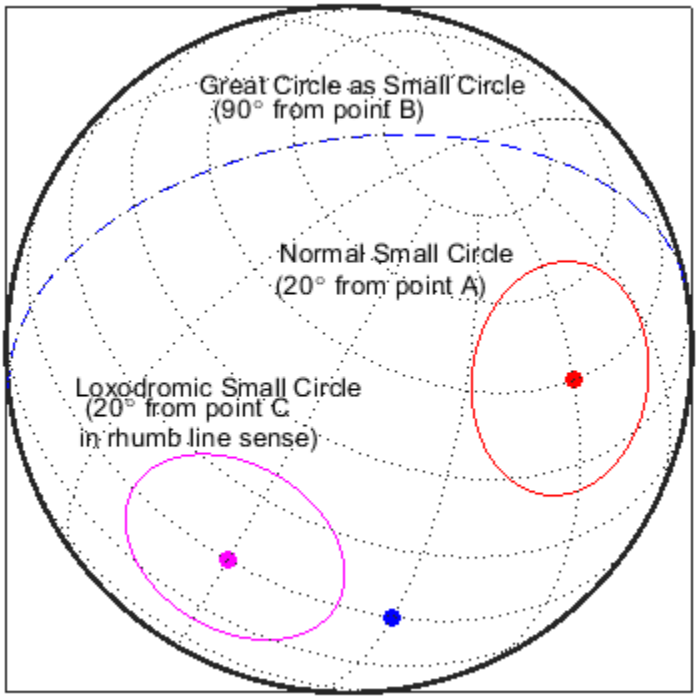
```
A = [45 90];
B = [0 60];
C = [0 30];
```

Create the three small circles around the three center points. The first, `sca`, is a true small circle. The second, `scb`, is a loxodromic small circle. The third, `scc`, is a great circle as the limiting case of a small circle.

```
sca = scircle1(A(1), A(2), 20);
scb = scircle2(B(1), B(2), 0, 150);
scc = scircle1('rh',C(1), C(2), 20);
```

Display the points and their corresponding small circles with different colors. Label the small circles.

```
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)')
```



See Also

`scircle1` | `scircle2`

More About

- “Small Circles” on page 3-36

Measure 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:

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

area =
      0.0187
```

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

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

area =
     3.6788e+06
```

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

Plotting a 3-D Dome as a Mesh Over a Globe

This example shows how to start with a 3-D feature in a system of local east-north-up (ENU) coordinates, then transform and combine it with a globe display in Earth-Centered, Earth-Fixed (ECEF) coordinates.

Step 1: Set De ning Parameters

Use Geodetic Reference System 1980 (GRS80) and work in units of kilometers. Place the origin of the local system near Washington, DC, USA.

```
grs80 = referenceEllipsoid('grs80','km');
domeRadius =  3000;  % km
domeLat =  39;       % degrees
domeLon = -77;       % degrees
domeAlt = 0;         % km
```

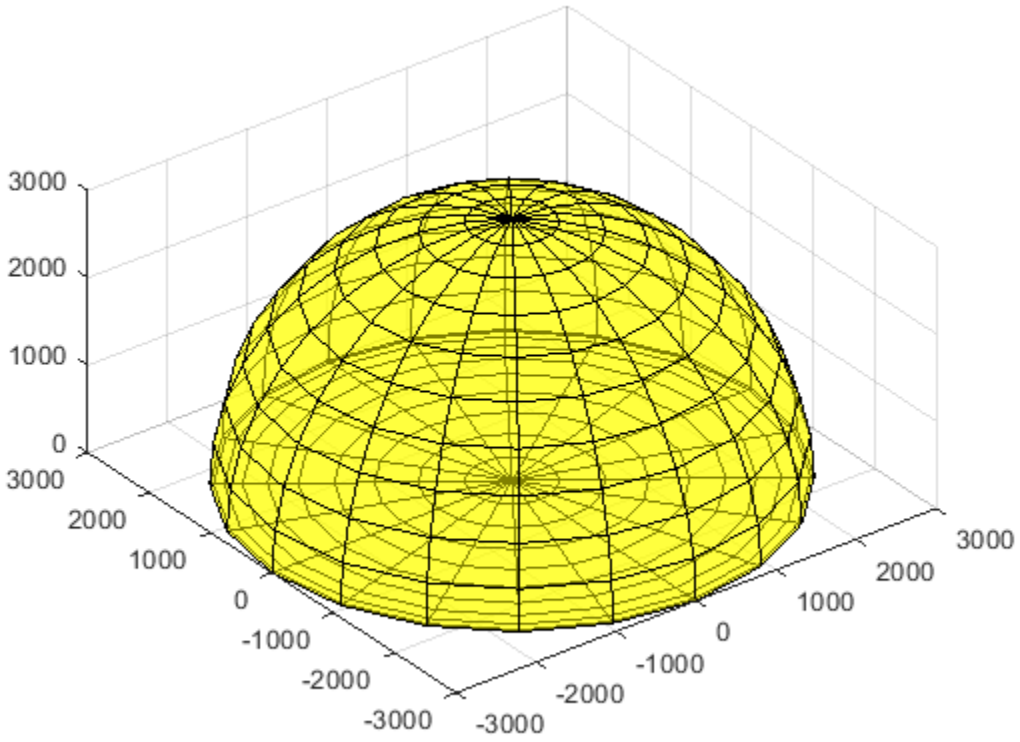
Step 2: Construct the Dome in Local East-North-Up Coordinates

The local ENU system is defined with respect to a geodetic reference point, specified in this case by (`domeLat`, `domeLon`, and `domeAlt`). It is a 3-D Cartesian system in which the positive x-axis is directed to the east, the positive y-axis is directed to the north, and the z-axis is normal to the reference ellipsoid and directed upward.

In this example, the 3-D feature is a hemisphere in the z >= 0 half-space with a radius of 3000 kilometers. This hemisphere could enclose, hypothetically, the volume of space within range of a idealized radar system having uniform coverage from the horizon to the zenith, in all azimuths. Volumes of space such as this, when representing zones of effective surveillance coverage, are sometimes known informally as "radar domes."

A quick way to construct coordinate arrays outlining a closed hemispheric dome is to start with a unit sphere, scale up the radius, and collapse the lower hemisphere. It's easier to visualize if you make it semitransparent -- setting the `FaceAlpha` to 0.5 in this case.

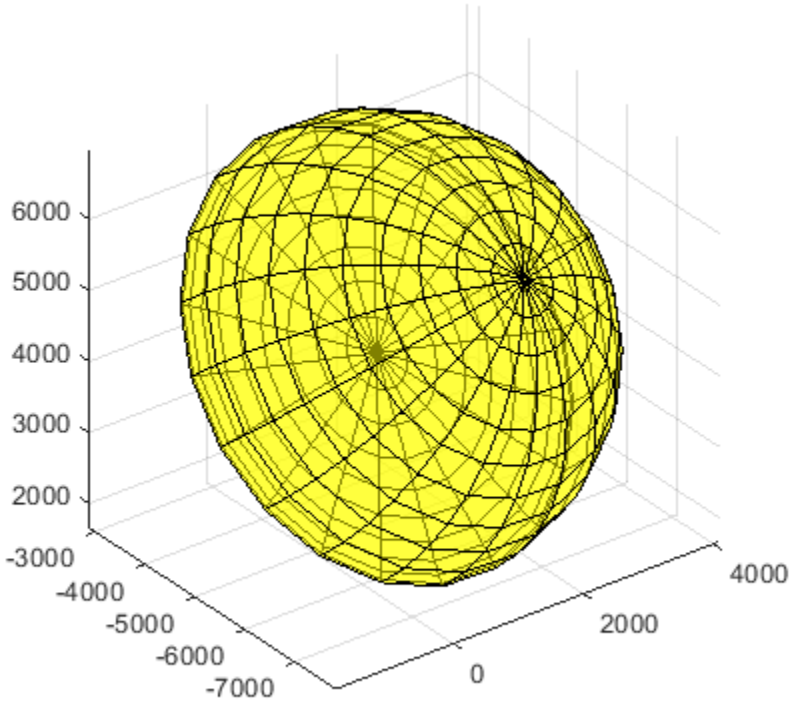
```
[x,y,z] = sphere(20);
xEast  = domeRadius * x;
yNorth = domeRadius * y;
zUp    = domeRadius * z;
zUp(zUp < 0) = 0;
figure('Renderer','opengl')
surf(xEast, yNorth, zUp,'FaceColor','yellow','FaceAlpha',0.5)
axis equal
```



Step 3: Convert Dome to the Earth-Centered Earth-Fixed (ECEF) System

Use the `enu2ecef` function to convert the dome from local ENU to an ECEF system, based on the GRS 80 reference ellipsoid. It applies a 3-D translation and rotation. Notice how the hemisphere becomes tilted and how its center moves thousands of kilometers from the origin.

```
[xECEF, yECEF, zECEF] ...
    = enu2ecef(xEast, yNorth, zUp, domeLat, domeLon, domeAlt, grs80);
surf(xECEF, yECEF, zECEF,'FaceColor','yellow','FaceAlpha',0.5)
axis equal
```



Step 4: Construct a Globe Display

Construct a basic globe display using `axesm` and `globe`.

```
figure('Renderer','opengl')
ax = axesm('globe','Geoid',grs80,'Grid','on', ...
    'GLineWidth',1,'GLineStyle','-',...
    'Gcolor',[0.9 0.9 0.1],'Galtitude',100);
ax.Position = [0 0 1 1];
axis equal off
view(3)
```

Step 5: Add Various Global Map Data

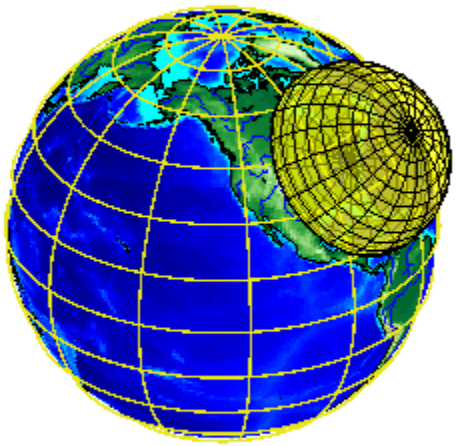
Add low-resolution global topography, coastlines, and rivers to the globe.

```
load topo60c
geoshow(topo60c,topo60cR,'DisplayType','texturemap')
demcmap(topo60c)
land = shaperead('landareas','UseGeoCoords',true);
plotm([land.Lat],[land.Lon],'Color','black')
rivers = shaperead('worldrivers','UseGeoCoords',true);
plotm([rivers.Lat],[rivers.Lon],'Color','blue')
```

Step 6: Add the Dome to the Globe Display

Add the ECEF version of dome to the globe axes as a semitransparent mesh.

```
surf(xECEF, yECEF, zECEF,'FaceColor','yellow','FaceAlpha',0.5)
```



You can view the dome and globe from different angles by interactively rotating the axes in the MATLAB® figure.

Credit

Thanks to Edward J. Mayhew, Jr. for providing technical background on "radar domes" and for bringing to our attention the problem of visualizing them with the Mapping Toolbox™.

Choose a 3-D Coordinate System

Coordinate systems represent position on the Earth using coordinates. Mapping Toolbox functions transform coordinates between Earth-centered Earth-fixed (ECEF), geodetic, east-north-up (ENU), north-east-down (NED), and azimuth-elevation-range (AER) systems.

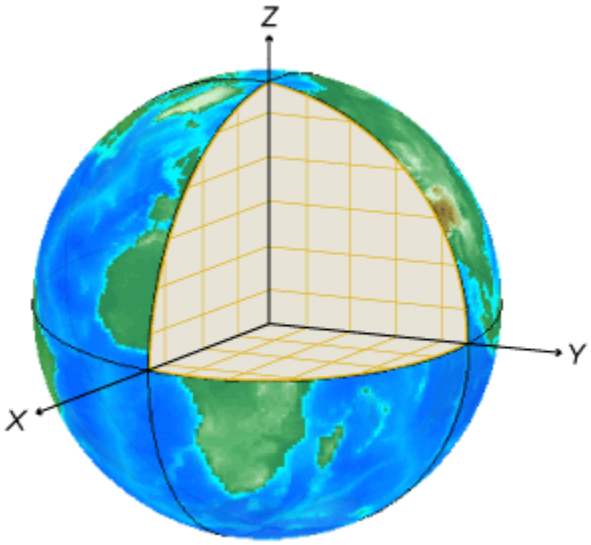
Global systems such as ECEF and geodetic systems describe the position of an object using a triplet of coordinates. Local systems such as ENU, NED, and AER systems require two triplets of coordinates: one triplet describes the location of the origin, and the other triplet describes the location of the object with respect to the origin.

When you work with 3-D coordinate systems, you must specify an ellipsoid model that approximates the shape of the Earth. For more information about ellipsoid models, see “Reference Spheroids” on page 3-4. All of the sample coordinates on this page use the World Geodetic System of 1984 (WGS84).

Earth-Centered Earth-Fixed Coordinates

An Earth-centered Earth-fixed (ECEF) system uses the Cartesian coordinates (*X*,*Y*,*Z*) to represent position relative to the center of the reference ellipsoid. The distance between the center of the ellipsoid and the center of the Earth depends on the reference ellipsoid.

- The positive *X*-axis intersects the surface of the ellipsoid at 0° latitude and 0° longitude, where the equator meets the prime meridian.
- The positive *Y*-axis intersects the surface of the ellipsoid at 0° latitude and 90° longitude.
- The positive *Z*-axis intersects the surface of the ellipsoid at 90° latitude and 0° longitude, the North Pole.

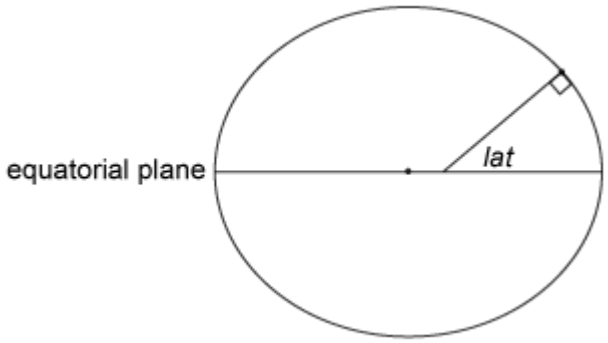


For example, the ECEF coordinates of Parc des Buttes-Chaumont are (4198945 m, 174747 m, 4781887 m).

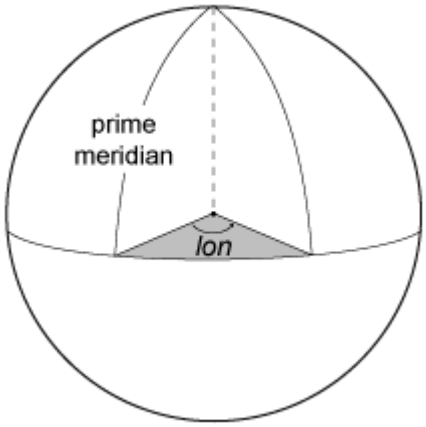
Geodetic Coordinates

A geodetic system uses the coordinates (*lat*,*lon*,*h*) to represent position relative to a reference ellipsoid.

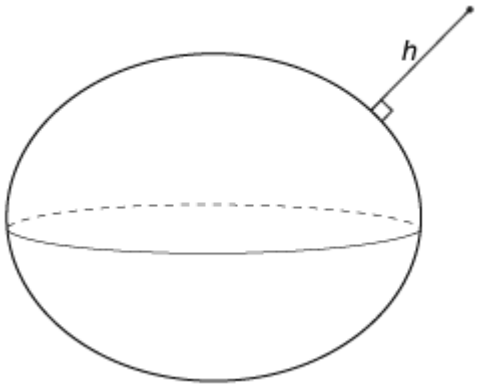
- *lat*, the latitude, originates at the equator. More specifically, the latitude of a point is the angle a normal to the ellipsoid at that point makes with the equatorial plane, which contains the center and equator of the ellipsoid. An angle of latitude is within the range [–90°, 90°]. Positive latitudes correspond to north and negative latitudes correspond to south.



- *lon*, the longitude, originates at the prime meridian. More specifically, the longitude of a point is the angle that a plane containing the ellipsoid center and the meridian containing that point makes with the plane containing the ellipsoid center and prime meridian. Positive longitudes are measured in a counterclockwise direction from a vantage point above the North Pole. Typically, longitude is within the range [–180°, 180°] or [0°, 360°].



- *h*, the ellipsoidal height, is measured along a normal of the reference spheroid. Coordinate transformation functions such as `geodetic2ecef` require you to specify *h* in the same units as the reference ellipsoid. You can change the units of the reference ellipsoid using the `LengthUnit` property. Terrain models typically supply data using orthometric height rather than ellipsoidal height. For information about calculating ellipsoidal height from orthometric height, see “Find Ellipsoidal Height from Orthometric and Geoid Height” on page 3-57.

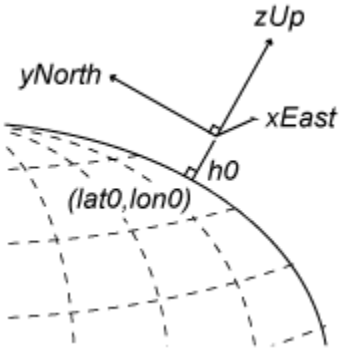


For example, the geodetic coordinates of Parc des Buttes-Chaumont are (48.8800°, 2.3831°, 124.5089 m).

East-North-Up Coordinates

An east-north-up (ENU) system uses the Cartesian coordinates (*xEast*,*yNorth*,*zUp*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Note that the origin does not necessarily lie on the surface of the ellipsoid.

- The positive *xEast*-axis points east along the parallel of latitude containing *lat0*.
- The positive *yNorth*-axis points north along the meridian of longitude containing *lon0*.
- The positive *zUp*-axis points upward along the ellipsoid normal.

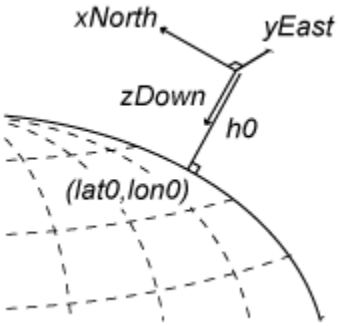


For example, Montmartre has geodetic coordinates (48.8862°, 2.3343°, 174.5217 m). The ENU coordinates of Parc des Buttes-Chaumont with respect to Montmartre are (3579.4232 m, –688.3514 m, –51.0524 m).

North-East-Down Coordinates

A north-east-down (NED) system uses the Cartesian coordinates (*xNorth*,*yEast*,*zDown*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Typically, the local origin of an NED system is above the surface of the Earth.

- The positive *xNorth*-axis points north along the meridian of longitude containing *lon0*.
- The positive *yEast*-axis points east along the parallel of latitude containing *lat0*.
- The positive *zDown*-axis points downward along the ellipsoid normal.



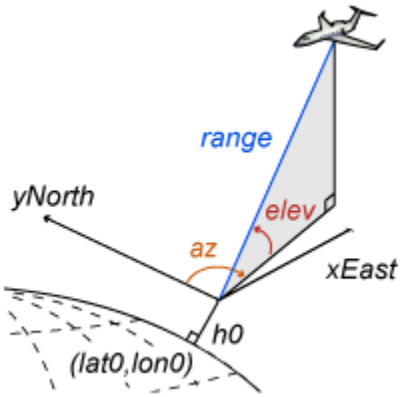
An NED coordinate system is commonly used to specify location relative to a moving aircraft. In this application, the origin and axes of an NED system change continuously. Note that the coordinates are not fixed to the frame of the aircraft.

For example, an aircraft flying into Charles de Gaulle airport has geodetic coordinates (48.9978°, 2.7594°, 699.8683 m). The NED coordinates of the airport with respect to the plane are (1645.8313 m, –15677.1868 m, 555.8221 m).

Azimuth-Elevation-Range Coordinates

An azimuth-elevation-range (AER) system uses the spherical coordinates (*az*,*elev*,*range*) to represent position relative to a local origin. The local origin is described by the geodetic coordinates (*lat0*,*lon0*,*h0*). Azimuth, elevation, and slant range are dependent on a local Cartesian system, for example, an ENU system.

- *az*, the azimuth, is the clockwise angle in the *xEast*-*yNorth* plane from the positive *yNorth*-axis to the projection of the object into the plane.
- *elev*, the elevation, is the angle from the *xEast*-*yNorth* plane to the object.
- *range*, the slant range, is the Euclidean distance between the object and the local origin.



For example, a lidar sensor at the Charles de Gaulle airport has geodetic coordinates (48.0124°, 2.5451°, 163.4885 m). The AER coordinates of an airplane with respect to the sensor are (95.8314°, 1.8781°, 15773.1381 m).

Tips

If you are transforming coordinates between ENU, NED, and AER systems with the same origin, then you do not need to specify a reference ellipsoid or the coordinates of the origin.

See Also

`aer2ned` | `ecef2enu` | `enu2aer` | `geodetic2aer` | `geodetic2ecef` | `ned2geodetic`

More About

• “Reference Spheroids” on page 3-4

References

[1] Guowei, C., B.M. Cheh, and T. H. Lee. *Unmanned Rotorcraft Systems*. London: Springer-Verlag London Limited: 2011.

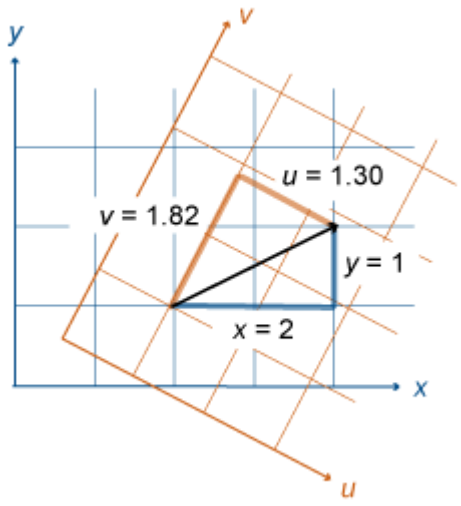
[2] Van Sickle, J. *Basic GIS Coordinates*. Boca Raton, FL: CRC Press LLC, 2004.

Vectors in 3-D Coordinate Systems

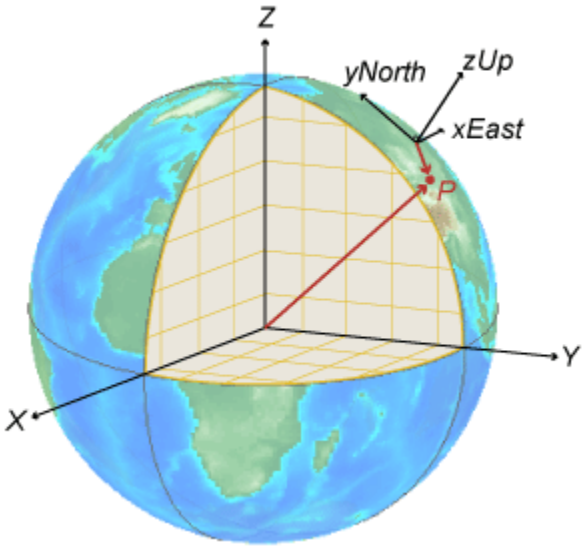
Vectors represent quantities such as velocity and acceleration. Mapping Toolbox functions transform vector components between Earth-centered Earth-fixed (ECEF) and east-north-up (ENU) or north-east-down (NED) systems. For more information about ECEF, ENU, and NED coordinate systems, see “Choose a 3-D Coordinate System” on page 3-47.

Unlike coordinates that measure position, vector components in a Cartesian system do not depend on a position in space. Therefore, when you transform a vector from one system to another, only the components of the vector change. The magnitude of the vector remains the same.

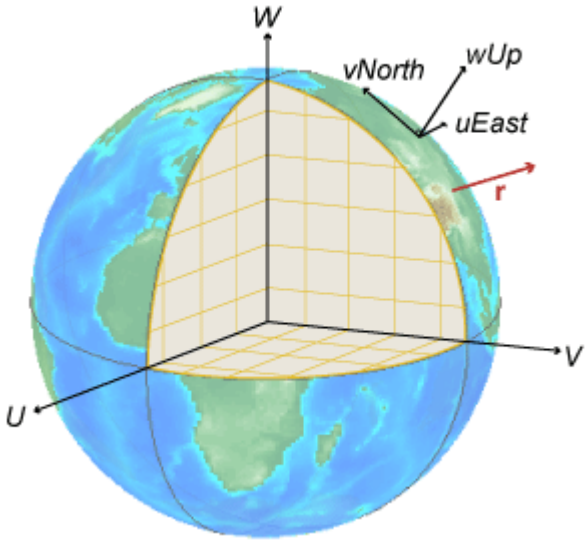
For example, this image shows a 2-D vector transformation from an *x*-*y* system to a *u*-*v* system. The vector has components $x = 2$ and $y = 1$ in the *x*-*y* system, and components $u = 1.30$ and $v = 1.82$ in the *u*-*v* system. The components of the vector are different, but in each system the magnitude of the vector is 2.24 units.



This image shows a coordinate transformation from a global ECEF system to a local ENU system using `ecef2enu`. The position vectors start at the origin of each system and end at point *P*. Therefore, the transformation changes the magnitude of the position vector.



This image shows a vector transformation from a global ECEF system to a local ENU system using `ecef2enuv`. The vector **r** does not depend on a position. Therefore, the transformation changes the components of the vector, but the magnitude of the vector is the same.



Tips

Unlike coordinate transformation functions such as `ecef2enu`, vector transformation functions such as `ecef2enuv` do not require you to specify a reference spheroid or the ellipsoidal height of the local origin. The geodetic latitude and longitude of the local origin is sufficient to define the orientation of the *uEast*, *vNorth*, and *wUp* axes.

See Also

`ecef2enuv` | `ecef2nedv` | `enu2ecefv` | `ned2ecefv`

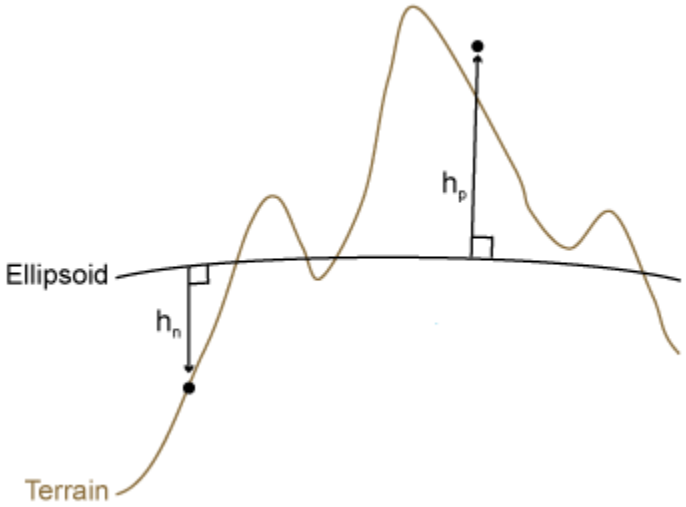
More About

- “Choose a 3-D Coordinate System” on page 3-47

Find Ellipsoidal Height from Orthometric Height

The *height* of an object may refer to its ellipsoidal height or its orthometric height. Mapping Toolbox functions such as `geodetic2enu` require the input argument *ellipsoidal height*, but data often quantifies *orthometric height* instead. You can convert orthometric height to ellipsoidal height by using a geoid model.

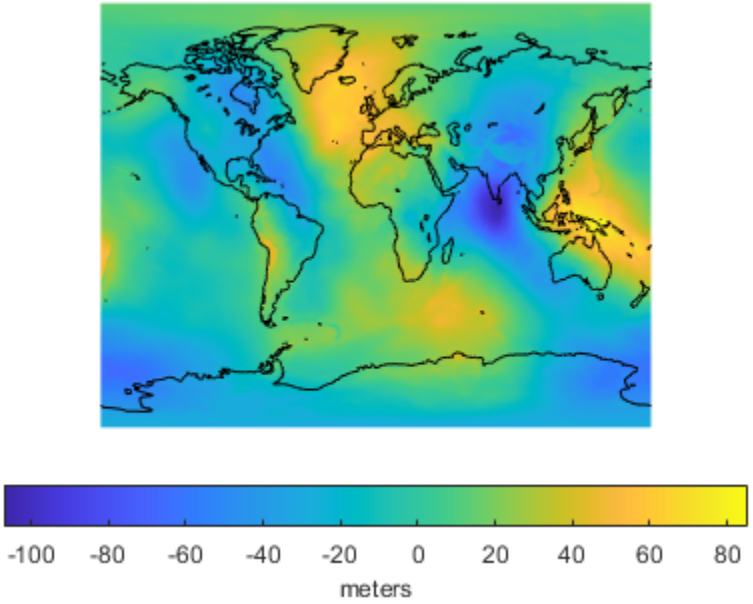
Ellipsoidal height, called h, is height measured along a normal of a reference ellipsoid. For more information about reference ellipsoids, see “Reference Spheroids” on page 3-4. This image shows a positive ellipsoidal height, h_p, and a negative ellipsoidal height, h_n.



Most terrain models provide data using orthometric height instead of ellipsoidal height. Orthometric height, called H, is height above the geoid.

The *geoid* models the average sea level of the Earth without effects such as weather, tides, and land. A geoid model is created by measuring variations in the Earth's gravitational field, so it has a smoothly undulating shape. Orthometric height is measured relative to the geoid.

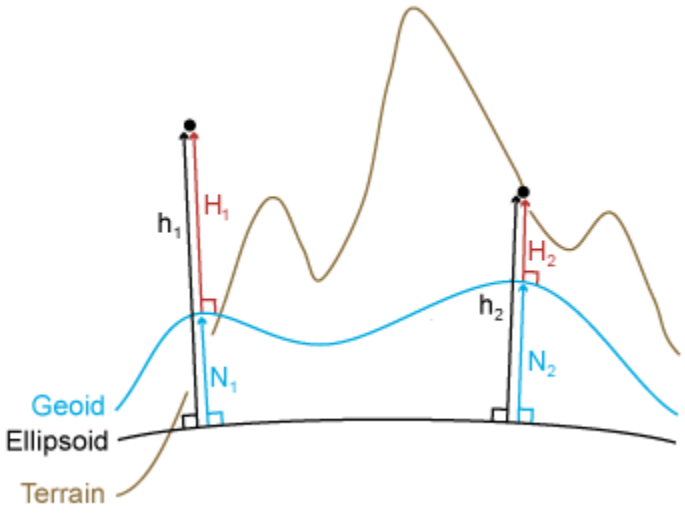
Geoid height, called N, is the height of the geoid measured along a normal of a reference ellipsoid. For example, geoid height values from the Earth Gravitational Model of 1996 (EGM96) are referenced to the ellipsoid defined by the World Geodetic System of 1984 (WGS84). Below is an illustration of the geoid from EGM96, with geoid heights in meters.



To find ellipsoidal height at a specified latitude and longitude, add the orthometric height and geoid height: $h = H + N$. You can find the height of the geoid from EGM96 at specified latitudes and longitudes using the `egm96geoid` function.

The equation $h = H + N$ is an approximation because the direction along which a geoid height is measured is not necessarily the same as the direction along which an orthometric height is measured. However, the approximation is suitable for most practical purposes.

The following image illustrates the relationship between ellipsoidal height, orthometric height, and the geoid. The values h_1, H_1, and N_1 demonstrate the relationship for an airborne object, while h_2, H_2, and N_2 demonstrate the relationship for an object on land.



Find Ellipsoidal Height from Orthometric and Geoid Height

Find the ellipsoidal height of the summit of Mount Everest, using its orthometric height and a geoid model.

Specify the latitude and longitude of the summit in degrees. Specify the orthometric height in meters.

```
lat = 27.988056;
lon = 86.925278;
H = 8848;
```

Find the height of the geoid at the location specified by `lat` and `lon` using `egm96geoid`.

```
N = egm96geoid(lat,lon);
```

Calculate the ellipsoidal height of the summit.

```
h = H + N
```

```
h = 8.8193e+03
```

References

[1] NOAA. "What is the geoid?" *National Ocean Service website*. https://oceanservice.noaa.gov/facts/geoid.html, 06/25/18.

See Also

`egm96geoid` | `geodetic2enu`

More About

• “Reference Spheroids” on page 3-4

Creating and Viewing Maps

- “Introduction to Mapping Graphics” on page 4-2
- “Continent, Country, Region, and State Maps Made Easy” on page 4-3
- “Set Background Colors for Map Displays” on page 4-4
- “Create Simple Maps Using worldmap” on page 4-5
- “Create Simple Maps Using usamap” on page 4-7
- “The Map Axes” on page 4-11
- “Access and Change Map Axes Properties” on page 4-13
- “Map Limit Properties” on page 4-19
- “Switch Between Projections” on page 4-34
- “Reprojection of Graphics Objects” on page 4-40
- “Create Maps Using geoshow” on page 4-43
- “Creating Maps Using MAPSHOW” on page 4-50
- “Change Map Projections Using geoshow” on page 4-68
- “Use Geographic and Nongeographic Objects in Map Axes” on page 4-72
- “The Map Frame” on page 4-75
- “Plot Regions of Robinson Frame and Grid Using Map Limits” on page 4-77
- “Map and Frame Limits” on page 4-82
- “The Map Grid” on page 4-83
- “Summary of Polygon Display Functions” on page 4-86
- “Display Vector Data as Points and Lines” on page 4-87
- “Display Vector Maps as Lines or Patches” on page 4-91
- “Types of Data Grids and Raster Display Functions” on page 4-98
- “Fit Gridded Data to the Graticule” on page 4-99
- “Create 3-D Displays with Raster Data” on page 4-103
- “Creating Map Displays with Latitude and Longitude Data” on page 4-106
- “Creating Map Displays with Data in Projected Coordinate Reference System” on page 4-116
- “Pick Locations Interactively” on page 4-125
- “Create an Interactive Map for Selecting Point Features” on page 4-127
- “Create Small Circle and Track Annotations on Maps Interactively” on page 4-133
- “Interactively Display Text Annotations on a Map” on page 4-135
- “Work with Objects by Name” on page 4-136

Introduction to Mapping Graphics

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

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

Mapping Toolbox software manages most of the details in displaying a map. It projects your data, cuts and trims it to specified limits, and displays the resulting map at various scales. With the toolbox you can also add customary cartographic elements, such as a frame, grid lines, coordinate labels, and text labels, to your displayed map. If you change your projection properties, or even the projection itself, some Mapping Toolbox map displays are automatically redrawn with the new settings, undoing any cuts or trims if necessary.

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 property editing tools you can invoke by clicking on the display.

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

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`.

Set Background Colors for Map Displays

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.

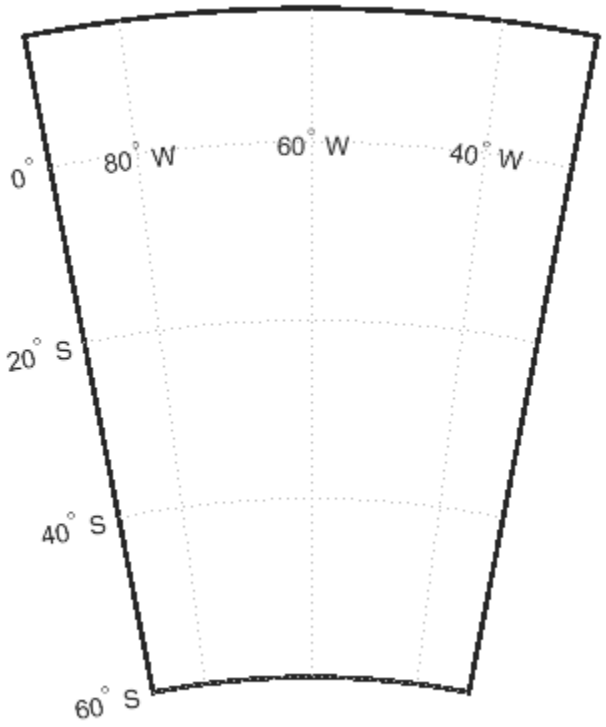
You can also use the Property Editor, part of the MATLAB plotting tools, to modify background colors for figures and axes.

Create Simple Maps Using worldmap

This example shows how to create simple maps using the `worldmap` function.

Set up the map frame, letting the `worldmap` function pick the projection. This example creates a map of South America.

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



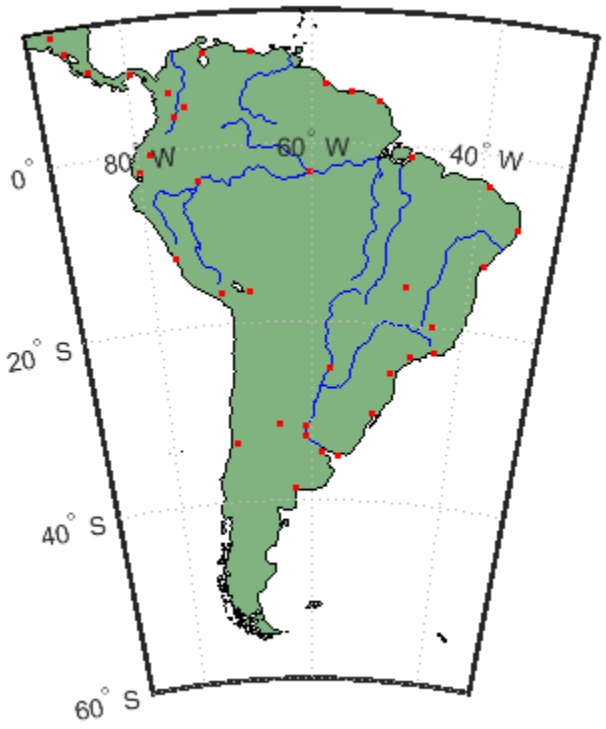
Determine which map projection the `worldmap` function used by looking at the value of the `MapProjection` property of the map axes. The value `eqdconic` stands for Equidistant Conic projection

```
getm(gca,'MapProjection')
```

```
ans =
'eqdconic'
```

Use the `geoshow` function to import sample data for land areas, major rivers, and major cities from shapefiles and display it using color you specify.

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


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

Create Simple Maps Using usamap

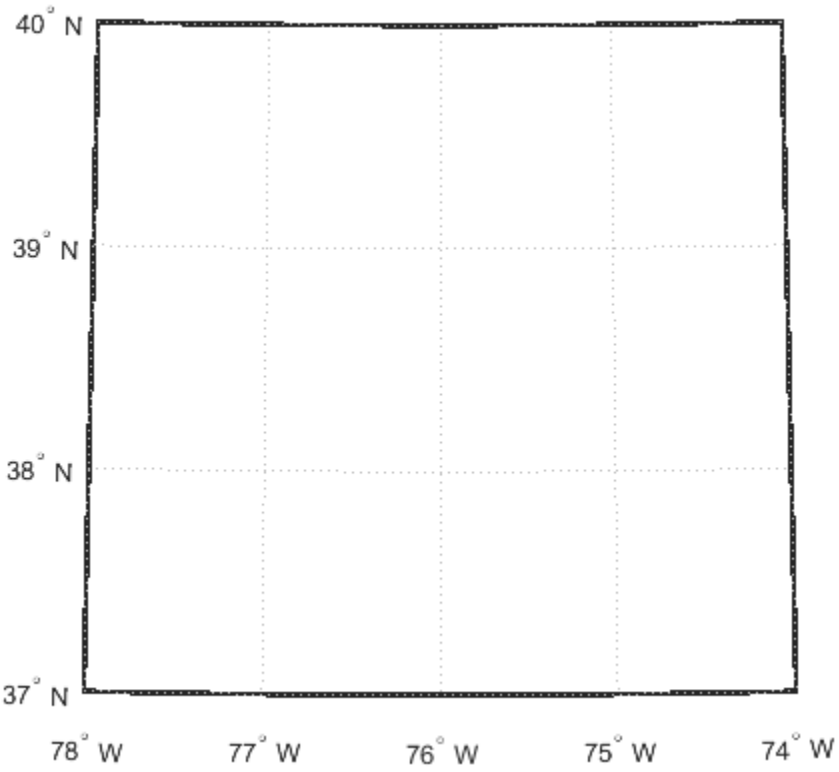
This example shows how to create maps of the United States using the `usamap` function. The `usamap` function lets you make maps of the United States as a whole, just the conterminous portion (the "lower 48" states), groups of states, or a single state. The map axes you create with the `usamap` function 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 the `geoshow` function.

Specify map limits and set up a map axes object. This example creates a map of the Chesapeake Bay region.

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

```
ax =
  Axes with properties:

            XLim: [-1.8118e+05 1.8118e+05]
            YLim: [4.4299e+06 4.7720e+06]
          XScale: 'linear'
          YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
axis off
```



Determine the map projection used by the `usamap` function. The Lambert Conformal Conic projection is often used for maps of the conterminous United States.

```
getm(gca,'MapProjection')
```

```
ans =
'lambert'
```

Use the `shaperead` function to read U.S. state polygon boundaries from the `usastatehi` shapefile. The function returns the data in a geostruct.

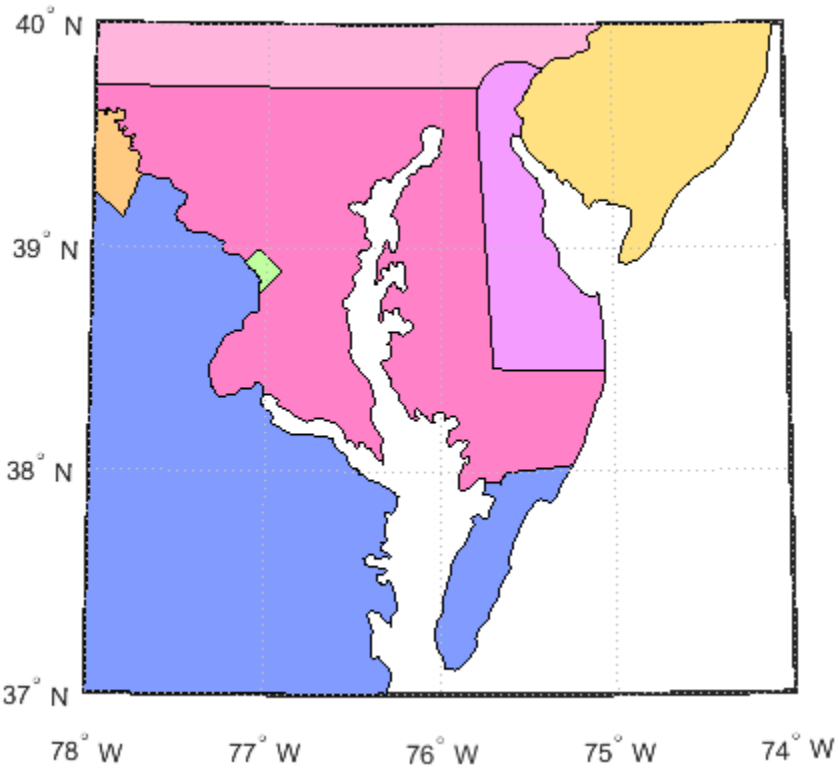
```
states = shaperead('usastatehi',...
    'UseGeoCoords',true,'BoundingBox',[lonlim',latlim']);
```

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

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

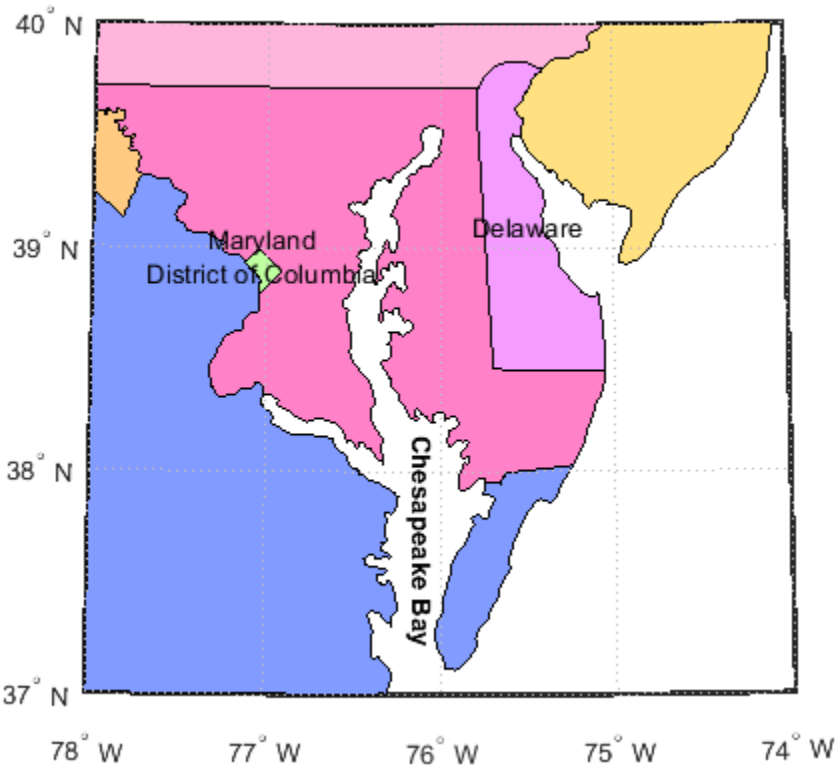
Display the filled polygons with the geoshow function.

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



Extract the names for states within the window from the geostruct and use the `textm` function to plot them at the label points provided by the geostruct. Because `polcmap` assigns random pastel colors to patches, your map might look different than this example.

```
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)
```


40° N
39° N
38° N
37° N
78° W
77° W
76° W
75° W
74° W
Maryland
District of Columbia
Delaware
Chesapeake Bay

The Map Axes

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

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

Mapping Toolbox functions include an extended version of `axes`, called `axesm`. Axes created with `axesm` share all properties associated with regular axes, and they 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, scale, and other properties. For complete descriptions of all map axes properties, see the `axesm` reference page.

The syntax of `axesm` is similar to that of `axes`:

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

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

The figure window created using `axesm` contains the same set of tools and menus as any MATLAB figure. By default, the figure window is 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.

Tips to Working with Map Axes

- You can list all the names, classes, and IDs of Mapping Toolbox map projections with the `maps` function.
- You can place many types of objects in a map axes, such as lines, patches, markers, scale rulers, north arrows, grids, and text. You can use the `handlem` function and its associated UI to list these objects. See the `handlem` reference page for a list of the objects that can occupy a map axes and how to query for them.
- You can define multiple independent figures containing map axes, but only one can be active at any one time. Use `axes(obj)` to activate an existing map axes object.
- 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)
  ```

See Also

`axesm` | `axesmui` | `handlem`

Access and Change Map Axes Properties

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

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

Create a map axes object containing no map data. Note that you specify `MapProjection` ID values in lowercase.

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

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`.

```
getm(gca,'FLineWidth')
```

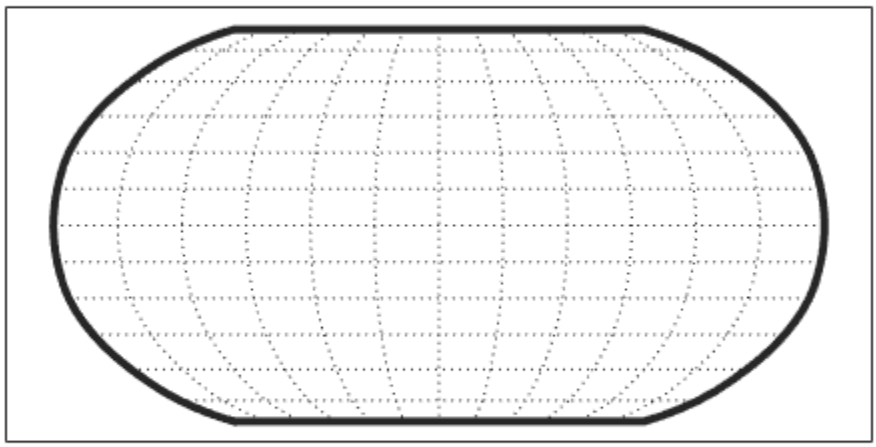
```
ans = 2
```

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

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

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,'Grid','on','MapProjection','robinson')
```



```
getm(gca,'FLineWidth')

ans = 3

getm(gca,'MapProjection')

ans =
'robinson'
```

Inspect the entire set of map axes properties at their current settings. Note that the list of properties includes both those particular to map axes and general ones that apply to all MATLAB® axes.

```
getm(gca)

ans = struct with fields:
     mapprojection: 'robinson'
              zone: []
        angleunits: 'degrees'
            aspect: 'normal'
     falsenorthing: 0
      falseeasting: 0
       fixedorient: []
             geoid: [1 0]
       maplatlimit: [-90 90]
       maplonlimit: [-180 180]
      mapparallels: 38
        nparallels: 0
            origin: [0 0 0]
```

```
      scalefactor: 1
          trimlat: [-90 90]
          trimlon: [-180 180]
            frame: 'on'
            ffill: 100
       fedgecolor: [0.1500 0.1500 0.1500]
       ffacecolor: 'none'
        flatlimit: [-90 90]
       flinewidth: 3
        flonlimit: [-180 180]
             grid: 'on'
        galtitude: Inf
           gcolor: [0.1500 0.1500 0.1500]
      glinestyle: ':'
      glinewidth: 0.5000
   mlineexception: []
       mlinefill: 100
      mlinelimit: []
    mlinelocation: 30
     mlinevisible: 'on'
   plineexception: []
       plinefill: 100
      plinelimit: []
    plinelocation: 15
     plinevisible: 'on'
        fontangle: 'normal'
        fontcolor: [0.1500 0.1500 0.1500]
         fontname: 'Helvetica'
         fontsize: 10
        fontunits: 'points'
       fontweight: 'normal'
     labelformat: 'compass'
    labelrotation: 'off'
       labelunits: 'degrees'
    meridianlabel: 'off'
   mlabellocation: 30
   mlabelparallel: 90
      mlabelround: 0
    parallellabel: 'off'
   plabellocation: 15
   plabelmeridian: -180
      plabelround: 0
```

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,'FontUnits')
```

```
FontUnits                   [ inches | centimeters | normalized | {points} | pixels ]
```

```
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
```

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

```
getm(gca,'FontUnits')

ans =
'points'

getm(gca,'MapProjection')

ans =
'robinson'

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 which properties apply to a given map projection, see the reference page for that projection.

See Also

`axesm` | `getm` | `setm`

Map Limit Properties

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

You typically use the `MapLatLimit` and `MapLonLimit` properties to set up a map axes with a non-oblique, non-azimuthal projection, with its origin on the Equator. (Most of the projections included in the Mapping Toolbox fall into this category; e.g., cylindrical, pseudo-cylindrical, conic, or modified azimuthal.) In addition, even with a non-zero origin latitude (origin off the Equator), you can use the `MapLatLimit` and `MapLonLimit` properties with projections that are implemented directly rather than via rotations of the sphere (e.g., `tranmerc`, `utm`, `lambertstd`, `cassinistd`, `eqaconicstd`, `eqdconicstd`, and `polyconicstd`). This list includes the projections used most frequently for large-scale maps, such as U.S. Geological Survey topographic quadrangle maps. Finally, when the origin is located at a pole or on the Equator, you can use the map limit properties with any azimuthal projection (e.g., `stereo`, `ortho`, `breusing`, `eqaazim`, `eqdazim`, `gnomonic`, or `vperspec`).

On the other hand, you should avoid the map limit properties, working instead with the `Origin`, `FLatLimit`, and `FLonLimit` properties, when:

- You want your map frame to be positioned asymmetrically with respect to the origin longitude.
- You want to use an oblique aspect (that is, assign a non-zero rotation angle to the third element of the "orientation vector" supplied as the `Origin` property value).
- You want to change your projection's default aspect (normal vs. transverse).
- You want to use a nonzero origin latitude, except in one of the special cases noted above.
- You are using one of the following projections:
  - `globe` — No need for map limits; always covers entire planet
  - `cassini` — Always in a transverse aspect
  - `wetch` — Always in a transverse aspect
  - `bries` — Always in an oblique aspect

There's no need to supply a value for the `MapLatLimit` property if you've already supplied one for the `Origin` and `FLatLimit` properties. In fact, if you supply all three when calling either `axesm` or `setm`, the `FLatLimit` value will be ignored. Likewise, if you supply values for `Origin`, `FLonLimit`, and `MapLonLimit`, the `FLonLimit` value will be ignored.

If you do supply a value for either `MapLatLimit` or `MapLonLimit` in one of the situations listed above, `axesm` or `setm` will ignore it and issue a warning. For example,

```
axesm('lambert','Origin',[40 0],'MapLatLimit',[20 70])
```

generates the warning message:

```
Ignoring value of MapLatLimit due to use of nonzero origin
 latitude with the lambert projection.
```

It's important to understand that `MapLatLimit` and `MapLonLimit` are extra, redundant properties that are coupled to the `Origin`, `FLatLimit`, and `FLonLimit` properties. On the other hand, it's not too difficult to know how to update your map axes if you keep in mind the following:

- The `Origin` property takes precedence. It is set (implicitly, if not explicitly) every time you call `axesm` and you cannot change it just by changing the map limits. (Note that when creating a new map axes from scratch, the map limits are used to help set the origin if it is not explicitly specified.)
- `MapLatLimit` takes precedence over `FLatLimit` if both are provided in the same call to `axesm` or `setm`, but changing either one alone affects the other.
- `MapLonLimit` and `FLonLimit` have a similar relationship.

The precedence of `Origin` means that if you want to reset your map limits with `setm` and have `setm` also determine a new origin, you must set `Origin` to [] in the same call. For example,

```
setm(gca,'Origin',[],'MapLatLimit',newMapLatlim,...
   'MapLonLimit',newMapLonlim)
```

On the other hand, a call like this will automatically update the values of `FLatLimit` and `FLonLimit`. Similarly, a call like:

```
setm(gca,'FLatLimit',newFrameLatlim,'FLonLimit',newFrameLonlim)
```

will update the values of `MapLatLimit` and `MapLonLimit`.

Finally, you probably don't want to try the following:

```
setm(gca,'Origin',[],'FLonLimit',newFrameLonlim)
```

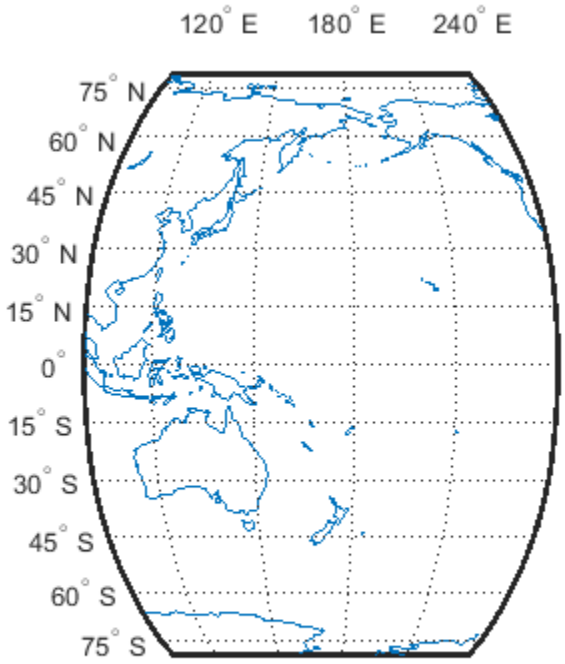
because the value of `FLonLimit` (unlike `MapLonLimit`) will not affect `Origin`, which will merely change to a projection-dependent default value (typically `[0 0 0]`).

Specify Map Projection Origin and Frame Limits Automatically

This example shows how to specify the map projection origin and frame limits using the two map limit properties: `MapLatLimit` and `MapLonLimit`. While the map axes supports properties to set these values directly, `Origin`, `FLatLimit`, and `FLonLimit`, it is easier and more intuitive to use the map limit properties, especially when creating a new map axes with `axesm`. This example highlights the interdependency of the map axes limits and the map limit properties.

Create a map using a cylindrical projection or pseudo-cylindrical projection showing all or most of the Earth, with the Equator running as a straight horizontal line across the center of the map. The map is bounded by a geographic quadrangle and the projection origin is located on the Equator, centered between the longitude limits you specify using the map projection limits.

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



Check that the `axesm` function set the origin and frame limits based on the values you specified using the `MapLatLim` and `MapLonLim` properties. The longitude of the origin should be located halfway between the longitude limits of 100 E and 120 W. Since the map spans 140 degrees, adding half of 140 to the western limit, the origin longitude should be 170 degrees. The frame is centered on this longitude with a half-width of 70 degrees and the origin latitude is on the Equator.

```
origin = getm(gca,'Origin')
```

```
origin = 1×3

     0   170     0
```

```
flatlim = getm(gca,'FLatLimit')
```

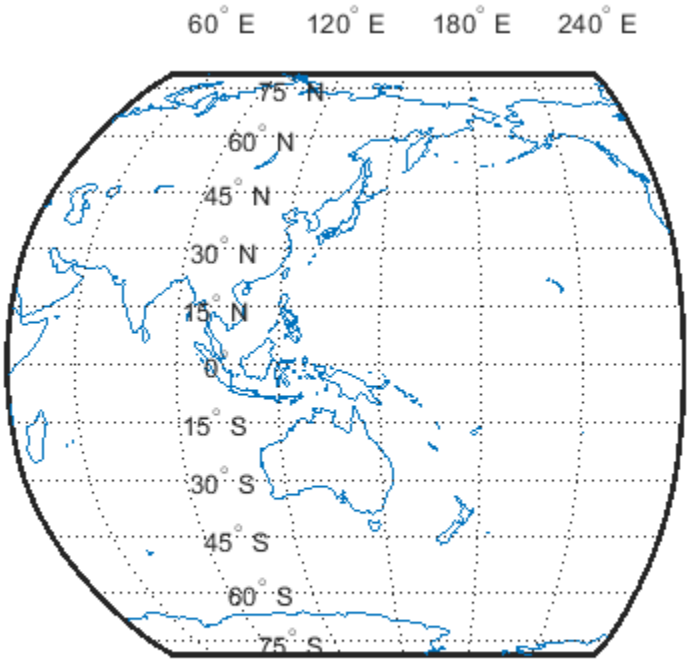
```
flatlim = 1×2

   -80    80
```

```
flonlim = getm(gca,'FLonLimit')
```

```
flonlim = 1×2

   -70    70
```

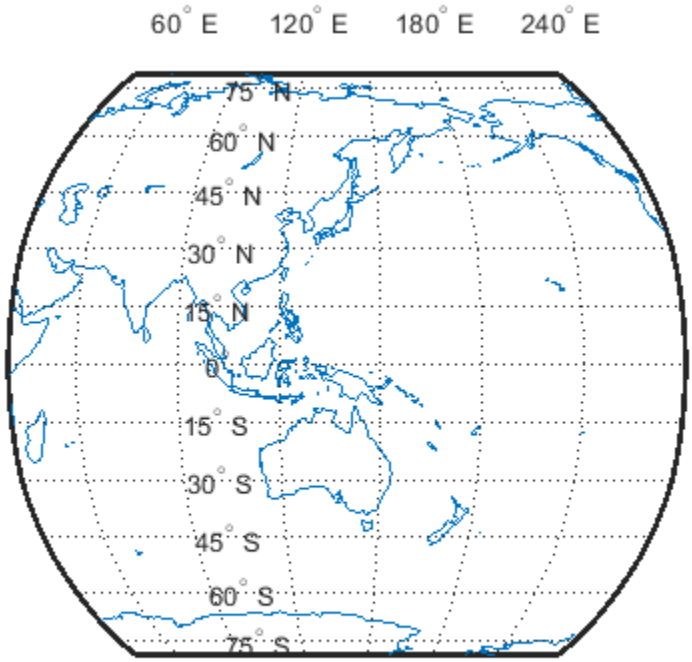
Shift the western longitude to 40 degrees E (rather than 100 degrees) to include a little more of Asia. Use the `setm` function to assign a new value to the `MapLonLimit` property. Note the asymmetric appearance of the map.

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



To correct the asymmetry, shift the western longitude again, this time specifying the origin. While the `MapLatLimit` and `MapLonLimit` properties are convenient, the values of the `Origin`, `FLatLimit`, and `FLonLimit` properties take precedence. You must specify the value of the origin to achieve the map you intended. The best way to do this is to specify an empty value for the `Origin` property and let the `setm` command calculate the value.

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



Create Cylindrical Projection Using Map Limit Properties

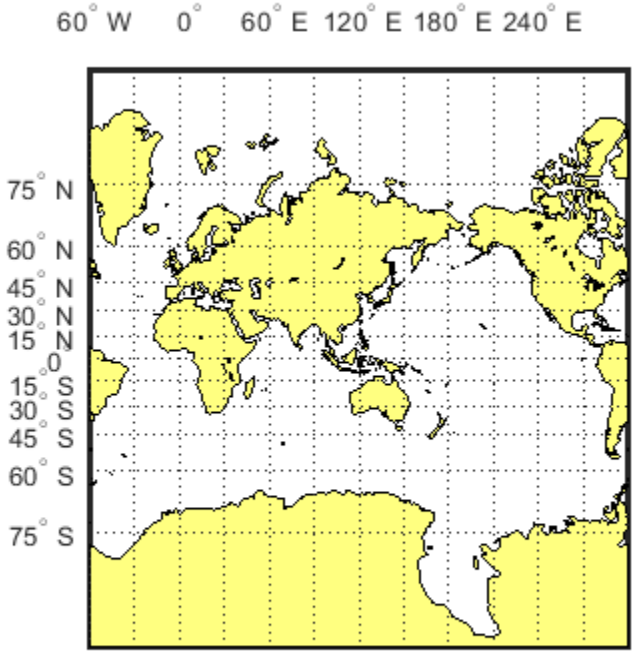
This example shows how to create cylindrical projection using map limit properties.

Load the coastline data.

```
load coastlines
```

Construct a Mercator projection covering the full range of permissible latitudes with longitudes covering a full 360 degrees starting at 60 West.

```
figure('Color','w')
axesm('mercator','MapLatLimit',[-90 90],'MapLonLimit',[-60 300])
axis off;
framem on;
gridm on;
mlabel on;
plabel on;
setm(gca,'MLabelLocation',60)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```



The previous call to `axesm` is equivalent to:

```
axesm('mercator','Origin',[0 120 0],'FlatLimit',[-90 90],'FLonLimit',[-180
180]);
```

You can verify this by checking the properties.

```
getm(gca,'Origin')
```

```
ans = 1×3

     0   120     0
```

```
getm(gca,'FLatLimit')
```

```
ans = 1×2

   -86    86
```

```
getm(gca,'FLonLimit')
```

```
ans = 1×2

  -180   180
```

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

```
getm(gca,'MapLatLimit')
```

```
ans = 1×2

   -86    86
```

```
getm(gca,'FLatLimit')
```

```
ans = 1×2

   -86    86
```

```
getm(gca,'TrimLat')
```

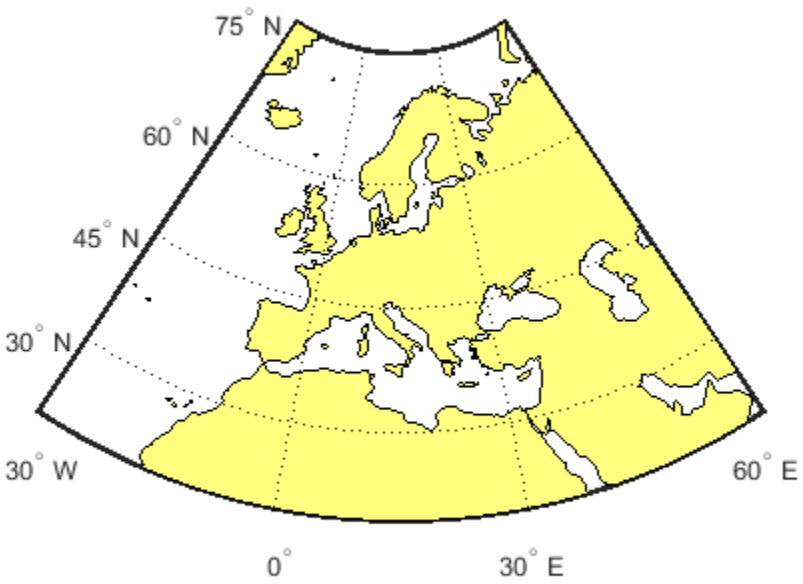
```
ans = 1×2

   -86    86
```

Create Conic Projection Using Map Limit Properties

This example shows how to create a map of the standard version of the Lambert Conformal Conic projection covering latitudes 20 North to 75 North and longitudes covering 90 degrees starting at 30 degrees West.

Load coastline data and display it. The call to `axesm` above is equivalent to: `axesm('lambertstd','Origin', [0 15 0], 'FLatLimit',[20 75],FLonLimit',[-45 45])`

```
load coastlines
figure('Color','w')
axesm('lambertstd','MapLatLimit',[20 75],'MapLonLimit',[-30 60])
axis off;
framem on;
gridm on;
mlabel on;
plabel on;
geoshow(coastlat, coastlon, 'DisplayType', 'polygon')
```



Create Southern Hemisphere Conic Projection

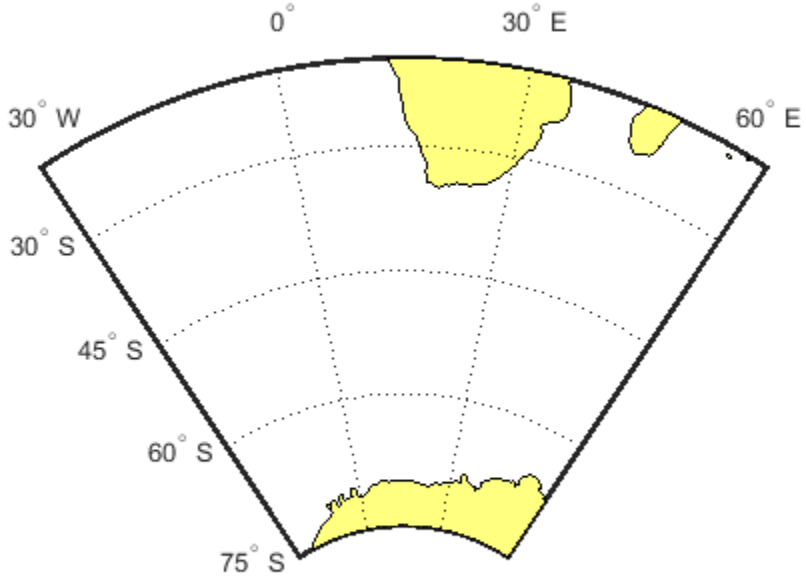
This example shows how to create a map of the standard version of the Lambert Conformal Conic projection into the Southern Hemisphere. The example overrides the default standard parallels and sets the `MapLatLimit` and `MapLonLimit` properties.

Load the coastline data MAT file, `coastlines.mat`.

```
load coastlines
```

Display the map, setting the `MapLatLimit` and `MapLonLimit` properties.

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



Create North-Polar Azimuthal Projection

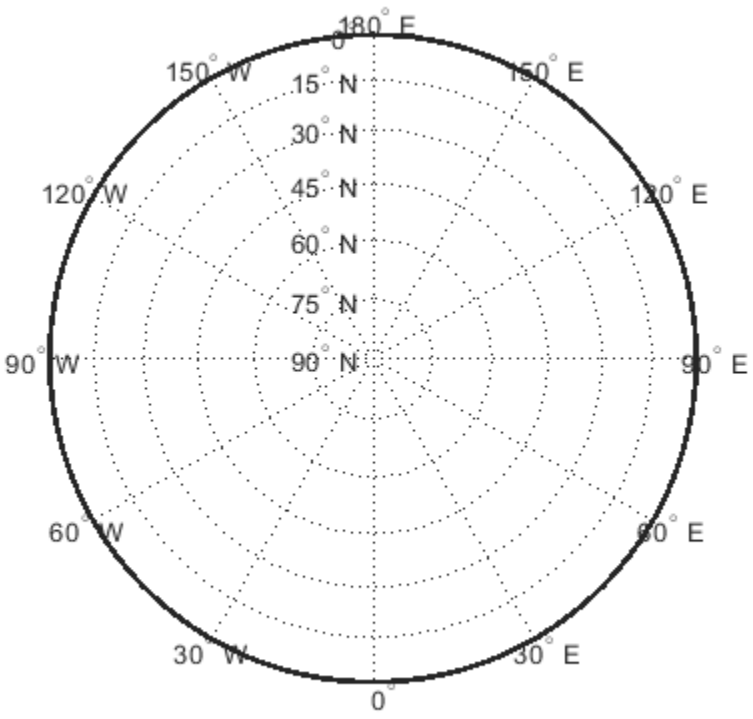
This example shows how to construct a North-polar Equal-Area Azimuthal projection map extending from the Equator to the pole and centered by default on longitude 0.

Load coastline data set MAT file, `coastlines.mat`.

```
load coastlines
```

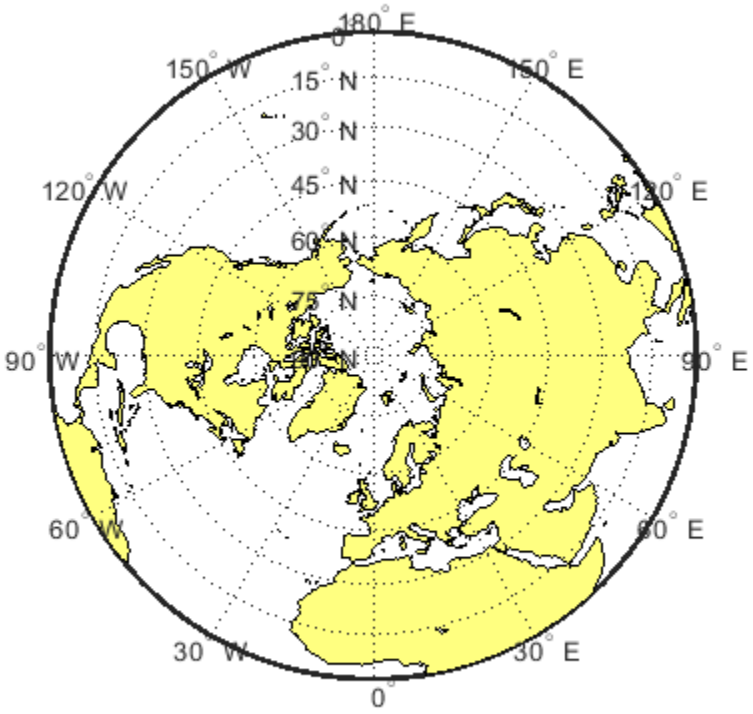
Create map. The call to `axesm` is equivalent to:
`axesm('eqaazim','MLabelParallel',0,'Origin',[90 0 0],'FLatLimit',[-Inf 90]);`

```
figure('Color','w')
axesm('eqaazim','MapLatLimit',[0 90])
axis off
framem on
gridm on
mlabel on
plabel on;
setm(gca,'MLabelParallel',0)
```



Plot the coast lines.

```
geoshow(coastlat,coastlon,'DisplayType','polygon')
```

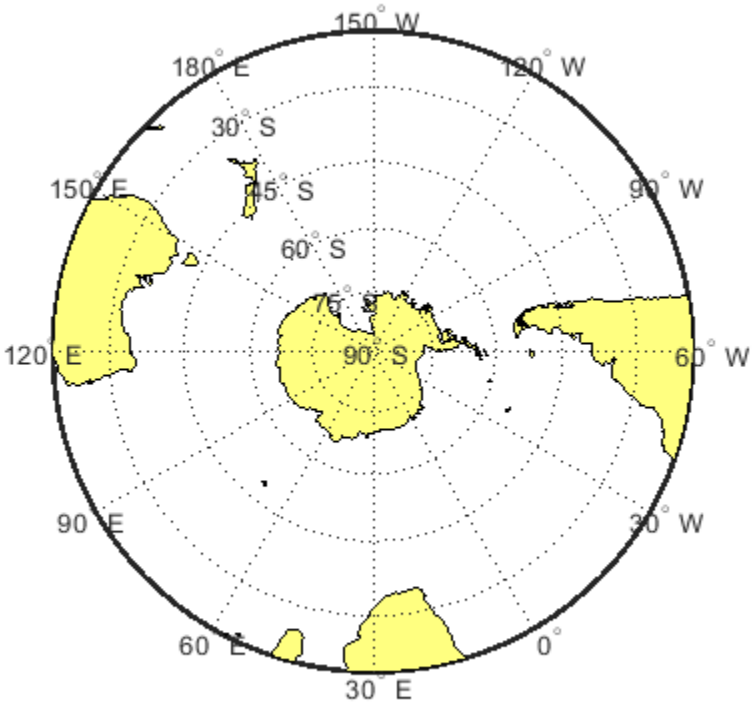


Create South-Polar Azimuthal Projection

This example shows how to create a South-polar Stereographic Azimuthal projection map extending from the South Pole to 20 degrees S, centered on longitude 150 degrees West. Include a value for the `Origin` property in order to control the central meridian.

Load coastline data and display map.

```
load coastlines
figure('Color','w')
axesm('stereo','Origin',[-90 -150],'MapLatLimit',[-90 -20])
axis off;
framem on;
gridm on;
mlabel on;
plabel on;
setm(gca,'MLabelParallel',-20)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```



The call to the `axesm` function above is equivalent to:

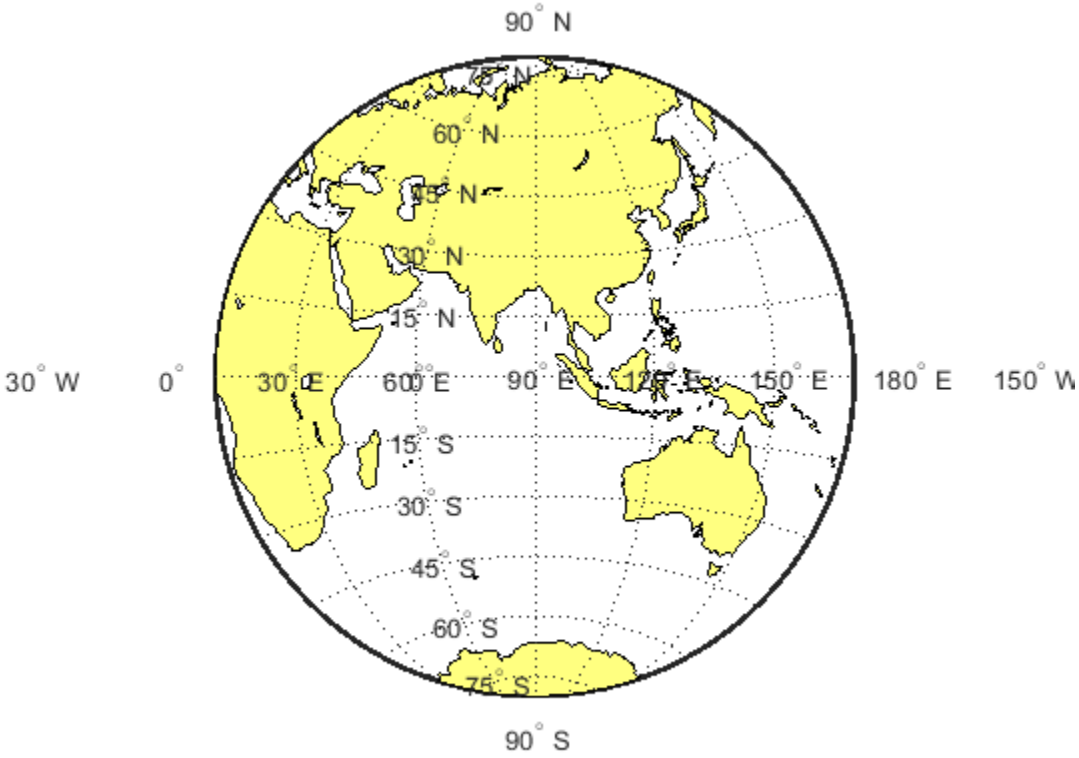
```
axesm('stereo','Origin',[-90 -150 0],'FLatLimit',[-Inf 70])
```

Create Equatorial Azimuthal Projection

This example shows how to create a map of an Equidistant Azimuthal projection with the origin on the Equator, covering from 10° E to 170° E. The origin longitude falls at the center of this range (90 E), and the map reaches north and south to within 10° of each pole.

Read coast data and display. The call to `axesm` is equivalent to `axesm('eqaazim','Origin',[0 90 0],'FLatLimit',[-Inf 80])`.

```
load coastlines
figure('Color','w')
axesm('eqdazim','FLatLimit',[],'MapLonLimit',[10 170])
axis off;
framem on;
gridm on;
mlabel on;
plabel on;
setm(gca,'MLabelParallel',0,'PLabelMeridian',60)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```

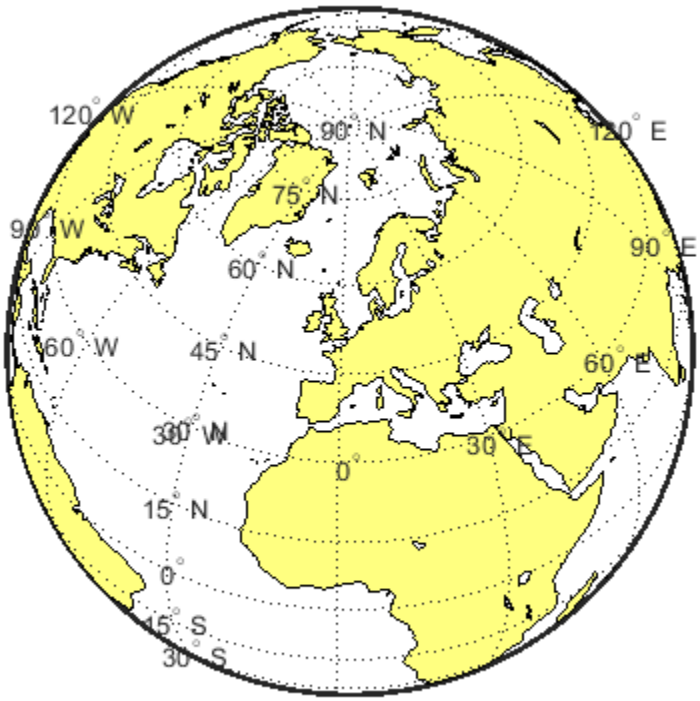


Create General Azimuthal Projection

This example shows how to construct an Orthographic projection map with the origin centered near Paris, France. You can't use `MapLatLimit` or `MapLonLimit` here.

Read in coast data and display.

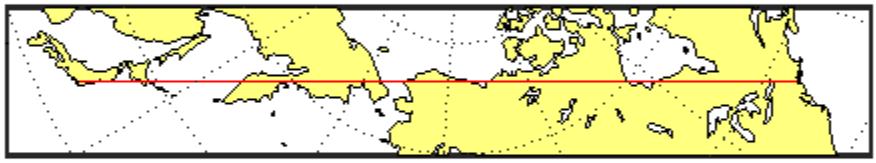
```
load coastlines
originLat = dm2degrees([48 48]);
originLon = dm2degrees([ 2 20]);

figure('Color','w')
axesm('ortho','Origin',[originLat originLon])
axis off; framem on; gridm on; mlabel on; plabel on;
setm(gca,'MLabelParallel',30,'PLabelMeridian',-30)
geoshow(coastlat,coastlon,'DisplayType','polygon')
```



Create Long Narrow Oblique Mercator Projection

This example shows how to create a map with a long, narrow, oblique Mercator projection. The example shows the area 10 degrees to either side of the great-circle flight path from Tokyo to New York. You can't use `MapLatLimit` or `MapLonLimit` .

```
load coastlines
latTokyo = dm2degrees([ 35 40]);
lonTokyo = dm2degrees([139 45]);

latNewYork = dm2degrees([ 40 47]);
lonNewYork = dm2degrees([-73 58]);

[dist,az] = distance(latTokyo,lonTokyo,latNewYork,lonNewYork);
[midLat,midLon] = reckon(latTokyo,lonTokyo,dist/2,az);
midAz = azimuth(midLat,midLon,latNewYork,lonNewYork);

buf = [-10 10];

figure('Color','w')
axesm('mercator','Origin',[midLat midLon 90-midAz], ...
    'FLatLimit',buf,'FLonLimit',[-dist/2 dist/2] + buf)
axis off; framem on; gridm on; tightmap
geoshow(coastlat,coastlon,'DisplayType','polygon')
plotm([latTokyo latNewYork],[lonTokyo lonNewYork],'r-')
```



See Also

More About

- “The Map Frame” on page 4-75
- “Map and Frame Limits” on page 4-82

Switch Between Projections

Once a map axes object has been created with `axesm`, whether map data is displayed or not, it is possible to change the current projection as well as many of its parameters. You can use `setm` or the `maptool` UI to reset the projection. The rest of this section describes the considerations and parameters involved in switching projections in a map axes. Additional details are given for doing this with the `geoshow` function in “Change Map Projections Using geoshow” on page 4-68.

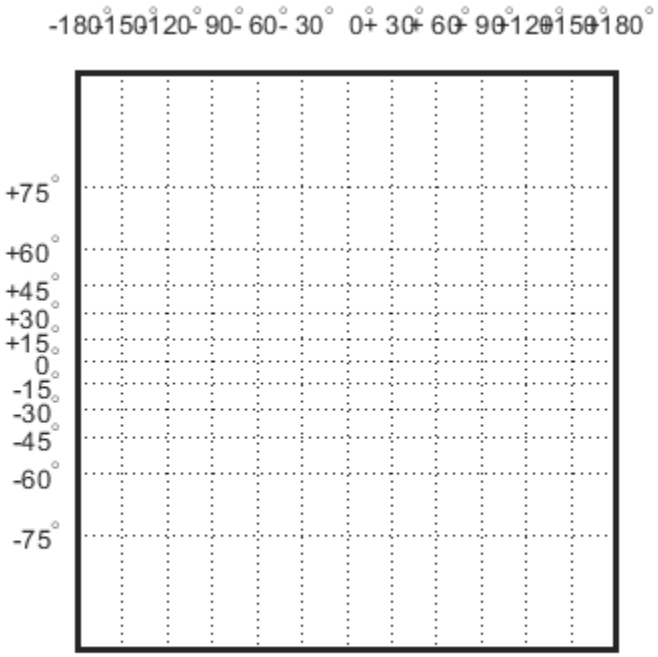
When you switch from one projection to another, `setm` clears out settings that were specific to the earlier projection, updates the map frame and graticule, and generally keeps the map covering the same part of the world—even when switching between azimuthal and non-azimuthal projections. But in some cases, you might need to further adjust the map axes properties to achieve proper appearance. Settings that are suitable for one projection might not be appropriate for another. Most often, you'll need to update the positioning of your meridian and parallel labels.

Change Projection Updating Meridian and Parallel Labels

This example shows how to change the projection of a map and update the meridian and parallel labels.

Create a Mercator projection with meridian and parallel labels.

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



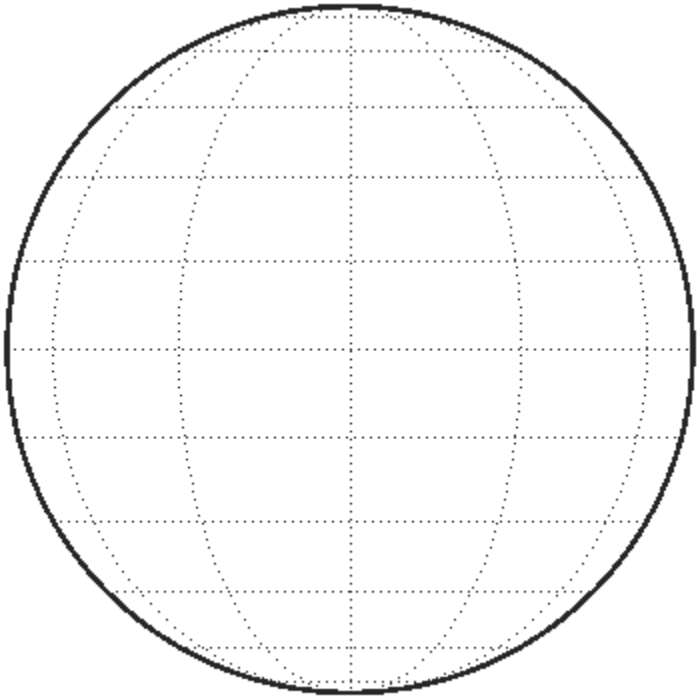
Get the default map and frame latitude limits for the Mercator projection. Note that both the frame and map latitude limits are set to 86 degrees north and south for the Mercator projection to maintain a safe distance from the singularity at the poles.

```
[getm(gca,'MapLatLimit'); getm(gca,'FLatLimit')]
```

```
ans = 2×2

   -86    86
   -86    86
```

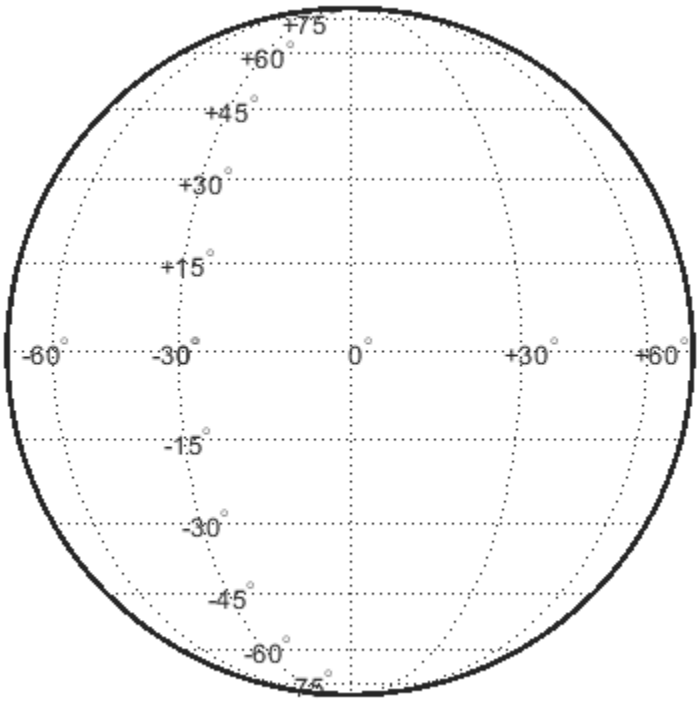
Switch the projection to an orthographic azimuthal.

```
setm(gca,'MapProjection','ortho')
```



Specify new locations for the meridian and parallel labels.

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

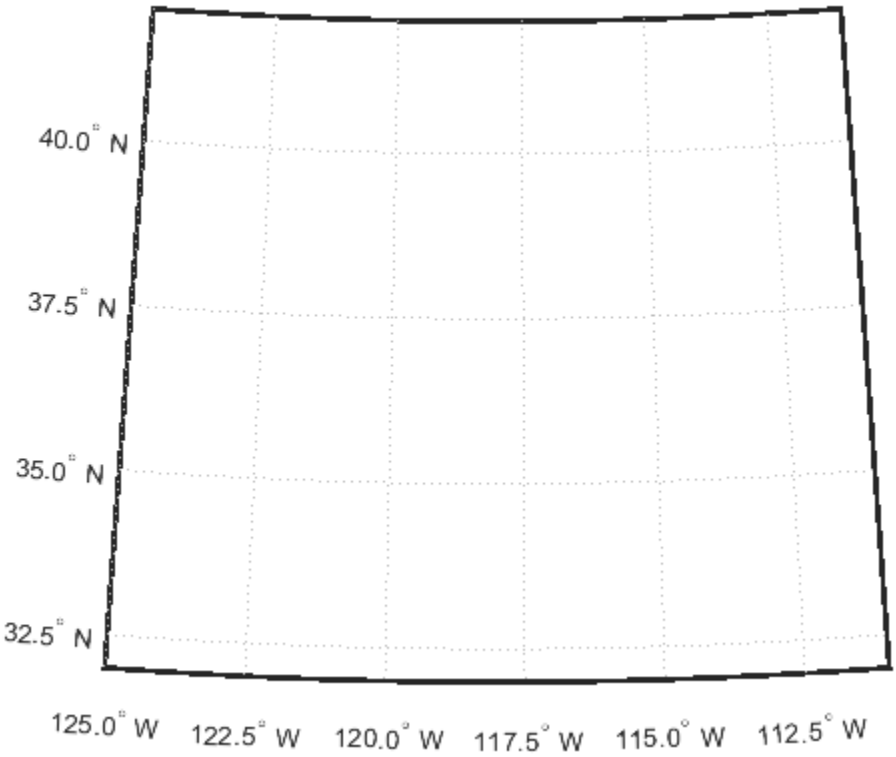


Change Projection Resetting Frame Limits

This example shows how to switch from one projection to another and reset the origin and frame limits, especially when mapping a small portion of the Earth.

Construct an empty map axes for a region of the United States in the Lambert Conformal Conic projection (the default projection for the `usamap` function).

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



Read the `usastatehi` shapefile and return a subset of the shapefile contents, as defined by the latitude and longitude limits. The `shaperead` function returns the data in a structure called `states` .

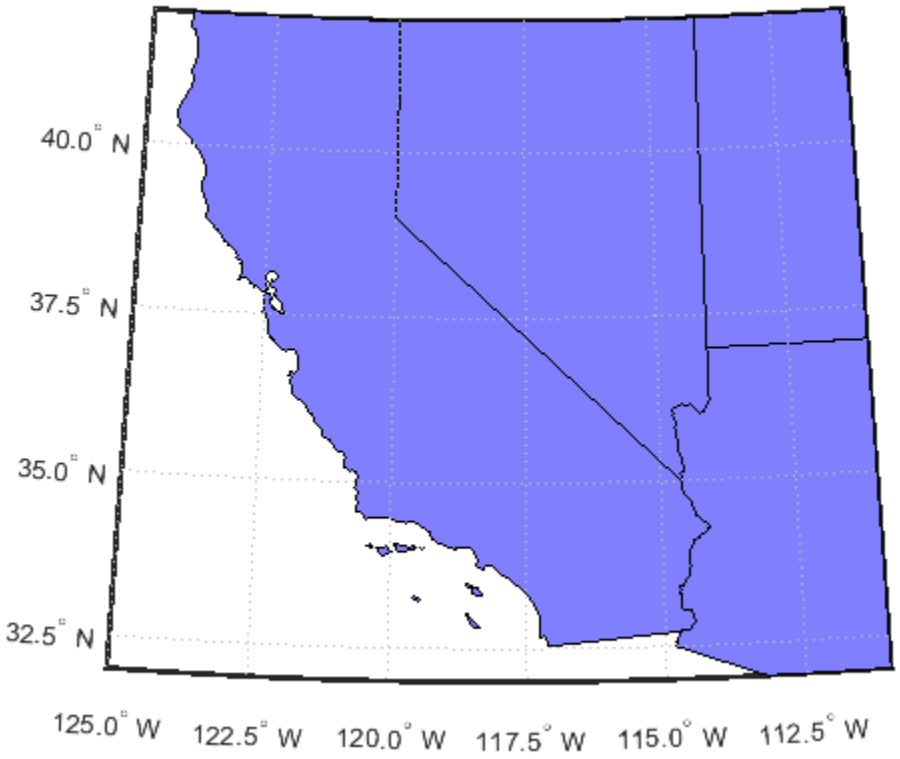
```
states = shaperead('usastatehi', 'UseGeoCoords', true, ...
   'BoundingBox', [lonlim', latlim']);
```

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

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

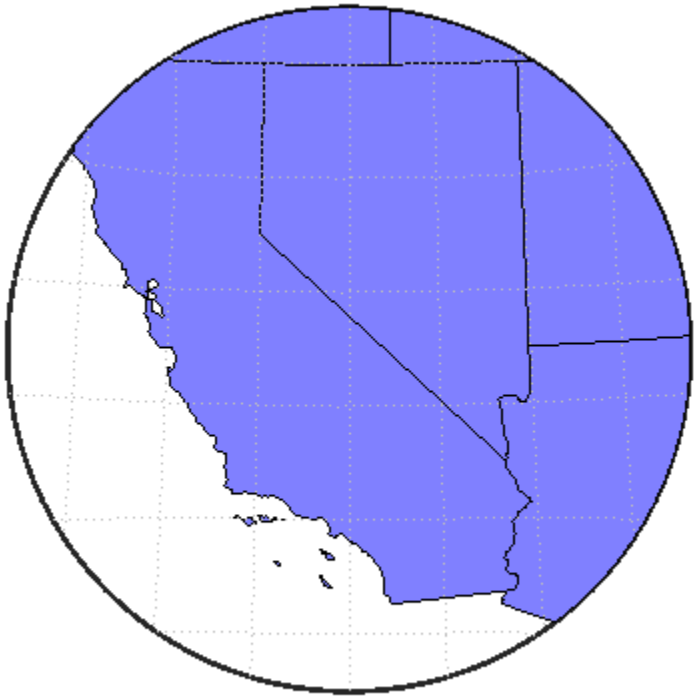
Project patch objects on the map axes.

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



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

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



Reprojection of Graphics Objects

Many Mapping Toolbox cartographic functions project features on a map axes based on their designated latitude-longitude positions. The latitudes and longitudes are mathematically transformed to *x* and *y* positions using the formulas for the current map projection. If the map projection or its parameters change, objects on a map axes can be automatically reprojected to update the map display accordingly.

The table summarizes the four common use cases for changing a map projection in a map axes with `setm` or for reprojecting map data plotted on a regular MATLAB axes.

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

You can use `handlem` to help identify which objects to delete when manual deletion is necessary. See “Work with Objects by Name” on page 4-136 for an example of its use.

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`

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.

In general, objects created with `geoshow` or with a combination of calls to `projfwd` 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.

If you have preprojected vector or raster map data or read such data from files, you can display it with `mapshow`, `mapview`, or standard MATLAB graphics functions, such as `plot` or `mesh`. If its projection is known and is included in the Mapping Toolbox projection libraries, you can use its parameters to project geodata in geographic coordinates to display it in the same axes.

Reprojectability of Maps Generated Using geoshow

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.

For examples of reprojection behavior with vector data and raster data, see “Change Map Projections Using geoshow” on page 4-68.

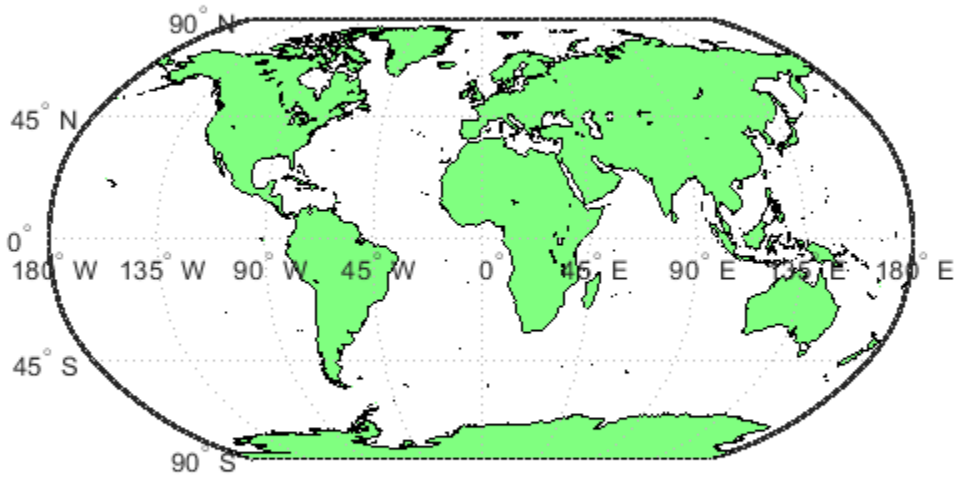
Create Maps Using geoshow

Create a range of different maps using `geoshow`.

Geographic map 1: World Land Area

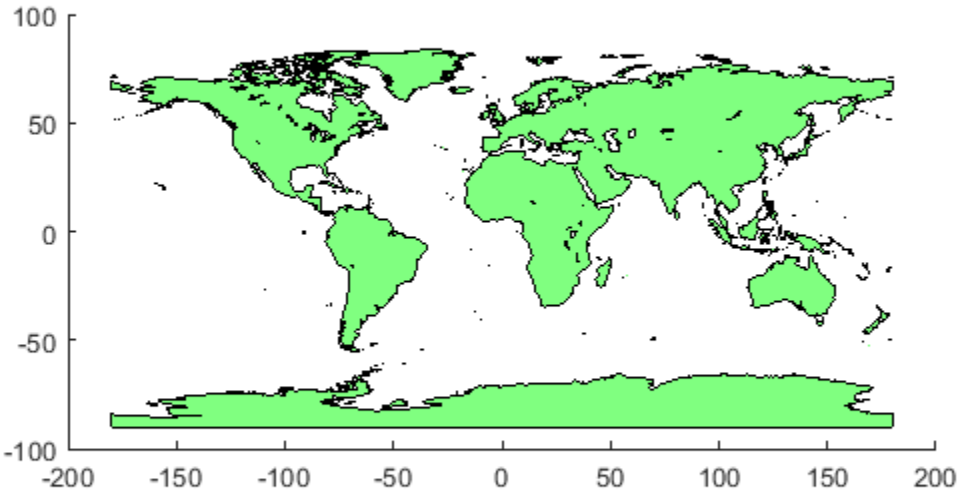
Create a worldmap. Then project and display world land areas.

```
worldmap world
geoshow('landareas.shp','FaceColor',[0.5 1.0 0.5])
```



You can also project and display world land areas using a default Plate Carree projection.

```
figure
geoshow('landareas.shp','FaceColor',[0.5 1.0 0.5])
```



The axes show position in latitude and longitude, but are displayed on a set of ordinary axes. To display geographic data on a set of map axes instead, use `axesm`, `usamap`, or `worldmap` before calling `geoshow`.

```
ismap
```

```
ans = 0
```

Geographic map 2: North America with Custom Colored States in the U.S.

Read the USA high resolution data.

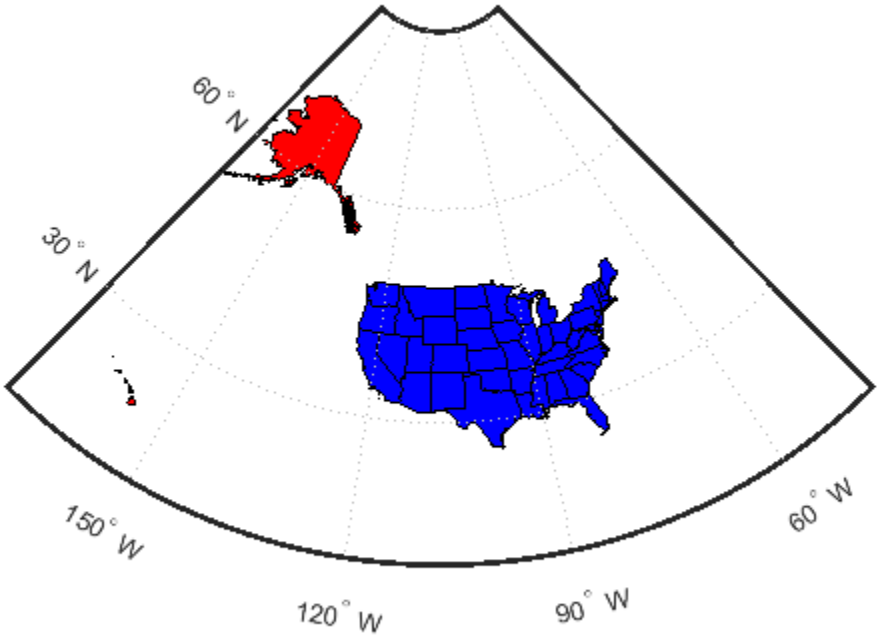
```
states = shaperead('usastatehi','UseGeoCoords',true);
```

Create a SymbolSpec to display Alaska and Hawaii as red polygons.

```
symbols = makesymbolspec('Polygon', ...
    {'Name','Alaska','FaceColor','red'}, ...
    {'Name','Hawaii','FaceColor','red'});
```

Create a world map of North America with Alaska and Hawaii in red, and all other states in blue.

```
figure
worldmap('north america')
geoshow(states,'SymbolSpec',symbols, ...
    'DefaultFaceColor','blue','DefaultEdgeColor','black')
axis off
```



Geographic map 3: Korea Elevation Grid

Load elevation data and a geographic cells reference object for the Korean peninsula. Import a land area boundary using `shaperead`.

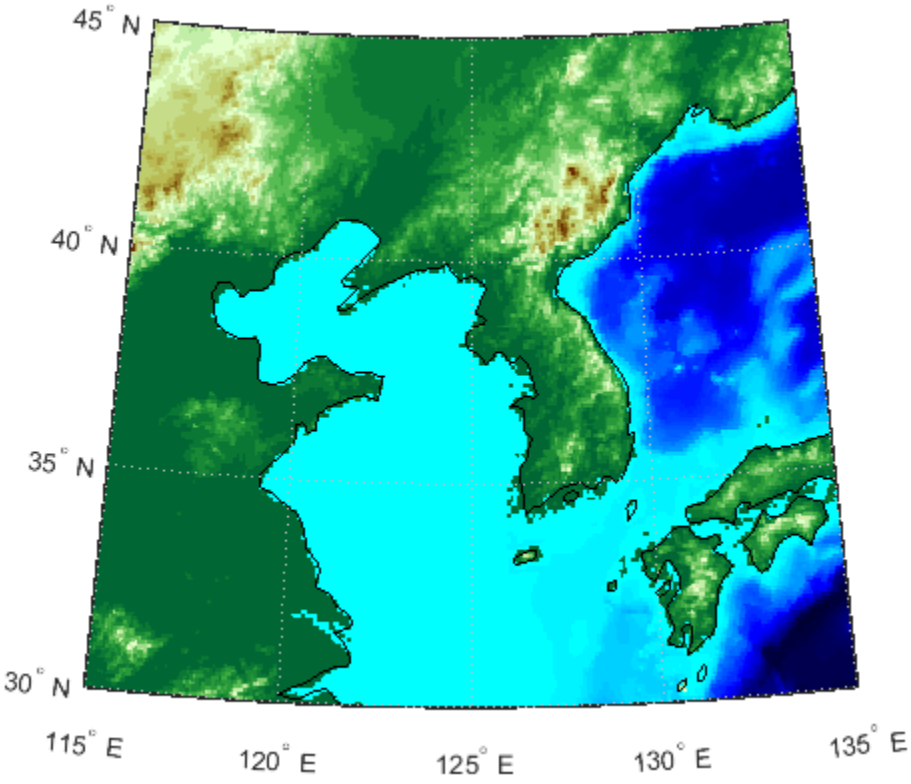
```
load korea5c
S = shaperead('landareas','UseGeoCoords',true);
```

Create a world map. Then project and display the elevation data as a texture map.

```
figure
worldmap(korea5c,korea5cR)
geoshow(korea5c,korea5cR,'DisplayType','texturemap')
demcmap(korea5c)
```

Overlay the land area boundary as a line.

```
geoshow([S.Lat],[S.Lon],'Color','k')
```



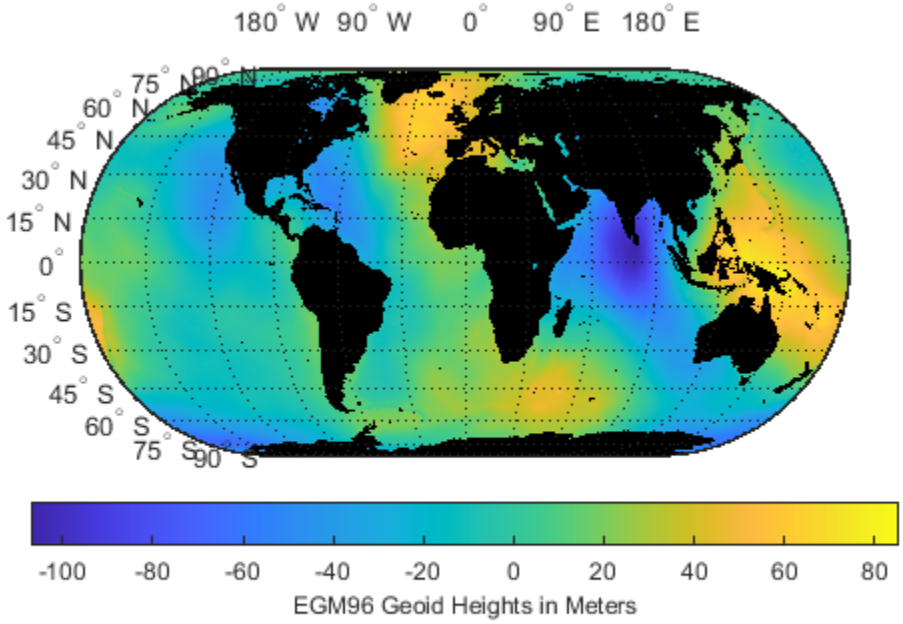
Geographic map 4: EGM96 Geoid Heights

Get geoid heights and a geographic postings reference object from the EGM96 geoid model. Then, display the heights as a surface using an Eckert projection. Ensure the surface appears below the grid lines by setting the `'CData'` name-value pair to the geoid height data and the `'ZData'` name-value pair to a matrix of zeros. Display the frame and grid of the map using `framem` and `gridm`. Display the parallel and meridian labels using `plabel` and `mlabel`.

```
[N,R] = egm96geoid;
figure
axesm eckert4
Z = zeros(R.RasterSize);
geoshow(N,R,'DisplayType','surface','CData',N,'ZData',Z)
framem
gridm
plabel
mlabel('MLabelLocation',90)
axis off
```

Create a colorbar and add a text description. Then, mask out all the land.

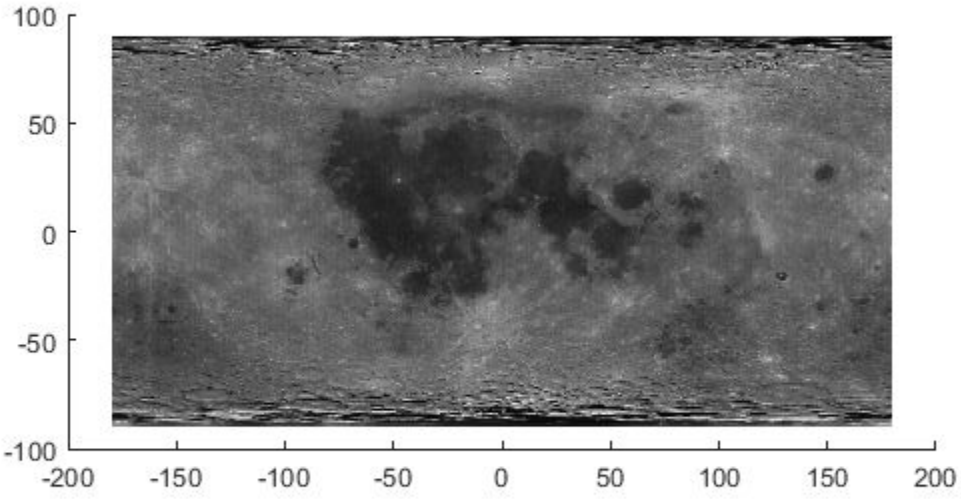
```
cb = colorbar('southoutside');
cb.Label.String = 'EGM96 Geoid Heights in Meters';
geoshow('landareas.shp','FaceColor','k')
```



Geographic map 5: Moon Albedo Image

Load moon albedo data and a geographic cells reference object. Project and display the data using the default Plate Carree projection.

```
load moonalb20c
figure
geoshow(moonalb20c,moonalb20cR)
```



Project and display the moon albedo data using an orthographic projection. To do this, create a map axes object and specify the projection as orthographic. Display the data in the map axes as a texture map using `geoshow`. Then, change the colormap to grayscale and remove the axis lines.

```
figure
axesm ortho
geoshow(moonalb20c,moonalb20cR,'DisplayType','texturemap')
colormap gray
axis off
```

See Also

`axesm` | `framem` | `geoshow` | `makesymbolspec` | `mapshow` | `shaperead` | `worldmap`

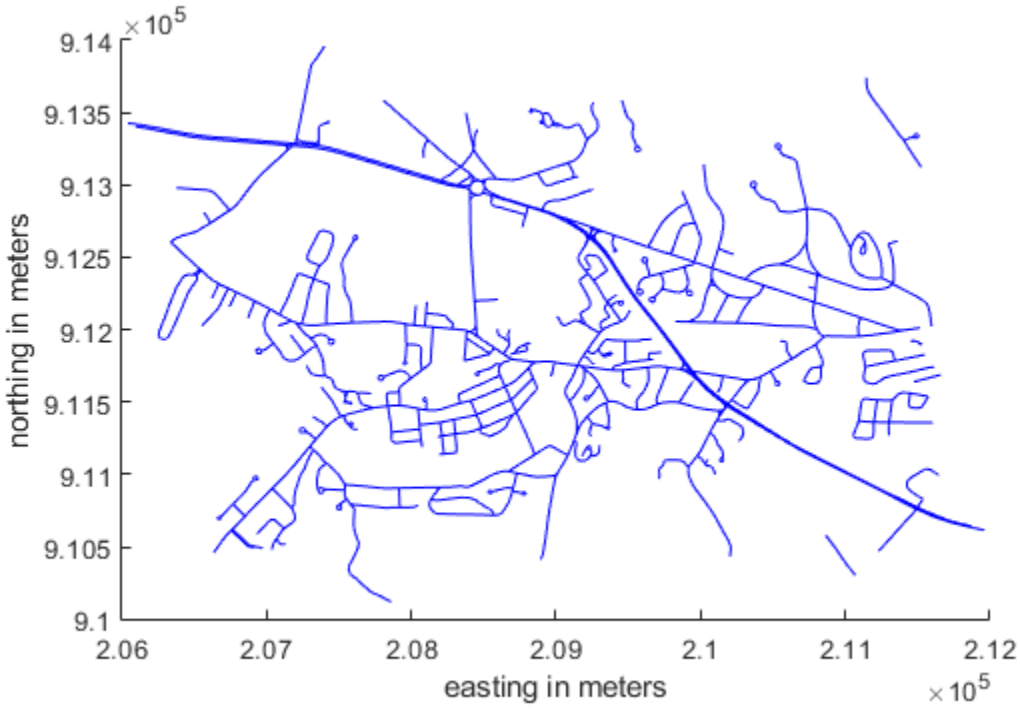
Creating Maps Using MAPSHOW

This example shows how to create a range of different maps using `mapshow`.

Map 1: Concord Roads - A Geographic Data Structure

Display a geographic data structure array with lines representing roads. In the shapefile 'concord_roads.shp', the road coordinates have been pre-projected to the Massachusetts Mainland State Plane system (in meters), so the shapefile is imported into a mapstruct (the variable 'roads').

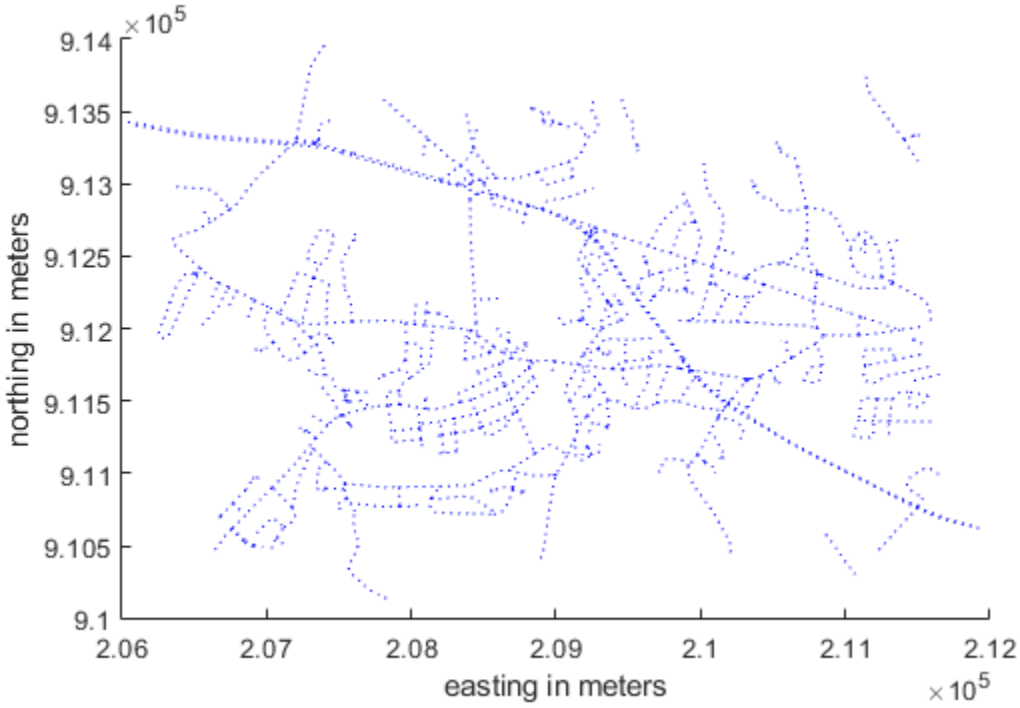
```
roads = shaperead('concord_roads.shp');
figure
mapshow(roads);
xlabel('easting in meters')
ylabel('northing in meters')
```



Map 2: Concord Roads with Custom Line Style

Display the roads shape and change the line style.

```
figure
mapshow('concord_roads.shp','LineStyle',':');
xlabel('easting in meters')
ylabel('northing in meters')
```



Map 3: Concord Roads with SymbolSpec

Display the roads shape, and render using a SymbolSpec.

To learn about the `concord_roads.shp` dataset, read its associated `concord_roads.txt` metadata file which describes the attributes.

```
type concord_roads.txt
```

```
A shapefile data set for roads in part of Concord, Massachusetts,
USA comprising the following files:

   concord_roads.dbf
   concord_roads.shp
   concord_roads.shx

Source
------
Office of Geographic and Environmental Information (MassGIS),
Commonwealth of Massachusetts  Executive Office of Environmental Affairs
(http://www.state.ma.us/mgis/)

Coordinate system/projection
----------------------------
All data distributed by MassGIS are registered to the NAD83 datum,
Massachusetts State Plane Mainland Zone coordinate system. Units are in
meters.
```

```
  Data set construction
  ---------------------
  This data set was constructed by concatenating Massachusetts Highway
  Department road shapefiles for the Maynard and Concord USGS Quadrangles,
  from compressed files mrd97.exe and mrd104.exe.

  Features were selected with bounding boxes intersecting the following
  box:
            [206500 (min easting)  910500 (min northing)
             211500 (max easting)  913500 (max northing)]

  The following attributes were retained:

      'STREETNAME', 'RT_NUMBER', 'CLASS', 'ADMIN_TYPE', 'LENGTH'

  Attributes 'CLASS' and 'ADMIN_TYPE' contain numerical codes defined by
  MassGIS as follows:

  Road classes (from file mrdac.dbf)
  ----------------------------------
  CLASS 1 Limited access highway
  CLASS 2 Multi-lane highway, not limited access
  CLASS 3 Other numbered route
  CLASS 4 Major road - collector
  CLASS 5 Minor street or road
  CLASS 6 Minor street or road
  CLASS 7 Highway ramp

  Road admin types (from file mrdac.dbf)
  --------------------------------------
  ADMIN_TYPE 0 Local road
  ADMIN_TYPE 1 Interstate
  ADMIN_TYPE 2 U.S. Federal
  ADMIN_TYPE 3 State

  Construction date
  -----------------
  November 17, 2003.
```

Query the attributes in this roads file.

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

```
roads =

  609×1 struct array with fields:

    Geometry
    BoundingBox
    X
    Y
    STREETNAME
    RT_NUMBER
    CLASS
    ADMIN_TYPE
```

```
    LENGTH
```

Find out how many roads fall in each CLASS.

```
histcounts([roads.CLASS],'BinLimits',[1 7],'BinMethod','integer')
```

```
ans =

     0    14    93    26   395    81     0
```

Find out how many roads fall in each ADMIN_TYPE.

```
histcounts([roads.ADMIN_TYPE],'BinLimits',[0 3],'BinMethod','integer')
```

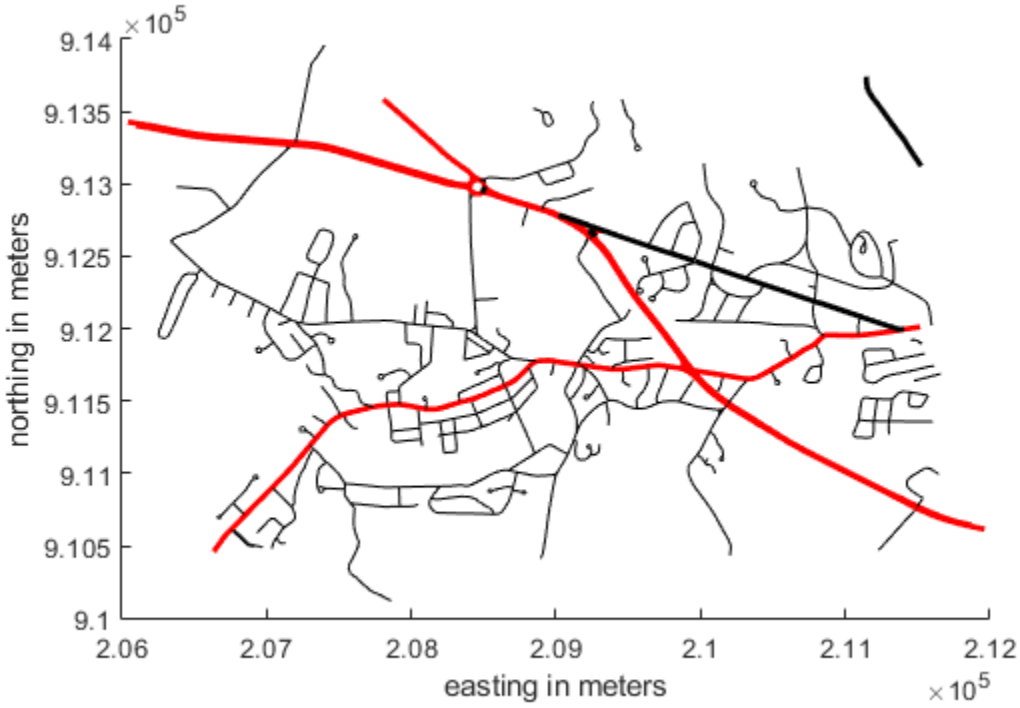
```
ans =

   502     0     0   107
```

Notice that there are no roads in this file that are CLASS 1 or 7, and the roads are either ADMIN_TYPE 0 or 3.

Create a SymbolSpec to:

- Color local roads (ADMIN_TYPE=0) black.
- Color state roads (ADMIN_TYPE=3) red.
- Hide very minor roads (CLASS=6).
- Set major or larger roads (CLASS=1-4) with a `LineWidth` value of 2.0.

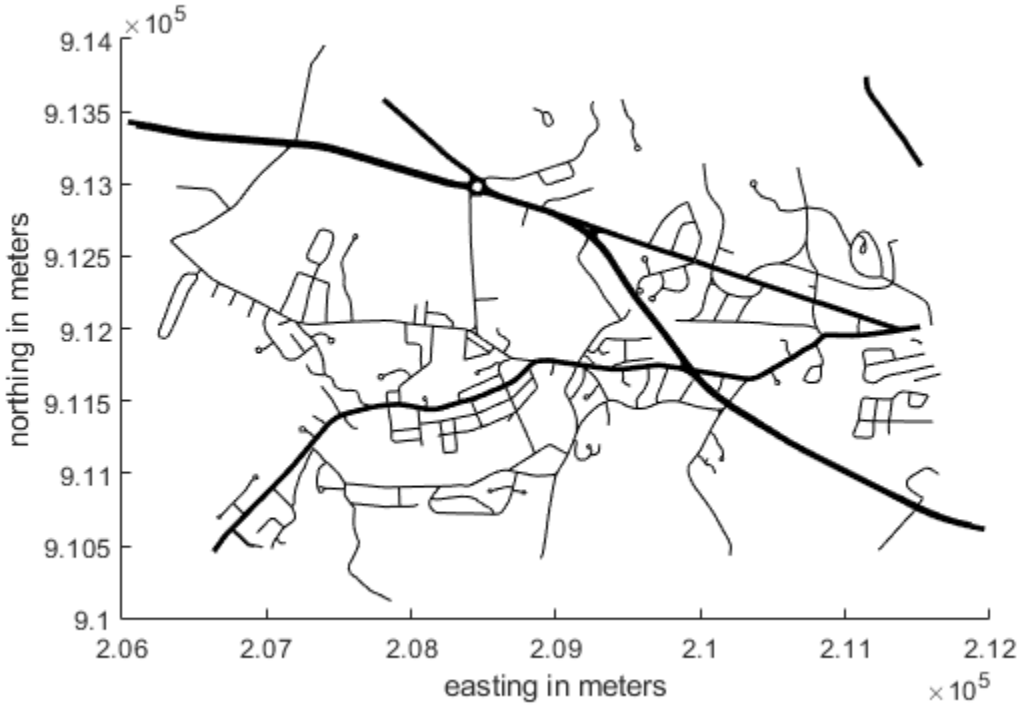
```
roadspec = makesymbolspec('Line',...
    {'ADMIN_TYPE',0, 'Color','black'}, ...
    {'ADMIN_TYPE',3, 'Color','red'},...
    {'CLASS',6, 'Visible','off'},...
    {'CLASS',[1 4], 'LineWidth',2});
figure
mapshow('concord_roads.shp','SymbolSpec',roadspec);
xlabel('easting in meters')
ylabel('northing in meters')
```



Map 4: Concord Roads, Override SymbolSpec

Override a graphics property of the SymbolSpec.

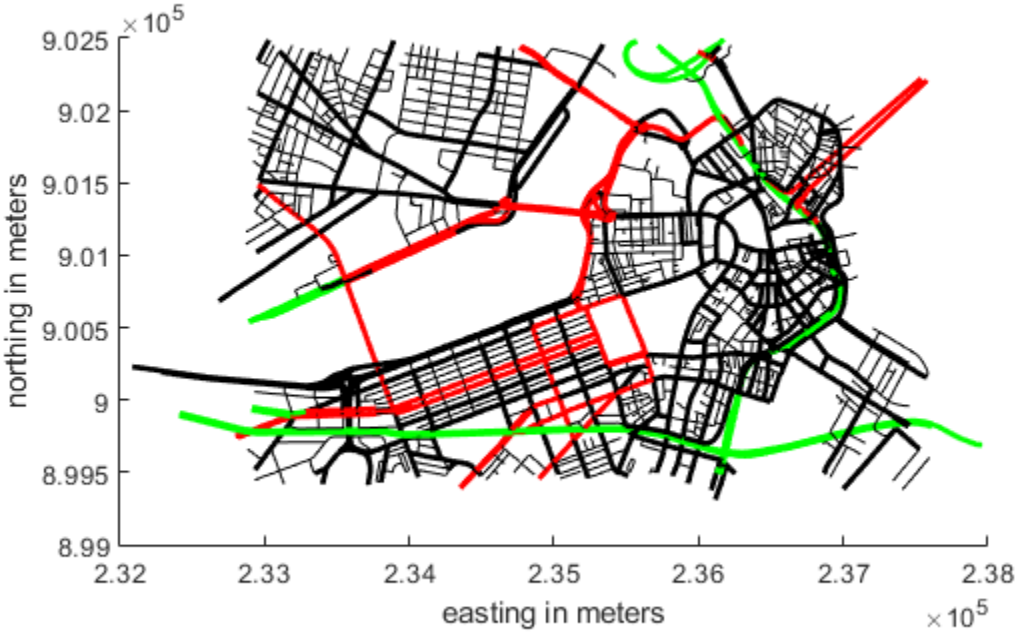
```
roadspec = makesymbolspec('Line',...
    {'ADMIN_TYPE',0, 'Color','black'}, ...
    {'ADMIN_TYPE',3, 'Color','red'},...
    {'CLASS',6, 'Visible','off'},...
    {'CLASS',[1 4], 'LineWidth',2});
figure
mapshow('concord_roads.shp','SymbolSpec',roadspec,'Color','black');
xlabel('easting in meters')
ylabel('northing in meters')
```



Map 5: Boston Roads with SymbolSpec, Override Defaults

Override default property of the SymbolSpec.

```
roadspec = makesymbolspec('Line',...
    {'Default', 'Color','green'}, ...
    {'ADMIN_TYPE',0, 'Color','black'}, ...
    {'ADMIN_TYPE',3, 'Color','red'},...
    {'CLASS',6, 'Visible','off'},...
    {'CLASS',[1 4], 'LineWidth',2});
figure
mapshow('boston_roads.shp','SymbolSpec',roadspec);
xlabel('easting in meters')
ylabel('northing in meters')
```



Map 6: GeoTIFF Image of Boston

Display the Boston GeoTIFF image; includes material (c) GeoEye™, all rights reserved.

```
figure
mapshow boston.tif
axis image manual off
```

Read Boston placenames in order to overlay on top of the GeoTIFF image.

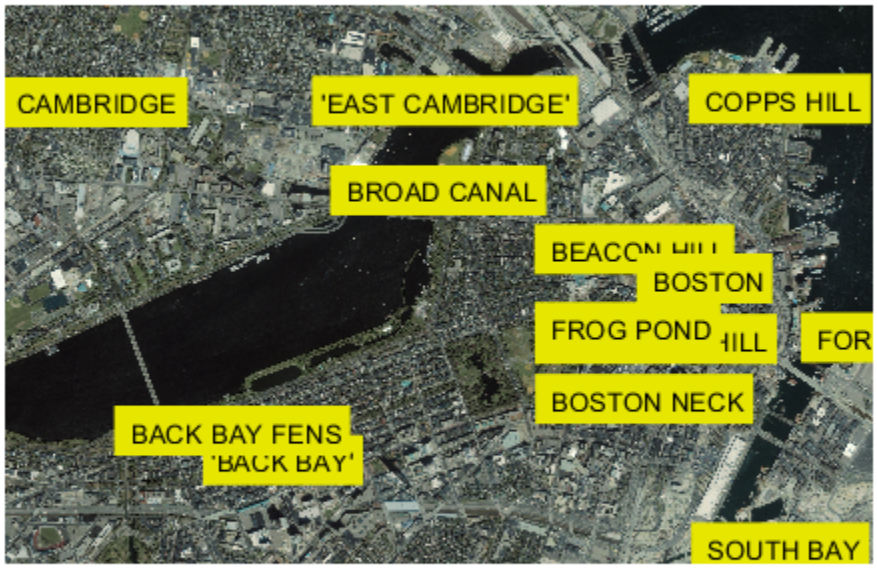
```
S = shaperead('boston_placenames.shp');
```

The projection in the GeoTIFF file is in units of survey feet. The point coordinates in the shapefile are in meters. Therefore, we need to convert the placename coordinates from meters to survey feet in order to overlay the points on the image.

```
surveyFeetPerMeter = unitsratio('sf', 'meter');
for k = 1:numel(S)
    S(k).X = surveyFeetPerMeter * S(k).X;
    S(k).Y = surveyFeetPerMeter * S(k).Y;
end
```

Display the placenames.

```
text([S.X], [S.Y], {S.NAME}, 'Color', [0 0 0], ...
    'BackgroundColor',[0.9 0.9 0],'Clipping','on');
```



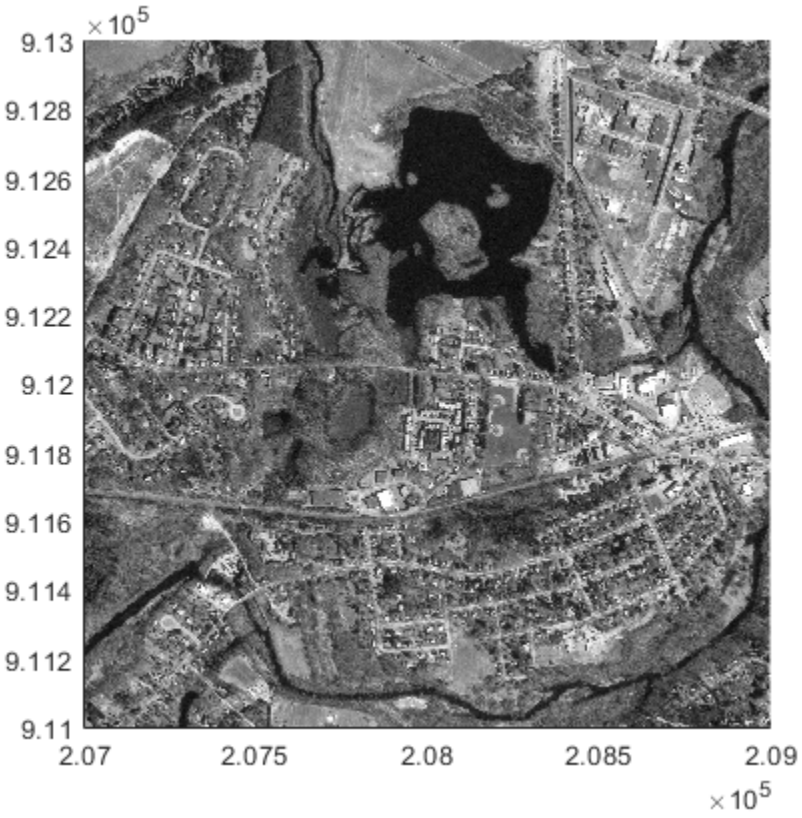
Zoom in on a selected region.

```
xlim([ 772007,  775582])
ylim([2954572, 2956535])
```

Map 7: Pond with Islands over Orthophoto Backdrop

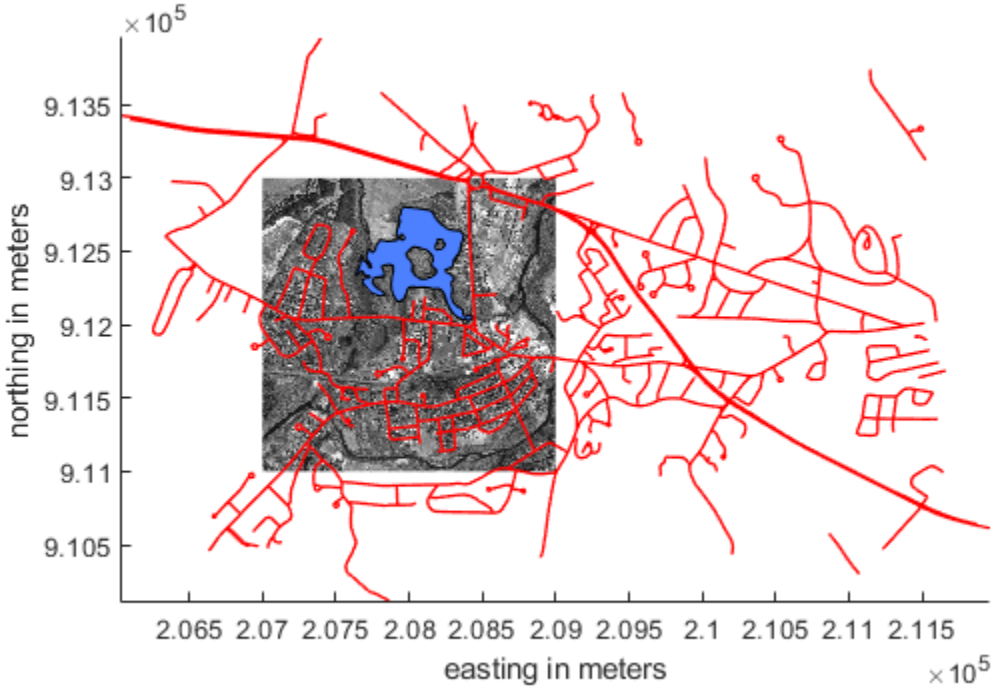
Display a pond with three large islands (feature 14 in the concord_hydro_area shapefile). Note that islands are visible in the orthophoto through three "holes" in the pond polygon. Display roads in the same figure.

```
[ortho, cmap] = imread('concord_ortho_w.tif');
R = worldfileread('concord_ortho_w.tfw', 'planar', size(ortho));
figure
mapshow(ortho, cmap, R)
```



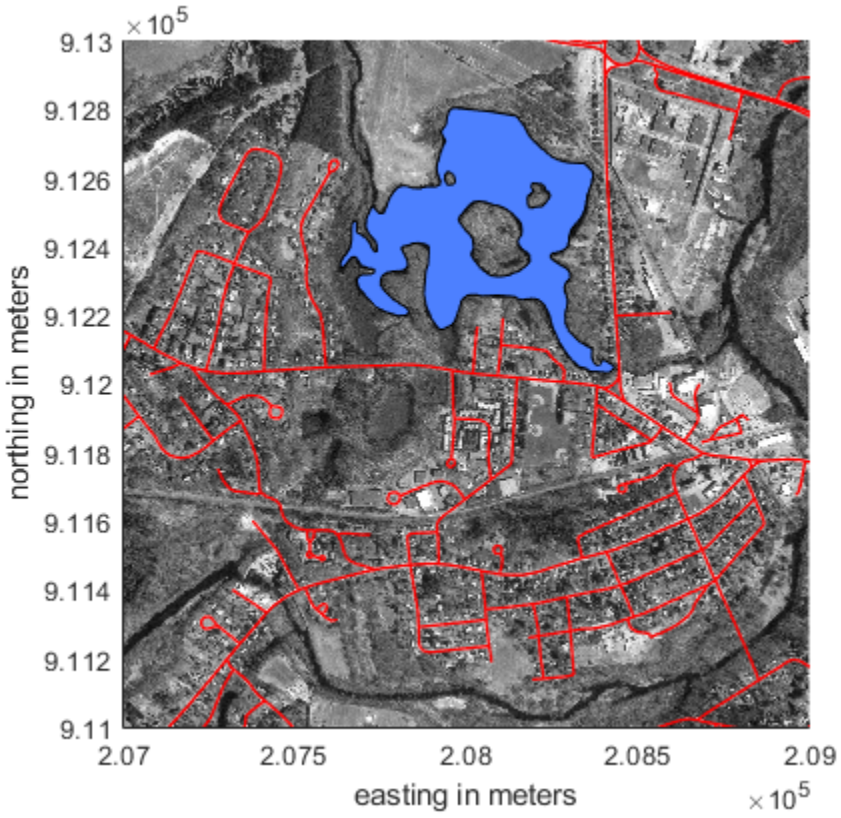
Save map limits used for image

```
xLimits = xlim;
yLimits = ylim;
pond = shaperead('concord_hydro_area.shp', 'RecordNumbers', 14);
hold on
mapshow(pond, 'FaceColor', [0.3 0.5 1], 'EdgeColor', 'black')
mapshow('concord_roads.shp', 'Color', 'red', 'LineWidth', 1);
xlabel('easting in meters')
ylabel('northing in meters')
```



Restore map limits to match image

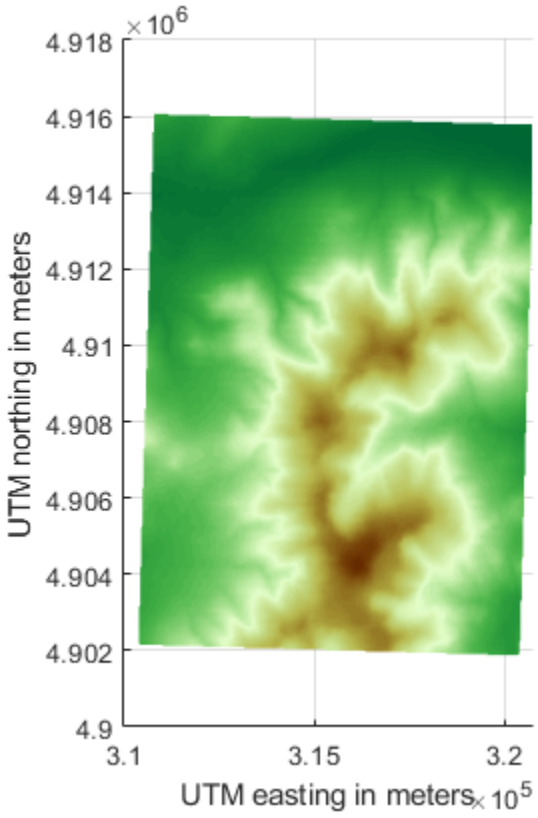
```
xlim(xLimits)
ylim(yLimits)
```



Map 8: Mount Washington SDTS Digital Elevation Model

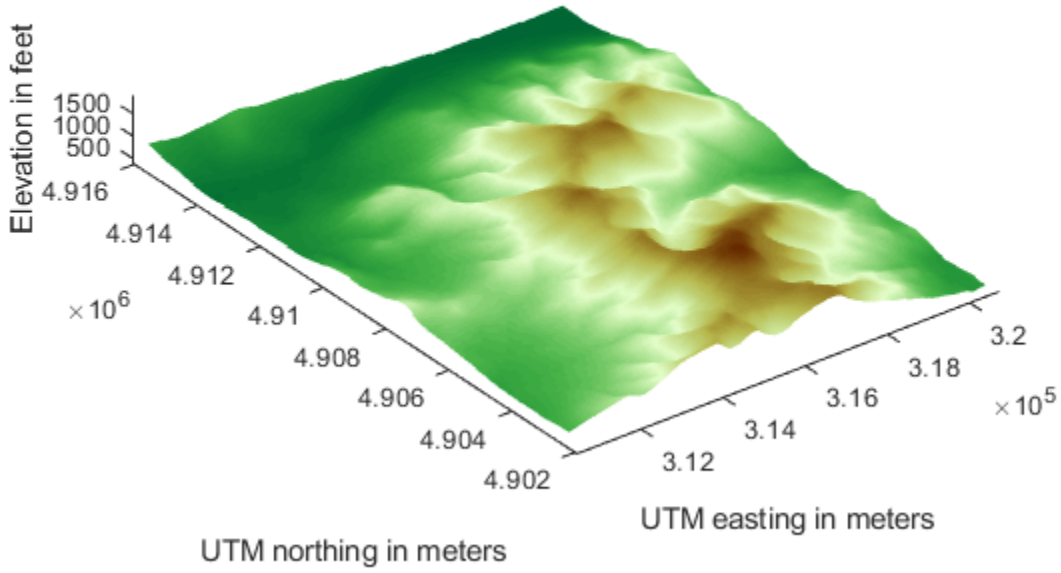
View the Mount Washington terrain data as a mesh. The data grid is georeferenced to Universal Transverse Mercator (UTM) zone 19.

```
addpath('sdts');
figure
h = mapshow('9129CATD.ddf','DisplayType','mesh');
Z = h.ZData;
demcmap(Z)
xlabel('UTM easting in meters')
ylabel('UTM northing in meters')
```



View the Mount Washington terrain data as a 3-D surface. Use the default 3-D view, which shows how the range looks from the southwest.

```
figure
mapshow('9129CATD.ddf');
demcmap(Z)
view(3);
axis equal;
xlabel('UTM easting in meters')
ylabel('UTM northing in meters')
zlabel('Elevation in feet')
```



Map 9: Mount Washington and Mount Dartmouth on One Map with Contours

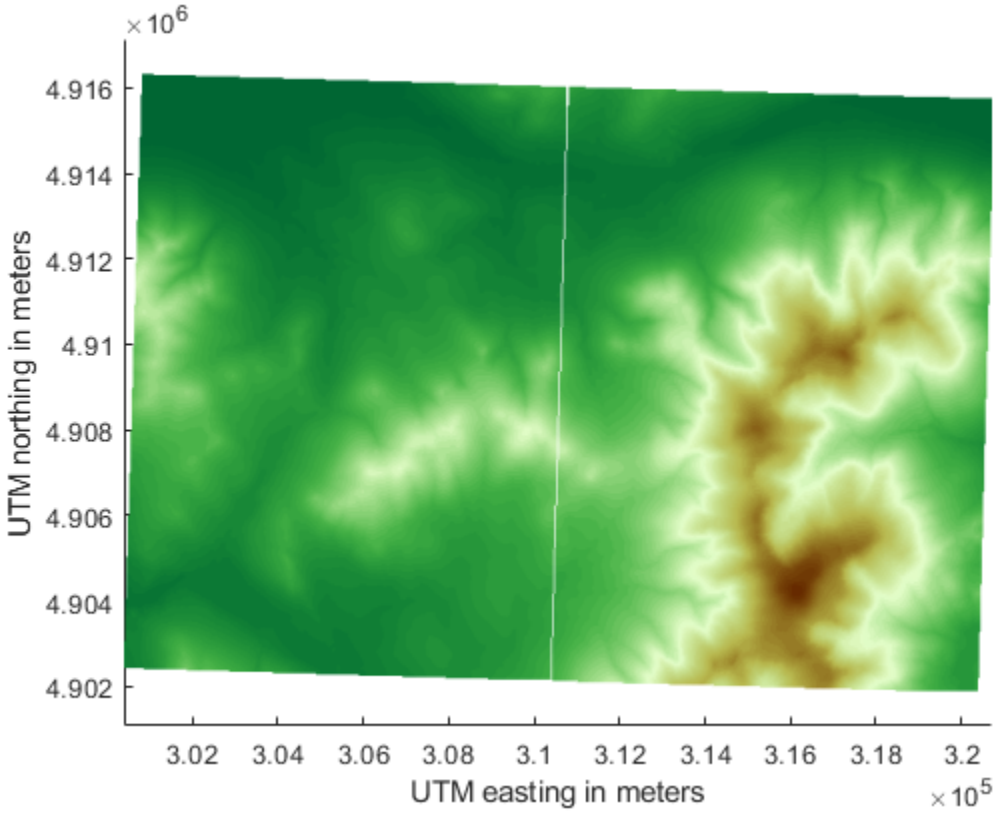
Display the grid and contour lines of Mount Washington and Mount Dartmouth.

Read the terrain data files for Mount Washington and Mount Dartmouth. To plot the data using `mapshow`, the raster data must be of type `single` or `double`. Specify the data type for the raster using the `'OutputType'` name-value pair.

```
[ZWash,RWash] = readgeoraster('MtWashington-ft.grd', ...
    'OutputType','double');
[ZDart,RDart] = readgeoraster('MountDartmouth-ft.grd', ...
    'OutputType','double');

% Find missing data using the |georasterinfo| function. The function
% returns an object with a |MissingDataIndicator| property that indicates
% which value represents missing data. Replace the missing data with |NaN|
% values using the |standardizeMissing| function.

infoWash = georasterinfo('MtWashington-ft.grd');
ZWash = standardizeMissing(ZWash,infoWash.MissingDataIndicator);

infoDart = georasterinfo('MountDartmouth-ft.grd');
ZDart = standardizeMissing(ZDart,infoDart.MissingDataIndicator);
```

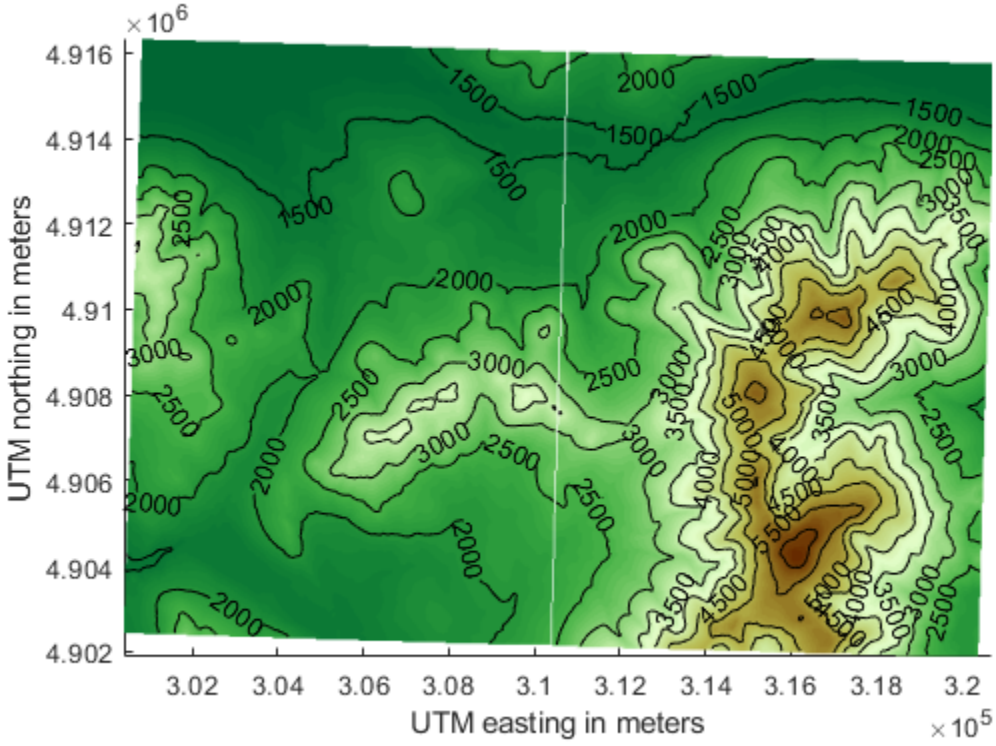
Ensure the contour lines and labels appear over the terrain data by specifying the `'ZData'` name-value pair as a matrix of zeros. Apply a colormap appropriate for terrain data using `demcmap`.

```
figure
hold on
mapshow(ZWash,RWash,'DisplayType','surface', ...
    'ZData',zeros(RWash.RasterSize))
mapshow(ZDart,RDart,'DisplayType','surface', ...
    'ZData',zeros(RDart.RasterSize))
demcmap(ZWash)
xlabel('UTM easting in meters')
ylabel('UTM northing in meters')
axis equal
```



Overlay black contour lines and labels.

```
mapshow(ZWash,RWash,'DisplayType','contour', ...
   'LineColor','k','ShowText','on');
mapshow(ZDart,RDart,'DisplayType','contour', ...
   'LineColor','k','ShowText','on');
```



Credits

boston_roads.shp, concord_roads.shp, concord_hydro_area.shp, concord_ortho_e.tif:

```
Office of Geographic and Environmental Information (MassGIS),
Commonwealth of Massachusetts Executive Office of Environmental Affairs
http://www.state.ma.us/mgis
```

boston.tif

```
Copyright GeoEye
Includes material copyrighted by GeoEye, all rights reserved.
(GeoEye was merged into the DigitalGlobe corporation January 29th,
2013.)

For more information, run:

>> type boston.txt
```

9129CATD.ddf (and supporting files):

```
United States Geological Survey (USGS) 7.5-minute Digital Elevation
Model (DEM) in Spatial Data Transfer Standard (SDTS) format for the
Mt. Washington quadrangle, with elevation in meters.
http://edc.usgs.gov/products/elevation/dem.html

For more information, run:

>> type 9129.txt
```

MtWashington-ft.grd, MountDartmouth-ft.grd:

```
MtWashington-ft.grd is the same DEM as 9129CATD.ddf, but converted to
Arc ASCII Grid format with elevation in feet.

MountDartmouth-ft.grd is an adjacent DEM, also converted to Arc ASCII
Grid with elevation in feet.

For more information, run:

>> type MtWashington-ft.txt
>> type MountDartmouth-ft.txt
```

See Also

geoshow | makesymbolspec | mapshow | shaperead

Change Map Projections 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')`).

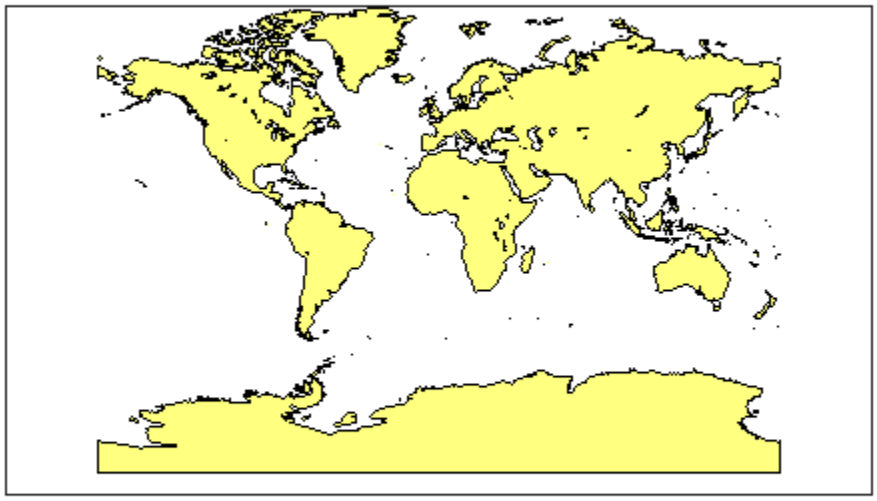
You can also use `geoshow` to display latitude-longitude data on a regular axes (created by the `axes` function, for example). When you do this, the latitude-longitude data are displayed using a pcarree, which linearly maps longitude to *x* and latitude to *y*.

Change Map Projection with Vector Data Using geoshow

This example shows how to change a map projection when displaying vector data using `geoshow` . If you need to change projections when displaying both raster and vector geodata, you can combine these techniques. Removing vector graphic objects does not affect raster data already displayed.

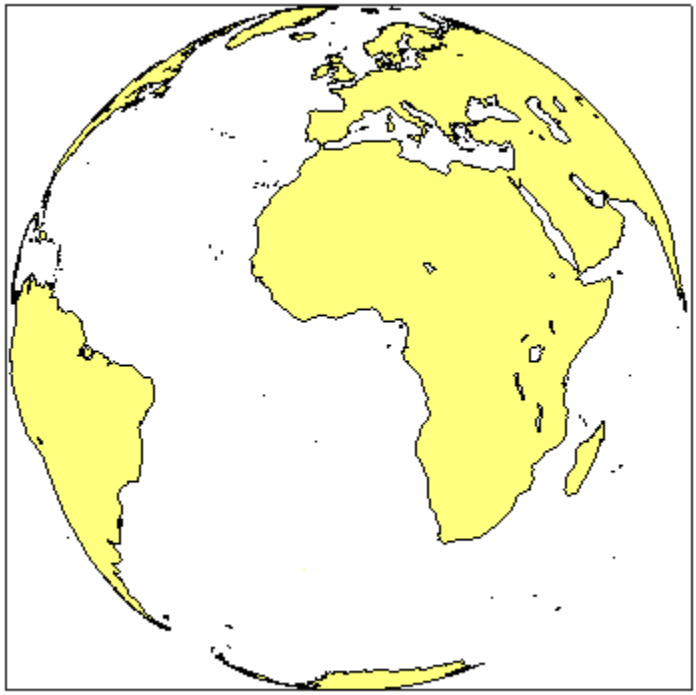
Display vector data using `geoshow`.

```
figure;
axesm miller
h = geoshow('landareas.shp');
```



Delete the original map and change the projection.

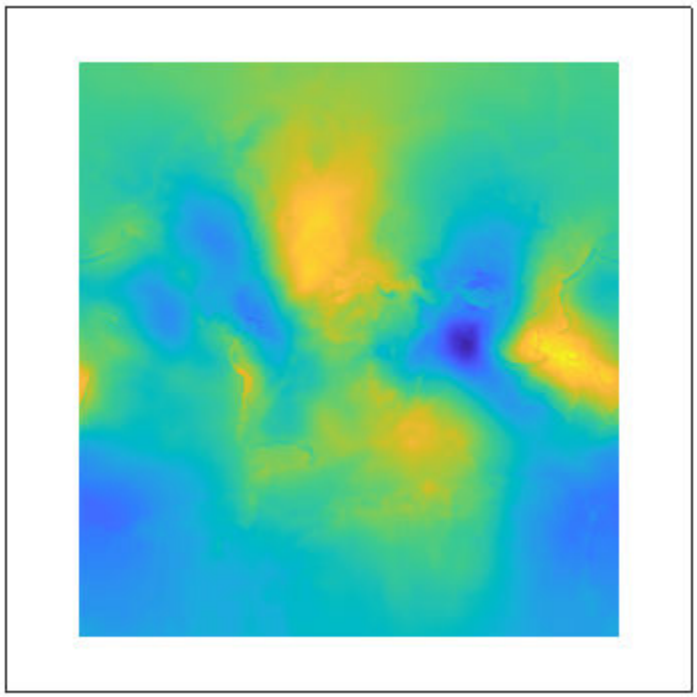
```
delete(h)
setm(gca,'mapprojection','ortho')
geoshow('landareas.shp')
```



Change Map Projection with Raster Data Using geoshow

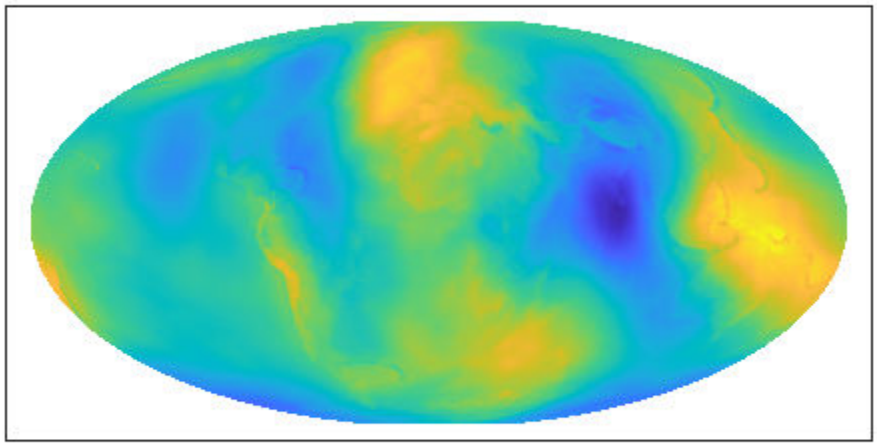
Get geoid heights and a geographic postings reference object from the EGM96 geoid model. Then, display the data using a Mercator projection.

```
[N,R] = egm96geoid;
axesm mercator
geoshow(N,R,'DisplayType','surface')
```



Change the projection using the `setm` function.

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

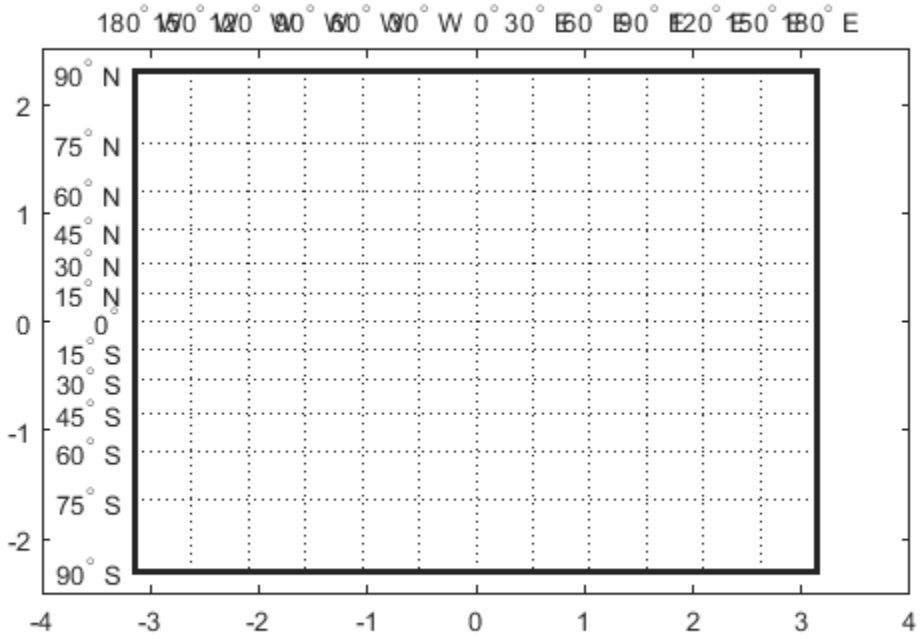


Use Geographic and Nongeographic Objects in Map Axes

This example shows how to use geographic and nongeographic objects in a map axes. The example illustrates the difference between using MATLAB functions, such as `plot` and `grid`, and their Mapping Toolbox counterparts, `plotm` and `gridm` .

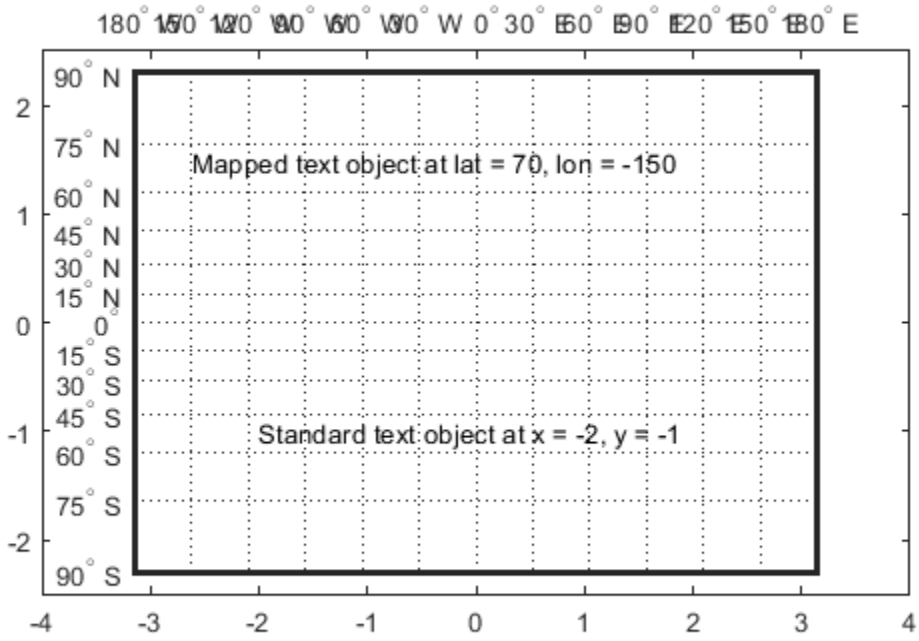
Make a Miller map axes with a latitude and longitude grid. These functions create a map axes object, a map frame enclosing the region of interest, and geographic grid lines. The x-y axes, which are normally hidden, are displayed, and the axes x-y grid is turned off. The `gridm` function constructs lines to illustrate the latitude-longitude grid, unlike the MATLAB `grid` function, 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.

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



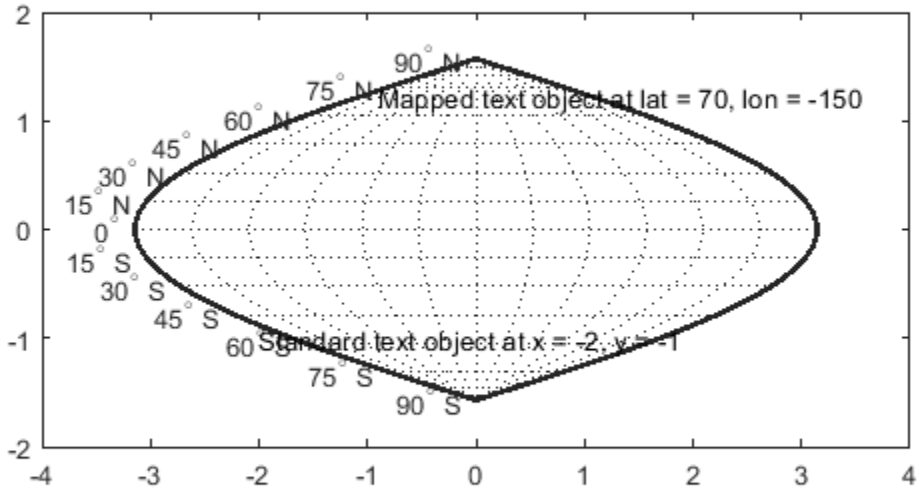
Place a standard MATLAB text object and a mapped text object, using the two separate coordinate systems. In the figure, a standard text object is placed at x=-2 and y=-1, while the mapped text object is placed at (70 degrees N, 150 degrees W) in the Miller projection.

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



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. Similarly, vector and raster (matrix) data can be displayed using either mapping functions (`plotm`) or standard functions (`plot`).

```
setm(gca,'MapProjection','sinusoid')
showaxes;
grid off;
mlabel off
```


Mapped text object at lat = 70, lon = -150
Standard text object at x = -2, y = -1
90° N
75° N
60° N
45° N
30° N
15° N
0°
15° S
30° S
45° S
60° S
75° S
90° S
2
1
0
-1
-2
-4
-3
-2
-1
0
1
2
3
4

The Map Frame

The Mapping Toolbox map frame is the outline of the limits of a map, often in the form of a *box*, the "edge of the world," so to speak. The frame is displayed if the map axes property `Frame` is set to `'on'`. This can be accomplished upon map axes creation with `axesm`, or later with `setm`, or with the direct command `framem on`. The frame is geographically defined as a latitude-longitude quadrangle that is projected appropriately. For example, on a map of the world, the frame might extend from pole to pole and a full 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.

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.

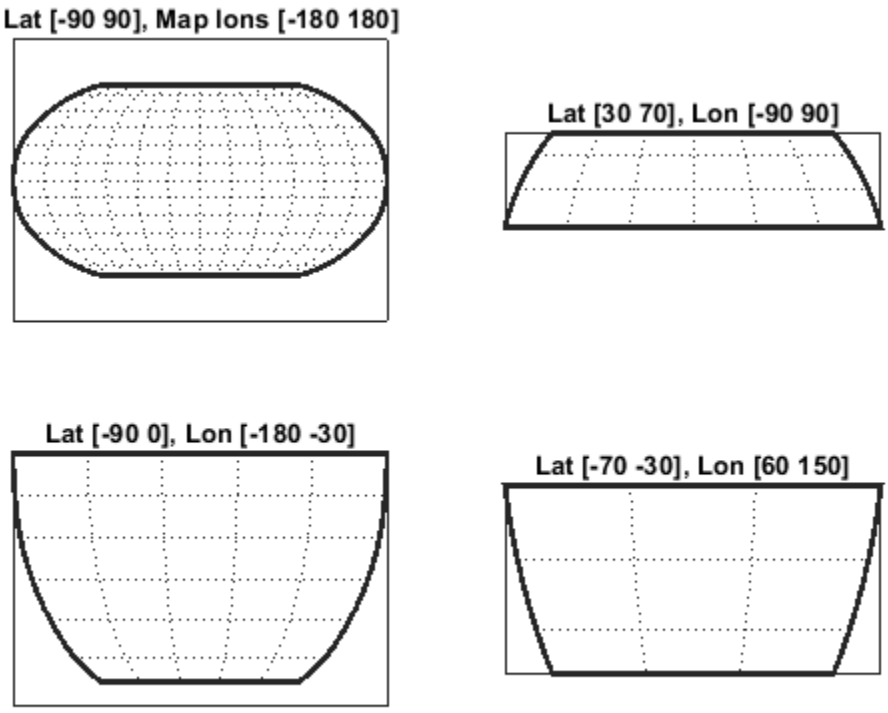
See the `framem` reference page for more details.

Plot Regions of Robinson Frame and Grid Using Map Limits

This example shows how to plot four regions of Robinson frame and grid using map limits. Initially, each of the plots shows the entire world, `FLatLimit` is [-90 90], and `FLonLimit` is `[-180 180]` for each case. The frame quadrangle can encompass smaller regions, as well, in which case the shape is a section of a full-world outline or simply a quadrilateral with straight or curving sides.

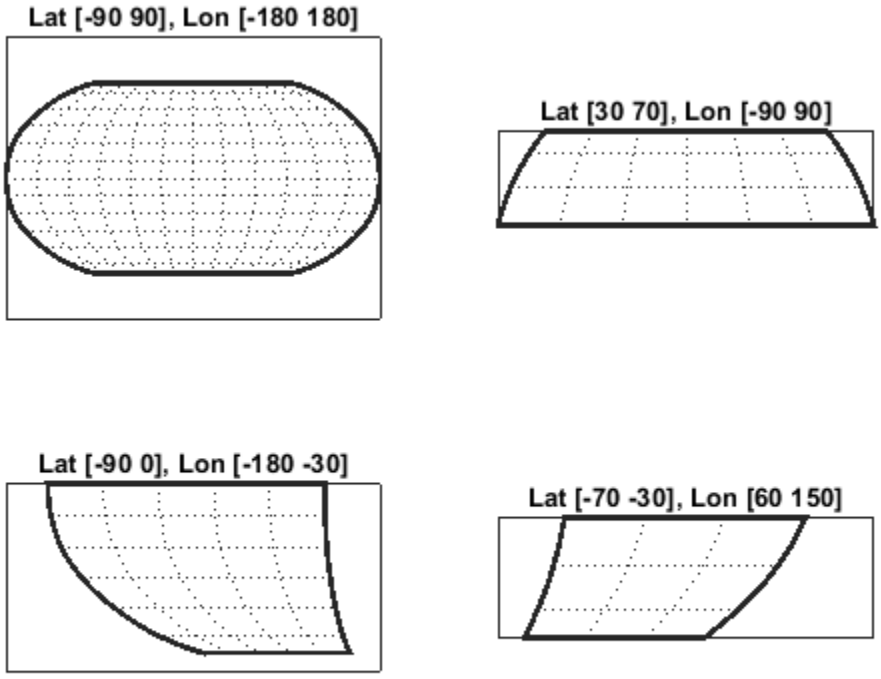
Plot four quadrangles in the Robinson Projection, symmetric about prime meridian.

```
figure('color','white')
subplot(2,2,1);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
title('Lat [-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('Lat [30 70], Lon [-90 90]','FontSize',10)
subplot(2,2,3);
axesm('MapProjection','robinson',...
    'MapLatLimit',[-90 0],'MapLonLimit',[-180 -30],....
    'Frame','on','Grid','on')
title('Lat [-90 0], Lon [-180 -30]','FontSize',10)
subplot(2,2,4);
axesm('MapProjection','robinson',...
    'MapLatLimit',[-70 -30],'MapLonLimit',[60 150],...
    'Frame','on','Grid','on')
title('Lat [-70 -30], Lon [60 150]','FontSize',10)
```



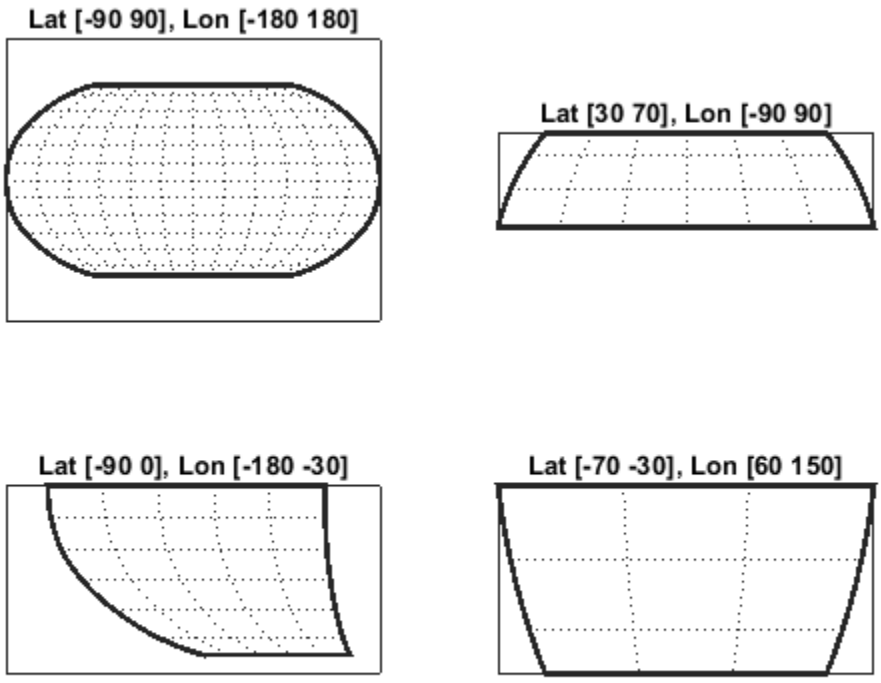
Plot the same regions but with frame limits altered after projecting. The projections are not centered on the prime meridian. Instead, the projections are symmetric about map limits.

```
figure('color','white')
h11 = subplot(2,2,1);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
title('Lat [-90 90], Lon [-180 180]')
h12 = subplot(2,2,2);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
setm(h12,'FLatLimit',[30 70],'FLonLimit',[-90 90])
title('Lat [30 70], Lon [-90 90]')
h21 = subplot(2,2,3);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
setm(h21,'FLatLimit',[-90 0],'FLonLimit',[-180 -30])
title('Lat [-90 0], Lon [-180 -30]')
h22 = subplot(2,2,4);
axesm('MapProjection','robinson',...
    'Frame','on','Grid','on')
setm(h22,'FLatLimit',[-70 -30],'FLonLimit',[60 150])
title('Lat [-70 -30], Lon [60 150]')
```



To create a symmetric frame in the lower right subplot, reset the map limits instead of the frame limits, but be sure to reset the origin.

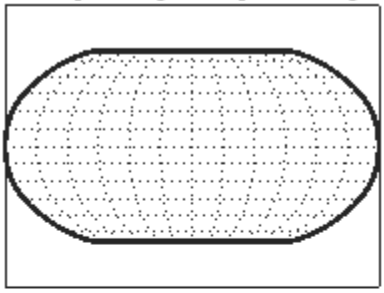
```
setm(h22,'MapLonLimit',[60 150],'Origin',[])
```



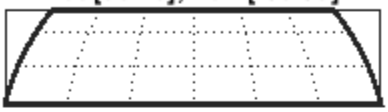
Alter the properties of the frame, which is actually a patch with face color set to `'none'`. Set the face color to `'cyan'`.

```
setm(gca,'FFaceColor','cyan')
```

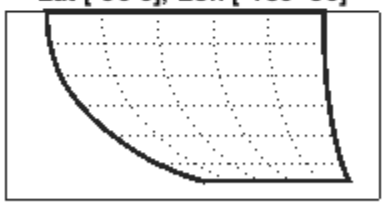
Lat [-90 90], Lon [-180 180]



Lat [30 70], Lon [-90 90]



Lat [-90 0], Lon [-180 -30]



Lat [-70 -30], Lon [60 150]

Map and Frame Limits

The Mapping Toolbox map and frame limits are two related map axes properties that limit the map display to a defined region. The map latitude and longitude limits define the extents of geodata to be displayed, while the frame limits control how the frame fits around the displayed data. Any object that extends outside the frame limits is automatically trimmed.

The frame limits are also specified differently from the map limits. The map limits are in absolute geographic coordinates referenced to an origin at the intersection of the prime meridian and the equator, while the frame limits are referenced to the rotated coordinate system defined by the map axes origin.

For all nonazimuthal projections, frame limits are specified as quadrangles (`[`*`latmin latmax`*`]` and `[`*`longmin longmax`*`]`) in the frame coordinate system. In the case of azimuthal projections, the frames are circular and are described by a polar coordinate system. One of the frame latitude limits must be a negative infinity (`-Inf`) to indicate an azimuthal frame (think of this as the center of the circle), while the other limit determines the radius of the circular frame (*`rlatmax`*). The longitude limits of azimuthal frames are inconsequential, since a full circle is always displayed.

If you are uncertain about the correct format for a particular projection frame limit, you can reset the formats to the default values using empty matrices.

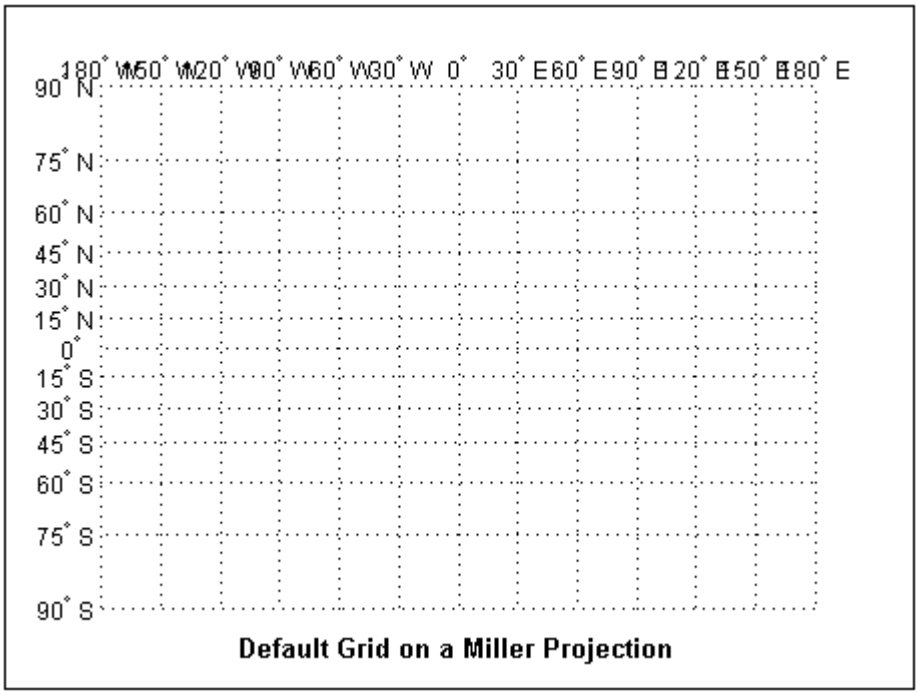
`Note` For nonazimuthal projections in the normal aspect, the map extent is limited by the minimum of the map limits and the frame limits; hence, the two limits will coincide after evaluation. Therefore, if you manually change one set of limits, you might want to clear the other set to get consistent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

The Map Grid

The *map grid* is the set of displayed meridians and parallels, also known as a *graticule*. Display the grid by setting the map axes property `Grid` to `'on'`. You can do this when you create map axes with `axesm`, with `setm`, or with the direct command `gridm on`.

Control 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° for meridians and every 15° for parallels.



Default Grid on a Miller Projection

Layer Grids

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.

Limit 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.

Label Grids

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

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

Properties affecting grid labeling are listed below.

Property	E ect
`MeridianLabel`	Toggle display of meridian labels
`ParallelLabel`	Toggle display of parallel labels
`MlabelLocation`	Alternate interval for labeling meridians
`PlabelLocation`	Alternate interval for labeling parallels
`MlabelParallel`	Keyword or latitude for placing meridian labels
`PlabelMeridian`	Keyword or longitude for placing parallel labels
`mlabelzero22pi`(function)	Relabel meridians with positive angle from 0° to 360°

For complete descriptions of all map axes properties, refer to the `axesm` reference page.

Summary of Polygon Display Functions

The following table lists the available Mapping Toolbox patch polygon display functions.

Function	Used For
`fillm`	Filled 2-D map polygons
`fill3m`	Filled 3-D map polygons in 3-D space
`geoshow`	Display map latitude and longitude data in 2-D
`mapshow`	Display map data without projection in 2-D
`patchm`	Patch objects projected on map axes
`patchesm`	Patches projected as individual objects on map axes

The `fillm` function makes use of the low-level function `patchm`. The toolbox provides another patch drawing function called `patchesm`. The optimal use of either depends on the application and user preferences. The `patchm` function creates one displayed object, which can contain multiple faces that do not necessarily connect. Mapping Toolbox data arrays contain NaNs to separate unconnected patch faces, unlike MATLAB patch display functions, which cannot handle NaN-delimited data for patches. The `patchesm` function, on the other hand, treats each face as a separate object and returns an array of patch objects. In general, `patchm` requires more memory but is faster than `patchesm`. The `patchesm` function is useful if you need to manipulate the appearance of individual patches (as thematic maps often require).

The `geoshow` and `mapshow` functions provide a superset of functionality for displaying unprojected and projected geodata, respectively, in two dimensions. These functions accept geographic data structures (geostructs and mapstructs) and coordinate vector arrays, but can also directly read shapefiles and geolocated raster files. With them, you can map polygon data, controlling rendering by constructing *symbolspecs*, data structures that you can construct with the `makesymbolspec` function. You can easily construct symbolspecs for point and line data as well as polygon data to control its display in `geoshow`, `mapshow`, and `mapview`.

See Also

More About

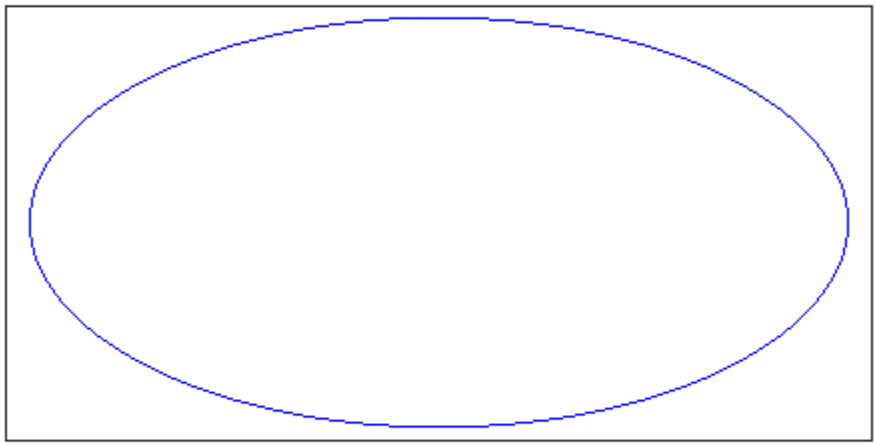
- “Create and Display Polygons” on page 2-12

Display Vector Data as Points and Lines

This example shows how to display vector data as points and lines. Mapping Toolbox vector map display of line objects works much like MATLAB line display functions. Mapping Toolbox supports versions of many MATLAB functions that work with geographic coordinates and map projections.

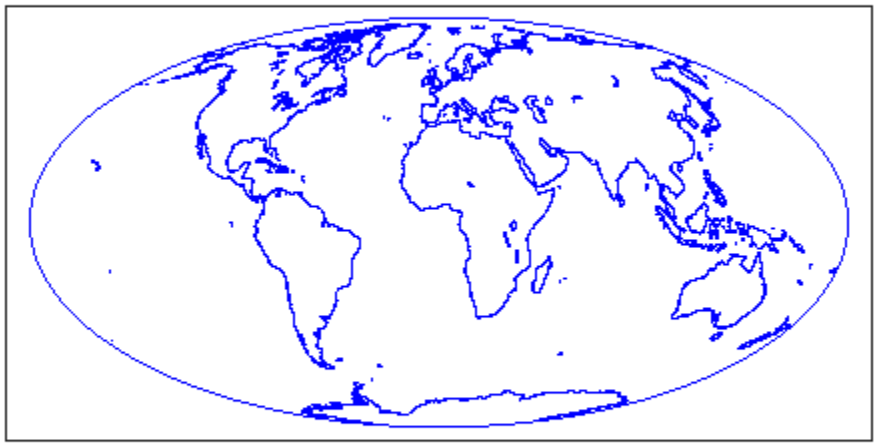
Set up a map axes and frame.

```
load coastlines
axesm mollweid
framem('FEdgeColor','blue','FLineWidth',0.5)
```



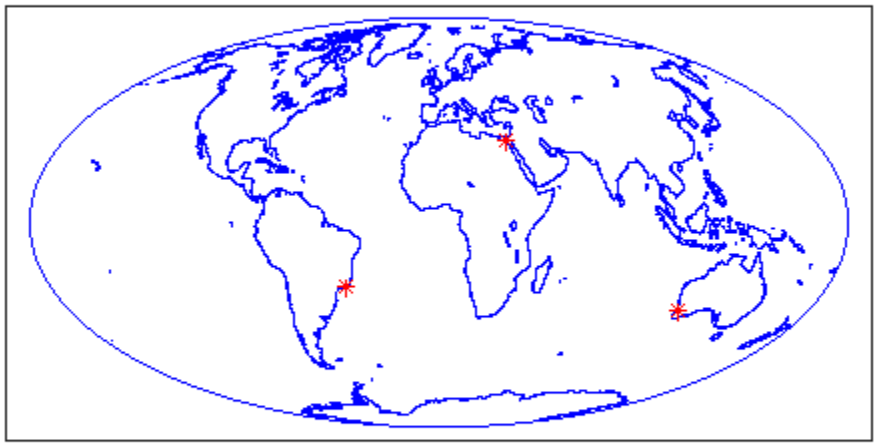
Plot the coast vector data using `plotm` and specify line property names and values.

```
plotm(coastlat,coastlon,'LineWidth',1,'Color','blue')
```



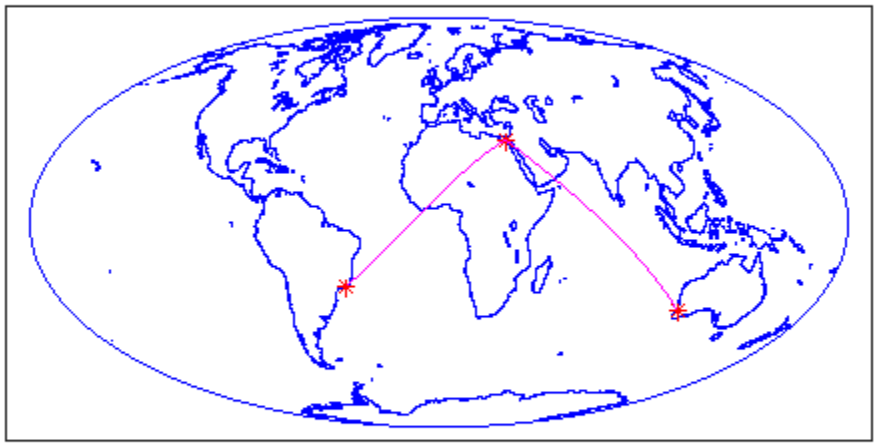
Define the three city geographic locations and plot symbols at these locations. Suppose you have variables representing the locations of Cairo (30 degrees N, 32 degrees E), Rio de Janeiro (23 degrees S, 43 degrees W), and Perth (32 degrees S, 116 degrees E), and you want to plot them as markers only, without connecting line segments. You can also use `geoshow` (for data in geographic coordinates) or `mapshow` (for data in projected coordinates) to create such maps in either a map axes or a regular axes.

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



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-')
```



Display Vector Maps as Lines or Patches

This example shows how to display vector maps as lines or patches (filled-in polygons). Mapping Toolbox functions let you display patch vector data that uses NaNs to separate closed regions.

Use the `who` command to examine the contents of the `conus` (conterminous U.S.) MAT-file and then load it into the workspace. Vector map data for lines or polygons can be represented by simple coordinate arrays, geostructs, or mapstructs. The variables `uslat` and `uslon` together describe three polygons (separated by NaNs) the largest of which represent 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.

```
who -file conus.mat
```

```
Your variables are:

description  gtlakelon    statelat     uslat
gtlakelat    source       statelon     uslon
```

```
load conus
```

Verify that line and polygon data contains NaNs (hence multiple objects).

```
find(isnan(gtlakelon))
```

```
ans = 3×1

         881
        1056
        1227
```

Read the `worldrivers` shapefile for the region that covers the conterminous United States.

```
uslatlim = [min(uslat) max(uslat)]
```

```
uslatlim = 1×2

   25.1200   49.3800
```

```
uslonlim = [min(uslon) max(uslon)]
```

```
uslonlim = 1×2

 -124.7200  -66.9700
```

```
rivers = shaperead('worldrivers', 'UseGeoCoords', true, ...
    'BoundingBox', [uslonlim', uslatlim'])
```

```
rivers=23×1 struct array with fields:
    Geometry
    BoundingBox
    Lon
    Lat
```

```
    Name
```

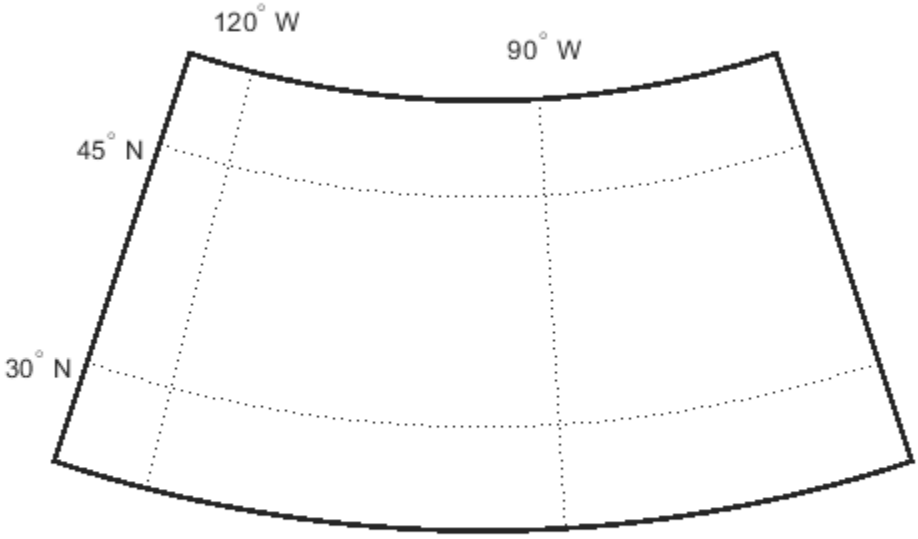
Note that the `Geometry` field specifies whether the data is stored as a `Point` , `MultiPoint` , `Line` , or `Polygon` .

```
rivers(1).Geometry
```

```
ans =
'Line'
```

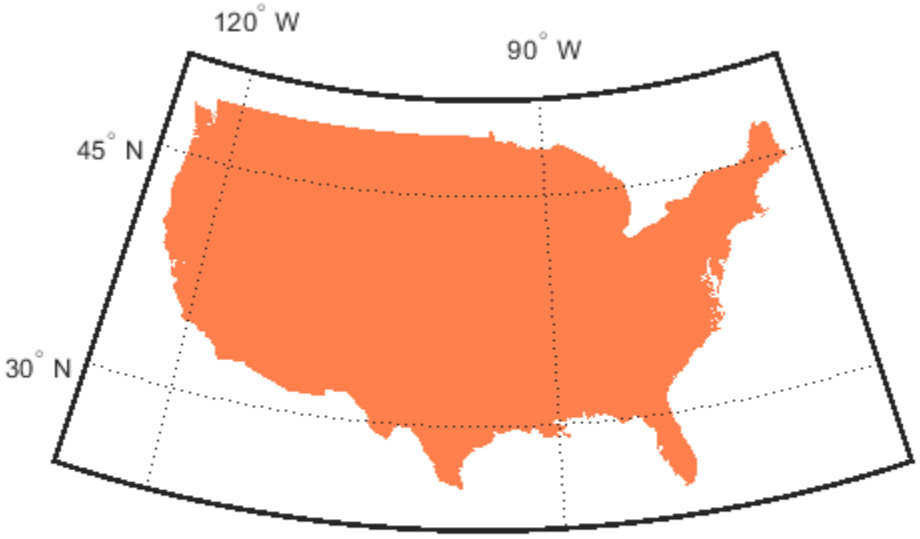
Set up a map axes to display the state coordinates, turning on the map frame, map grid, and the meridian and parallel labels.. As conic projections are appropriate for mapping the entire United States, create a map axes object using an Albers equal-area conic projection (`'eqaconic'`). Specifying map 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). 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` .

```
figure
axesm('MapProjection', 'eqaconic', 'MapParallels', [], ...
      'MapLatLimit', uslatlim + [-2 2], ...
      'MapLonLimit', uslonlim + [-2 2])
axis off;
framem;
gridm;
mlabel;
plabel
```



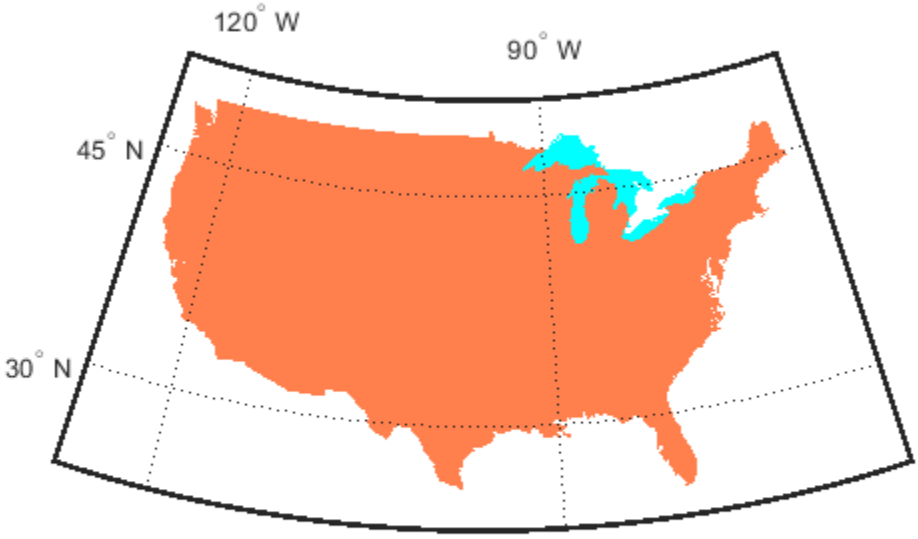
Plot a patch to display the area occupied by the conterminous United States. Use the `geoshow` function with `DisplayType` set to `'polygon'` . Note that the order in which add layers to a map can affect visibility because some layers can hide other layers. For example, because some U.S. state boundaries follow major rivers, display the rivers last to avoid obscuring them.

```
geoshow(uslat,uslon, 'DisplayType','polygon','FaceColor',...
    [1 .5 .3], 'EdgeColor','none')
```



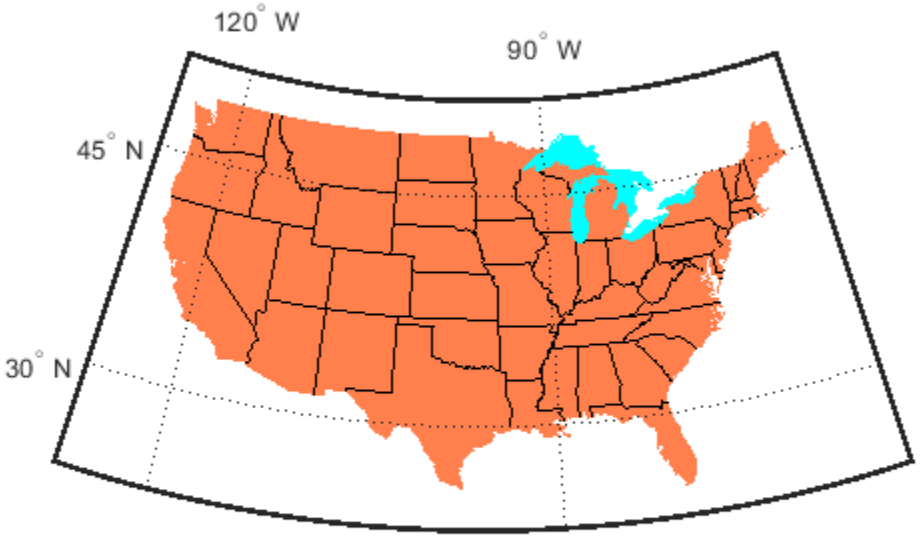
Plot the Great Lakes on top of the land area, using `geoshow` .

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



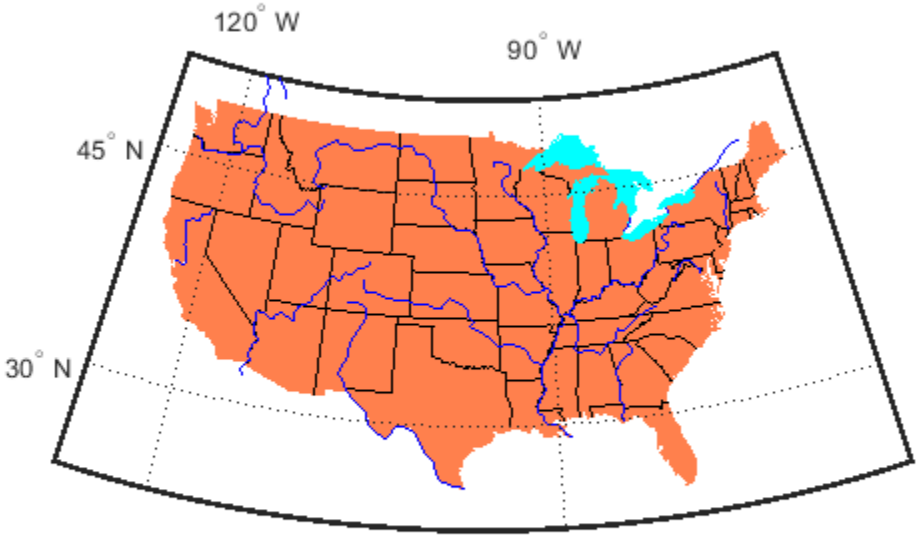
Plot the line segment data showing state boundaries, using `geoshow` with `DisplayType` set to `'line'`.

```
geoshow(statelat,statelon,'DisplayType','line','Color','k')
```



Use `geoshow` to plot the river network. Note that you can omit `DisplayType`

```
geoshow(rivers, 'Color', 'blue')
```



See Also

`axesm` | `geoshow` | `shaperead`

More About

- “Create and Display Polygons” on page 2-12

Types of Data Grids and Raster Display Functions

Mapping Toolbox functions and GUIs display both regular and geolocated data grids originating in a variety of formats. Recall that regular data grids require a *referencing vector or matrix* that describes the sampling and location of the data points, while geolocated data grids require matrices of latitude and longitude coordinates.

The data grid display functions are geographic analogies to the MATLAB surface drawing functions, but operate specifically on map axes objects. Like the line-plotting functions discussed in the previous chapter, some Mapping Toolbox grid function names correspond to their MATLAB counterparts with an `m` appended.

`Note` Mapping Toolbox functions beginning with `mesh` are used for regular data grids, while those beginning with `surf` are reserved for geolocated data grids. This usage differs from the MATLAB definition; `mesh` plots are used for colored wire-frame views of the surface, while `surf` displays colored faceted surfaces.

Surface map objects can be displayed in a variety of different ways. You can assign colors from the figure colormap to surfaces according to the values of their data. You can also display images where the matrix data consists of indices into a colormap or display the matrix as a three-dimensional surface, with the *z*-coordinates given by the map matrix. You can use monochrome surfaces that reflect a pseudo-light source, thereby producing a three-dimensional, shaded relief model of the surface. Finally, you can use a combination of color and light shading to create a lighted shaded relief map.

The following table lists the available Mapping Toolbox surface map display functions.

Function	Used For
`geoshow`	Display map data gridded in latitude and longitude in 2-D
`mapshow`	Display gridded map data without projection in 2-D
`meshm`	Regular data grid warped to projected graticule mesh
`surfm`	Geolocated data grid projected on map axes
`pcolorm`	Projected data grid in z = 0 plane
`surfacem`	Data grid warped to projected graticule mesh
`surflm`	3-D shaded surface with lighting projected on map axes
`meshlsrm`	3-D lighted shaded relief of regular data grid
`surflsrm`	3-D lighted shaded relief of geolocated data grid

Fit Gridded Data to the Graticule

The toolbox projects surface objects in a manner similar to the traditional methods of map making. A cartographer first lays out a grid of meridians and parallels called the *graticule*. Each graticule cell is a geographic quadrangle. The cartographer calculates or interpolates the appropriate *x*-*y* locations for every vertex in the graticule grid and draws the projected graticule by connecting the dots. Finally, the cartographer draws the map data freehand, attempting to account for the shape of the graticule cells, which usually change shape across the map. Similarly, the toolbox calculates the *x*-*y* locations of the four vertices of each graticule cell and warps or samples the matrix data to fit the resulting quadrilateral.

In mapping data grids using the toolbox, as in traditional cartography, the finer the mesh (analogous to using a graticule with more meridians and parallels), the greater precision the projected map display will have, at the cost of greater effort and time. The graticule in a printed map is analogous to the spacing of grid elements in a regular data grid, the Mapping Toolbox representation of which is two-element vectors of the form `[`*`number-of-parallels`*`,` *`number-of-meridians`*`]`. The graticule for geolocated data grids is similar; it is the size of the latitude and longitude coordinate matrices: *`number-of-parallels`* = `mrows-1` and *`number-of-meridians`* = `ncols-1`. However, because geolocated data grids have arbitrary cell corner locations, spacing can vary and thus their graticule is not a regular mesh.

Fit Gridded Data to Fine and Coarse Graticules

This example shows how to fit gridded data to fine and coarse graticules. The choice of graticule is a balance of speed over precision in terms of positioning the grid on the map. Typically, there is no point to specifying a mesh finer than the data resolution (in this example, 180-by-360 grid cells). In practice, it makes sense to use coarse graticules for development tasks and fine graticules for final graphics production.

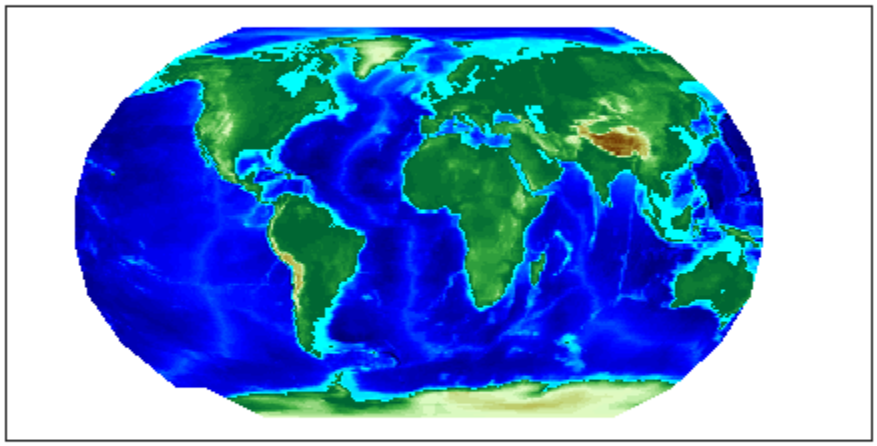
Note that, regardless of the graticule resolution, the grid data is unchanged. In this case, the data grid is a 180-by-360 matrix, and regardless of where it is positioned, the data values are unchanged.

Load elevation raster data and a geographic cells reference object.

```
load topo60c
```

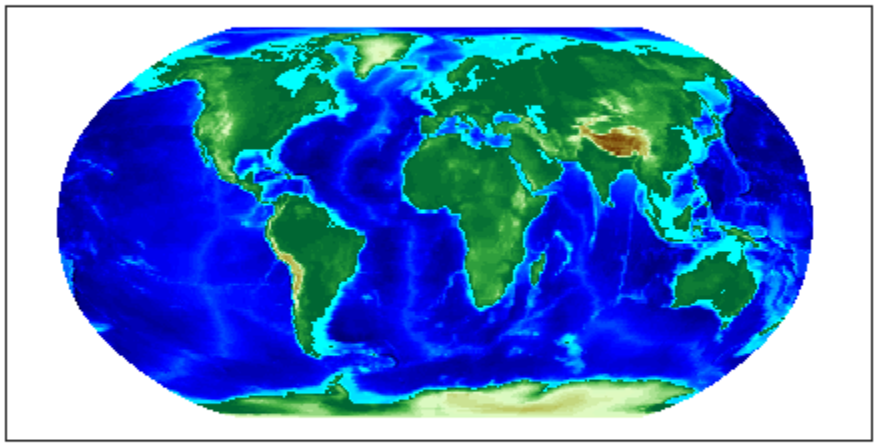
Set up a Robinson projection, specify a coarse (10-by-20) cell graticule, and display the data mapped to the graticule using a colormap appropriate for elevation data. Notice that for this coarse graticule, the edges of the map do not appear as smooth curves.

```
figure
axesm robinson
spacing = [10 20];
m = meshm(topo60c,topo60cR,spacing);
demcmap(topo60c)
```



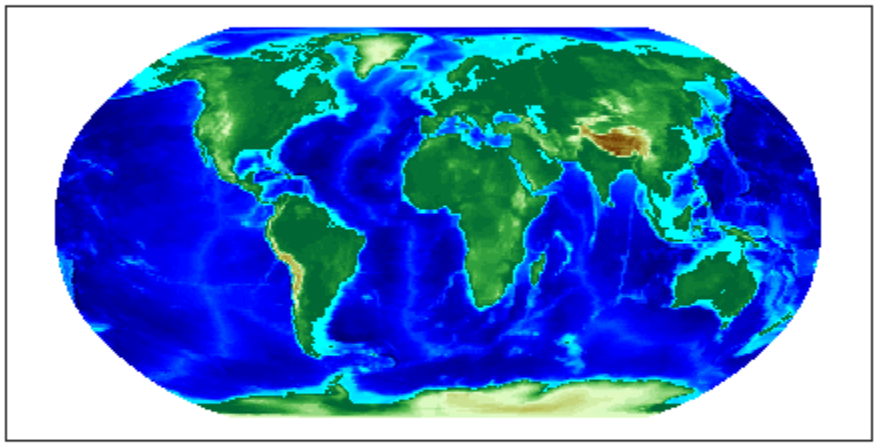
Now reset the graticule, using the `setm` function, to make it less coarse, [50 100]. (You can also reset the graticule using the `meshgrat` function.) Notice that the jagged edges effect is now negligible.

```
setm(m,'MeshGrat',[50 100])
```



Reset the graticule again, this time to a very fine grid using the `setm` function. 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. Making the mesh more precise is a trade-off of resolution versus time and memory usage.

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



Create 3-D Displays with Raster Data

This example shows how to create 3-D displays with raster data by setting up surface views, which requires explicit horizontal coordinates. 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. 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 reference vector, matrix, or object).

Load elevation data and a geographic cells reference object for the Korean peninsula. Transform the data and reference object to a fully geolocated data grid using the `meshgrat` function.

```
load korea5c
[lat,lon] = meshgrat(korea5c,korea5cR);
```

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.

```
korea5c = km2deg(korea5c/1000);
```

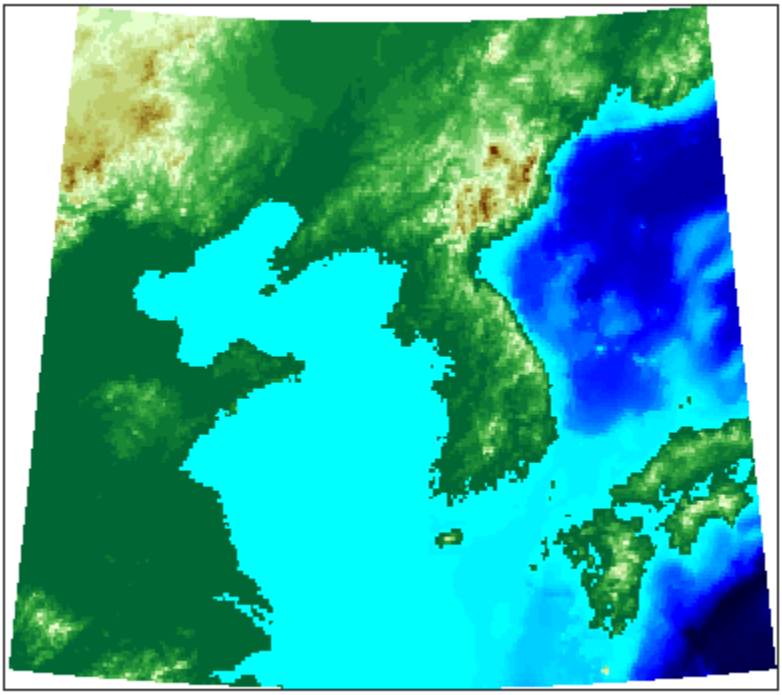
Observe the results by typing the `whos` command. The `lat` and `lon` coordinate matrices form a mesh the same size as `korea5c`. This is a requirement for constructing 3-D surfaces. In `lon`, all columns contain the same number for a given row, and in `lat`, all rows contain the same number for a given column. This is because the `mesh` produced by `meshgrat` in this case is regular, but such data grids need not have equal spacing.

```
whos
```

```
  Name               Size              Bytes  Class                                    Attribute

  description        2x64                256  char
  korea5c          180x240            345600  double
  korea5cR           1x1                 128  map.rasterref.GeographicCellsReference
  lat              180x240            345600  double
  lon              180x240            345600  double
  source             2x76                304  char
```

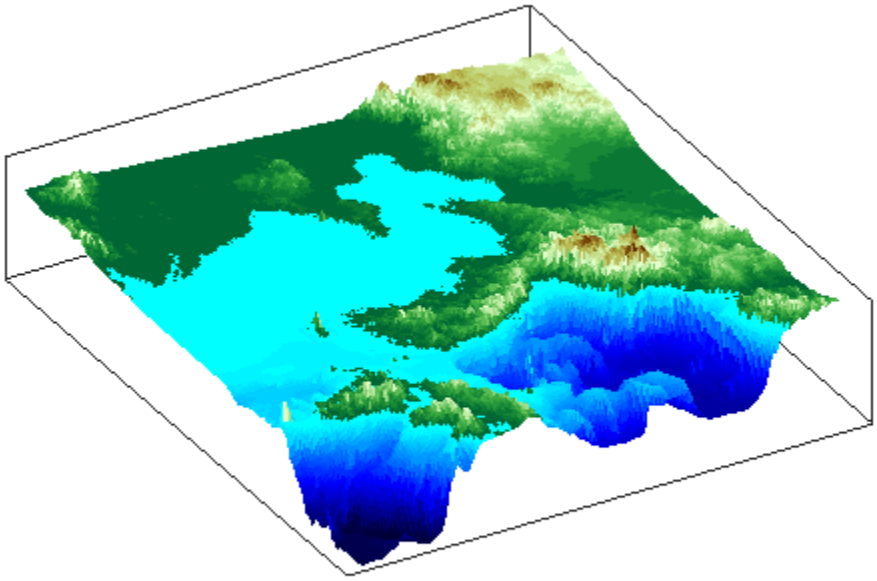
Now set up a map axes object with the equal area conic projection and, instead of using the `meshm` function to make this map, display the geolocated data grid using the `surfm` function. Set an appropriate colormap. This produces a map that is really a 3-D view seen from directly overhead (the default perspective). To appreciate that, all you need to do is to change your viewpoint.

```
axesm('MapProjection','eqaconic','MapParallels',[],...
     'MapLatLimit',[30 45],'MapLonLimit',[115 135])
surfm(lat,lon,korea5c,korea5c)
demcmap(korea5c)
tightmap
```



Specify a viewing azimuth of 60 degrees (from the east southeast) and a viewing elevation of 30 degrees above the horizon, using the `view` function.

```
view(60,30)
```



Creating Map Displays with Latitude and Longitude Data

There are many geospatial data sets that contain data with coordinates in latitude and longitude in units of degrees. This example illustrates how to import geographic data with coordinates in latitude and longitude, display geographic data in a map display, and customize the display.

In particular, this example illustrates how to

- Import specific geographic vector and raster data sets
- Create map displays and visualize the data
- Display multiple data sets in a single map display
- Customize a map display with a scale ruler and north arrow
- Customize a map display with an inset map

Example 1: Import Polygon Geographic Vector Data

Geographic vector data can be stored in a variety of different formats including standard file formats such as shapefiles, GPS Exchange (GPX), NetCDF, HDF4, or HDF5 and specific vector data products such as Vector Smart Map Level 0 (VMAP0), and the Global Self-Consistent Hierarchical High-Resolution Geography (GSHHG). This example imports polygon geographic vector data from a shapefile. Vertices in a shapefile can be either in geographic coordinates (latitude and longitude) or in a projected coordinate reference system.

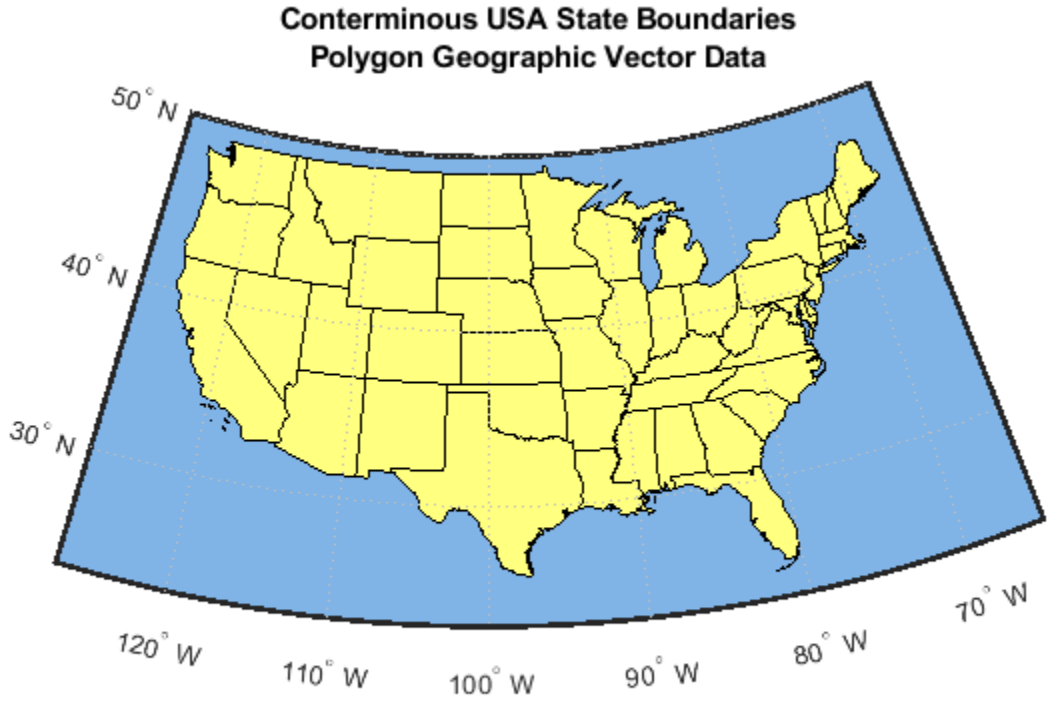
Read USA state boundaries from the `usasatehi.shp` file included with the Mapping Toolbox™ software. The state boundaries are in latitude and longitude. Convert the geostruct array to a `geoshape vector` and store the results in `states`.

```
states = geoshape(shaperead('usastatehi', 'UseGeoCoords', true));
```

Example 2: Display Polygon Geographic Vector Data

Display the polygon geographic vector data onto a map axes. Since the geographic extent is in the United States, you can use `usamap` to setup a map axes. Use `geoshow` to project and display the geographic data onto the map axes. Display an ocean color in the background by setting the frame's face color.

```
figure
ax = usamap('conus');
oceanColor = [.5 .7 .9];
setm(ax, 'FFaceColor', oceanColor)
geoshow(states)
title({ ...
    'Conterminous USA State Boundaries', ...
    'Polygon Geographic Vector Data'})
```



Example 3: Import Point and Line Geographic Vector Data

Import point geographic vector data from the `boston_placenames.gpx` file included with the Mapping Toolbox™ software. The file contains latitude and longitude coordinates of geographic point features in part of Boston, Massachusetts, USA. Use `gpxread` to read the GPX file and return a `geopoint vector`.

```
placenames = gpxread('boston_placenames');
```

Import line vector data from the `sample_route.gpx` file included with the Mapping Toolbox™ software. The file contains latitude and longitude coordinates for a GPS route from Boston Logan International Airport to The MathWorks, Inc in Natick Massachusetts, USA. Use `gpxread` to read the GPX file and return a `geopoint vector`.

```
route = gpxread('sample_route.gpx');
```

Example 4: Display Point and Line Geographic Vector Data

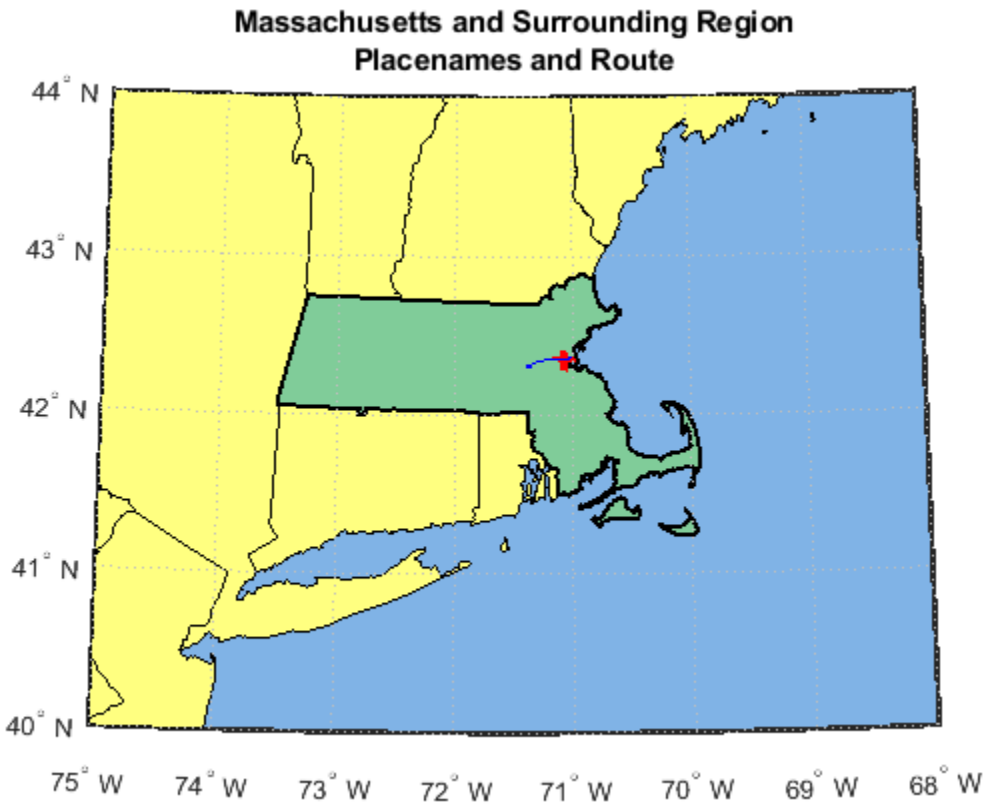
Display the geographic vector data in a map axes centered around the state of Massachusetts, using the data from the state boundaries and the GPX files. The coordinates for all of these data sets are in latitude and longitude.

Find the state boundary for Massachusetts.

```
stateName = 'Massachusetts';
ma = states(strcmp(states.Name, stateName));
```

Use `usamap` to setup a map axes for the region surrounding Massachusetts. Color the ocean by setting the frame's face color. Display the state boundaries and highlight Massachusetts by using `geoshow` to display the geographic data onto the map axes. Since the GPX route is a set of points stored in a `geopoint vector`, supply the latitude and longitude coordinates to `geoshow` to display the route as a line.

```
figure
ax = usamap('ma');
setm(ax, 'FFaceColor', oceanColor)
geoshow(states)
geoshow(ma, 'LineWidth', 1.5, 'FaceColor', [.5 .8 .6])
geoshow(placenames);
geoshow(route.Latitude, route.Longitude);
title({'Massachusetts and Surrounding Region', 'Placenames and Route'})
```



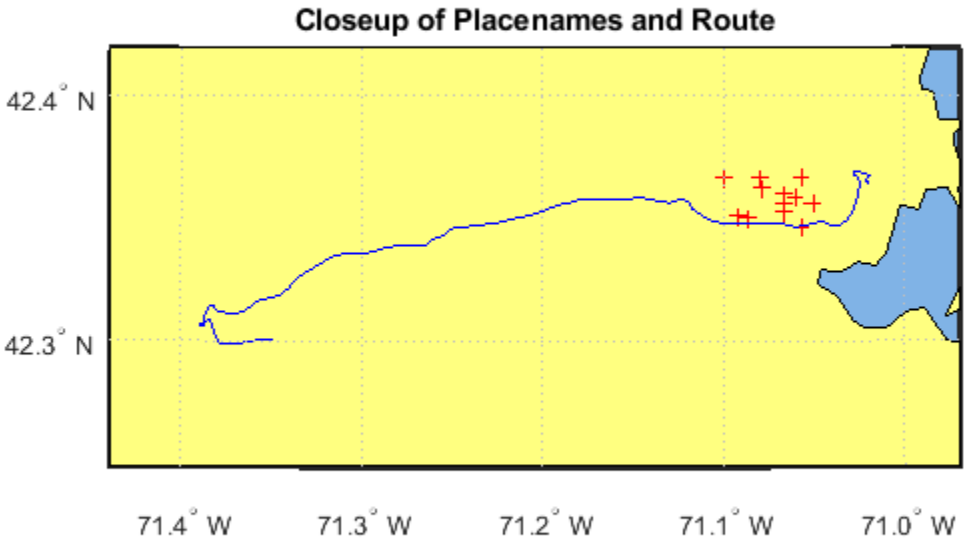
Example 5: Set Latitude and Longitude Limits Based on Data Extent

Zoom into the map by computing new latitude and longitude limits for the map using the extent of the placenames and route data. Extend the limits by .05 degrees.

```
lat = [route.Latitude placenames.Latitude];
lon = [route.Longitude placenames.Longitude];
latlim = [min(lat) max(lat)];
lonlim = [min(lon) max(lon)];
[latlim, lonlim] = bufgeoquad(latlim, lonlim, .05, .05);
```

Construct a map axes with the new limits and display the geographic data.

```
figure
ax = usamap(latlim, lonlim);
setm(ax, 'FFaceColor', oceanColor)
geoshow(states)
geoshow(placenames)
geoshow(route.Latitude, route.Longitude)
title('Closeup of Placenames and Route')
```



Example 6: Import Geographic Raster Data

Geographic raster data can be stored in a variety of different formats including standard file formats such as GeoTIFF, Spatial Data Transfer Standard (SDTS), NetCDF, HDF4, or HDF5 and specific raster data products such as DTED, GTOPO30, ETOPO, and the USGS 1-degree (3-arc-second) DEM formats. This example imports a raster data file and a worldfile of the region surrounding Boston, Massachusetts. The coordinates of the image are in latitude and longitude. Use `imread` to read the image and `worldfileread` to read the worldfile and construct a spatial referencing object.

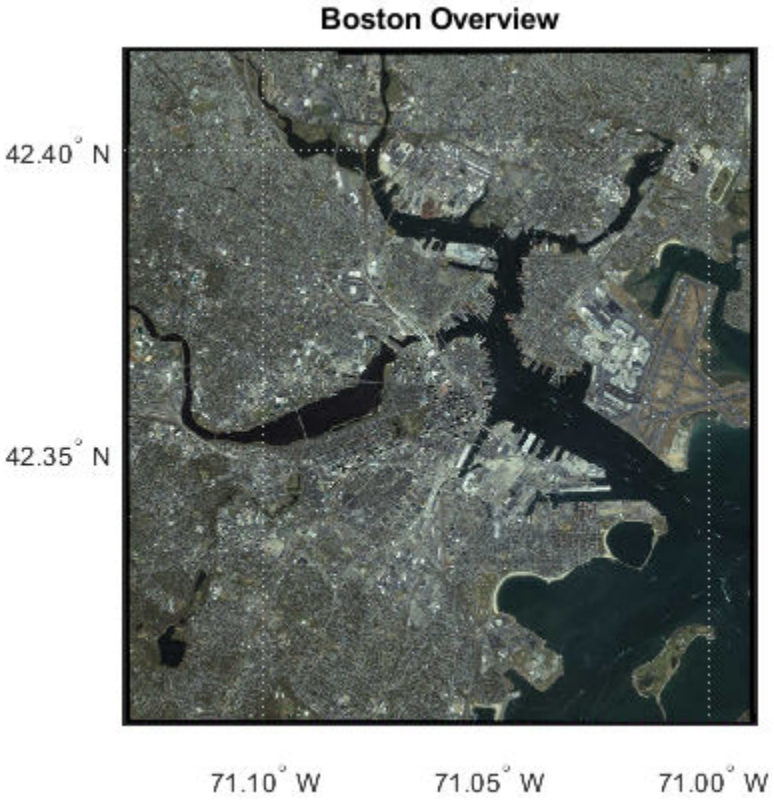
```
filename = 'boston_ovr.jpg';
RGB = imread(filename);
R = worldfileread(getworldfilename(filename), 'geographic', size(RGB));
```

Example 7: Display Geographic Raster Data

Display the RGB image onto a map axes. The limits of the map are set to the limits defined by the spatial referencing object, `R`. The coordinates of the data are in latitude and longitude.

```
figure
ax = usamap(RGB, R);
```

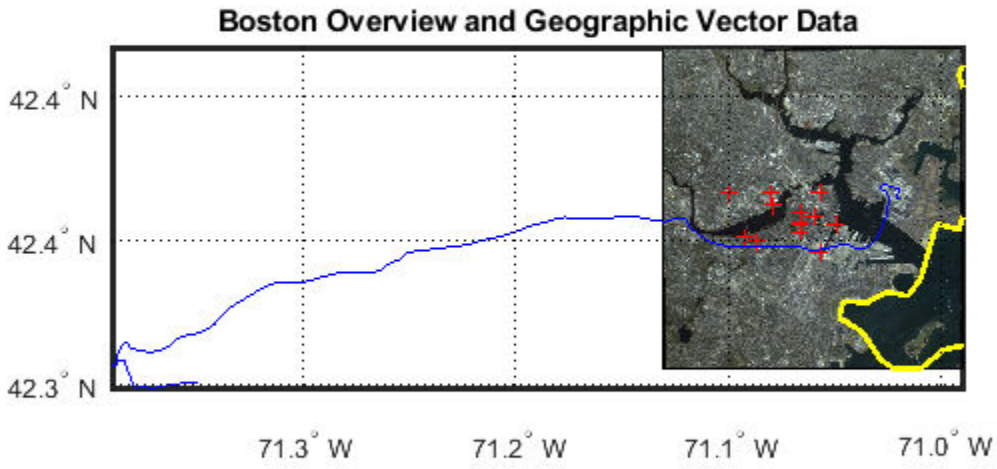
```
setm(ax, ...
    'MLabelLocation',.05, 'PLabelLocation',.05, ...
    'MLabelRound',-2, 'PLabelRound',-2)
geoshow(RGB, R)
title('Boston Overview')
```



Example 8: Display Geographic Vector and Raster Data

You can display raster and vector data in a single map display. Since the coordinates for all of these data sets are in latitude and longitude, use `geoshow` to display them in a single map display. Setup new limits based on the limits of the route, placenames, and the overview image.

```
lat = [route.Latitude  placenames.Latitude  R.LatitudeLimits];
lon = [route.Longitude placenames.Longitude R.LongitudeLimits];
latlim = [min(lat) max(lat)];
lonlim = [min(lon) max(lon)];

figure
ax = usamap(latlim, lonlim);
setm(ax, 'GColor','k', ...
    'PLabelLocation',.05, 'PLineLocation',.05)
geoshow(RGB, R)
geoshow(states.Latitude, states.Longitude, 'LineWidth', 2, 'Color', 'y')
geoshow(placenames)
geoshow(route.Latitude, route.Longitude)
title('Boston Overview and Geographic Vector Data')
```



Example 9: Customize a Map Display with a Scale Ruler

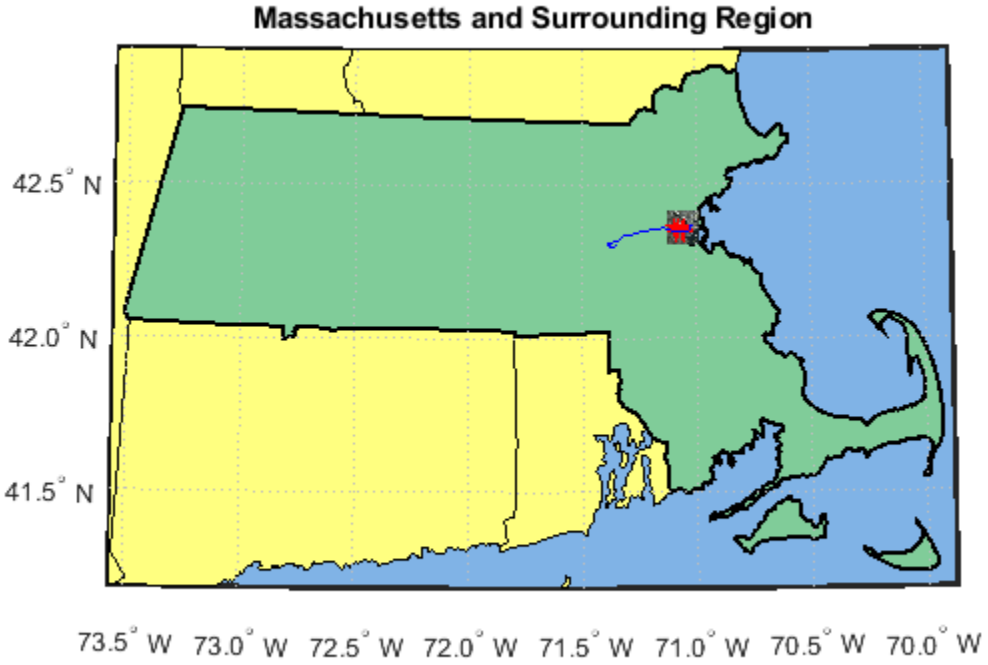
Customize a map display by including a scale ruler. A scale ruler is a graphic object that shows distances on the ground at the correct size for the projection. This example illustrates how to construct a scale ruler that displays horizontal distances in international miles.

Compute latitude and longitude limits of Massachusetts and extend the limits by .05 degrees by using the `bufgeoquad` function.

```
latlim = [min(ma.Latitude),  max(ma.Latitude)];
lonlim = [min(ma.Longitude), max(ma.Longitude)];
[latlim, lonlim] = bufgeoquad(latlim, lonlim, .05, .05);
```

Display the state boundary, placenames, route, and overview image onto the map.

```
figure
ax = usamap(latlim, lonlim);
setm(ax, 'FFaceColor', oceanColor)
geoshow(states)
geoshow(ma, 'LineWidth', 1.5, 'FaceColor', [.5 .8 .6])
geoshow(RGB, R)
geoshow(placenames)
geoshow(route.Latitude, route.Longitude)
titleText = 'Massachusetts and Surrounding Region';
title(titleText)
```

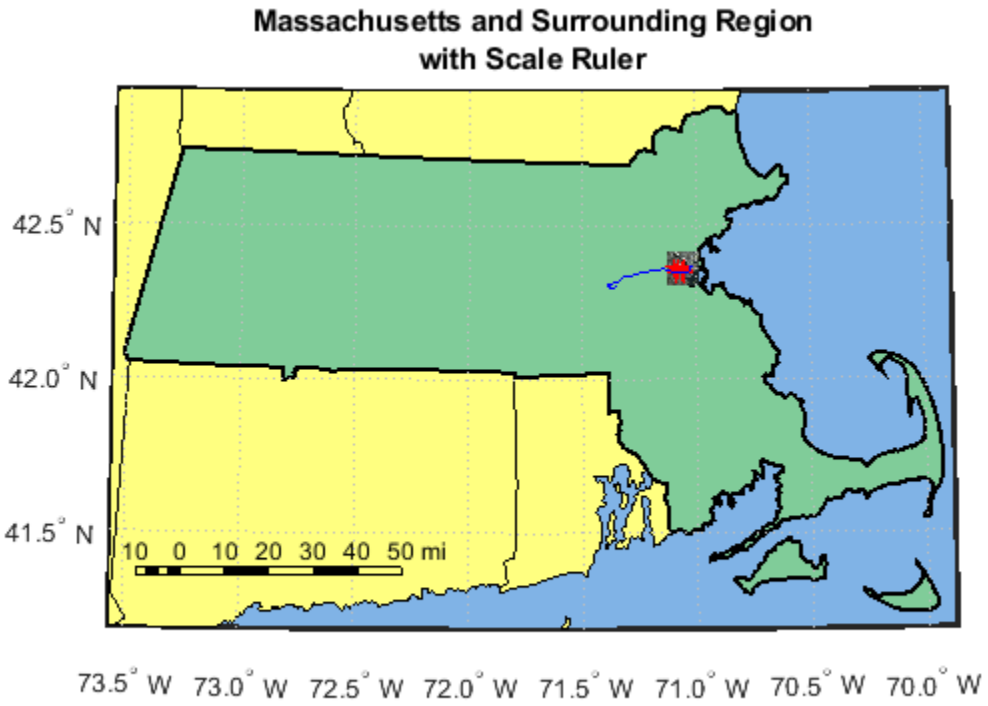


Insert a scale ruler. You can determine a location for the scale ruler by using the `ginput` function as shown below:

```
[xLoc,yLoc] = ginput(1);
```

A location previously chosen is set below.

```
xLoc = -127800;
yLoc = 5014700;
scaleruler('Units', 'mi', 'RulerStyle', 'patches',  ...
    'XLoc', xLoc, 'YLoc', yLoc);
title({titleText, 'with Scale Ruler'})
```

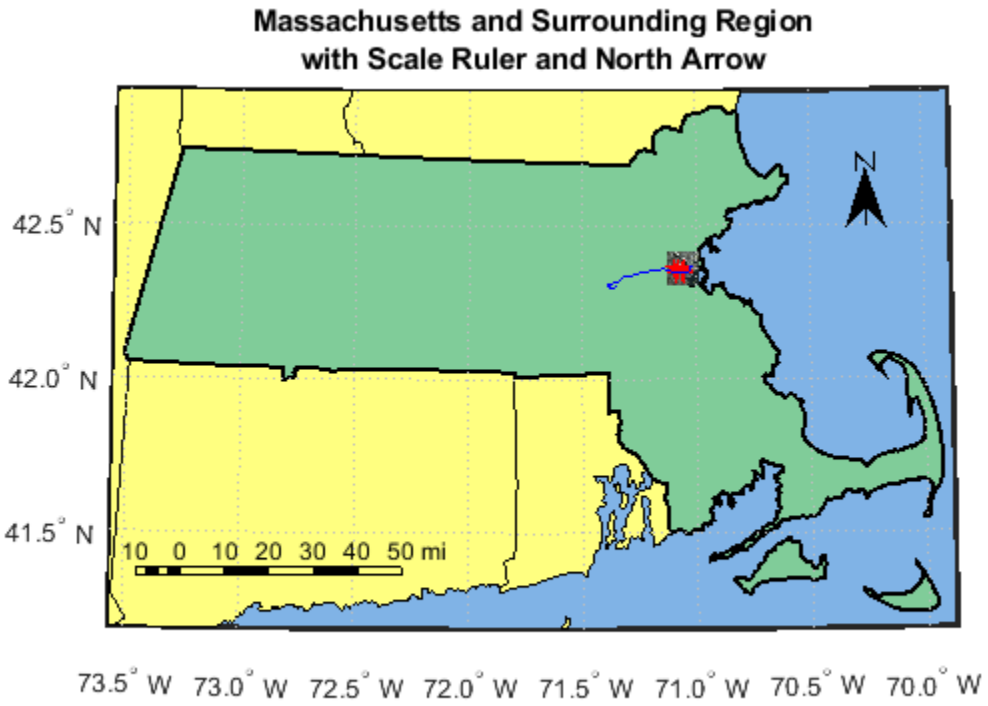


Example 10: Customize a Map Display with a North Arrow

Customize the map by adding a north arrow. A north arrow is a graphic element pointing to the geographic North Pole.

Use latitude and longitude values to position the north arrow.

```
northArrowLat =  42.5;
northArrowLon = -70.25;
northarrow('Latitude', northArrowLat, 'Longitude', northArrowLon);
title({titleText, 'with Scale Ruler and North Arrow'})
```

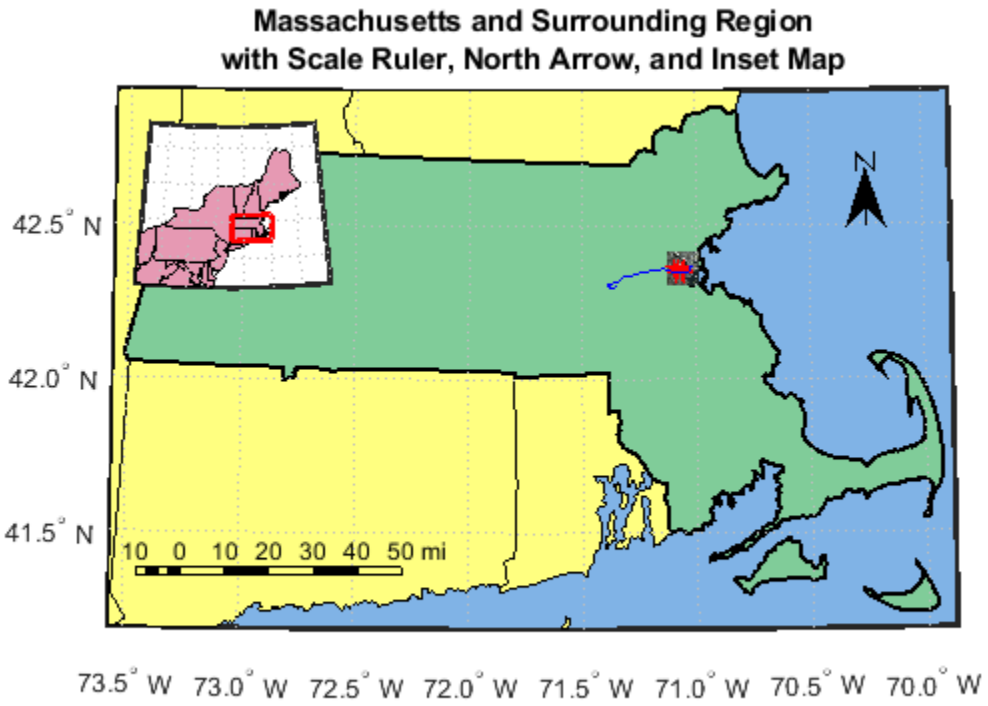


Example 11: Customize a Map Display with an Inset Map

Customize the map by adding an inset map. An inset map is a small map within a larger map that enables you to visualize the larger geographic region of your main map. Create a map for the surrounding region as an inset map. Use the `axes` function to contain and position the inset map. In the inset map:

- Display the state boundaries for the surrounding region
- Plot a red box to show the extent of the main map

```
h2 = axes('Position', [.15 .6 .2 .2], 'Visible', 'off');
usamap({'PA','ME'})
plabel off; mlabel off
setm(h2, 'FFaceColor', 'w');
geoshow(states, 'FaceColor', [.9 .6 .7], 'Parent', h2)
plotm(latlim([1 2 2 1 1]), lonlim([2 2 1 1 2]), ...
    'Color', 'red', 'LineWidth', 2)
title(ax, {titleText, 'with Scale Ruler, North Arrow, and Inset Map'})
```



Data Set Information

The file `boston_placenames.gpx` is from the Bureau of Geographic Information (MassGIS), Commonwealth of Massachusetts, Executive Office of Technology and Security Services. For more information about the data sets, use the command `type boston_placenames_gpx.txt`.

The file `boston_ovr.jpg` includes materials copyrighted by GeoEye, all rights reserved. GeoEye was merged into the DigitalGlobe corporation on January 29th, 2013. For more information about the data set, use the command `type boston_ovr.txt`.

See Also

`geoshow` | `gpxread` | `northarrow` | `scaleruler` | `setm` | `shaperead` | `usamap`

Creating Map Displays with Data in Projected Coordinate Reference System

This example illustrates how to import and display geographic data that contain coordinates in a projected coordinate reference system.

In particular, this example illustrates how to

- Import specific raster and vector data sets
- Create map displays for visualizing the data
- Display multiple data sets in a map display
- Display multiple data sets with coordinates in geographic and projected coordinate reference systems in a single map display

Example 1: Import Raster Data in Projected Coordinate Reference System

Geographic raster data that contains coordinates in a projected coordinate reference system can be stored in a variety of different formats, including standard file formats such as GeoTIFF, Spatial Data Transfer Standard (SDTS), NetCDF, HDF4, or HDF5. This example illustrates importing data from a GeoTIFF file. The data in the file contains coordinates in the projected map coordinate reference system *Massachusetts State Plane Mainland Zone coordinate system*.

The coordinates of the image in the GeoTIFF file, `boston.tif`, are in a projected coordinate reference system. You can determine that by using the `geotiffinfo` function and examine the `PCS` and `Projection` field values.

```
info = geotiffinfo('boston.tif');
disp(info.PCS)
```

```
NAD83 / Massachusetts Mainland
```

```
disp(info.Projection)
```

```
SPCS83 Massachusetts Mainland zone (meters)
```

The length unit of the coordinates are defined by the `UOMLength` field in the `info` structure.

```
disp(info.UOMLength)
```

```
US survey foot
```

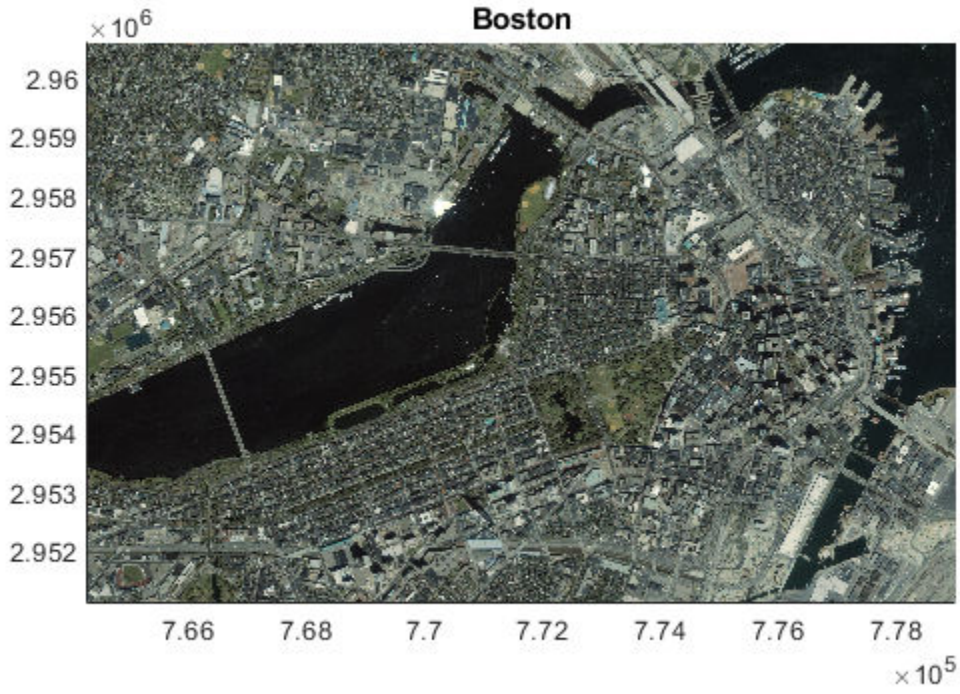
To import the image and the spatial referencing object, use `readgeoraster`.

```
[boston,R] = readgeoraster('boston.tif');
```

Example 2: Display Raster Data in Projected Coordinate Reference System

You can display the image on a regular MATLAB axes using `mapshow`, which displays the image and sets the axes limits to the limits defined by the referencing object, `R`. The coordinates, as mentioned above, are in `US survey foot` and are relative to an origin to the southwest of the map, which is why the numbers are large. The coordinates are always positive within the zone.

```
mapshow(boston,R)
axis image
title('Boston')
```



Example 3: Import Vector Data in Projected Coordinate Reference System

Geographic vector data that contains coordinates in a projected coordinate reference system can be stored in shapefiles. This example illustrates how to import vector data in a projected coordinate reference system from the shapefile, `boston_roads.shp`.

Import vector line data from the `boston_roads.shp` file.

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

To get information about the projected coordinate reference system, first return information about the shapefile as a structure. Then, query the `CoordinateReferenceSystem` field.

```
roadsInfo = shapeinfo('boston_roads.shp');
roadsInfo.CoordinateReferenceSystem
```

```
ans =
  projcrs with properties:

                    Name: "NAD83 / Massachusetts Mainland"
           GeographicCRS: [1×1 geocrs]
        ProjectionMethod: "Lambert Conic Conformal (2SP)"
              LengthUnit: "meter"
    ProjectionParameters: [1×1 map.crs.ProjectionParameters]
```

Example 4: Display Vector and Raster Data in Projected Coordinate Reference System

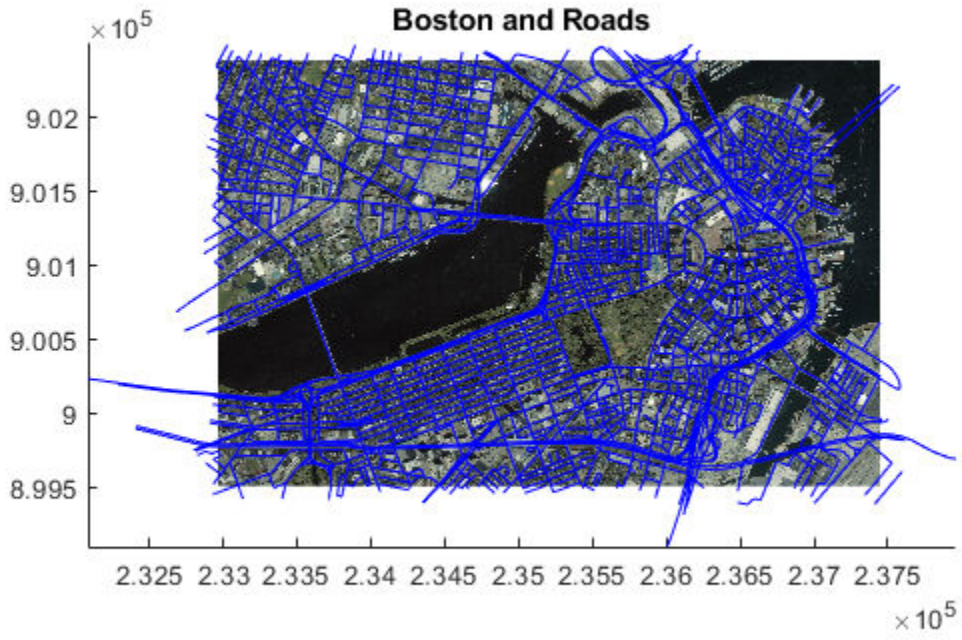
The vector and raster data in this example are in the same projected coordinate reference system. However, the vector data is in length units of meter, while the raster data is in length unit of survey foot. Convert the raster data to length units of meter and display the data on the same axes.

Convert the coordinates of the raster image from units of US survey foot to meter.

```
R.XWorldLimits = R.XWorldLimits * unitsratio('m','sf');
R.YWorldLimits = R.YWorldLimits * unitsratio('m','sf');
```

Display the raster image and vector data using `mapshow`.

```
figure
mapshow(boston,R)
mapshow(roads)
title('Boston and Roads')
```



Example 5: Display Data in both Geographic and Projected Coordinate Reference Systems

You may have geographic data whose coordinates are in latitude and longitude and other data whose coordinates are in a projected coordinate reference system. You can display these data sets in the same map display. This example illustrates how to display data in a geographic coordinate reference system (latitude and longitude) with data in a projected map coordinate reference system (Massachusetts State Plane Mainland Zone coordinate system).

Read a raster image with a worldfile whose coordinates are in latitude and longitude. Use `imread` to read the image and `worldfileread` to read the worldfile and construct a spatial referencing object.

```
filename = 'boston_ovr.jpg';
overview = imread(filename);
overviewR = worldfileread(getworldfilename(filename), 'geographic', size(overview));
```

To display the overview image and the GeoTIFF image in the same map display, you need to create a map display using a Mapping Toolbox™ projection structure containing the projection information for the data in the projected coordinate reference system, *Massachusetts State Plane Mainland Zone coordinate system*. To make a map display in this system, you can use the projection information contained in the GeoTIFF file. Use the `geotiff2mstruct` function to construct a Mapping Toolbox™ projection structure, from the contents of the GeoTIFF information structure. The `geotiff2mstruct` function returns a projection in units of meters. Use the projection structure to define the projection parameters for the map display.

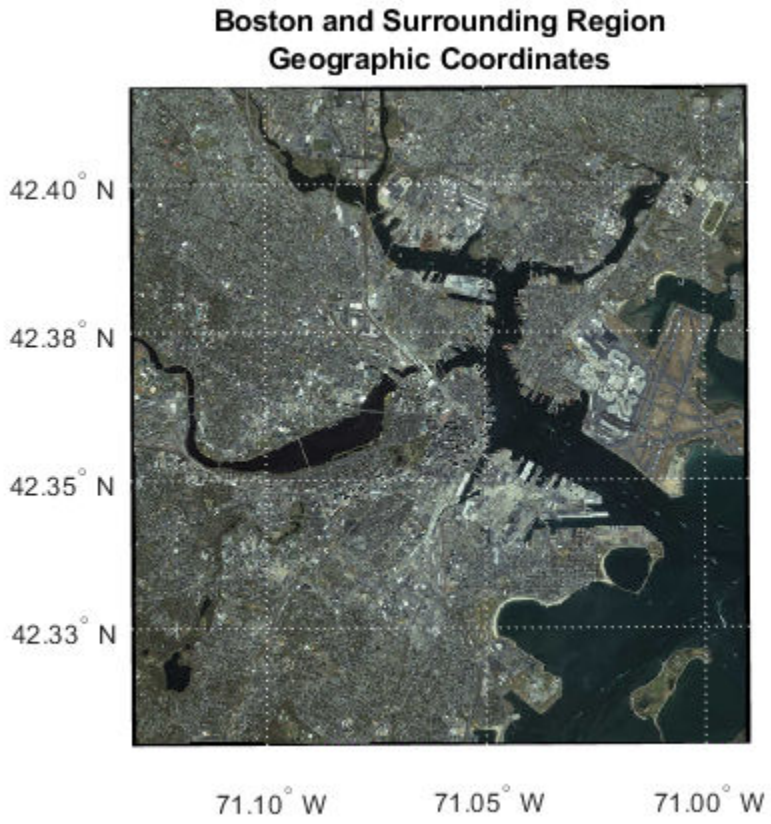
```
mstruct = geotiff2mstruct(info);
```

Use the latitude and longitude limits of the Boston overview image.

```
latlim = overviewR.LatitudeLimits;
lonlim = overviewR.LongitudeLimits;
```

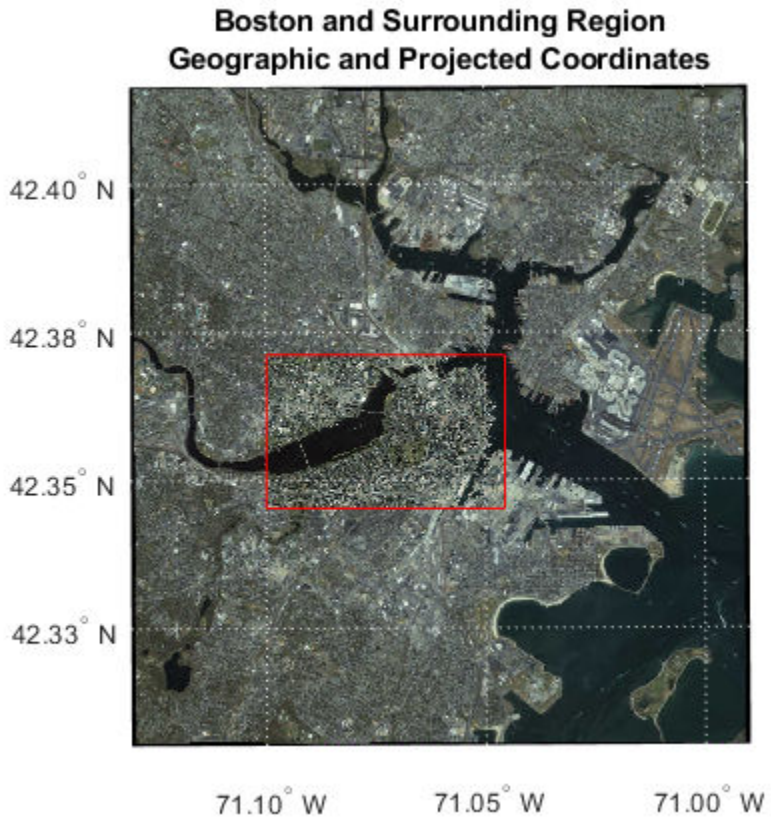
Create a map display by using the projection information stored in the map projection structure and set the map latitude and longitude limits. Display the geographic data in the map axes. `geoshow` projects the latitude and longitude coordinates.

```
figure('Renderer', 'opengl')
ax = axesm(mstruct, 'Grid', 'on',...
    'GColor', [.9 .9 .9], ...
    'MapLatlimit', latlim, 'MapLonLimit', lonlim, ...
    'ParallelLabel', 'on', 'PLabelLocation', .025, 'PlabelMeridian', 'west', ...
    'MeridianLabel', 'on', 'MlabelLocation', .05, 'MLabelParallel', 'south', ...
    'MLabelRound', -2, 'PLabelRound', -2, ...
    'PLineVisible', 'on', 'PLineLocation', .025, ...
    'MLineVisible', 'on', 'MlineLocation', .05);
geoshow(overview, overviewR)
axis off
tightmap
title({'Boston and Surrounding Region', 'Geographic Coordinates'})
```



Since the coordinates of the GeoTIFF image are in a projected coordinate reference system, use `mapshow` to overlay the more detailed Boston image onto the display. Plot the boundaries of the Boston image in red.

```
mapshow(boston, R)
plot(R.XWorldLimits([1 1 2 2 1]), R.YWorldLimits([1 2 2 1 1]), 'Color', 'red')
title({'Boston and Surrounding Region', 'Geographic and Projected Coordinates'})
```

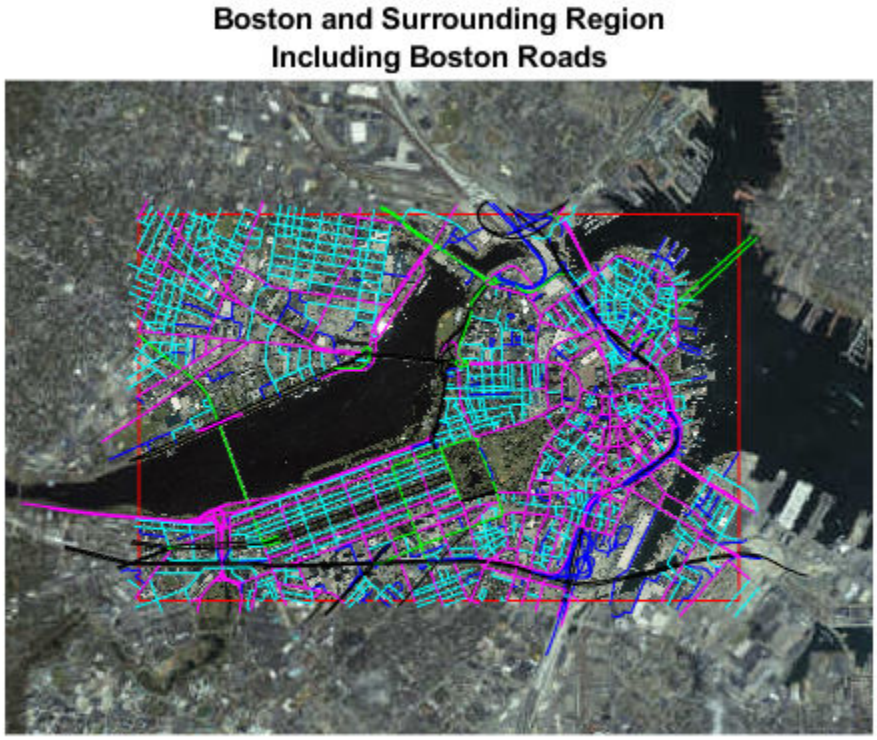


Zoom to the geographic region of the GeoTIFF image by setting the axes limits to the limits of the Boston image and add a small buffer. Note that the buffer size (`delta`) is expressed in meters.

```
delta = 1000;
xLimits = R.XWorldLimits + [-delta delta];
yLimits = R.YWorldLimits + [-delta delta];
xlim(ax,xLimits)
ylim(ax,yLimits)
setm(ax, 'Grid', 'off');
```

You can overlay the road vectors onto the map display. Use a symbol specification to give each class of road its own color.

```
roadColors = makesymbolspec('Line',...
    {'CLASS', 2, 'Color', 'k'}, ...
    {'CLASS', 3, 'Color', 'g'},...
    {'CLASS', 4, 'Color', 'magenta'}, ...
    {'CLASS', 5, 'Color', 'cyan'}, ...
    {'CLASS', 6, 'Color', 'b'},...
    {'Default',  'Color', 'k'});
mapshow(roads, 'SymbolSpec', roadColors)
title({'Boston and Surrounding Region','Including Boston Roads'})
```



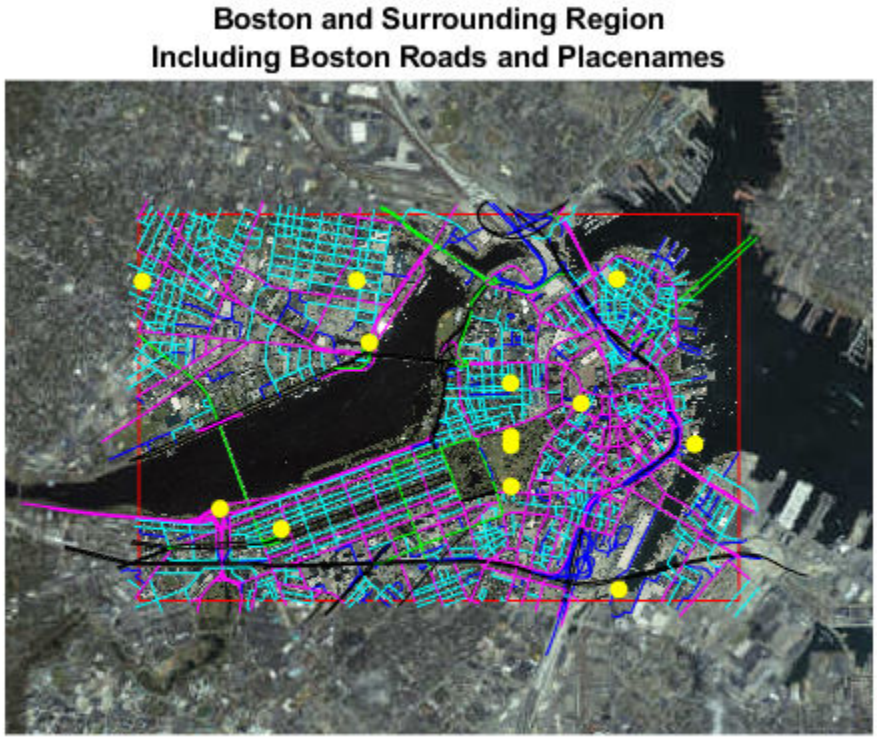
You can also overlay data from a GPS stored in a GPX file. Import point geographic vector data from the `boston_placenames.gpx` file included with the Mapping Toolbox™ software. The file contains latitude and longitude coordinates of geographic point features in part of Boston, Massachusetts, USA. Use `gpxread` to read the GPX file and return a `geopoint vector`.

```
placenames = gpxread('boston_placenames')
```

```
placenames =
 13×1 geopoint vector with properties:

 Collection properties:
       Geometry: 'point'
       Metadata: [1×1 struct]
 Feature properties:
       Latitude: [42.3501 42.3515 42.3598 42.3584 42.3529 42.3626 42.3668 42.3668 42.3668 42.355
      Longitude: [-71.0870 -71.0926 -71.0662 -71.0598 -71.0662 -71.0789 -71.0995 -71.0564 -71.080
           Name: {''BACK BAY''  'BACK BAY FENS'  'BEACON HILL'  'BOSTON'  'BOSTON NECK'  'BROAD C
    Description: {'PPL-SUBDVSN'  'MARSH'  'HILL'  'PPL'  'PENINSULA'  'CANAL'  'PPL'  'HILL'  'PP
```

Overlay the placenames onto the map and increase the marker size, change the markers to circles and set their edge and face colors to yellow.

```
geoshow(placenames, 'Marker','o', 'MarkerSize', 6, ...
    'MarkerEdgeColor', 'y', 'MarkerFaceColor','y')
title({'Boston and Surrounding Region','Including Boston Roads and Placenames'})
```



Data Set Information

The files `boston.tif` and `boston_ovr.jpg` include materials copyrighted by GeoEye, all rights reserved. GeoEye was merged into the DigitalGlobe corporation on January 29th, 2013. For more information about the data sets, use the commands `type boston.txt` and `type boston_ovr.txt`.

The files `boston_roads.shp` and `boston_placenames.gpx` are from the Bureau of Geographic Information (MassGIS), Commonwealth of Massachusetts, Executive Office of Technology and Security Services. For more information about the data sets, use the commands `type boston_roads.txt` and `type boston_placenames_gpx.txt`.

Pick Locations Interactively

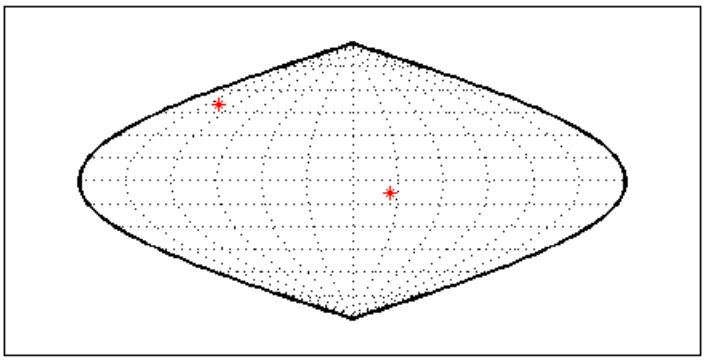
You can use Mapping Toolbox functions and GUIs to interact with maps, both in `mapview` and in figures created with `axesm`. This section describes two useful graphic input functions, `inputm` and `gcpmap`. The `inputm` function (analogous to the MATLAB `ginput` function) allows you to get the latitude-longitude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

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

```
axesm sinusoid
framem on; gridm on
points=inputm(3)
```

```
points =
  -41.7177 -145.0293
    7.9211   -0.5332
   38.5492  149.2237
```

```
plotm(points,'r*')
```



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

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

```
[areascale,angledef] = distortcalc(points(1,1),points(1,2))
```

```
areascale =
    1.0000
angledef =
    85.9284
```

```
[areascale,angledef] = distortcalc(points(2,1),points(2,2))
```

```
areascale =
    1.0000
angledef =
    3.1143
```

```
[areascale,angledef] = distortcalc(points(3,1),points(3,2))
```

```
areascale =
    1.0000
angledef =
    76.0623
```

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

See Also

`gcpmap` | `inputm`

Related Examples

- “Create an Interactive Map for Selecting Point Features” on page 4-127

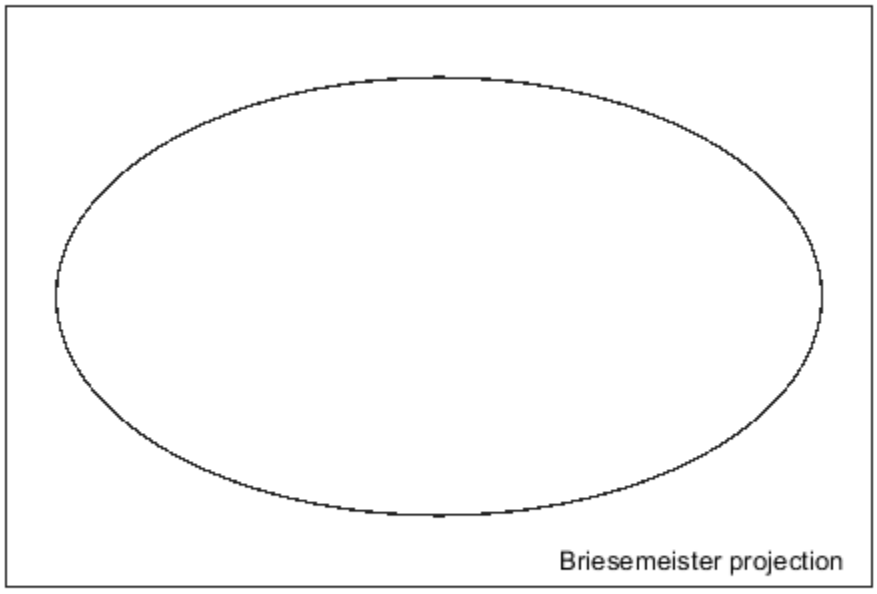
Create an Interactive Map for Selecting Point Features

This example shows how to construct a map of major world cities enhanced with coastlines and terrain. It uses the modified azimuthal Briesemeister map projection. The example includes some optional code that allows a user to interactively pick a location and get the name and location of the nearest city. To see this part of the example, you must run the complete example, pop-out the last illustration into a separate MATLAB figure, and then run the optional code at the MATLAB command line.

Step 1: Set up a Map Axes Object and Render a Global Elevation Grid

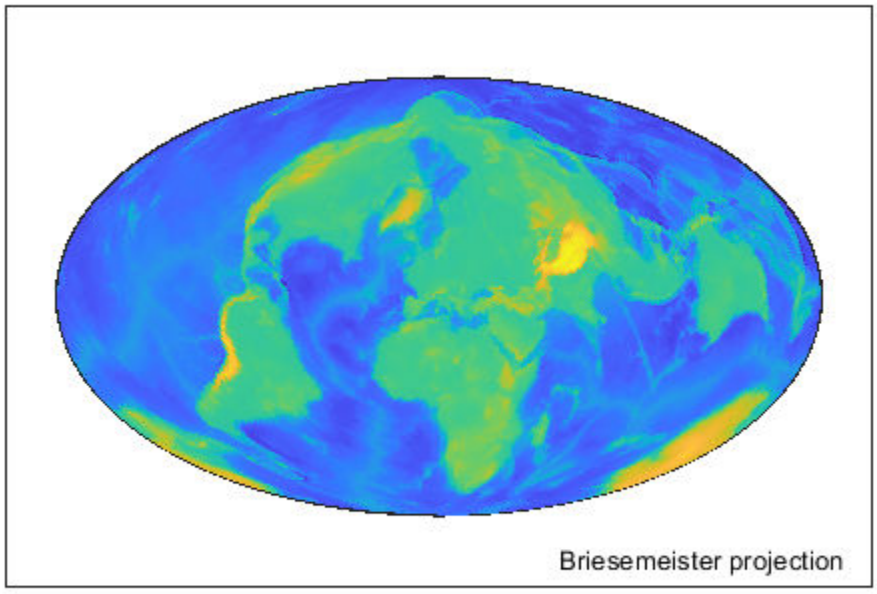
Create a map axes object.

```
figure
axesm bries
text(2.8,-1.8,'Briesemeister projection','HorizontalAlignment','right')
framem('FLineWidth',1)
```



Load elevation raster data and a geographic cells reference object. Display the data on the map.

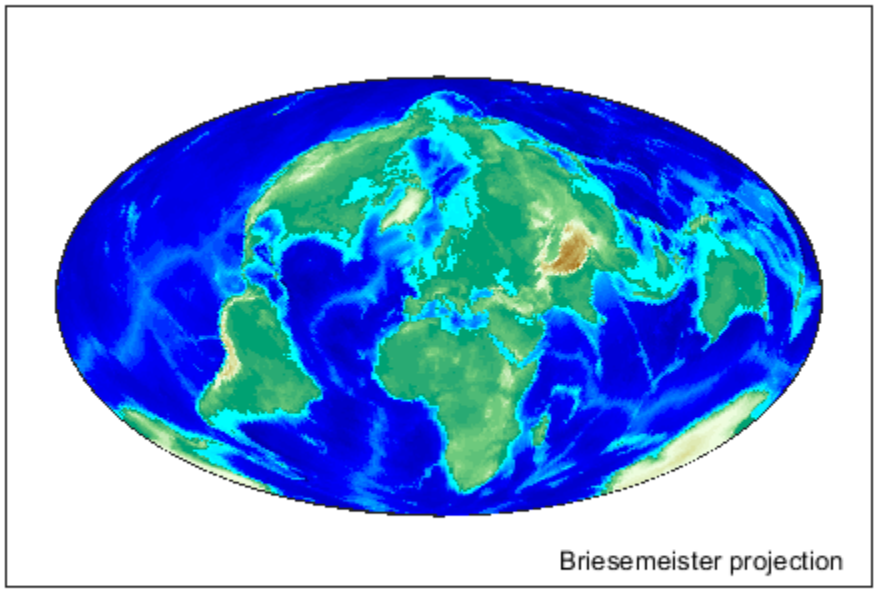
```
load topo60c
geoshow(topo60c,topo60cR,'DisplayType','texturemap')
```



Step 2: Improve the Terrain Display

Apply a colormap appropriate for elevation data. Make the display brighter.

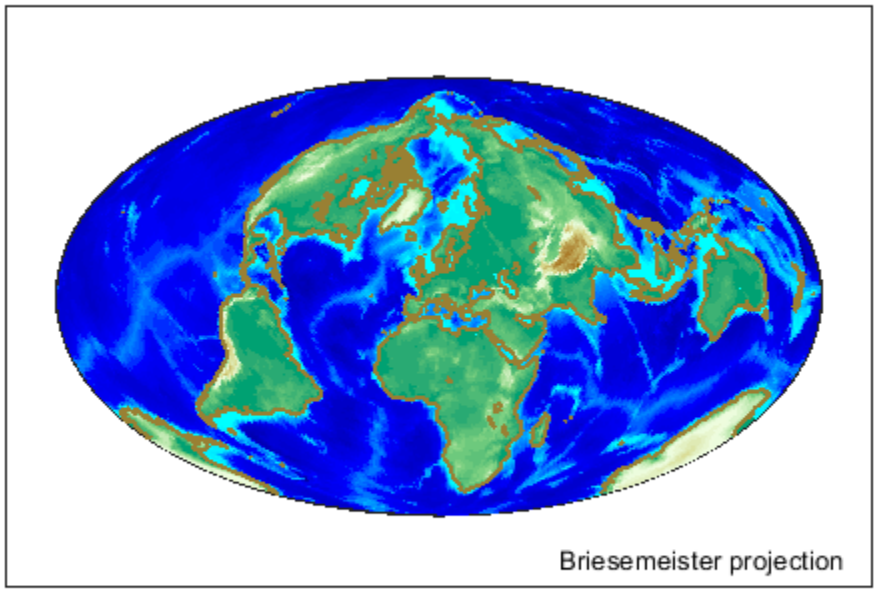
```
demcmap(topo60c)
brighten(0.5)
```



Step 3: Add Simplified Coastlines

Load global coastline coordinates. Generalize the coastlines to 0.25-degree tolerance. Then, plot the coastlines in brown.

```
load coastlines
[rlat,rlon] = reducem(coastlat,coastlon,0.25);
geoshow(rlat,rlon,'Color',[.6 .5 .2],'LineWidth',1.5)
```



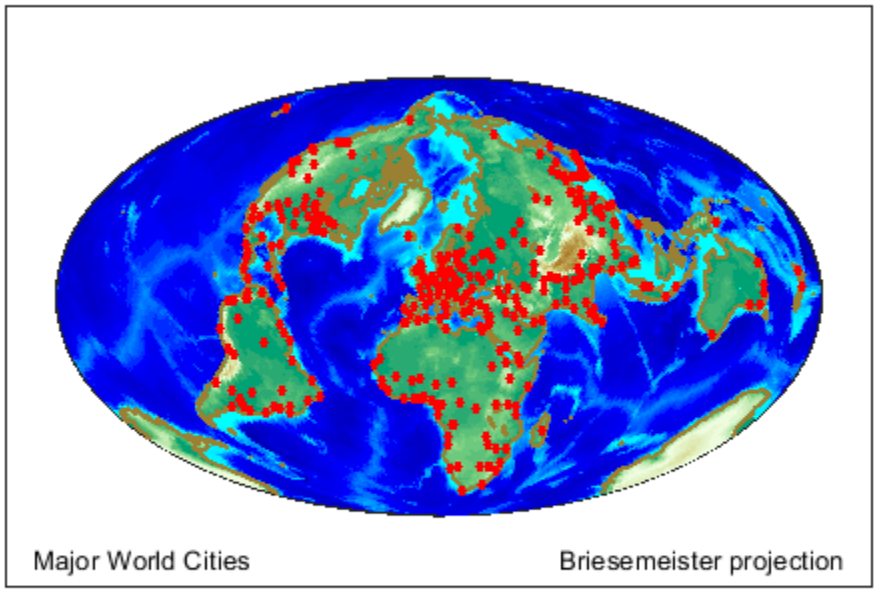
Step 4: Plot City Locations with Red Point Markers

Read a shapefile containing names of cities worldwide and their coordinates in latitude and longitude.

```
cities = shaperead('worldcities','UseGeoCoords',true);
```

Extract the point latitudes and longitudes with `extractfield`, and add them to the map.

```
lats = extractfield(cities,'Lat');
lons = extractfield(cities,'Lon');
geoshow(lats, lons,...
        'DisplayType', 'point',...
        'Marker', 'o',...
        'MarkerEdgeColor', 'r',...
        'MarkerFaceColor', 'r',...
        'MarkerSize', 3)
text(-2.8,-1.8,'Major World Cities')
```



Step 5: Select Cities Interactively (Optional)

Now, using the map you've created, you can set up a simple loop to prompt for clicks on the map and display the name and coordinates of the nearest city. You must pop the last map you created in Step 4 into a separate MATLAB figure window, using the button that appears at the top of the map. Also, in the following code, set `runCitySelectionLoop` to `true`, and execute the code at the command line.

The code first displays text instructions at the upper left of the map. Then, it enters a loop in which it captures selected latitudes and longitudes with `inputm`. Use `distance` to calculate the great circle distance between each selected point and every city in the database. Determine index of the closest city, change the appearance of its marker symbol, and display the city's name and latitude/longitude coordinates.

```
runCitySelectionLoop = false; % Set to true to run optional city selection loop

if(runCitySelectionLoop)
    h1 = text(-2.8, 1.9, 'Click on a dot for its city name. Press ENTER to stop');
    h2 = text(-2.8, 1.7, '');
    h3 = text(-2.8, 1.5, 'City Coordinates.');
    while true
        [selected_lat,selected_lon] = inputm(1);
        if isempty(selected_lat)
            break % User typed ENTER
        end
        d = distance(lats, lons, selected_lat, selected_lon);
        k = find(d == min(d(:)),1);
        city = cities(k);
```

```
        geoshow(city.Lat, city.Lon, ...
                'DisplayType', 'point', ...
                'Marker', 'o', ...
                'MarkerEdgeColor', 'k', ...
                'MarkerFaceColor', 'y', ...
                'MarkerSize', 3)
       h2.String = city.Name;
       h3.String = num2str([city.Lat, city.Lon],'%10.2f');
    end
    disp('End of input.')
end
```

See Also

demcmap | geoshow | inputm | shaperead

Create Small Circle and Track Annotations on Maps Interactively

You can generate geographic line annotations, such as navigational tracks and small circles, interactively. Great circle tracks are the shortest distance between points that, 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** the tracks 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).

Set up an orthographic view centered over the Pacific Ocean. Use the `coastlines` MAT-file.

```
axesm('ortho','origin',[30 180])
framem;
gridm
load coastlines
plotm(coastlat,coastlon,'k')
```

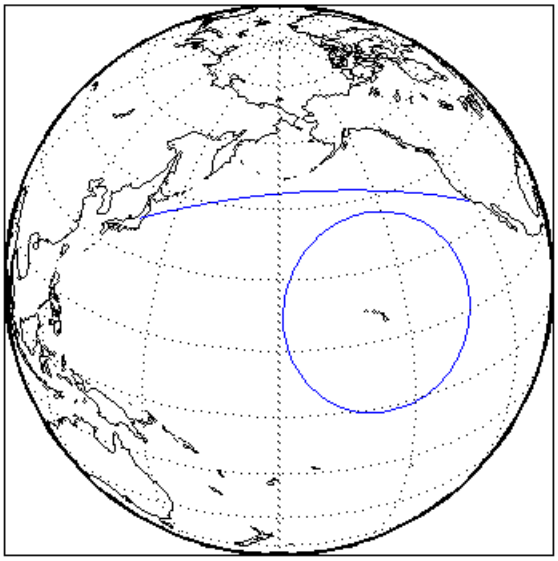
Create a track with the `trackg` function, which prompts for two endpoints. The default track type is a great circle. Create a great circle track from Los Angeles, California, to Tokyo, Japan, and a 1000 km radius small circle centered on the Hawaiian Islands.

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

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.



To adjust the size of the small circle to be 2000 km, **Shift+click** anywhere on its perimeter. The Small Circles dialog box appears.

Type `2000` into the **Radius** field.

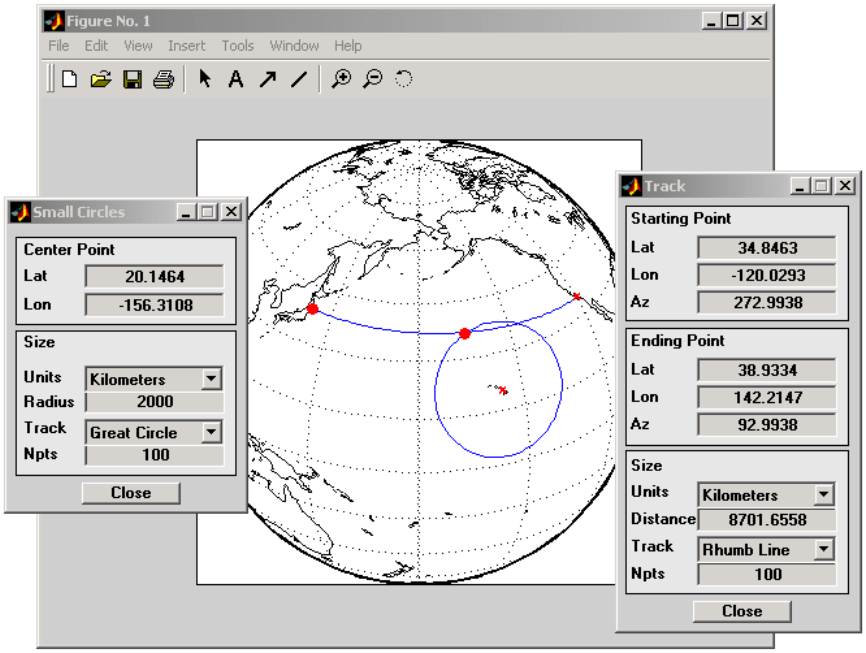
Click **Close**. The small circle adjusts to be 2000 km around Hawaii.

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.

Change the track type from `Great Circle` to `Rhumb Line` with the Track pop-up menu. The track immediately changes shape.

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.

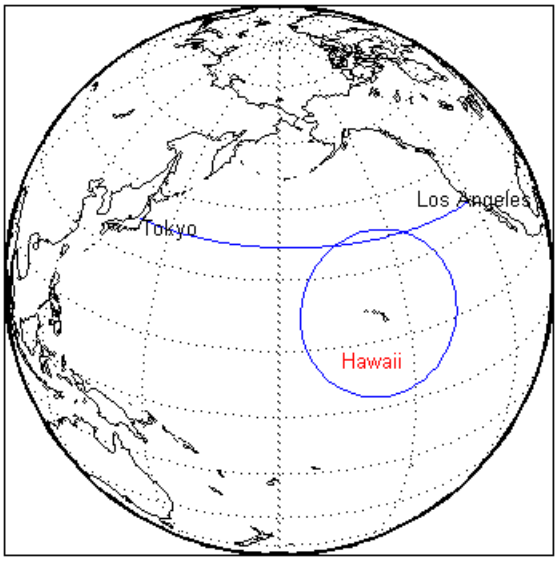
The following figure shows the Small Circles and Track dialog boxes.



Interactively Display Text Annotations on a Map

Interactively place text annotations on a map using the `gtextm` function. Call the function by specifying text and optional properties as arguments. Then, choose a location for the text by clicking on the map.

```
gtextm('Hawaii','color','r')
gtextm('Tokyo')
gtextm('Los Angeles')
```



After you place text, you can move it interactively using the selection tool in the map figure window.

To display text on a map by specifying numerical arguments, use the `textm` function. For more information, see “Use Geographic and Nongeographic Objects in Map Axes” on page 4-72.

Work with Objects by Name

You can manipulate displayed map objects by name. Many Mapping Toolbox functions assign descriptive names to the `Tag` property of the objects they create. The `namem` and related functions allow you to control the display of groups of similarly named objects, determine the names and change them if desired, and use the name in the `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 value to the `Tag` property in mapping display functions that use property name/property value pairs. If the `Tag` does not contain a value, the name defaults to an object's `Type` property, such as `'line'` or `'text'`.

Manipulate Displayed Map Objects By Name

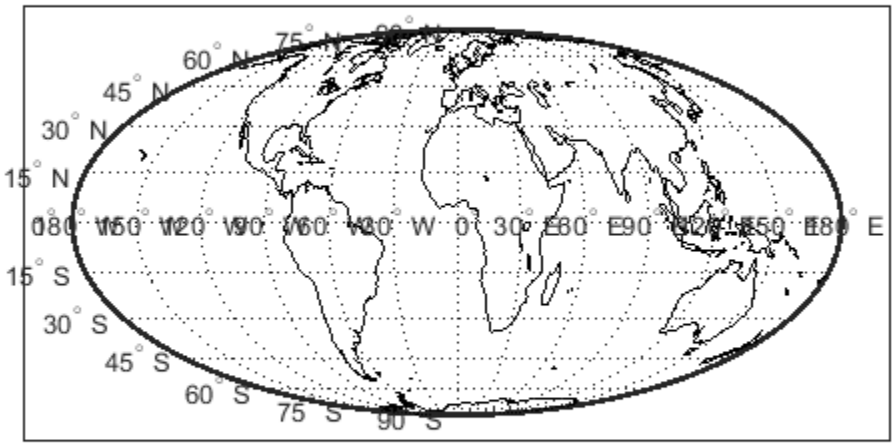
This example shows how to manipulate displayed map objects by name. Many functions assign descriptive names to the `Tag` property of the objects they create. The `namem` and related functions allow you to control the display of groups of similarly named objects, determine the names and change them, if desired, and use the name in calls to `get` and `set` .

Display a vector map of the world.

```
f = axesm('fournier')
```

```
f =
  Axes with properties:

            XLim: [0 1]
            YLim: [0 1]
          XScale: 'linear'
          YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.1100 0.7750 0.8150]
            Units: 'normalized'

  Show all properties
```

```
framem on;
gridm on;
plabel on;
mlabel('MLabelParallel',0)
load coastlines
plotm(coastlat,coastlon,'k','Tag','Coastline')
```



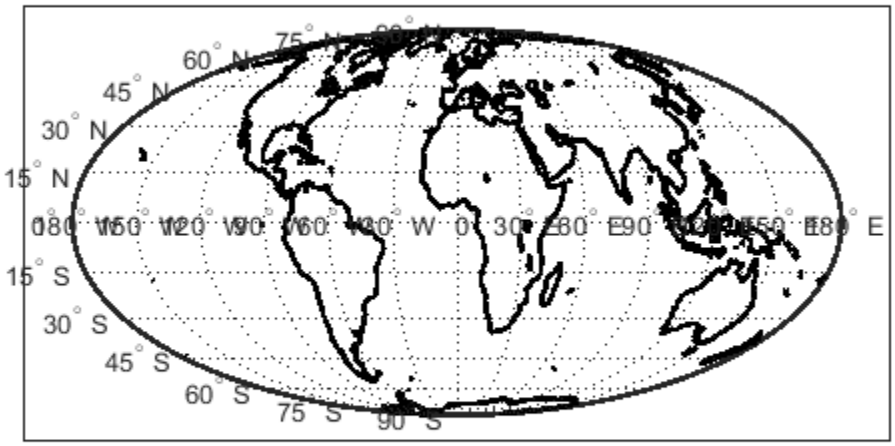
List the names of the objects in the current axes using `namem` .

```
namem
```

```
ans = 6x9 char array
    'PLabel   '
    'MLabel   '
    'Parallel '
    'Meridian '
    'Coastline'
    'Frame    '
```

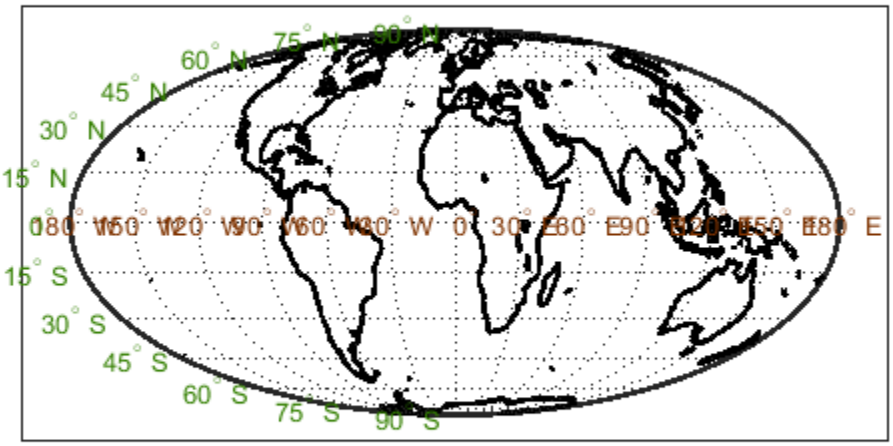
Use `handlem` to get handles to graphic objects in the map. You use these handles to get or set object properties. For example, to change the line width of the coastline with `set` . If you call `handlem` with no arguments, it opens a graphical user interface that lists all the map axes objects. You can select objects interactively.

```
set(handlem('Coastline'),'LineWidth',2)
```



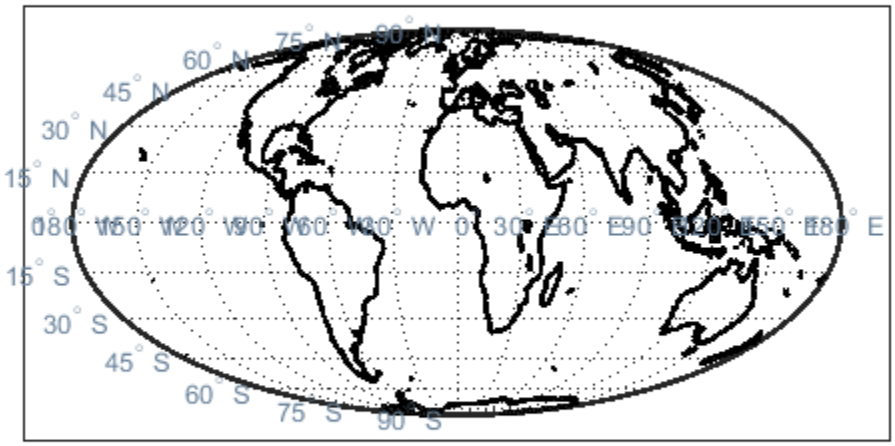
Change the colors of the meridian and parallel labels separately.

```
set(handlem('Mlabel'),'Color',[.5 .2 0])
set(handlem('Plabel'),'Color',[.2 .5 0])
```



Change the color of the labels to be the same.

```
setm(f,'fontcolor', [.4 .5 .6])
```



Use `handlem` specifying the `'all'` modifier to get a list of all text objects or all line objects.

```
t = handlem('alltext')
```

```
t =
  26x1 Text array:

  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (PLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
```

```
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
  Text    (MLabel)
```

```
l = handlem('allline')
```

```
l =
  3x1 Line array:

  Line    (Parallel)
  Line    (Meridian)
  Line    (Coastline)
```

Making Three-Dimensional Maps

- “Sources of Terrain Data” on page 5-2
- “Determine and Visualize Visibility Across Terrain” on page 5-3
- “Light a Terrain Map of a Region” on page 5-5
- “Surface Relief Shading” on page 5-8
- “Colored Surface Shaded Relief” on page 5-13
- “Relief Mapping with Light Objects” on page 5-17
- “Drape Data on Elevation Maps” on page 5-24
- “Drape Geoid Heights Over Topography” on page 5-25
- “Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid” on page 5-30
- “Drape Geolocated Grid on Regular Data Grid via Texture Mapping” on page 5-36
- “The Globe Display Compared with the Orthographic Projection” on page 5-39
- “Use Opacity and Transparency in Globe Displays” on page 5-45
- “Access Basemaps and Terrain for Geographic Globe” on page 5-48
- “Create Interactive Basemap Picker” on page 5-50
- “Visualize Aircraft Line-of-Sight Over Terrain” on page 5-52
- “Visualize UAV Flight Path on 2-D and 3-D Maps” on page 5-61

Sources of Terrain Data

Nearly all published terrain elevation data is in the form of data grids. “Types of Data Grids and Raster Display Functions” on page 4-98 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.

Digital Terrain Elevation Data from NGA

The Digital Terrain Elevation Data (DTED) Model is a series of gridded elevation models with global coverage at resolutions of 1 kilometer or finer. DTED files 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 DTED files, 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 file names with the extension `dtN`, where `N` is the level of the DTED product. You can find published specifications for DTED at the NGA website.

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 website: “Find Geospatial Data Online” on page 2-70.

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 DTED files. 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.

Determine and Visualize Visibility Across Terrain

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

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

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

Compute Line of Sight

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

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

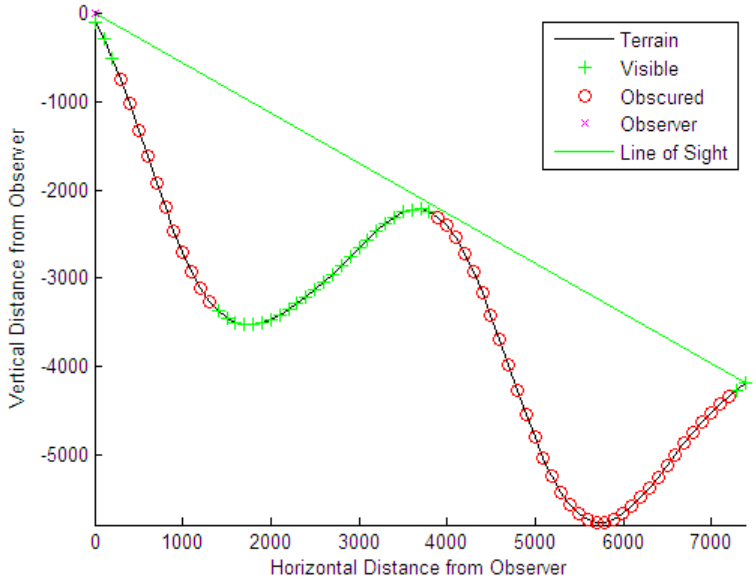
```
map = 500*peaks(100);
maplegend = [ 1000 0 0];
```

2 Define two locations on this grid to test intervisibility:

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

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

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

     1
```



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

You can also compute the *viewshed*, a name derived from *watershed*, which indicates the elements of a terrain elevation grid that are visible from a particular location. The `viewshed` function checks for a line of sight between a fixed observer and each element in the grid. See the `viewshed` function reference page for an example.

Light a Terrain Map of a Region

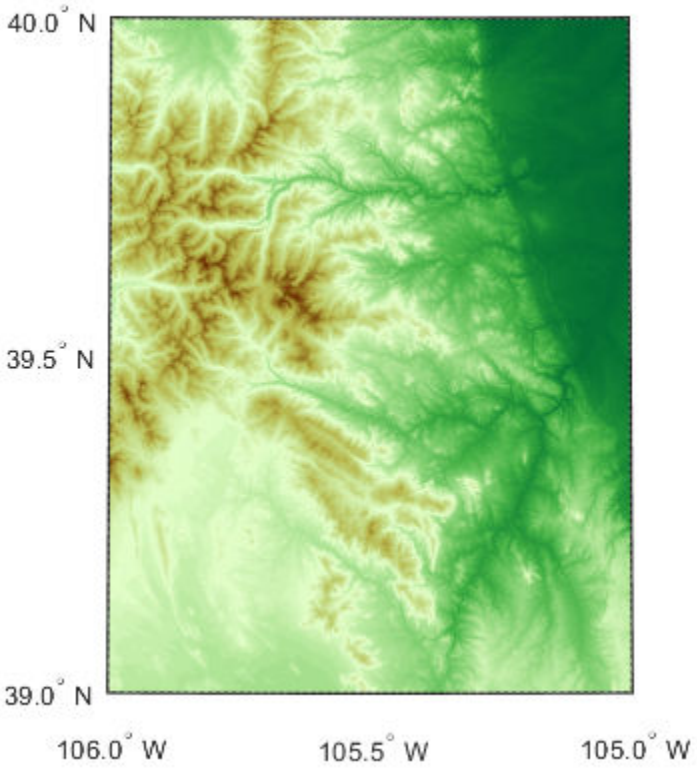
Light a terrain map of a region around South Boulder Peak in Colorado.

First, import elevation data and a geographic postings reference object. To plot the data using `geoshow`, the raster data must be of type `double` or `single`. Therefore, specify the data type for the raster using the `'OutputType'` name-value pair.

```
[Z,R] = readgeoraster('n39_w106_3arc_v2.dt1','OutputType','double');
```

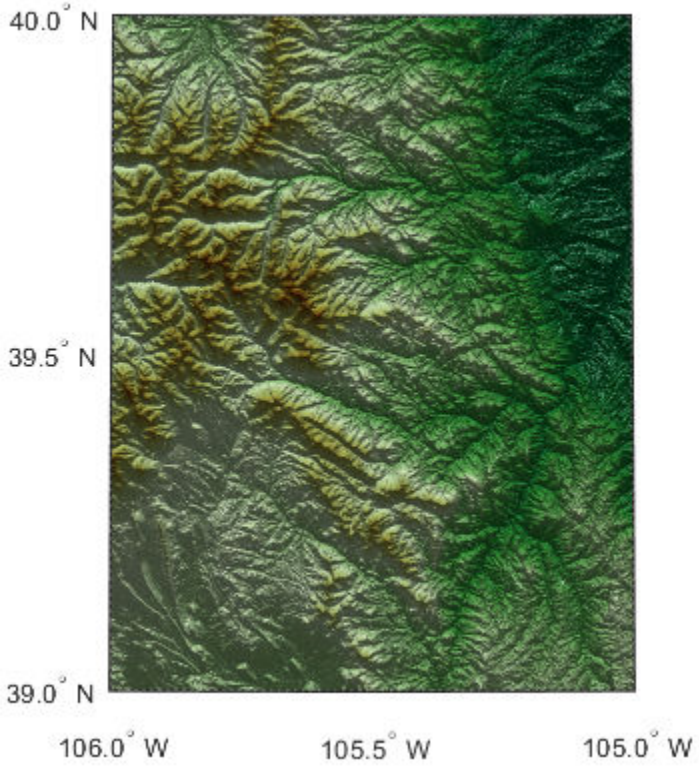
Then, display the data as a surface. Apply a colormap appropriate for terrain data using the `demcmap` function.

```
usamap(R.LatitudeLimits,R.LongitudeLimits);
geoshow(Z,R,'DisplayType','surface')
demcmap(Z)
```



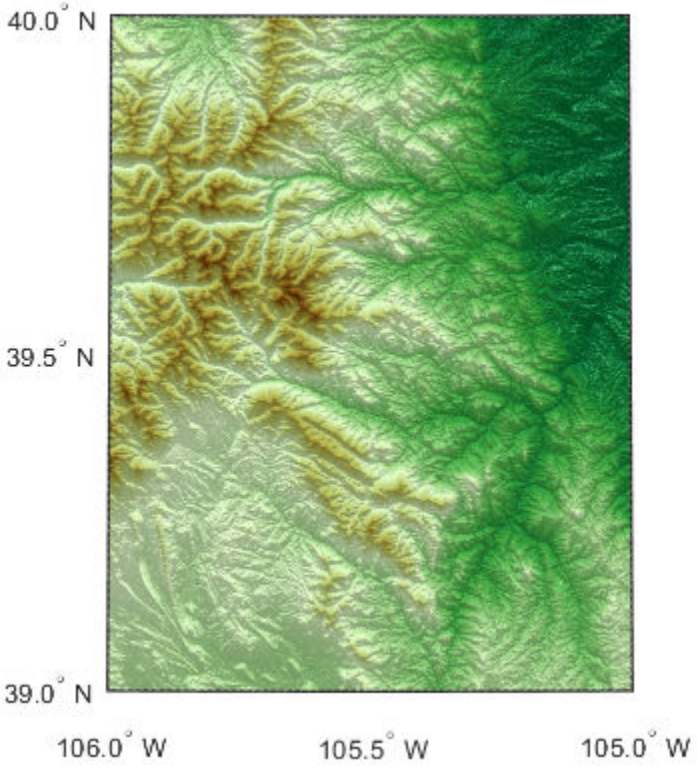
Set the vertical exaggeration using the `daspectm` function. Then, specify a light source in the top left corner of the map. Find the coordinates of the top left corner by querying the `LatitudeLimits` and `LongitudeLimits` properties of the reference object.

```
daspectm('m',20)
cornerlat = R.LatitudeLimits(2);
cornerlon = R.LongitudeLimits(1);
lightm(cornerlat,cornerlon)
```



Restore the luminance of the map by specifying the ambient, diffuse, and specular light strength.

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



The DTED file used in this example is courtesy of the US Geological Survey.

See Also

`daspectm` | `lightm`

More About

- “Lighting Overview”

Surface Relief Shading

You can make dimensional monochrome shaded-relief maps with the function `surflm`, which is analogous to the MATLAB `surfl` function. The effect of `surflm` is similar to using lights, but the function models illumination itself (with one “light source” that you specify when you invoke it, but cannot reposition) by weighting surface normals rather than using light objects.

Shaded relief maps of this type are usually portrayed two-dimensionally rather than as perspective displays. The `surflm` function works with any projection except `globe`.

The `surflm` function accepts geolocated data grids only. Recall, however, that regular data grids are a subset of geolocated data grids, to which they can be converted using `meshgrat` (see “Fit Gridded Data to the Graticule” on page 4-99). The following example illustrates this procedure.

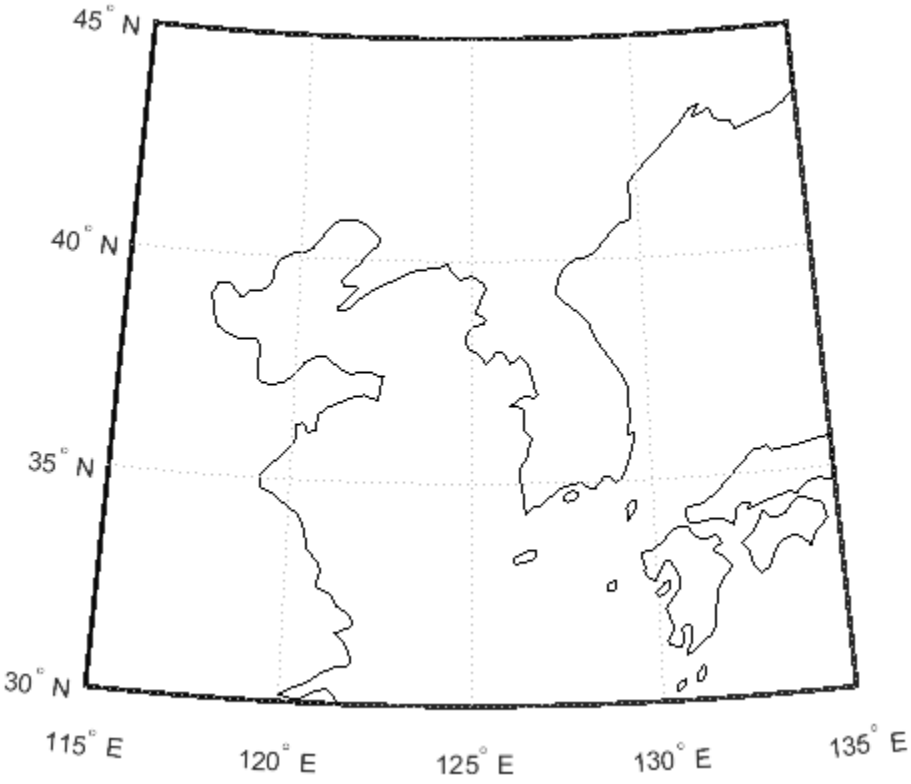
Create Monochrome Shaded Relief Map

Simulate a single light source in a figure using `surflm`. First, load elevation data and a geographic cells reference object for the Korean peninsula. Import coastline vector data using `shaperead`. Create a map with appropriate latitude and longitude limits using `worldmap`.

```
load korea5c
latlim = korea5cR.LatitudeLimits;
lonlim = korea5cR.LongitudeLimits;
coastline = shaperead('landareas',...
    'UseGeoCoords', true,...
    'BoundingBox', [lonlim' latlim']);

worldmap(latlim,lonlim)
```

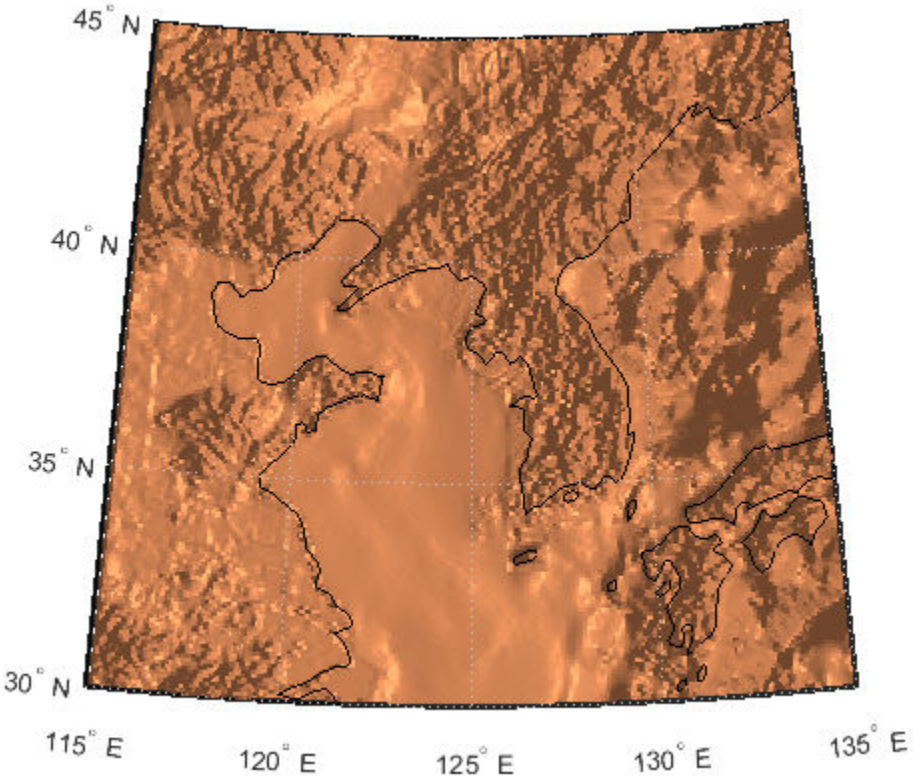
Display the coastline data using `geoshow`.

```
geoshow(coastline,'FaceColor','none')
```



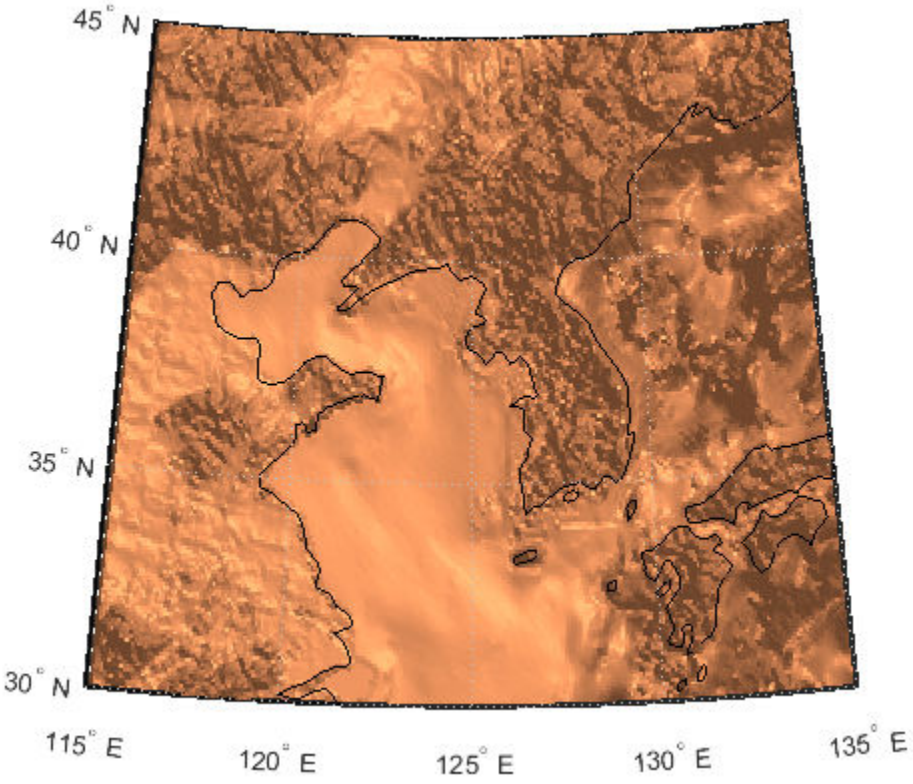
Transform the regular data grid to a geolocated data grid using `meshgrat`. Then, generate a shaded relief map using `surflm`. By default, the lighting direction is 45º counterclockwise from the viewing direction. Therefore, the light source is in the southeast. Change the colormap to the monochromatic colormap `'copper'`.

```
[klat,klon] = meshgrat(korea5c,korea5cR);
s = surflm(klat,klon,korea5c);
colormap('copper')
```



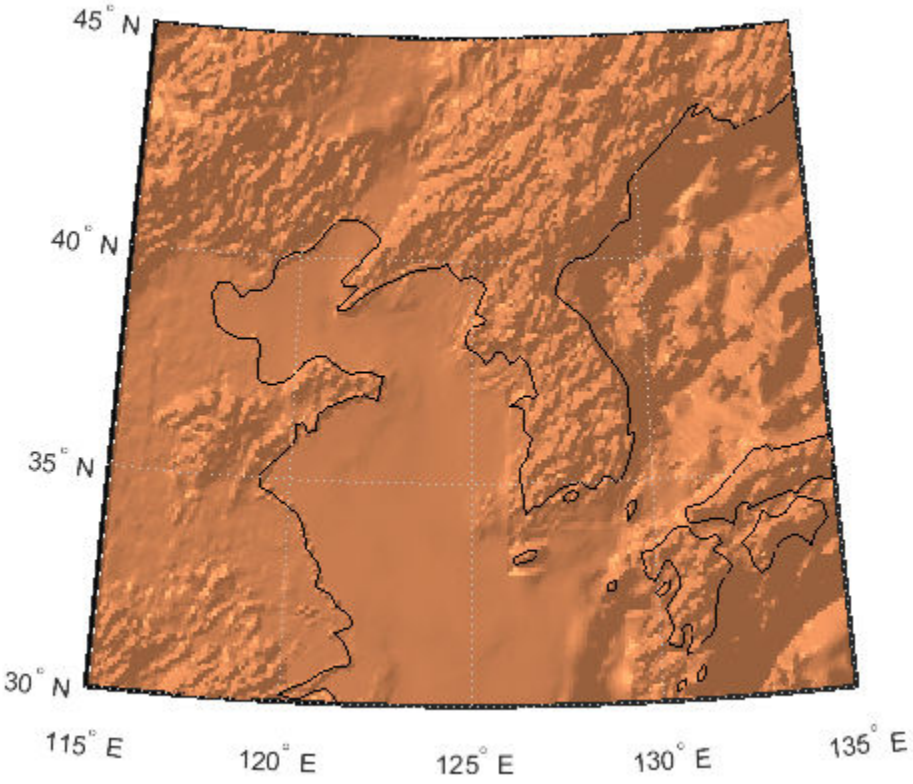
Clear the map. Then, display the shaded relief map with a different light source by specifying the azimuth as 135º and the elevation as 60º. The surface lightens and has a new character because it is lit closer to overhead and from a different direction.

```
clmo(s)
s = surflm(klat,klon,korea5c,[135 60]);
```



Shift the light to the northwest by specifying the azimuth as -135º. Lower the light to 40º above the horizon. A lower light source decreases the overall reflectance of the surface when viewed from above. Therefore, specify a 1-by-4 vector of reflectance constants that describe the relative contributions of ambient light, diffuse reflection, specular reflection, and the specular shine coefficient.

```
clmo(s);
ht = surflm(klat,klon,korea5c,[-135 30],[0.65 0.4 0.3 10]);
```



The mountain ridges that run from northeast to southwest are approximately perpendicular to the light source. Therefore, these parameters demonstrate appropriate lighting for the terrain.

The monochromatic coloration in this example does not differentiate land from water. For an example that differentiates land from water, see “Colored Surface Shaded Relief” on page 5-13.

Colored Surface Shaded Relief

The functions `meshlsrm` and `surflsrm` display maps as shaded relief with surface coloring as well as light source shading. You can think of them as extensions to `surflm` that combine surface coloring and surface light shading. Use `meshlsrm` to display regular data grids and `surflsrm` to render geolocated data grids.

These two functions construct a new colormap and associated `CData` matrix that uses grayscales to lighten or darken a matrix component based on its calculated surface normal to a light source. While there are no analogous MATLAB display functions that work like this, you can obtain similar results using MATLAB light objects, as “Relief Mapping with Light Objects” on page 5-17 explains.

For further information, see the reference pages for `surflsrm`, `meshlsrm`, `daspectm`, and `view`.

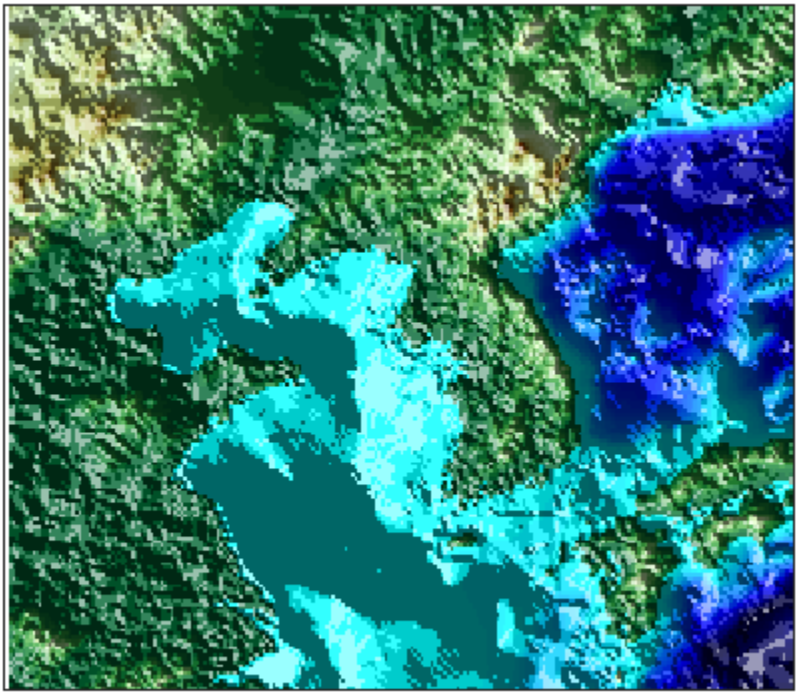
Create Colored Shaded Relief Map

Display surface illumination over colored elevation data using `surflsrm`. First, load elevation data and a geographic cells reference object for the Korean peninsula. Georeference the regular data grid using `meshgrat`.

```
load korea5c
[klat,klon] = meshgrat(korea5c,korea5cR);
```

Create a colormap appropriate for elevation data. Plot the colored shaded relief map by specifying a light source with an azimuth of -130º and an altitude of 50º. The `surflsrm` function transforms the colormap to shade relief according to the light source. Eliminate white space around the map using `tightmap`.

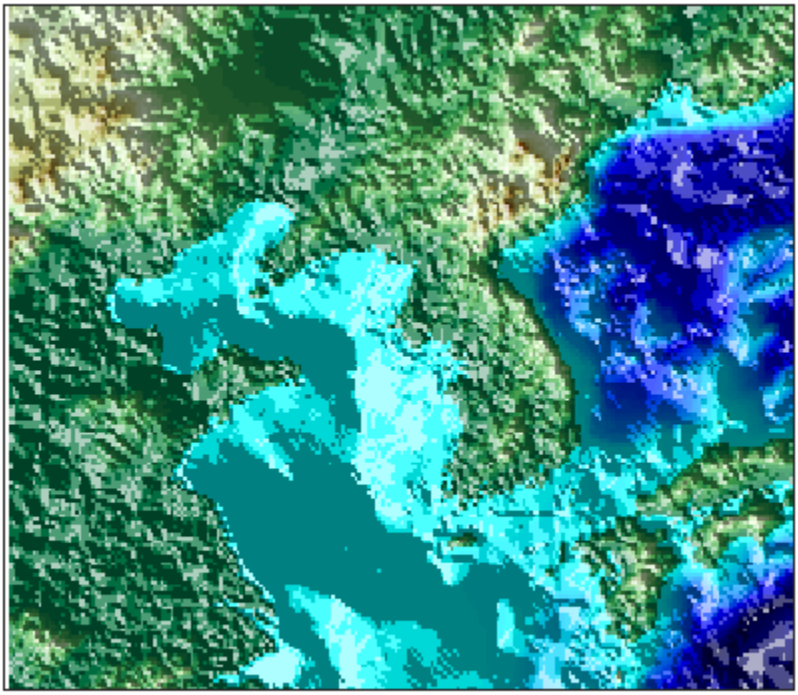
```
[cmap,clim] = demcmap(korea5c);
axesm('miller','MapLatLimit',[30 45],'MapLonLimit',[115 135])
surflsrm(klat,klon,korea5c,[-130 50],cmap,clim)
tightmap
```



You can achieve the same effect using `meshlsrm`, which operates on regular data grids.

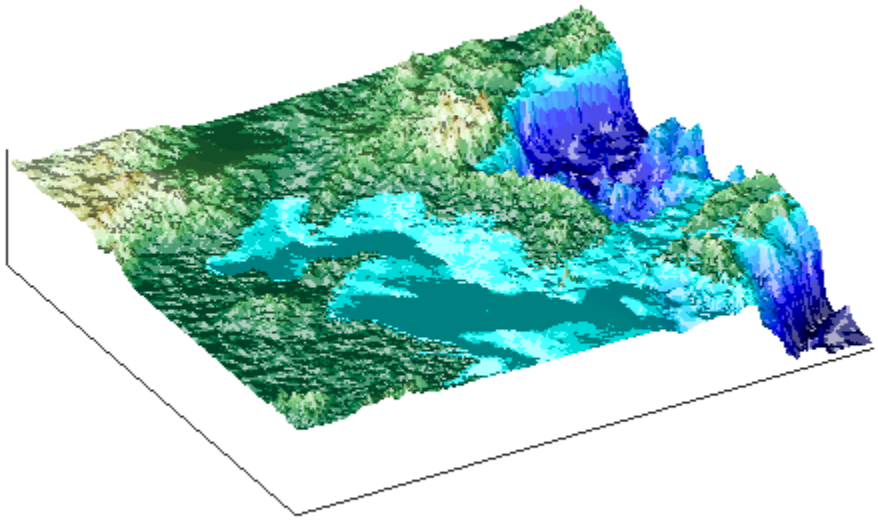
The surface has more contrast than if it were not shaded. Lighten the surface uniformly by 25%.

```
brighten(0.25)
```



Display an oblique view of the surface. Hide the bounding box by setting the `Box` property, exaggerate terrain relief by a factor of 50 using `daspectm`, and set the view to an azimuth of -30º and an altitude of 30º.

```
set(gca,'Box','off')
daspectm('meters',50)
view(-30,30)
```



You can continue rotating the perspective using `view` or the **Rotate 3D** tool in the figure window. You can continue changing the vertical exaggeration using `daspectm`. To change the built-in lighting direction, you must generate a new view using `surflsrm`.

Relief Mapping with Light Objects

This example shows how to create a light object to mimic the map produced in “Colored Surface Shaded Relief” on page 5-13, which uses shaded relief computations rather than light objects.

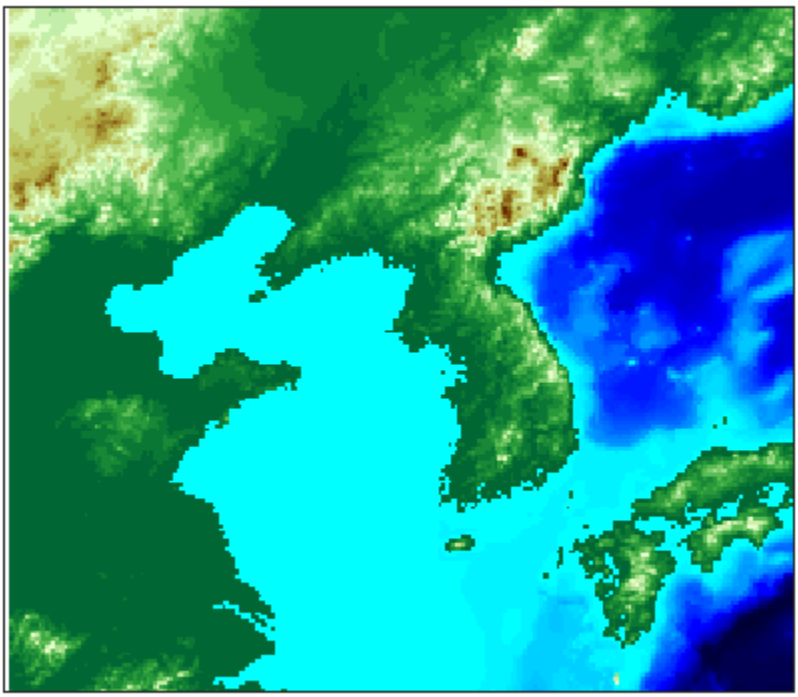
The `meshlsrm` and `surflsrm` functions simulate lighting by modifying the colormap with bands of light and dark. The map matrix is then converted to indices for the new "shaded" colormap based on calculated surface normals. Using light objects allows for a wide range of lighting effects. The toolbox manages light objects with the `lightm` function, which depends upon the MATLAB `light` function. Lights are separate MATLAB graphic objects.

For more information, consult the reference pages for `lightm`, `daspectm`, `material`, `lighting`, and `view`, along with “Lighting, Transparency, and Shading”.

Illuminate Color 3-D Relief Maps with Light Objects

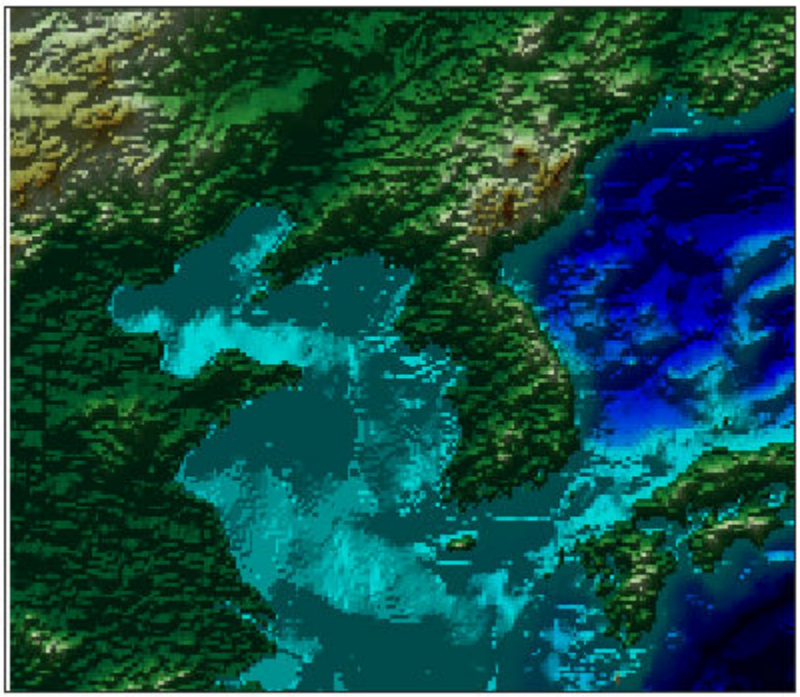
Add a light source to a surface colored data grid using `lightm`. First, load elevation data and a geographic cells reference object for the Korean peninsula. Display the data without lighting effects using `meshm`. Apply a colormap appropriate for elevation data using `demcmap`. Eliminate extra white space around the map using `tightmap`.

```
load korea5c
axesm('miller','MapLatLimit',[30 45],'MapLonLimit',[115 135])
meshm(korea5c,korea5cR,size(korea5c),korea5c)
demcmap(korea5c)
tightmap
```



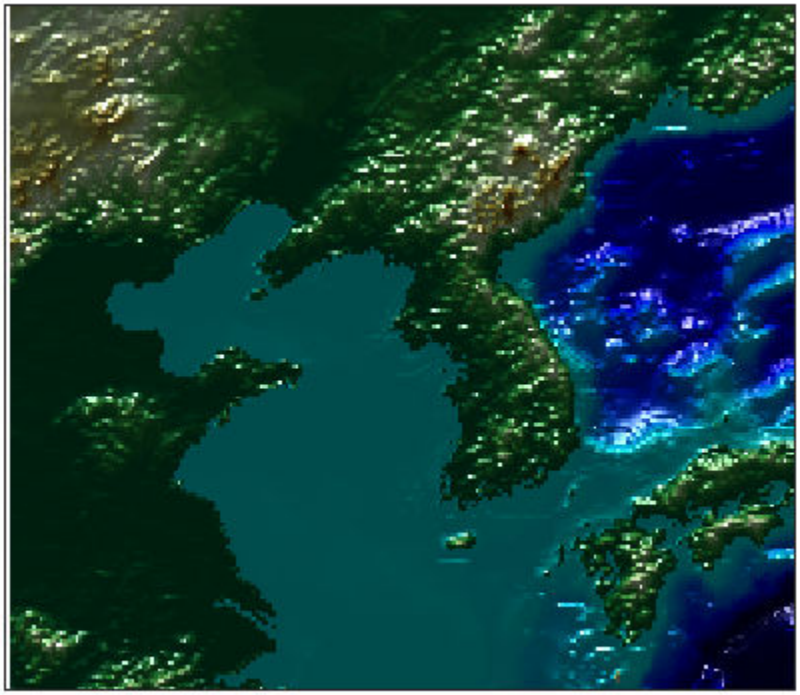
Place a light source at the northwest corner of the grid, one degree high, using `lightm`. The `lightm` function is similar to the MATLAB® function `light`, but accepts latitude and longitude inputs instead of *x*, *y*, and *z*. Note that the figure becomes darker.

```
lightm(45,115,1)
```



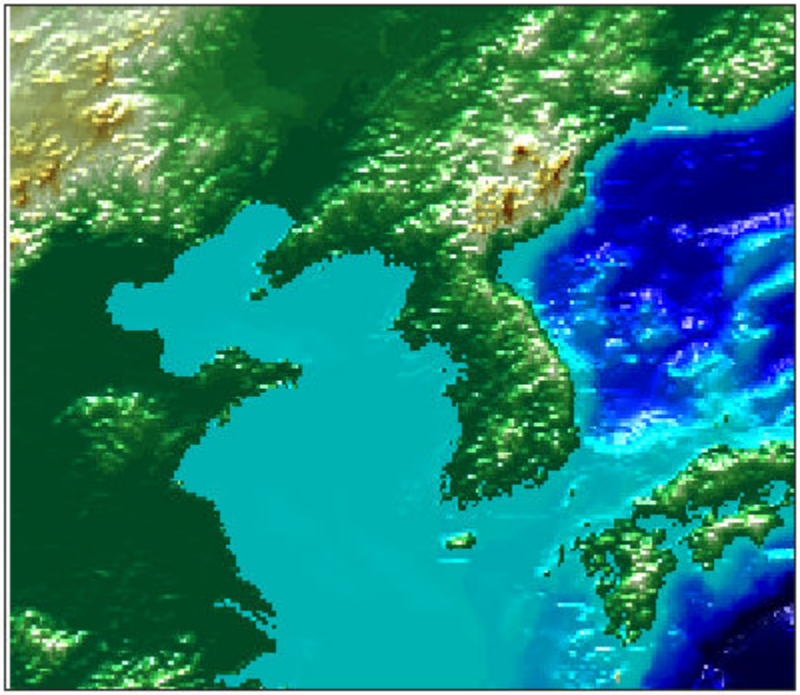
Exaggerate the vertical dimension to make any relief viewable in perspective. Note that the figure becomes darker still.

```
daspectm('meters',50)
```



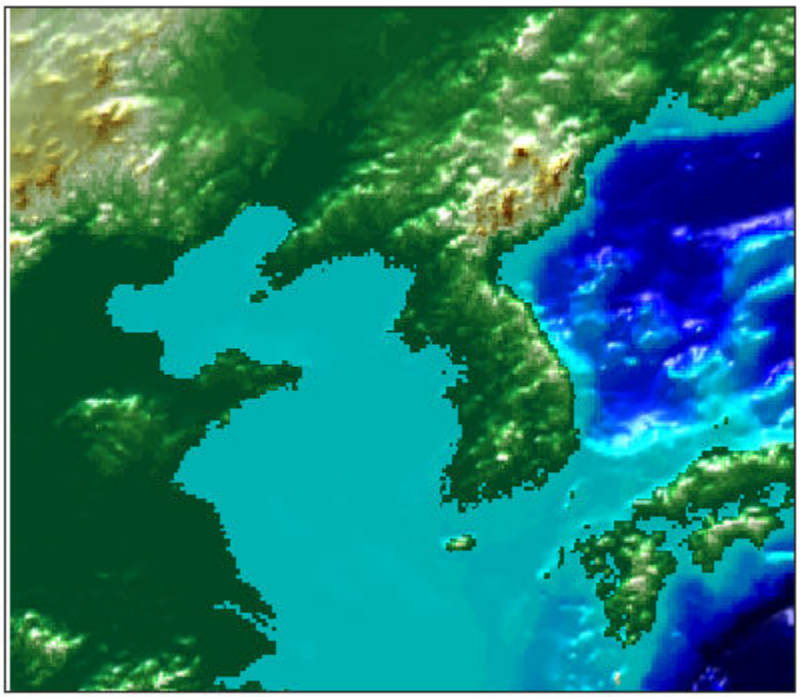
Set the ambient (direct), diffuse (skylight), and specular (highlight) surface reflectivity characteristics, respectively.

```
material([0.7 0.9 0.8])
```



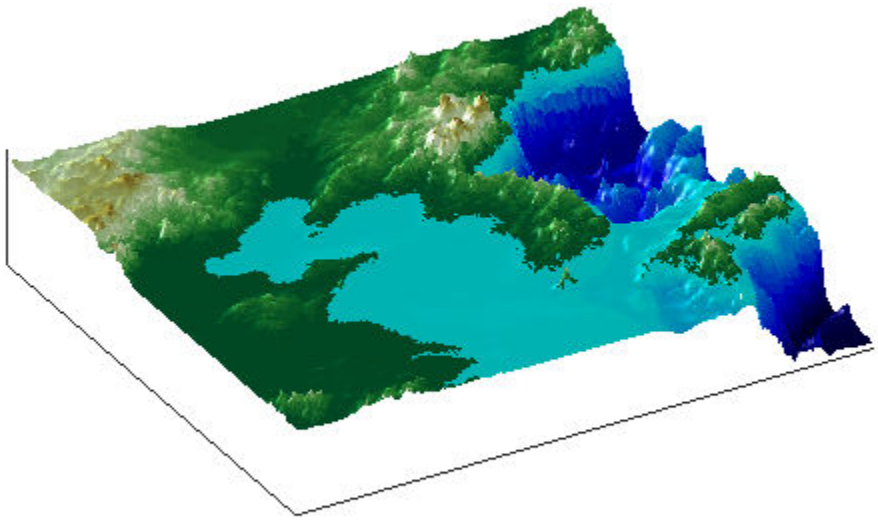
By default, the lighting is flat (plane facets). Change the light to use Gouraud shading (interpolated normal vectors at facet corners).

```
lighting Gouraud
```



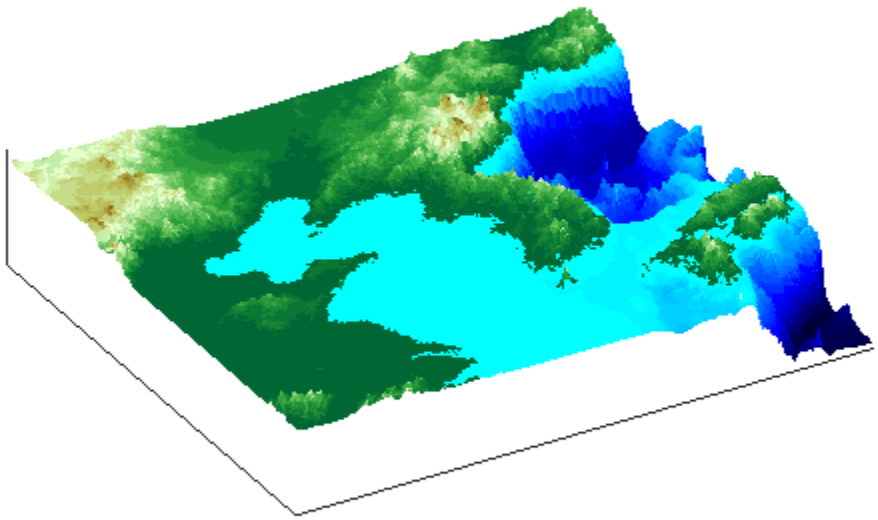
Remove the edges of the bounding box. Change the view by specifying an azimuth of -30º and an altitude of 30º.

```
ax = gca;
ax.Box = 'off';
view(-30,30)
```



If there is only one light in the current figure, you can remove it using `clmo`.

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



Drape Data on Elevation Maps

Combine Elevation Maps with Other Kinds of Data

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.

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.

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.

Drape Data over Terrain with Di erent 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. That is, you can construct a geolocated grid version of the regular data grid values or construct a regular grid version of the geolocated data grid values.

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 `geointerp` function. To combine dissimilar grids, you can do one of the following:

The following two examples illustrate these closely related approaches.

- “Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid” on page 5-30
- “Drape Geolocated Grid on Regular Data Grid via Texture Mapping” on page 5-36

Drape Geoid Heights Over Topography

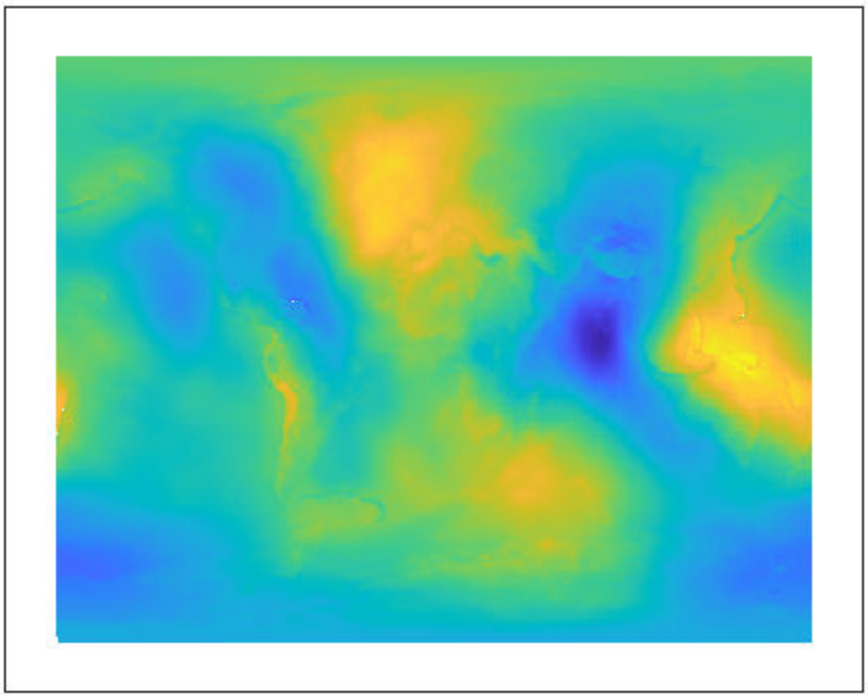
Display geoid data draped over topographic relief. For this example, display the geoid data as a color attribute instead of a 3-D surface.

Load topographic raster data and a geographic cells reference object. Get geoid heights by calling the `egm96geoid` function and specifying the reference object.

```
load topo60c
[N,R] = egm96geoid(topo60cR);
```

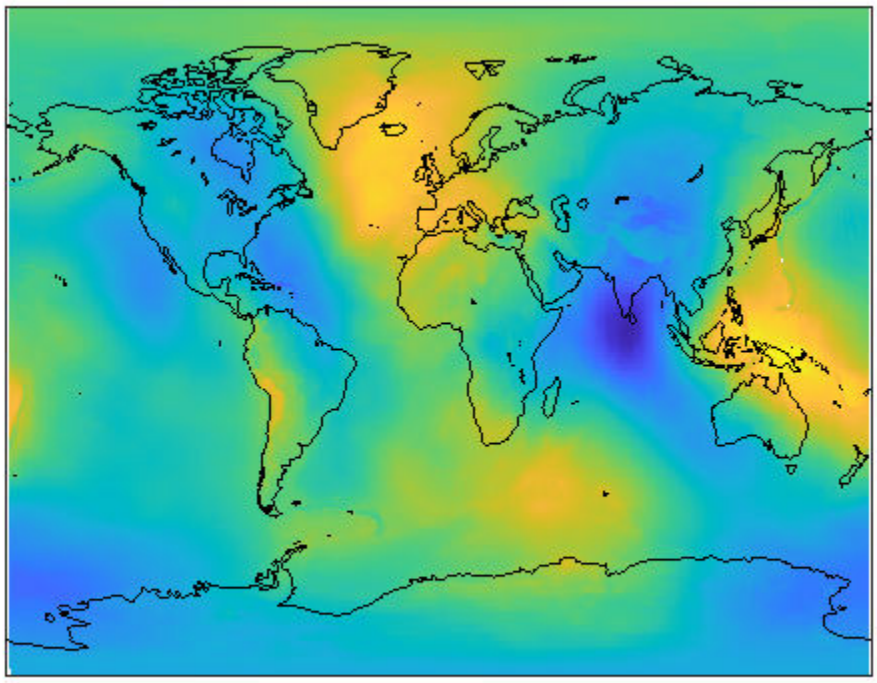
Create a map axes object using a Gall stereographic cylindrical projection (a perspective projection). Use `meshm` to plot a colored display of the geoid's variations, but specify `topo60c` as the final argument, to give each geoid grid cell the height (*z* value) of the corresponding topographic grid cell. Low geoid heights are shown as blue, high ones as yellow.

```
axesm gstereo;
meshm(N,R,size(N),topo60c)
```



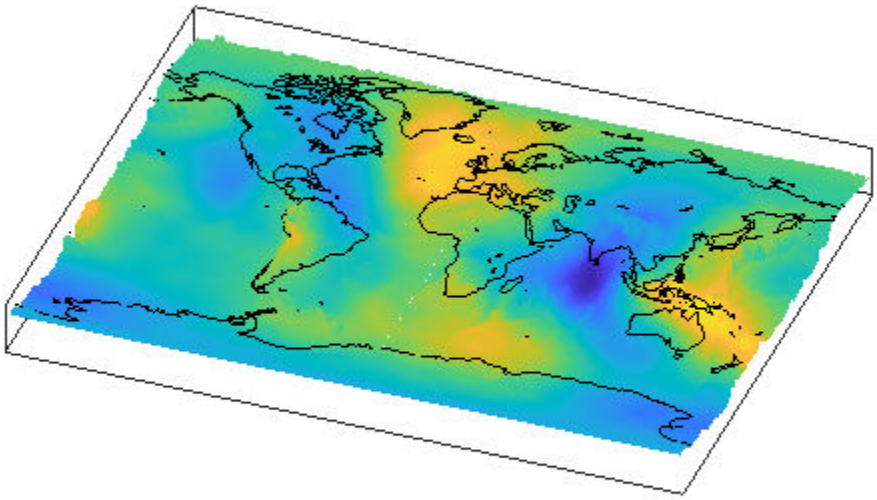
For reference, plot the world coastlines in black, raise their elevation to 1000 meters (high enough to clear the surface in their vicinity), and expand the map to fill the frame.

```
load coastlines
plotm(coastlat,coastlon,'k')
zdatam(handlem('allline'),1000)
tightmap
```



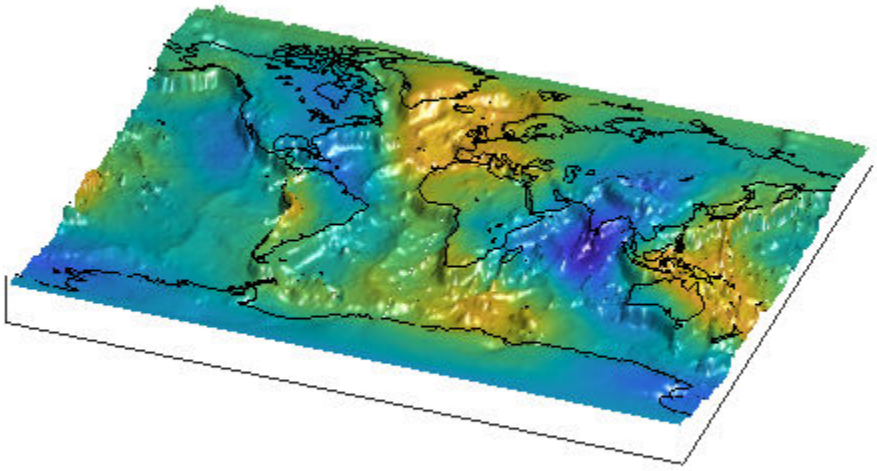
Due to the vertical view and lack of lighting, the topographic relief is not visible, but it is part of the figure's surface data. Bring it out by exaggerating relief greatly, and then setting a view from the south-southeast.

```
daspectm('m',200); tightmap
view(20,35)
```



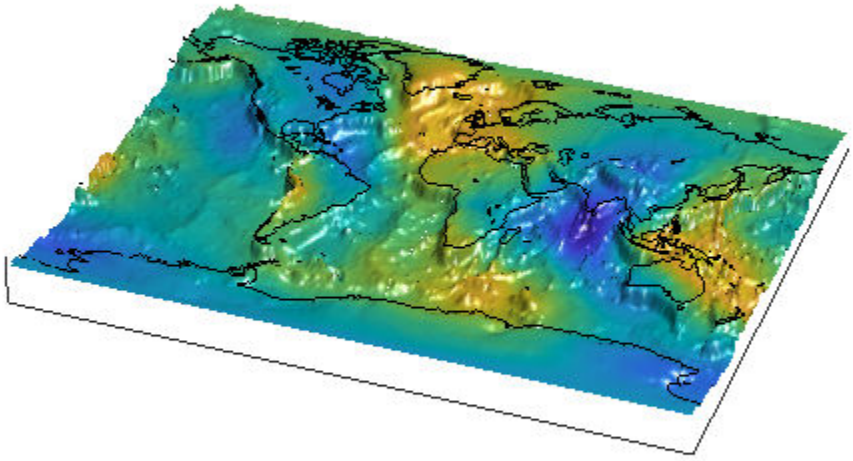
Remove the bounding box, shine a light on the surface (using the default position, offset to the right of the viewpoint), and render again with Gouraud shading.

```
ax = gca;
ax.Box = 'off';
camlight;
lighting Gouraud
```



Finally, set the perspective to converge slightly (the default perspective is orthographic). Notice 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.

```
ax.Projection = 'perspective';
```



Combine Dissimilar Grids by Converting Regular Grid to Geolocated Data Grid

This example shows how to combine an elevation data grid and an attribute (color) data grid that cover the same region but are gridded differently. The example drapes slope data from a regular data grid on top of elevation data from a geolocated data grid. 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. This approach works with any dissimilar grids, although the two data sets in this example actually have the same origin (the geolocated grid derives from the `topo60c` data set).

Load the geolocated data grid from the `mapmtx` file and the regular data grid from the `topo60c` file. The `mapmtx` file actually contains two regions but this example only uses the diamond-shaped portion, `lt1`, `lg1`, and `map1`, centered on the Middle East.

```
load mapmtx lt1 lg1 map1
load topo60c
```

Compute the surface aspect, slope, and gradients for `topo60c`. This example only uses the slopes in subsequent steps.

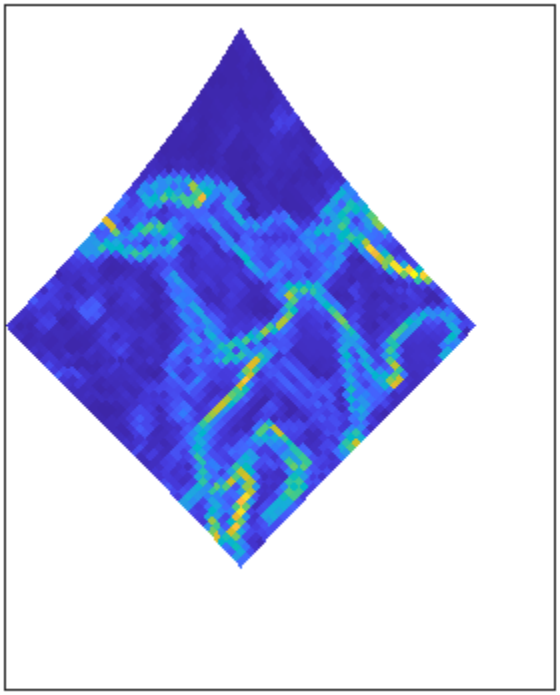
```
[aspect,slope,gradN,gradE] = gradientm(topo60c,topo60cR);
```

Use the `geointerp` function to interpolate slope values to the geolocated grid specified by `lt1` and `lg1` . The output is a 50-by-50 grid of elevations matching the coverage of the `map1` variable.

```
slope1 = geointerp(slope,topo60cR,lt1,lg1);
```

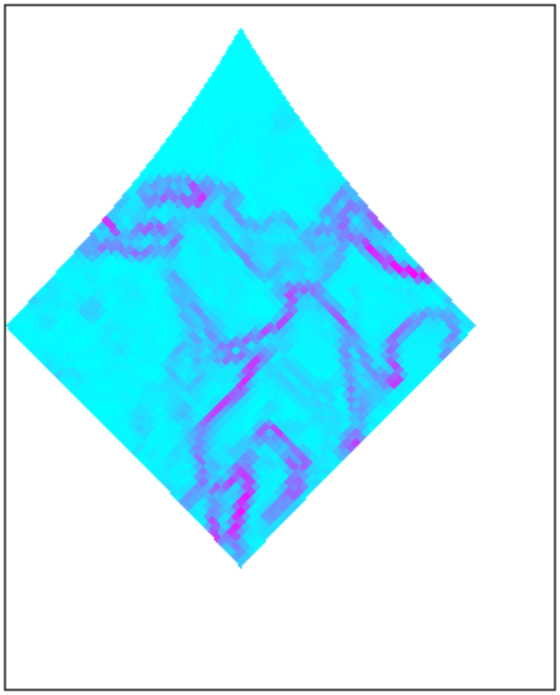
Set up a figure with a Miller projection and use `surfm` to display the slope data. Specify the *z* -values for the surface explicitly as the `map1` data, which is terrain elevation. The map mainly depicts steep cliffs, which represent mountains (the Himalayas in the northeast), and continental shelves and trenches.

```
figure
axesm miller
surfm(lt1,lg1,slope1,map1)
```



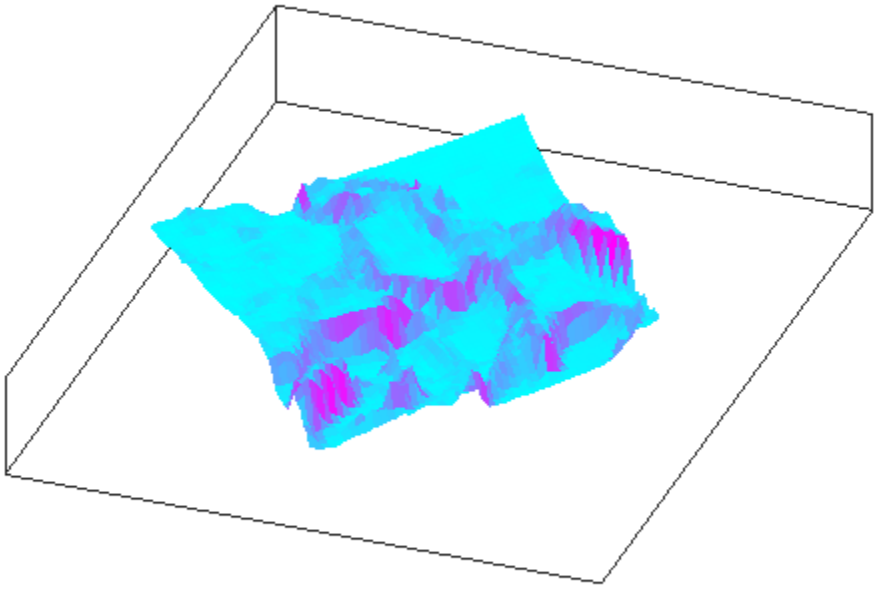
The coloration depicts steepness of slope. Change the colormap to make the steepest slopes magenta, the gentler slopes dark blue, and the flat areas light blue:

```
colormap cool
```



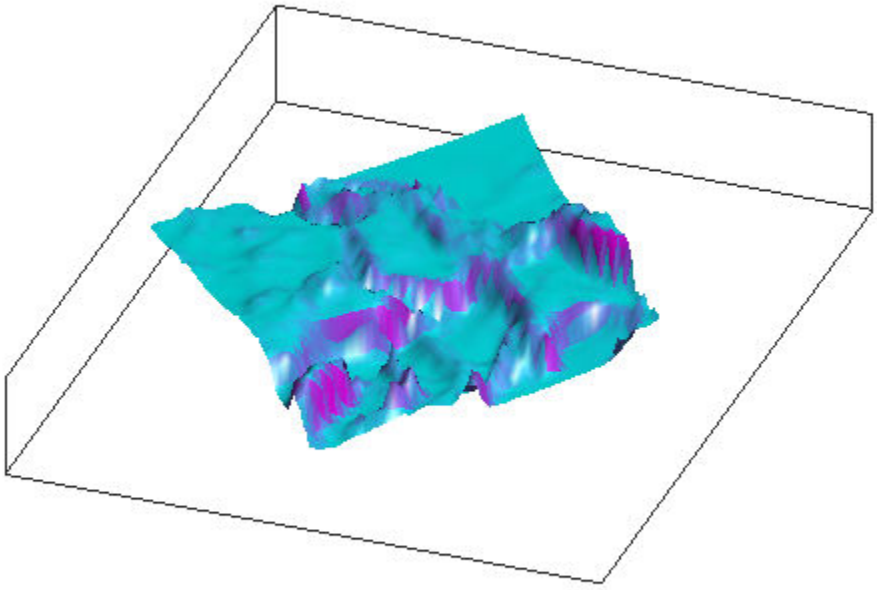
Use `view` to get a southeast perspective of the surface from a low viewpoint. In 3-D, you immediately see the topography as well as the slope.

```
view(20,30)
daspectm('meter',200)
```



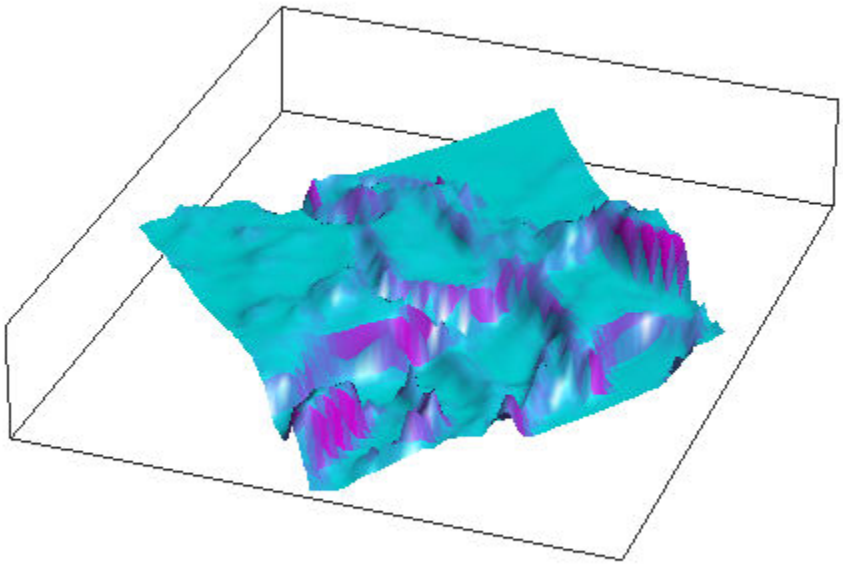
The default rendering uses faceted shading (no smooth interpolation). Render the surface again, this time making it shiny with Gouraud shading and lighting from the east (the default of `camlight` lights surfaces from over the right shoulder of the viewer).

```
material shiny
camlight
lighting Gouraud
```



Finally, remove white space and re-render the figure in perspective mode.

```
axis tight
ax = gca;
ax.Projection = 'perspective';
```



Drape Geolocated Grid on Regular Data Grid via Texture Mapping

This example shows how to 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. The example combines dissimilar data grids by creating a new regular data grid that covers the region of the 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. Color and elevation grids do not have to be the same size. If the resolutions of the two data grids 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'`.

Load elevation raster data and a geographic cells reference object from `topo60c.mat`. Get individual variables containing terrain data from `mapmtx.mat`.

```
load topo60c
load mapmtx lt1 lg1 map1
```

Identify the geographic limits of the geolocated grid that was loaded from `mapmtx`.

```
latlim(1) = 2*floor(min(lt1(:))/2);
lonlim(1) = 2*floor(min(lg1(:))/2);
latlim(2) = 2*ceil(max(lt1(:))/2);
lonlim(2) = 2*ceil(max(lg1(:))/2);
```

Crop the global elevation data to the rectangular region enclosing the smaller grid.

```
[topo1,topo1R] = geocrop(topo60c,topo60cR,latlim,lonlim);
```

Allocate a regular grid filled uniformly with `-Inf`, to receive texture data.

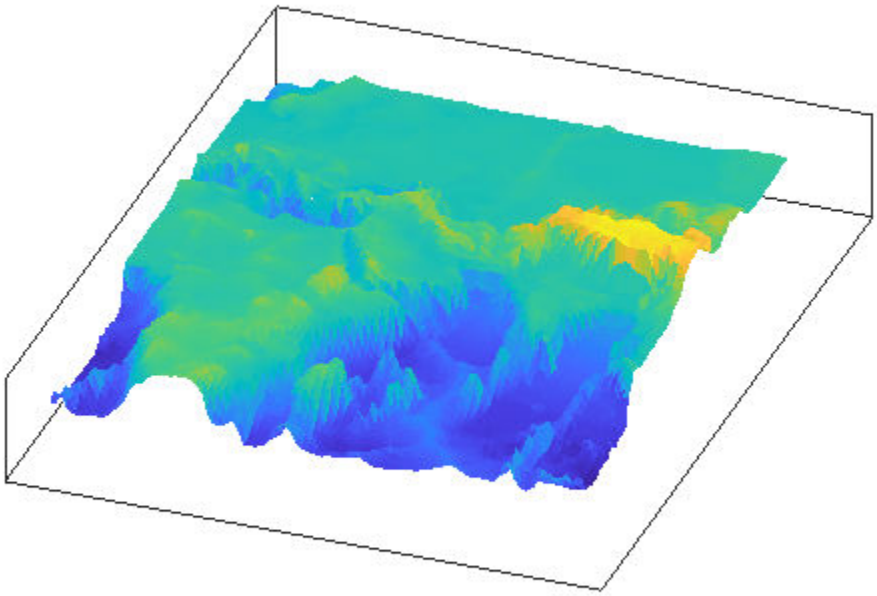
```
L1R = georefcells(latlim,lonlim,2,2);
L1 = zeros(L1R.RasterSize);
L1 = L1 - Inf;
```

Overwrite `L1` using `imbedm`, converting it from a geolocated grid to a regular grid, in which the values come from the discrete Laplacian of the elevation grid `map1`.

```
L1 = imbedm(lt1,lg1,del2(map1),L1,L1R);
```

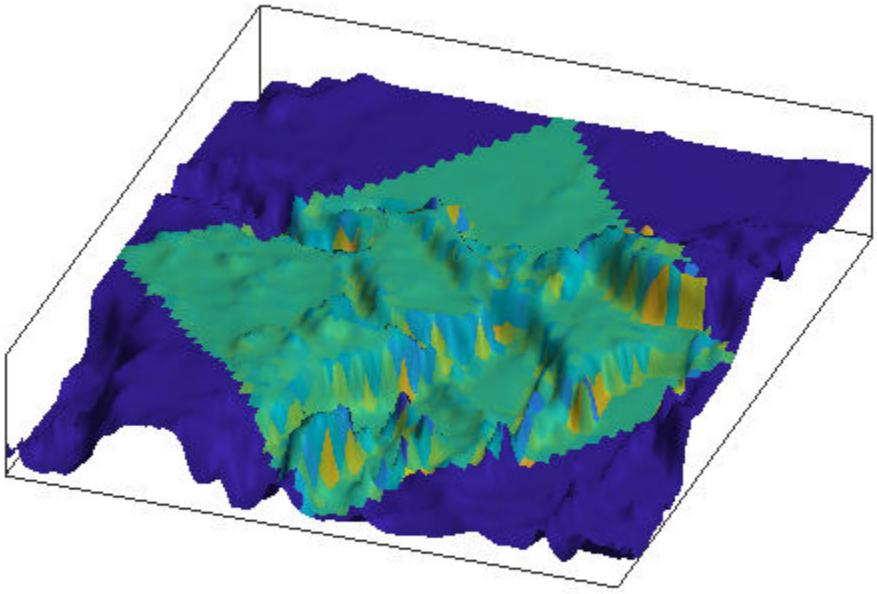
Set up a map axes with the Miller projection and use `meshm` to display the cropped data. Render the figure as a 3-D view from a 20 degree azimuth and 30 degree altitude, and exaggerate the vertical dimension by a factor of 200. Both the surface relief and coloring represent topographic elevation.

```
figure
axesm miller
h = meshm(topo1,topo1R,size(topo1),topo1);
view(20,30)
daspectm('m',200)
```



Apply the `L1` matrix as a texture map directly to the surface using the `set` function. The area not covered by the `[lt1,lg1,map1]` geolocated data grid appears dark blue because the corresponding elements of L1 were set to `-Inf`.

```
h.CData = L1;
h.FaceColor = 'texturemap';
material shiny
camlight
lighting gouraud
axis tight
```



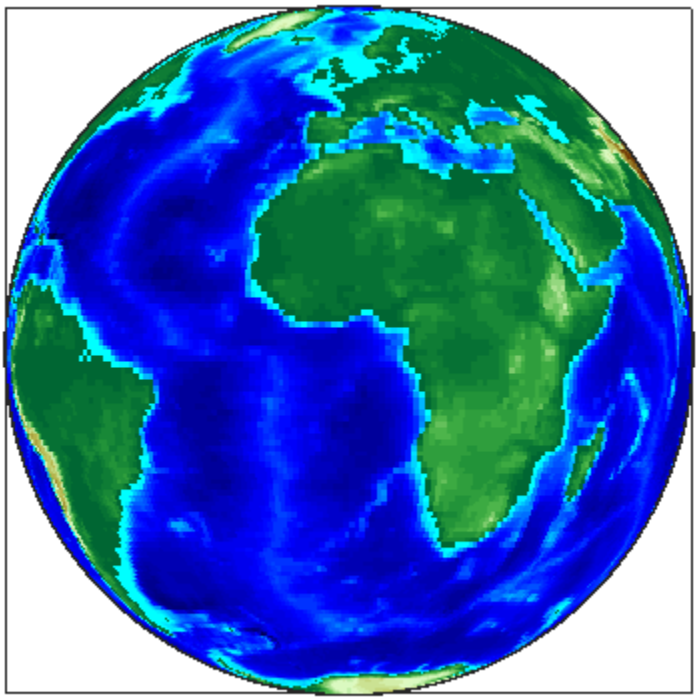
The Globe Display Compared with the Orthographic Projection

This 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.

Render 2-D Orthographic Projection

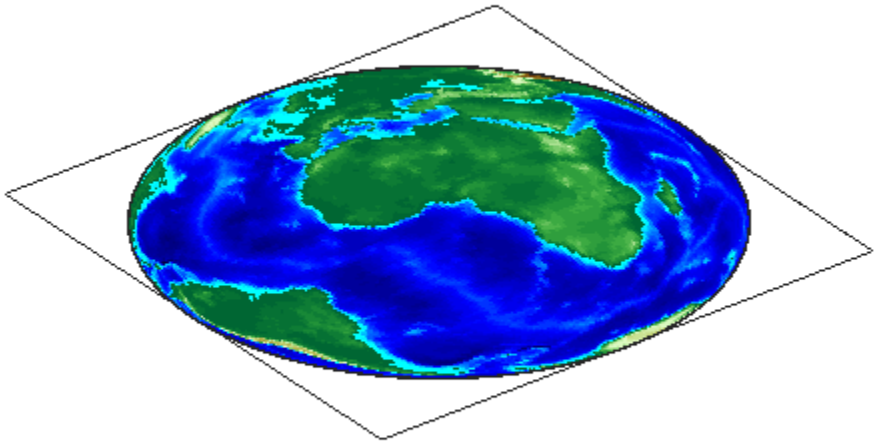
Load elevation raster data and a geographic cells reference object. Display the data using a two-dimensional orthographic map projection.

```
load topo60c
axesm ortho
framem
meshm(topo60c,topo60cR)
demcmap(topo60c)
```



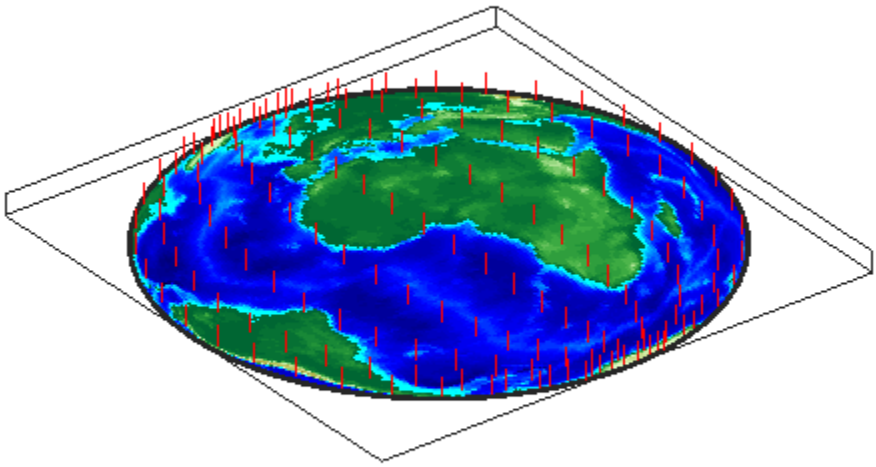
View the map obliquely.

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



You can view the map 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(topo60c,topo60cR,[20 20]);
stem3m(latgrat,longrat,500000*ones(size(latgrat)),'r')
```

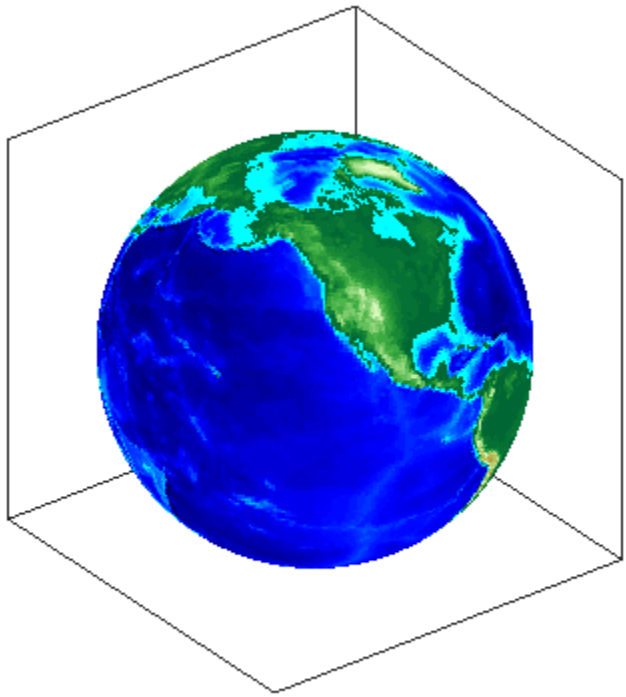


Use the **Rotate 3D** tool on the figure window toolbar to change your viewpoint. No matter how you position the view, you are looking at a disc with stems protruding perpendicularly.

Render 3-D Globe Display

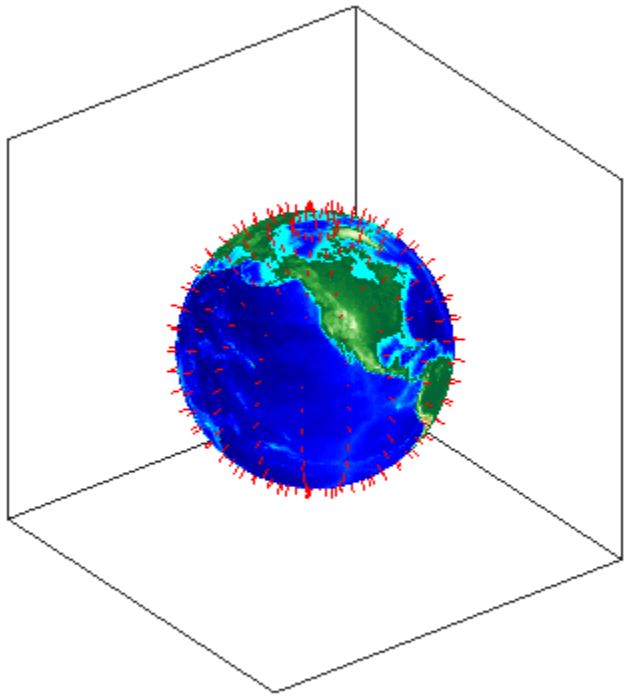
Display the elevation data using a three-dimensional globe rather than an orthographic projection.

```
figure
axesm('globe','Geoid',earthRadius)
meshm(topo60c,topo60cR)
demcmap(topo60c)
view(3)
```



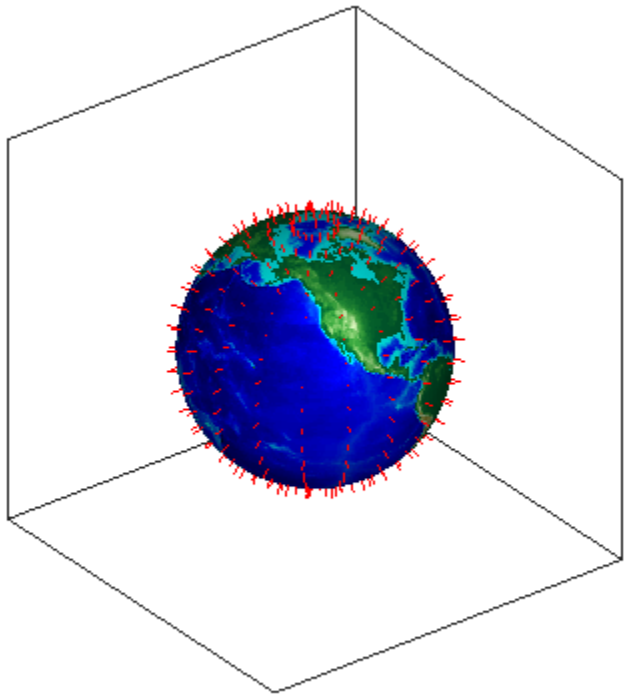
Include the stem plot to visualize the difference in surface normals on a sphere.

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



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')
```



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.

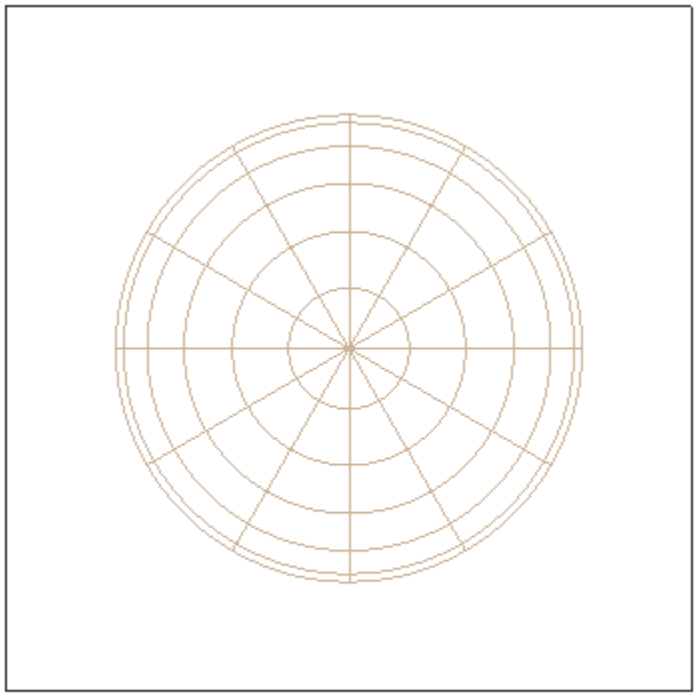
For additional information on functions used in this example, see the reference pages for `view`, `camlight`, `meshgrat`, and `stem3m`.

Use Opacity and Transparency in Globe Displays

This example shows how to create an opaque surface over which you can display line and point data. This can be useful with Globe displays that 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.

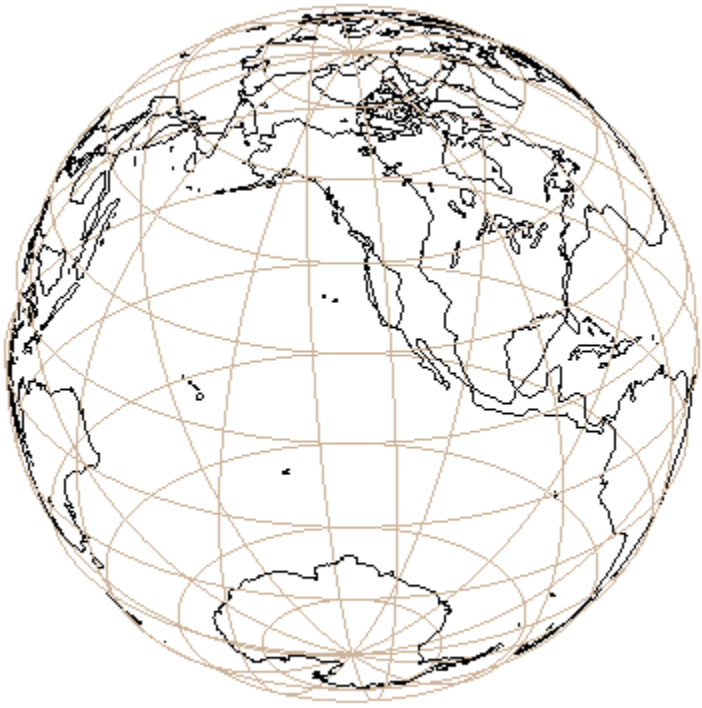
Create a figure, set up a Globe display, and draw a graticule in a light color, slightly raised from the surface. To ensure that the surface displays over the entire globe, set the Clipping property of the axes object to `'off'`.

```
figure
ax = axesm('globe');
ax.Clipping = 'off';
gridm('GLineStyle','-','Gcolor',[.8 .7 .6],'Galtitude',.02)
```



Load and plot the coast data in black, and set up a 3-D perspective. Use the Rotate 3D tool on the figure's toolbar to rotate the view. Note how confusing the display is because of its transparency.

```
load coastlines
plot3m(coastlat,coastlon,.01,'k')
view(3)
axis off
zoom(2)
```



Make a uniform 1-by-1-degree grid and create a raster referencing object for it.

```
base = zeros(180,360);
baseR = georefcells([-90 90],[0 360],size(base));
```

Render the grid onto the globe, color it copper, light it from camera right, and make the surface reflect more light. The copper surface effectively hides all lines on the back side of the globe.

```
copperColor = [0.62 0.38 0.24];
geoshow(base,baseR,'FaceColor',copperColor)
camlight right
material([.8 .9 .4])
```

Access Basemaps and Terrain for Geographic Globe

Geographic globe objects created using the `geoglobe` function plot data over basemaps and terrain. You can access different basemap and terrain choices in different ways.

MathWorks offers a selection of basemaps, including two-tone, color terrain, and high-zoom-level displays. Six of the basemaps are tiled data sets created using Natural Earth. Five of the basemaps are high-zoom-level maps provided by Esri. For more information about basemap options, see `geobasemap`.

Use Installed Basemap

The `'darkwater'` basemap is installed with MATLAB. The other basemaps are not installed with MATLAB, but you can access them over an Internet connection.

Download Basemaps

To work offline or to improve map responsiveness, you can download the basemaps created using Natural Earth onto your local system. The basemaps provided by Esri are not available for download.

Download basemaps using the Add-On Explorer.

1. On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Add-Ons**.
2. In the Add-On Explorer, scroll to the **MathWorks Optional Features** section, and click **Show All** to find the basemap add-ons. You can also search for the basemap add-ons by name (listed in the following table) or click **Optional Features** in **Filter by Type**.
3. Select the basemap add-ons that you want to download.

Basemap Name	Basemap Data Package Name
`'bluegreen'`	MATLAB Basemap Data - bluegreen
`'grayland'`	MATLAB Basemap Data - grayland
`'colorterrain'`	MATLAB Basemap Data - colorterrain
`'grayterrain'`	MATLAB Basemap Data - grayterrain
`'landcover'`	MATLAB Basemap Data - landcover

Add Custom Basemaps

Add custom basemaps using the `addCustomBasemap` function. An active Internet connection is required to add and use custom basemaps.

Access Terrain

By default, the geographic globe uses terrain data hosted by MathWorks and derived from the GMTED2010 model by the USGS and NGA. You need an active Internet connection to access this terrain data, and you cannot download it.

To work offline or to improve terrain responsiveness, add custom terrain from DTED files using the `addCustomTerrain` function. You do not need an active Internet connection to add or use custom terrain.

Alternatively, you can set the `Terrain` property of the geographic globe object to `'none'`.

Specify Basemaps and Terrain

To specify a basemap for a geographic globe, you can either:

- Use the `geobasemap` function. Specify the geographic globe as the first argument.

  ```
  uif = uifigure;
  g = geoglobe(uif);
  geobasemap(g,'streets')
  ```

- Set the `Basemap` property of the `GeographicGlobe` object. You can set this property by using a name-value pair or by using dot notation.

  ```
  uif = uifigure;
  g = geoglobe(uif,'Basemap','streets');
  g.Basemap = 'topographic';
  ```

To specify terrain for the geographic globe, set the `Terrain` property of the `GeographicGlobe` object. You can set this property by using a name-value pair or by using dot notation.

```
uif = uifigure;
g = geoglobe(uif,'Terrain','none');
g.Terrain = 'gmted2010';
```

See Also

`addCustomBasemap` | `addCustomTerrain` | `geobasemap` | `geoglobe`

More About

- “System Requirements for Graphics”
- “Resolving Low-Level Graphics Issues”

Create Interactive Basemap Picker

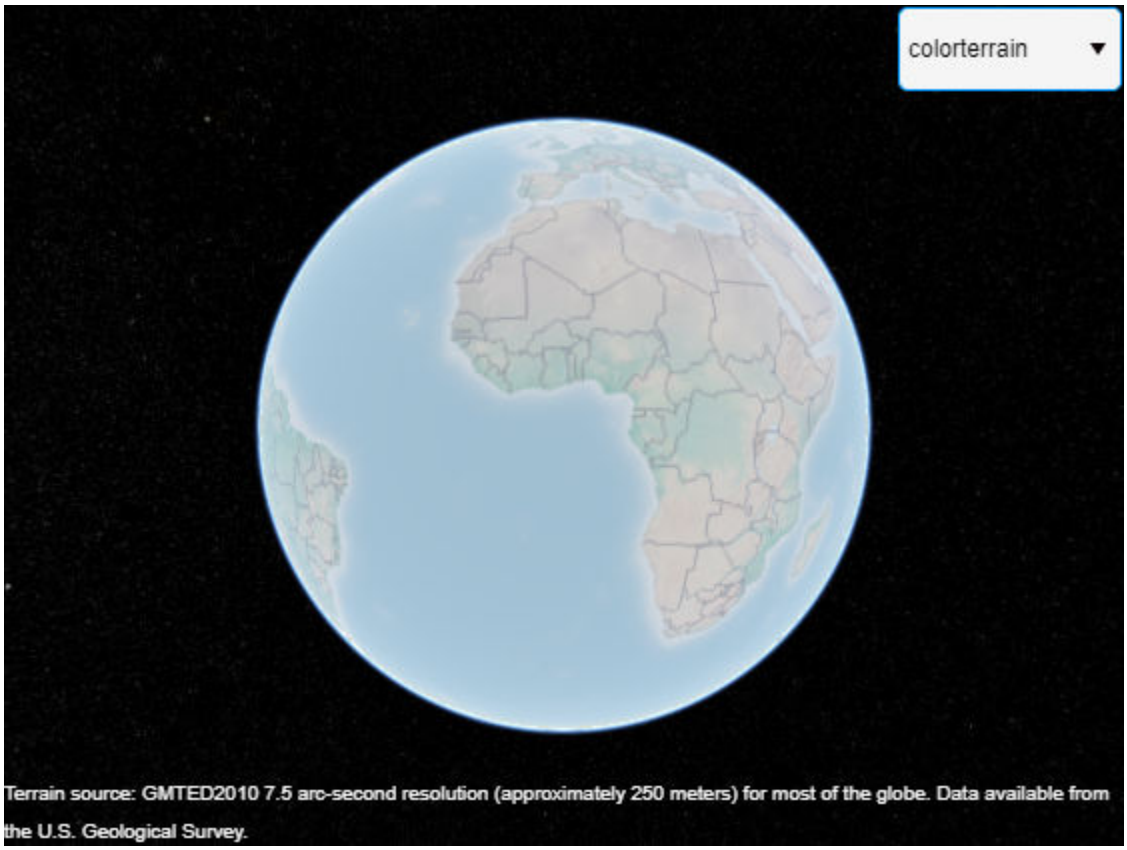
Interactively change the basemap of a geographic globe by adding a drop-down menu to the figure.

First, create a program file called `basemapPicker.m`. Within the program file:

- Create a geographic globe in a figure created using the `uifigure` function.
- Specify a position for the menu. In this example, the values of `x`, `y`, `w`, and `h` position the menu in the upper-right corner of the figure window.
- Specify the basemaps to include in the menu.
- Create the menu. Use a `ValueChangedFcn` callback that executes when you make a selection from the menu. The callback changes the basemap using the `geobasemap` function.
- Write custom code to reposition the menu when you change the size of the figure. To do this, disable automatic resizing of the menu. Then, create custom behavior by defining a `SizeChangedFcn` callback. The `repositionDropdown` function repositions the menu, so that it stays in the upper-right corner of the figure.

```
function basemapPicker
    uif = uifigure;
    gl = geoglobe(uif);

    x = 0.8;
    y = 0.9;
    w = 0.2;
    h = 0.1;
    uifW = uif.Position(3);
    uifH = uif.Position(4);
    pos = [x*uifW y*uifH w*uifW h*uifH];

    basemaps = ["satellite" "streets" "streets-light" "streets-dark" ...
                "landcover" "darkwater" "grayland" "bluegreen" ...
                "grayterrain" "colorterrain"];

    dd = uidropdown(uif,'Position',pos,'Items',basemaps);
    dd.ValueChangedFcn = @(src,eventdata)geobasemap(gl,src.Value);

    uif.AutoResizeChildren = 'off';
    uif.SizeChangedFcn = @(src,eventdata)repositionDropdown(dd,x,y,w,h);
end

function repositionDropdown(dd,x,y,w,h)
    fig = dd.Parent;
    uifW = fig.Position(3);
    uifH = fig.Position(4);
    dd.Position = [x*uifW y*uifH w*uifW h*uifH];
end
```

Run the program file. Change the basemap to `'colorterrain'` using the drop-down menu.



See Also

`geobasemap` | `geoglobe` | `uidropdown`

More About

- “Callback Definition”

Visualize Aircraft Line-of-Sight Over Terrain

This example shows how to compute and visualize the line-of-sight visibility of an aircraft from a ground location over terrain. First, import terrain data for a region and apply it to a 3-D geographic globe. Then, perform point-to-point visibility analysis from a ground location to a simulated flight path, and display the results on a 3-D geographic globe. Finally, perform point-to-area visibility analysis from the ground location corresponding to aircraft flying at constant altitude, and display the results on a 2-D geographic axes.

Use line-of-sight analysis for ground-to-air scenarios where unobstructed visibility is important, such as for radar surveillance, communications, and UAV path planning. This example applies the analysis to radar surveillance for an airport.

Import Terrain Data

Specify a DTED-format terrain file to use for data analysis and 3-D visualization. The terrain file was downloaded from the "SRTM Void Filled" data set available from the United States Geological Survey (USGS).

```
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey.";
```

Import DTED file data into the workspace as an array and a geographic raster reference object, specifying the return type as double so that the data works with all analysis functions.

```
[Zterrain,Rterrain] = readgeoraster(dtedfile,"OutputType","double");
```

View the geographic limits and sample resolution of the terrain data by accessing properties of the geographic raster reference object. The limits for the file correspond to the region around Boulder, Colorado, US, and the resolution corresponds to the DTED level-1 format, which has sample resolution of 3 arc seconds, or about 90 meters.

```
latlim = Rterrain.LatitudeLimits;
lonlim = Rterrain.LongitudeLimits;
latspc = Rterrain.SampleSpacingInLatitude;
lonspc = Rterrain.SampleSpacingInLongitude;
disp("Latitude limits of terrain: " + mat2str(latlim) + newline + ...
    "Longitude limits of terrain: " + mat2str(lonlim) + newline + ...
    "Terrain resolution in latitude: " + latspc*3600 + " arc seconds" + newline + ...
    "Terrain resolution in longitude: " + lonspc*3600 + " arc seconds")
```

```
Latitude limits of terrain: [39 40]
Longitude limits of terrain: [-106 -105]
Terrain resolution in latitude: 3 arc seconds
Terrain resolution in longitude: 3 arc seconds
```

Visualize Aircraft Trajectory Line-of-Sight on a 3-D Map

Create Geographic Globe with Custom Terrain

Add custom data with the DTED file for use with 3-D visualization.

```
addCustomTerrain("southboulder",dtedfile,"Attribution",attribution)
```

Specify the custom terrain with a new geographic globe. Preserve the custom terrain on the globe when data is added by setting hold on.

```
fig = uifigure;
g = geoglobe(fig,"Terrain","southboulder");
hold(g,"on")
```

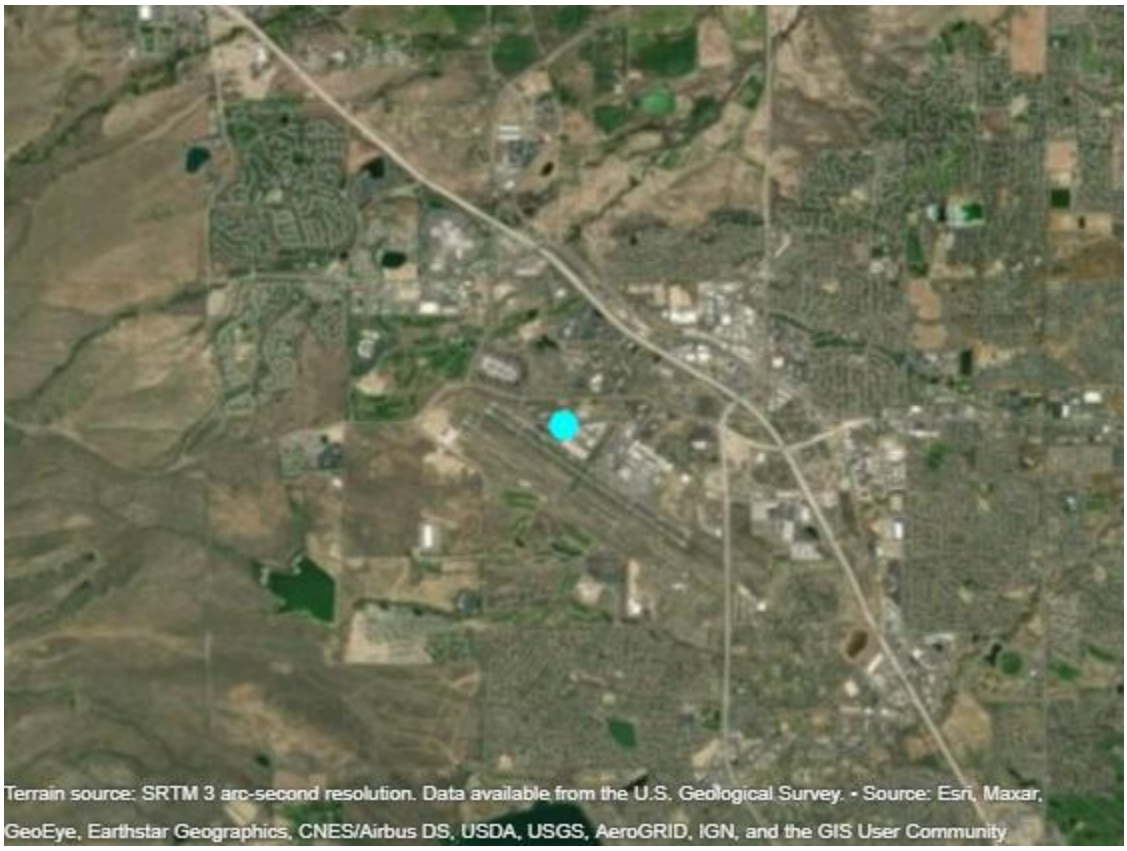
View Radar Ground Location

Define a radar ground location at Rocky Mountain Metropolitan Airport. The radar is mounted on a tower 10 meters above the ground. The radar altitude is the sum of the ground elevation and the radar tower height, referenced to mean sea level.

```
rdrlat = 39.913756;
rdrlon = -105.118062;
rdrtowerht = 10;
rdralt = 1717 + rdrtowerht;
```

Plot the radar location on the geographic globe.

```
geoplot3(g,rdrlat,rdrlon,rdralt,"co", ...
    "LineWidth",6, ...
    "MarkerSize",1)
```



Simulate Aircraft Trajectory

Simulate the trajectory of an aircraft circling over the mountains.

Define the center location of a circling aircraft.

```
tlat0 = 39.80384;
tlon0 = -105.49916;
tht0 = 3000;
```

Define trajectory waypoints for the aircraft using east-north-up (ENU) Cartesian coordinates. Specify a curve with a radius of 5 km (5000 m) and a vertical offset of 1 km (1000 m) over 1.5 revolutions. Then, convert the ENU coordinates to geodetic coordinates that are referenced to the WGS84 ellipsoid.

```
azs = 1:540;
r = 5000;
[X,Y] = pol2cart(deg2rad(azs),r);
Z = linspace(0,1000,numel(azs));
wgs84 = wgs84Ellipsoid;
[tlat,tlon,tht] = enu2geodetic(X,Y,Z,tlat0,tlon0,tht0,wgs84);
```

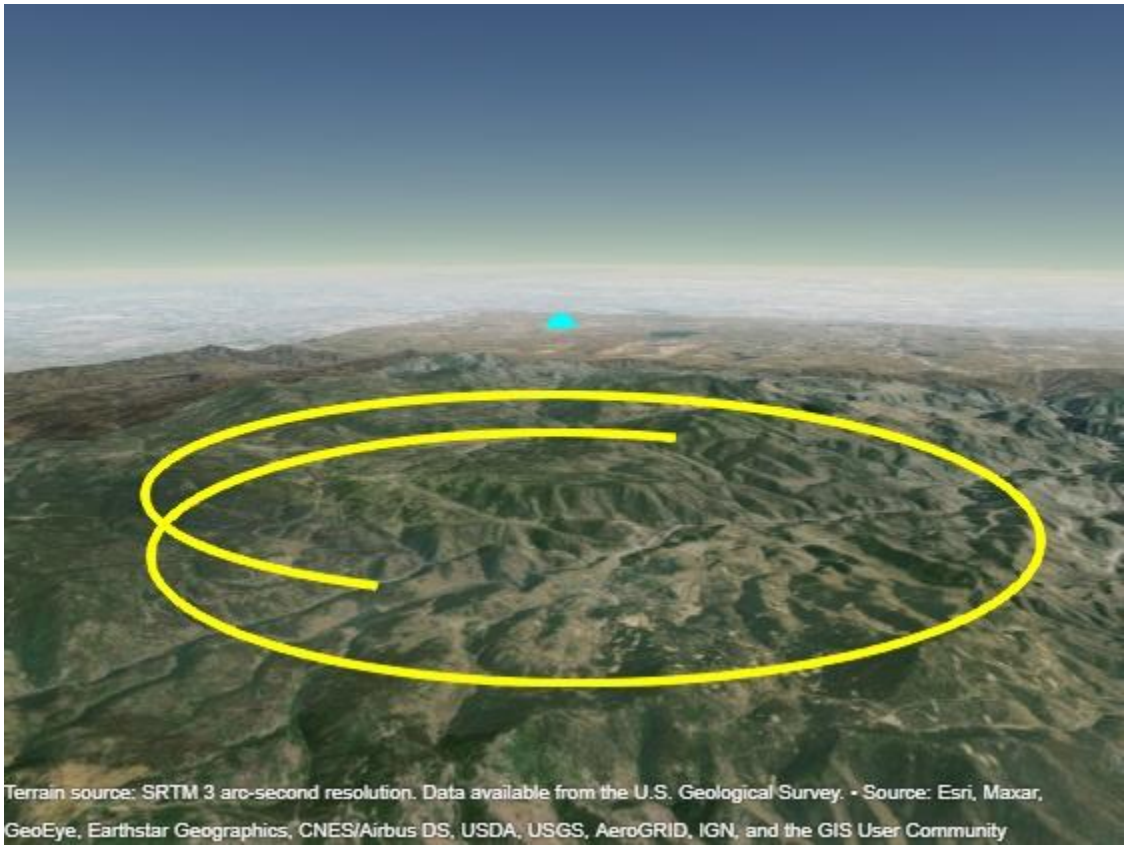
View Aircraft Trajectory Over Terrain

Plot the aircraft trajectory on the geographic globe. The default view, or camera position, is overhead and oriented down.

```
traj = geoplot3(g,tlat,tlon,tht,"y", ...
    "HeightReference","ellipsoid", ...
    "LineWidth",3);
```

View the 3-D terrain and radar location from a distance by changing the camera position and rotation angles.

```
campos(g,39.77114,-105.62662,6670)
camheading(g,70)
campitch(g,-12)
```



Compute Line-of-Sight Visibility with Aircraft Trajectory

Compute line-of-sight visibility with the `los2` function and the DTED data.

The `los2` function supports either orthometric height (height above mean sea level) or height above ground level. Convert the aircraft trajectory heights from ellipsoidal height to orthometric height. Then, compute the line-of-sight from the airport radar location to each aircraft trajectory waypoint and convert the results to a logical array.

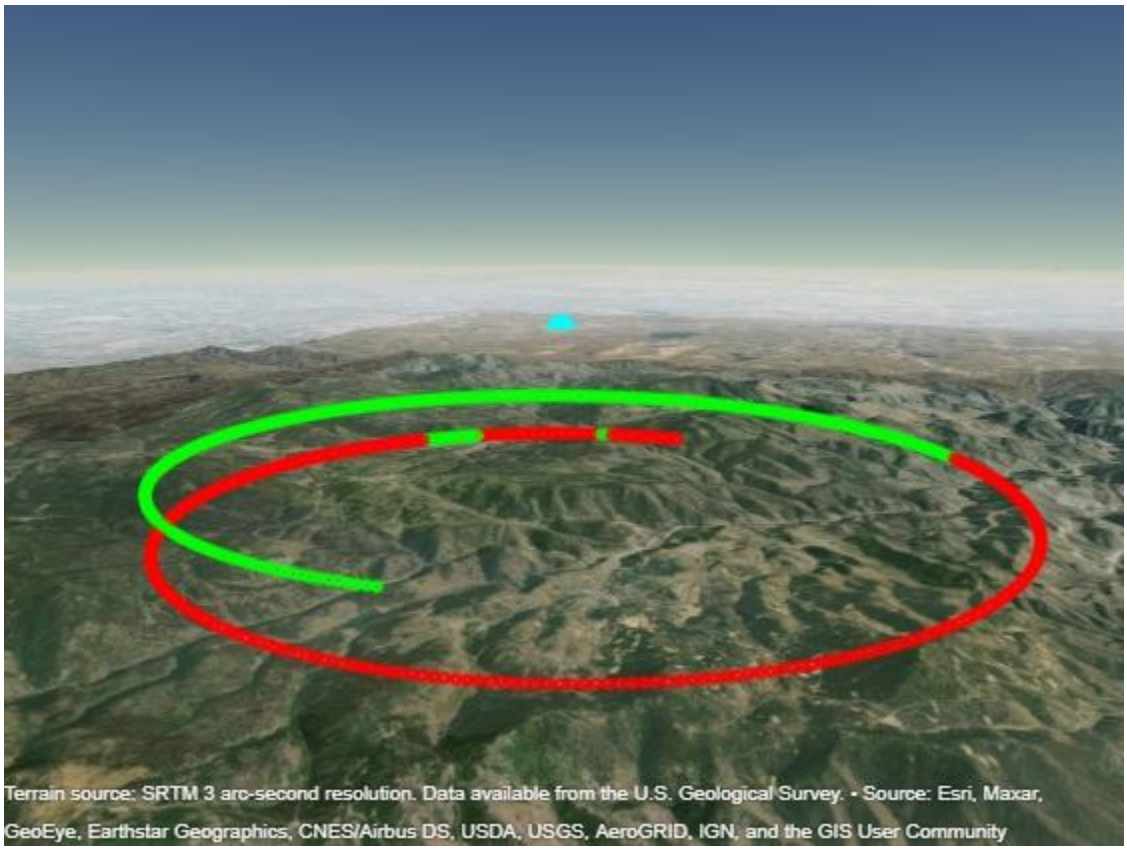
```
numwaypts = numel(tlat);
isvis = zeros(1,numwaypts);
talt = tht - egm96geoid(tlat,tlon);
for k = 1:numwaypts
    isvis(k) = los2(Zterrain,Rterrain,rdrlat,rdrlon,tlat(k),tlon(k),rdralt,talt(k),"MSL","MSL");
end
isvis = logical(isvis);
```

Note that `los2` calculates line-of-sight visibility assuming the data is referenced to a spherical Earth, whereas the data is actually referenced to the WGS84 ellipsoid, and as a result there may be minor discrepancies. The line-of-sight calculation also corresponds to optical line-of-sight and does not account for refraction through the atmosphere.

Visualize Line-of-Sight Visibility Over Terrain

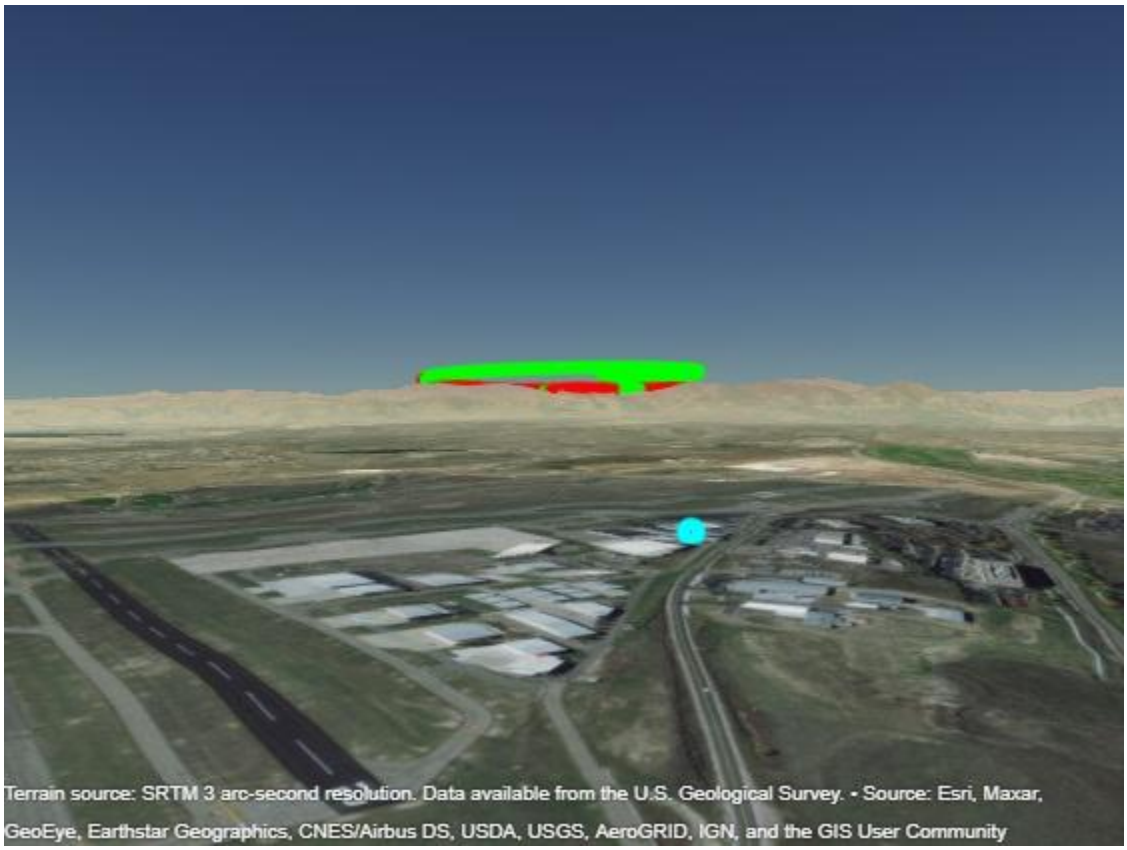
Plot the line-of-sight visibility. Use green markers where the aircraft is visible from the airport and red markers where it is not visible.

```
delete(traj)
geoplot3(g,tlat(isvis),tlon(isvis),tht(isvis),"og", ...
    "HeightReference","ellipsoid", ...
    "LineWidth",2, ...
    "MarkerSize",1)
geoplot3(g,tlat(~isvis),tlon(~isvis),tht(~isvis),"or", ...
    "HeightReference","ellipsoid", ...
    "LineWidth",2, ...
    "MarkerSize",1)
```



View the line-of-sight plot from the perspective of the airport. Get the geodetic coordinates of the position that is 900 meters east, 200 meters north, and 100 meters up from the radar location. Then, set the camera position and rotation angles. The green markers appear in view, but the red markers are either completely or partially obstructed by terrain.

```
rdrht = rdralt + egm96geoid(rdrlat,rdrlon);
[camlat,camlon,camht] = enu2geodetic(900,200,100,rdrlat,rdrlon,rdrht,wgs84);
campos(g,camlat,camlon,camht)
camheading(g,-110)
campitch(g,0)
```



Visualize Aircraft Line-of-Sight Visibility Contours on a 2-D Map

The previous sections performed point-to-point line-of-sight analysis and visualization from a radar location to an aircraft trajectory. Now perform point-to-area line-of-sight analysis and visualization from the same radar location over the terrain region. The visualization displays the edge of visibility for an aircraft flying at constant altitude.

Plot Radar Location and Terrain Limits on a 2-D Map

Plot the radar location in a new figure with a topographic 2-D map.

```
figure
geoplot(rdrlat,rdrlon,"co", ...
    "LineWidth",6, ...
    "MarkerSize",3, ...
    "DisplayName","Radar location")
hold on
geobasemap topographic
gx = gca;
gx.InnerPosition = gx.OuterPosition;
```

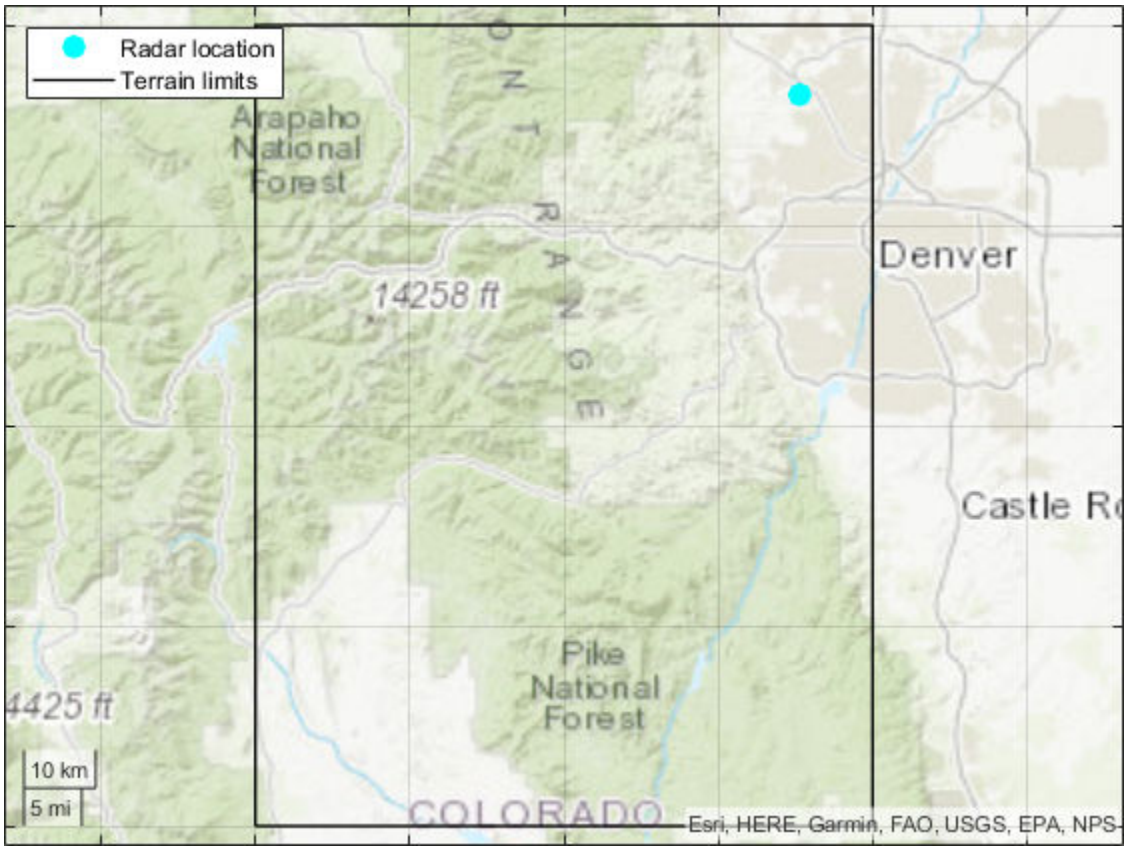
Display the limits of the custom terrain as a rectangle on the map.

```
latmin = latlim(1);
latmax = latlim(2);
lonmin = lonlim(1);
lonmax = lonlim(2);
```

```
geoplot([latmin latmin latmax latmax latmin],[lonmin lonmax lonmax lonmin lonmin], ...
    "LineWidth",1, ...
    "Color","k", ...
    "DisplayName","Terrain limits")
```

Display a legend in the northwest corner.

```
legend("Location","northwest")
```



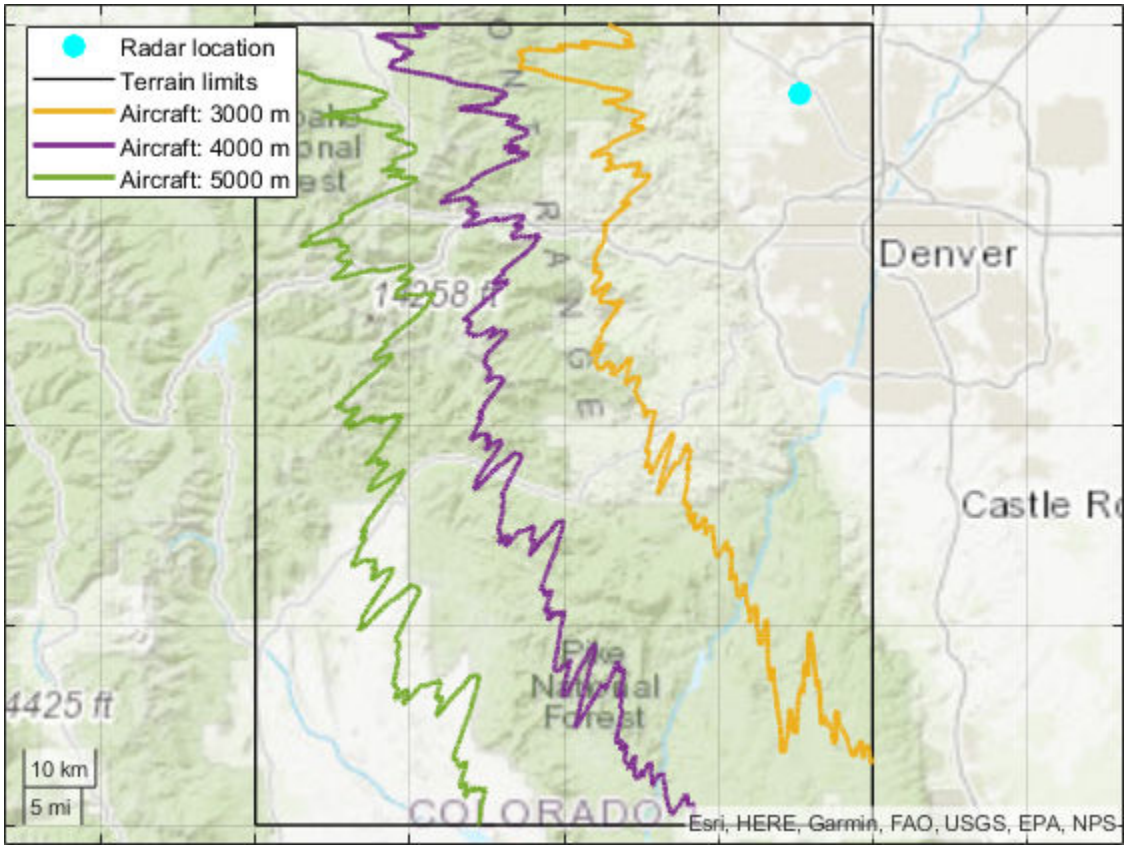
Plot Visibility Contours for Aircraft Flying at Constant Altitude

Specify three altitudes above mean sea level for the aircraft. For each altitude:

- Compute the viewshed using the radar location as the observer. The viewshed defines the area that has line-of-sight visibility.
- Find the edge of aircraft visibility by computing contours from the viewshed data.
- Remove small contour segments.
- Plot the contours on the geographic axes.

```
tgtalts = [3000 4000 5000];

minVertices = 10;
cfig = figure("Visible","off"); % Suppress contour plot using invisible figure
cax = axes("Parent",cfig);
for tgtalt = tgtalts
    vis = viewshed(Zterrain,Rterrain,rdrlat,rdrlon,rdralt,tgtalt,"MSL","MSL");
```

```
    C = contourm(vis,Rterrain,"LevelList",1,"Parent",cax);
    clat = C(2,:);
    clon = C(1,:);

    clats = [];
    clons = [];
    k = 1;
    while k < size(C,2)
        numVertices = clat(k);
        if numVertices > minVertices % Do not plot small segments
            clats = [clats clat(k+1:k+numVertices) NaN]; %#ok<AGROW>
            clons = [clons clon(k+1:k+numVertices) NaN]; %#ok<AGROW>
        end
        k = k + numVertices + 1;
    end

    geoplot(gx,clats,clons,"LineWidth",2, ...
        "DisplayName", "Aircraft: " + string(tgtalt) + " m");
end
```



The contours appear primarily to the west of the radar location over the mountains. The contours do not appear in other directions because visibility is not constrained by the terrain in those directions within the terrain data limits.

If the radar is constrained by line-of-sight visibility, then the contours correspond to radar coverage regions for varying altitude, where the nearest contour to the radar corresponds to radar coverage

for an aircraft flying at 3000 meters and the furthest contour corresponds to radar coverage for an aircraft flying at 5000 meters.

As with `los2`, the `viewshed` function calculates line-of-sight visibility assuming the data is referenced to a spherical Earth, whereas the data is actually referenced to the WGS84 ellipsoid, and as a result there may be minor discrepancies. The line-of-sight calculation also corresponds to optical line-of-sight and does not account for refraction through the atmosphere.

Clean Up

Clean up by closing the geographic globe and removing the imported terrain data.

```
if isvalid(fig)
    close(fig)
end
removeCustomTerrain("southboulder")
```

See Also

Functions

`addCustomTerrain` | `campos` | `geoglobe` | `geoplot3` | `los2` | `readgeoraster`

Objects

`GeographicPostingsReference`

More About

- “Find Ellipsoidal Height from Orthometric Height” on page 3-55
- “Find Geospatial Raster Data” on page 2-73

Visualize UAV Flight Path on 2-D and 3-D Maps

This example visualizes a simulated unmanned aerial vehicle (UAV) flight from the Mauna Loa Baseline Observatory to the top of the Mauna Loa Volcano in Hawaii. First, display the track on geographic axes and a geographic globe. Then, synchronize the view and visualize the flight path by using camera navigation functions. Finally, view the top of the Mauna Loa volcano as a panorama.

Visualize Region of Interest in 2-D

The use of UAVs to track characteristics of changing topology, gasses, and ash clouds around volcanos is becoming an important area of research for scientists [1]. A UAV can travel in regions that are hazardous for a volcanologist. Simulating the flight path of the UAV prior to sending it out on a mission can assist with understanding the topology and terrain. To gain an overview and 2-D perspective of the region, view the locations of the Mauna Loa Baseline Observatory and the Mauna Loa Volcano in a geographic axes.

Get Coordinates of Mauna Loa Baseline Observatory

Specify the coordinates of the Mauna Loa Baseline Observatory [2]. The height of the observatory is in meters above mean sea level (MSL).

```
obslat = 19.5362;
obslon = -155.5763;
obsH = 3397.00;
```

Get Coordinates of Mauna Loa Volcano

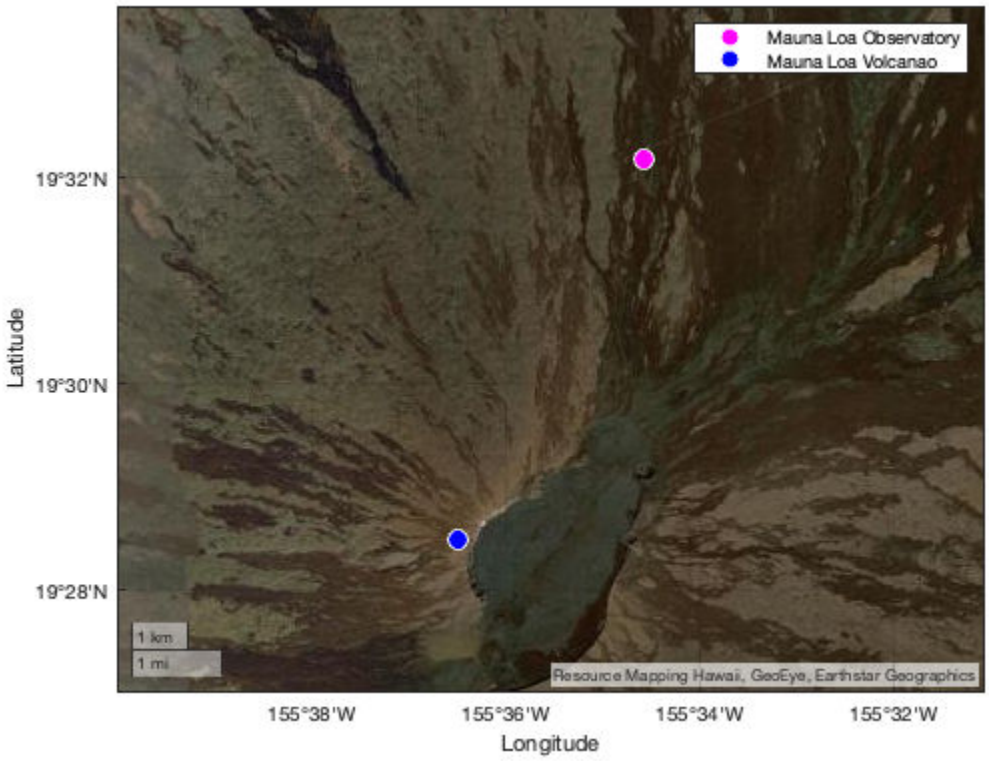
Specify the coordinates of the top of Mauna Loa [3]. The height of the volcano is orthometric and is in meters.

```
mllat = 19.475;
mllon = -155.608;
mlH = 4169;
```

View Mauna Loa Baseline Observatory and Mauna Loa Volcano in 2-D

For a 2-D perspective of the region, use `geoaxes` and `geoplot` to plot the location of the observatory and the top of the volcano.

```
figure
geoaxes("Basemap","satellite","ZoomLevel",12);
hold("on")
geoplot(obslat,obslon,"ow","MarkerSize",10,"MarkerFaceColor","magenta", ...
    "DisplayName","Mauna Loa Observatory");
geoplot(mllat,mllon,"ow","MarkerSize",10,"MarkerFaceColor","blue", ...
    "DisplayName","Mauna Loa Volcanao");
legend
```



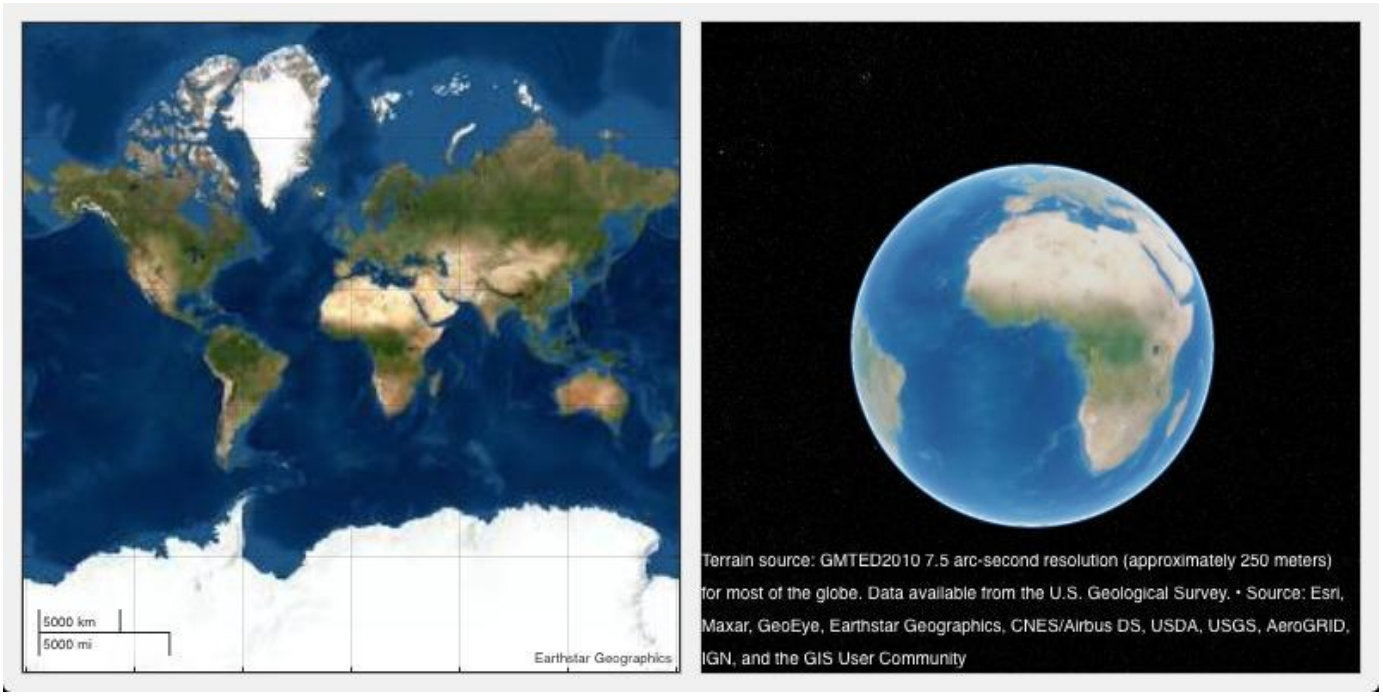
Synchronize View of Mauna Loa Baseline Observatory in 2-D and 3-D

Use the geographic axes to view the observatory in 2-D and the geographic globe to view the observatory in 3-D.

Create Geographic Axes and Geographic Globe in Same Figure

Set up a 2-D and 3-D map display by creating geographic axes and a geographic globe in the same UI figure. To view more of the 2-D map, set the `InnerPosition` of the geographic axes to its `OuterPosition`. To view both map displays with the same basemap, set the basemap of the geographic axes to "satellite".

```
figpos = [1000 500 800 400];
uif = uifigure("Position",figpos);
ug = uigridlayout(uif,[1,2]);
p1 = uipanel(ug);
p2 = uipanel(ug);
gx = geoaxes(p1,"Basemap","satellite");
gg = geoglobe(p2);
gx.InnerPosition = gx.OuterPosition;
gg.Position = [0 0 1 1];
```



View Observatory in 2-D

View the observatory from 200 meters above the terrain. Control the view of the geographic axes by changing its map center and zoom level. You can synchronize the view of the geographic axes with the view of the geographic globe by converting the camera height of the globe to a zoom level for the axes. Calculate an approximate zoom level from terrain height by using the `heightToZoomLevel` local function.

```
heightAboveTerrain = 200;
gx.MapCenter = [obslat, obslon];
zoomLevel = heightToZoomLevel(heightAboveTerrain, obslat);
gx.ZoomLevel = zoomLevel;
```

View Observatory in 3-D

Control the view of the geographic globe by changing the position of the camera. The `campos` function requires you to specify ellipsoidal height (relative to the WGS84 ellipsoid) instead of orthometric height (relative to mean sea level). Convert the height of the observatory to ellipsoidal height. All heights are in meters.

```
N = egm96geoid(obslat, obslon);
obsh = obsH + N;
ellipsoidalHeight = obsh + heightAboveTerrain;
campos(gg,obslat,obslon,ellipsoidalHeight)
drawnow
```

Import Flight Track Data and Calculate Heading and 3-D Distance

Import the simulated flight track from the Mauna Loa Baseline Observatory to the top of the Mauna Loa volcano. The file contains the latitudes, longitudes, and altitudes of the UAV path, referenced to mean sea level.

```
trk = gpxread("sample_uavtrack.gpx");
tlat = trk.Latitude;
tlon = trk.Longitude;
talt = trk.Elevation;
```

Calculate Flight Headings

Calculate the UAV heading at each track point using the `azimuth` function.

```
wgs84 = wgs84Ellipsoid;
theading = azimuth(tlat(1:end-1),tlon(1:end-1),tlat(2:end),tlon(2:end),wgs84);
theading = [theading(1);theading(:)];
```

Calculate 3-D Distances

Calculate the cumulative distance for the UAV flight track. The `distance` function does not take into account changes in elevation or altitude. In order to calculate the distance the UAV moves from point to point in 3-D, you need to work in geocentric Cartesian coordinates (X, Y, Z). Compute the point-to-point offset components (in meters) using the `ecefOffset` function. The altitude data of the UAV flight is referenced to the mean sea level. To use the `ecefOffset` function, the heights must be referenced to the ellipsoid. Convert the orthometric heights of the flight track to ellipsoidal height (relative to the WGS84 ellipsoid). All heights are in meters.

```
N = egm96geoid(tlat,tlon);
h = talt + N;
```

Compute distance offsets.

```
lat1 = tlat(1:end-1);
lat2 = tlat(2:end);
lon1 = tlon(1:end-1);
lon2 = tlon(2:end);
h1 = h(1:end-1);
h2 = h(2:end);
[dx,dy,dz] = ecefOffset(wgs84,lat1,lon1,h1,lat2,lon2,h2);
```

Calculate the Euclidean distance between each pair of adjacent points using the `hypot` function. The distance is in meters.

```
distanceIncrementIn3D = hypot(hypot(dx, dy), dz);
```

Compute cumulative distance in 3-D and the total distance in meters.

```
cumulativeDistanceIn3D = cumsum(distanceIncrementIn3D);
totalDistanceIn3D = sum(distanceIncrementIn3D);
fprintf("Total UAV track distance is %f meters.\n",totalDistanceIn3D)
```

```
Total UAV track distance is 8931.072120 meters.
```

Assign a variable for the cumulative distance to be used for plotting the animation.

```
tdist = [0 cumulativeDistanceIn3D];
```

Plot Flight Line from Mauna Loa Baseline Observatory to top of Mauna Loa Volcano

Plot the simulated flight line from the Mauna Loa Baseline Observatory to the top of the Mauna Loa volcano.

Plot the flight line. By default, the geographic globe places the line at the center of the display. Hold the geographic axes to preserve the basemap. Its location will not change because you have previously set the `MapCenter` and `ZoomLevel`.

```
geoplot3(gg,tlat,tlon,talt,"c","LineWidth",2,"HeightReference","geoid")
hold(gx,"on")
ptrack = geoplot(gx,tlat,tlon,"c","LineWidth",2);
```

Set the map center and zoom level to be consistent with the 3-D view by converting the camera height for the globe to the zoom level for the axes.

```
[clat,clon,cheight] = campos(gg);
gx.MapCenter = [clat,clon];
gx.ZoomLevel = heightToZoomLevel(cheight, clat);
drawnow
```

Set Initial View from Mauna Loa Baseline Observatory to Top of Mauna Loa Volcano

View the flight line from the start position by setting the camera position to the first coordinate of the track. For a better perspective, set the camera height to 75 meters about the height of the track. View straight down to the observatory by setting the camera pitch to -90. View the track by setting the heading to the third element of the calculated heading array since the first two points of the track are the same location and the calculated heading for those locations is 0.

```
campos(gg,tlat(1),tlon(1))
camheight(gg,talt(1) + 75)
campitch(gg,-90)
camheading(gg,theading(3))
```

Show the location of the UAV in the 2-D map, and the start and end locations of the flight track with markers. Create a legend for the UAV track and markers.

```
marker = geoplot(gx,tlat(1),tlon(1),"ow","MarkerSize",10,"MarkerFaceColor","k");
mstart = geoplot(gx,tlat(1),tlon(1),"ow","MarkerSize",10,"MarkerFaceColor","magenta");
mend = geoplot(gx,tlat(end),tlon(end),"ow","MarkerSize",10,"MarkerFaceColor","blue");

marker.DisplayName = "Current Location";
mstart.DisplayName = "Start Location";
mend.DisplayName = "End Location";
ptrack.DisplayName = "UAV Track";
legend(gx)
```

View the topology of the region by changing the basemap.

```
gx.Basemap = "topographic";
```

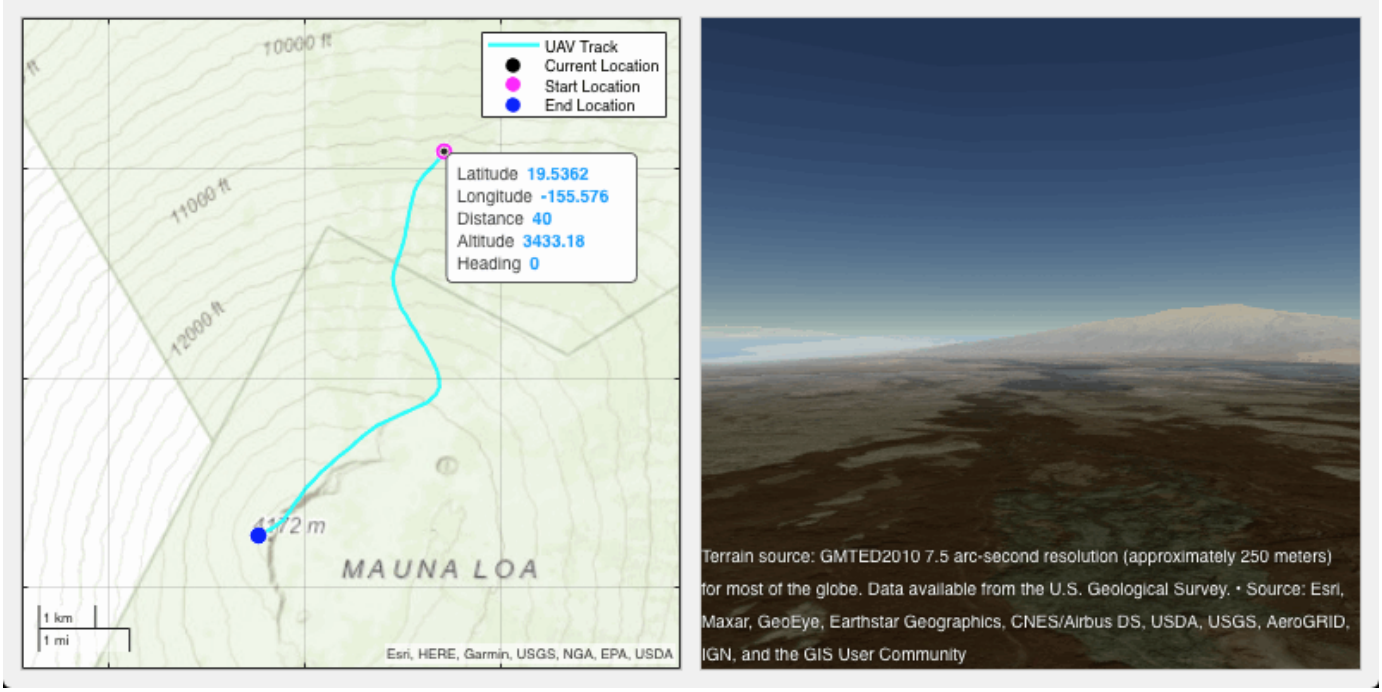
View the coordinate location, altitude, and heading of the UAV by using a custom data tip that corresponds with the location of the UAV. Include the distance from the observatory.

```
dt = datatip(ptrack,"DataIndex",1,"Location","southeast");
dtrow = dataTipTextRow("Distance",tdist);
dtrow(end+1) = dataTipTextRow("Altitude",talt);
dtrow(end+1) = dataTipTextRow("Heading",theading);
ptrack.DataTipTemplate.DataTipRows(end+1:end+3) = dtrow;
```

Fly from Mauna Loa Baseline Observatory to Top of Mauna Loa Volcano

Animate a flight from the Mauna Loa Baseline Observatory to the top of the Mauna Loa volcano. View the location of the UAV on the 2-D map by animating a marker and data tip. Animate the 3-D flight by setting the camera position. For a better view of the UAV track, set the camera height to 100 meters above the flight track. Update the camera pitch value for a better view of the flight track as the UAV navigates to the top of the volcano. To view the current location, altitude, and heading of the UAV, update the data tip with the current index.

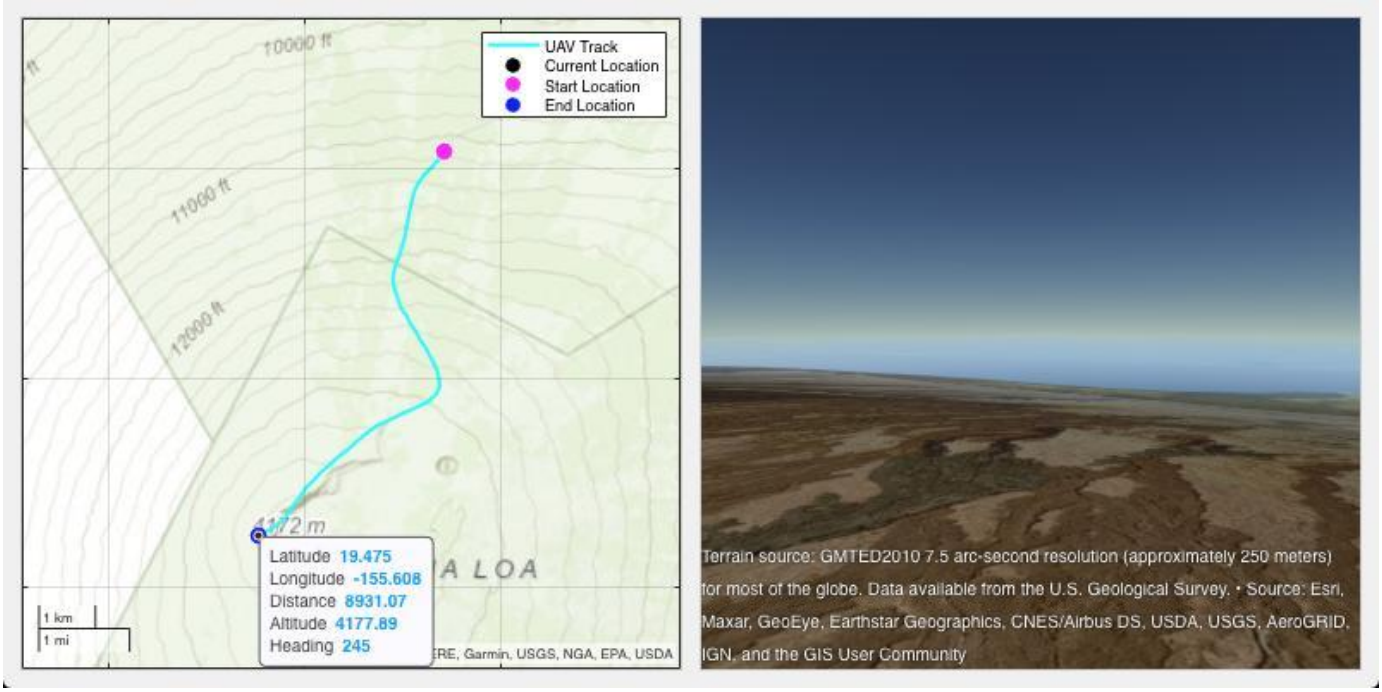
```
pitch = -2.7689;
campitch(gg,pitch)

for k = 2:(length(tlat)-1)
    campos(gg,tlat(k),tlon(k))
    camheight(gg,talt(k)+100)
    camheading(gg,theading(k))

    set(marker,"LatitudeData",tlat(k),"LongitudeData",tlon(k));
    dt.DataIndex = k;

    drawnow
    pause(.25)
end

campos(gg,tlat(end),tlon(end),talt(end)+100)
dt.DataIndex = length(tlat);
```



View a 360-Degree Panorama from Top of Mauna Loa Volcano

View a 360-degree panorama from the top of Mauna Loa by rotating the camera heading 360 degrees. Rotate clockwise with a step size of 5-degrees and start at the next 5 degree step. Update the heading data tip.

```
initialHeading = camheading(gg);
increment = 5;
initialHeading = initialHeading + (increment - mod(initialHeading,increment));

filename = 'panoramic.gif';
for degree = initialHeading:increment:initialHeading+360
    heading = mod(degree,360);
    ptrack.DataTipTemplate.DataTipRows(end).Value(dt.DataIndex) = heading;
    camheading(gg,heading);
    drawnow
end
```



Local Functions

Convert Height (in meters above WGS84 ellipsoid) to Zoom Level

```
function zoomLevel = heightToZoomLevel(height, lat)
    earthCircumference = 2 * pi * 6378137;
    zoomLevel = log2((earthCircumference *cosd(lat)) / height) + 1;
    zoomLevel = max(0, zoomLevel);
    zoomLevel = min(19, zoomLevel);
end
```

References

[1] Williams, Sarah C. P. “Studying Volcanic Eruptions with Aerial Drones.” *Proceedings of the National Academy of Sciences of the United States of America* 110, no. 27 (July 2, 2013): 10881. https://doi.org/10.1073/pnas.1309922110.

[2] NOAA. “Mauna Loa Baseline Observatory.” Global Monitoring Laboratory. Accessed June 16, 2020. https://www.esrl.noaa.gov/gmd/obop/mlo/.

[3] USGS. “Mauna Loa.” Hawaiian Volcano Observatory. Accessed June 16, 2020. https://www.usgs.gov/volcanoes/mauna-loa.

See Also

Functions

`azimuth` | `camheading` | `campitch` | `campos` | `camroll` | `egm96geoid`

Objects

`GeographicGlobe`

Properties
GeographicAxes Properties

More About

- “Find Ellipsoidal Height from Orthometric Height” on page 3-55

Customizing and Printing Maps

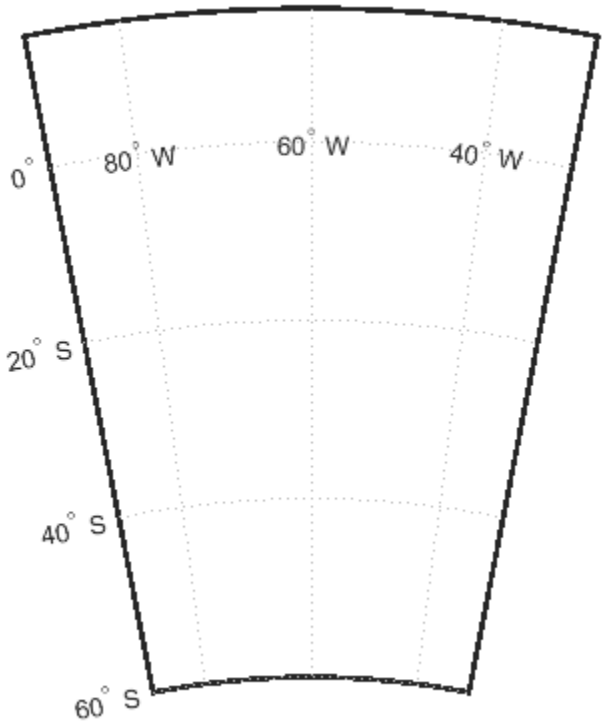
- “Inset Maps” on page 6-2
- “Graphic Scales” on page 6-9
- “North Arrows” on page 6-15
- “Thematic Maps” on page 6-18
- “Create Choropleth Map of Population Density” on page 6-21
- “Contour Colormaps” on page 6-24
- “Colormaps for Political Maps” on page 6-27
- “Scale Maps for Printing” on page 6-31

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. In this 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.

Begin by defining a map frame for South America using `worldmap`.

```
figure
h1 = worldmap('south america');
```

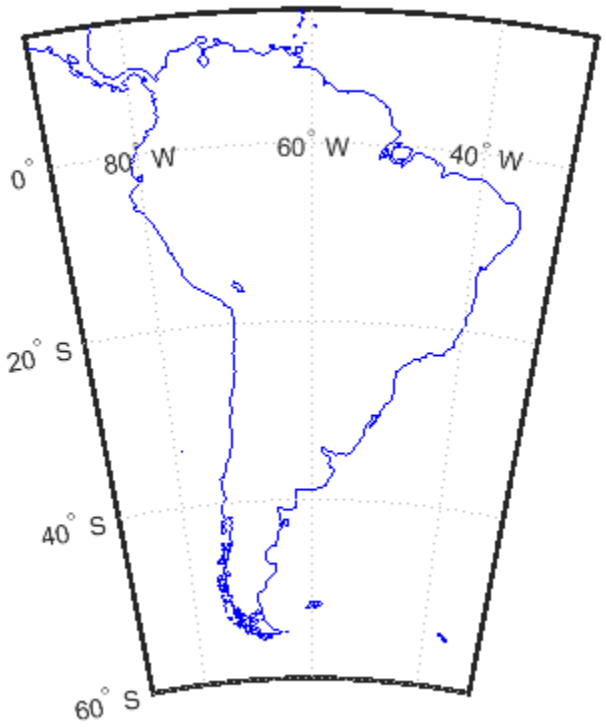


Use `shaperead` to read the world land areas polygon shapefile.

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

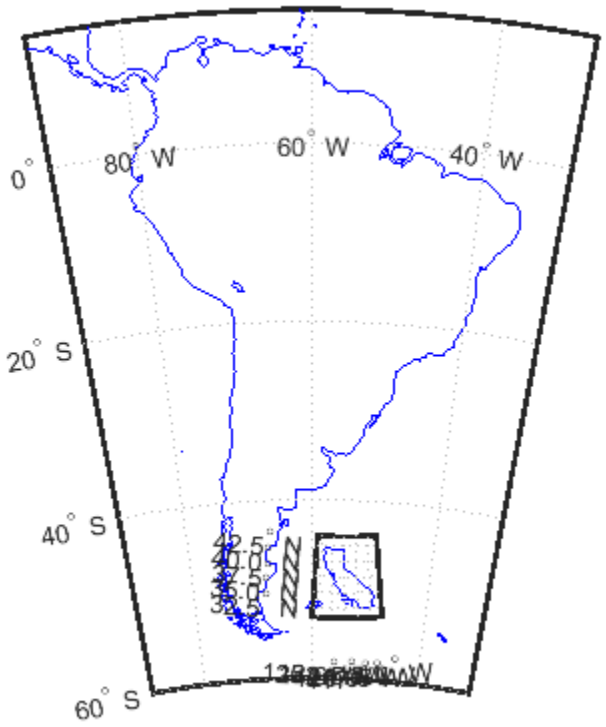
Display the data in the map axes.

```
geoshow([land.Lat],[land.Lon])
setm(h1,'FFaceColor','w') % set the frame fill to white
```



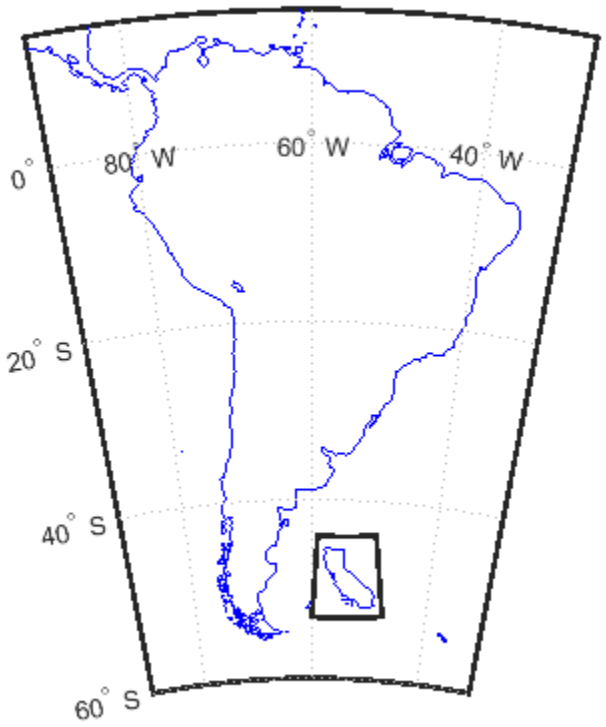
Place axes for an inset in the lower middle of the map frame, and project a line map of California:

```
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])
```



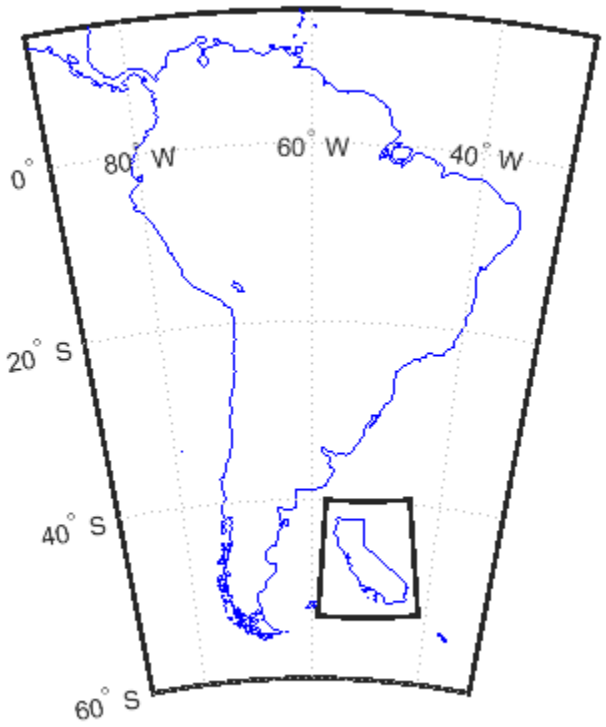
Set the frame fill color and set the labels.

```
setm(h2,'FFaceColor','w')
mlabel; plabel; gridm % toggle off
```



Make the scale of the inset axes, `h2` (California), match the scale of the original axes, `h1` (South America). Hide the map border.

```
axesscale(h1)
```




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

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

Find out what map projections are used, and then make South America's projection the same as California's.

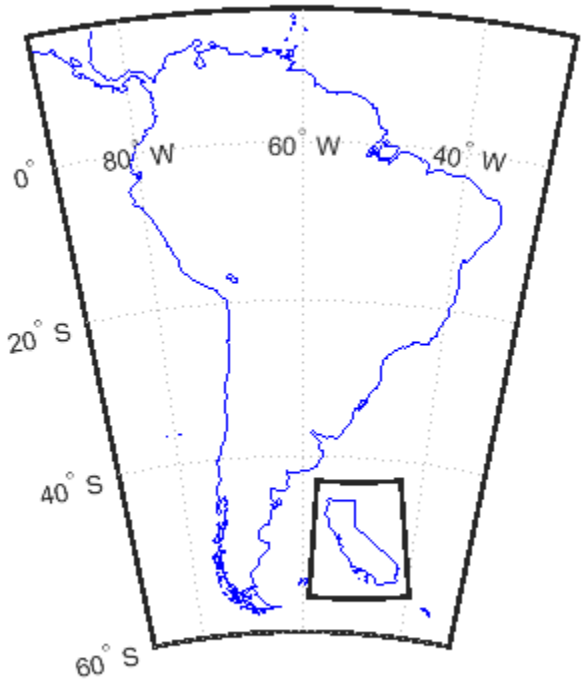
```
getm(h1, 'mapprojection')
```

```
ans =
'eqdconic'
```

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

```
ans =
'lambert'
```

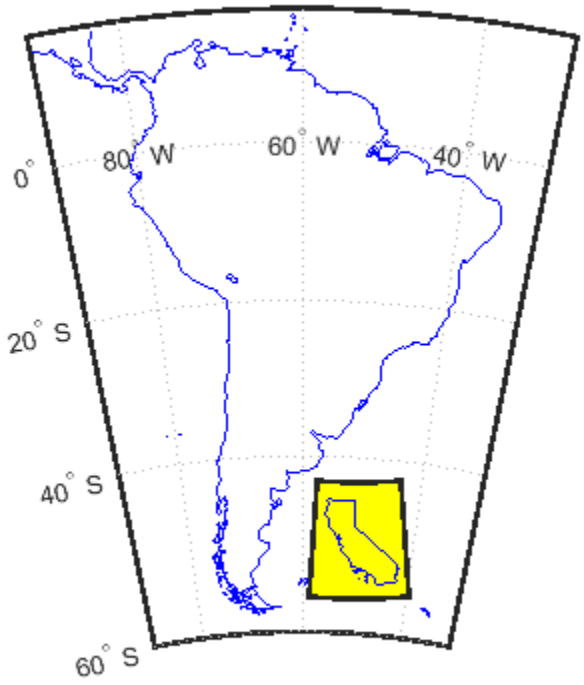
```
setm(h1, 'mapprojection', getm(h2, 'mapprojection'))
```



Note that the parameters for South America defaulted properly (those appropriate for California were not used).

Finally, experiment with changing properties of the inset, such as its color.

```
setm(h2, 'ffacecolor', 'y')
```


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

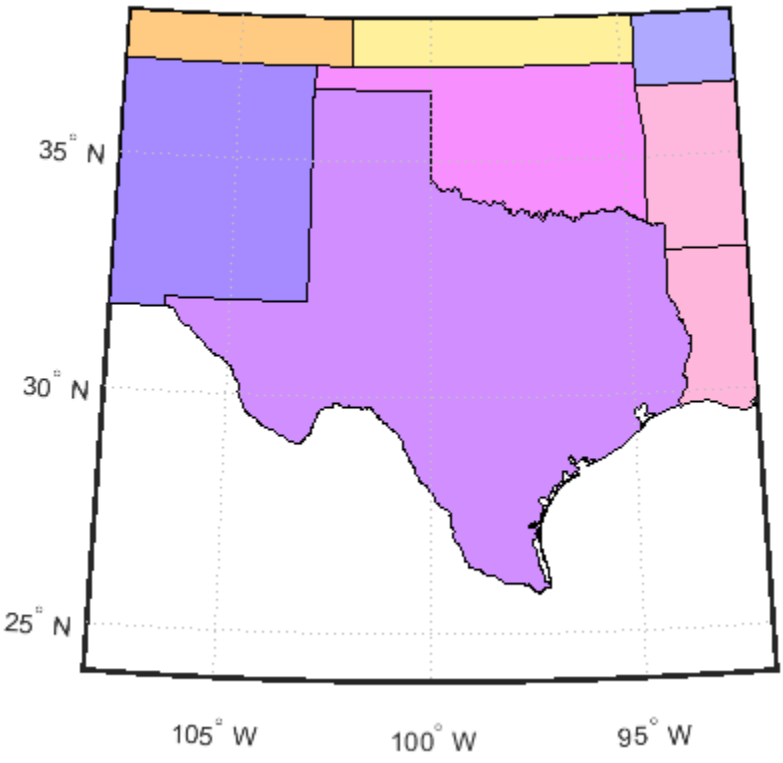
Graphic Scales

This example shows how to add graphic scales to maps and how to modify the display properties of 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.

Use `usamap` to plot a map of Texas and surrounding states as filled polygons.

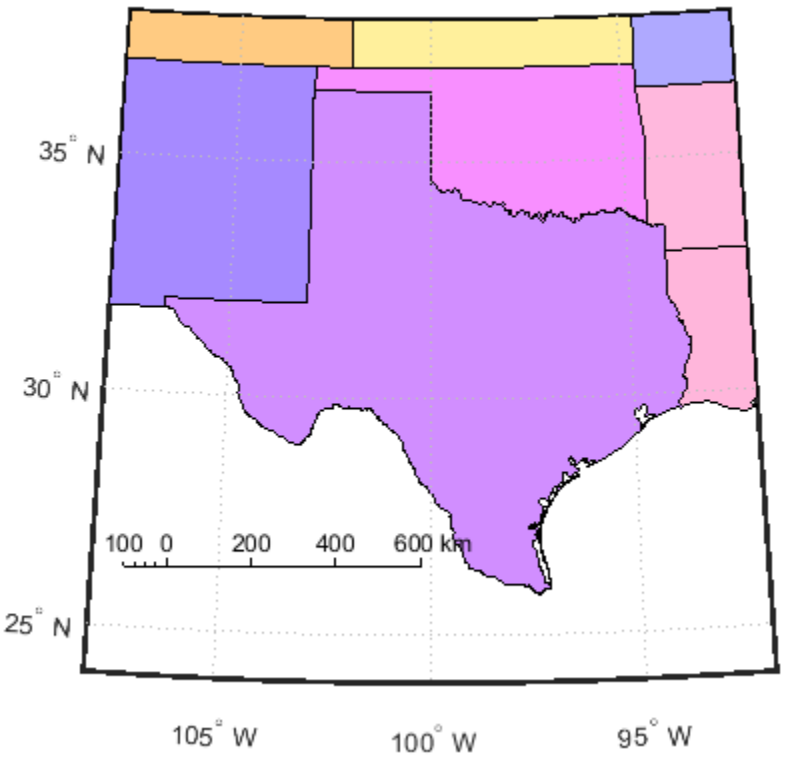
```
states = shaperead('usastatehi.shp', 'UseGeoCoords', true);
figure
usamap('Texas')
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], ...
    'FaceColor', polcmap(numel(states))});
geoshow(states,'DisplayType', 'polygon', ...
    'SymbolSpec', faceColors)
```



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

Add a default graphic scale and then move it to a new location.

```
scaleruler on
setm(handlem('scaleruler1'), ...
    'XLoc',-6.2e5,'YLoc',3.1e6, ...
    'MajorTick',0:200:600)
```



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.

Return the `scaleruler` as a `Group` object using the `handlem` function and inspect its properties using `getm`.

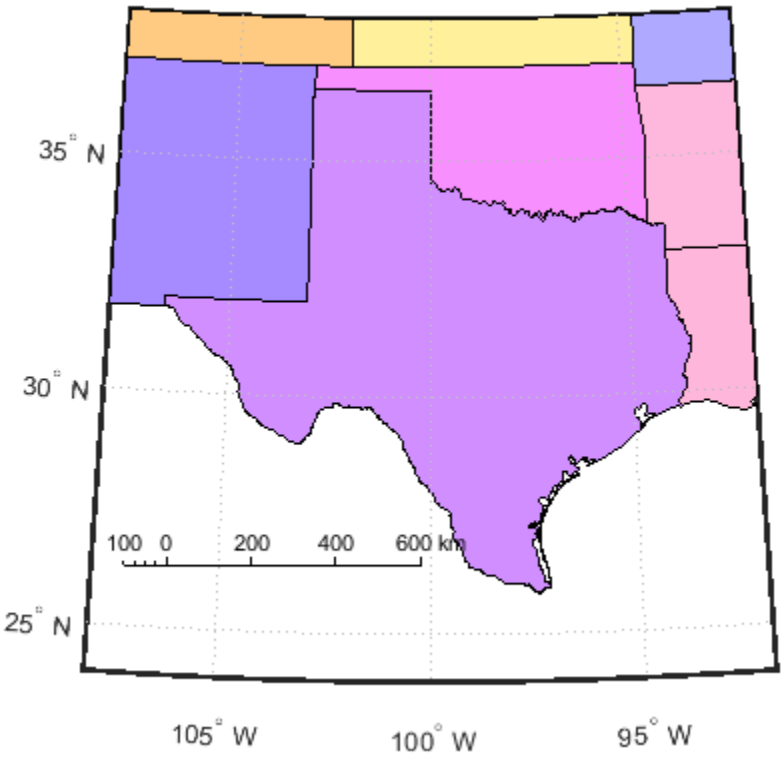
```
s = handlem('scaleruler');
getm(s)
```

```
ans = struct with fields:
            Azimuth: 0
           Children: []
              Color: [0 0 0]
          FontAngle: 'normal'
           FontName: 'Helvetica'
           FontSize: 9
          FontUnits: 'points'
         FontWeight: 'normal'
              Label: ''
                Lat: 29.6479
```

```
            Long: -101.7263
       LineWidth: 0.5000
       MajorTick: [0 200 400 600]
    MajorTickLabel: {4x1 cell}
    MajorTickLength: 20
       MinorTick: [0 25 50 75 100]
    MinorTickLabel: '100'
    MinorTickLength: 12.5000
          Radius: 'earth'
      RulerStyle: 'ruler'
         TickDir: 'up'
        TickMode: 'manual'
           Units: 'km'
            XLoc: -620000
            YLoc: 3100000
            ZLoc: []
```

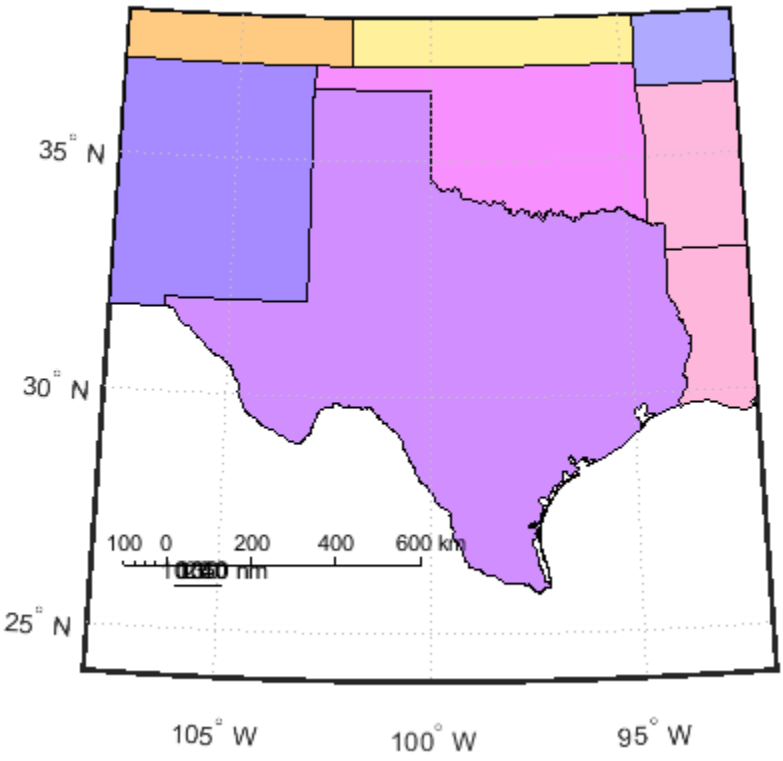
Change the font size of the `scaleruler` to 8 points.

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



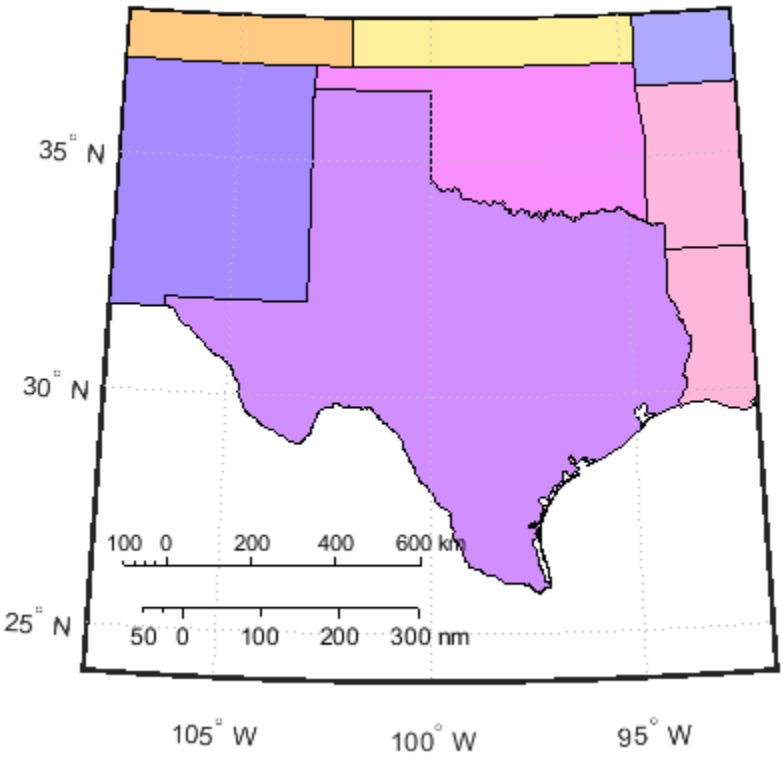
Place a second graphic scale, this one in units of nautical miles.

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



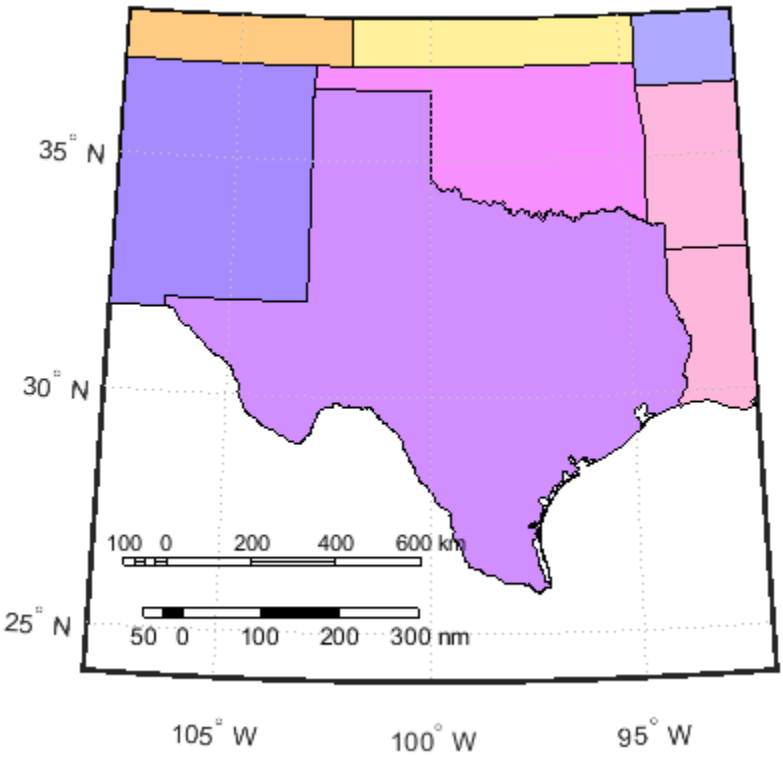
Modify the tick properties of the second graphic scale.

```
setm(handlem('scaleruler2'), 'YLoc', 3.0e6, ...
    'MajorTick', 0:100:300,...
    'MinorTick', 0:25:50, 'TickDir', 'down', ...
    'MajorTickLength', km2nm(25),...
    'MinorTickLength', km2nm(12.5))
```



Experiment with the two other ruler styles available.

```
setm(handlem('scaleruler1'), 'RulerStyle', 'lines')
setm(handlem('scaleruler2'), 'RulerStyle', 'patches')
```

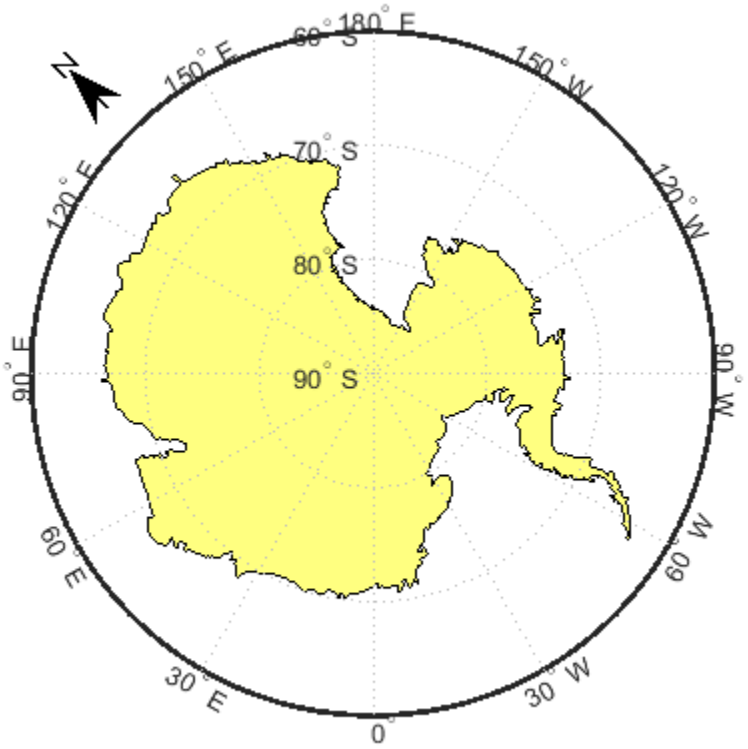

35° N
30° N
25° N
100 0 200 400 600 km
50 0 100 200 300 nm
105° W
100° W
95° W

North Arrows

The north arrow element provides the orientation of a map by pointing to the geographic North Pole. You can use the `northarrow` function to display a symbol indicating the direction due north on the current map. The north arrow symbol can be repositioned by clicking and dragging its icon. The orientation of the north arrow is computed, and does not need manual adjustment no matter where you move the symbol. **Ctrl**+clicking the icon creates an input dialog box with which you can change the location of the north arrow:

Create a map centered at the South Pole. 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);
```



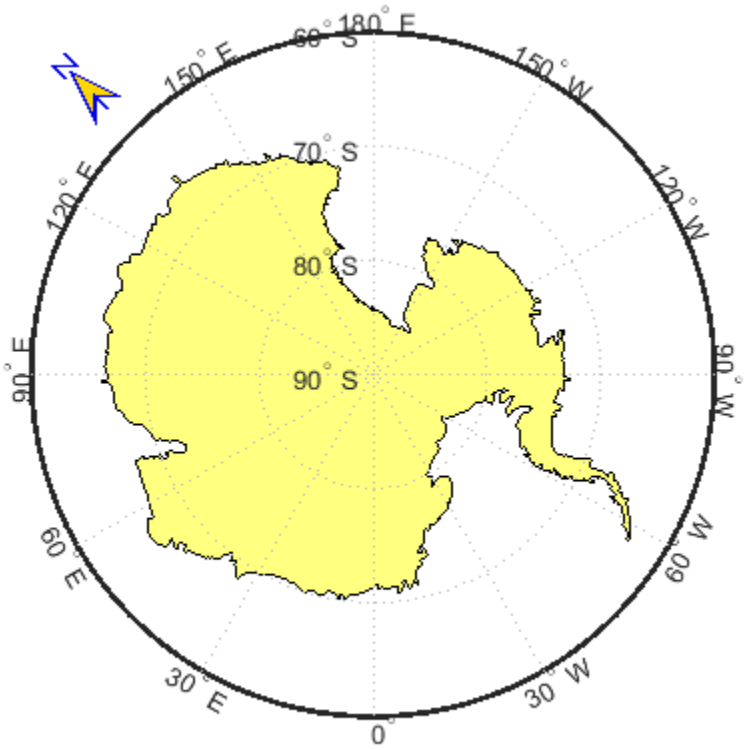
Click and drag the north arrow symbol to another corner of the map. Note that it always points to the North Pole.

Drag the north arrow back to the top left corner.

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**.

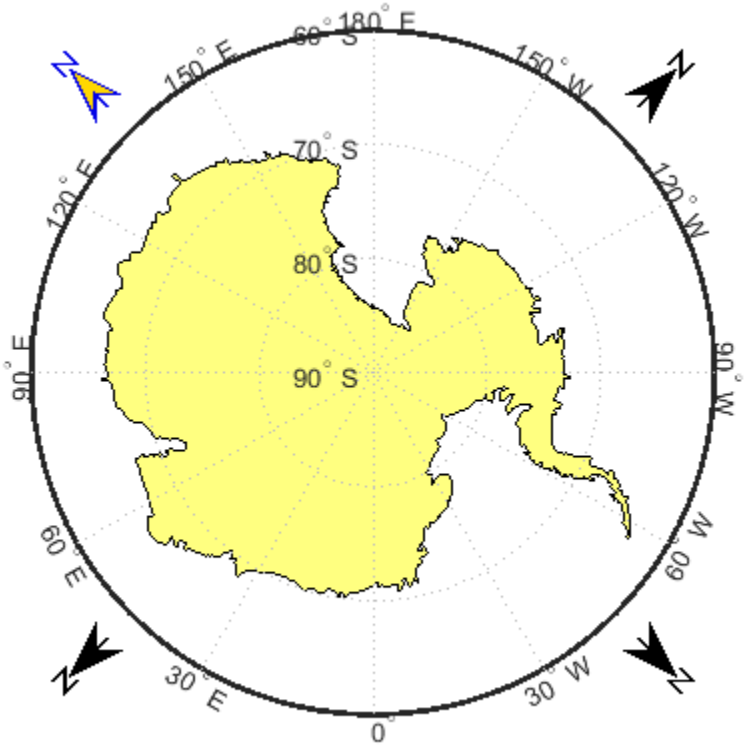
Also set some north arrow properties manually, just to get a feel for them.

```
h = handlem('NorthArrow');
set(h, 'FaceColor', [1.000 0.8431 0.0000],...
   'EdgeColor', [0.0100 0.0100 0.9000])
```



Make three more north arrows, to show that from the South Pole, every direction is north. 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.

```
northarrow('latitude',-57,'longitude', 45);
northarrow('latitude',-57,'longitude',225);
northarrow('latitude',-57,'longitude',315);
```


180° E
150° E
150° W
120° E
120° W
90° E
90° W
60° E
60° W
30° E
30° W
0
60° S
70° S
80° S
90° S

Thematic Maps

Most published and online maps fall into four categories:

- Navigation maps, including topographic maps and nautical and aeronautical charts
- Geophysical maps, that show the structure and dynamics of earth, oceans and atmosphere
- Location maps, that depict the locations and names of physical features
- Thematic maps, that portray attribute data about locations and features

Although online maps often combine these categories in new and unexpected ways, published maps and atlases tend to respect them.

Thematic maps tend to be more highly stylized than other types of maps and frequently omit locational information such as place names, physical features, coordinate grids, and map scales. This is because rather than showing physical features on the ground, such as shorelines, roads, settlements, topography, and vegetation, a thematic map displays quantified facts (a "theme"), such as statistics for a region or sets of regions. Examples include the locations of traffic accidents in a city, or election results by state. Thematic maps have a wide vocabulary of cartographic symbols, such as point symbols, dot distributions, "quiver" vectors, isolines, colored zones, raised prisms, and continuous 3-D surfaces. Mapping Toolbox functions, listed in the following table, can generate most of these types of map symbology.

Function	Used For
`quiverm`	Plots directed vectors in 2-D from specified latitudes and longitudes with lengths also specified as latitudes and longitudes
`quiver3m`	Plots directed vectors in 3-D from specified latitudes, longitudes, and altitudes with lengths also specified as latitudes and longitudes and altitudes
`scatterm`	Draws fixed or proportional symbol maps for each point in a vector with specified marker symbol. Similar maps can be generated using `geoshow` and `mapshow` using appropriate symbol specifications ("symbolspecs").
`stem3m`	Projects a 3-D stem plot map on the current map axes. The `stem3m` function allows you to display geographic bar graphs.
`tissot`	Calculates and displays Tissot Indicatrices, which graphically portray the shape distortions of any map projection.

Choropleth Maps

The most familiar form of thematic map is probably the choropleth map (from the Greek *choros*, for place, and *plethos*, for magnitude). Choropleth maps use colors or patterns to represent attributes associated with certain geographic regions. For example, the global distribution of malaria-carrying mosquitoes can be illustrated in a choropleth map, with the habitat of each mosquito represented by a different color. In this example, colors are used to represent nominal data; the categories of mosquitoes have no inherent ranking. If the data is ordinal, rather than nominal, the map may contain a colorbar with shades of colors representing the ranking. For instance, a map of crime rates in different areas could show high crime areas in red, lower crime areas in pink, and lowest crime areas in white. See “Create Choropleth Map of Population Density” on page 6-21 for an example of creating a choropleth map where the color of each location indicates the population density.

To create a choropleth map with the Mapping Toolbox:

1 Start with a geographic data structure on page 2-22.

2 Create a symbolspec to map attribute values to face colors.

3 Apply either `geoshow` or `mapshow`, depending on whether you are working with latitude-longitude or pre-projected map coordinates.

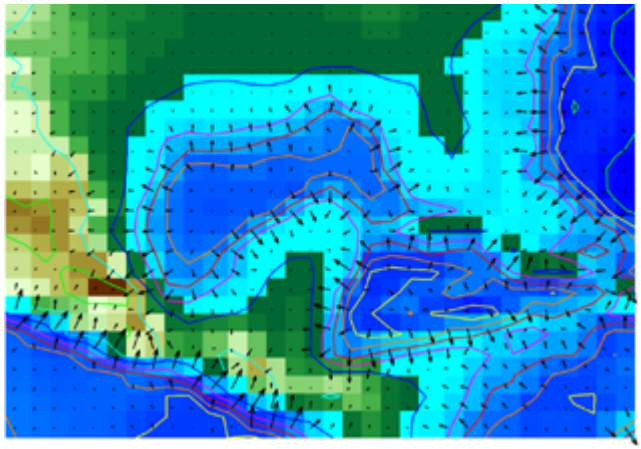
Stem Maps

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 of the Gulf of Mexico. 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*.

Create Choropleth Map of Population Density

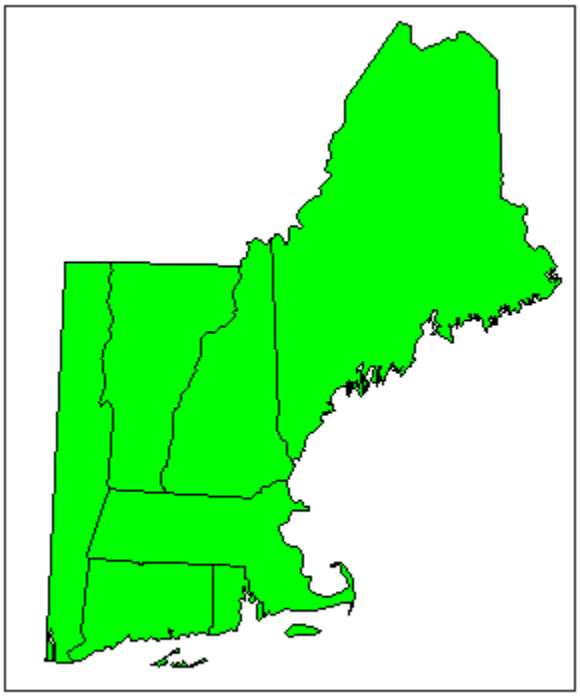
This example shows how to create a choropleth map of population density for the six New England states in the year 2000.

Import low-resolution U.S. state boundary polygons, setting the map limits for the New England region.

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

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

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

```
axesm('MapProjection', 'eqaconic', 'MapParallels', [],...
  'MapLatLimit', MapLatLimit, 'MapLonLimit', MapLonLimit,...
  'GLineStyle', '-')
geoshow(NEstates, 'DisplayType', 'polygon', 'FaceColor','green')
```



Identify the maximum population density for New England states.

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

```
maxdensity = 1.1345e+03
```

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

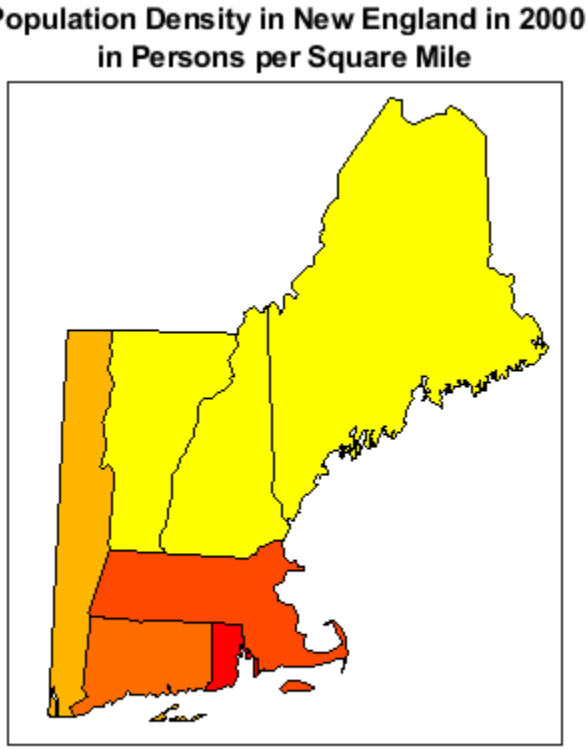
```
fall = flipud(autumn(numel(NEstates)));
```

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

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

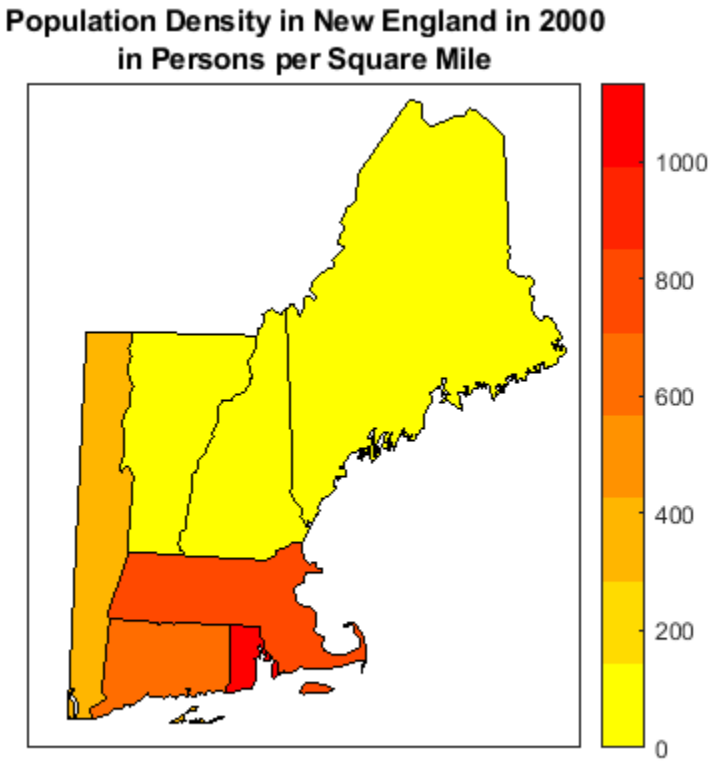
Display the map.

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



Add a colorbar. You can also experiment with other colormaps.

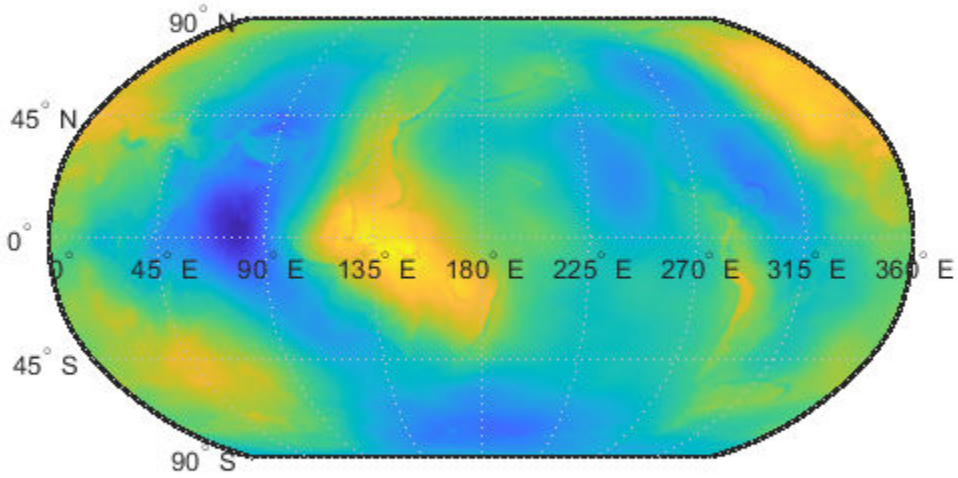
```
caxis([0 maxdensity])
colormap(fall)
colorbar
```


Population Density in New England in 2000
in Persons per Square Mile
1000
800
600
400
200
0

Contour Colormaps

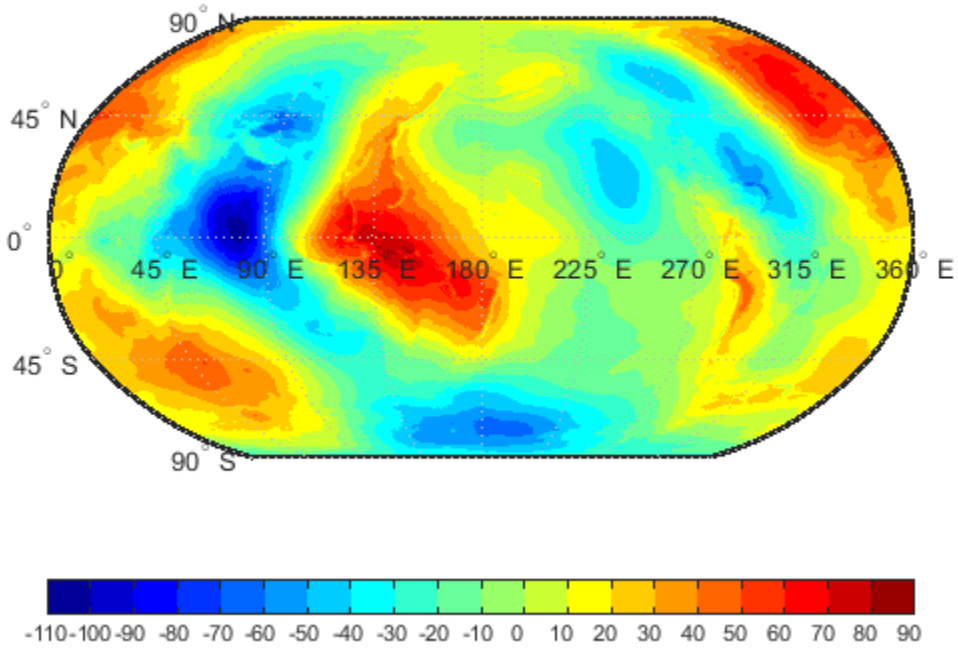
Use colormaps to display surfaces as contour maps for data types other than terrain, such as geoid heights. First, get geoid heights and a geographic postings reference object from the EGM96 geoid model. Display the geoid heights using the default colormap.

```
[N,R] = egm96geoid;
worldmap(N,R)
geoshow(N,R,'DisplayType','surface')
```



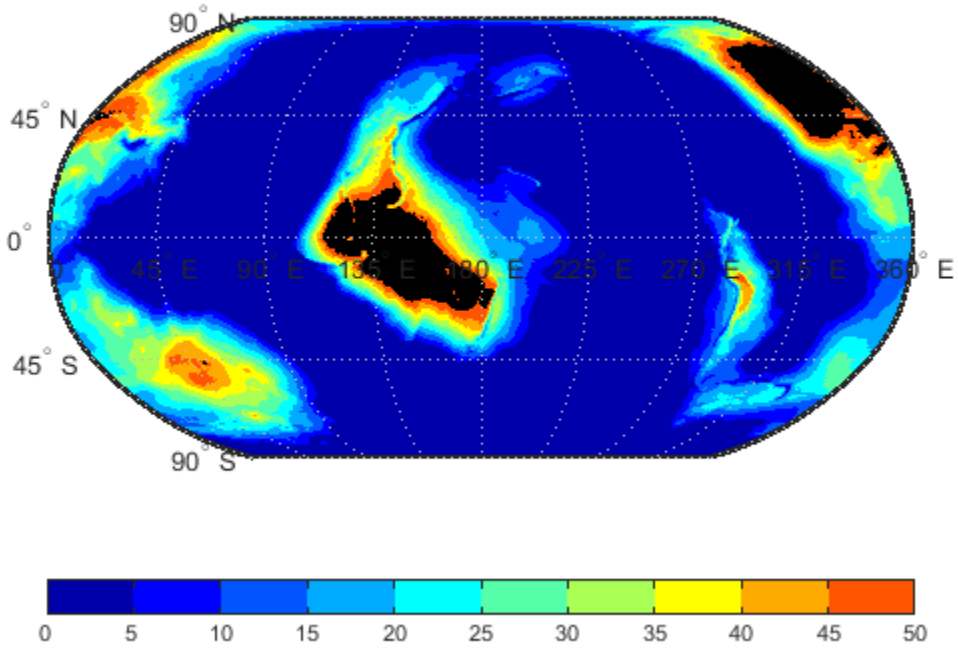
Then, display a colorbar using the `'jet'` colormap. Specify the contour interval as 10 meters.

```
contourcmap('jet',10,'Colorbar','on','Location','horizontal')
```



Display a restricted value range by specifying a vector of evenly spaced values.

```
range = 0:5:50;
contourcmap('jet',range,'Colorbar','on','Location','horizontal')
```


90° N
45° N
0°
45° S
90° S
45° E
90° E
135° E
180° E
225° E
270° E
315° E
360° E
0
5
10
15
20
25
30
35
40
45
50

Colormaps for Political Maps

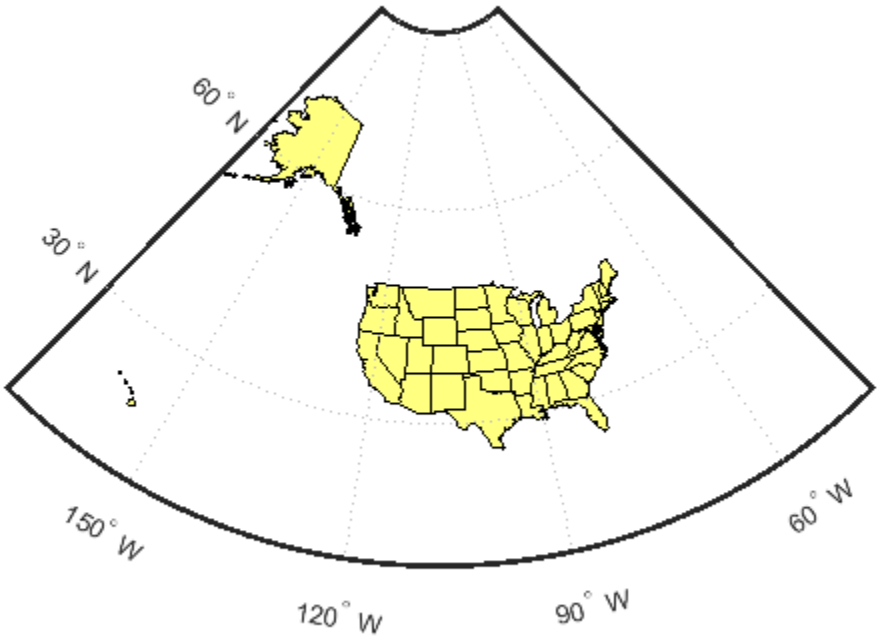
Political maps typically use muted, contrasting colors that make it easy to distinguish one country from its neighbors. You can create colormaps of this kind using the `polcmap` function. The `polcmap` function creates a colormap with randomly selected colors of all hues. Since the colors are random, if you don't like the result, execute `polcmap` again to generate a different colormap.

`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`.

Explore Colormaps for Political Maps

Display the `usastatelo` data set as patches, setting up the map with `worldmap` and plotting it with `geoshow`. Note that the default face color is yellow.

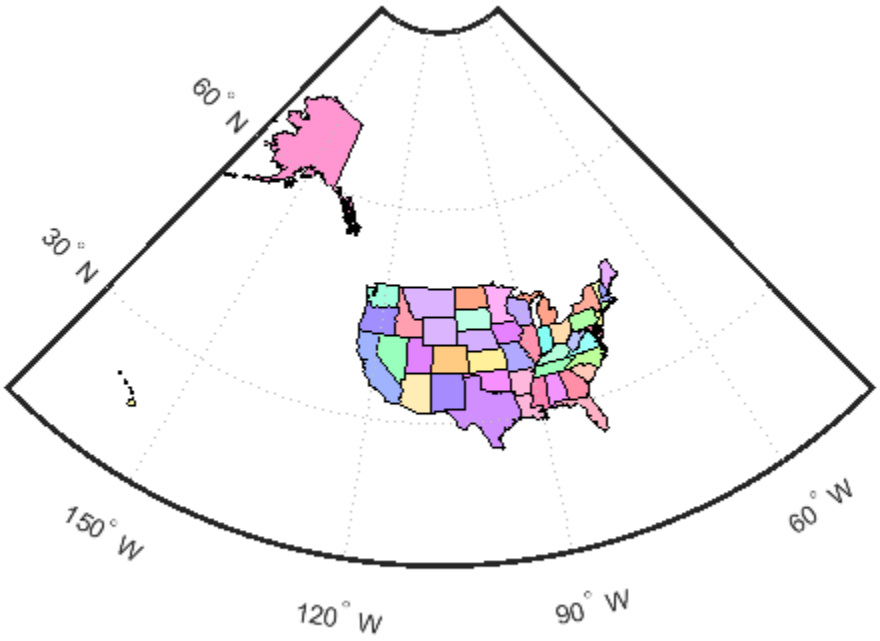
```
figure
worldmap na
states = shaperead('usastatelo', 'UseGeoCoords', true);
geoshow(states)
```



Use `polcmap` to populate color definitions to a symbolspec to recolor the patches randomly.

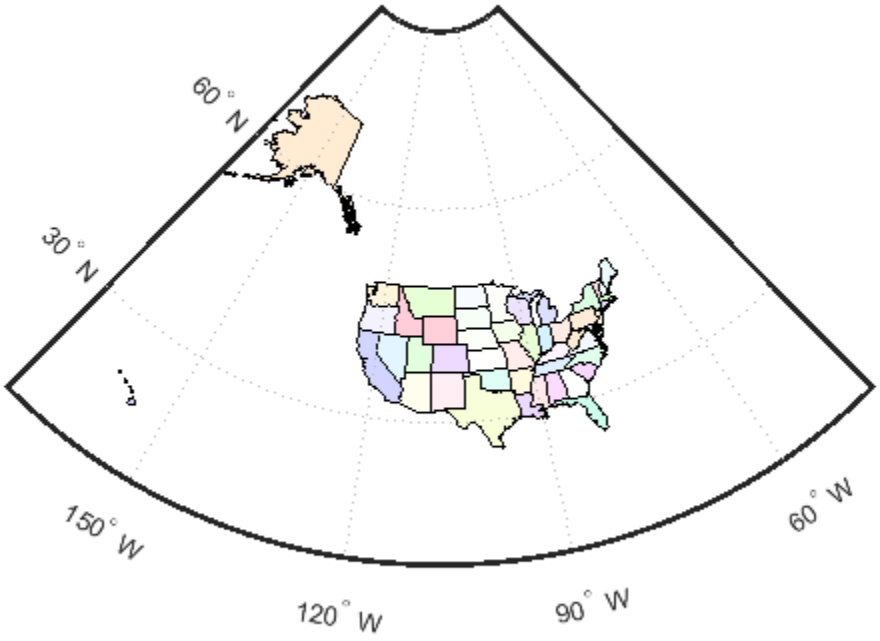
```
faceColors = makesymbolspec('Polygon',...
    {'INDEX', [1 numel(states)], 'FaceColor',...
```

```
    polcmap(numel(states))});
geoshow(states,'SymbolSpec',faceColors)
```



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, seed the MATLAB® random number function using the `rng` function and a fixed value of your choice.

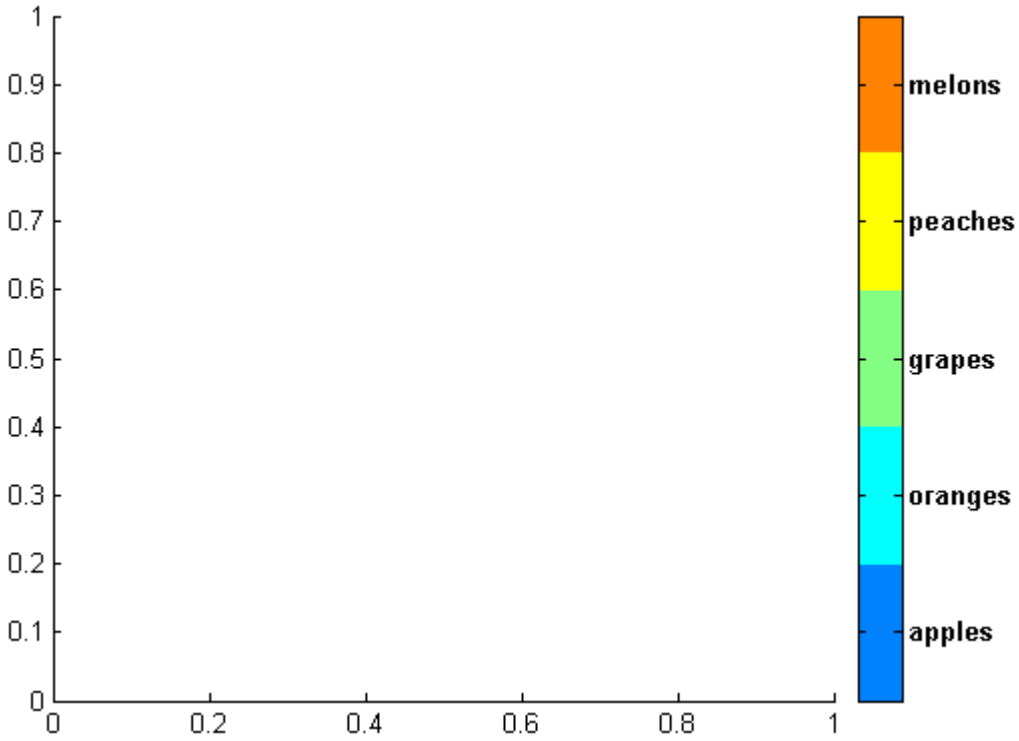
```
figure
worldmap na
rng(0)
faceColors = makesymbolspec('Polygon',...
   {'INDEX', [1 numel(states)], 'FaceColor', polcmap(256,.2)});
geoshow(states, 'SymbolSpec', faceColors)
```



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, **Colormap Editor**, described in the MATLAB Function Reference pages. Also see the MATLAB `colormap` function documentation.

Scale Maps for Printing

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 “Printing and Saving”.

Manipulating Geospatial Data

For some purposes, geospatial data is fine to use as is. Sooner or later, though, you need to extract, combine, massage, and transform geodata. This chapter discusses some Mapping Toolbox tools and techniques provided for such purposes.

- "Extract and Join Polygons or Line Segments" on page 7-2
- "Link Line Segments with Common Endpoints into Polygons" on page 7-4
- "Geographic Interpolation of Vectors" on page 7-5
- "Interpolate Vertices Between Known Data Points" on page 7-7
- "Interpolate Coordinates at Specific Locations" on page 7-8
- "Vector Intersections" on page 7-9
- "Calculate Intersections of Small Circles" on page 7-11
- "Calculate Intersection of Rhumb Line Tracks" on page 7-12
- "Calculate Intersections of Vector Data" on page 7-13
- "Calculate Area of Geographic Polygons" on page 7-15
- "Polygon Set Logic" on page 7-16
- "Overlay Polygons Using Set Logic" on page 7-17
- "Remove Longitude Coordinate Discontinuities at Date Line Crossings" on page 7-22
- "Polygon Buffer Zones" on page 7-26
- "Trim Vectors to Preserve Polygonal Patches" on page 7-28
- "Simplify Vector Coordinate Data" on page 7-31
- "Simplify Polygon and Line Data" on page 7-32
- "Convert Vector Data to Raster Format" on page 7-37
- "Rasterize Polygons Interactively" on page 7-42
- "Data Grids as Logical Variables" on page 7-44
- "Compute Elevation Profile Along Straight Line" on page 7-45
- "Compute Gradient, Slope, and Aspect from Regular Data Grid" on page 7-48

Extract and Join Polygons or Line Segments

This example shows how 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 the `polysplit` function, which takes column vectors as inputs. To join together individual polygon or line vectors use `polyjoin`.

Create two NaN-delimited arrays in the form of column vectors.

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

Split the column vectors into individual line segment cell arrays at the NaN separators using `polysplit`.

```
[latc,lonc] = polysplit(lat,lon)
```

```
latc=2×1 cell array
    {3x1 double}
    {4x1 double}


lonc=2×1 cell array
    {3x1 double}
    {4x1 double}
```

Inspect the contents of the cell arrays. Note that each cell array element contains a segment of the original line.

```
[latc{1} lonc{1}]
```

```
ans = 3×2

   45.6000   13.0000
  -23.4700  -97.4500
   78.0000  165.0000
```

```
[latc{2} lonc{2}]
```

```
ans = 4×2

   43.9000         0
  -67.1400 -114.2000
   90.0000  -18.0000
  -89.0000         0
```

To reverse the process, use `polyjoin`.

```
[lat2,lon2] = polyjoin(latc,lonc);
```

Perform a logical comparison of the joined segments. Note that they are identical with the initial `lat` and `lon` arrays. The logical comparison is false for the NaN delimiters, by definition.

```
[lat lon] == [lat2 lon2]
```

```
ans = 8x2 logical array

   1   1
   1   1
   1   1
   0   0
   1   1
   1   1
   1   1
   1   1
```

Test for global equality, including `NaN` values.

```
isequaln(lat,lat2) & isequaln(lon,lon2)
```

```
ans = logical
   1
```

See Also

`polyjoin` | `polysplit`

More About

- “Create and Display Polygons” on page 2-12

Link Line Segments with Common Endpoints into Polygons

This example shows how to link line segments into polygons using the `polymerge` function. `polymerge` links sets of line segments together by concatenating segments that have matching endpoints. An end point can be either the first or last vertex in a given part. The `polymerge` function 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. For more information, see the `polymerge` reference page.

Construct column vectors representing coordinate values. The vectors use `NaN` separators to define four line segments.

```
lat = [3 2 NaN 1 2 NaN 5 6 NaN 3 4]';
lon = [13 12 NaN 11 12 NaN 15 16 NaN 13 14]';
```

Concatenate the segments with matching endpoints. Three of the line segments have overlapping end points, so `polymerge` returns two line segments.

```
[latm, lonm] = polymerge(lat,lon)
```

```
latm = 8×1

     1
     2
     3
     4
   NaN
     5
     6
   NaN


lonm = 8×1

    11
    12
    13
    14
   NaN
    15
    16
   NaN
```

See Also

`polysplit`

More About

- “Create and Display Polygons” on page 2-12

Geographic Interpolation of Vectors

When using vector data, remember that, like raster data, coordinates are sampled measurements. This involves unavoidable assumptions concerning what the geographic reality is between specified data points. The normal assumption when plotting vector data requires that points be connected with straight line segments, which essentially indicates a lack of knowledge about conditions between the measured points. For lines that are by nature continuous, such as most rivers and coastlines, such piecewise linear interpolation can be false and misleading, as the following figure depicts.

Interpolating Sparse Vector Data

Despite the possibility of misinterpretation, circumstances do exist in which geographic data interpolation is useful or even necessary. To do this, use the `interpm` function to interpolate between known data points. One value of linearly interpolating points is to fill in lines of constant latitude or longitude (e.g., administrative boundaries) that can curve when projected. To see an example that uses `interpm`, view “Interpolate Vertices Between Known Data Points” on page 7-7.

`interpm` returns both the original data and new linearly interpolated points. Sometimes, however, you might want only the interpolated values. The functions `intrplat` and `intrplon` work similarly to the MATLAB `interp1` function, and give you control over the method used for interpolation. Note that they only interpolate and return one value at a time. Use `intrplat` to interpolate a latitude for a given longitude. Given a monotonic set of longitudes and their matching latitude points, you can interpolate a new latitude for a longitude you specify, interpolating along linear, spline, cubic, rhumb line, or great circle paths. The longitudes must increase or decrease monotonically. If this is not the case, you might be able to use the `intrplon` companion function if the latitude values are monotonic. The following diagram illustrates these three types of interpolation. The `intrplat` function also can perform spline and cubic spline interpolations.

Three Types of Interpolation

To see an example that uses `intrplat`, view “Interpolate Coordinates at Specific Locations” on page 7-8.

Interpolate Vertices Between Known Data Points

This example shows how to interpolate values in a set of vertices using the `interpm` function. In this example, you specify that no gap greater than 1 degree should exist between existing vertices, as specified by the `maxdiff` parameter. See `interpm` for more information.

Define two vectors containing the latitude and longitude values for a set of vertices. In `lat`, note that a gap of 2 degrees exists between the values 2 and 4. Similarly, in `lon`, a gap of 2 degrees exists between the values 1 and the 3.

```
lat = [1 2 4 5];
lon = [1 3 4 5];
```

Call `interpm` to fill in any gaps greater than 1 degree in either vector. For example, `interpm` interpolates and inserts the value 2 into the `lon` vector to fill the gap between the values 1 and 3, and inserts the value 1.5 in the `lat` vector for this new vertex. Similarly, `interpm` inserts the value 3 into the `lat` vector to fill the gap between the values 2 and 4, and inserts the value 3.5 in the `lon` vector for this new vertex. Now, the separation of adjacent vertices is no greater than `maxdiff` in either `newlat` or `newlon`.

```
maxdiff = 1;
[newlat,newlon] = interpm(lat,lon,maxdiff)
```

```
newlat = 6×1

    1.0000
    1.5000
    2.0000
    3.0000
    4.0000
    5.0000


newlon = 6×1

    1.0000
    2.0000
    3.0000
    3.5000
    4.0000
    5.0000
```

Interpolate Coordinates at Specific Locations

This example shows how to interpolate coordinates at specific locations using `intrplat` and `intrplon` functions. `intrplat` and `intrplon` return one value at a time and give you control over the interpolation method used. For more information, see the `intrplat` and `intrplon` reference pages.

Define latitudes and longitudes.

```
lat = [57 68 60 65 56];
lon = [1 3 4 9 13];
```

Specify the longitude for which you want to compute a latitude.

```
newlon = 7.3;
```

Linear Interpolation

Generate a new latitude with linear interpolation.

```
newlat = intrplat(lon,lat,newlon,'linear')
```

```
newlat = 63.3000
```

Great Circle Interpolation

Generate a new latitude using great circle interpolation.

```
newlat = intrplat(lon,lat,newlon,'gc')
```

```
newlat = 63.5029
```

Rhumb Line Interpolation

Generate a new latitude using interpolation along a rhumb line.

```
newlat = intrplat(lon,lat,newlon,'rh')
```

```
newlat = 63.3937
```

To see an illustration comparing these three interpolations, see “Geographic Interpolation of Vectors” on page 7-5.

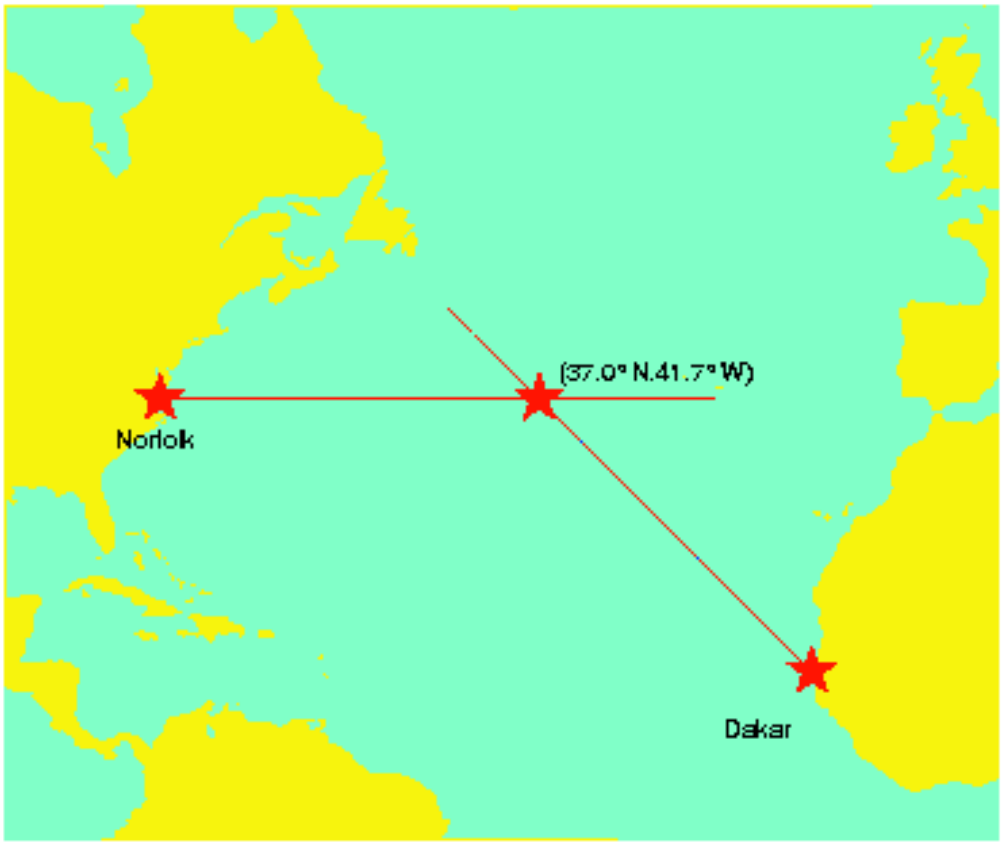
Vector Intersections

Mapping Toolbox includes a set of functions that calculate the intersections of vector data, such as great circles, small circles, and rhumb line tracks. The functions also determine intersections of arbitrary vector data.

Function	Description
`gcxgc`	Find intersection points for pairs of great circles on the sphere
`scxsc`	Find intersection points for pairs of small circles on the sphere
`rhxrh`	Find intersection points, if any, for pairs of rhumb lines
`gcxsc`	Find intersection points, if any, between a great circle and a small circle on the sphere
`polyxpoly`	Find intersection points for lines or polygon edges in the plane

In general, small circles intersect twice or never, as shown in the following figure. For the case of exact tangency, `scxsc` returns two identical intersection points. To see an example of using `scxsc`, see “Calculate Intersections of Small Circles” on page 7-11.

To illustrate finding the intersection of rhumb lines, imagine a ship setting sail from Norfolk, Virginia (37°N,76°W), maintaining a steady due-east course (90°), and another ship setting sail from Dakar, Senegal (15°N,17°W), with a steady northwest course (315°). Where would the tracks of the two vessels cross? 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. To see an example of using `rhxrh`, see “Calculate Intersection of Rhumb Line Tracks” on page 7-12.



See Also

More About

- “Calculate Intersections of Vector Data” on page 7-13
- “Calculate Intersection of Rhumb Line Tracks” on page 7-12
- “Calculate Intersections of Small Circles” on page 7-11

Calculate Intersections of Small Circles

This example shows how to calculate the intersection of vector data, in particular, two small circles. The `scxsc` function returns the intersecting points on the circles.

Calculate the intersection of two small circles. One circle is centered at (0,0) degrees with a radius of 1250 nautical miles. The other circle is centered at 5 degrees north and 30 degrees east with a radius of 2500 kilometers. The function returns the latitude and longitude of the two points of intersection. (Circles typically intersect at two points.) To see an illustration of this calculation, see “Vector Intersections” on page 7-9.

```
[lat,lon] = scxsc(0,0,nm2deg(1250),5,30,km2deg(2500))
```

```
lat = 1×2

  -12.9839   17.7487


lon = 1×2

   16.4170   11.0624
```

Calculate Intersection of Rhumb Line Tracks

This example shows how to calculate the intersection of rhumb lines using the `rhxrh` function.

Calculate the intersection of two rhumb lines. One line starts at latitude 37 degrees North and longitude 76 degrees West and continues due-east at 90 degrees. The other line starts at latitude 15 degrees North and longitude 17 degrees West and continues on a north-west track. To see an illustration of this example, see “Vector Intersections” on page 7-9.

```
[lat,long] = rhxrh(37,-76,90,15,-17,315)
```

```
lat = 37.0000
```

```
long = -41.7028
```

Calculate Intersections of Vector Data

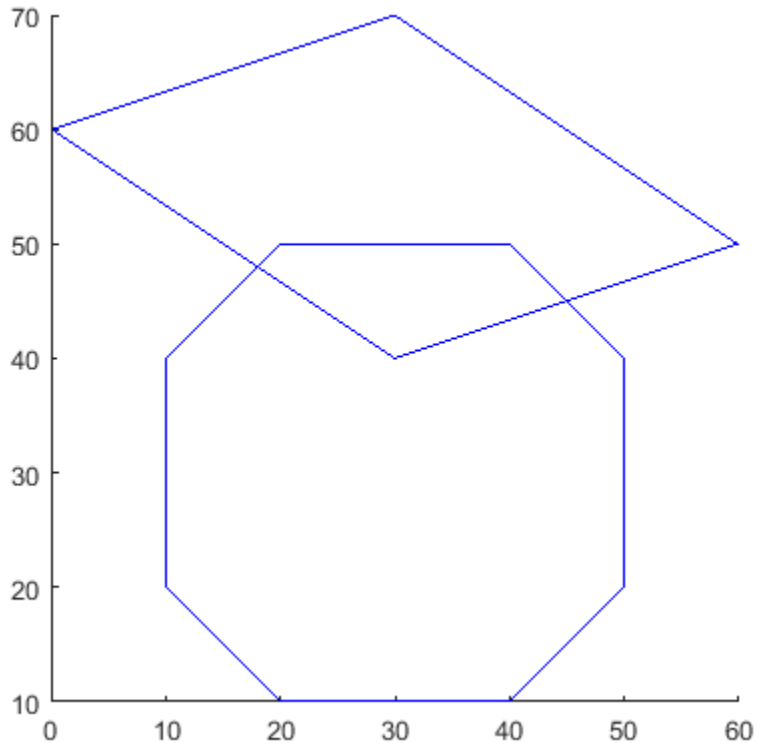
This example shows how to calculate the intersections of arbitrary vector data, such as polylines or polygons, using the `polyxpoly` function.

Create two polygons that intersect.

```
x1 = [10 20 40 50 50 40 20 10 10];
y1 = [20 10 10 20 40 50 50 40 20];

x2 = [30 60 30 0 30];
y2 = [40 50 70 60 40];
```

Plot the polygons.

```
mapshow(x1,y1)
mapshow(x2,y2)
```



Calculate the points where these two polygons intersect. The `polyxpoly` command finds the segments that intersect and interpolates to find the intersection points.

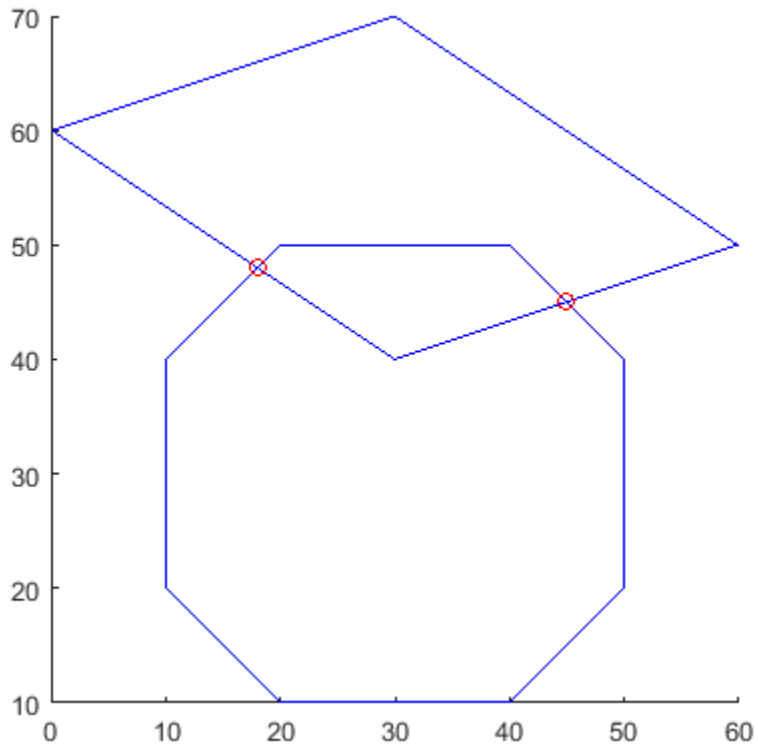
```
[xint,yint] = polyxpoly(x1,y1,x2,y2)
```

```
xint = 2×1

    45
    18
```

```
yint = 2×1

    45
    48
```

Display the points of intersection. 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.

```
mapshow(xint,yint,'Displaytype','point','Marker','o')
```



Calculate Area of Geographic Polygons

This example shows how to calculate geographic areas for vector data in polygon format using the `areaint` function. `areaint` 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. For more information, see `areaint`.

Load the continental United States MAT-file, `conus.mat` , and calculate the radius of the Earth.

```
load conus
earthradius = almanac('earth','radius');
```

Calculate the area of the continental United States, along with the area of Long Island and Martha's Vineyard. `areaint` like the other area functions, `areaquad` and `areamat`, returns the area as a fraction of the entire planet's surface, unless you provide a radius. Because the default Earth radius is in kilometers, the area is in square kilometers.

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

```
area = 3×1
10^6 ×

    7.9256
    0.0035
    0.0004
```

Calculate the areas of the Great Lakes using the same variables, this time in square miles. `areaint` returns three areas: the largest for the polygon representing Superior, Michigan, and Huron together, the other two for Erie and Ontario.

```
earthradius = almanac('earth','radius','miles');
area = areaint(gtlakelat,gtlakelon,earthradius)
```

```
area = 3×1
10^4 ×

    8.0120
    1.0382
    0.7634
```

Polygon 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.

For an example, view “Overlay Polygons Using Set Logic” on page 7-17.

Overlay Polygons Using Set Logic

This example shows how to overlay polygons using set logic. The `polybool` function can perform standard set operations, such as intersection, union, subtraction, and exclusive OR, on two sets of vectors, which can represent *x*-*y* or latitude-longitude coordinate pairs. For more information, see “Polygon Set Logic” on page 7-16.

Create Two Polygons

To illustrate these set operations, create a 12-sided polygon and a triangle that overlaps it.

```
az = (0:pi/6:2*pi)';
lat1 = cos(az);
lon1 = sin(az);

lat2 = [0 1 -1 0]';
lon2 = [0 2 2 0]';
```

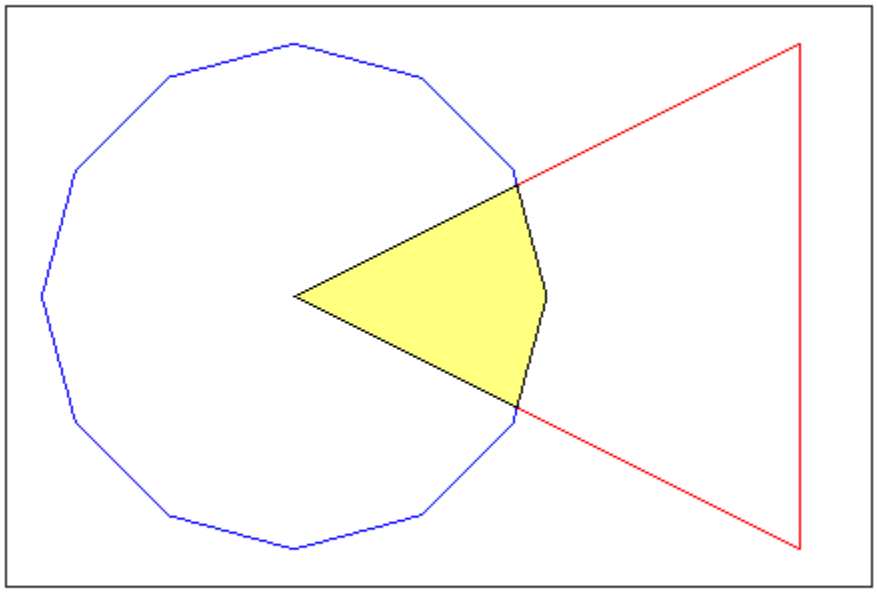
Compute the Intersection of the Two Polygons

Plot the two shapes together with blue and red lines. Compute the intersection polygon using `polybool` and plot it using `geoshow`.

```
figure
axesm miller
plotm(lat1,lon1,'b')
plotm(lat2,lon2,'r')
[loni,lati] = polybool('intersection',lon1,lat1,lon2,lat2);
[lati loni]
```

```
ans = 5×2

    0.0000    1.0000
   -0.4409    0.8819
    0.0000         0
    0.4409    0.8819
    0.0000    1.0000
```

```
geoshow(lati,loni,'DisplayType','polygon')
```



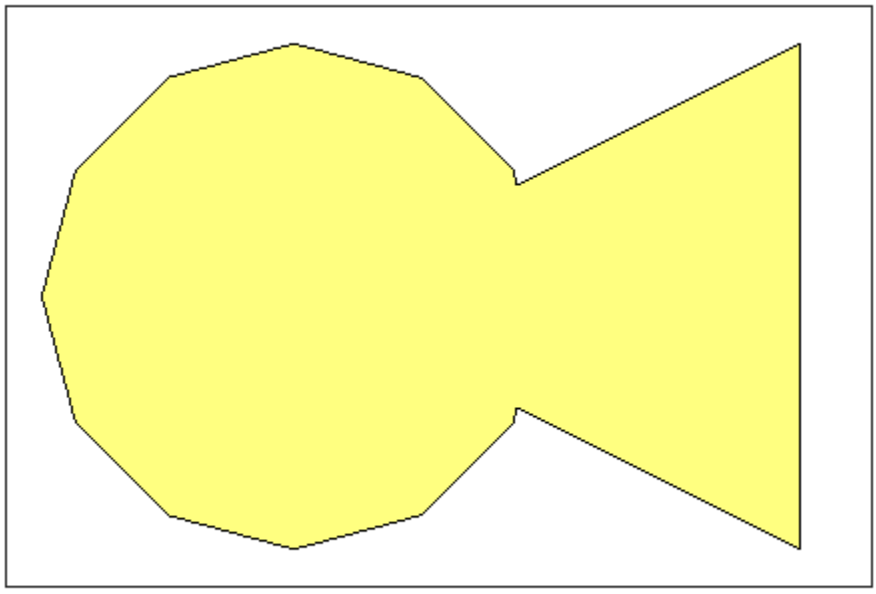
Compute the Union of the Two Polygons

Plot the two shapes together with blue and red lines. Compute the union polygon using `polybool` and plot it using `geoshow`.

```
figure
axesm miller
plotm(lat1,lon1,'b')
plotm(lat2,lon2,'r')
[lonu,latu] = polybool('union',lon1,lat1,lon2,lat2);
[latu lonu]
```

```
ans = 16×2

   -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
    0.0000   -1.0000
    0.5000   -0.8660
    0.8660   -0.5000
      ⋮
```

```
geoshow(latu,lonu,'DisplayType','polygon')
```



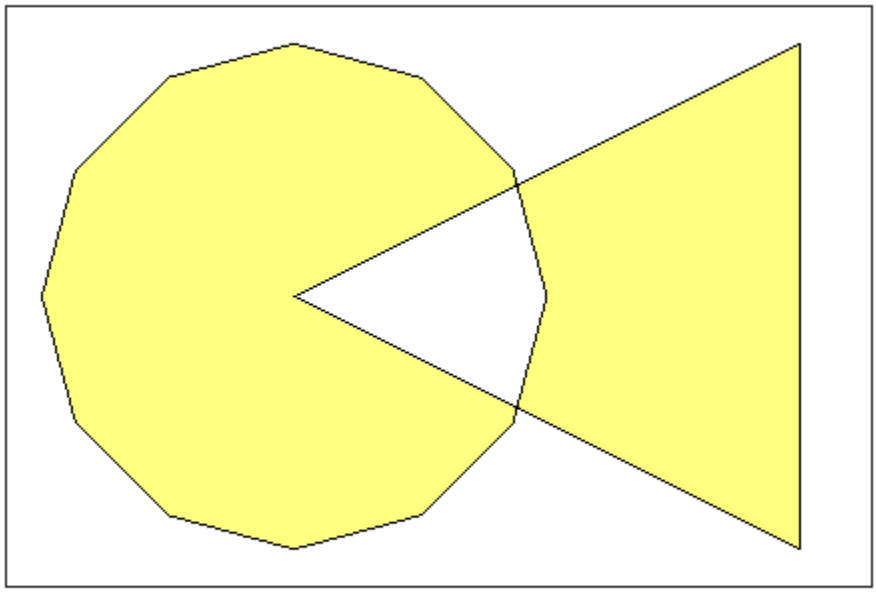
Compute the Exclusive-Or of the Two Polygons

Plot the two shapes together with blue and red lines. Compute the Exclusive-Or polygon using `polybool` and plot it using `geoshow`.

```
figure
axesm miller
plotm(lat1,lon1,'b')
plotm(lat2,lon2,'r')
[lonx,latx] = polybool('xor',lon1,lat1,lon2,lat2);
[latx lonx]
```

```
ans = 22×2

   -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
    0.0000   -1.0000
    0.5000   -0.8660
    0.8660   -0.5000
      ⋮
```

```
geoshow(latx,lonx,'DisplayType','polygon')
```



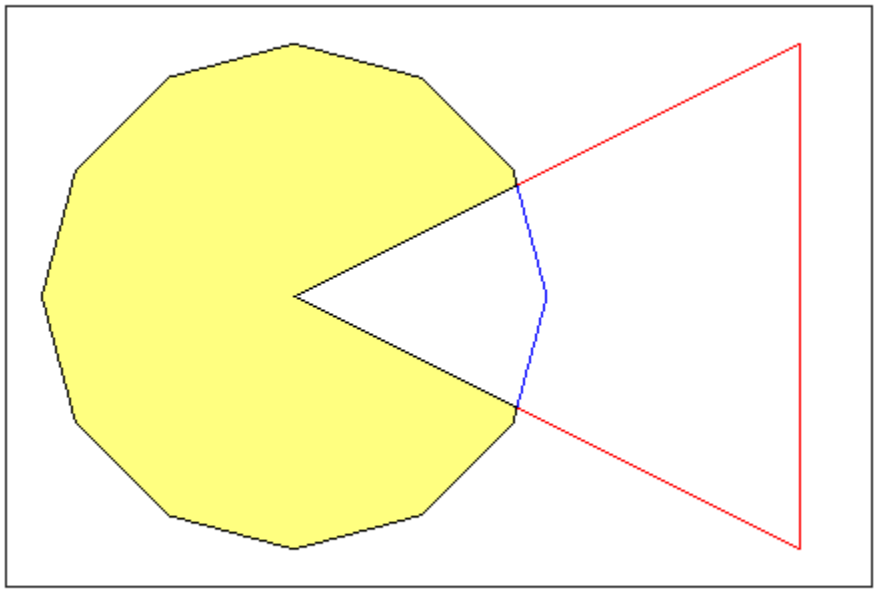
Compute the Subtraction of the Two Polygons

Plot the two shapes together with blue and red lines. Subtract the Exclusive-Or triangle from the 12-sided polygon and plot the resulting concave polygon using `geoshow`.

```
figure
axesm miller
plotm(lat1,lon1,'b')
plotm(lat2,lon2,'r')
[lonm,latm] = polybool('minus',lon1,lat1,lon2,lat2);
[latm lonm]
```

```
ans = 15×2

    0.8660    0.5000
    0.5000    0.8660
    0.4409    0.8819
    0.0000         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
      ⍀
```

```
geoshow(latm,lonm,'DisplayType','polygon')
```



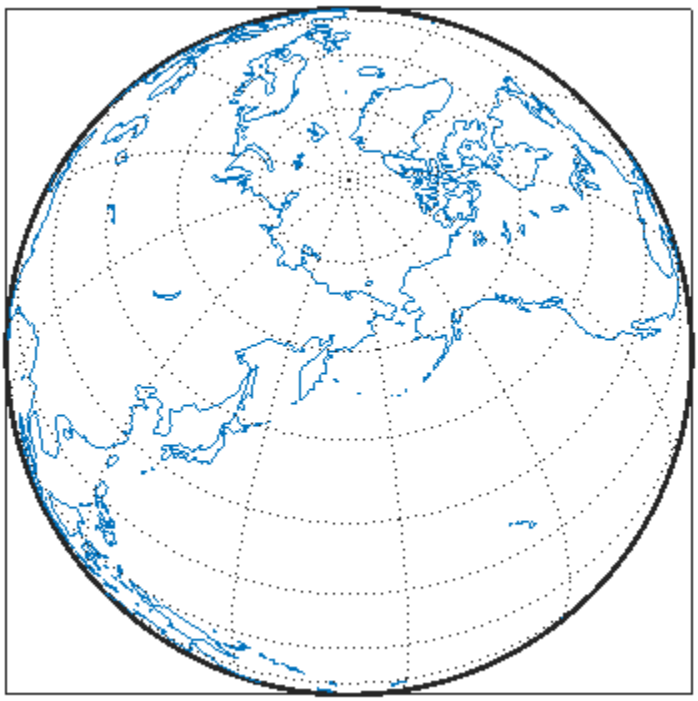
Remove Longitude Coordinate Discontinuities at Date Line Crossings

This example shows how to remove longitude coordinate discontinuities at date line crossings that can confuse set operations on polygons. This can happen when points with longitudes near 180 degrees connect to points with longitudes near -180 degrees, as might be the case for eastern Siberia and Antarctica, and also for small circles and other patch objects. To prepare geographic data for use with `polybool` or for patch rendering, cut the polygons at the date line with the `flatearthpoly` function. `flatearthpoly` returns 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.

Note: The toolbox display functions automatically cut and trim geographic data if required by the map projection. Use `flatearthpoly` only when performing set operations on polygons.

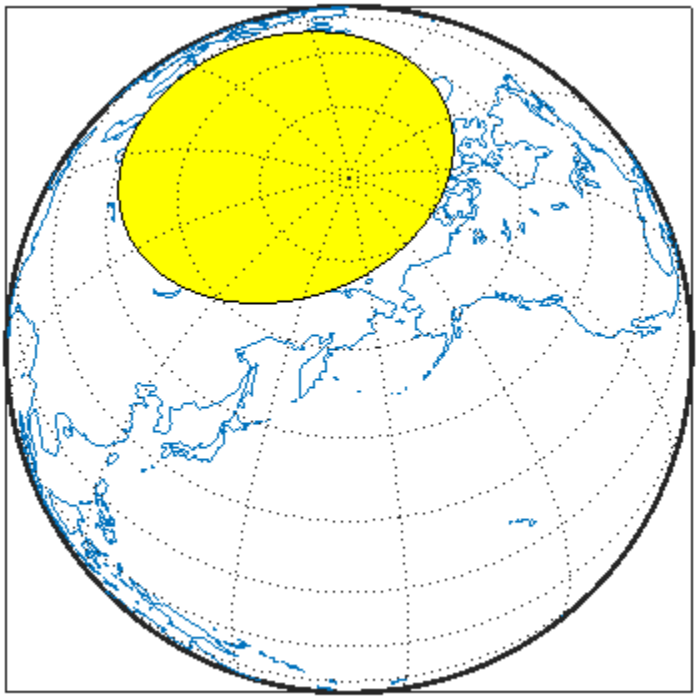
Create an orthographic view of the Earth and plot the coastlines on it.

```
axesm ortho
setm(gca,'Origin', [60 170]); framem on; gridm on
load coastlines
plotm(coastlat,coastlon)
```



Generate a small circle that encompasses the North Pole and color it yellow.

```
[latc,lonc] = scircle1(75,45,30);
patchm(latc,lonc,'y')
```

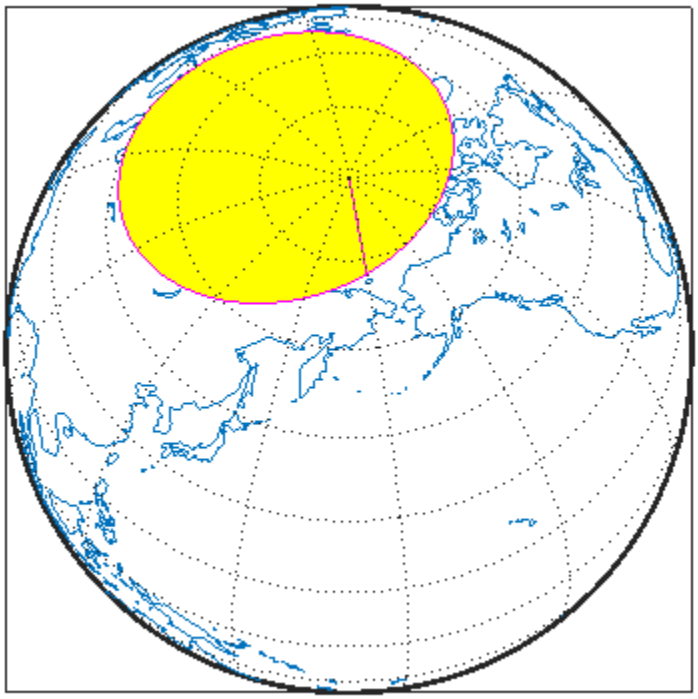


Flatten the small circle using the `flatearthpoly` function.

```
[latf,lonf] = flatearthpoly(latc,lonc);
```

Plot the cut circle that you just generated as a magenta line.

```
plotm(latf,lonf,'m')
```

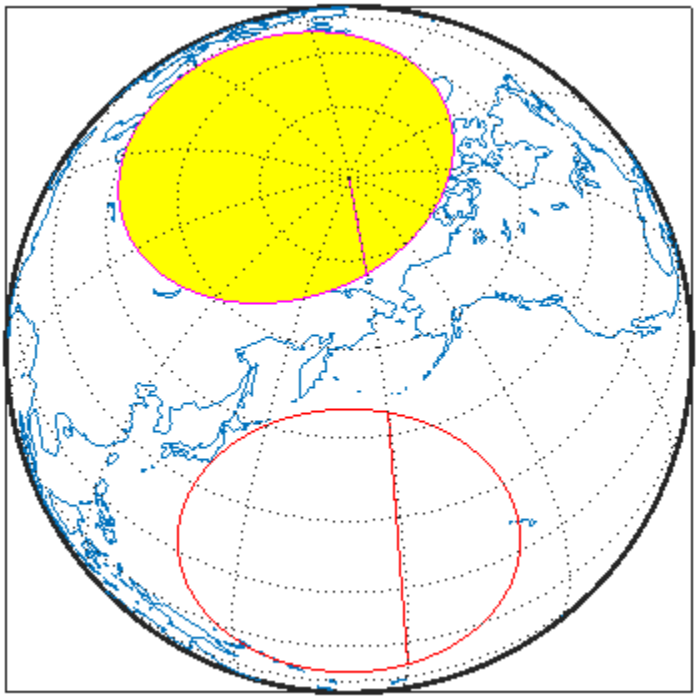


Generate a second small circle that does not include a pole.

```
[latc1 lonc1] = scircle1(20, 170, 30);
```

Flatten the circle and plot it as a red line. Note that the second small circle, which does not cover a pole, is clipped into two pieces along the date line. 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.

```
[latf1,lonf1] = flatearthpoly(latc1,lonc1);
plotm(latf1,lonf1,'r')
```



See Also

`flatearthpoly` | `ispolycw` | `poly2ccw` | `poly2cw`

More About

- “Create and Display Polygons” on page 2-12

Polygon 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. A buffer zone can be defined as the locus of points within a certain distance of the boundary of the feature polygon, either inside or outside the polygon. Buffer zones form equidistant contour lines around objects.

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

Generate Buffer Internal to Polygon

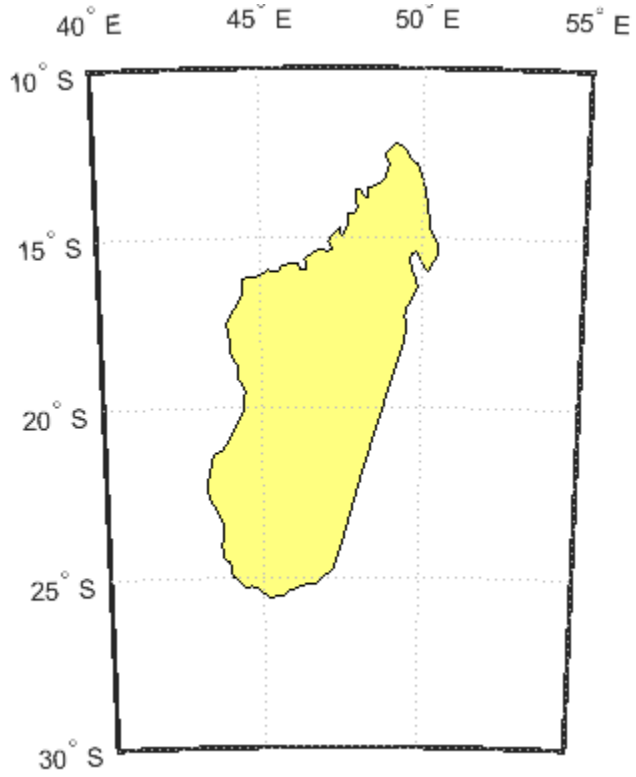
This example shows how to use the `bufferm` function to generate a buffer zone internal to a land area polygon.

Import Madagascar polygon shape.

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

Create a map showing Madagascar.

```
figure
worldmap('madagascar')
geoshow(madagascar)
```

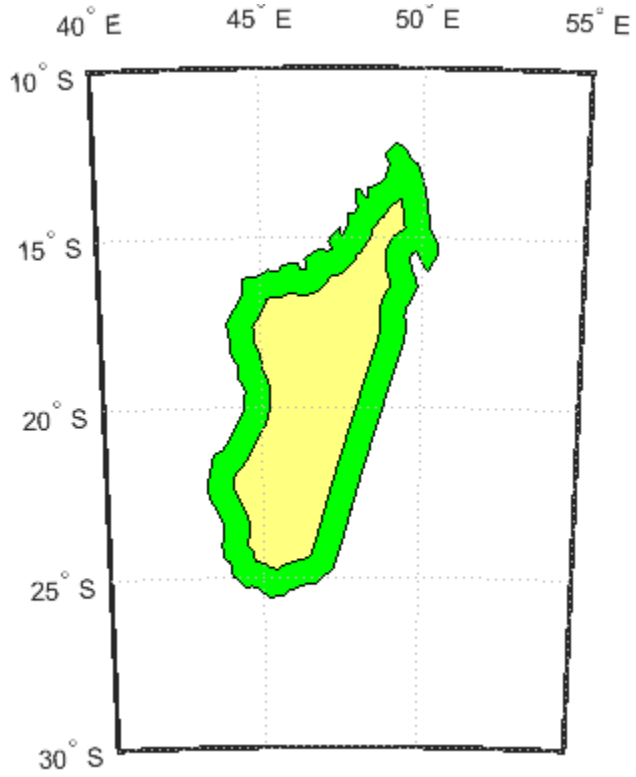


Use `bufferm` to create a buffer zone that extends 0.75 degrees inland from the coast of Madagascar.

```
madlat = madagascar.Lat;
madlon = madagascar.Lon;
bufwidth = 0.75;
direction = 'in';
[latbuf,lonbuf] = bufferm(madlat,madlon,bufwidth,direction);
```

Show the buffer zone in green.

```
geoshow(latbuf,lonbuf,'DisplayType','polygon','FaceColor','green')
```



Trim Vectors to Preserve Polygonal Patches

This example shows how to trim vectors to form lines and polygons using the `maptriml` and `maptrimp` functions. 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, 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. If you want to handle vectors as line data, the `maptriml` function returns variables containing only those points that lie within the defined region. If, instead, you want to maintain polygon format, use the `maptrimp` function. Be aware, however, that patch-trimmed data is usually larger and more expensive to compute.

Note: When drawing maps, Mapping Toolbox display functions automatically trim vector geodata to the region specified by the frame limits (`FLatLimit` and `FLonLimit` map axes properties) for azimuthal projections, or to frame or map limits (`MapLatLimit` and `MapLonLimit` map axes properties) for nonazimuthal projections. The trimming is done internally in the display routine, keeping the original data intact.

Load the `coastlines` MAT-file. This file contains data for the entire world.

```
load coastlines
```

Define a region-of-interest centered on Australia.

```
latlim = [-50 0];
lonlim = [105 160];
```

Use `maptriml` to delete all line data outside these limits, producing line vectors.

```
[linelat,linelon] = maptriml(coastlat,coastlon,latlim,lonlim);
```

Use `maptrimp` to delete all polygon data outside these limits, producing polygon vectors.

```
[polylat,polylon] = maptrimp(coastlat,coastlon,latlim,lonlim);
```

Examine the variables to see how much data has been reduced. The clipped data is only 10% as large as the original data set.

```
whos
```

```
  Name             Size            Bytes  Class     Attributes

  coastlat      9865x1             78920  double
  coastlon      9865x1             78920  double
  latlim           1x2                16  double
  linelat        977x1              7816  double
  linelon        977x1              7816  double
  lonlim           1x2                16  double
  polylat        961x1              7688  double
  polylon        961x1              7688  double
```

Plot the trimmed patch vectors using a Miller projection.

```
axesm('MapProjection', 'miller', 'Frame', 'on',...
'FlatLimit', latlim, 'FlonLimit', lonlim)
patchesm(polylat, polylon, 'c')
```

Plot the trimmed line vectors to see that they conform to the patches.

```
plotm(linelat, linelon, 'm')
```

Simplify Vector Coordinate Data

Avoiding visual clutter in composing maps is an essential part of cartographic presentation. In cartography, this is described as map generalization, which involves coordinating many techniques, both manual and automated. Limiting the number of points in vector geodata is an important part of generalizing maps, and is especially useful for conditioning cartographic data, plotting maps at small scales, and creating versions of geodata for use at small scales.

An easy, but naive, approach to point reduction is to discard every *n*th element in each coordinate vector (simple decimation). However, this can result in poor representations of the original shapes. The toolbox provides a function to eliminate insignificant geometric detail in linear and polygonal objects, while still maintaining accurate representations of their shapes. The `reducem` function implements a powerful line simplification algorithm (known as Douglas-Peucker) that intelligently selects and deletes visually redundant points.

The `reducem` function takes latitude and longitude vectors, plus an optional linear tolerance parameter as arguments, and outputs reduced (simplified) versions of the vectors, in which deviations perpendicular to local "trend lines" in the vectors are all greater than the tolerance criterion. Endpoints of vectors are preserved. Optional outputs are an error measure and the tolerance value used (it is computed when you do not supply a value). For an example, see “Simplify Polygon and Line Data” on page 7-32

`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.

Simplify Polygon and Line Data

This example shows how to simplify polygon and line data using the `reducem` function. Simplifying polygon and line data can speed up certain calculations without making any noticeable impact on the data. One way to approach simplification is to use `reducem` with the default tolerance value at first and view the output. If the results do not meet your requirements, repeat the operation, increasing or decreasing the tolerance value to achieve the result you desire. `reducem` returns the tolerance value used. For more information about vector data simplification, see “Simplify Vector Coordinate Data” on page 7-31.

Simplify Line Data Using Default Settings

Extract Massachusetts coastlines and state borders from the `usastatehi.shp` shapefile.

```
ma = shaperead('usastatehi.shp','UseGeoCoords',true, ...
    'Selector', {@(name)strcmpi(name,'Massachusetts'),'Name'});
masslat = ma.Lat;
masslon = ma.Lon;
```

The result is a pair of vectors that outline the state with 957 vertices.

```
numel(masslat)
```

```
ans = 957
```

Simplify the outline using the `reducem` function with the default tolerance value.

```
[masslat1, masslon1, cerr, tol] = reducem(masslat', masslon');
```

Inspect the results. The number of vertices has been reduced to 252. The vectors have been reduced to about a quarter of their original lengths.

```
numel(masslat1)
```

```
ans = 252
```

```
numel(masslat1)/numel(masslat)
```

```
ans = 0.2633
```

Examine the error and tolerance values returned by `reducem`. The `cerr` value indicates that `reducem` has reduced the total length of the outline by about 3.3 percent. The tolerance that `reducem` used to achieve this reduction was 0.006 degrees, or about 660 meters.

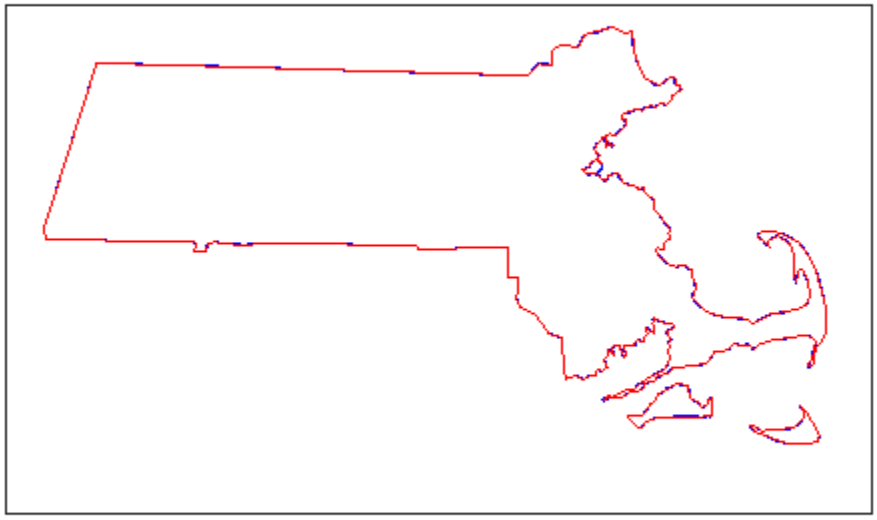
```
[cerr tol]
```

```
ans = 1×2

    0.0331    0.0060
```

Plot the reduced outline in red over the original outline in blue, using `geoshow`. At this resolution, it's hard to see any difference between the original outline and the reduced outline.

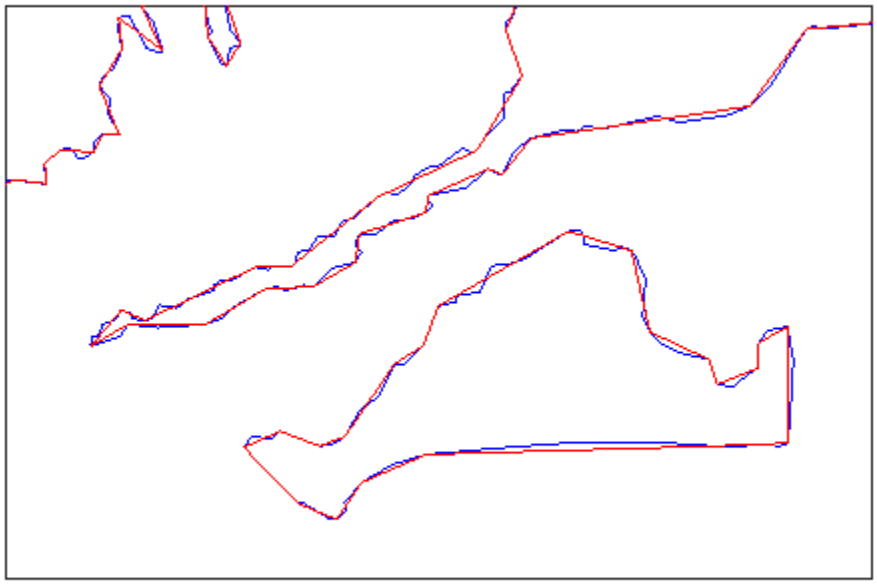
```
figure
axesm('MapProjection', 'eqdcyl', 'MapLatLim', [41.1 43.0],...
'MapLonLim', [-73.6, -69.8], 'Frame', 'off', 'Grid', 'off');
```

```
geoshow(masslat, masslon, 'DisplayType', 'line', 'color', 'blue')
geoshow(masslat1, masslon1, 'DisplayType', 'line', 'color', 'red')
```



To get a better look at the two outlines, use `xlim` and `ylim` to zoom in on a portion of the map. Notice how the reduced outline conforms to the general contours of the original map but loses a lot of the detail.

```
axesm('MapProjection', 'eqdcyl', 'MapLatLim', [41.1 43.0],...
'MapLonLim', [-73.6, -69.8], 'Frame', 'off', 'Grid', 'off');
xlim([0.0104 0.0198])
ylim([0.7202 0.7264])
geoshow(masslat, masslon, 'DisplayType', 'line', 'color', 'blue')
geoshow(masslat1, masslon1, 'DisplayType', 'line', 'color', 'red')
```



Simplify Line Data Changing the Default Tolerance Value

Perform the operation again, this time doubling the tolerance value.

```
[masslat2,masslon2,cerr2,tol2] = reducem(masslat', masslon', 0.012);
numel(masslat2)
```

```
ans = 157
```

```
numel(masslat2)/numel(masslat)
```

```
ans = 0.1641
```

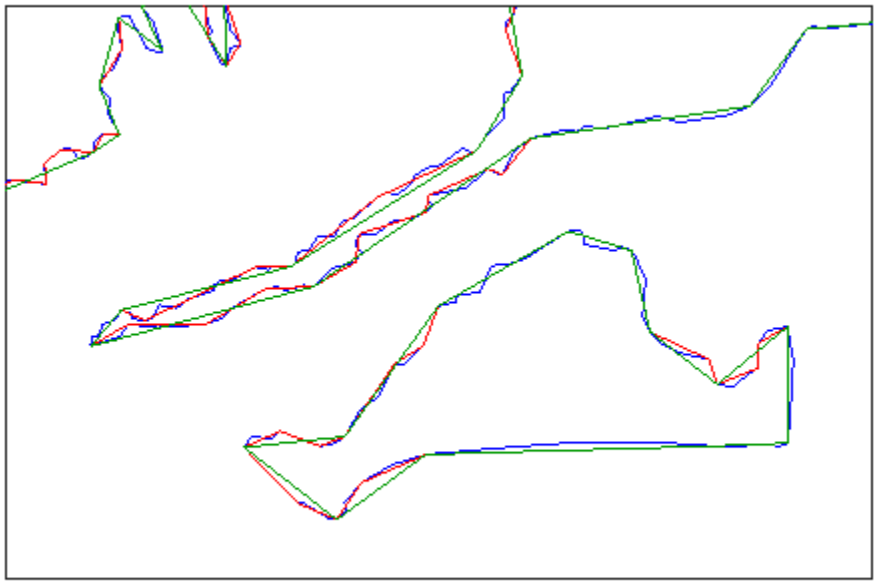
Examine the error and tolerance values returned by `reducem`. This time, the `cerr` value indicates that `reducem` has reduced the total length of the outline by about 5.2 percent. The tolerance that `reducem` used to achieve this reduction was 0.012 degrees.

```
[cerr2 tol2]
```

```
ans = 1×2

    0.0517    0.0120
```

Plot this reduced outline in dark green over the original outline in blue. Note how this reduced outline maintains the general shape of the original map but loses much of the fine detail.

```
geoshow(masslat2, masslon2, 'DisplayType', 'line', 'color', [0 .6 0])
```



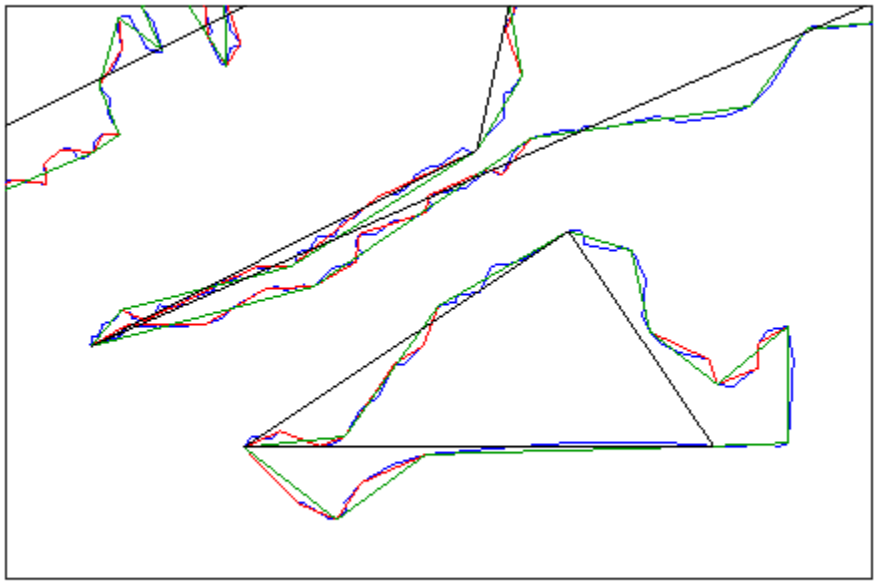
Simplify Line Data Increasing the Tolerance Again

Increase the tolerance to 0.1 degrees.

```
[masslat3, masslon3, cerr3, tol3] = reducem(masslat', masslon', 0.1);
```

Plot this reduced outline in black. Note how this reduced map only retains the broadest elements of the original shape and loses much of the detail.

```
geoshow(masslat3, masslon3, 'DisplayType', 'line', 'color', 'black')
```



Convert Vector Data to Raster Format

You can convert latitude-longitude vector data to a grid at any resolution you choose to make a raster base map or grid layer. Certain Mapping Toolbox GUI tools help you do some of this, but you can also perform vector-to-raster conversions from the command line. The principal function for gridding vector data is `vec2mtx`, which allocates lines to a grid of any size you indicate, marking the lines with 1s and the unoccupied grid cells with 0s. The grid contains doubles, but if you want a logical grid (see "Data Grids as Logical Variables" on page 7-44) cast the result to be a logical array. To see an example, view "Creating Data Grids from Vector Data" on page 7-37.

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`. To see an example, view "Rasterize Polygons Interactively" on page 7-42.

Creating Data Grids from Vector Data

This example shows how to convert vector data to raster data using the `vec2mtx` function. The example uses patch data for Indiana from the `usastatehi` shapefile. For more information, see "Convert Vector Data to Raster Format" on page 7-37.

Use `shaperead` to get the patch data for the boundary.

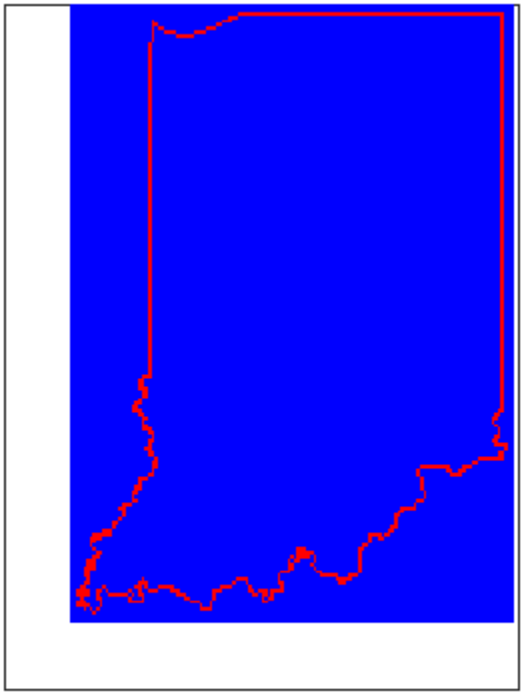
```
indiana = shaperead('usastatehi.shp',...
    'UseGeoCoords', true,...
    'Selector', {@(name)strcmpi('Indiana',name),'Name'});
inLat = indiana.Lat;
inLon = indiana.Lon;
```

Convert the vectors to a regular data grid using `vec2mtx`. Set the grid density to be 40 cells per degree. Rasterize the boundary and generate a referencing vector for it. Convert the referencing vector to a geographic raster reference object.

```
gridDensity = 40;
[inGrid,inRefVec] = vec2mtx(inLat,inLon,gridDensity);
R = refvecToGeoRasterReference(inRefVec,size(inGrid));
```

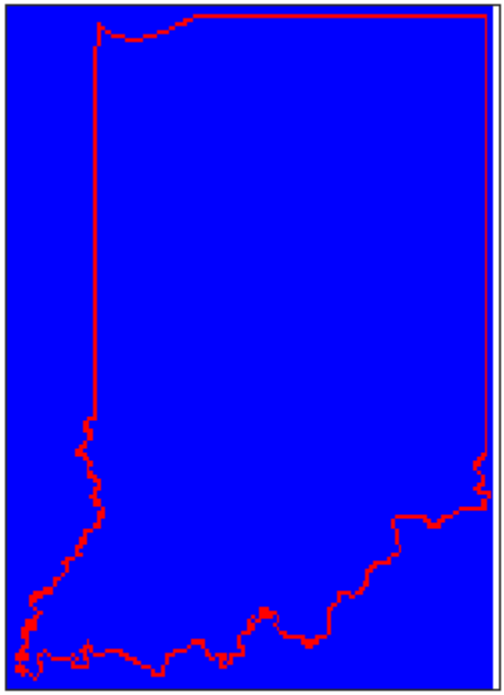
Make a map of the data grid in contrasting colors.

```
figure
axesm eqdcyl
meshm(inGrid,R)
colormap jet(4)
```



Set up the map limits.

```
setm(gca,'Flatlimit',R.LatitudeLimits,'FlonLimit',R.LongitudeLimits)
tightmap
```

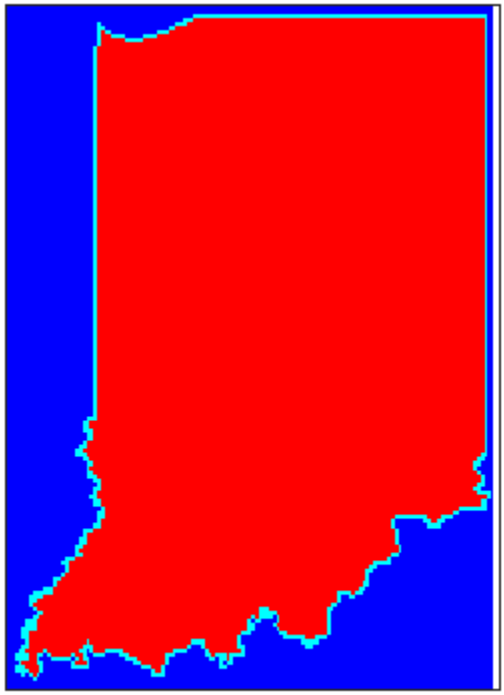


Specify the seed point and seed value. To fill (recode) the interior of Indiana, you need a seed point (which must be identified by row and column) and a seed value (to be allocated to all cells within the polygon). Select the middle row and column of the grid and choose an index value of 3 to identify the territory when calling `encodem` to generate a new grid. The last argument (1) identifies the code for boundary cells, where filling should halt.

```
inPt = round([size(inGrid)/2,3]);
inGrid3 = encodem(inGrid,inPt,1);
```

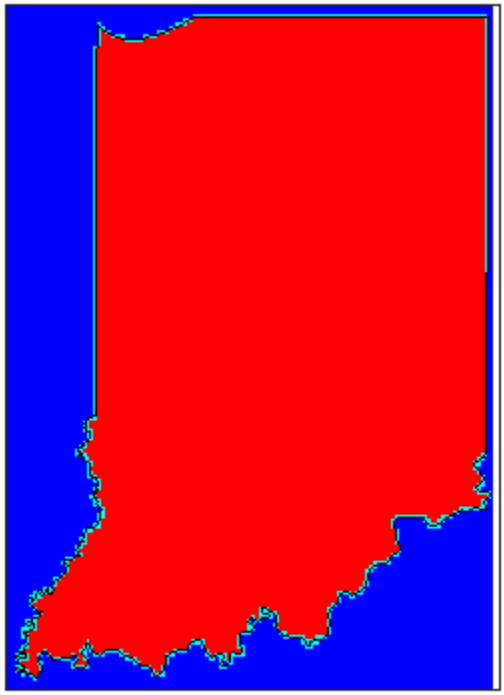
Clear and redraw the map using the filled grid.

```
meshm(inGrid3,R)
```



Plot the original vectors on the grid to see how well data was rasterized. You can use the Zoom tool on the figure window to examine the gridding results more closely.

```
plotm(inLat,inLon,'k')
```



Rasterize Polygons Interactively

This example shows how to use the `encodem` function with seed points found using the `getseeds` function to fill multiple polygons after they are gridded. The example extracts data for Indiana and its surrounding states, and then deletes unwanted areas of these polygons using `maptrimp` .

Extract 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];
```

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);
```

Set up a map figure and display the binary grid you just created.

```
figure
axesm eqdcyl
geoshow(meLat, meLon, 'Color', 'black');
```

Use the `getseeds` function to select five seed points, one in each of the outlines of Indiana, Michigan, Ohio, Kentucky, and Illinois. The `getseeds` function changes the cursor to a cross-hairs. You pick seed points by positioning the cursor within a state boundary and clicking the mouse. The

`getseeds` function returns control to the command prompt after you pick five locations in the figure window.

```
[row,col,val] = getseeds(meGrid, meRefVec, 5, [3 4 5 6 7]);
```

Fill each state with a unique value, producing a new grid, using the `encodem` function.

```
meGrid5 = encodem(meGrid, [row col val], 1);
```

Display `meGrid5` to see the result.

```
clma
meshm(meGrid5, meRefVec)
```

Data Grids as Logical Variables

You can apply logical criteria to numeric data grids to create *logical grids*. Logical grids are data grids consisting entirely of 1s and 0s. You can create them by performing logical tests on data grid variables. The resulting binary grid is the same size as the original grid(s) and can use the same referencing vector, as the following hypothetical data operation illustrates:

```
logicalgrid = (realgrid > 0);
```

This transforms all values greater than 0 into 1s and all other values to 0s. You can apply multiple conditions to a grid in one operation:

```
logicalgrid = (realgrid >- 100)&(realgrid < 100);
```

If several grids are the same size and share the same referencing vector (i.e., the grids are co-registered), you can create a logical grid by testing joint conditions, treating the individual data grids as map layers:

```
logicalgrid = (population > 10000)&(elevation < 400)&...
              (country == nigeria);
```

Several Mapping Toolbox functions enable the creation of logical grids using logical and relational operators. Grids resulting from such operations contain logical rather than numeric values (which reduce storage by a factor of 8), but might need to be cast to `double` in order to be used in certain functions. Use the `ones` and `zeros` functions to create grids of all 1s and all 0s.

Compute Elevation Pro le Along Straight Line

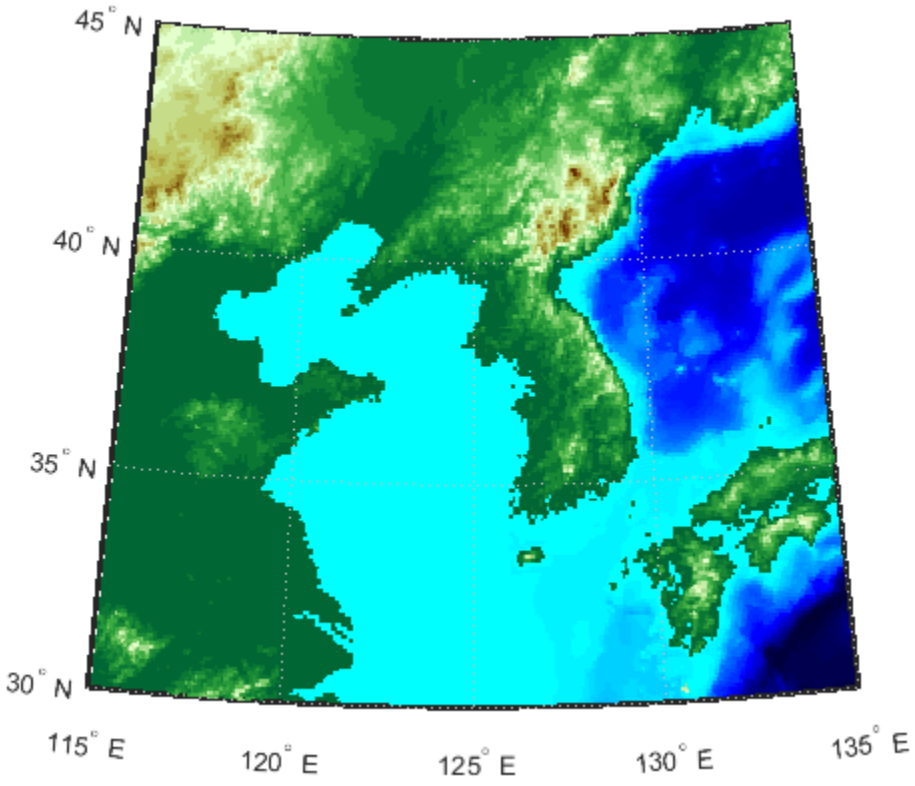
Compute an elevation profile along a straight line using the `mapprofile` function. Calculating data values along a path is a common application when working with gridded geodata. For example, you might want to calculate the terrain height along a transect, a road, or a flight path. The `mapprofile` 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.

Load elevation data and a geographic cells reference object for the Korean peninsula.

```
load korea5c
```

Get the latitude and longitude limits of the elevation data and set up a world map. Display the map and apply a colormap appropriate for elevation data.

```
latlim = korea5cR.LatitudeLimits;
lonlim = korea5cR.LongitudeLimits;
worldmap(latlim, lonlim)
meshm(korea5c,korea5cR,size(korea5c),korea5c)
demcmap(korea5c)
```

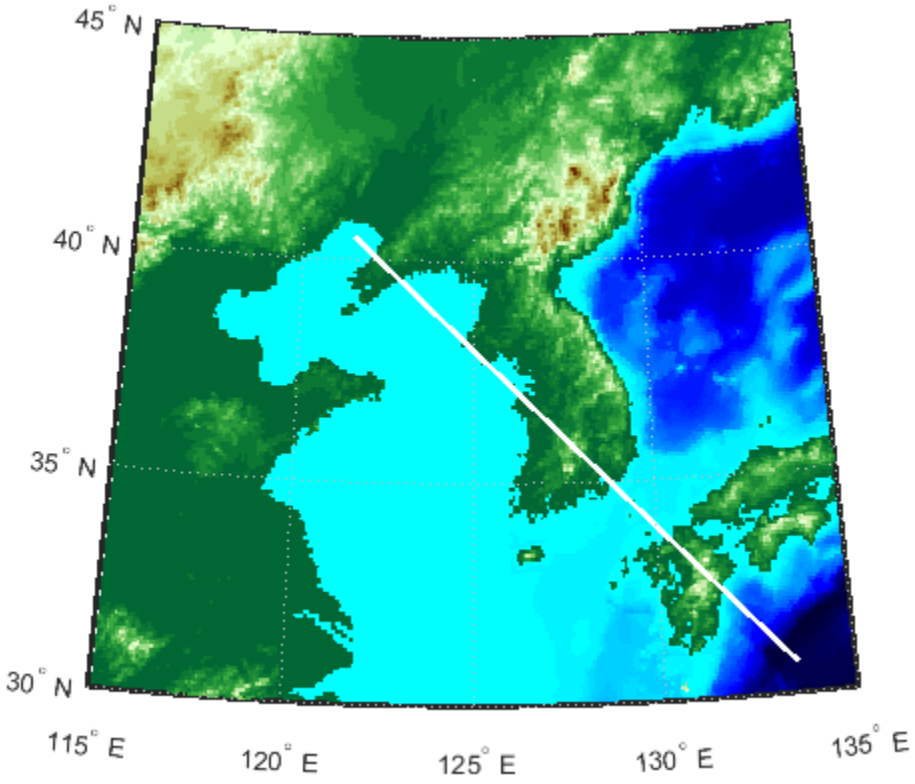


Define endpoints for a straight-line transect through the region. Then, compute the elevation profile using `mapprofile`. By default, `mapprofile` uses bilinear interpolation along a great circle track.

```
plat = [40.5 30.7];
plon = [121.5 133.5];
[z,rng,lat,lon] = mapprofile(korea5c,korea5cR,plat,plon);
```

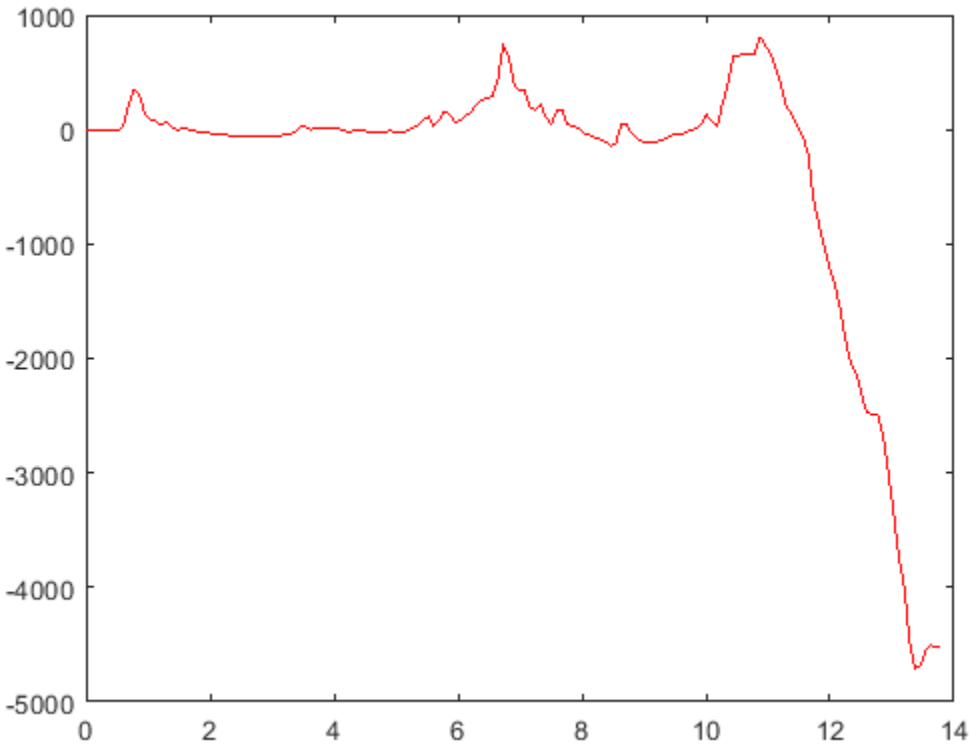
Plot the transect in 3-D so it follows the terrain.

```
plot3m(lat,lon,z,'w','LineWidth',2)
```



Plot the transect range and elevation on a set of Cartesian axes.

```
figure
plot(rng,z,'r')
```


1000
0
-1000
-2000
-3000
-4000
-5000
0
2
4
6
8
10
12
14

Compute Gradient, Slope, and Aspect from Regular Data Grid

This example shows how to compute the gradient, slope, and aspect for a regular data grid. The gradient components are the change in the grid variable per meter of distance in the north and east directions. 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. 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. 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 angles are in units of degrees by default.

Construct a 100-by-100 grid using the `peaks` function and construct a referencing matrix for it.

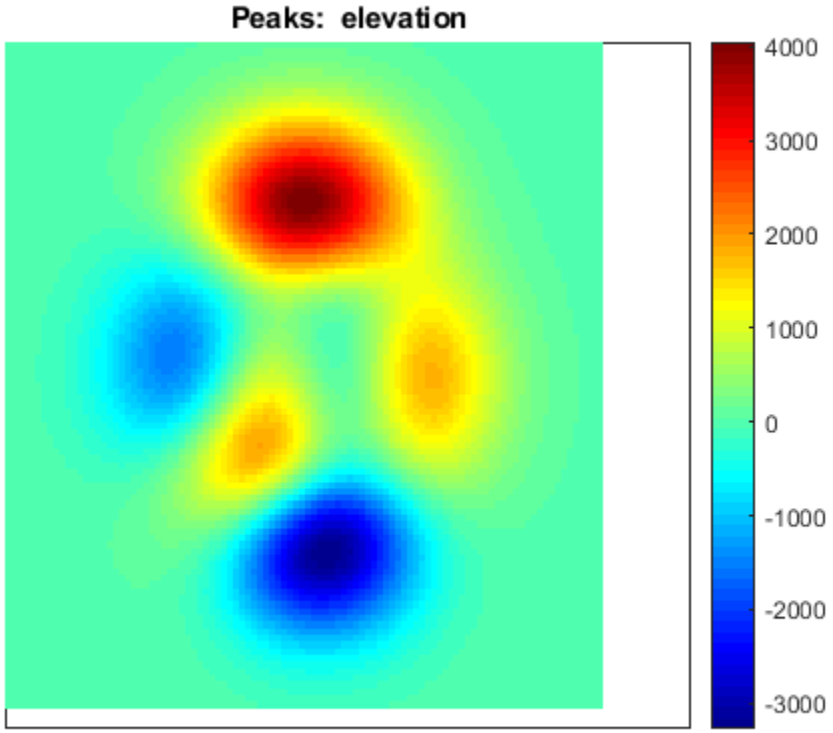
```
datagrid = 500*peaks(100);
gridrv = [1000 0 0];
```

Generate grids containing aspect, slope, gradients to north, and gradients to east.

```
[aspect,slope,gradN,gradE] = gradientm(datagrid,gridrv);
```

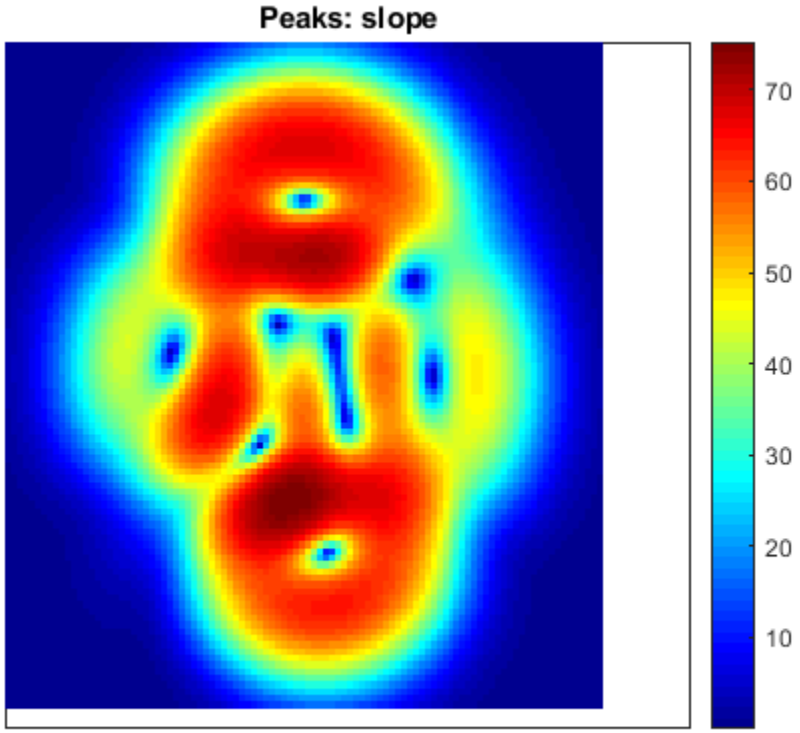
Map the surface data in a cylindrical equal area projection. Start with the original elevations.

```
axesm eqacyl
meshm(datagrid,gridrv)
colormap (jet(64))
colorbar('vert')
title('Peaks:  elevation')
axis square
```



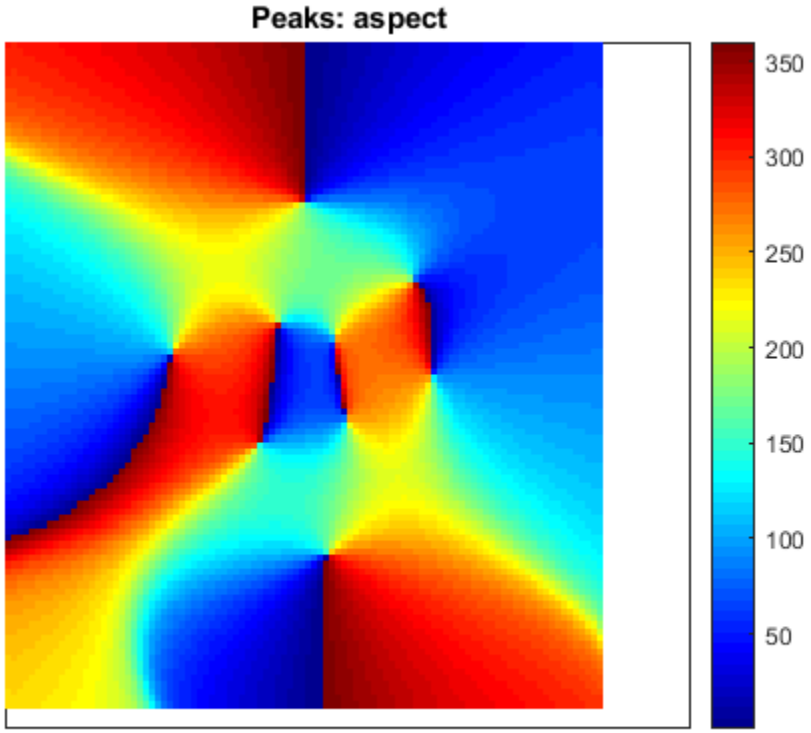
Clear the frame and display the slope grid.

```
clma
meshm(slope,gridrv)
colorbar('vert');
title('Peaks: slope')
```



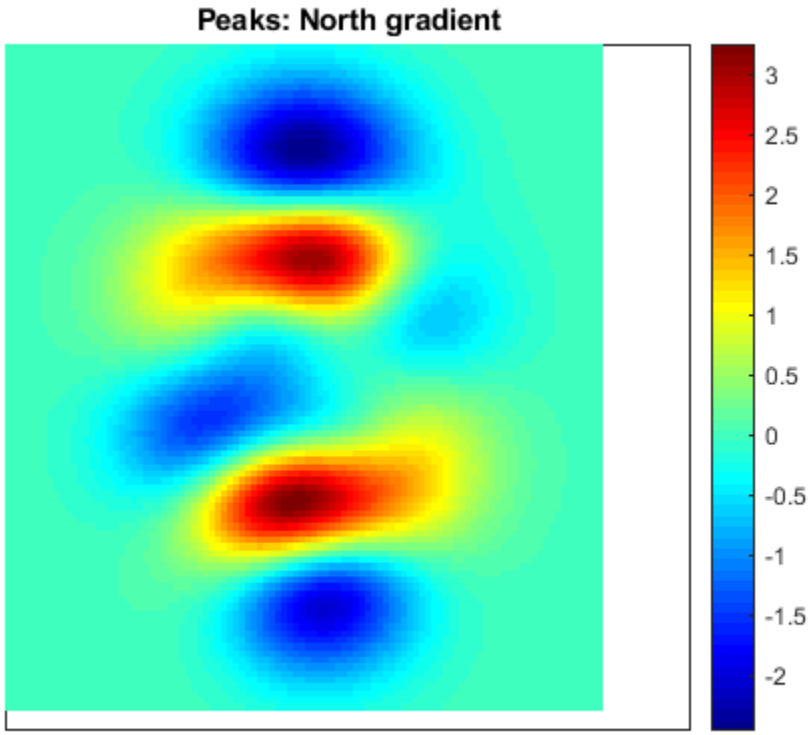
Map the aspect grid.

```
clma
meshm(aspect,gridrv)
colorbar('vert');
title('Peaks: aspect')
```



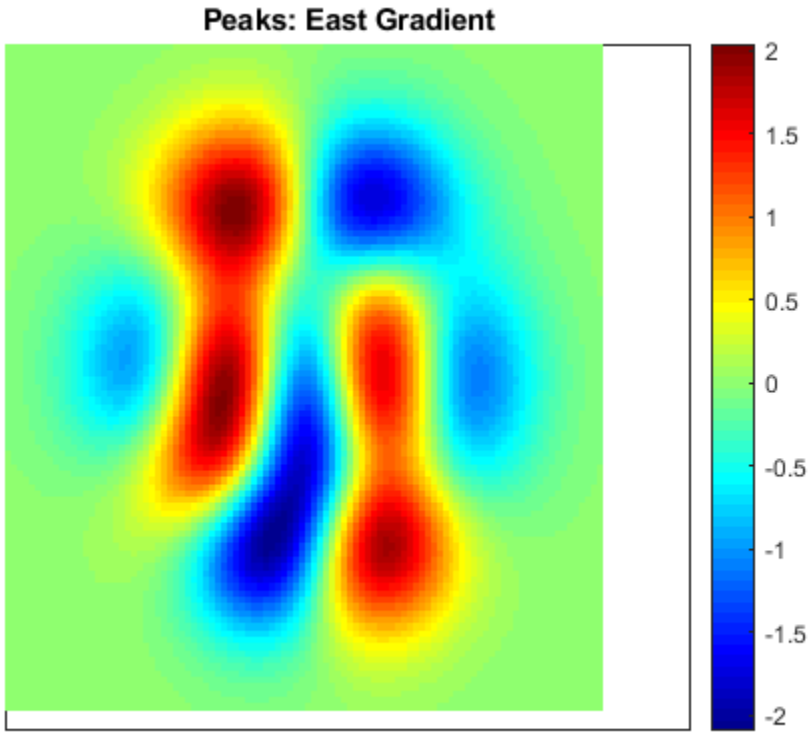
Map the gradients to the north.

```
clma
meshm(gradN,gridrv)
colorbar('vert');
title('Peaks: North gradient')
```



Map the gradients to the east.

```
clma
meshm(gradE,gridrv)
colorbar('vert');
title('Peaks: East Gradient')
```


Peaks: East Gradient
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2

Using Map Projections and Coordinate Systems

All geospatial data must be flattened onto a display surface in order to visually portray what exists where. The mathematics and craft of map projection are central to this process. Although there is no limit to the ways geodata can be projected, conventions, constraints, standards, and applications generally prescribe its usage. This chapter describes what map projections are, how they are constructed and controlled, their essential properties, and some possibilities and limitations.

- "Map Projections and Distortions" on page 8-2
- "Quantitative Properties of Map Projections" on page 8-4
- "The Three Main Families of Map Projections" on page 8-5
- "Projection Aspect" on page 8-9
- "Projection Parameters" on page 8-16
- "Visualize Spatial Error Using Tissot Indicatrices" on page 8-22
- "Visualize Projection Distortions Using Isolines" on page 8-26
- "Quantify Map Distortions at Point Locations" on page 8-30
- "Rotational Transformations on the Globe" on page 8-34
- "Create a UTM Map" on page 8-38
- "Set UTM Parameters Interactively" on page 8-42
- "Work in UTM Without a Displayed Map" on page 8-45
- "Use the Transverse Aspect to Map Across UTM Zones" on page 8-47
- "Summary and Guide to Projections" on page 8-49
- "Transform Coordinates to a Different Projected CRS" on page 8-62
- "Project and Display Raster Data" on page 8-65

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.

Map Projections and Distortions

Humans 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.

A map projection is a procedure that flattens a curved surface such as the Earth onto a plane. Usually this is done through an intermediate surface such as a cylinder or a cone, which is then unwrapped to lie flat. Consequently, map projections are classified as cylindrical, conical, and azimuthal (a direct transformation of the surface of part of a spheroid to a circle). See “The Three Main Families of Map Projections” on page 8-5 for discussions and illustrations of how these transformations work. The toolbox can project both vector data and raster data.

Get a map projection by using a `projcrs` object. For example, get information about a raster data file by creating a `RasterInfo` object. Then, find the projection method by querying the `CoordinateReferenceSystem` property of the object.

```
info = georasterinfo('boston.tif');
info.CoordinateReferenceSystem.ProjectionMethod
```

```
ans =

    "Lambert Conic Conformal (2SP)"
```

Alternatively, Mapping Toolbox includes a library of map projections that you can control with `axesm` and `defaultm`. Some are suitable for showing the entire world, others for half of it, and some are only useful over small areas. For more information, see “Projection Distortions” on page 8-2. For a list of available projections, see “Summary and Guide to Projections” on page 8-49. If you want to use a projection that is not included in this list, create a `projcrs` object instead.

Project coordinates by using the `projfwd` function and unproject coordinates using the `projinv` function.

Use Inverse Projection to Recover Geographic Coordinates

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

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

Projection Distortions

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

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

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

References

[1] Snyder, J. P. "Map Projections – A working manual." *U.S. Geological Survey Professional Paper 1395*. Washington, D.C.: U.S. Government Printing Office, 1987. doi:10.3133/pp1395

[2] Maling, D. H. *Coordinate Systems and Map Projections*. 2nd ed. New York: Pergamon Press, 1992.

[3] Snyder, J. P., and P. M. Voxland. "An album of map projections." *U.S. Geological Survey Professional Paper 1453*. Washington, D.C.; U.S. Government Printing Office, 1989. doi:10.3133/pp1453

[4] Snyder, J. P. *Flattening the Earth – 2000 Years of Map Projections*. Chicago, IL: University of Chicago Press, 1993.

See Also

More About

- “Quantitative Properties of Map Projections” on page 8-4
- “Projection Parameters” on page 8-16

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 8-49.

The Three Main Families of Map Projections

In this section...

"Unwrapping the Sphere to a Plane" on page 8-5
"Cylindrical Projections" on page 8-5
"Conic Projections" on page 8-6
"Azimuthal Projections" on page 8-7

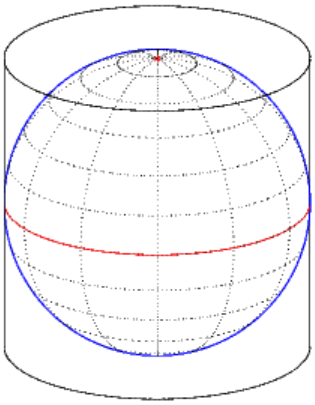
Unwrapping the Sphere to a Plane

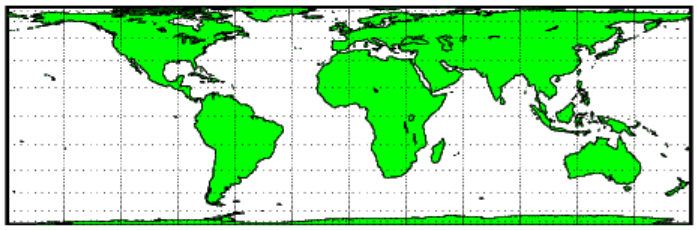
Mapmakers have developed hundreds of map projections, over several thousand years. Three large families of map projection, plus several smaller ones, are generally acknowledged. These are based on the types of geometric shapes that are used to transfer features from a sphere or spheroid to a plane. Map 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 8-9.

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

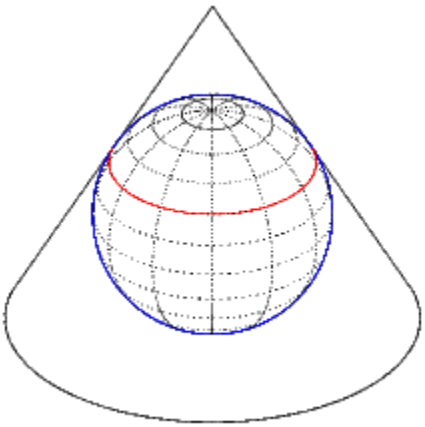
All cylindrical projections fill a rectangular plane. *Pseudocylindrical* projection outlines tend to be barrel-shaped rather than rectangular. However, they do resemble cylindrical projections, with straight and parallel latitude lines, and can have equally spaced meridians, but meridians are curves, not straight lines. Pseudocylindrical projections can be equal-area, but are not conformal or equidistant.

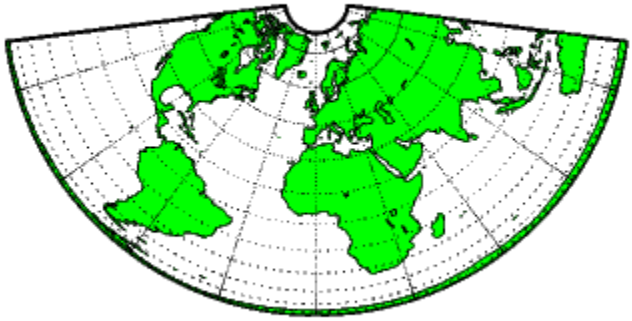
Some widely-used pseudocylindrical map projections are

- 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).



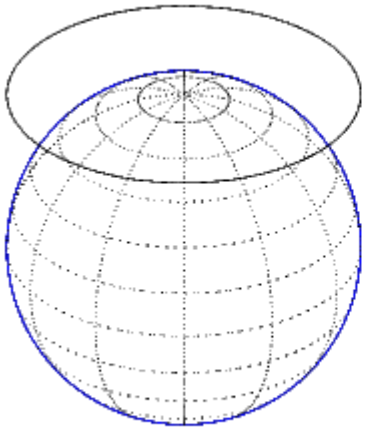


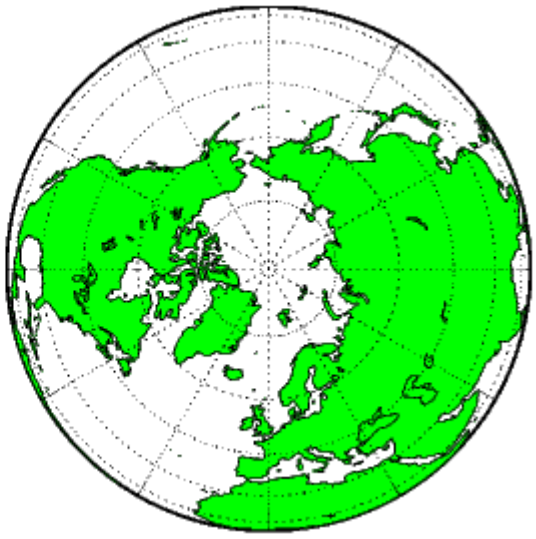
Some widely-used conic projections are

- 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

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. For an example, see “Control the Map Projection Aspect with an Orientation Vector” on page 8-11.

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.

`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 Orientation Vector

A map axes `Origin` property is a vector describing the geometry of the displayed projection. This Mapping Toolbox property is called an *orientation vector* (prior versions called it the *origin vector*). The vector takes this form:

```
orientvec = [latitude longitude orientation]
```

The latitude and longitude represent the geographic coordinates of the center point of the display from which the projection is calculated. The orientation refers to the clockwise angle from *straight up* at which the North Pole points from this center point. The default orientation vector is [0 0 0]; that is, the projection is centered on the geographic point (0°,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:

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.

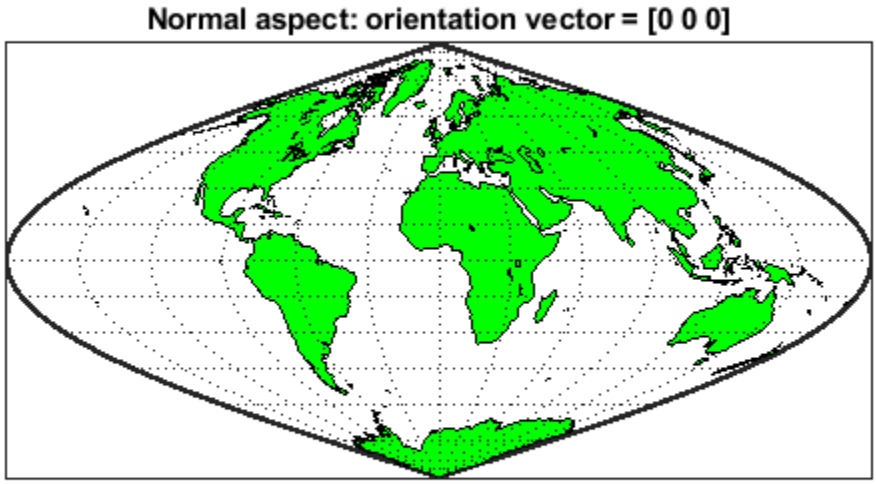
For an example that uses the orientation vector, see “Control the Map Projection Aspect with an Orientation Vector” on page 8-11.

Control the Map Projection Aspect with an Orientation Vector

The best way to gain an understanding of projection aspect is to experiment with orientation vectors. The following example uses a pseudocylindrical projection, the sinusoidal.

Create a default map axes in a sinusoidal projection, turn on the graticule, and display the coast data set as filled polygons. The continents and graticule appear in *normal* aspect.

```
figure
axesm sinusoid
framem on
gridm on
tightmap tight
load coastlines
patchm(coastlat,coastlon,'g')
title('Normal aspect: orientation vector = [0 0 0]')
```



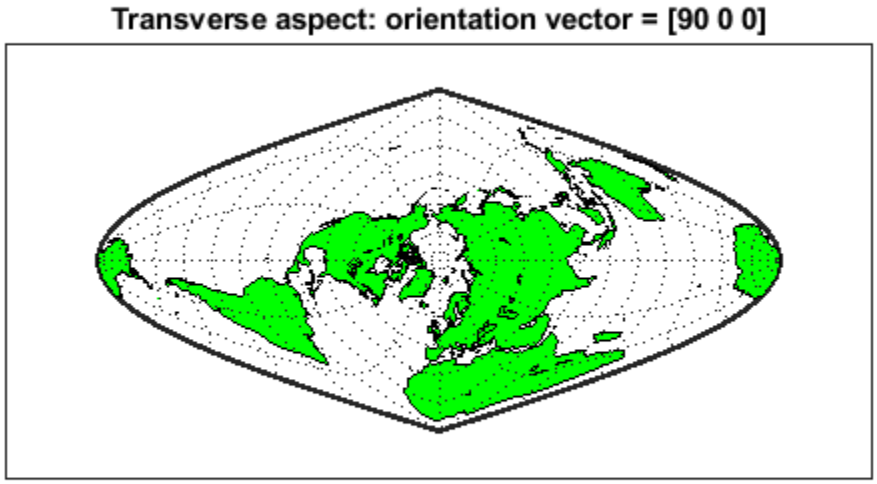
Inspect the orientation vector from the map axes. By default, the origin is set at (0°E, 0°N), oriented 0° from vertical.

```
getm(gca,'Origin')
```

```
ans = 1×3

     0     0     0
```

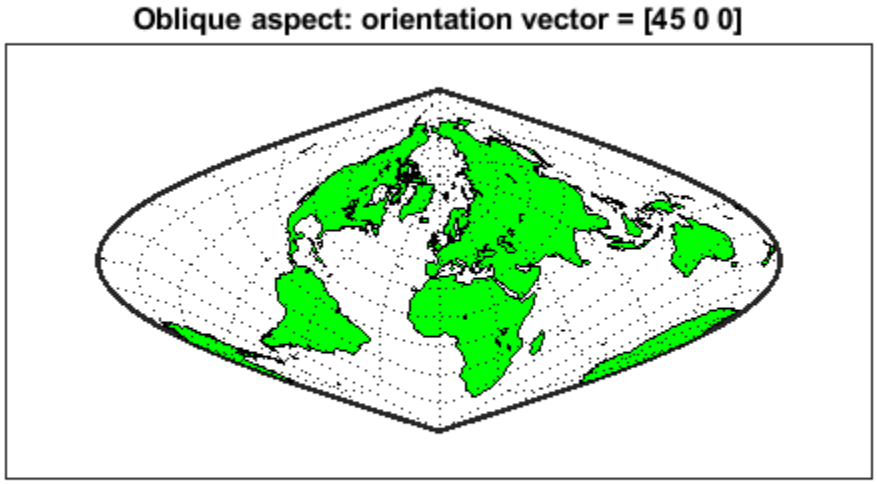
In the normal aspect, the North Pole is at the top of the image. To create a *transverse* aspect, imagine pulling the North Pole down to the center of the display, which was originally occupied by the point (0°,0°). Do this by setting the first element of `Origin` parameter to a latitude of 90°N. The shape of the frame is unaffected. This is still a sinusoidal projection.

```
setm(gca,'Origin',[90 0 0])
title('Transverse aspect: orientation vector = [90 0 0]')
```



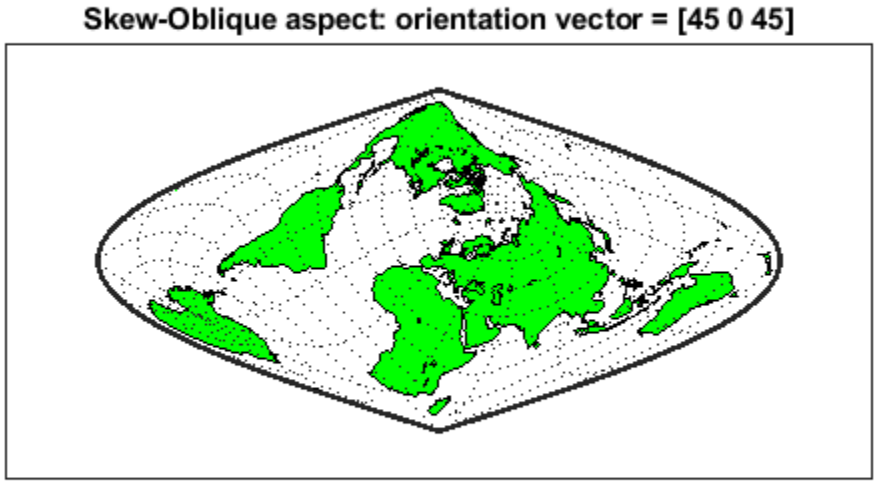
The normal and transverse aspects can be thought of as limiting conditions. Anything else is an *oblique* aspect. Conceptually, if you push the North Pole halfway back to its original position, that is, to the position originally occupied by the point (45°N, 0°E) in the normal aspect, the result is a simple oblique aspect. 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.

```
setm(gca,'Origin',[45 0 0])
title('Oblique aspect: orientation vector = [45 0 0]')
```



The previous examples of projection aspect kept the aspect orientation at 0°. If you alter the orientation, an oblique aspect becomes a *skew-oblique* orientation. 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. As in the previous example, the location (45°N,0°E) still occupies the center of the map.

```
setm(gca,'Origin',[45 0 45])
title('Skew-Oblique aspect: orientation vector = [45 0 45]')
```



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.

Any projection can be viewed in alternate aspects and this can often be 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.

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
```

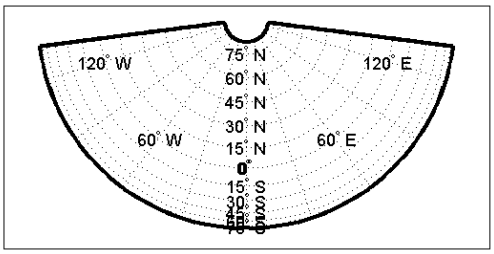
For more information on these and other map axes properties, see the reference page for `axesm`.

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

```
setm(h,'Mapprojection', 'eqaconic')
getm(h,'aspect')

ans =
normal

getm(h,'origin')

ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     2
```

```
getm(h,'mapparallels')

ans =
    15    75

getm(h,'falsenorthing')

ans =
     0

getm(h,'falseeasting')

ans =
     0

getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -86    86

getm(h,'maplonlimit')

ans =
  -135   135

getm(h,'Flatlimit')

ans =
   -86    86

getm(h,'Flonlimit')

ans =
  -135   135
```

The `eqaconic` projection has two standard parallels, at 15° and 75°. It also has reduced longitude limits (covering 270° rather than 360°). The resulting `eqaconic` graticule is shown below.

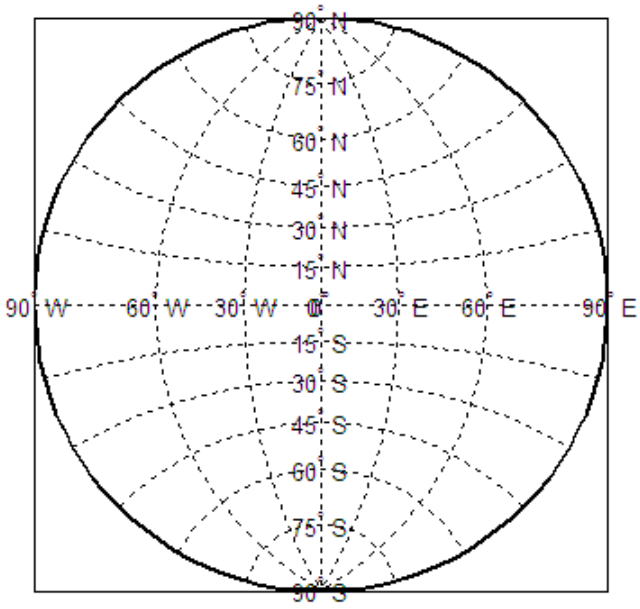


4 Now set the projection type to Stereographic (`'stereo'`) and examine the same properties as you did for the previous projections:

```
setm(h,'Mapprojection','stereo')
setm(gca,'MLabelParallel',0,'PLabelMeridian',0)
getm(h,'aspect')

ans =
normal

getm(h,'origin')

ans =
     0     0     0

getm(h,'scalefactor')

ans =
     1

getm(h,'nparallels')

ans =
     0

getm(h,'mapparallels')

ans =
     []

getm(h,'falsenorthing')

ans =
     0

getm(h,'falseeasting')

ans =
     0

getm(h,'zone')

ans =
     []

getm(h,'maplatlimit')

ans =
   -90    90

getm(h,'maplonlimit')

ans =
  -180   180

getm(h,'Flatlimit')

ans =
   -Inf    90

getm(h,'Flonlimit')
```

```
ans =
  -180   180
```

The stereographic projection, being azimuthal, does not have standard parallels, so none are indicated. The map limits do not change from the previous projection. The map figure is shown below.



Visualize Spatial Error Using Tissot Indicatrices

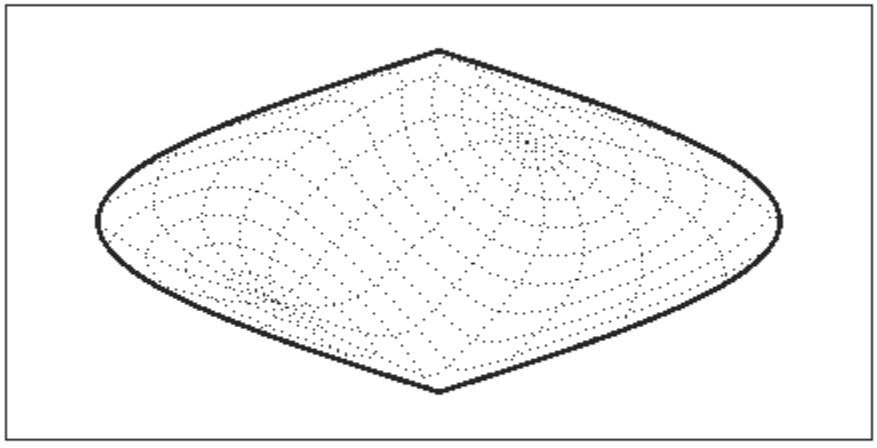
A standard method of visualizing the distortions introduced by the map projection is to display small circles at regular intervals across the globe. After projection, the small circles appear as ellipses of various sizes, elongations, and orientations. The sizes and shapes of the ellipses reflect the 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.

Visualize Projection Distortions using Tissot Indicatrices

This example shows how to add Tissot indicatrices to a map display.

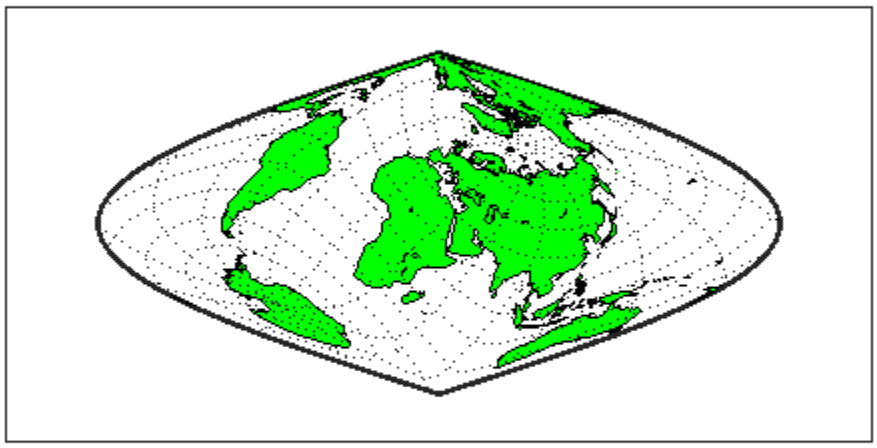
Set up a Sinusoidal projection in a skewed aspect, plotting the graticule.

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



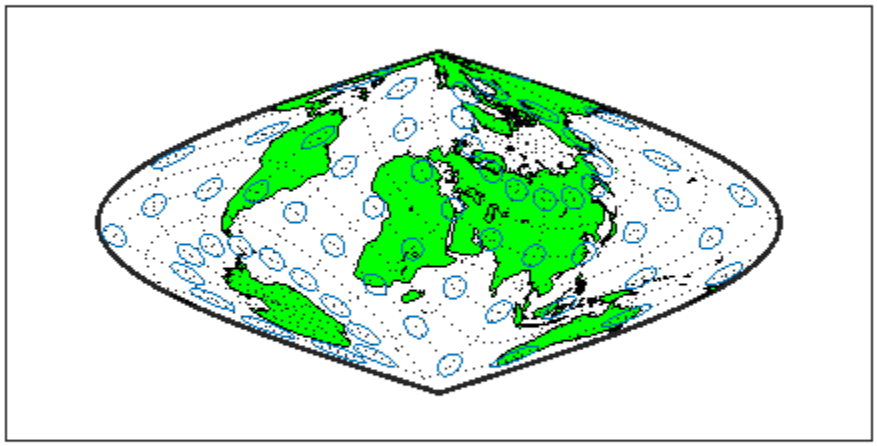
Load the coast data set and plot it as green patches.

```
load coastlines
patchm(coastlat,coastlon,'g')
```



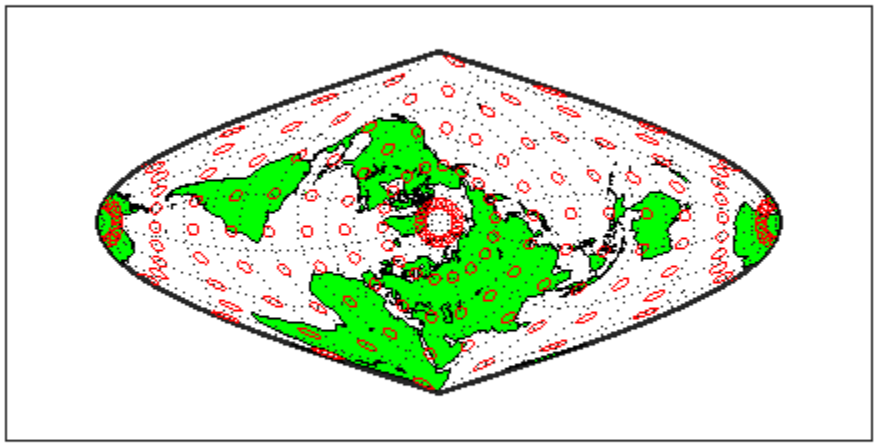
Plot the default Tissot diagram. Notice that the circles vary considerably in shape. This indicates that the Sinusoidal projection is not conformal. Despite the distortions, however the circles all cover equal amounts of area on the map because the projection has the equal-area property. Default Tissot diagrams are drawn with blue unfilled 100-point circles spaced 30 degrees apart in both directions. The default circle radius is 1/10 of the current radius of the reference ellipsoid (by default that radius is 1).

```
tissot
```



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. In the result, note that the 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.

```
clmo tissot
setm(gca,'Origin',[90 0 45])
tissot([20 20 .05 20],'Color','r')
```



See Also

`distortcalc` | `mdistort` | `tissot`

More About

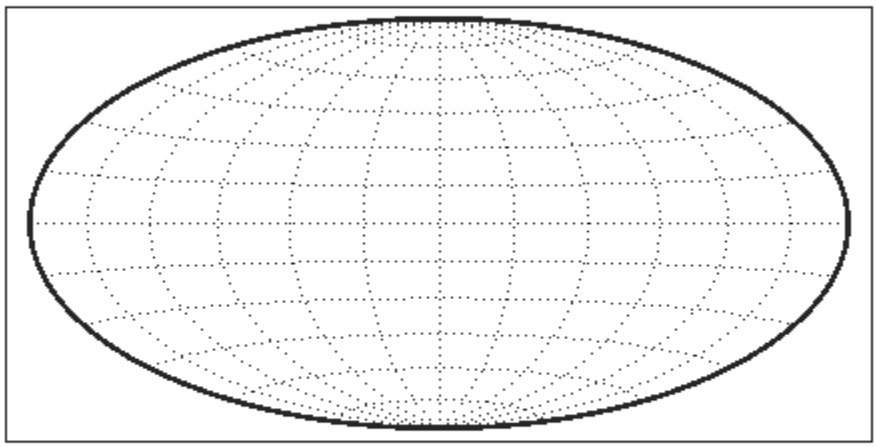
- “Quantitative Properties of Map Projections” on page 8-4
- “Visualize Projection Distortions Using Isolines” on page 8-26

Visualize Projection Distortions Using Isolines

This example shows how to visualize map projection distortions using isolines (contour lines). Since distortions are rather orderly and vary continuously, they are well-suited for isolines. 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).

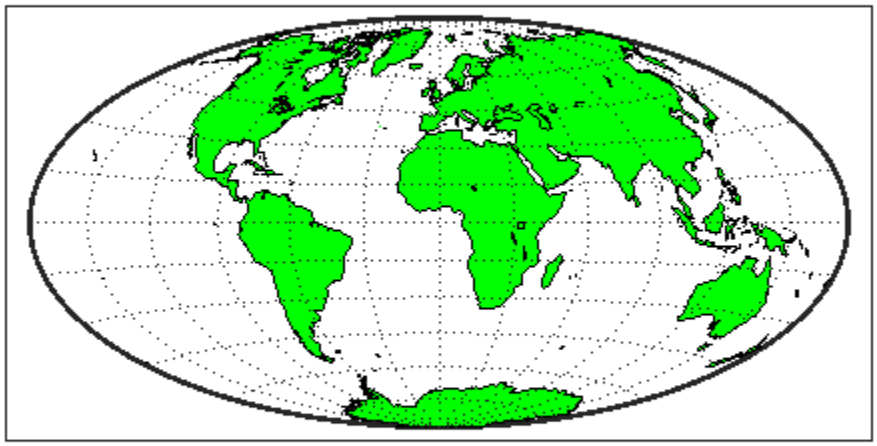
Create a Hammer projection map axes in normal aspect and plot a graticule and frame.

```
figure
axesm('MapProjection','hammer','Grid','on','Frame','on')
```



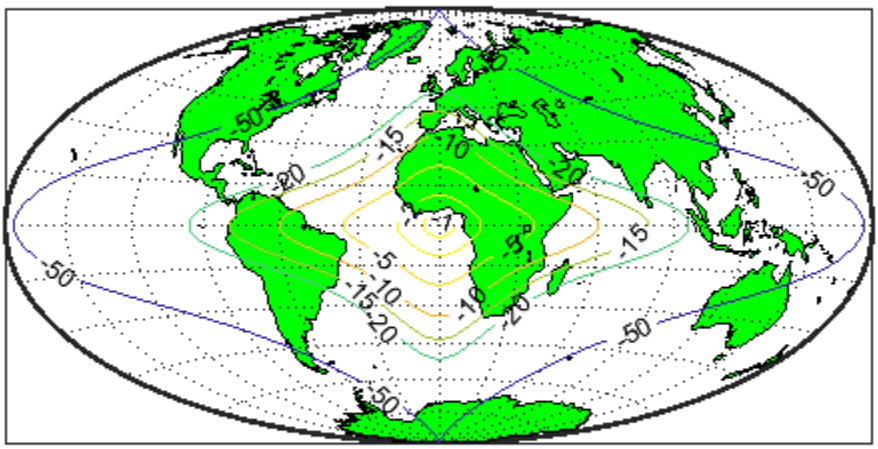
Load the coast data set and plot it as green patches.

```
load coastlines
patchm(coastlat,coastlon,'g')
```



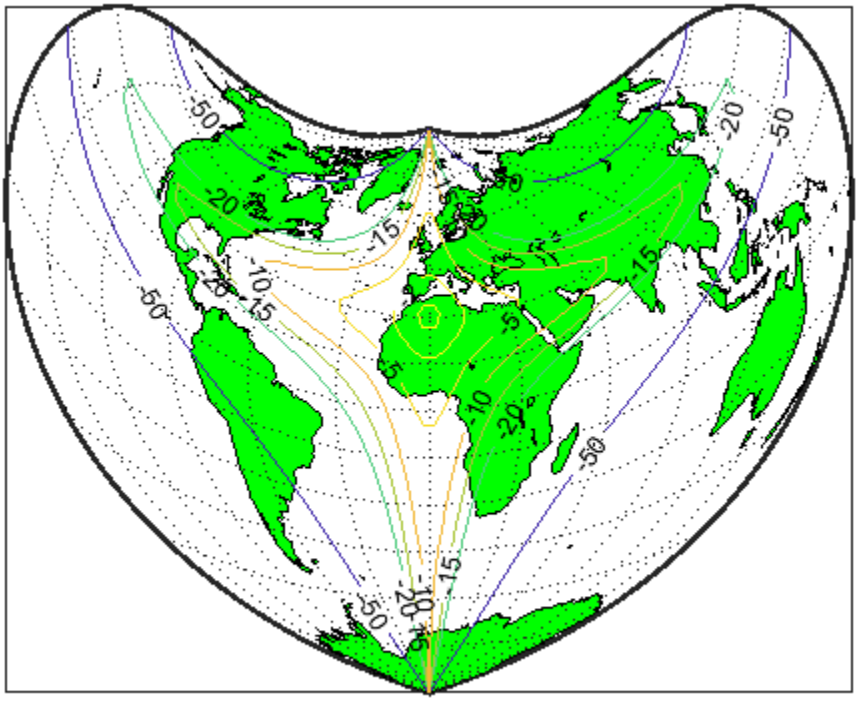
Plot contours of minimum-to-maximum scale ratios, using `mdistort` . Notice that the region of minimum distortion is centered around (0,0).

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



Repeat this diagram with a Bonne projection in a new figure window. Notice that the region of minimum distortion is centered around (30,0) which is where the single standard parallel is. You can toggle the isolines by typing `mdistort` or `mdistort off` .

```
figure
axesm('MapProjection','bonne','Grid','on','Frame','on')
patchm(coastlat,coastlon,'g')
mdistort('scaleratio')
```


-50
-20
-15
-10
-5
5
10
15
-50
-20
-15
-10
-5
5
10
15
-50
-20
-15
-10
-5
5
10
15

Quantify Map Distortions at Point Locations

The `tissot` and `mdistort` functions provide synoptic visual overviews of different forms of map projection error. Sometimes, however, you need numerical estimates of error at specific locations in order to quantify or correct for map distortions. This is useful, for example, if you are sampling environmental data on a uniform basis across a map, and want to know precisely how much area is associated with each sample point, a statistic that will vary by location and be projection dependent. Once you have this information, you can adjust environmental density and other statistics you collect for areal variations induced by the map projection.

A Mapping Toolbox function returns location-specific map error statistics from the current projection or an `mstruct`. The `distortcalc` function computes the same distortion statistics as `mdistort` does, but for specified locations provided as arguments. You provide the latitude-longitude locations one at a time or in vectors. The general form is

```
[areascale,angdef,maxscale,minscale,merscale,parscale] = ...
    distortcalc(mstruct,lat,long)
```

However, if you are evaluating the current map figure, omit the `mstruct`. You need not specify any return values following the last one of interest to you.

Use 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 land areas data set.

1 Read the North and South America polygon:

```
Americas = shaperead('landareas','UseGeoCoords',true, ...
    'Selector', {@(name) ...
    strcmpi(name,{'north and south america'}),'Name'});
```

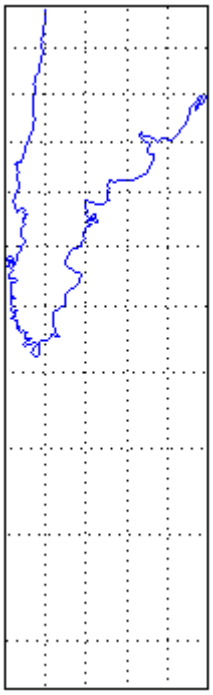
2 Set the spatial extent (map limits) to contain the southern part of South America and also include an area closer to the South Pole:

```
mlatlim = [-72.0 -20.0];
mlonlim = [-75.0 -50.0];
[alat, alon] = maptriml([Americas.Lat], ...
    [Americas.Lon], mlatlim, mlonlim);
```

3 Create a Mercator cylindrical conformal projection using these limits, specify a five-degree graticule, and then plot the outline for reference:

```
figure;
axesm('MapProjection','mercator','grid','on', ...
    'MapLatLimit',mlatlim,'MapLonLimit',mlonlim,...
    'MLineLocation',5, 'PLineLocation',5)
plotm(alat,alon,'b')
```

The map looks like this:



4 Sample every tenth point of the patch outline for analysis:

```
alats = alat(1:10:numel(alat));
alons = alon(1:10:numel(alat));
```

5 Compute the area distortions (the first value returned by `distortcalc`) at the sample points:

```
adistort = distortcalc(alats, alons);
```

6 Find the range of area distortion across Argentina (percent of a unit area on, in this case, the equator):

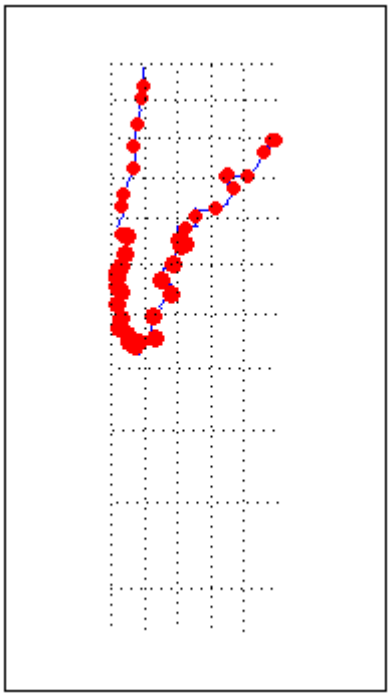
```
adistortmm = [min(adistort) max(adistort)]

adistortmm =
    1.1790    2.7716
```

As Argentina occupies mid southern latitudes, its area on a Mercator map is overstated, and the errors vary noticeably from north to south.

7 Remove any `NaN`s from the coordinate arrays and plot symbols to represent the relative distortions as proportional circles, using `scatterm`:

```
nanIndex = isnan(adistort);
alats(nanIndex) = [];
alons(nanIndex) = [];
adistort(nanIndex)  = [];
scatterm(alats,alons,20*adistort,'red','filled')
```

The resulting map is shown below:



8 The degree of area overstatement would be considerably larger if it extended farther toward the pole. To see how much larger, get the area distortion for 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`.

Rotational Transformations on the Globe

In “The Orientation Vector” on page 8-9, 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° minus its transformed latitude (also known as a *colatitude*). The point antipodal to your town would become the *South Pole*, at -90°. Its distance from your hometown is 90°-(-90°), or 180°, as expected. Points 90° distant from your hometown all have a transformed latitude of 0°, and thus make up the transformed *equator*. Transformed longitudes correspond to their respective great circle azimuths from your home town.

Reorient 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°N,102°W). You have a brother in Tulsa (36.2°N,96°W) and a sister in New Orleans (30°N,90°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

[dist2neworl az2neworl] = distance(midl_lat,midl_lon,...
```

```
                                    newo_lat,newo_lon)

dist2neworl =
  10.4727

az2neworl =
  97.8644
```

Tulsa is about 6.5 degrees distant, New Orleans about 10.5 degrees distant.

3 Compute the absolute difference in azimuth, a fact you will use later.

```
azdif = abs(az2tuls-az2neworl)

azdif =
   49.7258
```

4 Today, you feel on top of the world, so make Midland, Texas, the *north pole* of a transformed coordinate system. To do this, first determine the origin required to put Midland at the pole using `newpole`:

```
origin = newpole(midl_lat,midl_lon)

origin =
    58    78     0
```

The origin of the new coordinate system is (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°. 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 round-off error and trigonometric functions.

For further information, see the reference pages for `rotatem`, `newpole`, `putpole`, `neworig`, and `org2pol`.

Reorient Gridded Data

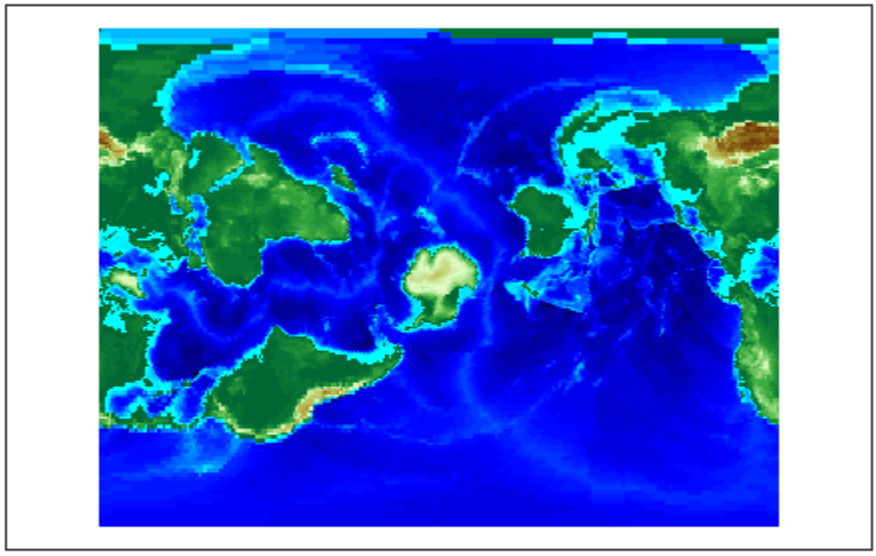
This example shows how to transform a regular data grid into a new one with its data rearranged to correspond to a new coordinate system using the `neworig` function. You can transform coordinate systems of data grids as well as vector data. When regular data grids are manipulated in this manner, distance and azimuth calculations with the map variable become row and column operations.

Load elevation raster data and a geographic cells reference object. Transform the data set to a new coordinate system in which a point in Sri Lanka is the north pole. Reorient the data grid by using the `neworig` function. Note that the result, `[Z,lat,lon]`, is a geolocated data grid, not a regular data grid like the original data.

```
load topo60c
origin = newpole(7,80);
[Z,lat,lon] = neworig(topo60c,topo60cR,origin);
```

Display the new map, in normal aspect, as its orientation vector shows. Note that every cell in the first row of the new grid is 0 to 1 degrees distant from the point new origin. Every cell in its second row is 1 to 2 degrees distant, and so on. In addition, every cell in a particular column has the same great circle azimuth from the new origin.

```
axesm miller
latlim = [ -90  90];
lonlim = [-180 180];
gratsize = [90 180];
[lat,lon] = meshgrat(latlim,lonlim,gratsize);
surfm(lat,lon,Z);
demcmap(topo60c)
```



```
mstruct = getm(gca);
mstruct.origin
```

```
ans = 1×3

     0     0     0
```

Create a UTM Map

The Universal Transverse Mercator (UTM) system divides the world into a regular, nonoverlapping grid of quadrangles called *zones*. Each zone is 8-by-6 degrees in extent and uses a transverse Mercator projection that is designed to limit distortion. UTM zones are identified using a grid reference in the form of a number followed by a letter, for example `31T`. Each UTM zone has a false northing and a false easting. These are offsets that enable each zone to have positive coordinates in both directions. All UTM zones have a false easting of 500,000 meters. All zones in the northern hemisphere have a false northing of 0 meters, and all zones in the southern hemisphere have a false northing of 10,000,000 meters.

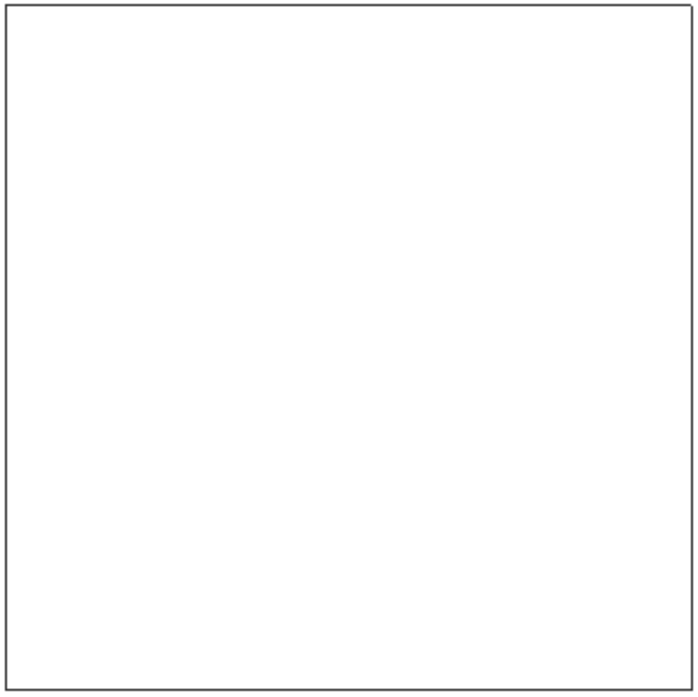
The UTM system is defined between 80 degrees south and 84 degrees north. Beyond these limits, use the Universal Polar Stereographic (UPS) system instead. The UPS system has two zones, `north` and `south`. Both UPS zones have a false northing and false easting of 2,000,000 m.

Create a UTM Map

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).

Create a UTM map axes.

```
axesm utm
```



Get the map axes properties and inspect them in the Command Window or with the Variables editor. Note that the default zone is 31N. This is selected because the map origin defaults to `[0 3 0]` ,

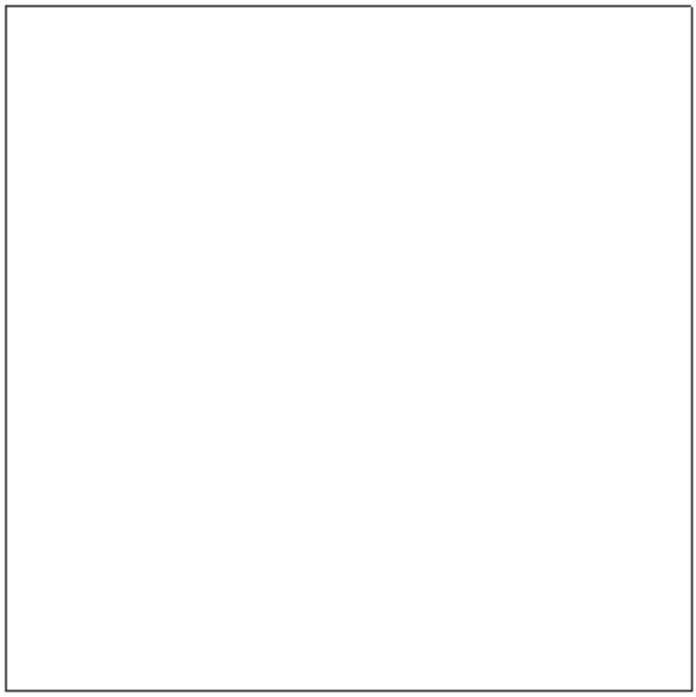
which is on the equator and at a longitude of 3° E. This is the center longitude of zone 31N, which has a latitude limit of `[0 8]` , and a longitude limit of `[0 6]`.

```
h = getm(gca);
h.zone
```

```
ans =
'31N'
```

Change the zone to 32N, one zone to the east of the default, and inspect the other parameters again. Note that the map origin and limits are adjusted for zone 32N.

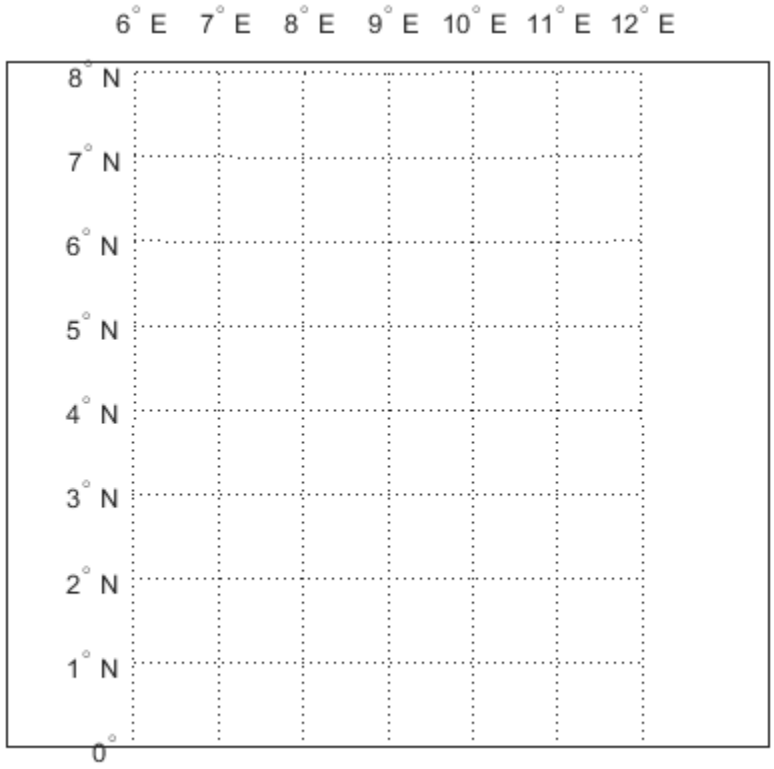
```
setm(gca,'zone','32n')
```



```
h = getm(gca);
```

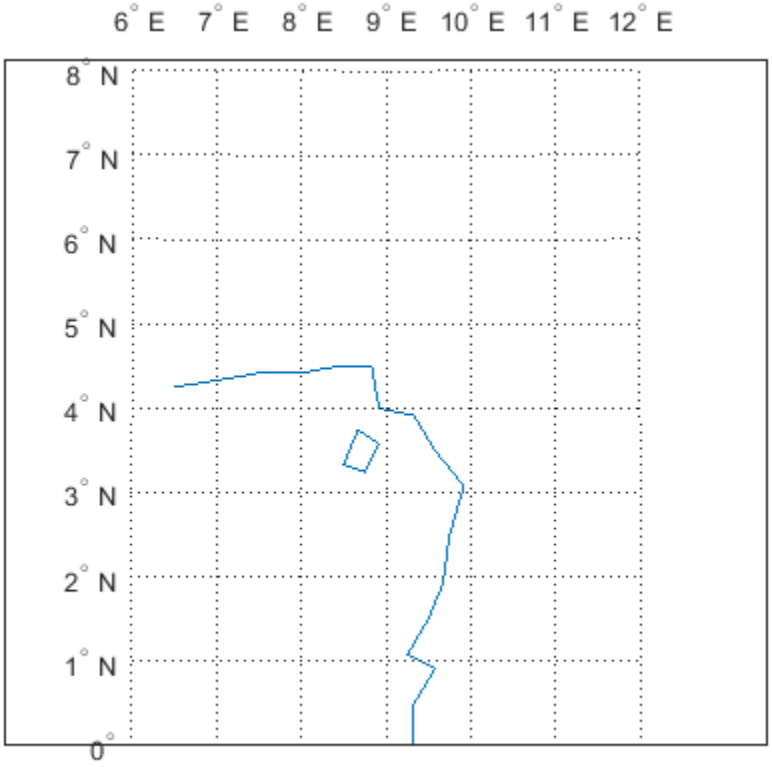
Draw the map grid and label it.

```
setm(gca,'grid','on','meridianlabel','on','parallellabel','on')
```



Load and plot the coastline data set to see a close-up of the Gulf of Guinea and Bioko Island in UTM.

```
load coastlines
plotm(coastlat,coastlon)
```



See Also

`ups` | `utm`

Set UTM Parameters Interactively

The easiest way to use the UTM projection is through a graphical user interface. You can create or modify a UTM area of interest with the `axesmui` projection control panel, and get further assistance form the `utmzoneui` control panel.

1 You can **Shift**+click in a map axes window, or type `axesmui` to display the projection control panel. Here you start from scratch:

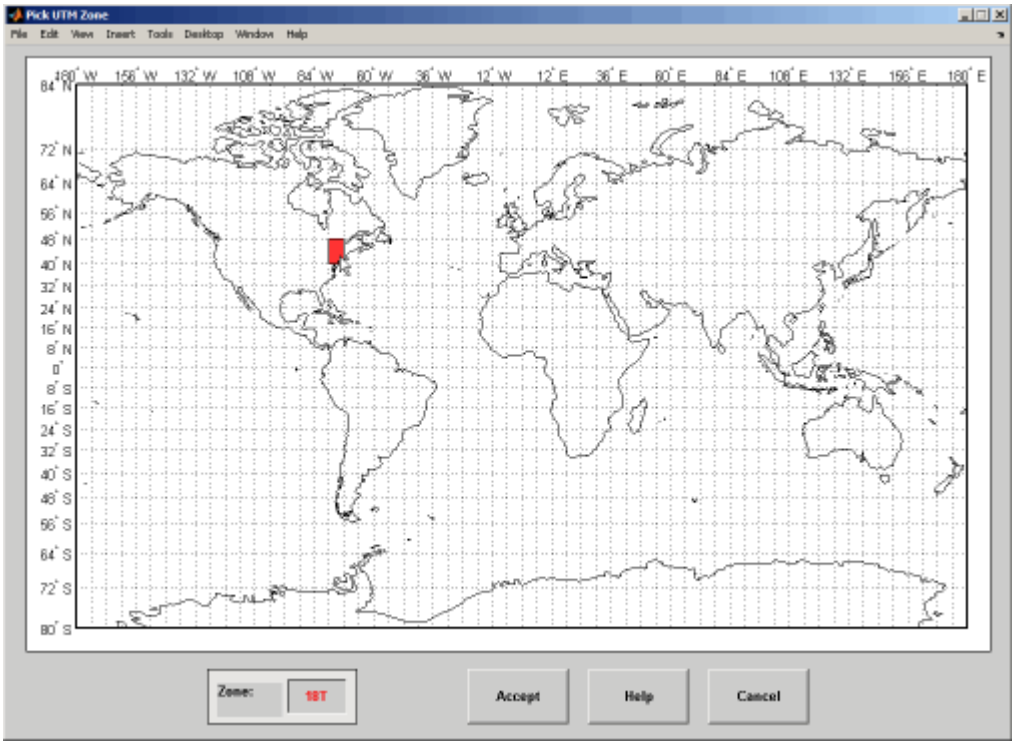
```
figure
axesm utm
axesmui
```

The **Map Projection** field is set to `cyln: Universal Transverse Mercator (UTM)`.

`Note` For UTM and UPS maps, the **Aspect** field is set to `normal` and cannot be changed. If you attempt to specify `transverse`, an error results.

---

2 Click the **Zone** button to open the `utmzoneui` panel. Click the map near your area of interest to pick the zone:



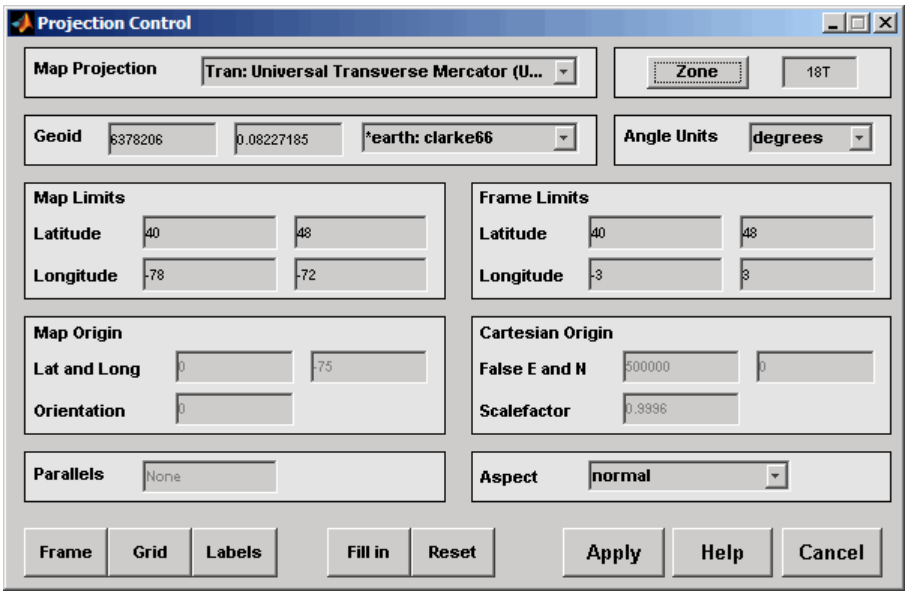
Note that while you can open the `utmzoneui` control panel from the command line, you then have to manually update the figure with the zone name it returns with a `setm` command:

```
setm(gca,'zone',ans)
```

3 Click the **Accept** button.

The `utmzoneui` panel closes, and the `zone` field is set to the one you picked. The map limits are updated accordingly, and the geoid parameters are automatically set to an appropriate ellipsoid

definition for that zone. You can override the default choice by selecting another ellipsoid from the list or by typing the parameters in the **Geoid** field.

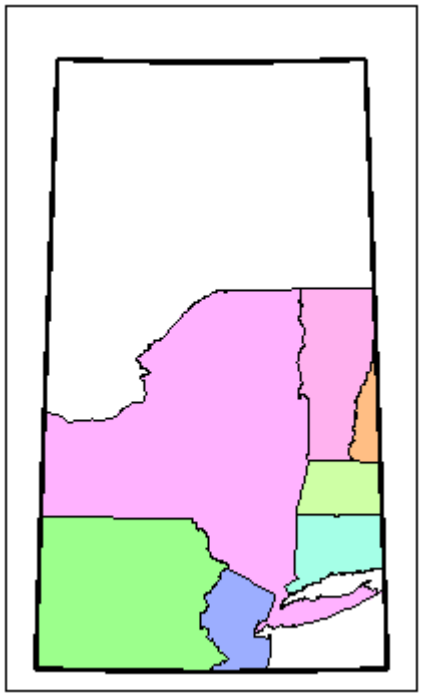


4 Click **Apply** to close the projection control panel.

The projection is then ready for projection calculations or map display commands.

5 Now view a choropleth base map from the `usstatehi` 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

mstruct = gcm;
[x,y] = projfwd(mstruct, latlim, lonlim)

x =

   1.0e+06 *

   -1.5029   -0.7829


y =

   1.0e+06 *

    3.7403    4.5369
```

Work in UTM Without a Displayed Map

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 this example, you generate UTM coordinate data for a location in New York City, using that point to define the projection itself.

Define a location in New York City. Obtain the UTM zone for this point.

```
lat = 40.7;
lon = -74.0;
z = utmzone(lat,lon)
```

```
z =
'18T'
```

Get the suggested ellipsoid vector and name for this zone.

```
[ellipsoid,estr] = utmgeoid(z)
```

```
ellipsoid = 1×2
10^6 ×

    6.3782    0.0000


estr =
'clarke66'
```

Set up the UTM coordinate system based on this information.

```
utmstruct = defaultm('utm');
utmstruct.zone = z;
utmstruct.geoid = ellipsoid;
utmstruct = defaultm(utmstruct);
```

Transform the coordinates, without a map display.

```
[x,y] = projfwd(utmstruct,lat,lon)
```

```
x = 5.8448e+05

y = 4.5057e+06
```

Compute the zone limits (latitude and longitude limits) for a specified zone by using the `utmzone` function. You can also call `utmzone` recursively to obtain the limits of the UTM zone within which a point location falls.

```
utmzone('18T')
```

```
ans = 1×4

    40    48   -78   -72


[zonelats,zonelons] = utmzone(utmzone(40.7,-74.0))

zonelats = 1×2
```

```
    40    48

zonelons = 1×2

   -78   -72
```

Alternatively, set up a UTM coordinate system using a `projcrs` object. To create the object, specify an EPSG code. For information about EPSG codes, see http://www.epsg.org/. Verify that the `projcrs` object has the correct name and ellipsoid. Then, transform the coordinates.

```
p = projcrs(26718);
p.Name

ans =
"NAD27 / UTM zone 18N"

p.GeographicCRS.Spheroid.Name

ans =
'Clarke 1866'

[xp,yp] = projfwd(p,lat,lon)

xp = 5.8448e+05

yp = 4.5057e+06
```

See Also

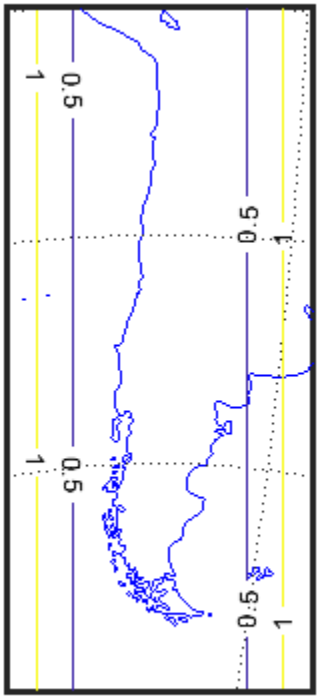
`defaultm` | `projcrs` | `projfwd` | `utmgeoid` | `utmzone`

Use the Transverse Aspect to Map Across UTM Zones

To display areas that extend across more than one UTM zone, use the Mercator projection in a transverse aspect. UTM is a zone-based coordinate system and 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. This example shows a transverse Mercator projection appropriate to Chile. In the example, note how the projection's line of zero distortion is aligned with the predominantly north-south axis of the country. 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. To place the zero distortion line exactly on the midline of the country, use better estimates of the orientation vector's central meridian and orientation angle.

Setup a map axes with a transverse aspect and display a map of Chile. Calculate the map distortion.

```
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
```



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

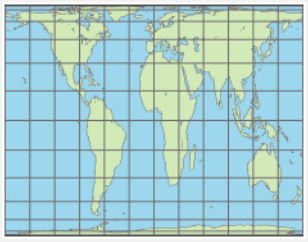
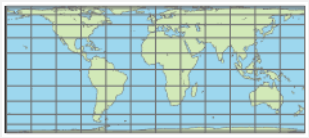
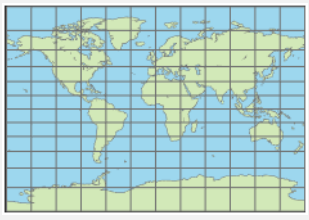
Use projections to display latitude-longitude coordinate data on maps. Choose a projection method by considering these criteria:

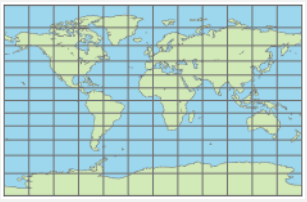
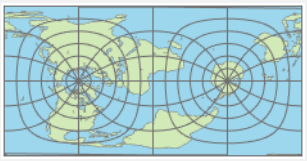
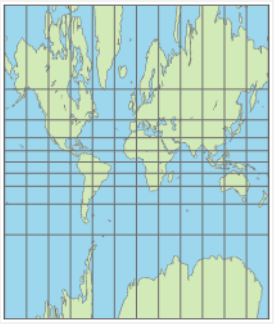
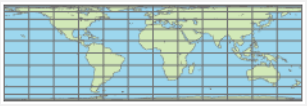
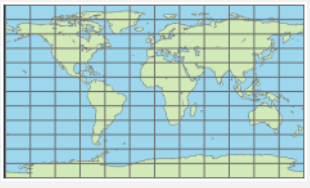
- Family – Choose a cylindrical, conic, or azimuthal projection based on your purpose and region of interest. For more information, see “The Three Main Families of Map Projections” on page 8-5.
- Properties – Choose a projection based on the properties you want to preserve, such as shape, distance, direction, scale, and area. For more information, see “Quantitative Properties of Map Projections” on page 8-4.
- Distortion – Choose a projection based on the distortion you want to minimize or eliminate. For more information, see “Map Projections and Distortions” on page 8-2.

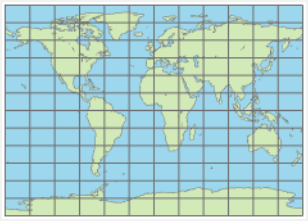
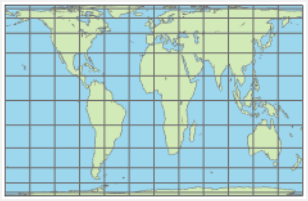
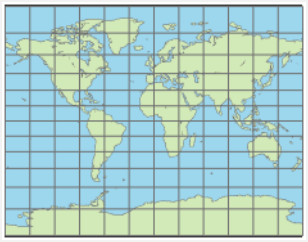
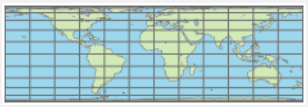
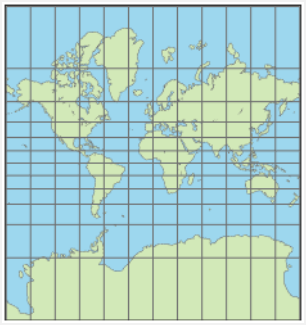
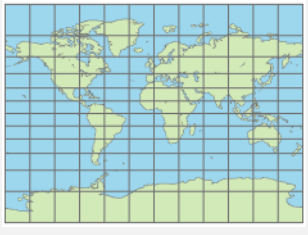
These tables show the map projections you can use with map projection structures and map axes. For more information about map projection structures, see `defaultm`. For more information about map axes, see `axesm`.

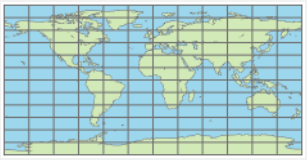
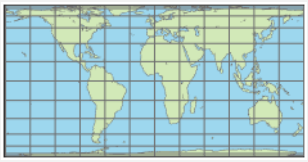
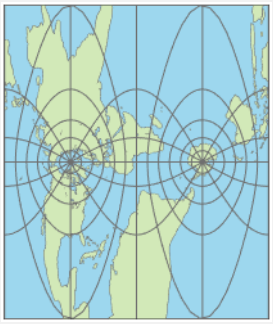
Note Most projection IDs are also functions on the MATLAB search path. These functions are only used in the implementation of functions such as `defaultm` and `axesm`, and therefore their syntaxes are not documented.

Cylindrical Projections

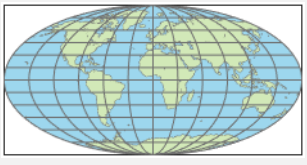
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Balthasart	`balthsrt`		x	x	—	
Behrmann	`behrmann`		x	x	—	
Bolshoi Sovietskii Atlas Mira	`bsam`	x	x	x	—	

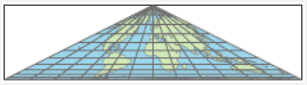
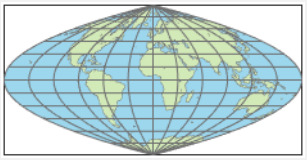
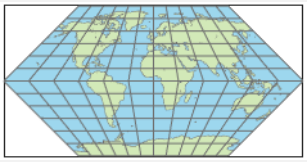
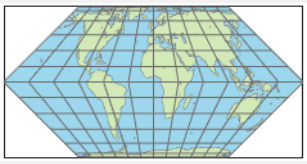
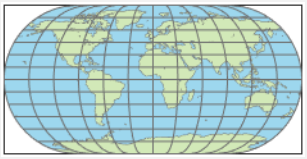
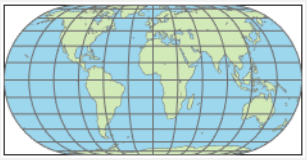
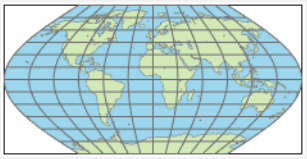
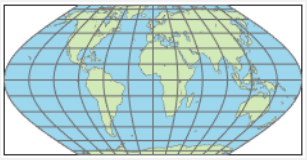
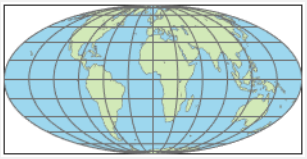
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Braun Perspective	`braun`	x	x	x	—	
Cassini	`cassini`	x	x		—	
Cassini – Standard	`cassinistd`	x	x	x	—	
Central	`ccylin`	x	x	x	—	
Equal-Area Cylindrical	`eqacylin`		x	x	—	
Equidistant Cylindrical	`eqdcylin`	x	x		—	

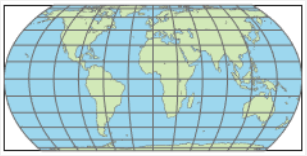
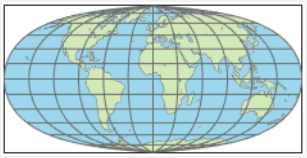
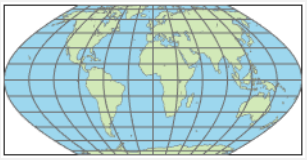
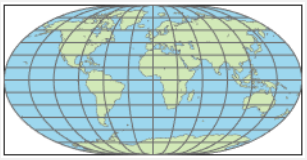
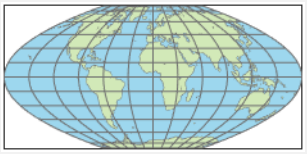
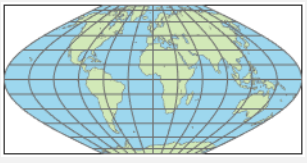
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Gall Isographic	`giso`	x	x		—	
Gall Orthographic	`gortho`		x	x	—	
Gall Stereographic	`gstereo`	x	x	x	—	
Lambert Equal-Area Cylindrical	`lambcyln`		x	x	—	
Mercator	`mercator`	x		x	Rhumb lines are straight.	
Miller	`miller`	x	x	x	—	

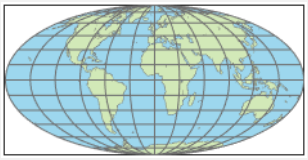
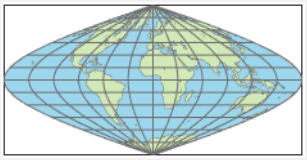
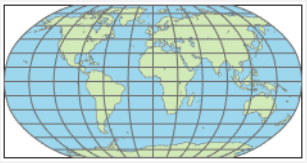
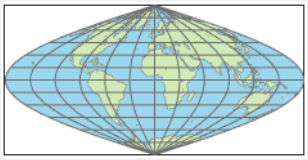
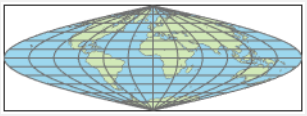
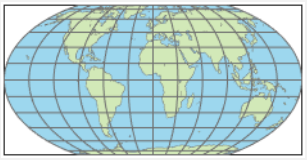
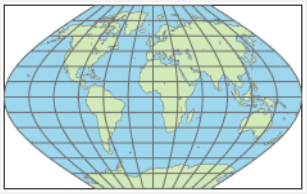
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Plate Carrée	`pcarree`	x	x		—	
Transverse Mercator	`tranmerc`	x		x	—	
Trystan Edwards	`trystan`		x	x	—	
Universal Transverse Mercator (UTM)	`utm`	x		x	—	—
Wetch	`wetch`	x	x	x	—	

Pseudocylindrical Projections

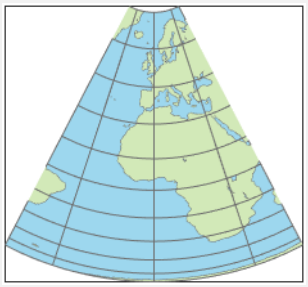
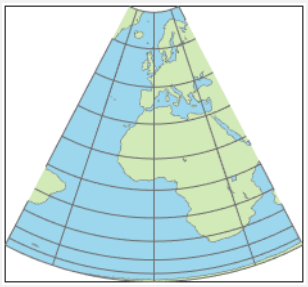
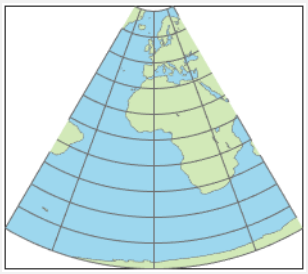
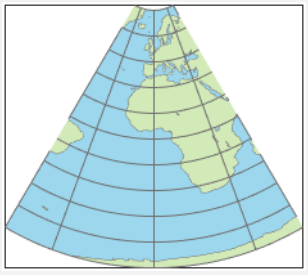
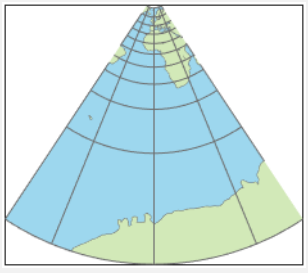
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Apianus II	`apianus`	x	x	x	—	

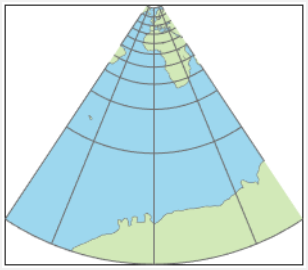
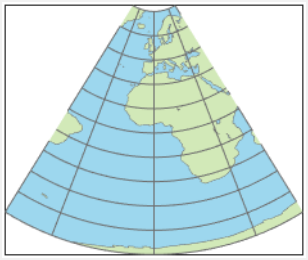
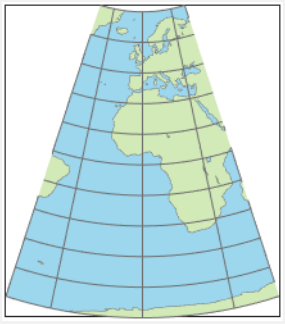
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Collignon	`collig`		x	x	—	
Craster Parabolic	`craster`		x	x	—	
Eckert I	`eckert1`	x	x	x	—	
Eckert II	`eckert2`		x	x	—	
Eckert III	`eckert3`	x	x	x	—	
Eckert IV	`eckert4`		x	x	—	
Eckert V	`eckert5`	x	x	x	—	
Eckert VI	`eckert6`		x	x	—	
Fournier	`fournier`		x	x	—	

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Goode Homolosine	`goode`		x	x	—	
Hatano Asymmetrical Equal-Area	`hatano`		x	x	—	
Kavraisky V	`kavrsky5`		x	x	—	
Kavraisky VI	`kavrsky6`		x	x	—	
Loximuthal	`loximuth`	x	x	x	Rhumb lines from the central point are straight, true to scale, and correct in azimuth.	
McBryde-Thomas Flat-Polar Parabolic	`flatplrp`		x	x	—	
McBryde-Thomas Flat-Polar Quartic	`flatplrq`		x	x	—	
McBryde-Thomas Flat-Polar Sinusoidal	`flatplrs`		x	x	—	

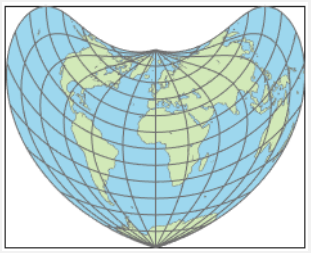
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Mollweide	`mollweid`		x	x	—	
Putnins P5	`putnins5`	x	x	x	—	
Quartic Authalic	`quartic`		x	x	—	
Robinson	`robinson`	x	x	x	—	
Sinusoidal	`sinusoid`		x	x	—	
Tissot Modified Sinusoidal	`modsine`		x	x	—	
Wagner IV	`wagner4`		x	x	—	
Winkel 1	`winkel`	x	x	x	—	

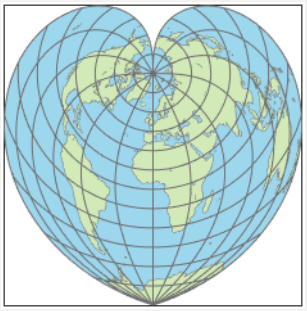
Conic Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Albers Equal-Area Conic	`eqaconic`		x	x	—	
Albers Equal-Area Conic – Standard	`eqaconicstd`		x	x	—	
Equidistant Conic	`eqdconic`	x	x		—	
Equidistant Conic – Standard	`eqdconicstd`	x	x		—	
Lambert Conformal Conic	`lambert`	x		x	—	

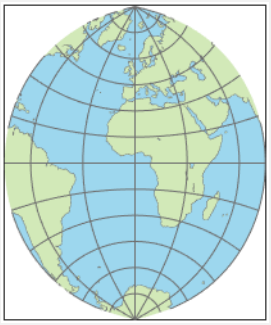
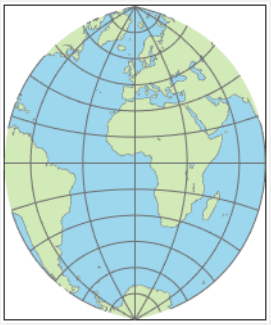
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Lambert Conformal Conic – Standard	`lambertstd`	x		x	—	
Murdoch I Conic	`murdoch1`	x	x		The total area is correct.	
Murdoch III Minimum Error Conic	`murdoch3`	x	x		The total area is correct.	

Pseudoconic Projections

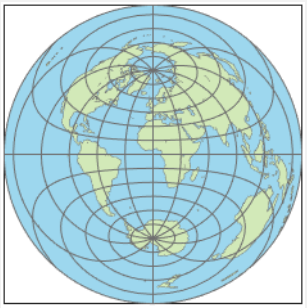
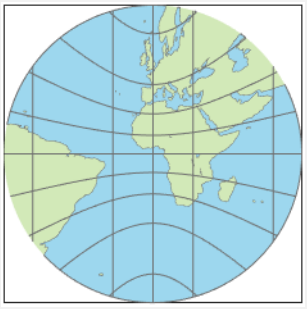
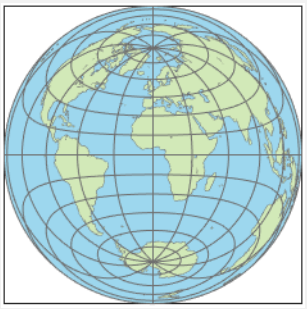
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Bonne	`bonne`		x	x	—	

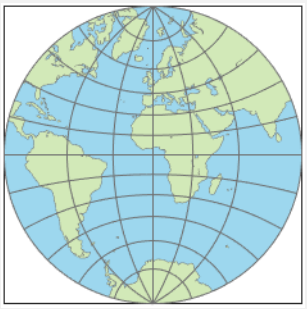
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Werner	`werner`		x	x	—	

Polyconic Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Polyconic	`polycon`	x	x	x	—	
Polyconic – Standard	`polyconstd`	x	x	x	—	
Van Der Grinten I	`vgrint1`	x	x	x	—	

Azimuthal Projections

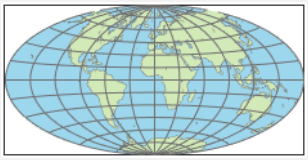
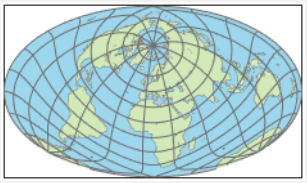
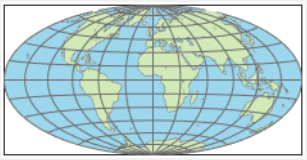
Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Breusing Harmonic Mean	`breusing`	x	x	x	—	
Equidistant Azimuthal	`eqdazim`	x	x		—	
Gnomonic	`gnomonic`	x	x	x	Great circles appear as straight lines.	
Lambert Azimuthal Equal-Area	`eqaazim`		x	x	—	

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Orthographic	`ortho`	x	x	x	—	
Stereographic	`stereo`	x		x	Great and small circles appear as either straight lines or circular arcs.	
Universal Polar Stereographic (UPS)	`ups`	x		x	Great and small circles appear as either straight lines or circular arcs.	—
Vertical Perspective Azimuthal	`vperspec`	x	x	x	—	

Pseudoazimuthal Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Wiechel	`wiechel`		x	x	—	

Modified Azimuthal Projections

Projection Name	Projection ID	Equal-Area	Conformal	Equidistant	Special Features	Example
Aitoff	`aitoff`	x	x	x	—	
Briesemeister	`bries`		x	x	—	
Hammer	`hammer`		x	x	—	

See Also

`axesm` | `defaultm` | `geoshow`

Transform Coordinates to a Different Projected CRS

If you directly compare data sets with different projected coordinate reference systems (CRSs), then the results are inaccurate. Therefore, before comparing data sets, first verify that the CRSs are the same. If different projected CRSs have the same underlying geographic CRS, then you can transform the coordinates from one projected CRS to the other. Once the data sets are referenced to the same projected CRS, you can compare them.

To transform projected *x*-*y* coordinates to a different projected CRS, first unproject the *x*-*y* coordinates to latitude-longitude coordinates by using the `projinv` function. Then, project the latitude-longitude coordinates to *x*-*y* coordinates in a different projected CRS by using the `projfwd` function.

For example, import a shapefile containing the *x*- and *y*-coordinates of roads in Boston. Also import information about the shapefile as a structure. Find the projected CRS for the coordinates by accessing the `CoordinateReferenceSystem` field of that structure.

```
s = shaperead('boston_roads.shp');
x1 = [s.X];
y1 = [s.Y];
info = shapeinfo('boston_roads.shp');
p1 = info.CoordinateReferenceSystem;
```

Unproject the *x*-*y* coordinates and return latitude-longitude coordinates.

```
[lat,lon] = projinv(p1,x1,y1);
```

Select a new projected CRS for the target projection. For this example, create a `projcrs` object for UTM zone 19N. Verify that both projected CRSs have the same geographic CRS. If the geographic CRSs are different, then the projected coordinates may be inaccurate. You can find the geographic CRS by querying the `GeographicCRS` property of the `projcrs` object.

```
p2 = projcrs(26919);
p2.GeographicCRS.Name
```

```
ans =
"NAD83"
```

```
p1.GeographicCRS.Name
```

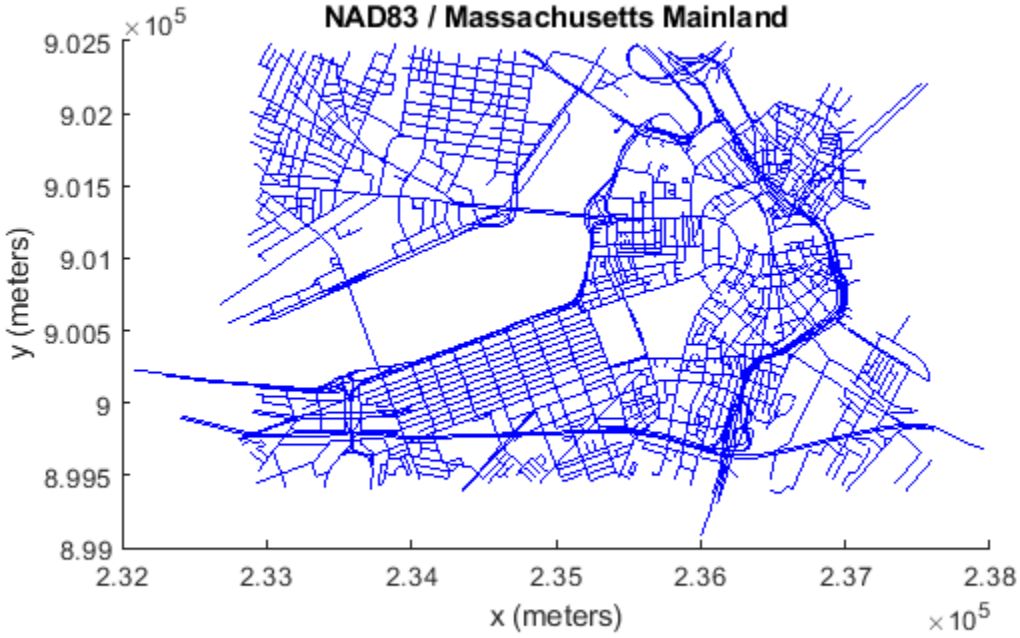
```
ans =
"NAD83"
```

Project the latitude-longitude coordinates to *x*-*y* coordinates by specifying the `projcrs` object you created.

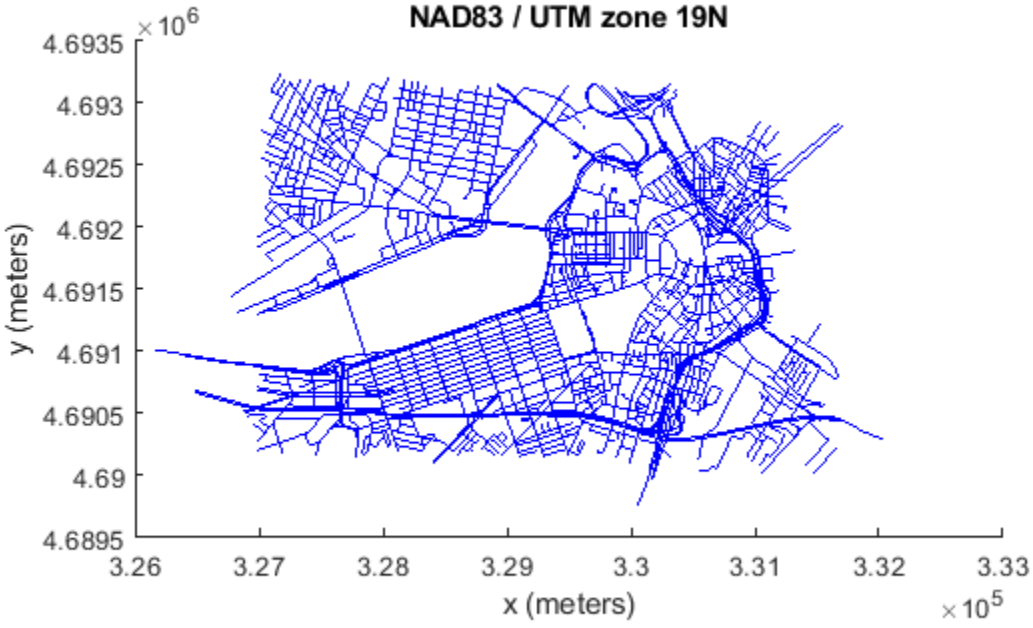
```
[x2,y2] = projfwd(p2,lat,lon);
```

Compare the original *x*-*y* coordinates with the new *x*-*y* coordinates by displaying them. Add labels and a title to each figure.

```
figure
mapshow(x1,y1)
xlabel('x (meters)')
ylabel('y (meters)')
title(p1.Name)
```




```
figure
mapshow(x2,y2)
xlabel('x (meters)')
ylabel('y (meters)')
title(p2.Name)
```



The visualizations are similar, but the coordinates displayed along the axis rulers correspond to different projected CRSs.

See Also

`geocrs` | `projcrs` | `projfwd` | `projinv` | `shaperead`

More About

- “Project and Display Raster Data” on page 8-65

Project and Display Raster Data

To project or unproject regularly-spaced raster data that is associated with a geographic or map reference object, you must first create a coordinate grid that matches the size of the raster. Use different grid creation functions depending on which way you are projecting. When you project latitude-longitude coordinates to *x*-*y* coordinates, create a grid using the `meshgrat` function. When you unproject *x*-*y* coordinates to latitude-longitude coordinates, create a grid using the `meshgrid` and `intrinsicToWorld` functions.

After transforming the raster data, you can display it on a map using visualization functions such as `mapshow` and `geoshow`. Use `mapshow` for projected *x*-*y* coordinates and `geoshow` for unprojected latitude-longitude coordinates.

Project Raster Data

To project data that is associated with a geographic raster reference object, first create a grid of latitude-longitude coordinates for each point in the raster. Then, project the geographic coordinates to *x*-*y* map coordinates.

For example, import elevation raster data as an array and a geographic cells reference object. Get the latitude-longitude coordinates for each point in the raster by using the `meshgrat` function.

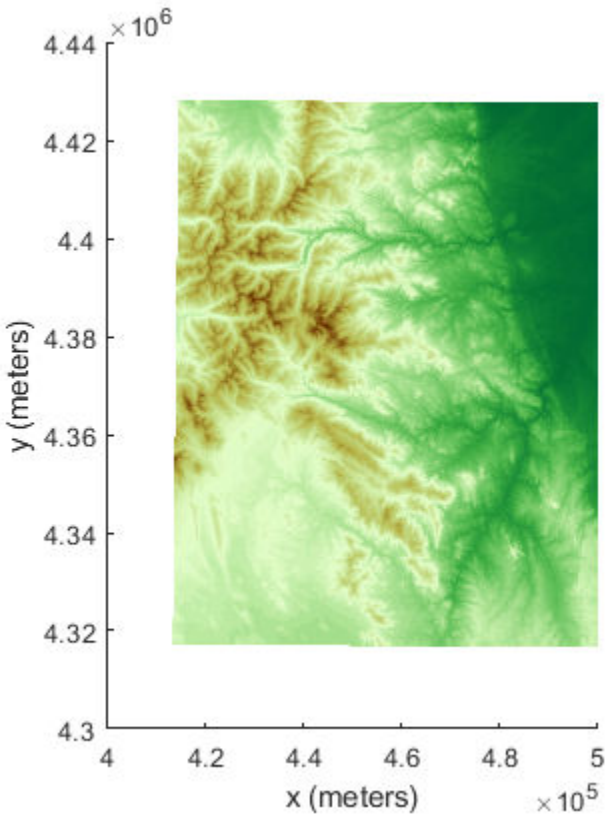
```
[Z,R] = readgeoraster('n39_w106_3arc_v2.dt1');
[lat,lon] = meshgrat(Z,R);
```

Now that you have your grid, select a map projection to use when projecting the coordinates. For this example, create a `projcrs` object for UTM zone 13 in the northern hemisphere. Then, project the latitude-longitude coordinates to *x*-*y* coordinates.

```
p = projcrs(32613);
[x,y] = projfwd(p,lat,lon);
```

Display the projected raster as a surface by calling `mapshow` and specifying the *x*-*y* coordinates and elevation array. Add axis labels and apply a colormap appropriate for elevation data.

```
figure
mapshow(x,y,Z,'DisplayType','surface')
xlabel('x (meters)')
ylabel('y (meters)')
demcmap(Z)
```



If the geographic CRS of the latitude-longitude coordinates does not match the geographic CRS of the projected CRS, then the projected coordinates may be inaccurate. You can find the geographic CRS of a `projcrs` object or a geographic raster reference object by querying their `GeographicCRS` properties.

```
p.GeographicCRS.Name
```

```
ans =
"WGS 84"
```

```
R.GeographicCRS.Name
```

```
ans =
"WGS 84"
```

The DTED file used in this example is courtesy of the US Geological Survey.

Unproject Raster Data

To unproject data that is associated with a map raster reference object, first create a grid of *x*-*y* coordinates for each point in the raster. Then, unproject the *x*-*y* map coordinates to geographic coordinates.

For example, import an image of Boston as an array and a map cells reference object. Get information about the map projection as a `projcrs` object by querying the `ProjectedCRS` property of the reference object.

```
[Z,R] = readgeoraster('boston.tif');
p = R.ProjectedCRS;
```

Create a coordinate grid by getting the *x*-*y* coordinates for each point in the raster. To do this, first create a grid of intrinsic coordinates that matches the size of the raster by using the `meshgrid` function. The intrinsic *y*-coordinates increase from row to row and the intrinsic *x*-coordinates increase from column to column. Then, transform the intrinsic coordinates to *x*-*y* coordinates by using the `intrinsicToWorld` function.

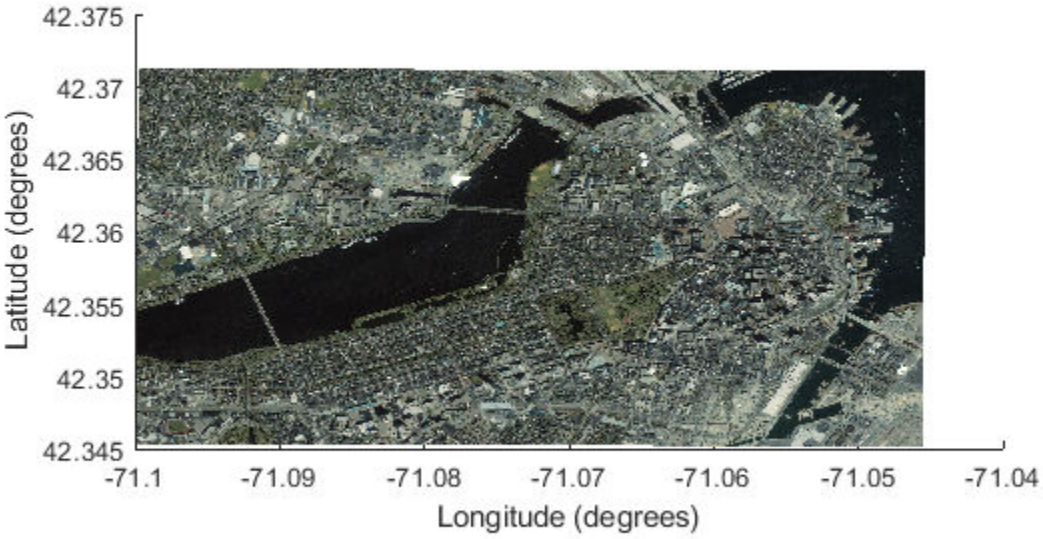
```
s = size(Z);
[xIntrinsic,yIntrinsic] = meshgrid(1:s(2),1:s(1));
[x,y] = intrinsicToWorld(R,xIntrinsic,yIntrinsic);
```

Unproject the *x*-*y* coordinates to latitude-longitude coordinates by using the `projinv` function and specifying the `projcrs` object and coordinate grid.

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

Display the unprojected image by calling `geoshow` and specifying the latitude-longitude coordinates and image array. By default, `geoshow` displays coordinates using a Plate Carrée projection. Then, add axis labels.

```
figure
geoshow(lat,lon,Z)
xlabel('Longitude (degrees)')
ylabel('Latitude (degrees)')
```



See Also

Functions
`intrinsicToWorld` | `meshgrat` | `meshgrid` | `projfwd` | `projinv`

Objects
`GeographicCellsReference` | `MapCellsReference`

Creating Web Map Service Maps

- “Basic WMS Terminology” on page 9-2
- “Basic Workflow for Creating WMS Maps” on page 9-3
- “Search the WMS Database” on page 9-5
- “Refine Your Search” on page 9-7
- “Update Your Layer” on page 9-8
- “Retrieve Your Map” on page 9-10
- “Modify Your Map Request” on page 9-24
- “Overlay Multiple Layers” on page 9-27
- “Animate Data Layers” on page 9-33
- “Display Animation of Radar Images over GOES Backdrop” on page 9-39
- “Retrieve Data from Web Map Server” on page 9-41
- “Save Your Favorite Servers” on page 9-49
- “Explore Other Layers using a Capabilities Document” on page 9-50
- “Write WMS Images to a KML File” on page 9-53
- “Search for Layers Outside the Database” on page 9-55
- “Troubleshoot WMS Servers” on page 9-56
- “Troubleshoot Access to the Hosted WMS Database” on page 9-61
- “Introduction to Web Map Display” on page 9-62
- “Basic Workflow for Displaying Web Maps” on page 9-66
- “Display a Web Map” on page 9-67
- “Select a Base Layer Map” on page 9-68
- “Specify a Custom Base Layer” on page 9-70
- “Specify a WMS Layer as a Base Layer” on page 9-72
- “Add an Overlay Layer to the Map” on page 9-74
- “Add Line, Polygon, and Marker Overlay Layers to Web Maps” on page 9-76
- “Remove Overlay Layers on a Web Map” on page 9-82
- “View Multiple Web Maps in a Browser” on page 9-86
- “Navigate a Web Map” on page 9-89
- “Close a Web Map” on page 9-92
- “Annotate a Web Map with Measurement Information” on page 9-93
- “Compositing and Animating Web Map Service (WMS) Meteorological Layers” on page 9-97
- “Troubleshoot Common Problems with Web Maps” on page 9-112

Basic WMS Terminology

- **Open Geospatial Consortium, Inc. (OGC)** — An organization comprising companies, government agencies, and universities that defines specifications for providers of geospatial data and developers of software designed to access that data. The specifications ensure that providers and clients can talk to each other and thus promote the sharing of geospatial data worldwide. You can access the Web Map Server Implementation Specification at the OGC website.
- **Web Map Service** — The OGC® defines a Web Map Service (WMS) as an entity that "produces maps of spatially referenced data dynamically from geographic information."
- **WMS server**— A server that follows the guidelines of the OGC to render maps and return them to clients.
- **georeferenced** — Tied to a specific location on the Earth.
- **raster data** — Data represented as a matrix in which each element corresponds to a specific rectangular or quadrangular geographic area.
- **map** — The OGC defines a map as "a portrayal of geographic information as a digital image file suitable for display on a computer screen."
- **raster map** — Geographically referenced information stored as a regular array of cells.
- **layer** — A data set containing a specific type of geographic information. Information can include temperature, elevation, weather, orthoimagery, boundaries, demographics, topography, transportation, environmental measurements, or various data from satellites.
- **capabilities document** — An XML document containing metadata describing the geographic content offered by a server.

See Also

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Basic Work ow for Creating WMS Maps

Work ow Summary

1. Search the WMS Database on page 9-5 for layers and servers that are of interest to you.
2. Refine your search on page 9-7 to include only servers or layers with specified field values.
3. Update your layer on page 9-8 to synchronize your selected layer with the server.
4. Modify your WMS request on page 9-24 to set properties like geographic limits, image dimensions, or background color of the map.
5. Retrieve your map on page 9-10 as a raster image from the server.
6. Display the map.

Create a Map of Elevation in Europe

Follow the example to learn the basic steps in creating a WMS map.

1. Search the WMS Database. on page 9-5 Mapping Toolbox software simplifies the process of WMS map creation by using a stored database of WMS servers. You can search the database for layers and servers that are of interest to you. WMS servers store map data in units called layers. For this example, search for layers that contain the phrase `'etopo1 hillshade'`.

   ```
   elevationLayer = wmsfind('etopo1 hillshade');
   ```

2. Refine your search. on page 9-7 In this example, the `wmsfind` function returns only one layer. As a result, you do not need to refine your search.
3. Update your layer. on page 9-8 Contact the web server identified in the database search to get the most up-to-date information. The `wmsupdate` function accomplishes two tasks:

   - Updates your `WMSLayer` object to include the most recent data
   - Fills in its `Details`, `CoordRefSysCodes`, and `Abstract` fields

   ```
   elevationLayer = wmsupdate(elevationLayer);
   ```

4. Modify your WMS request. on page 9-24 Create map axes with geographic limits appropriate for Europe. Then, get the map axes map structure (`mstruct`), which contains the settings for all the current map axes properties. You can use this `mstruct` to modify your WMS request. For example, you can set geographic limits, image dimensions, background color, and other properties of the map.

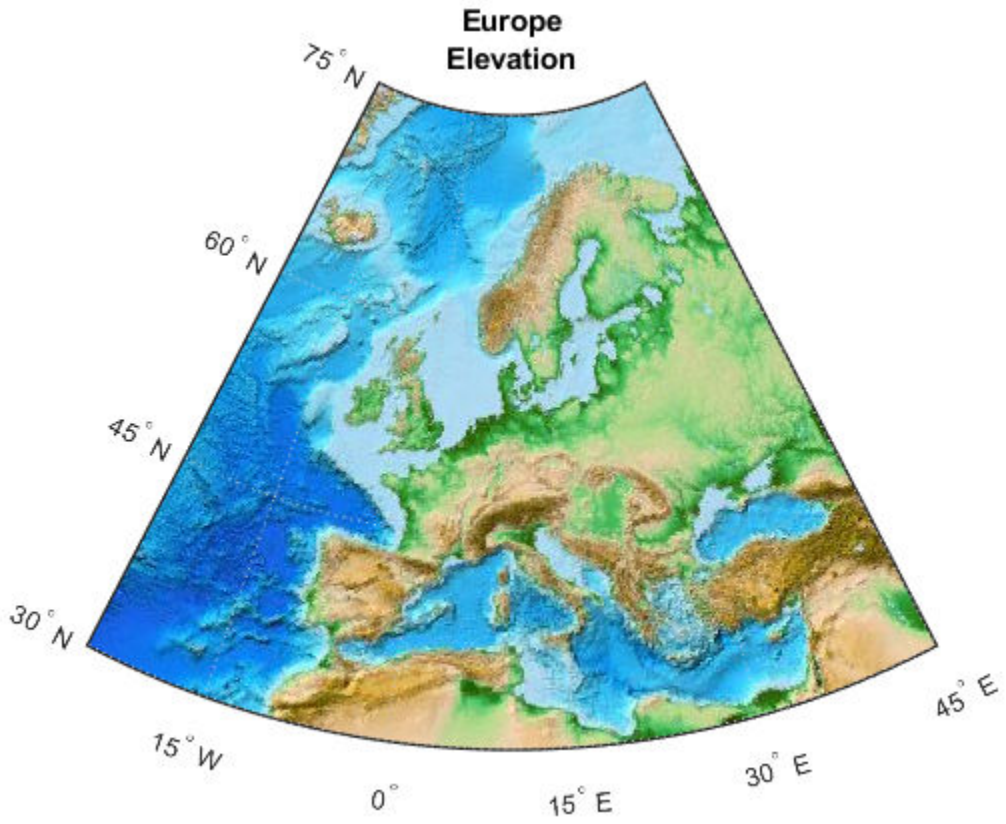
   ```
   figure
   worldmap europe
   mstruct = gcm;
   ```

5. Retrieve your map. on page 9-10 Read the layer using the `wmsread` function. Set the longitude and latitude limit parameters to the current map axes limits.

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

   The `wmsread` function returns a map called `elevationImage` and a raster reference object `R`, which ties the map to a specific location on Earth.
6. Display the map on the map axes and add a title.

   ```
   geoshow(elevationImage,R)
   title({'Europe','Elevation'})
   ```



See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic WMS Terminology” on page 9-2

Search the WMS Database

Introduction to the WMS Database

The Mapping Toolbox contains a database of over 1,000 stored WMS servers and over 100,000 layers. MathWorks creates this database, called the WMS Database, by conducting a series of Internet searches and qualifying the search results.

Note MathWorks cannot guarantee the stability and accuracy of WMS data, as the servers listed in the WMS Database are located on the Internet and are independent from MathWorks. Occasionally, you may receive error messages from servers experiencing difficulties. Servers can go down or become unavailable.

`wmsfind` is the only WMS function that accesses the stored WMS Database. By default, `wmsfind` searches the WMS database installed with the product. Using the `Version` parameter, you can also search a version of the WMS database hosted on the MathWorks website or a WMS database from a previous release. The information found in the database installed with the product is static and is not automatically updated—it was validated at the time of the software release. The web-hosted database is updated regularly.

Note Searching the web-hosted version of the WMS database requires a connection to the Internet. If you encounter problems, refer to “Troubleshoot Access to the Hosted WMS Database” on page 9-61 for tips.

The WMS Database contains the following fields.

Field Name	Data Type	Field Content
`ServerTitle`	Character vector	Title of the WMS server, descriptive information about the server
`ServerURL`	Character vector	URL of the WMS server
`LayerTitle`	Character vector	Title of the layer, descriptive information about the layer
`LayerName`	Character vector	Name of the layer, keyword the server uses to retrieve the layer
`Latlim`	Two-element vector	Southern and northern latitude limits of the layer
`Lonlim`	Two-element vector	Western and eastern longitude limits of the layer

The `LayerTitle` and `LayerName` fields sometimes have the same values. The `LayerName` indicates a code used by the servers, such as `'29:2'`, while the `LayerTitle` provides more descriptive information. For instance, `'Elevation and Rivers with Backdrop'` is a `LayerTitle`.

For an example of searching the WMS database, see “Find Temperature Data in the WMS Database” on page 9-5.

Find Temperature Data in the WMS Database

For this example, assume that you work as a research scientist and study the relationship between global warming and plankton growth. Increased plankton growth leads to increased carbon dioxide

absorption and reduced global warming. The sea surface temperature is already rising, however, which may reduce plankton growth in some areas. You begin investigating this complex relationship by mapping sea surface temperature.

1 Search the WMS Database for temperature data. By default, `wmsfind` searches the WMS database installed with the product. You can also search a version of the WMS database hosted on the MathWorks website, or a database from a previous release. Searching the web-hosted database requires a connection to the Internet.

```
layers = wmsfind('temperature');
```

By default, `wmsfind` searches both the `LayerName` and `LayerTitle` fields of the WMS Database for partial matches. The function returns an array of `WMSLayer` objects, which contains one object for each layer whose name or title partially matches `'temperature'`.

2 Click `layers` in the Workspace browser and then click one of the objects labeled `<1x1 WMSLayer>`.

```
     ServerTitle: 'NASA SVS Image Server'
       ServerURL: 'http://svs.gsfc.nasa.gov/cgi-bin/wms?'
      LayerTitle: 'Background Image for Global Sea Surface
                  Temperature from June, 2002 to September,
                  2003 (WMS)'
       LayerName: '2905_17492_bg'
          Latlim: [-90.0000 90.0000]
          Lonlim: [-180.0000 180.0000]
        Abstract: '<Update using WMSUPDATE>'
CoordRefSysCodes: '<Update using WMSUPDATE>'
         Details: '<Update using WMSUPDATE>'
```

A `WMSLayer` object contains three fields that do not appear in the WMS Database—`Abstract`, `CoordRefSysCodes`, and `Details`. (By default, these fields do not display in the command window if they are not populated with `wmsupdate`. For more information, see “Update Your Layer” on page 9-8 in the *Mapping Toolbox User's Guide*.)

`Note` `WMSLayer` is one of several objects related to WMS. If you are new to object-oriented programming, you can learn more about objects, methods, and properties in “Classes”.

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Re ne Your Search

Re ne Search by Text

Your initial search may return hundreds or even thousands of layers. Scanning all these layers to find the most relevant one could take a long time. You need to refine your search.

1 Refine your search to receive only layers that include sea surface temperature.

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

2 Refine the search again to include only layers that contain the term "global."

```
global_sst = sst.refine('global');
```

Re ne Search by Geographic Limits

You can search for layers in a specific geographic area.

1 First, find hurricane layers.

```
layers = wmsfind('hurricane');
```

2 Refine your search by selecting layers that are in the western hemisphere.

```
western_hemisphere = layers.refineLimits ...
   ('Latlim',[-90 90], 'Lonlim', [-180 0]);
```

3 Refine again to include only layers in the western hemisphere that include temperature data.

```
temp_and_west = western_hemisphere.refine('temperature');
```

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Update Your Layer

After you find your specific layer of interest, you can leave the WMS Database and work with a WMS server. In this section, you learn how to synchronize your layer with the WMS source server.

Note When working with the Internet, you may have to wait several minutes for information to download, or servers can become unavailable. If you encounter problems, refer to “Troubleshoot WMS Servers” on page 9-56 for tips.

Use the `wmsupdate` function to synchronize a `WMSLayer` object with the corresponding WMS server. This synchronization populates the `Abstract`, `CoordRefSysCodes`, and `Details` fields.

1 Find all layers in the WMS Database with the title "Global Sea Surface Temperature."

```
global_sst = wmsfind ('Global Sea Surface Temperature', ...
   'SearchField','LayerTitle');
```

2 Use the `WMSLayer.servers` method to determine the number of unique servers.

```
global_sst.servers
```

3 If your search returns more than one server, consider setting the `wmsupdate` `'AllowMultipleServers'` property to `true`. (However, be aware that if you have many servers, updating them could take a long time.)

```
global_sst = wmsupdate(global_sst,'AllowMultipleServers',true);
```

4 Now that you have updated all the fields in your `WMSLayer` objects, you can search by the `Abstract` field. View the abstract of the first layer.

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

el_nino(1).Abstract
```

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

5 View the coordinate reference system codes associated with this layer. For more information, see “Understand Coordinate Reference System Codes” on page 9-10.

```
el_nino(1).CoordRefSysCodes
```

6 View the contents of the `Details` field.

```
el_nino(1).Details
```

```
ans =

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

```
          Style: [1x2 struct]
        Version: '1.1.1'
```

The `Style` field covers a wide range of information, such as the line styles used to render vector data, the background color, the numeric format of data, the month of data collection, or the dimensional units.

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Retrieve Your Map

In this section...

"Map Retrieval Methods" on page 9-10
"Understand Coordinate Reference System Codes" on page 9-10
"Retrieve Your Map with wmsread" on page 9-11
"Use wmsread with Optional Parameters" on page 9-12
"Add a Legend to Your Map" on page 9-12
"Retrieve Your Map with WebMapServer.getMap" on page 9-19

Map Retrieval Methods

To retrieve a map from a WMS server, use the function `wmsread` or, in a few specific situations, the `WebMapServer.getMap` method. Use the `getMap` method when:

- Working with non-EPSG:4326 reference systems
- Creating an animation of a specific geographic area over time
- Retrieving multiple layers from a WMS server

In most cases, use `wmsread` to retrieve your map. To use `wmsread`, specify either a `WMSLayer` object or a map request URL. Obtain a `WMSLayer` object by using `wmsfind` to search the WMS Database. Obtain a map request URL from:

- The output of `wmsread`
- The `RequestURL` property of a `WMSMapRequest` object
- An Internet search

The map request URL character vector is composed of a WMS server URL with additional WMS parameters. The map request URL can be inserted into a browser to make a request to a server, which then returns a raster map.

Understand Coordinate Reference System Codes

When using `wmsread`, request a map that uses the EPSG:4326 coordinate reference system. EPSG stands for European Petroleum Survey Group. This group, an organization of specialists working in the field of oil exploration, developed a database of coordinate reference systems. Coordinate reference systems identify position unambiguously. Coordinate reference system codes are numbers that stand for specific coordinate reference systems.

EPSG:4326 is based on the 1984 World Geodetic System (WGS84) datum and the latitude and longitude coordinate system, with angles in degrees and Greenwich as the central meridian. All servers in the WMS Database, and presumably all WMS servers in general, use the EPSG:4326 reference system. This system is a requirement of the OGC WMS specification. If a layer does not use EPSG:4326, Mapping Toolbox software uses the next available coordinate reference system code. The Mapping Toolbox does not support automatic coordinate reference systems (systems in which the user chooses the center of projection). For more information about coordinate reference system codes, please see the Spatial Reference website.

Retrieve Your Map with wmsread

NASA's Blue Marble Next Generation layer shows the Earth's surface for each month of 2004 at high resolution (500 meters/pixel). Read and display the Blue Marble Next Generation layer.

1 Search the WMS Database for all layers with `'nasa'` in the `ServerURL` field.

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

2 Use the `WMSLayer.refine` method to refine your search to include only those layers with the phrase `'bluemarbleng'` in the `LayerName` field. This syntax creates an exact search.

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

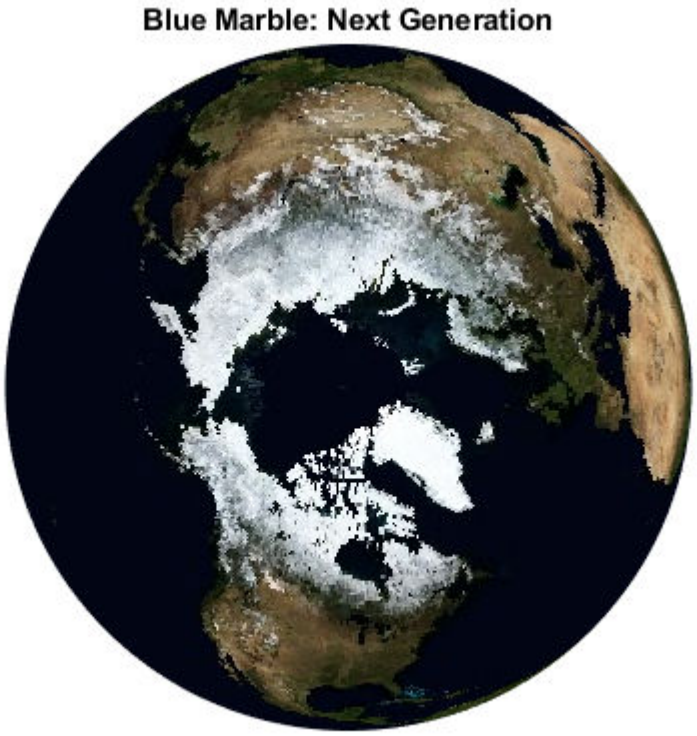
3 Use the `wmsread` function to retrieve the Blue Marble Next Generation layer.

```
[A,R] = wmsread(layer);
```

The `wmsread` function returns `A`, a geographically referenced raster map, and `R`, a raster reference object that ties `A` to the EPSG:4326 geographic coordinate system. The geographic limits of `A` span the full latitude and longitude extent of `layer`.

4 Open a figure window, set up your map axes, and display your map.

```
figure
axesm globe
axis off
geoshow(A,R)
title('Blue Marble: Next Generation')
```



The layer used in this example is courtesy of NASA/JPL-Caltech.

Use wmsread with Optional Parameters

The `wmsread` function allows you to set many optional parameters, such as image height and width and background color. This example demonstrates how to view an elevation map in 0.5-degree resolution by changing the cell size, and how to display the ocean in light blue by setting the background color. For a complete list of parameters, see `wmsread`.

1 Search the WMS database for layers that contain `foundation.gtopo30` in the `LayerName` field. GTOPO30, a digital elevation model developed by the United States Geological Survey (USGS), has a horizontal grid spacing of 30 arc seconds.

```
gtopo30Layer = wmsfind('foundation.gtopo30');
```

2 Define a background color, specifying red, green, and blue levels.

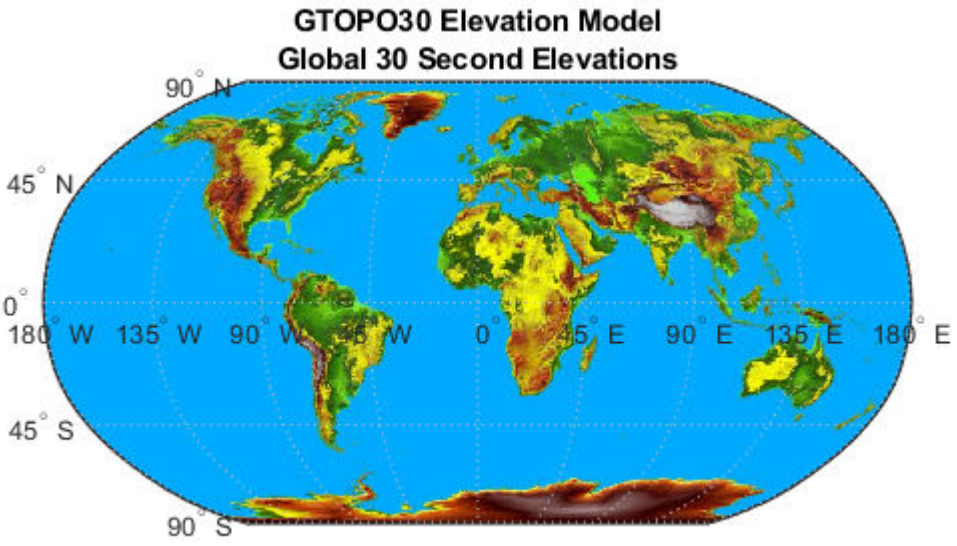
```
oceanColor = [0 170 255];
```

3 Use the `BackgroundColor` and `CellSize` parameters of the `wmsread` function to set the background color and cell size of your retrieved map.

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

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

```
figure
worldmap world
geoshow(A,R)
title({'GTOPO30 Elevation Model',gtopo30Layer.LayerTitle})
```



Add a Legend to Your Map

A WMS server renders a layer as an image. Without a corresponding legend, interpreting the pixel colors can be difficult. Some WMS servers provide access to a legend image for a particular layer via

a URL that appears in the layer's `Details.Style.LegendURL` field. (See the `WMSLayer.Details` reference page for more information.)

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

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

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

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

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

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

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

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

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

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

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

```
urban_temperature.Details
```

```
ans =

  struct with fields:

    MetadataURL: 'http://svs.gsfc.nasa.gov/vis/a000000/a003100/a003152/a003152.fgdc'
     Attributes: [1x1 struct]
    BoundingBox: [1x1 struct]
      Dimension: [1x1 struct]
   ImageFormats: {'image/png'}
    ScaleLimits: [1x1 struct]
          Style: [1x1 struct]
        Version: '1.3.0'
```

Display the `Style` field in the Command Window:

```
urban_temperature.Details.Style
```

```
ans =

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

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

```
urban_temperature.Details.Style.LegendURL
```

```
ans =

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

5 Download the legend URL:

```
url = urban_temperature.Details.Style.LegendURL.OnlineResource
```

The URL appears in the command window:

```
url =

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

6 Display the legend image using the `image` command and set properties, such that the image displays with one-to-one, screen-to-pixel resolution. This legend is simply an image of a colorbar, not the legend in MATLAB graphics.

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

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

7 Steps 7–10 demonstrate how to capture the output from a map frame and append the legend. By appending the legend in this fashion, you avoid warping text in the legend image. (Legend text warps if you display the image with `geoshow`.)

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

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

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

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

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

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

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

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

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

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

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

```
f = getframe(gcf);
legendImg = uint8(255*ones(size(temperatureLegend,1),size(f.cdata,2),3));
offset = dimension/2;
halfSize = size(temperatureLegend, 2)/2;
legendImg(:,offset-halfSize:offset+halfSize-1,:) = temperatureLegend;
combined = [f.cdata; legendImg];
figure
pos = get(gcf,'position');
set(gcf,'position',[10 100 size(combined,2) size(combined,1)])
set(gca,'units','normalized','position', ...
   [0 0 1 1]);
image(combined)
axis off image
```

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

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

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

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

Open the MATLAB Editor, and paste in this code:

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

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

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

Retrieve Your Map with WebMapServer.getMap

The `WebMapServer.getMap` method allows you to retrieve maps in any properly defined EPSG coordinate reference system. If you want to retrieve a map in the EPSG:4326 reference system, you can use `wmsread`. If you want to retrieve a layer whose coordinates are not in the EPSG:4326 reference system, however, you must use the `WMSMapRequest` object to construct the request URL and the `WebMapServer.getMap` method to retrieve the map. This example demonstrates how to create maps in "Web Mercator" coordinates, also known as "WGS 84/Pseudo-Mercator" coordinates, using the `WMSMapRequest` and `WebMapServer` objects. The Web Mercator coordinate system is commonly used by web applications.

The USGS National Map provides ortho-imagery and topography maps from various regions of the United States. The server provides the data in both EPSG:4326 and in Web Mercator coordinates, as defined by EPSG codes EPSG:102113, EPSG:3857. For more information about these codes, see the Spatial Reference website.

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

```
filename = 'boston.tif';
info = georasterinfo(filename);
R = info.RasterReference;
proj = R.ProjectedCRS;
[latlim,lonlim] = projinv(proj, ...
                          R.XWorldLimits,R.YWorldLimits);
```

2 Convert the geographic limits to Web Mercator. To do this, first create a `projcrs` object by specifying the EPSG:3857 coordinate system. Then, project the coordinates using `projfwd`. To obtain the imagery in this coordinate reference system, you need to use `WMSMapRequest` and `WebMapServer` since `wmsread` only requests data in the EPSG:4326 system.

```
p = projcrs(3857);
[x,y] = projfwd(p,latlim,lonlim);
xlimits = [min(x) max(x)];
ylimits = [min(y) max(y)];
```

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

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

4 Re-compute the new limits.

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

5 Find the USGS National Map from the WMS database and select the Digital Ortho-Quadrangle layer.

```
doqLayer = wmsfind('usgsnaipplus','SearchField','serverurl');
doqLayer = wmsupdate(refine(doqLayer,'image'));
doqLayer = refine(doqLayer,'USGSNaip','SearchField','abstract');
```

6 Create `WebMapServer` and `WMSMapRequest` objects.

```
server  = WebMapServer(doqLayer.ServerURL);
numberOfAttempts = 50;
attempt = 0;
request = [];
while(isempty(request))
    try
        request = WMSMapRequest(doqLayer, server);
    catch e
        attempt = attempt + 1;
        fprintf('%d\n', attempt)
        if attempt > numberOfAttempts
            throw(e)
        end
    end
end
```

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

```
request.CoordRefSysCode = 'EPSG:3857';
request.ImageHeight = imageHeight;
request.ImageWidth  = imageWidth;
request.XLim = xLim;
request.YLim = yLim;
```

8 Request a map of the ortho-imagery in Web Mercator coordinates.

```
A_PCS = getMap(server, request.RequestURL);
R_PCS = request.RasterReference;
```

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

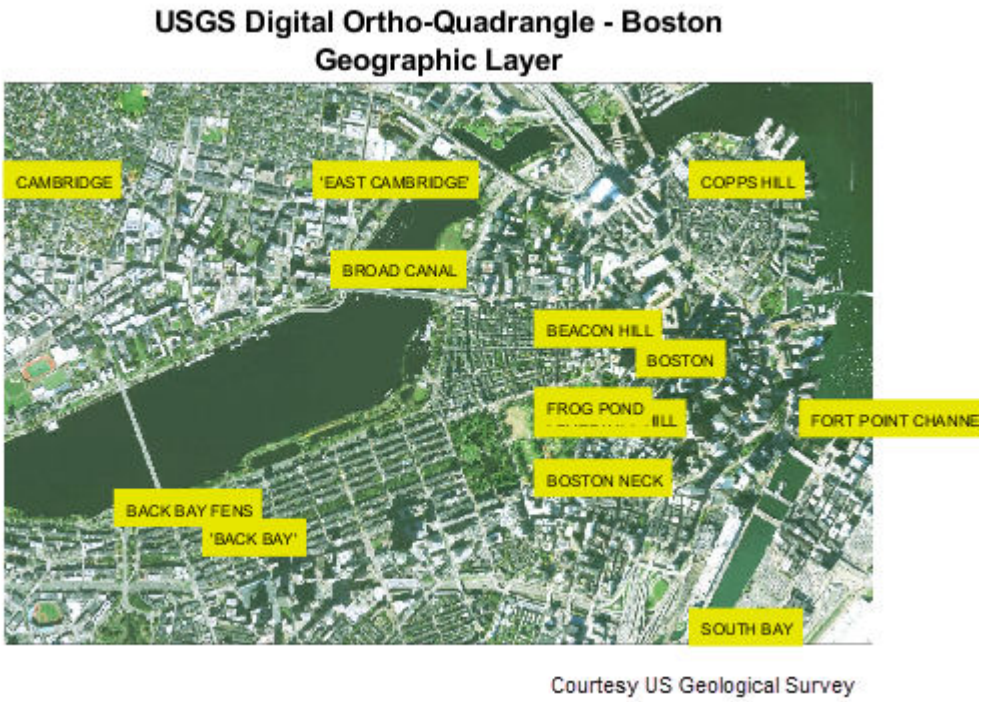
```
request.CoordRefSysCode = 'EPSG:4326';
request.Latlim = latlim;
request.Lonlim = lonlim;
A_Geo = getMap(server, request.RequestURL);
R_Geo = request.RasterReference;
```

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

```
S = shaperead('boston_placenames');
x = [S.X]*unitsratio('sf','meter');
y = [S.Y]*unitsratio('sf','meter');
names = {S.NAME};
[lat, lon] = projinv(proj, x, y);
[xPCS, yPCS] = projfwd(p, lat, lon);
```

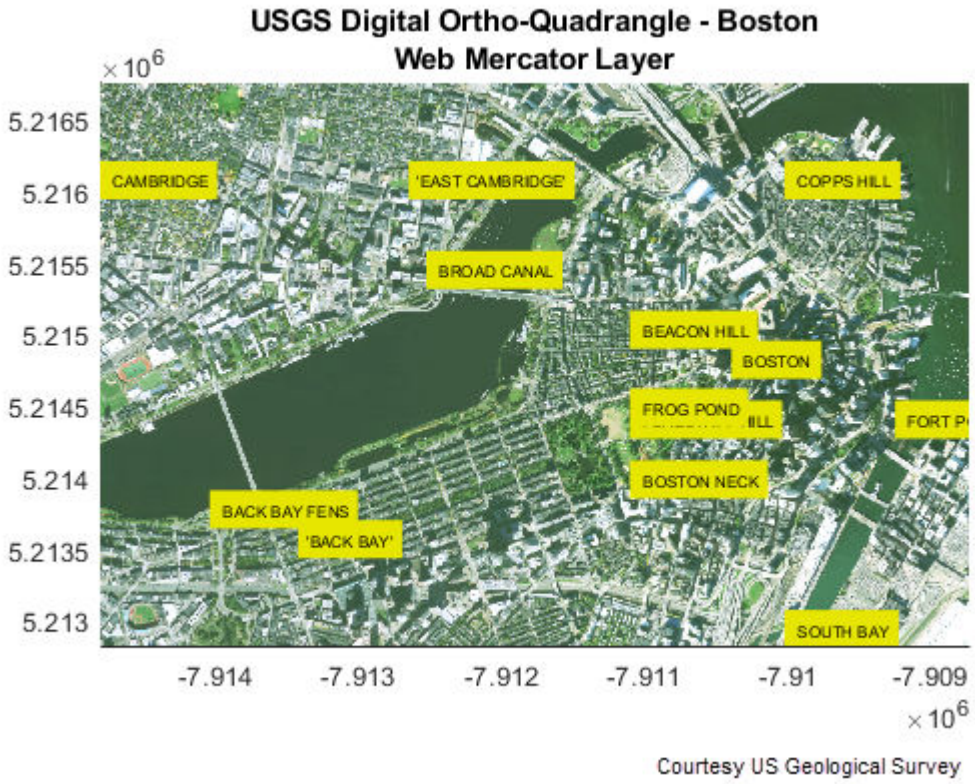
11 Project and display the ortho-imagery obtained in EPSG:4326 coordinates using geoshow.

```
figure
axesm('mercator','MapLatLimit',latlim,'MapLonLimit',lonlim)
geoshow(A_Geo,R_Geo)
textm(lat, lon, names, 'Color',[0 0 0], ...
    'BackgroundColor',[0.9 0.9 0],'FontSize',6);
axis tight
title({'USGS Digital Ortho-Quadrangle - Boston', ...
    'Geographic Layer'})
```



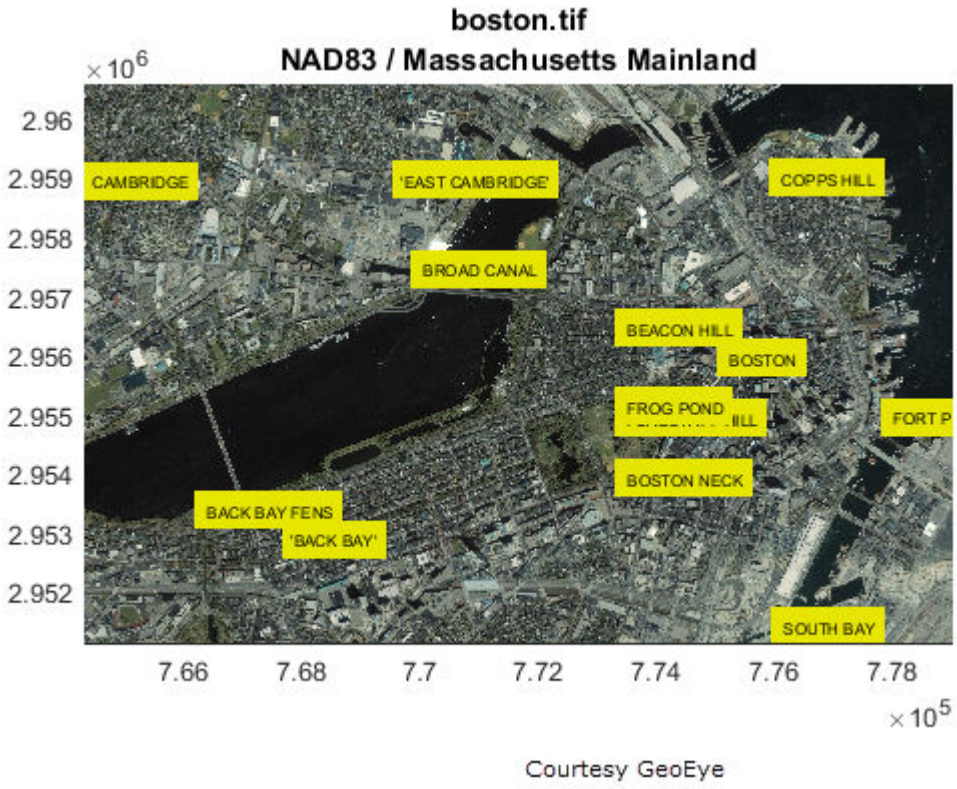
12 Display the ortho-imagery obtained in Web Mercator coordinates.

```
figure
mapshow(A_PCS, R_PCS);
text(xPCS, yPCS, names, 'Color',[0 0 0], ...
    'BackgroundColor',[0.9 0.9 0],'FontSize',6,'Clipping','on');
axis tight
title({'USGS Digital Ortho-Quadrangle - Boston', 'Web Mercator Layer'})
```



13 Display the image from `boston.tif` for comparison.

```
figure
mapshow(filename)
text(x, y, names, 'Color',[0 0 0], ...
    'BackgroundColor',[0.9 0.9 0],'FontSize',6,'Clipping','on');
axis tight
title({filename, proj.Name})
```



Courtesy GeoEye

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Modify Your Map Request

In this section...

“Set Map Request Geographic Limits and Time” on page 9-24
“Edit Web Map Request URL Manually” on page 9-25

Set Map Request Geographic Limits and Time

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

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

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

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

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

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

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

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

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

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

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

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

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

7 Set the time request to March 1, 2009.

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

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

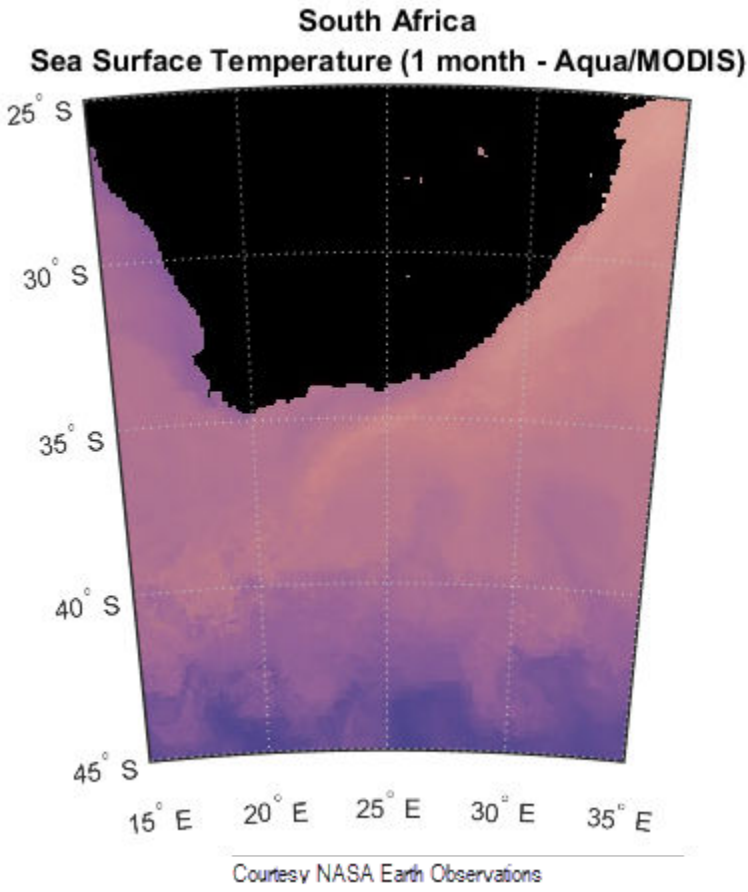
```
sstImage = server.getMap(mapRequest.RequestURL);
```

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

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

10 Project and display an image georeferenced to latitude and longitude. Use the raster reference object provided by the `RasterReference` property of the `WMSMapRequest` object.

```
geoshow(sstImage, mapRequest.RasterReference);
title({'South Africa', sst.LayerTitle}, ...
    'FontWeight', 'bold', 'Interpreter', 'none')
```



Edit Web Map Request URL Manually

You can modify a map request URL manually.

1 Obtain the map request URL.

```
nasa = wmsfind('nasa', 'SearchField', 'serverurl');
layer = nasa.refine('bluemarbleng', 'SearchField', 'layername', ...
   'MatchType', 'exact');
layer = layer(1);
mapRequest = WMSMapRequest(layer);
```

2 Set the map request URL to a variable.

```
mapURL = mapRequest.RequestURL;
```

3 Modify the bounding box to include the southern hemisphere. To do this, create a new variable called `modifiedURL` by copying and pasting the contents of `mapURL`. Then, change the bounding box section of the URL to:

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

Enter the URL as one continuous character vector.

4 Display the modified map.

```
[A, R] = wmsread(modifiedURL);
figure
```

```
axesm globe
axis off
geoshow(A, R)
title('Blue Marble: Southern Hemisphere Edition')
```

The image is courtesy of NASA/JPL-Caltech.

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Overlay Multiple Layers

In this section...

"Create Composite Map of Multiple Layers from One Server" on page 9-27
"Combine Layers from One Server with Data from Other Sources" on page 9-28
"Drape Orthoimagery Over DEM" on page 9-29

Create Composite Map of Multiple Layers from One Server

The WMS specification allows the server to merge multiple layers into a single raster map. The MetaCarta VMAP0 server contains many data layers, such as coastlines, national boundaries, ocean, and ground. Read and display a composite of multiple layers from the VMAP0 server. The rendered map has a spatial resolution of 0.5 degrees per cell.

1 Find and update the VMAP0 layers.

```
vmap0 = wmsfind('vmap0.tiles', 'SearchField', 'serverurl');
vmap0 = wmsupdate(vmap0);
```

2 Create an array of multiple layers that include ground and ocean, coastlines and national boundaries.

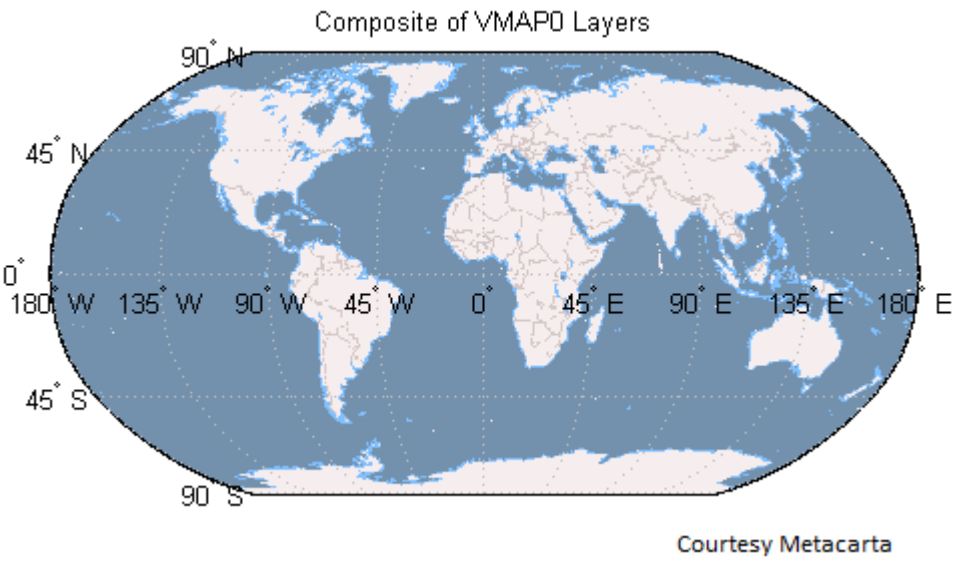
```
layers = [refine(vmap0, 'coastline_01'); ...
          refine(vmap0, 'country_01'); ...
          refine(vmap0, 'ground_01'); ...
          refine(vmap0, 'inwater'); ...
          refine(vmap0, 'ocean')];
```

3 Retrieve the composite map. Request a cell size of .5 degrees by setting the image height and image width parameters. Set 'Transparent' to true so that all pixels not representing features or data values in a layer are set to a transparent value in the resulting image, making it possible to produce a composite map.

```
[overlayImage, R] = wmsread(layers, 'Transparent', true, ...
                            'ImageHeight', 360, 'ImageWidth', 720);
```

4 Display your composite map.

```
figure
worldmap('world')
geoshow(overlayImage, R);
title('Composite of VMAP0 Layers')
```



Combine Layers from One Server with Data from Other Sources

This example illustrates how you can merge a boundaries raster map with vector data.

1 Layout out a global raster with 1/2-degree cells. Specify columns running north-to-south, for consistency with `wmsread`.

```
latlim = [-90 90];
lonlim = [-180 180];
cellExtent = 1/2;
R = georefcells(latlim,lonlim,  ...
    cellExtent,cellExtent,'ColumnsStartFrom','north')
```

2 Read the `landareas` polygon shapefile and convert it to a raster map.

```
land = shaperead('landareas', 'UseGeoCoords', true);
lat = [land.Lat];
lon = [land.Lon];
land = vec2mtx(lat, lon, zeros(R.RasterSize),R, 'filled');
```

3 Read the `worldrivers` polyline shapefile and convert it to a raster map.

```
riverLines = shaperead('worldrivers.shp','UseGeoCoords',true);
rivers = vec2mtx([riverLines.Lat], [riverLines.Lon], land, R);
```

4 Merge the rivers with the land.

```
merged = land;
merged(rivers == 1) = 3;
```

5 Obtain the boundaries image from the VMAP0 server.

```
vmap0 = wmsfind('vmap0.tiles', 'SearchField', 'serverurl');
vmap0 = wmsupdate(vmap0);
layer = refine(vmap0, 'country_01');
height = R.RasterSize(1);
width  = R.RasterSize(2);
[boundaries,boundariesR] = wmsread(layer, 'ImageFormat', 'image/png', ...
    'ImageHeight', height, 'ImageWidth', width);
```

6 Confirm that the boundaries and merged rasters are coincident.

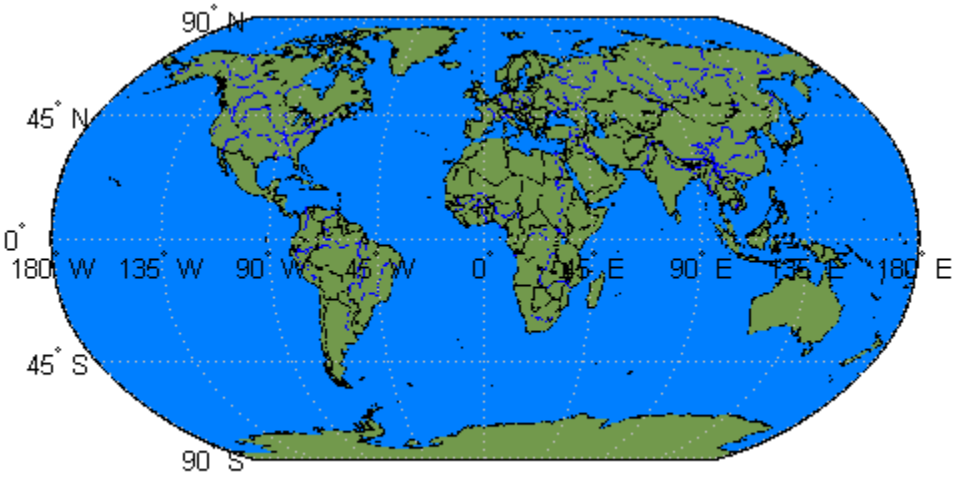
```
isequal(boundariesR,R)
```

7 Merge the rivers and land with the boundaries.

```
index = boundaries(:,:,1) ~= 255 ...
    & boundaries(:,:,2) ~= 255 ...
    & boundaries(:,:,3) ~= 255;
merged(index) = 1;
```

8 Display the result.

```
figure
worldmap(merged, R)
geoshow(merged, R, 'DisplayType', 'texturemap')
colormap([.45 .60 .30; 0 0 0; 0 0.5 1; 0 0 1])
```



Courtesy U.S.National Geospatial-Intelligence Agency (NGA) and Metacarta

Drape Orthoimagery Over DEM

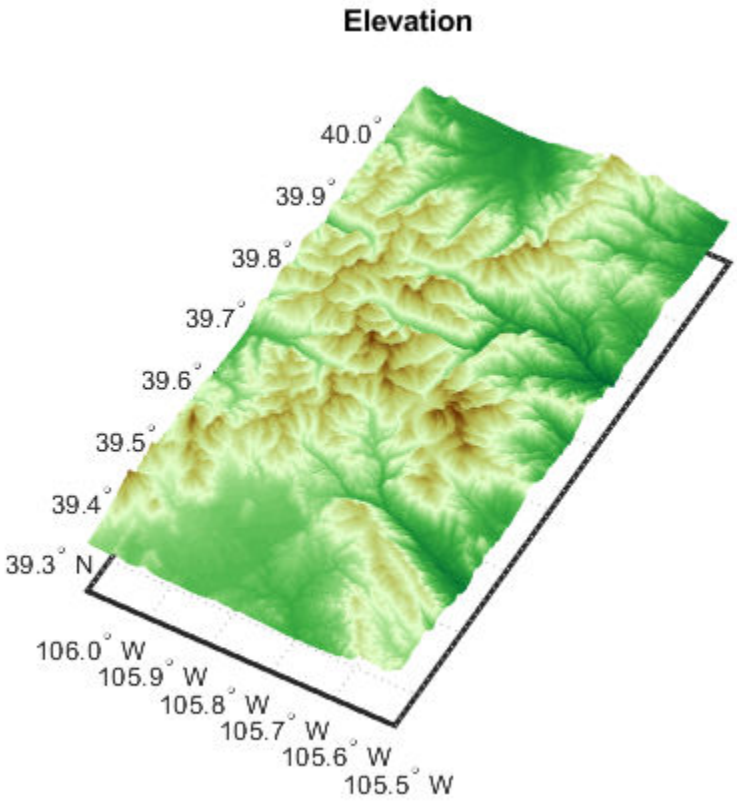
Read elevation data and a geographic postings reference for an area around South Boulder Peak in Colorado. Then, crop the elevation data to a smaller area using `geocrop`.

```
[fullZ,fullR] = readgeoraster('n39_w106_3arc_v2.dt1','OutputType','double');

latlim = [39.25 40.0];
lonlim = [-106 -105.5];
[Z,R] = geocrop(fullZ,fullR,latlim,lonlim);
```

Display the elevation data. To do this, create a set of map axes for the United States, plot the data as a surface, and apply an appropriate colormap. View the map in 3-D by adjusting the camera position and target. Set the vertical exaggeration using `daspectm`.

```
figure
usamap(R.LatitudeLimits, R.LongitudeLimits)
geoshow(Z,R,'DisplayType','surface')
demcmap(Z)
title('Elevation');
```

```
cameraPosition = [218100 4367600 183700];
cameraTarget = [0 4754200 2500];
set(gca,'CameraPosition', cameraPosition, ...
        'CameraTarget', cameraTarget)
daspectm('m',3)
```



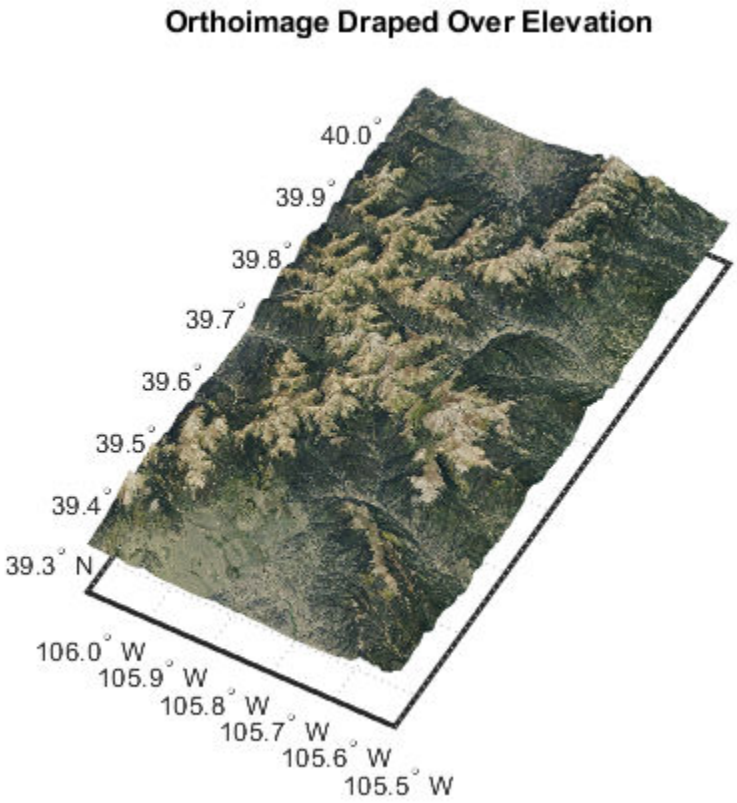
Drape an orthoimage over the elevation data. To do this, first get the names of high-resolution orthoimagery layers from the USGS National Map using `wmsinfo`. In this case, the orthoimagery layer is the only layer from the server. Use multiple attempts to connect to the server in case it is busy.

```
numberOfAttempts = 5;
attempt = 0;
info = [];
serverURL = ...
   'http://basemap.nationalmap.gov/ArcGIS/services/USGSImageryOnly/MapServer/WMSServer?';
while(isempty(info))
    try
        info = wmsinfo(serverURL);
        orthoLayer = info.Layer(1);
    catch e
        attempt = attempt + 1;
        if attempt > numberOfAttempts
            throw(e);
        else
            fprintf('Attempting to connect to server:\n"%s"\n', serverURL)
```

```
        end
    end
end
```

Request a map of the orthoimagery layer using `wmsread`. To display the orthoimagery, use `geoshow` and set the `CData` property to the layer.

```
imageHeight = size(Z,1);
imageWidth  = size(Z,2);

orthoImage = wmsread(orthoLayer,'Latlim',R.LatitudeLimits, ...
    'Lonlim',R.LongitudeLimits,'ImageHeight', imageHeight, ...
    'ImageWidth',  imageWidth);

figure
usamap(R.LatitudeLimits,R.LongitudeLimits)
geoshow(Z,R,'DisplayType','surface','CData',orthoImage);
title('Orthoimage Draped Over Elevation');
set(gca,'CameraPosition', cameraPosition, ...
        'CameraTarget', cameraTarget)
daspectm('m',3)
```



The DTED file used in this example is courtesy of the US Geological Survey.

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Animate Data Layers

In this section...

"Create Movie of Terra/MODIS Maps" on page 9-33
"Create Animated GIF File of WMS Maps" on page 9-34
"Animate Time-Lapse Radar Observations" on page 9-36

Create Movie of Terra/MODIS Maps

You can create maps of the same geographic region at different times and view them as a movie. For a period of seven days, read and display a daily composite of visual images from NASA's Moderate Resolution Imaging Spectroradiometer (MODIS) scenes captured during the month of December 2010.

1 Search the WMS Database for the MODIS layer.

```
neo = wmsfind('neowms*nasa', 'SearchField', 'serverurl');
modis = neo.refine('true*color*terra*modis');
modis = wmsupdate(modis);
```

2 Construct a `WebMapServer` object.

```
server = WebMapServer(modis.ServerURL);
```

3 Construct a `WMSMapRequest` object.

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

4 The `Extent` field provides the information about how to retrieve individual frames. You can request a single day since the extent is defined by day (`'/P1D'`). Note that for December 2010, the frames for December 8 and December 31 are not available.

```
modis.Details.Dimension.Extent
```

5 Create an array indicating the first seven days.

```
days = 1:7;
```

6 Set the value of `startTime` to December 01, 2010 and use a serial date number.

```
time = '2010-12-01';
startTime = datenum(time);
```

7 Open a figure window with axes appropriate for the region specified by the `modis` layer.

```
hFig = figure('Color', 'white');
worldmap(mapRequest.Latlim, mapRequest.Lonlim);
```

8 Save each frame into a video file.

```
videoFilename = 'modis_dec.avi';
writer = VideoWriter(videoFilename);
writer.FrameRate = 1;
writer.Quality = 100;
writer.open;
```

9 Retrieve a map of the `modis` layer for each requested day. Set the `Time` property to the day number. When obtaining the data from the server, use a `try/catch` statement to ignore either data not found on the server or any error issued by the server. Set `startTime` to one day less for correct indexing.

```
startTime = startTime - 1;
for k = days
```

```
        try
            mapRequest.Time = startTime + k;
            timeStr = datestr(mapRequest.Time);
            dailyImage = server.getMap(mapRequest.RequestURL);
            geoshow(dailyImage, mapRequest.RasterReference);
            title({mapRequest.Layer.LayerTitle, timeStr}, ...
                'Interpreter', 'none', 'FontWeight', 'bold')
            shg
            frame = getframe(hFig);
            writer.writeVideo(frame);
        catch e
            fprintf(['Server error: %s.\n', ...
                'Ignoring frame number %d on day %s.\n'], ...
                e.message, k, timeStr)
        end
        drawnow
        shg
    end
    writer.close
```

10 Read in all video frames.

```
    v = VideoReader(videoFilename);
    vidFrames = read(v);
    numFrames = get(v, 'NumberOfFrames');
```

11 Create a MATLAB movie structure from the video frames.

```
    frames = struct('cdata', [], 'colormap', []);
    frames(numFrames) = frames(1);
    for k = 1 : numFrames
        frames(k).cdata = vidFrames(:,:,:,k);
        frames(k).colormap = [];
    end
```

12 Playback movie once at the video's frame rate.

```
    movie(hFig, frames, 1, v.FrameRate)
```

Create Animated GIF File of WMS Maps

Read and display an animation of the Larsen Ice Shelf experiencing a dramatic collapse between January 31 and March 7, 2002.

1 Search the WMS Database for the phrase "Larsen Ice Shelf."

```
    iceLayer = wmsfind('Larsen Ice Shelf');
```

Try the first layer.

2 Construct a `WebMapServer` object.

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

3 Use the `WebMapServer.updateLayers` method to synchronize the layer with the WMS source server. Retrieve the most recent data and fill in the `Abstract`, `CoordRefSysCodes`, and `Details` fields.

```
    iceLayer = server.updateLayers(iceLayer(1));
```

4 View the abstract.

```
    fprintf('%s\n', iceLayer(1).Abstract)
```

5 Create the `WMSMapRequest` object.

```
request = WMSMapRequest(iceLayer(1), server);
```

6 Because you have updated your layer, the `Details` field now has content. Click `Details` in the MATLAB Variables editor. Then, click `Dimension`. The name of the dimension is `'time'`. Click `Extent`. The `Extent` field provides the available values for a dimension, in this case time. Save this information by entering the following at the command line:

```
extent = [',' iceLayer.Details.Dimension.Extent, ','];
```

7 Calculate the number of required frames. (The `extent` contains a comma before the first frame and after the last frame. To obtain the number of frames, subtract 1.)

```
frameIndex = strfind(extent, ',');
numFrames = numel(frameIndex) - 1;
```

8 Open a figure window and set up a map axes with appropriate geographic limits.

```
h = figure;
worldmap(request.Latlim, request.Lonlim)
```

9 Set the map axes properties. `MLineLocation` establishes the interval between displayed grid meridians. `MLabelParallel` determines the parallel where the labels appear.

```
setm(gca,'MLineLocation', 1, 'MLabelLocation', 1, ...
   'MLabelParallel',-67.5, 'LabelRotation', 'off');
```

10 Initialize the value of `animated` to 0.

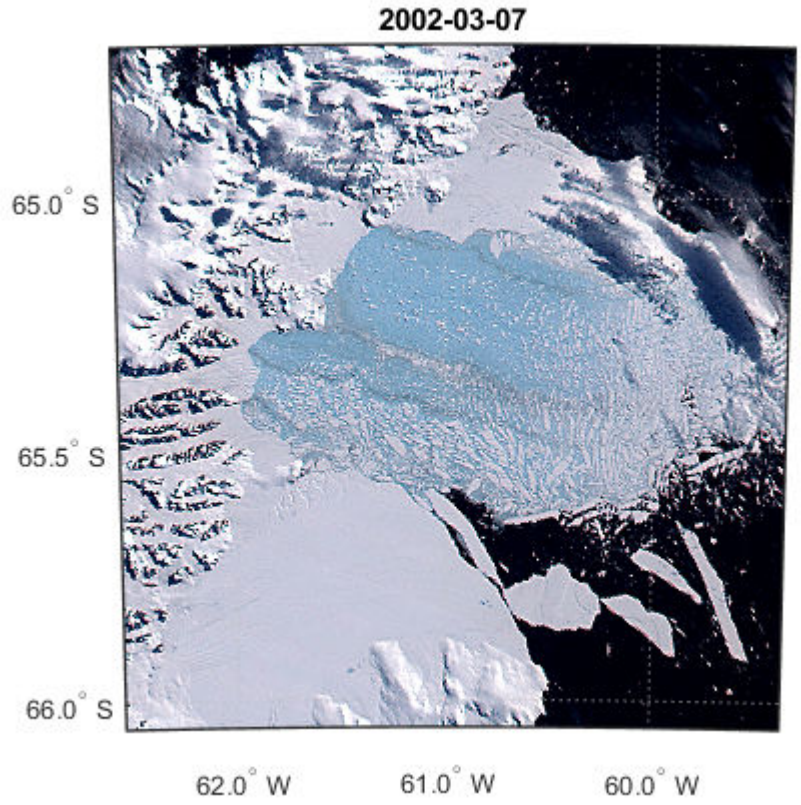
```
animated(1,1,1,numFrames) = 0;
```

11 Display the image of the Larsen Ice Shelf on different days.

```
for k=1:numFrames
   request.Time = extent(frameIndex(k)+1:frameIndex(k+1)-1);
   iceImage = server.getMap(request.RequestURL);
   geoshow(iceImage, request.RasterReference)
   title(request.Time, 'Interpreter', 'none')
   drawnow
   shg
   frame = getframe(h);
   if k == 1
      [animated, cmap] = rgb2ind(frame.cdata, 256, 'nodither');
   else
      animated(:,:,1,k) = rgb2ind(frame.cdata, cmap, 'nodither');
   end
end
```

12 Save and then view the animated GIF file.

```
filename = 'wmsanimated.gif';
imwrite(animated, cmap, filename, 'DelayTime', 1.5, ...
   'LoopCount', inf);
web(filename)
```



Snapshot from Animation of Larsen Ice Shelf

Courtesy NASA/Goddard Space Flight Center Scientific Visualization Studio

Animate Time-Lapse Radar Observations

Display Next-Generation Radar (NEXRAD) images for the United States using data from the Iowa Environmental Mesonet (IEM) Web map server. The server stores layers covering the past 50 minutes up to the present time in increments of 5 minutes. Read and display the merged layers.

1 Find layers in the WMS Database that include `'mesonet'` and `'nexrad'` in their `ServerURL` fields.

```
mesonet = wmsfind('mesonet*nexrad', 'SearchField', 'serverurl');
```

2 NEXRAD Base Reflect Current (`'nexrad-n0r'`) measures the intensity of precipitation. Refine your search to include only layers with this phrase in one of the search fields.

```
nexrad = mesonet.refine('nexrad-n0r', 'SearchField', 'any');
```

3 Remove the 900913 layers because they are intended for Google Maps overlay. Also remove the WMST layer because it contains data for different times.

```
layers_900913 = nexrad.refine('900913', 'SearchField', ...
   'layername');
layer_wmst = nexrad.refine('wmst', 'SearchField', 'layername');
rmLayerNames = {layers_900913.LayerName layer_wmst.LayerName};
index = ismember({nexrad.LayerName}, rmLayerNames);
nexrad = nexrad(~index);
```

4 Update your `nexrad` layer to fill in all fields and obtain most recent data.

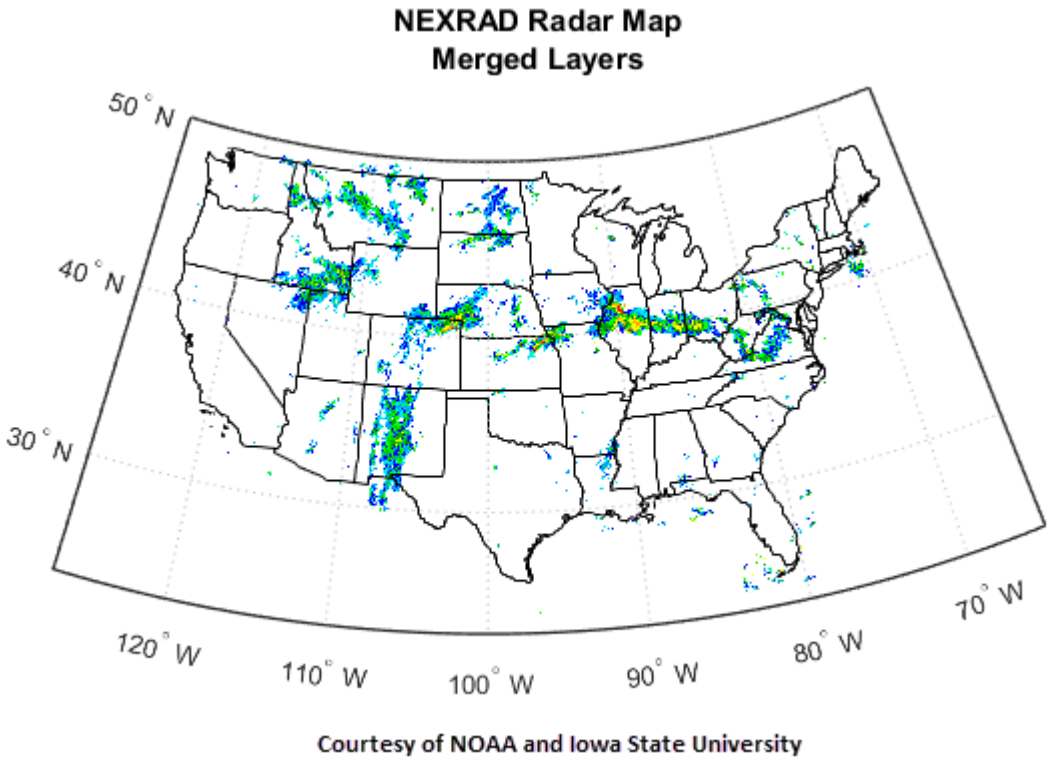
```
nexrad = wmsupdate(nexrad, 'AllowMultipleServers', true);
```

5 `'conus'` represents the conterminous 48 U.S. states (all except Hawaii and Alaska). Use the `usamap` function to construct a map axes for the conterminous states. Read in the `nexrad` layers.

```
region = 'conus';
figure
usamap(region)
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
[A, R] = wmsread(nexrad, 'Latlim', latlim, 'Lonlim', lonlim);
```

6 Display the NEXRAD merged layers map. Overlay with United States state boundary polygons.

```
geoshow(A, R);
geoshow('usastatehi.shp', 'FaceColor', 'none');
title({'NEXRAD Radar Map', 'Merged Layers'});
```



7 Loop through the sequence of time-lapse radar observations.

```
hfig = figure;
usamap(region)
hstates = geoshow('usastatehi.shp', 'FaceColor', 'none');
numFrames = numel(nexrad);
frames = struct('cdata', [], 'colormap', []);
frames(numFrames) = frames;
hmap = [];
frameIndex = 0;
for k = numFrames:-1:1
   frameIndex = frameIndex + 1;
```

```
        delete(hmap)
        [A, R] = wmsread(nexrad(k), 'Latlim', latlim, 'Lonlim', lonlim);
        hmap = geoshow(A, R);
        uistack(hstates,'top')
        title(nexrad(k).LayerName)
        drawnow
        frames(frameIndex) = getframe(hfig);
    end
```

8 Create an array to write out as an animated GIF file.

```
    animated(1,1,1,numFrames) = 0;
    for k=1:numFrames
        if k == 1
            [animated, cmap] = rgb2ind(frames(k).cdata, 256, 'nodither');
        else
            animated(:,:,1,k) = ...
                rgb2ind(frames(k).cdata, cmap, 'nodither');
        end
    end
```

9 View the animated GIF file.

```
    filename = 'wmsnexrad.gif';
    imwrite(animated, cmap, filename, 'DelayTime', 1.5, ...
        'LoopCount', inf);
    web(filename)
```

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Display Animation of Radar Images over GOES Backdrop

This example shows how to display NEXRAD radar images. The images cover the past 24 hours, sampled at one-hour intervals, for the United States using data from the IEM WMS server. Use the JPL Daily Planet layer as the backdrop.

Find the `'nexrad-n0r-wmst'` layer and update it.

```
wmst = wmsfind('nexrad-n0r-wmst', 'SearchField', 'layername');
wmst = wmsupdate(wmst);
```

Find a generated CONUS composite of GOES IR imagery and update it.

```
goes = wmsfind('goes*conus*ir', 'SearchField', 'layername');
goes = wmsupdate(goes);
```

Create a figure with the desired geographic extent.

```
hfig = figure;
region = 'conus';
usamap(region)
mstruct = gcm;
latlim = mstruct.maplatlimit;
lonlim = mstruct.maplonlimit;
```

Read the GOES layer as a backdrop image.

```
cellsize = .1;
[backdrop, R] = wmsread(goes, 'ImageFormat', 'image/png', ...
   'Latlim', latlim, 'Lonlim', lonlim, 'Cellsize', cellsize);
```

Calculate current time minus 24 hours and set up frames to hold the data from `getframe`.

```
now_m24 = datestr(now-1);
hour_m24 = [now_m24(1:end-5) '00:00'];
hour = datenum(hour_m24);
hmap = [];
numFrames = 24;
frames = struct('cdata', [], 'colormap', []);
frames(numFrames) = frames;
```

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

```
borders = geoshow('usastatehi.shp', 'FaceColor', 'none');
black = [0,0,0];
threshold = 0;
for k=1:numFrames
    time = datestr(hour);
    [A, R] = wmsread(wmst, 'Latlim', latlim, 'Lonlim', lonlim, ...
        'Time', time, 'CellSize', cellsize, ...
        'BackgroundColor', black, 'ImageFormat', 'image/png');
    delete(hmap)
    index = any(A > threshold, 3);
    combination = backdrop;
    index = cat(3,index,index,index);
    combination(index) = uint8(255*A(index));
```

```
    hmap = geoshow(combination, R);
    uistack(borders,'top')
    title({wmst.LayerName, time})
    drawnow
    frames(k) = getframe(hfig);
    hour = hour + 1/24;
end
```

View the movie loop.

```
numTimes = 10;
fps = 1.5;
movie(hfig, frames, numTimes, fps);
```

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Retrieve Data from Web Map Server

In this section...

“Merge Elevation Data with Rasterized Vector Data” on page 9-42

“Display Merged Elevation and Bathymetry Layer (SRTM30 Plus)” on page 9-44

“Drape WMS Imagery onto Elevation Data” on page 9-46

Typically, a WMS server returns a pictorial representation of a layer (or layers) back to a requester rather than the actual data. However, in some rare cases, you can request actual data from specific WMS servers, by using a certain option with `wmsread`.

A WMS server renders one or more layers and stores the results in a file, which is streamed to the requester. The `wmsread` function, or the `getMap` method of the `WebMapServer` object, makes the request on your behalf, captures the stream in a temporary file, and imports the file content into a variable in your MATLAB workspace. The format of the file may be a standard graphics format, such as JPEG, PNG, or GIF, or it may be the band-interleaved by line (BIL) format, which is popular in remote sensing. Almost all WMS servers support use of the JPEG format, and many support more than one standard graphics format. Only a very few WMS servers support the BIL format, although it is very useful.

The choice of format can affect the quality of your results. For example, PNG format avoids the tile-related artifacts that are common with JPEG. Format choice also determines whether you will get a pictorial representation, which is the case with any of the standard graphics formats, or an absolutely quantitative data grid (possibly including negative as well as positive values). Quantitative data sets are provided via the BIL format.

`Note` To request actual data, most often you need to create either a Web Coverage Service (WCS) request, for raster data, or a Web Feature Service (WFS) request, for vector data. The Mapping Toolbox does not support WCS and WFS requests.

With a server that supports multiple formats, you can control which format is used by specifying an `ImageFormat` name-value pair when calling `wmsread`. For example, if the server supports the PNG format you would choose PNG by specifying `'ImageFormat','image/png'`, thus avoiding the possibility of JPEG artifacts.

With a server that supports it, you can obtain an absolutely quantitative data grid by specifying BIL format when calling `wmsread`. To do this, use the name-value pair `'ImageFormat','image/bil'`. Although a BIL file typically contains multiple, co-registered bands (channels), the BIL files returned by a WMS server include only a single band. In this case, the output of `wmsread` enters the MATLAB workspace as a 2-D array.

For example, you can obtain signed, quantitative elevation data, rather than an RGB image, from the NASA WorldWind WMS server (the only server in the Mapping Toolbox WMS database known to support the `'image/bil'` option). See the output from the command:

```
wmsinfo('https://data.worldwind.arc.nasa.gov/elev?')
```

In fact, because the NASA WorldWind WMS server returns rendered layers only in the `'image/bil'` format, you do not need to provide an `'ImageFormat'` name-value pair when using this server. However, it is good practice to specify the image format, in case the server is ever updated to provide rendered layers in other image formats.

After retrieving data using the `'ImageFormat','image/bil'` option, display it as a surface or a texture-mapped surface, rather than as an image, as shown in the examples below.

Merge Elevation Data with Rasterized Vector Data

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

1 Find the layers from the NASA WorldWind server.

```
layers = wmsfind('data.worldwind*elev', 'SearchField', 'serverurl');
layers = wmsupdate(layers);
```

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

```
aster = layers.refine('EarthAsterElevations30m', ...
                      'SearchField', 'layername');
```

3 Define the region surrounding Afghanistan.

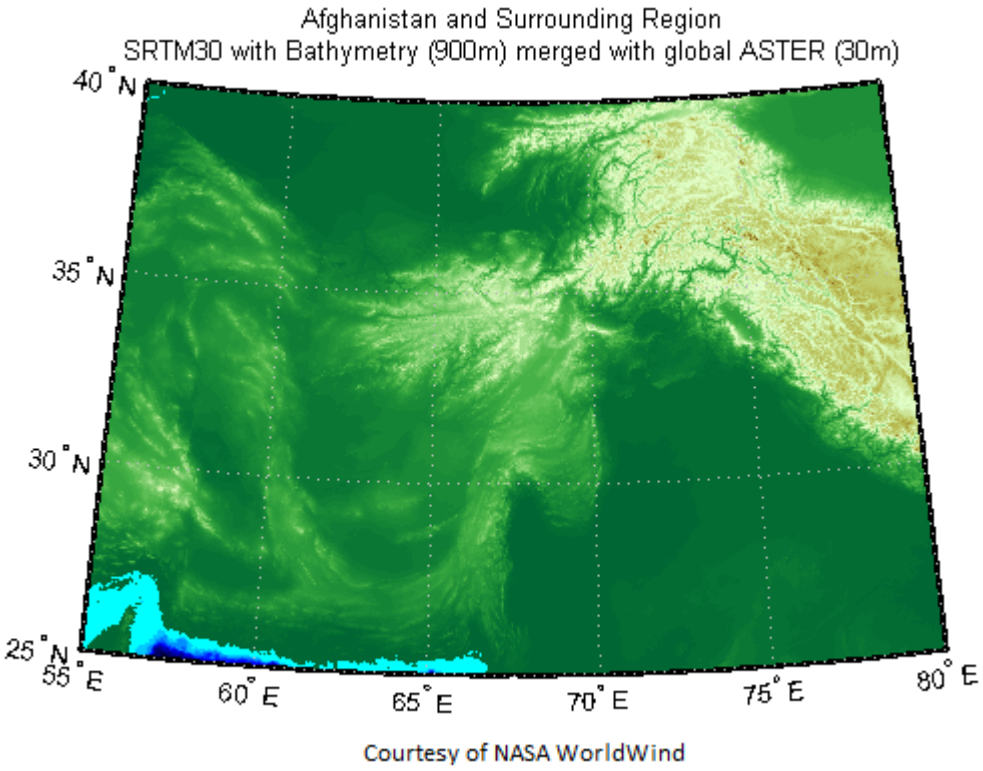
```
latlim = [25 40];
lonlim = [55 80];
```

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

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

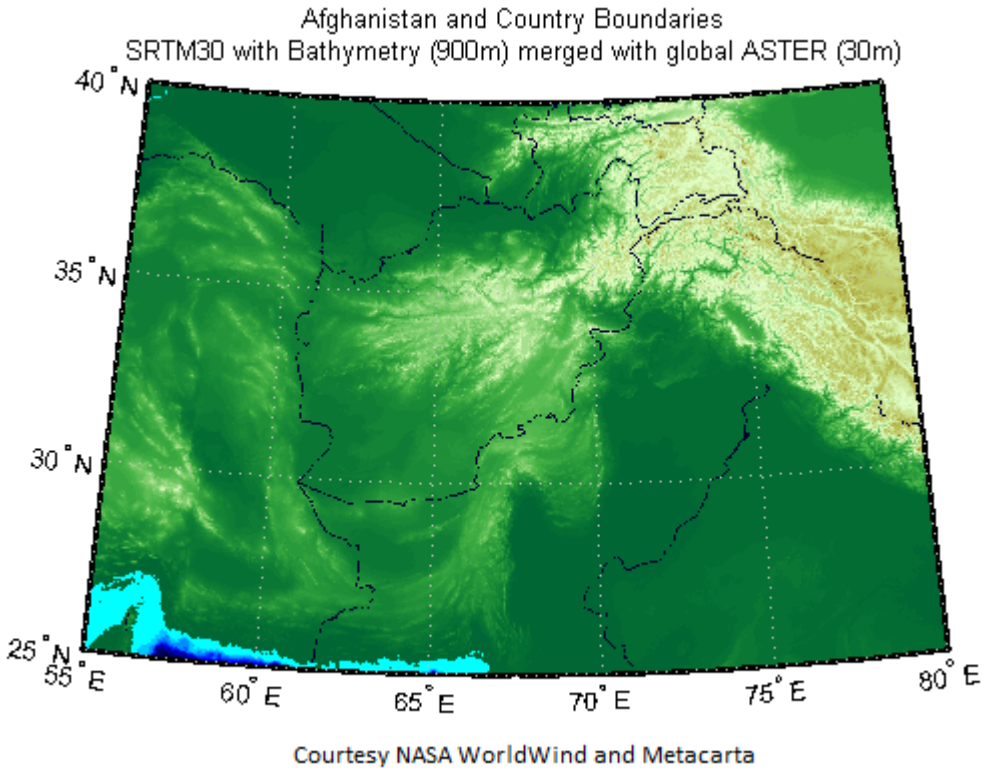
5 Display the elevation data as a texture map.

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



6 Embed national boundaries from the VMAP0 WMS server into the elevation map.

```
vmap0 = wmsfind('vmap0.tiles', 'SearchField', 'serverurl');
boundaries = refine(vmap0, 'country_02');
B = wmsread(boundaries, 'Latlim', latlim, ...
    'Lonlim', lonlim, 'CellSize', cellSize, 'ImageFormat','image/png');
ZB = ZA;
ZB(B(:,:,1) < 250) = min(ZA(:));
figure
worldmap('Afghanistan')
demcmap(double(ZA))
geoshow(ZB, RA, 'DisplayType', 'texturemap')
title({'Afghanistan and Country Boundaries', aster.LayerTitle});
```



Display Merged Elevation and Bathymetry Layer (SRTM30 Plus)

The Shuttle Radar Topography Mission (SRTM) is a project led by the U.S. National Geospatial-Intelligence Agency (NGA) and NASA. SRTM has created a high-resolution, digital, topographic database of Earth. The SRTM30 Plus data set combines GTOPO30, SRTM-derived land elevation and Sandwell bathymetry data from the University of California at San Diego.

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

1 Find and update the `'srtm30'` layer in the WMS Database. The `'srtm30'` layer name from NASA WorldWind is the name for the SRTM30 Plus data set.

```
wldwind = wmsfind('data.worldwind*elev', 'SearchField', 'serverurl');
wldwind = wmsupdate(wldwind);
srtmplus = wldwind.refine('srtm30', 'SearchField', 'layername');
```

2 Set the desired geographic limits.

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

3 Set the sampling interval to 30 arc-seconds.

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

4 Set the `ImageFormat` to `image/bil`.

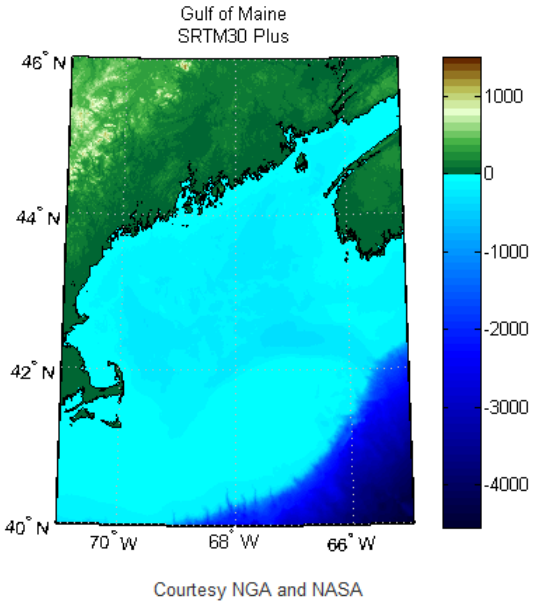
```
imageFormat = 'image/bil';
```

5 Request the map from the NASA server.

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

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

```
figure
worldmap(Z1, R1)
geoshow(Z1, R1, 'DisplayType', 'texturemap')
demcmap(double(Z1))
contourm(double(Z1), R1, [0 0], 'Color', 'black')
colorbar
title ({'Gulf of Maine', srtmplus.LayerTitle}, 'Interpreter','none')
```



7 Compare the NASA WorldWind SRTM30 Plus layer with the SRTM30 with Bathymetry (900m) merged with SRTM3 V4.1 (90m) and USGS NED (30m) (`mergedSrtm`) layer.

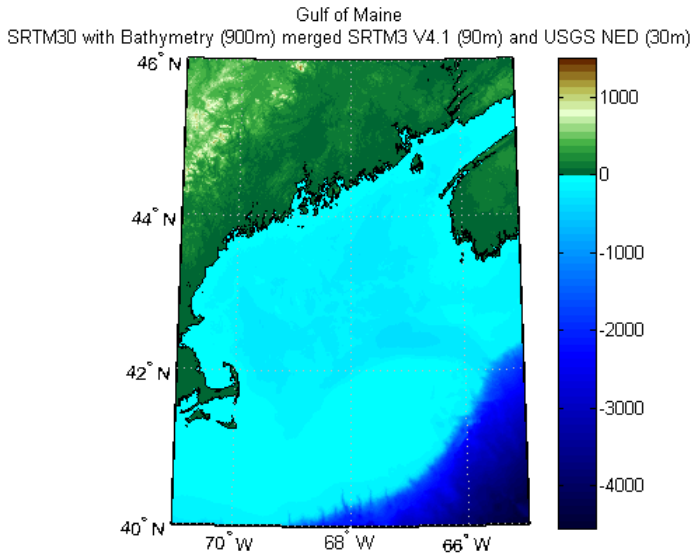
```
mergedSrtm = wldwind.refine('mergedSrtm');
```

8 Request the map from the NASA WorldWind server.

```
[Z2, R2] = wmsread(mergedSrtm, 'Latlim', latlim, 'Lonlim', lonlim, ...
    'CellSize', samplesPerInterval, 'ImageFormat', 'image/bil');
```

9 Display the data.

```
figure
worldmap(Z2, R2)
geoshow(Z2, R2, 'DisplayType', 'texturemap')
demcmap(double(Z2))
contourm(double(Z2), R2, [0 0], 'Color', 'black')
colorbar
title ({'Gulf of Maine', mergedSrtm.LayerTitle})
```



10 Compare the results.

```
disp(newline + "SRTM30 Plus - " + srtmplus.LayerName ...
+ newline + "Minimum value: " + min(Z1(:))...
+ newline + "Maximum value: " + max(Z1(:)))
disp(newline + "SRTM30 Plus Merged - " + mergedSrtm.LayerName ...
+ newline + "Minimum value: " + min(Z2(:))...
+ newline + "Maximum value: " + max(Z2(:)))
```

11 The output appears as follows:

```
SRTM30 Plus - srtm30
Minimum value: -4543
Maximum value: 1463

Merged SRTM30 Plus - mergedSrtm
Minimum value: -4543
Maximum value: 1463
```

Drape WMS Imagery onto Elevation Data

This example shows how to drape WMS imagery onto elevation data from the USGS National Elevation Dataset (NED).

1 Obtain the layers of interest.

```
ortho = wmsfind('/USGSImageryTopo/','SearchField','serverurl');

layers = wmsfind('data.worldwind', 'SearchField', 'serverurl');
us_ned = layers.refine('usgs ned 30');
```

2 Assign geographic extent and image size.

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

3 Read the `ortho` layer.

```
A = wmsread(ortho, 'Latlim', latlim, 'Lonlim', lonlim, ...
   'ImageHeight', imageHeight, 'ImageWidth', imageWidth);
```

4 Read the USGS `us_ned` layer.

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

5 Drape the ortho image onto the elevation data.

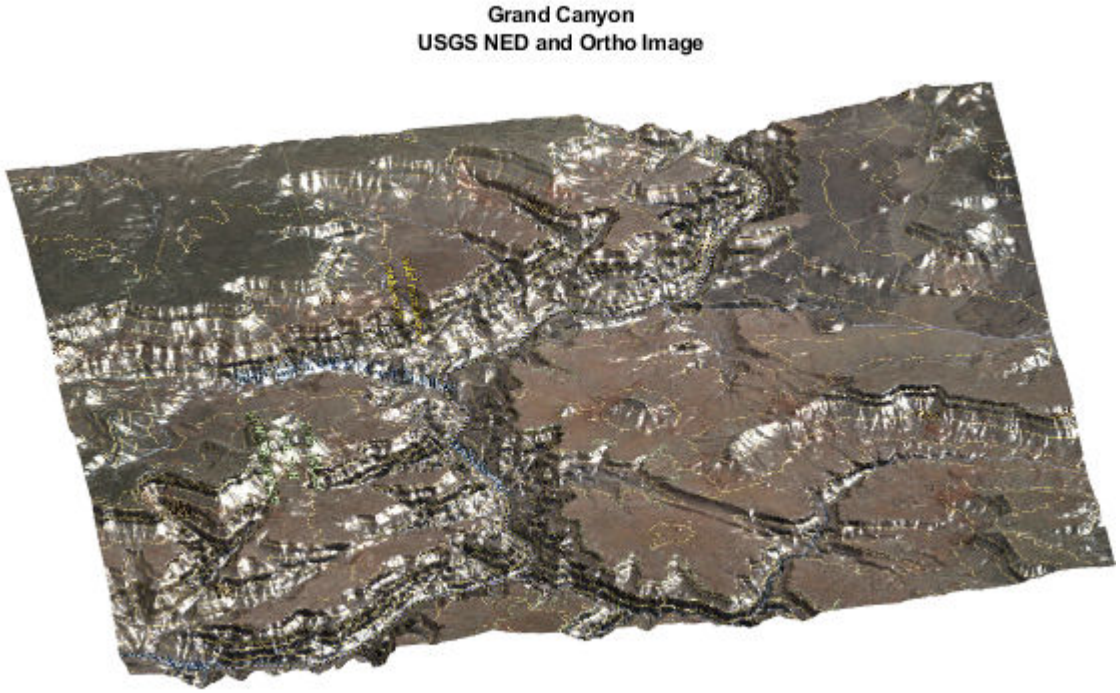
```
figure
usamap(latlim, lonlim)
framem off; mlabel off; plabel off; gridm off
geoshow(double(Z), R, 'DisplayType', 'surface', 'CData', A);
daspectm('m',1)
title({'Grand Canyon', 'USGS NED and Ortho Image'}, ...
   'FontSize',8);
axis vis3d
```

6 Assign camera parameters.

```
cameraPosition = [96431 4.2956e+06 -72027];
cameraTarget = [-82.211 4.2805e+06 3054.6];
cameraViewAngle = 8.1561;
cameraUpVector = [3.8362e+06 5.9871e+05 5.05123e+006];
```

7 Set camera and light parameters.

```
set(gca,'CameraPosition', cameraPosition, ...
    'CameraTarget', cameraTarget, ...
    'CameraViewAngle', cameraViewAngle, ...
    'CameraUpVector', cameraUpVector);
lightHandle = camlight;
camLightPosition = [7169.3 1.4081e+06 -4.1188e+006];
set(lightHandle, 'Position', camLightPosition);
```



See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Save Your Favorite Servers

You can save your favorite layers for easy access in the future. Use `wmsupdate` to fill in the `Abstract`, `CoordRefSysCodes`, and `Details` fields, and then save the layers. The next example demonstrates how to make a mini-database from the NASA, WHOI, and ESA servers.

1 Find the servers and update all fields.

```
nasa = wmsfind('nasa','SearchField','serverurl');
favoriteLayers = nasa;
favoriteLayers = wmsupdate(favoriteLayers, ...
   'AllowMultipleServers', true);
favoriteServers = favoriteLayers.servers;
```

2 Save your favorite layers in a MAT-file.

```
save favorites favoriteLayers
```

3 Search within your favorite layers for `'wind speed'`. You have updated all fields, so you can search within any field, including the `Abstract`.

```
windSpeed = favoriteLayers.refine('wind speed','SearchFields','any')
```

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Explore Other Layers using a Capabilities Document

You may find a layer you like in the WMS Database and then want to find other layers on the same server.

1 Use the `wmsinfo` function to return the contents of the capabilities document as a `WMSCapabilities` object. A capabilities document is an XML document containing metadata describing the geographic content offered by a server.

```
serverURL = 'http://svs.gsfc.nasa.gov/cgi-bin/wms?';
capabilities = wmsinfo(serverURL)
```

```
capabilities =

  WMSCapabilities

  Properties:
         ServerTitle: 'NASA SVS Image Server'
           ServerURL: 'http://svs.gsfc.nasa.gov/cgi-bin/wms?'
         ServiceName: 'WMS'
             Version: '1.3.0'
            Abstract: 'Web Map Server maintained by the
                       Scientific Visualization
                       Studio at NASA's Goddard Space Flight Center'
      OnlineResource: 'http://svs.gsfc.nasa.gov/'
  ContactInformation: [1x1 struct]
   AccessConstraints: 'none'
                Fees: 'none'
         KeywordList: {}
        ImageFormats: {'image/png'}
          LayerNames: {326x1 cell}
               Layer: [326x1 WMSLayer]
          AccessDate: '09-Jan-2017'

  Methods
```

2 View the layer names and layer titles.

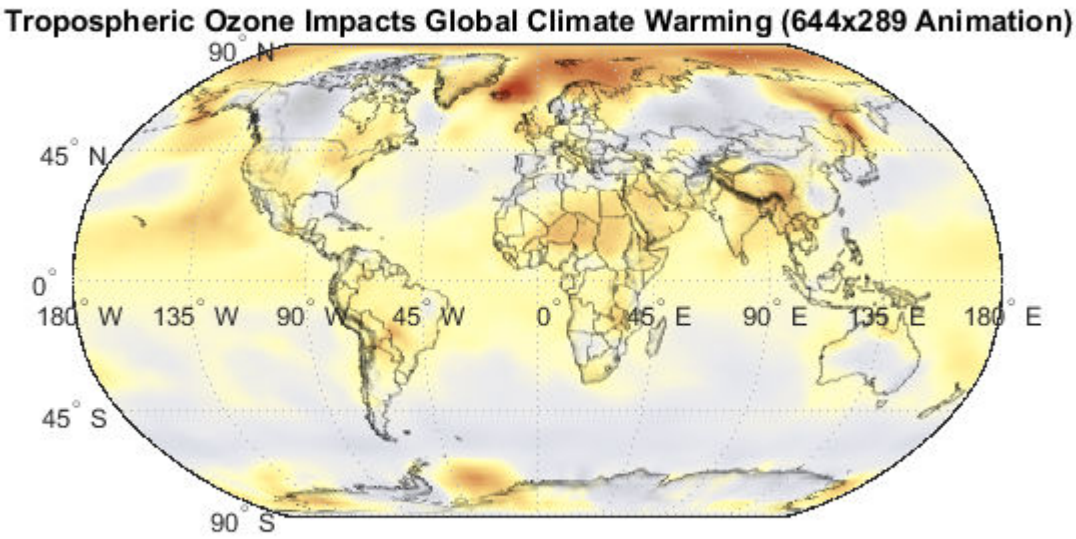
```
capabilities.LayerNames;

layerTitles = {capabilities.Layer.LayerTitle}';
```

3 Read the layer containing tropospheric ozone impacts.

```
layerTitle = 'Tropospheric Ozone Impacts Global Climate Warming';
layer = refine(capabilities.Layer, layerTitle);
[A, R] = wmsread(layer);
```

4 Display the map.

```
figure
worldmap(A,R)
geoshow(A,R)
title(layer.LayerTitle)
```

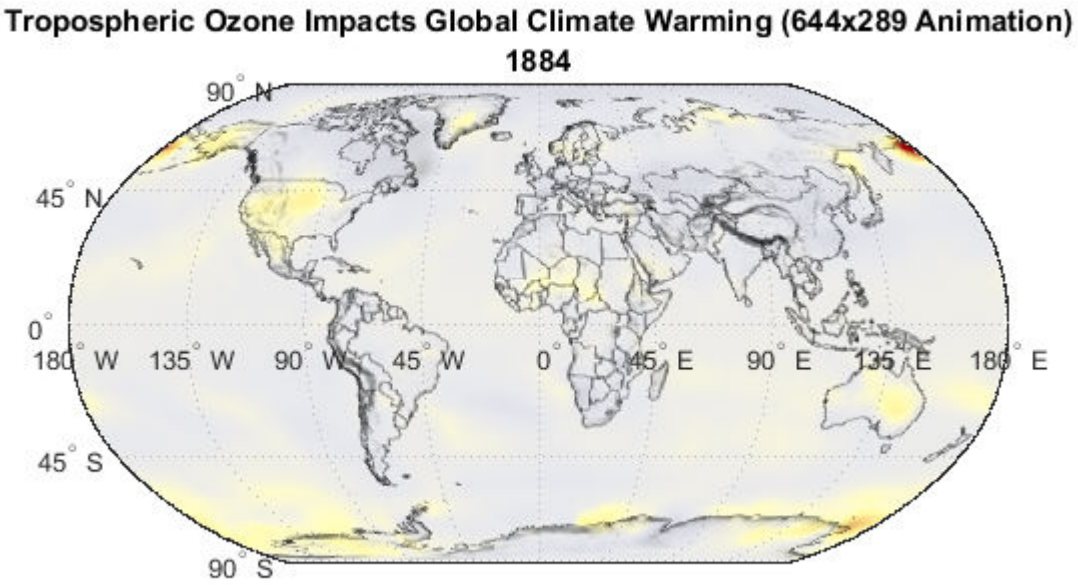


5 This layer contains data from different years. You can examine the available data by viewing the `layer.Details.Dimension` structure.

```
layer.Details.Dimension
```

6 Display the map for the year 1884 and compare it with the map for 1994, the default year (displayed previously).

```
year = '1884';
[A2,R] = wmsread(layer,'Time',year);
figure
worldmap(A2,R)
geoshow(A2,R)
title({layer.LayerTitle, year})
```



See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Write WMS Images to a KML File

Some WMS server implementations, such as GeoServer, can render their maps in a non-image format, such as KML. KML is an XML dialect used by Google Earth and Google Maps browsers. The `WebMapServer.getMap` method and the `wmsread` function do not allow you to use the KML format because they import only standard graphics image formats. Work around this limitation by using the `WMSMapRequest.RequestURL` property.

1 Search the WMS Database for layers on any GeoServer. Refine to include only the layers from the MassGIS server. Refine that list to return a FEMA Flood Zone layer.

```
geoserver = wmsfind('geoserver', 'SearchField', 'any');
massgis = geoserver.refine('massgis*wms', 'SearchField', ...
   'serverurl');
massgis = wmsupdate(massgis);
floodzone = massgis.refine('FEMA Flood Zones', 'SearchField', ...
   'LayerTitle');
floodzone = floodzone(1);
```

2 Set geographic limits for a region around Boston, Massachusetts.

```
latlim = [ 42.305  42.417];
lonlim = [-71.131 -70.99];
```

3 Create a `WMSMapRequest` object and set the geographic limits.

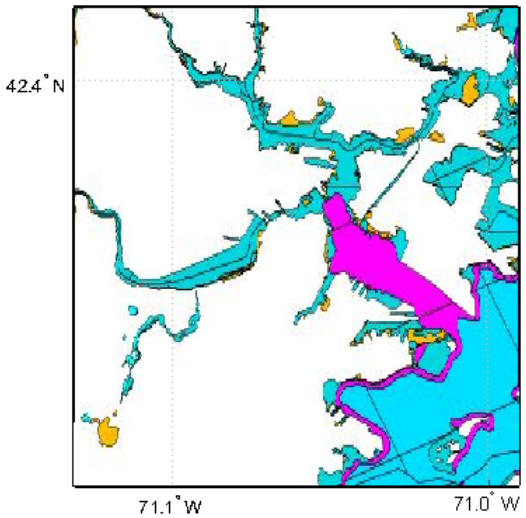
```
request = WMSMapRequest(floodzone);
request.Latlim = latlim;
request.Lonlim = lonlim;
```

4 Obtain the graphics image from the server.

```
[A, R] = wmsread(request.RequestURL);
```

5 Display the image in a figure window.

```
figure
usamap(A, R)
geoshow(A, R)
```



Courtesy the Office of Geographic and Environmental Information (MassGIS)

6 Request an image format that opens in Google Earth.

```
request.ImageFormat = 'application/vnd.google-earth.kml+xml';
```

7 Use the `urlwrite` function to write out a KML file.

```
filename = 'floodzone.kml';
websave(filename,request.RequestURL);
```

8 Open the file with Google Earth to view. On Windows® platforms, 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])
```

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Search for Layers Outside the Database

You can search for layers by using your Web browser rather than by using the WMS Database.

1. To search for layers outside the WMS Database, use your favorite search engine. If you are using Google®, select **Images** and enter the following in the search box: `getmap wms`.
2. View the images to choose a map. Click the map link and find the WMS `GetCapabilities` request somewhere on the page. If you cannot find a `GetCapabilities` request, try another map.

   For this example, the syntax for the URL of the WMS `GetCapabilities` request appears as follows:

   ```
   url = ['http://sampleserver1.arcgisonline.com/' ...
      'ArcGIS/services/Specialty/ESRI_StatesCitiesRivers_USA/' ...
      'MapServer/WMSServer?service=WMS&request=GetCapabilities' ...
      '&version=1.3.0'];
   ```
3. After you obtain the URL, you can use `wmsinfo` to return the capabilities document.

   ```
   c = wmsinfo(url);
   ```
4. Next, read in a layer and display it as a map.

   ```
   [A,R] = wmsread(c.Layer(1), ...
      'BackgroundColor', [0,0,255], 'ImageFormat', 'image/png');
   figure
   usamap(c.Layer(1).Latlim, c.Layer(1).Lonlim)
   geoshow(A,R)
   ```

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Troubleshoot WMS Servers

In this section...

"Connection Errors" on page 9-56
"Wrong Scale" on page 9-57
"Problems with Geographic Limits" on page 9-58
"Problems with Server Changing LayerName" on page 9-58
"Non-EPSG:4326 Coordinate Reference Systems" on page 9-58
"Map Not Returned" on page 9-59
"Unsupported WMS Version" on page 9-60
"Other Unrecoverable Server Errors" on page 9-60

Connection Errors

One of the challenges of working with WMS is that sometimes you can have trouble connecting to a server.

Time-Out Error

A server may issue a time-out error such as:

```
Connection timed out: connect
```

Or

```
Read timed out
```

Workaround: Try setting the `'TimeoutInSeconds'` parameter to a larger value. The time-out setting defaults to 60 seconds. (The functions `wmsread`, `wmsinfo`, and `wmsupdate` all have `'TimeoutInSeconds'` parameters.)

Server No Longer Provides Full WMS Services

The NASA Jet Propulsion Laboratory (JPL) server may issue the following error message:

```
This server no longer provides full WMS services!
```

The DataFed server, `http://webapps.datafed.net/OnEarth_JPL.ogc?`, cascades layers from the JPL server and may issue the following error message:

```
Error in Execution. Cannot fetch url.
```

Workaround: Use a `TiledWMS` URL or find a different server.

The JPL Global Imagery Service server, `http://onearth.jpl.nasa.gov/wms.cgi?`, is no longer providing full WMS services for any of the datasets. Any server (for example, `http://webapps.datafed.net/OnEarth_JPL.ogc?`) that cascades data from this server is also affected by the change.

A small subset of the data can be accessed using a non-standard `TiledWMS` request. The available tiled patterns can be found at:

```
http://pat.jpl.nasa.gov/wms.cgi?request=GetTileService
```

The WMS parameters must be in the exact order. If you wish to obtain a tile, you can prepend the prefix, `'http://onearth.jpl.nasa.gov/wms.cgi?/SERVICE=WMS&'` in front of the request found in the `CDATA` section of the `GetTileService` request.

For example:

```
url = ['http://onearth.jpl.nasa.gov/wms.cgi?/SERVICE=WMS&Version=1.1.1&' ...
'request=GetMap&layers=global_mosaic&srs=EPSG:4326&'  ...
'format=image/jpeg&styles=visual&width=512&height=512&' ...
'bbox=-180,58,-148,90'];
[A, R] = wmsread(url);
```

Elevation layers from `onearth.jpl.nasa.gov` can be replaced with layers from the NASA WorldWind server (`https://data.worldwind.arc.nasa.gov/elev?`). The Blue Marble layer can be replaced with a Blue Marble layer from the NASA Goddard Space Flight Center WMS SVS Image Server (`http://svs.gsfc.nasa.gov/cgi-bin/wms?`) or the Blue Marble: Next Generation layer from the NASA Earth Observations (NEO) WMS Server (`http://neowms.sci.gsfc.nasa.gov/wms/wms?`).

The Daily Planet layer can be replaced with the `'True Color (1 day - Terra/MODIS Rapid Response)'` layer from the NASA Earth Observations (NEO) WMS server.

HTTP Response Code 500

In some cases, the server becomes temporarily unavailable or the WMS server application experiences some type of issue. The server issues an HTTP response code of 500, such as:

```
Server returned HTTP response code: 500 for URL: http://xyz.com ...
```

Workaround: Try again later. Also try setting a different `'ImageFormat'` parameter.

WMSServlet Removed

If the `columbo.nrlssc.navy.mil` server issues an error such as:

```
WebMapServer cannot communicate to the host columbo.nrlssc.navy.mil.
The host is unknown.
```

This message indicates that the server it is trying to access is no longer available.

Workaround: Choose a different layer.

Wrong Scale

The `columbo.nrlssc.navy.mil` server often throws this error message:

```
This layer is not visible for this scale. The maximum valid scale
is approximately X. Zoom in and try again if desired. The scale of
the image requested is Y.
```

`X` and `Y` represent specific values that vary from layer to layer.

Workaround: Some of the WMS sources this server accesses have map layers sensitive to the requested scale. Zoom in (choose a smaller region of interest), or zoom out (choose a larger region of

interest). Alternatively, you can select a larger output image size to view the layer at the appropriate scale.

Problems with Geographic Limits

Some servers do not follow the guidelines of the OGC specification regarding latitude and longitude limits.

Latlim and Lonlim in Descending Order

The OGC specification requires, and the WMS functions expect, that the limits are ascending. Some sites, however, have descending limits. As a result, you may get this error message:

```
"??? Error using ==> WMSMapRequest>validateLimit at 1313
Expected the elements of 'Latlim' to be in ascending order."
```

Workaround: To address this problem, set the `Latlim` and `Lonlim` properties of `WMSLayer`:

```
layer = wmsfind('SampleServer.com', 'SearchField', 'serverurl');
layer = wmsupdate(layer);
latlim = [min(layer.Latlim), max(layer.Latlim)];
lonlim = [min(layer.Lonlim), max(layer.Lonlim)];
layer.Latlim = [max([ -90, latlim(1)]), min([ 90, latlim(2)])];
layer.Lonlim = [max([-180, lonlim(1)]), min([180, lonlim(2)])];
[A,R] = wmsread(layer);
```

Update your layer before setting the limits. Otherwise, `wmsread` updates the limits from the server, and you once again have descending limits.

Limits Exceed Bounds

Some servers have limits that exceed the bounds of [-180, 180] for longitude and [-90, 90] for latitude.

Workaround: To address this problem, follow the same procedure outlined in “Latlim and Lonlim in Descending Order” on page 9-58.

Problems with Server Changing LayerName

In most cases, the updated layer returned by `wmsupdate` should have `ServerURL` and `LayerName` properties that match those of the layer you enter as input. In some cases when the layer is updated from the `columbo.nrlssc.navy.mil` server, the server returns a layer with a different `LayerName`, but the `ServerURL` and `LayerTitle` are the same. The layers from the `columbo.nrlssc.navy.mil` server have names such as `'X:Y'`, where `X` and `Y` are ASCII numbers. Since the time of your last update, a layer has been added to or removed from the server causing a shift in the sequence of layers. Since the `LayerName` property is constructed with ASCII numbers based on the layer's position in this sequence, the `LayerName` property has changed. For layers from the `columbo.nrlssci.navy.mil` server, `wmsupdate` matches the `LayerTitle` property rather than the `LayerName` property.

Non-EPSG:4326 Coordinate Reference Systems

Some layers are not defined in the EPSG:4326 or CRS:84 coordinate reference system. You cannot read these layers with the `wmsread` function.

Workaround: Use the `WMSMapRequest` object to construct a request URL and the `WebMapServer.getMap` method to read the layer. See Understanding Coordinate Reference System Codes on page 9-10 and Retrieving Your Map with `WebMapServer.getMap` on page 9-19 for more information.

Map Not Returned

Sometimes you can connect to the WMS server, but you do not receive the map you are expecting.

Blank Map Returned

A server may return a blank map.

Workaround: You can change the scale of your map; either increase the image height and width or change the geographic bounds. Another possibility is that your requested geographic extent lies outside the extent of the layer, in which case you should change the extent of your request. A third possibility is that you have the wrong image format selected; in this case, change the `'ImageFormat'` parameter.

HTML File Returned

You may receive this error message:

```
The server returned an HTML file instead of an image file.
```

Workaround: Follow the directions in the error message. The following example, which uses a sample URL, illustrates the type of error message you receive.

```
% Example command.
[A,R] = wmsread(['https://www.mathworks.com?',...
'&BBOX=-180,-90,180,90&CRS=EPSG:4326&VERSION=1.1.1']);
```

Sample error message:

```
Error using WebMapServer>issueReadGetMapError (line 974)
The server returned an HTML file instead of an image file.
You may view the complete error message by issuing the command,
 web('https://www.mathworks.com?&BBOX=-180,-90,180,90&CRS=EPSG:4326&VERSION=1.1.1')
  or
 urlread('https://www.mathworks.com?&BBOX=-180,-90,180,90&CRS=EPSG:4326&VERSION=1.1.1').

Error in WebMapServer>readImageFormat (line 874)
        issueReadGetMapError(filename, requestURL);

Error in WebMapServer>readGetMapFile (line 852)
    A = readImageFormat(filename, requestURL);

Error in WebMapServer/getMap (line 299)
            A = readGetMapFile(filename, h.RequestURL);

Error in wmsread (line 376)
A = server.getMap(mapRequestURL);
```

XML File Returned

The server issues a very long error message, beginning with the following phrase:

```
An error occurred while attempting to get the map from the server.
The error returned is <?xml version="1.0" encoding="utf-8"?> ...
```

Workaround: This problem occurs because the server breaks with the requirements of the OGC standard and returns the XML capabilities document rather than the requested map. Choose a different layer or server.

Unsupported WMS Version

In rare cases, the server uses a different and unsupported WMS version. In this case, you receive an error message such as:

```
The WMS version, '1.2.0', listed in layer.Details.Version is not
supported by the server. The supported versions are: '1.0.0' '1.1.0'
'1.1.1' '1.3.0' .
```

Workaround: Choose a different server.

Other Unrecoverable Server Errors

The server issues an error indicating that no correction or workaround exists. These cases result in the following types of error messages:

```
Server redirected too many  times (20)

An error occurred while attempting to parse the XML capabilities
document from the server.

Unexpected end of file from server

An error occurred while attempting to get the map from the server.
The server returned a map containing no data.
```

See Also

`wmsfind` | `wmsread` | `wmsupdate`

More About

- “Basic Workflow for Creating WMS Maps” on page 9-3

Troubleshoot Access to the Hosted WMS Database

`wmsfind` can search a version of the WMS database hosted on the MathWorks website. The information found in the web-hosted database is updated regularly.

If your network uses a firewall or another method of protection that restricts Internet access, provide information about your proxy server to MATLAB. Be aware that:

- MATLAB supports non-authenticated, basic, digest, and NTLM proxy authentication types.
- You cannot specify the proxy server settings using a script.
- There is no automated way to provide the proxy server settings your system browser uses to MATLAB.

To specify the proxy server settings:

1. On the **Home** tab, in the **Environment** section, click **Preferences**. Select **MATLAB > Web**.
2. Select the **Use a proxy server to connect to the Internet** check box.
3. Specify values for **Proxy host** and **Proxy port**. Examples of acceptable formats for the host are: `172.16.10.8` and `ourproxy`. For the port, enter an integer only, such as `22`. If you do not know the values for your proxy server, ask your system or network administrator for the information. If your proxy server requires a user name and password, select the **Use a proxy with authentication** check box. Then enter your proxy user name and password. MATLAB stores the password without encryption in your `matlab.prf` file.
4. Ensure that your settings work by clicking the **Test connection** button. MATLAB attempts to connect to `https://www.mathworks.com`. If MATLAB can access the Internet, **Success!** appears next to the button. If MATLAB cannot access the Internet, **Failed!** appears next to the button. Correct the values you entered and try again. If you still cannot connect, try using the values you used when you authenticated your MATLAB license.
5. Click **OK** to accept the changes.

On Windows®, if no proxy is specified in MATLAB preferences, `wmsfind` uses the proxy set in the Windows system preferences. To specify system proxy server settings, refer to your Windows documentation for locating Internet Options. On the **Connections** tab, select **LAN settings**. The proxy settings are in the Proxy server section. MATLAB does not take into account proxy exceptions which you configure in Windows. Even if you have specified the correct credentials in the MATLAB preference panel or in the Windows system proxy settings, the `wmsfind` function returns the error `Proxy Authentication Required` if the proxy server in MATLAB preferences requires an authentication method other than Basic. The proxy server in Windows system preferences requires authentication of any type.

Introduction to Web Map Display

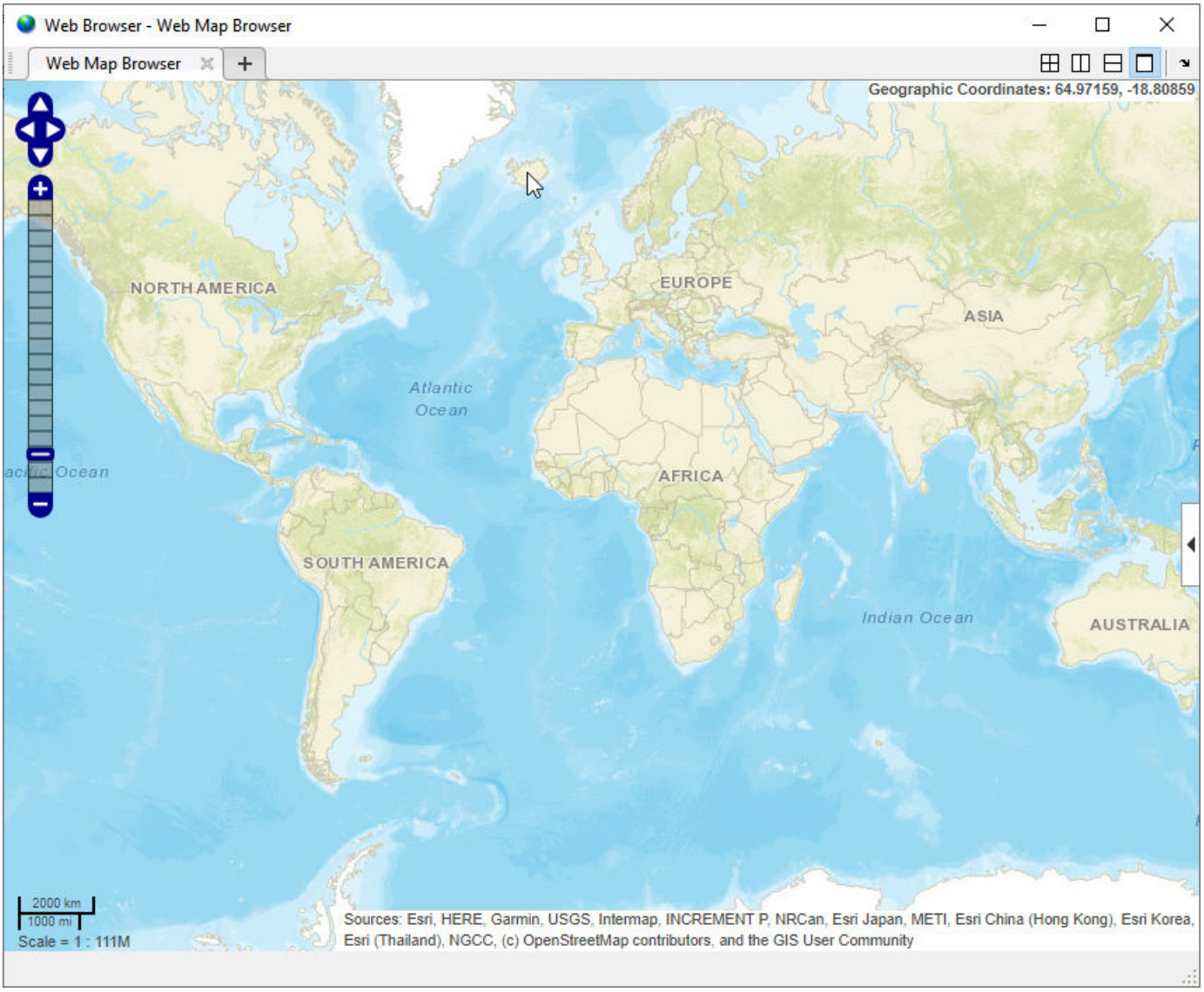
Web maps are interactive maps that are accessed through web pages. As a result, they require a live Internet connection. Using Mapping Toolbox software, you can:

- Display web maps in a browser.
- Interactively or programmatically pan and zoom.
- Select the maps to display, called base layers or basemaps. The `webmap` function provides a set of basemaps from which you can choose, for example `'Open Street Map'`. You can also use WMS Layers and define custom basemaps.
- Add vector data, called overlay layers, such as lines and markers.
- Share your results using printing capabilities or the MATLAB `publish` command.

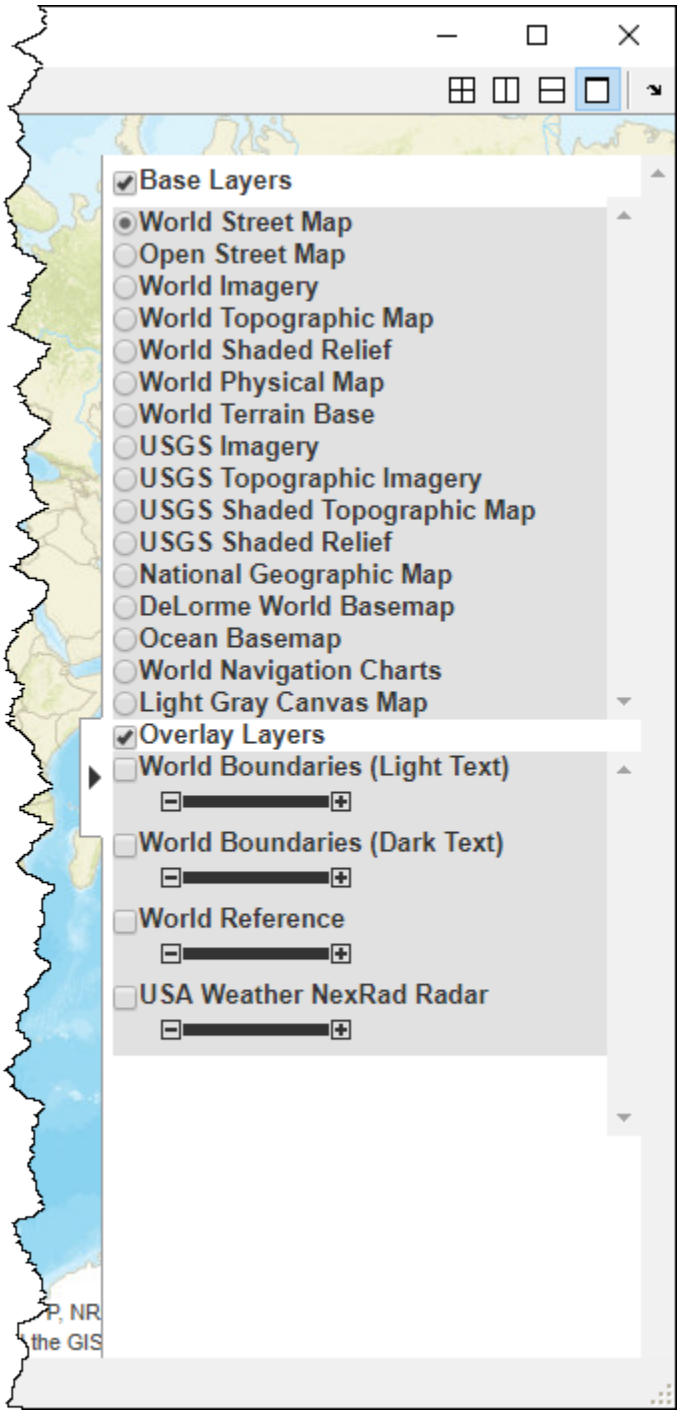
`Note` To display a web map using the toolbox, you must have an Internet connection. MathWorks cannot guarantee the stability and accuracy of web maps, as the servers are located on the Internet and are independent from MathWorks. Occasionally, maps may be slow to display, display partially, or fail to display, because web map servers can become unavailable for short periods of time.

Web maps display in a browser window, not a MATLAB figure window. The graphics in web maps are not part of MATLAB graphics. In MATLAB Online, multiple web maps appear as separate browser windows instead of tabs in a single browser window.

For example, this image shows the default web map display, including the pan tool, zoom tool, scale bar, Layer Manager expander arrow, current pointer location, and web map tabs.



Open the Layer Manager by clicking the expander arrow ◂ in the web map toolbar. Use the Layer Manager to select a basemap layer and display overlay layers.



Web Map Coordinate Systems

When displaying named base layers, or a `WMSLayer` array in a coordinate reference system of EPSG:900913, the projected coordinate system is "Web Mercator". Otherwise, when displaying a `WMSLayer` array, the projected coordinate system is EPSG:4326. These projections include a

geographic quadrangle bounded north and south by parallels (which map to horizontal lines) and east and west by meridians (which map to vertical lines).

See Also

`addCustomBasemap` | `webmap` | `wmcenter` | `wmclose` | `wmlimits` | `wmzoom`

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

Basic Work ow for Displaying Web Maps

Work ow Summary

The web map display is an interactive capability, so there is no specific workflow required. The following is one way to approach working with web map displays.

1. Display the default web map, using the `webmap` function. You can also specify a base layer (also called a basemap) when you create the web map with the `webmap` function.
2. Select a base layer map from the Layer Manager. The toolbox supports over a dozen base layers from popular web map providers. You can also add custom base layers.
3. Navigate around the web map, using the zoom tool and moving the map interactively (panning). You can also specify the visible portion of the web map programmatically using the `wmlimits`, `wmzoom`, and `wmcenter` functions.
4. Select additional layers to overlay on your web map from the overlay layers listed in the Layer Manager. You can also create overlay layers using the `wmline`, `wmmarker`, and `wmpolygon` functions. Use `wmremove` to remove layers that you've added.
5. Print the map, using the `wmprint` function.
6. Close the map, using the `wmclose` function.

See Also

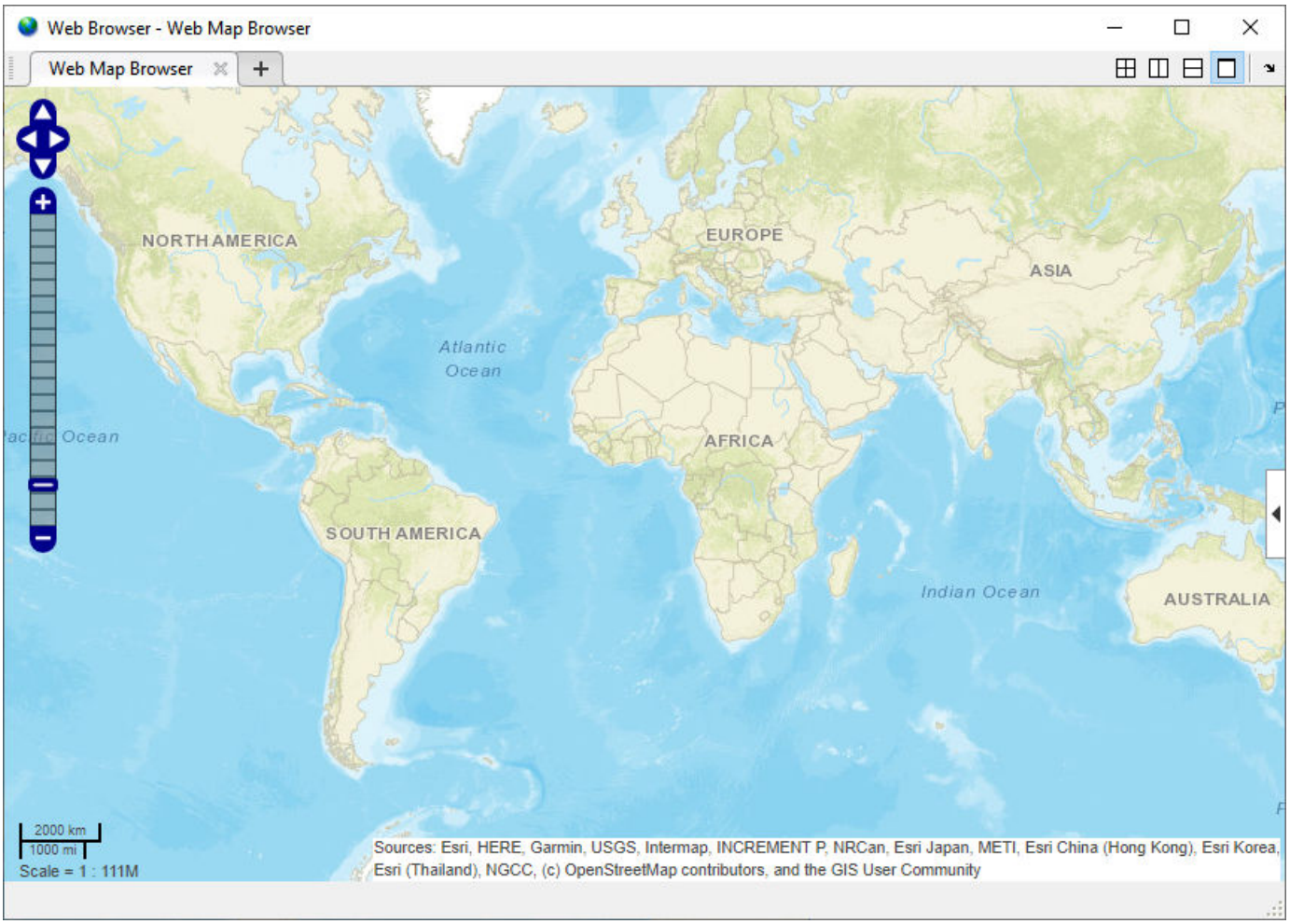
More About

- "Introduction to Web Map Display" on page 9-62
- "Display a Web Map" on page 9-67
- "Select a Base Layer Map" on page 9-68
- "Specify a Custom Base Layer" on page 9-70

Display a Web Map

To display web map data in a web browser, use the `webmap` function. By default, `webmap` displays the World Street Map, centered at latitude and longitude [0 0], but you select other base layers from the Layer Manager. Web maps are interactive, which means you can navigate the map interactively by using the pan and zoom controls, your mouse, or the arrow keys. By default, you can pan across the map continuously, across the 180 meridian.

```
webmap
```



See Also

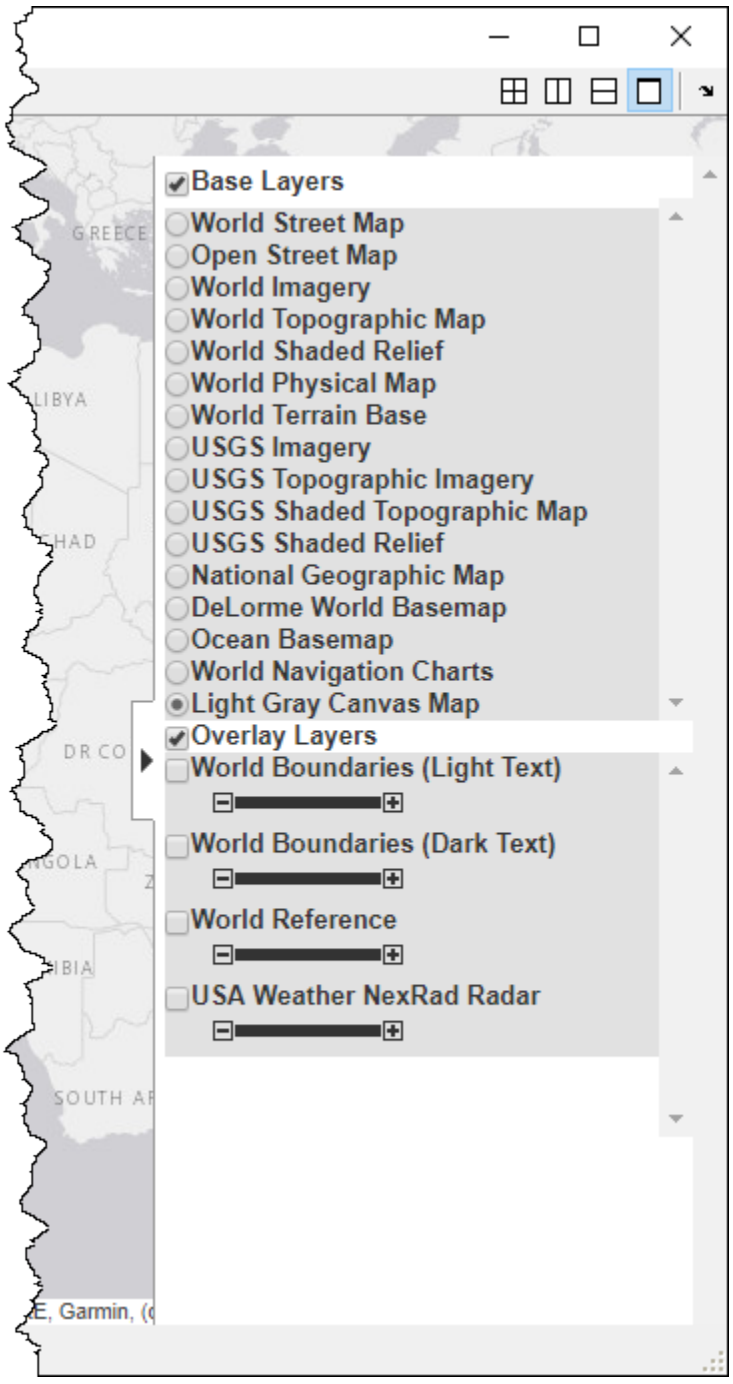
`webmap`

More About

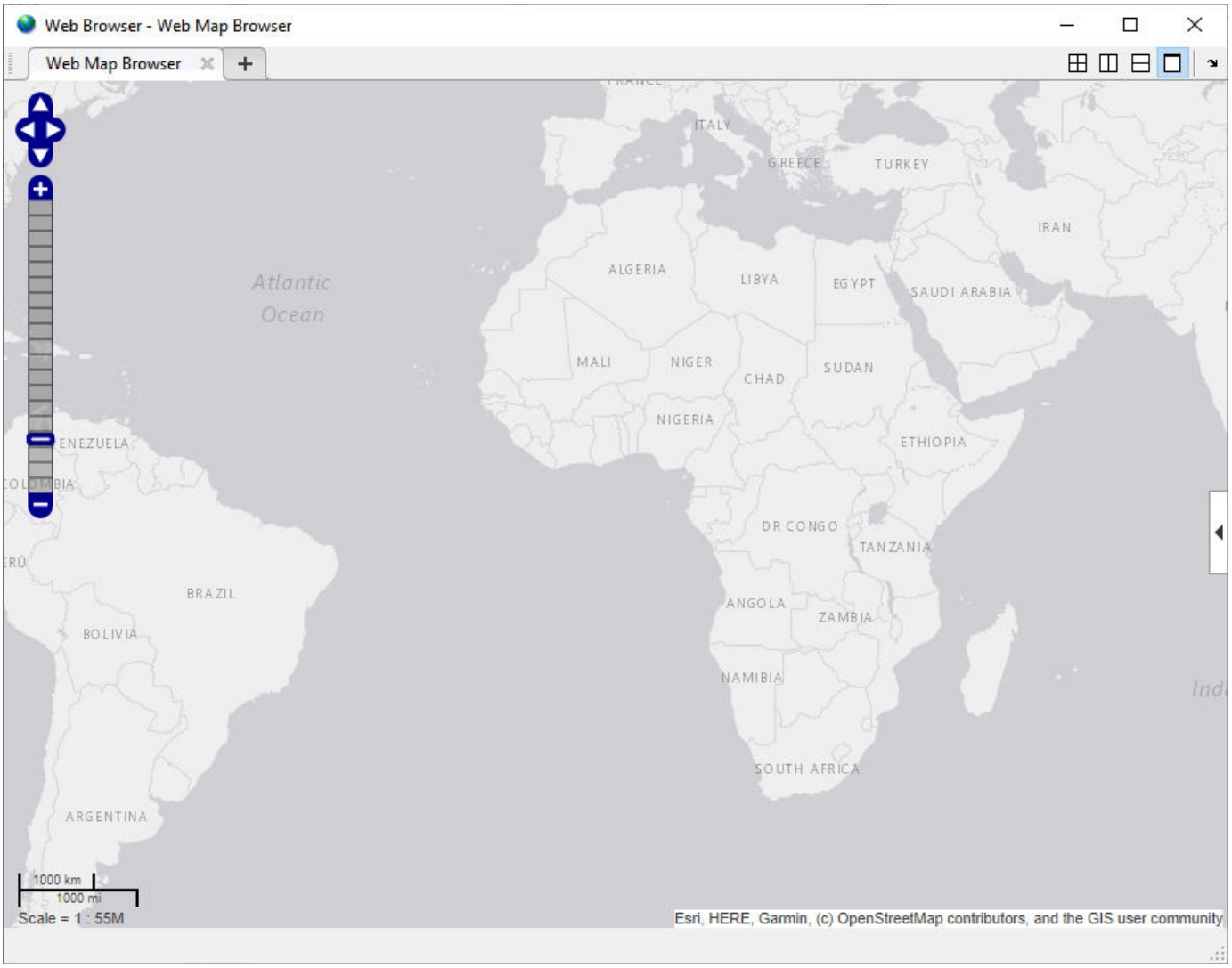
- “Basic Workflow for Displaying Web Maps” on page 9-66
- “Select a Base Layer Map” on page 9-68

Select a Base Layer Map

Once you open a web map, you can change the base layer (basemap) by using the Layer Manager. To open the Layer Manager, select the expander arrow on the right side of the browser window. For example, change the base map to Light Gray Canvas Map. This map is useful for displaying vector data.



Close the Layer Manager.



You can also specify the base layer programmatically when you open the web map, by including the name of the layer as an argument to the `webmap` function. The following example opens the web map browser, displaying the Light Gray Canvas Map. For a list of all the named base layers supported, see the `webmap` function. The examples includes the optional parameter `Wraparound` that causes the map display to be truncated at the -180 degree and +180 degree meridians. By default, maps are continuous.

```
webmap('Light Gray Canvas Map','WrapAround',false)
```

See Also

`webmap` | `wmsfind` | `wmsupdate`

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

Specify a Custom Base Layer

The `webmap` function provides a selection of over a dozen base layers (basemaps) which provide a variety of geographic backdrops on which you can plot your data. See the `webmap` function for a complete list. In some cases, you might want to plot your data over a map of your own choosing. To do this, specify a custom base layer by using the `addCustomBasemap` function. The following example shows how to specify a high-resolution topographical map as a custom base layer.

1. Specify the URL of the website that provides the map data. In this example, for better load balancing, the web map provides three servers that you can use: a, b, or c.

   ```
   url = 'a.tile.opentopomap.org';
   ```

2. Define the name that you will use to specify the custom base layer programmatically. For example, you can use this name with the `webmap` command or, if you want to delete the custom map, with the `removeCustomBasemap` function.

   ```
   name = 'opentopomap';
   ```

3. Create an attribution to display on the map that gives credit to the provider of the map data. Web map providers might define specific requirements for the attribution.

   ```
   copyright = char(uint8(169));
   attribution = [ ...
         "map data:  " + copyright + "OpenStreetMap contributors,SRTM", ...
         "map style: " + copyright + "OpenTopoMap (CC-BY-SA)"];
   ```

4. Define the name that will appear in the Layer Manager to identify the custom base layer.

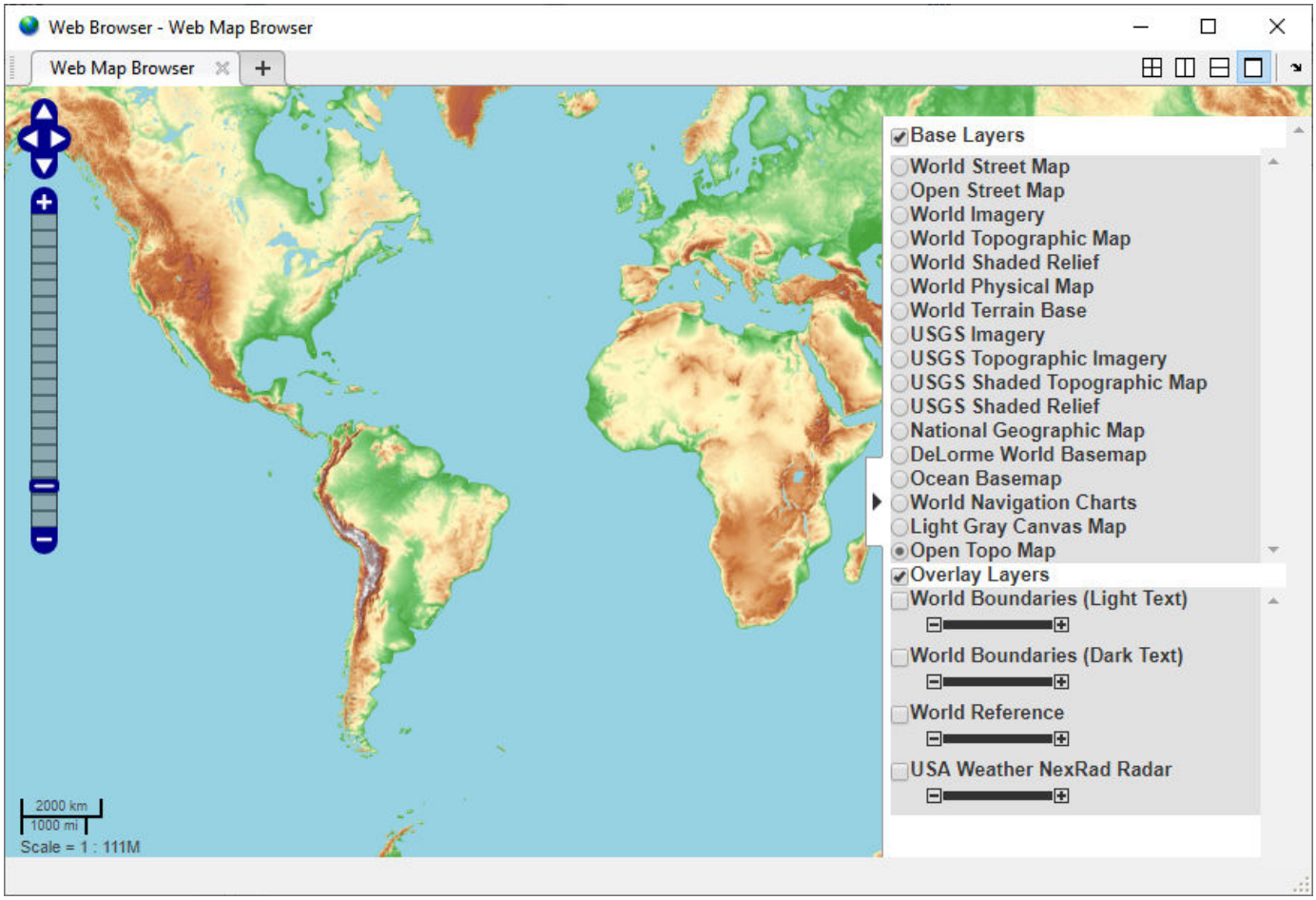
   ```
   displayName = 'Open Topo Map';
   ```

5. Add the custom base layer to the list of base layers available through the Layer Manager. When you add a custom base layer, the addition is persistent between MATLAB sessions.

   ```
   addCustomBasemap(name,url,'Attribution',attribution, ...
       'DisplayName',displayName)
   ```

6. Open a web map. Expand the Layer Manager and find the listing for the custom base layer in the list of base layers. To view the custom base layer, select the map in the Layer Manager. You can also specify the name you assigned to the map as an argument to the `webmap` function.

   ```
   webmap opentopomap
   ```



See Also

`addCustomBasemap` | `removeCustomBasemap` | `webmap`

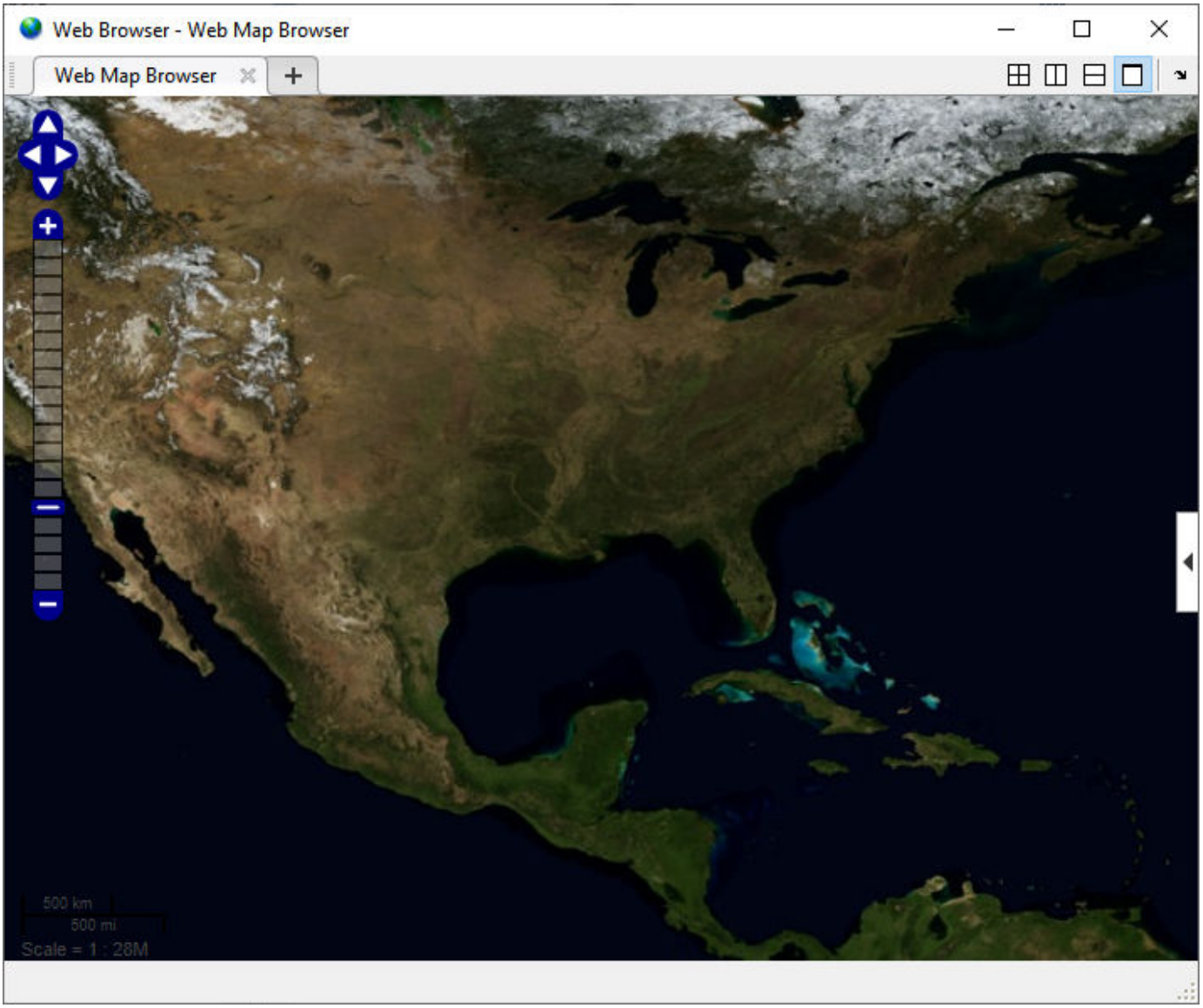
More About

- “Basic Workflow for Displaying Web Maps” on page 9-66
- “Specify a WMS Layer as a Base Layer” on page 9-72

Specify a WMS Layer as a Base Layer

Display a WMS layer in the web map browser by using the `webmap` function. The following example shows how to use Web Map Service functions to connect with a Web server, retrieve a layer, and display it in a web map browser. Change the view using your mouse.

```
info = wmsinfo('https://neo.sci.gsfc.nasa.gov/wms/wms?');
nasa = info.Layer;
baselayer = refine(nasa, 'bluemarbleng',  ...
      'SearchField', 'layername', 'MatchType', 'exact');
baselayer = wmsupdate(baselayer);
webmap(baselayer)
```



See Also

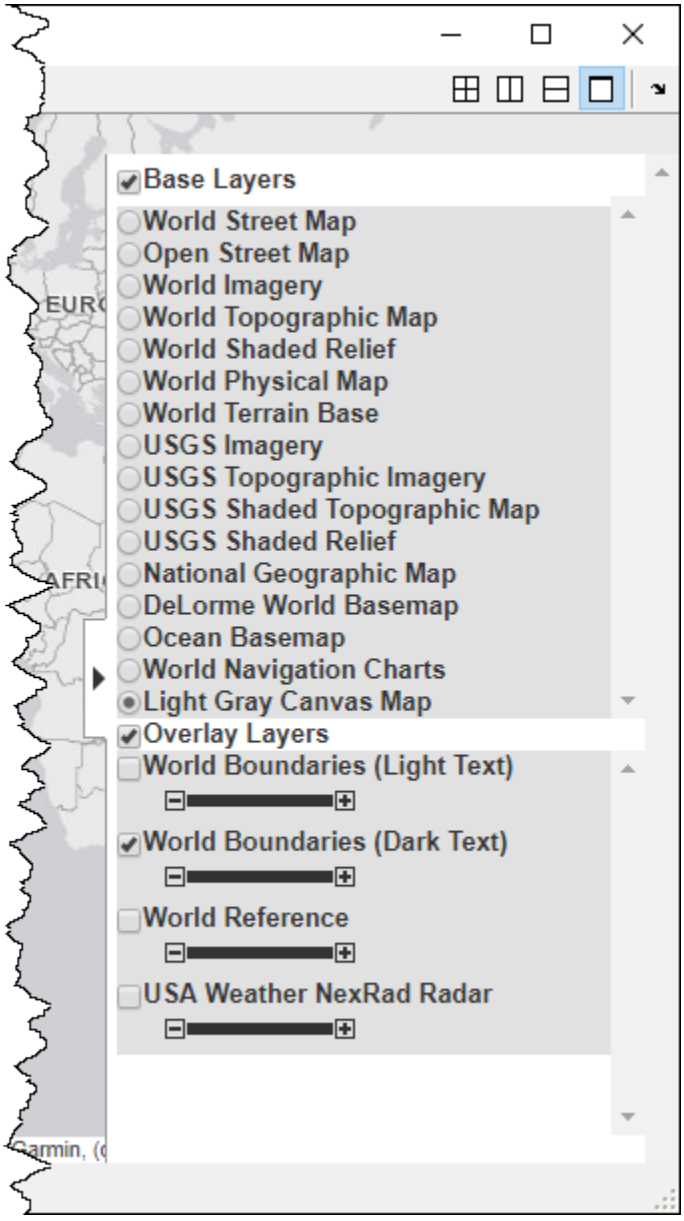
`webmap` | `wmsfind` | `wmsupdate`

More About

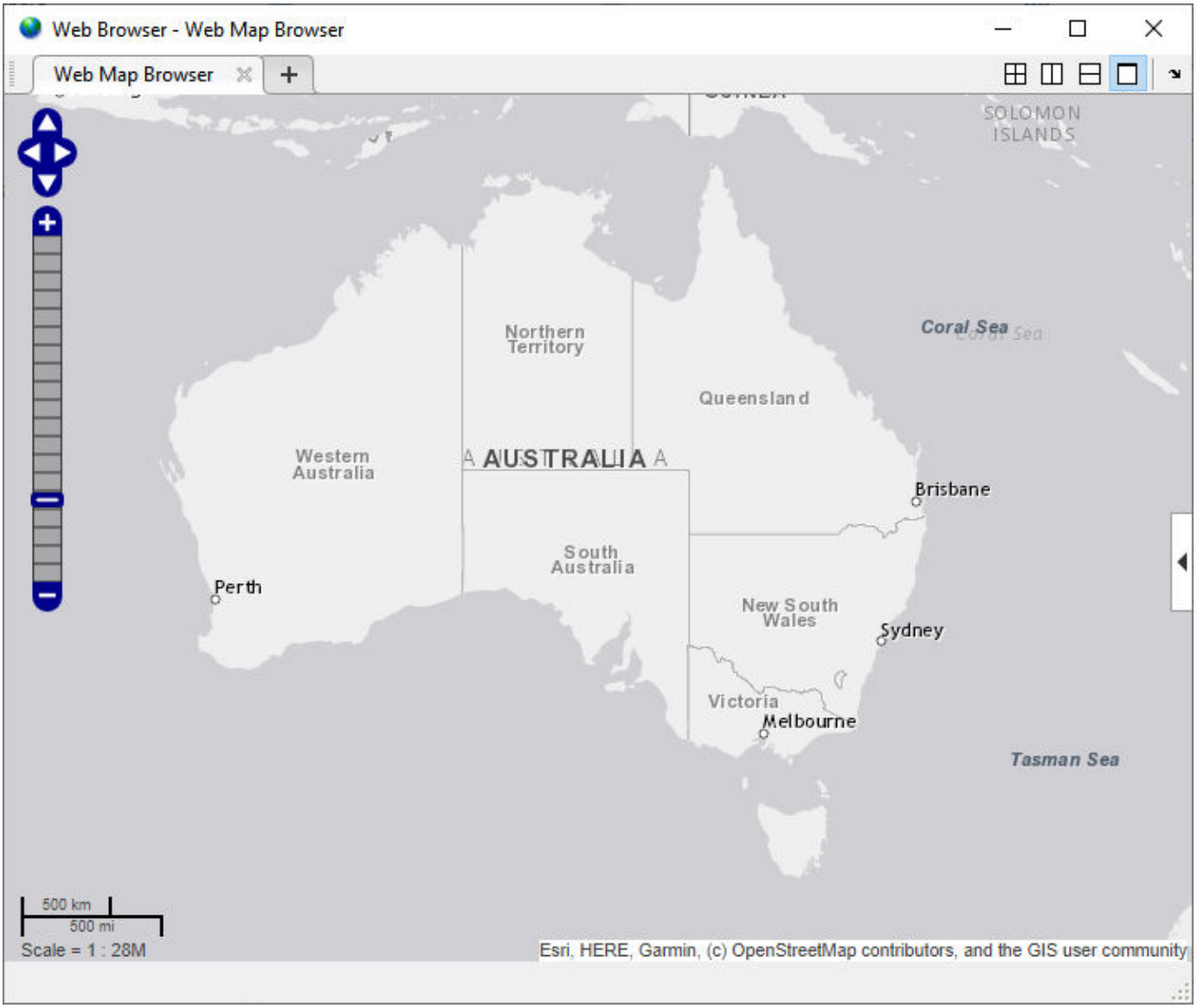
- “Basic Workflow for Displaying Web Maps” on page 9-66
- “Specify a Custom Base Layer” on page 9-70

Add an Overlay Layer to the Map

Add a layer of vector data over the base layer map by using the Layer Manager. For example open a web map by calling the `webmap` function and then open the Layer Manager by clicking the expander arrow ◀. Then, choose the **Light Gray Canvas Map** base layer and overlay the **World Boundaries (Dark Text)** vector data.



Close the layer manager and then navigate the map using your mouse or arrow keys.



See Also

`webmap` | `wmsfind` | `wmsupdate`

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

Add Line, Polygon, and Marker Overlay Layers to Web Maps

This example shows how to add line, polygon, and marker overlay layers to a web map. Overlay layers can add information, such as, state borders and coast lines, to a base layer map. The toolbox includes functions to draw lines, polygons, and web markers on a web map.

For example, to draw a multifaceted line or multiple lines on a map, use the `wmline` function. You use latitude and longitude values to specify the points that define the line. Similarly, to draw a polygon, use the `wmpolygon` function, specifying latitude and longitude values that define the vertices of the polygon. You can also add markers, or map pins, to identify points of interest on a web map using the `wmmarker` function.

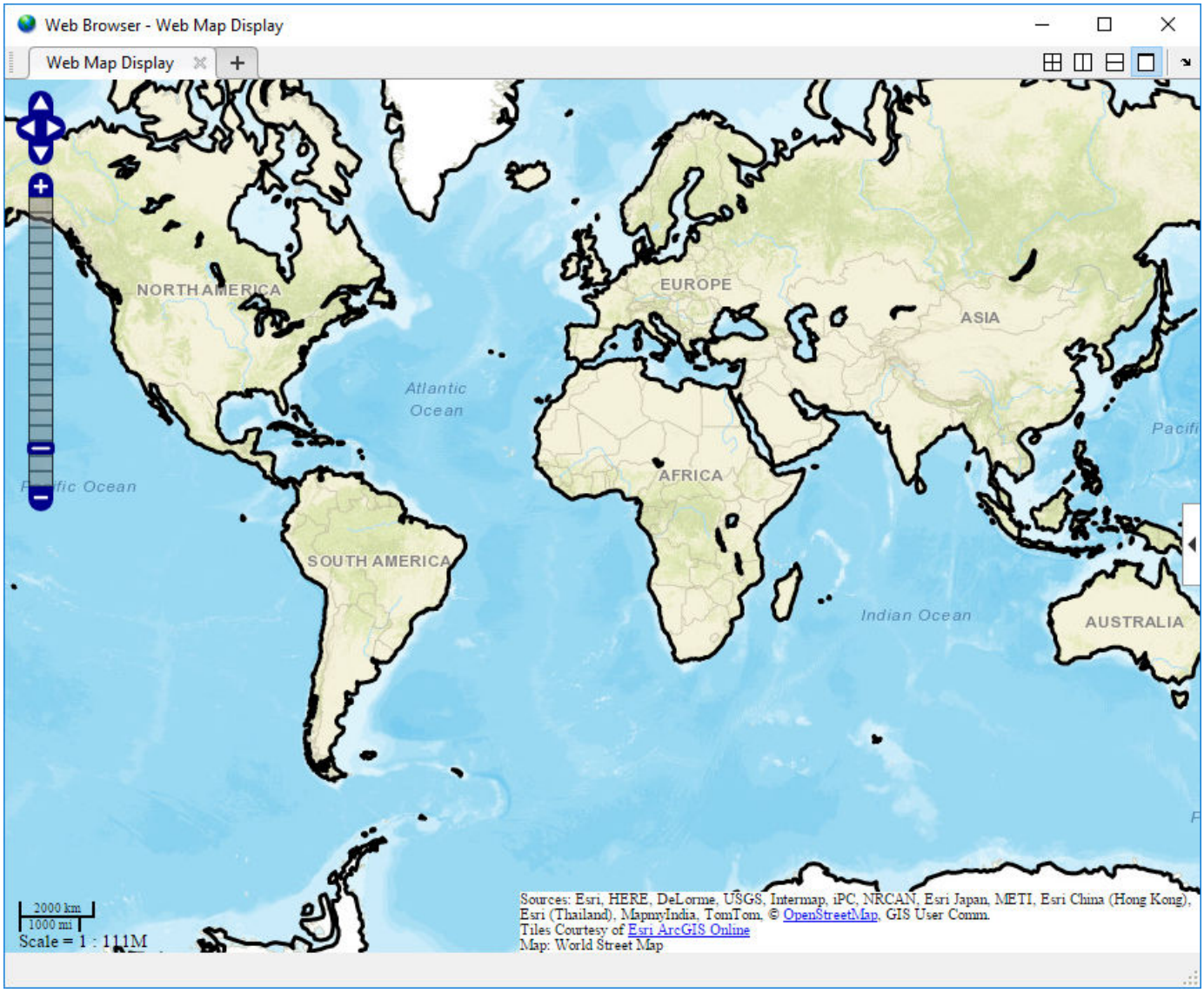
The following example illustrates these capabilities.

1. Load latitude and longitude data. This creates two variables in the workspace: `coastlat` and `coastlon`.

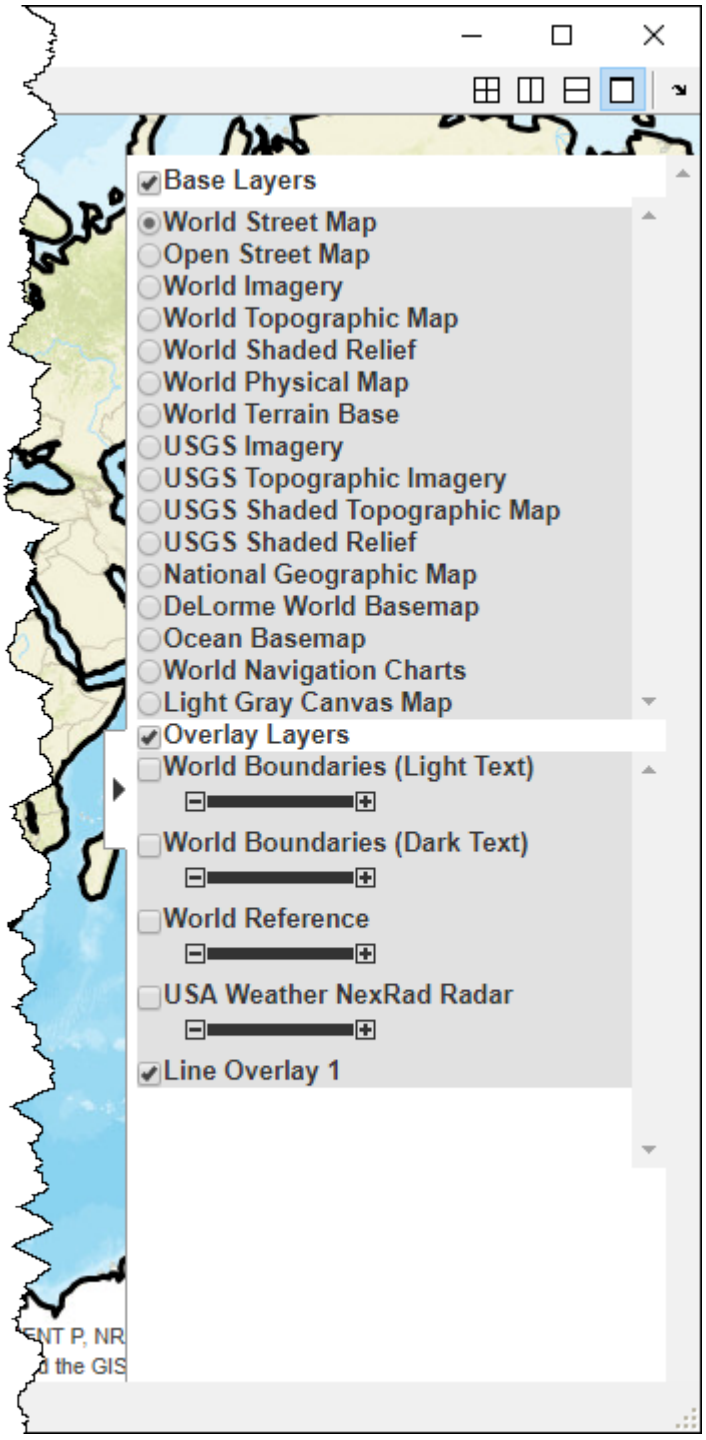
   ```
   load coastlines
   ```

2. Use the latitude and longitude data to define a line overlay. `wmline` draws the overlay on the current web map or, if none exists, it creates a new web map. The example includes several optional parameters to specify the line width and the name you want associated with the line.

   ```
   wmline(coastlat,coastlon,'LineWidth',3,'FeatureName','coastline')
   ```

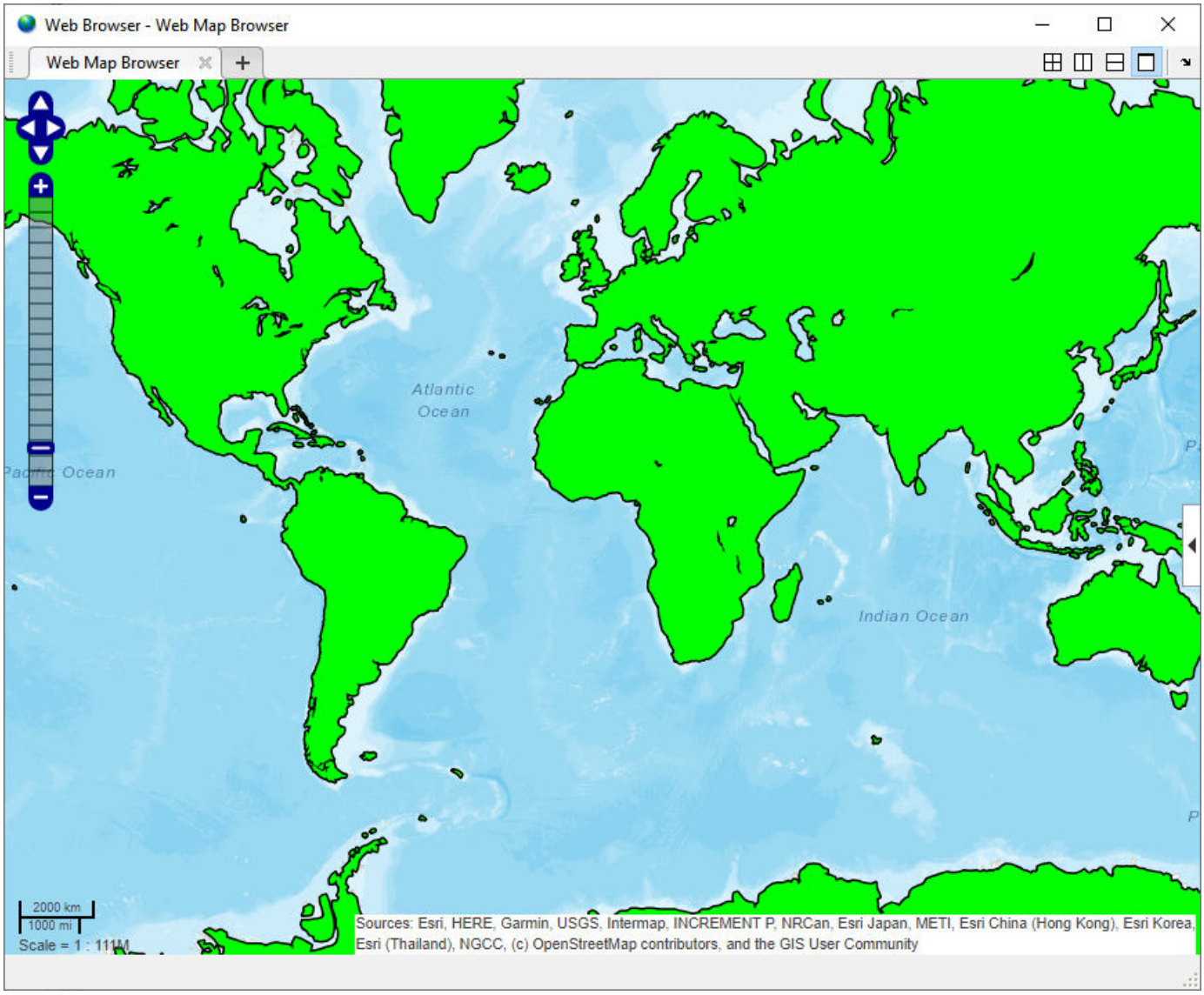


3 The `wmline` command adds the new overlay to the list of overlays in the Layer Manager. By default, this layer is called **Line Overlay 1**.

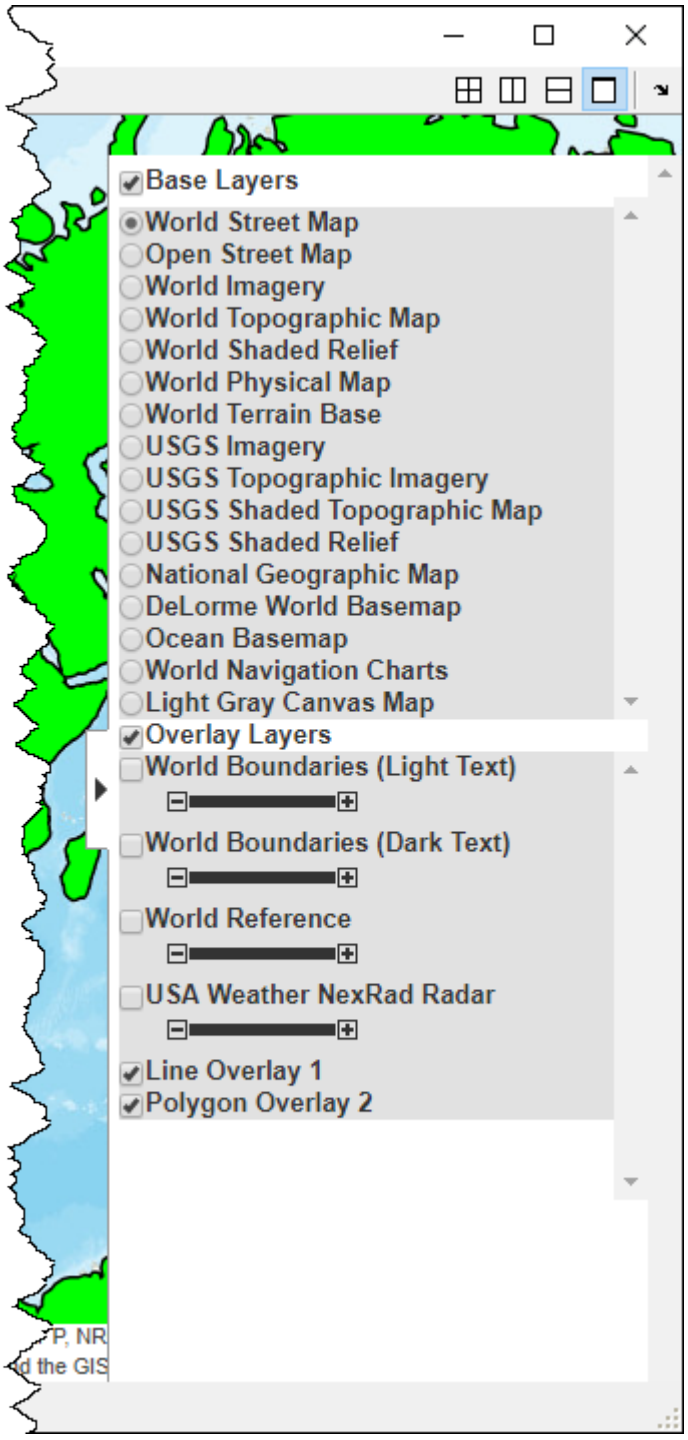


4 Use the same latitude and longitude data to define a polygon overlay. `wmpolygon` interprets the latitudes and longitudes as the vertices of a polygon, and draws the overlay on the current web map. The example includes several optional parameters to specify the line width and the name you want associated with the line.

```
wmpolygon(coastlat,coastlon,'FeatureName','coastline','FaceColor','green')
```

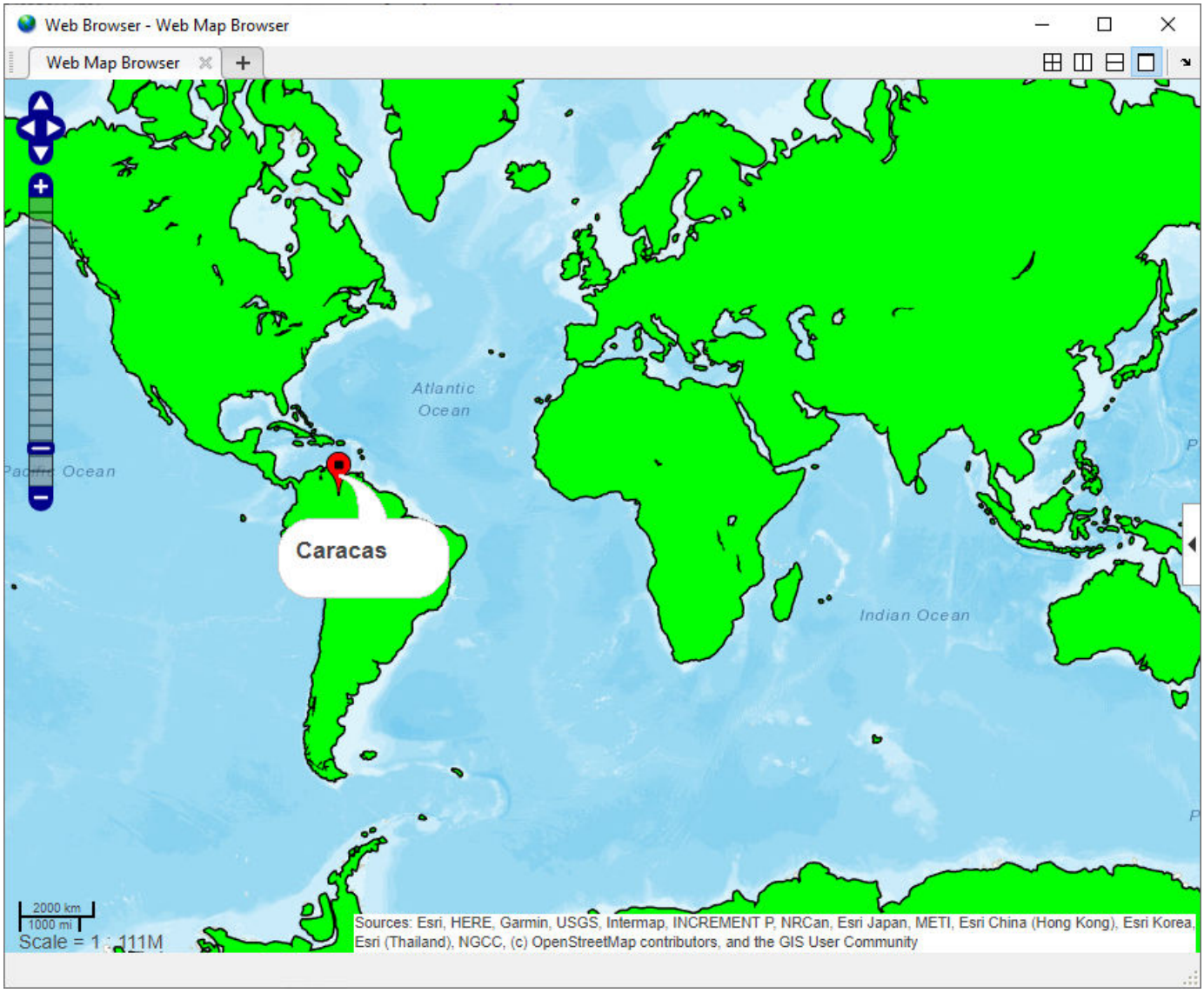


5 The `wmpolygon` command adds the polygon overlay to the list of overlays in the Layer Manager. By default, this layer is called **Polygon Overlay 2**.



6 Add a marker to the web map by using the `wmmarker` function. Display information about the marker by clicking on it. The `wmmarker` function adds the marker overlay to the list of overlays in the Layer Manager. By default, this layer is called **Marker Overlay 3**.

```
wmmarker(10.5000,-66.8992,'FeatureName','Caracas')
```



See Also

`webmap` | `wmline` | `wmmarker` | `wmpolygon` | `wmremove`

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

Remove Overlay Layers on a Web Map

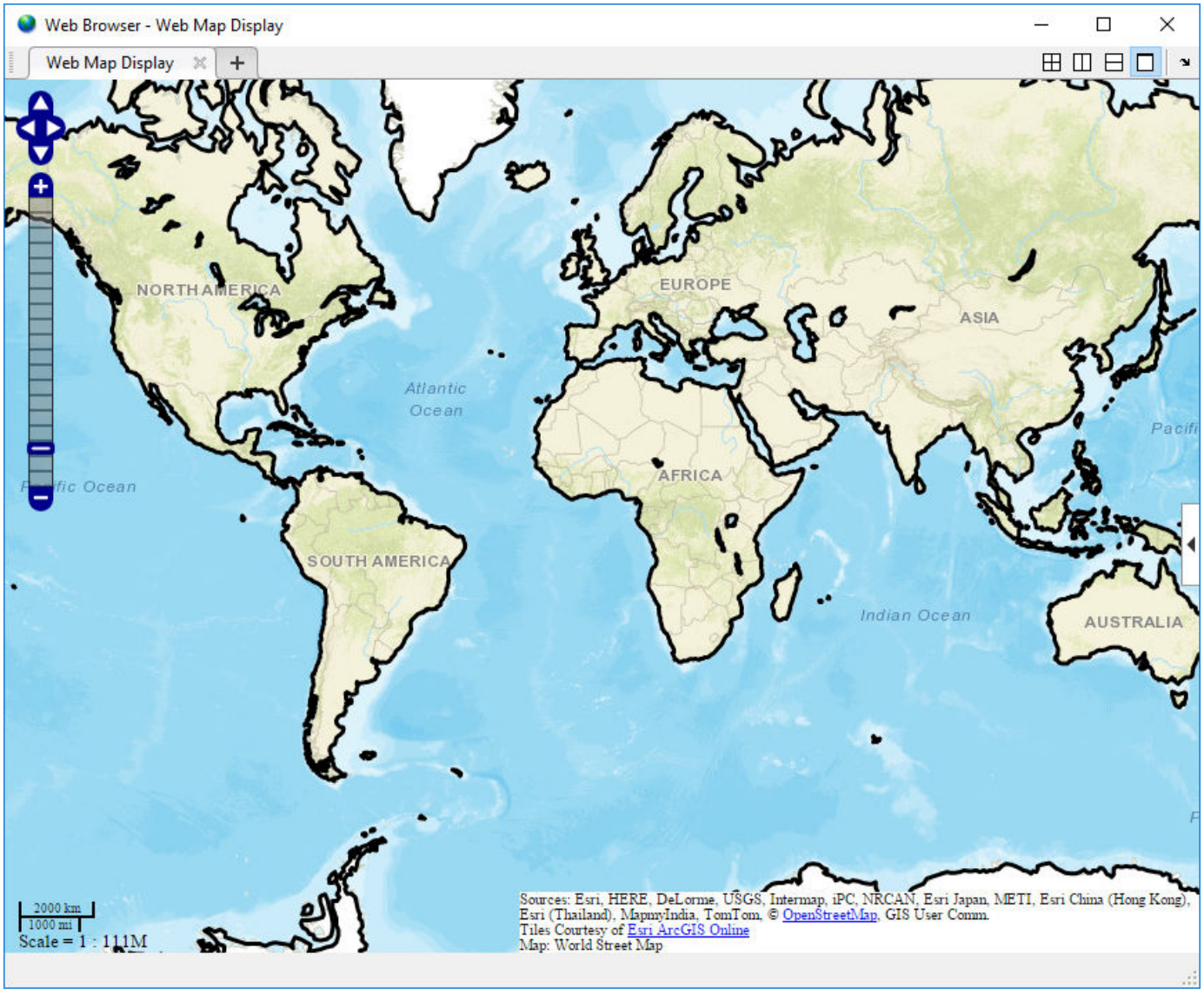
To remove an overlay layer on a web map, use the `wmremove` function. When called without an argument, `wmremove` deletes the most recently added overlay layer. You can also remove a particular overlay by specifying the handle of the line or marker overlay. The following example illustrates this capability.

1. Load latitude and longitude data. This command loads two variables into the workspace: `coastlat` and `coastlon`.

   ```
   load coastlines
   ```

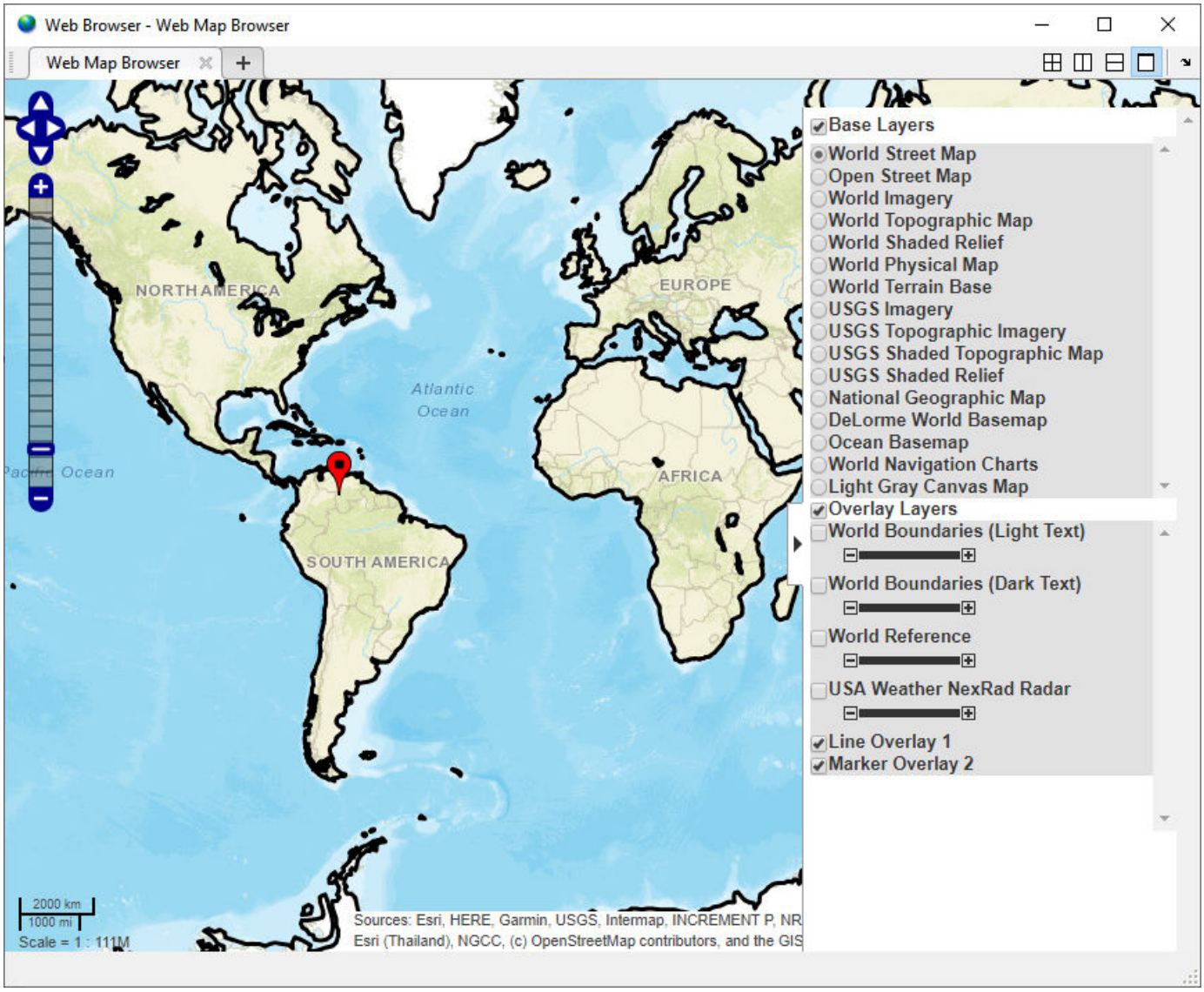
2. Add a line overlay of the coastline data and set the overlay to a variable using the `wmline` function. There is no current web map, so `wmline` creates one.

   ```
   h = wmline(coastlat,coastlon,'Width',3,'FeatureName','coastline');
   ```



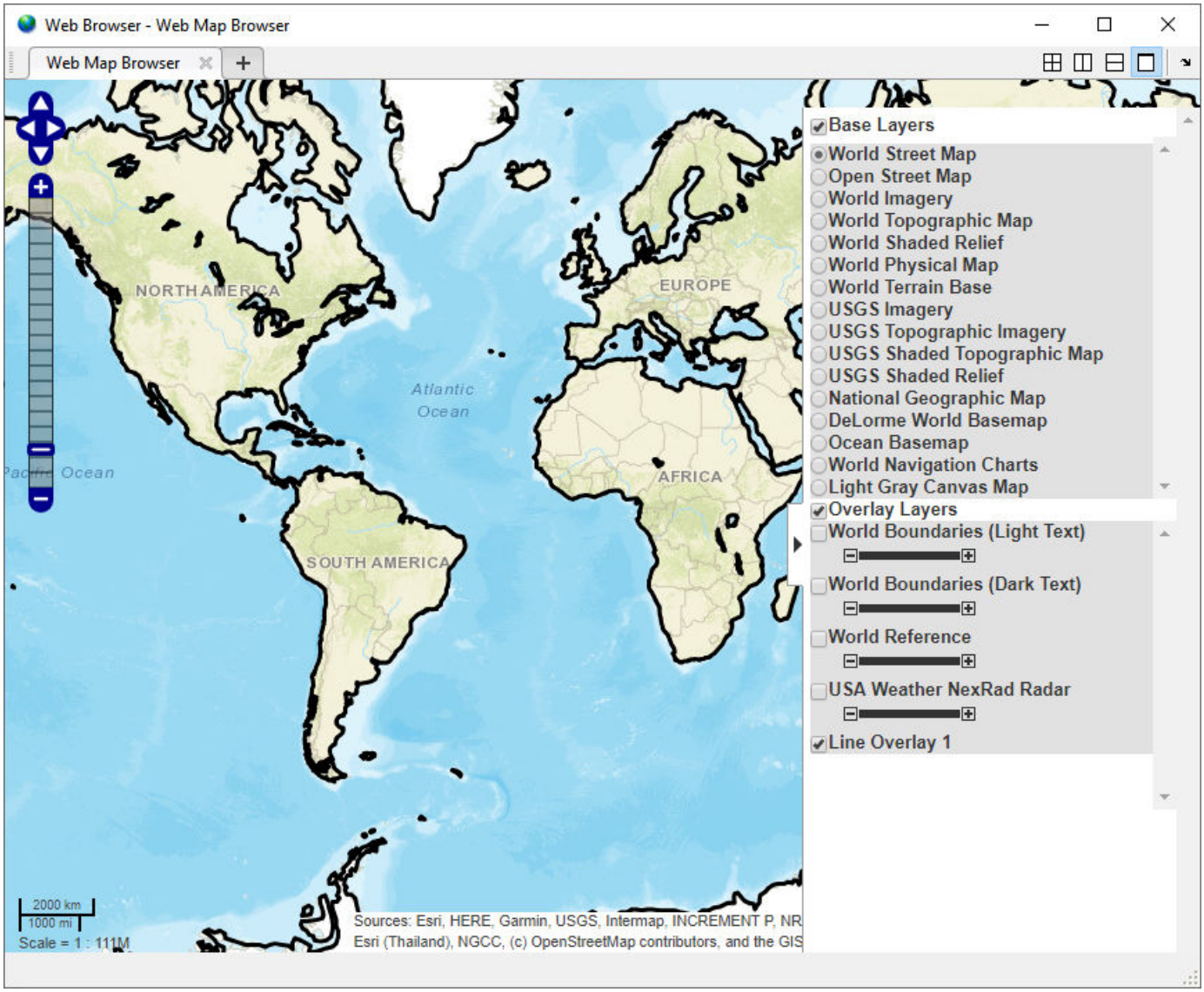
3 Add a marker overlay and set it to a variable using the `wmmarker` function. The marker highlights the location of the city of Caracas. Note that the overlays are listed in the Layer Manager as **Line Overlay 1** and **Marker Overlay 2**.

```
h2 = wmmarker(10.5000,-66.8992,'FeatureName','Caracas');
```



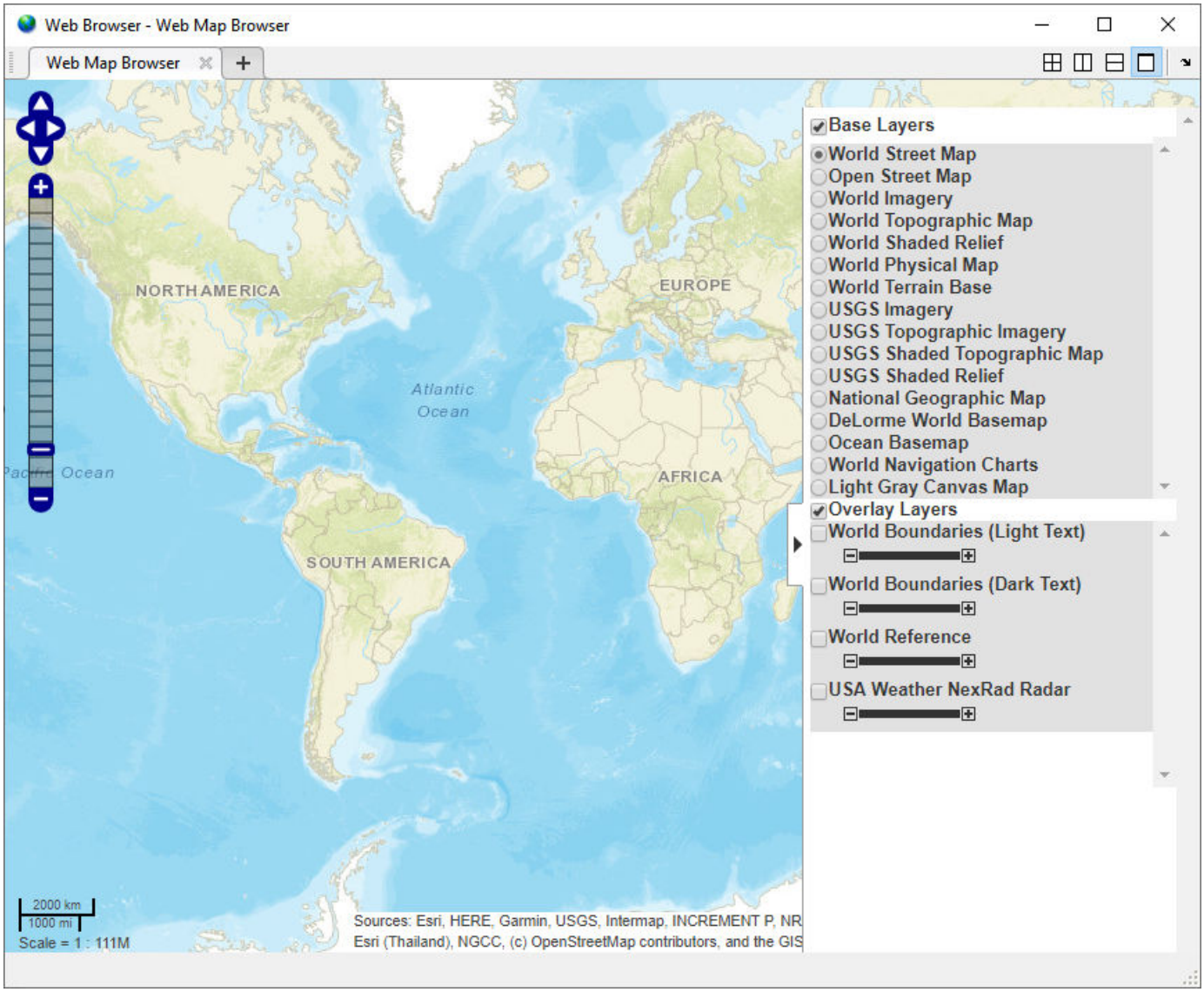
4 Remove one of the overlays by using the `wmremove` function. When called without an argument, `wmremove` deletes the most recent overlay. In this case, `wmremove` removes the marker overlay. The `wmremove` function also removes the marker entry in the Layer Manager.

```
wmremove
```



5 Remove a particular overlay by specifying it when you call `wmremove`. For example, remove the line overlay.

```
wmremove(h)
```



See Also

`webmap` | `wmline` | `wmmarker` | `wmpolygon` | `wmremove`

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

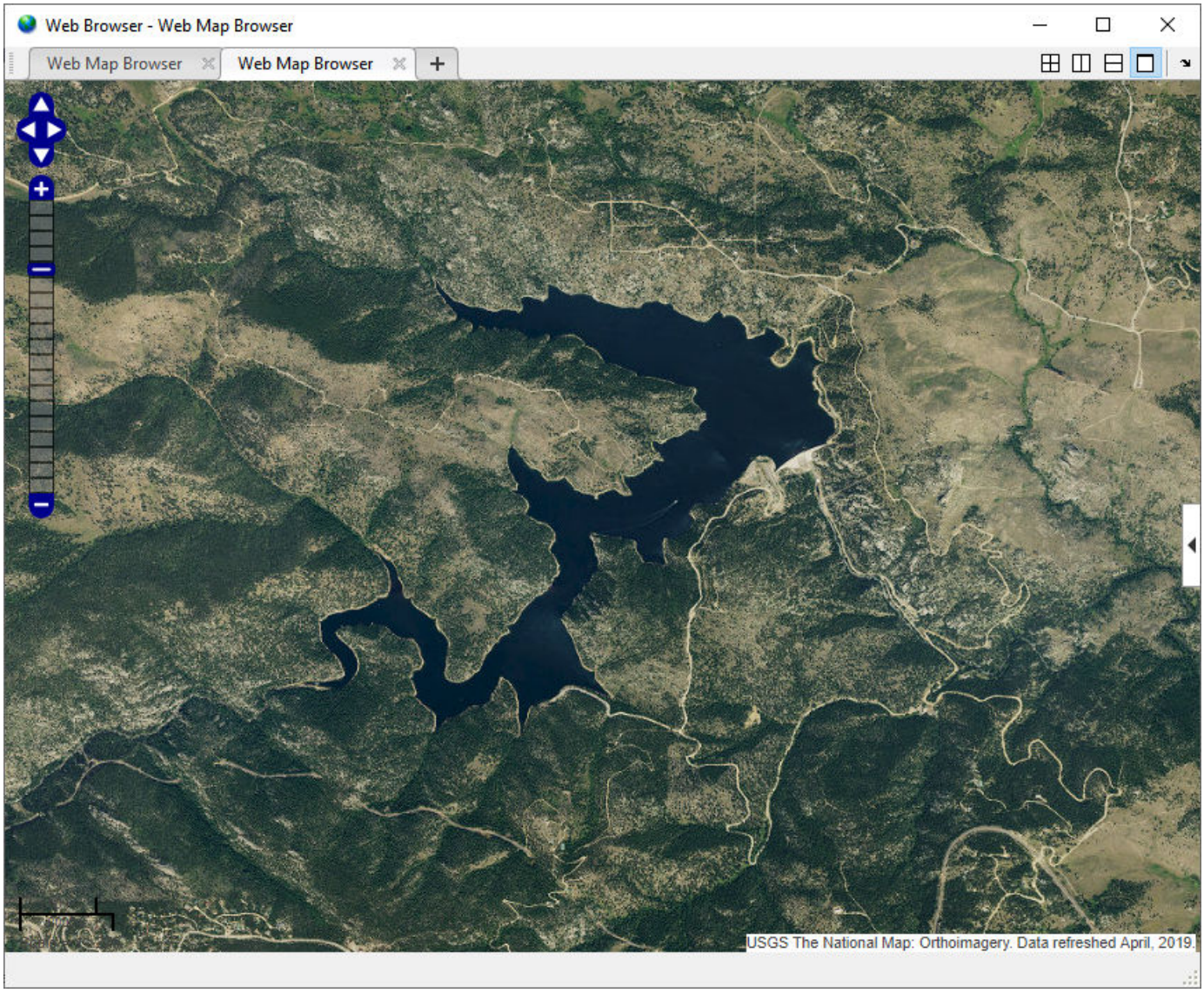
View Multiple Web Maps in a Browser

Using the web map browser tiling options you can get multiple views of the same location in different base layer maps.

The following example shows you how to open two web maps and display them side-by-side.

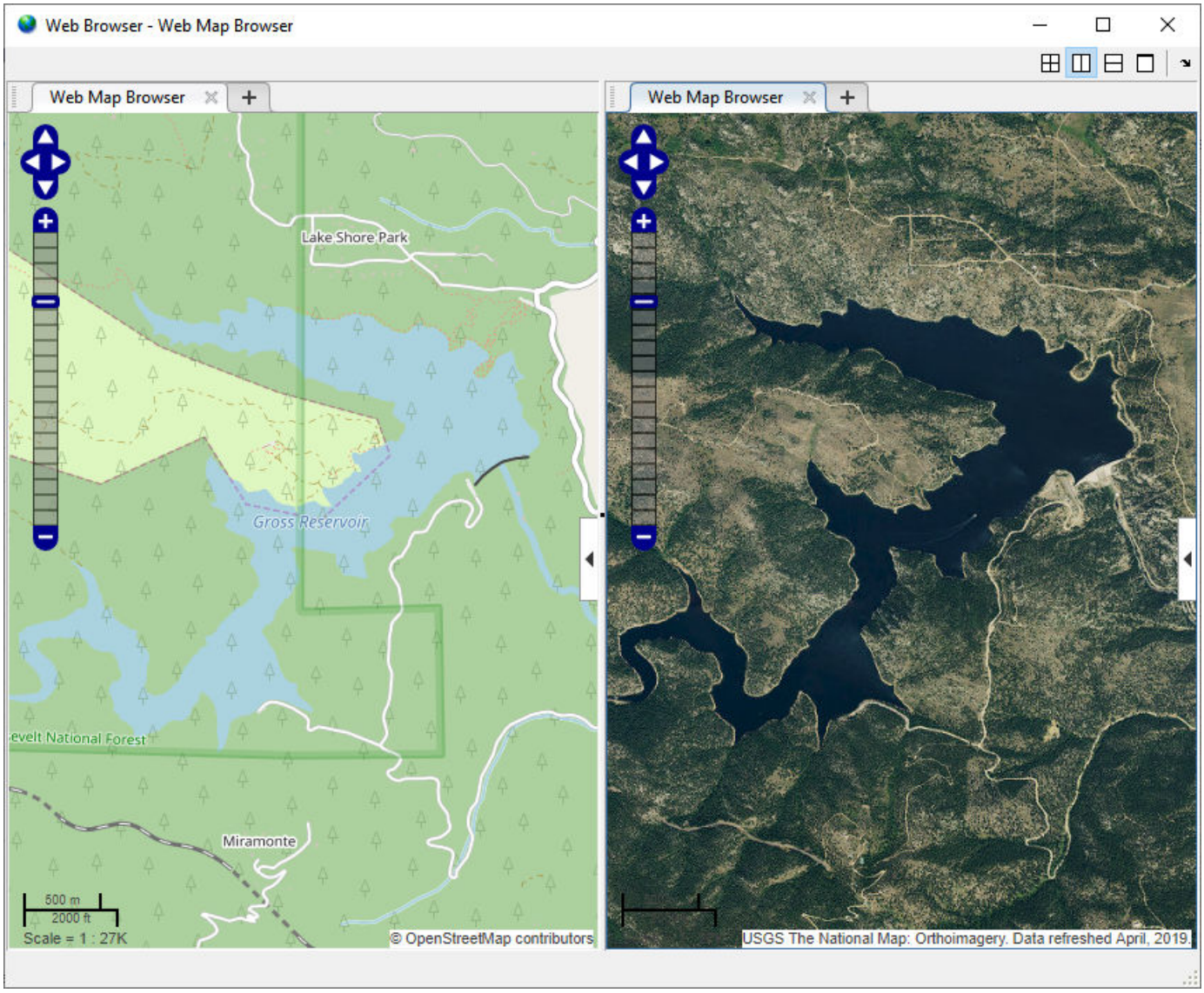
1 Open two web maps with different basemaps and set them to variables. Each call to `webmap` creates a new web map tab in the browser. Then, center the map at Gross Reservoir using the `wmcenter` function.

```
lat = 39.94509;
lon = -105.37008;
zoom = 14;
wm1 = webmap('OpenStreetMap');
wm2 = webmap('USGSImagery');
wmcenter(wm1,lat,lon,zoom)
wmcenter(wm2,lat,lon,zoom)
```



Note In MATLAB Online, multiple web maps appear as separate browser windows instead of tabs in a single browser window.

2 Display the maps side-by-side by selecting the tile vertically button ◫ from the web map toolbar.



See Also

`webmap` | `wmline` | `wmmarker` | `wmpolygon` | `wmremove`

More About

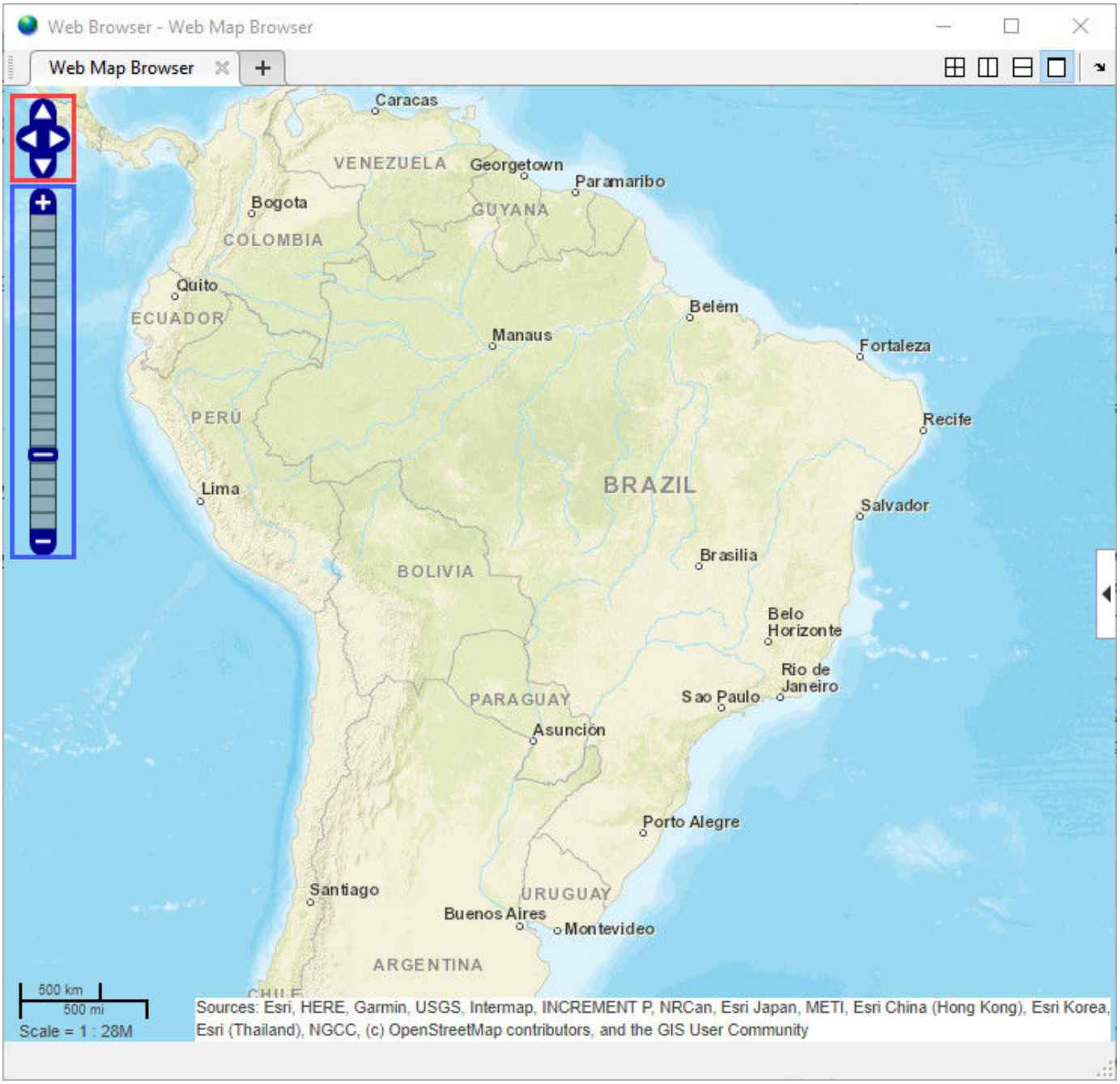
- “Basic Workflow for Displaying Web Maps” on page 9-66

Navigate a Web Map

Web maps displayed in a browser using the `webmap` function are interactive. View a portion of the map in more detail by using the zoom control or scroll wheel. Move the map in any direction by using the pan tool or arrow keys, or clicking and dragging the mouse. Alternatively, you can position the map programmatically using the `wmzoom`, `wmlimits`, and `wmcenter` functions.

For example, open a web map using the `webmap` function. By default, `webmap` displays the entire map, scaled to fit the browser and centered at latitude and longitude `[0 0]`. This image shows the pan tool outlined in red and the zoom tool outlined in blue. Use the tools to center the map on Brazil.

```
webmap
```



You can perform the same navigation programmatically. For example, open a web map that is centered on Brazil using the `wmcenter` function. Specify the latitude and longitude of the center point and the zoom level as arguments.

```
wmcenter(-15.6000,-56.1003,4)
```

You can also customize your view of a web map by specifying the latitude and longitude limits. For example, retrieve the current latitude and longitude limits using the `wmlimits` function. Depending on the size of the web map browser, your limits may be different.

```
[latlim,lonlim] = wmlimits

latlim =

  -36.5081    7.6476


lonlim =

  -88.7077  -23.4929
```

You can then open a new web map, specifying these latitude and longitude limits using the `wmlimits` function.

```
wmlimits(latlim,lonlim)
```

The displayed limits may not match the specified limits because the zoom level is quantized to discrete integer values and the longitude limits may be constrained.

See Also

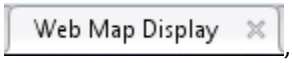
`webmap` | `wmcenter` | `wmsfind` | `wmsupdate`

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

Close a Web Map

To close a web map, click the Close button in the web map tab in the browser window



, or use the `wmclose` function.

When called without an argument, `wmclose` closes the current web map. You can also specify which web map to close by specifying a handle to the web map. To close all currently open web maps, call `wmclose` specifying the `'all'` argument.

The following example opens several web maps, closes a specific web map, and then closes all open web maps.

1 Open several web maps, getting the handles to the web maps.

```
wm1 = webmap;
wm2 = webmap('Light Gray');
wm3 = webmap('Open Street');
```

2 Close a specific web map, using its handle.

```
wmclose(wm3)
```

3 Close all web maps that remain open. You can also use the command form: `wmclose all`.

```
wmclose('all')
```

See Also

`webmap` | `wmclose`

More About

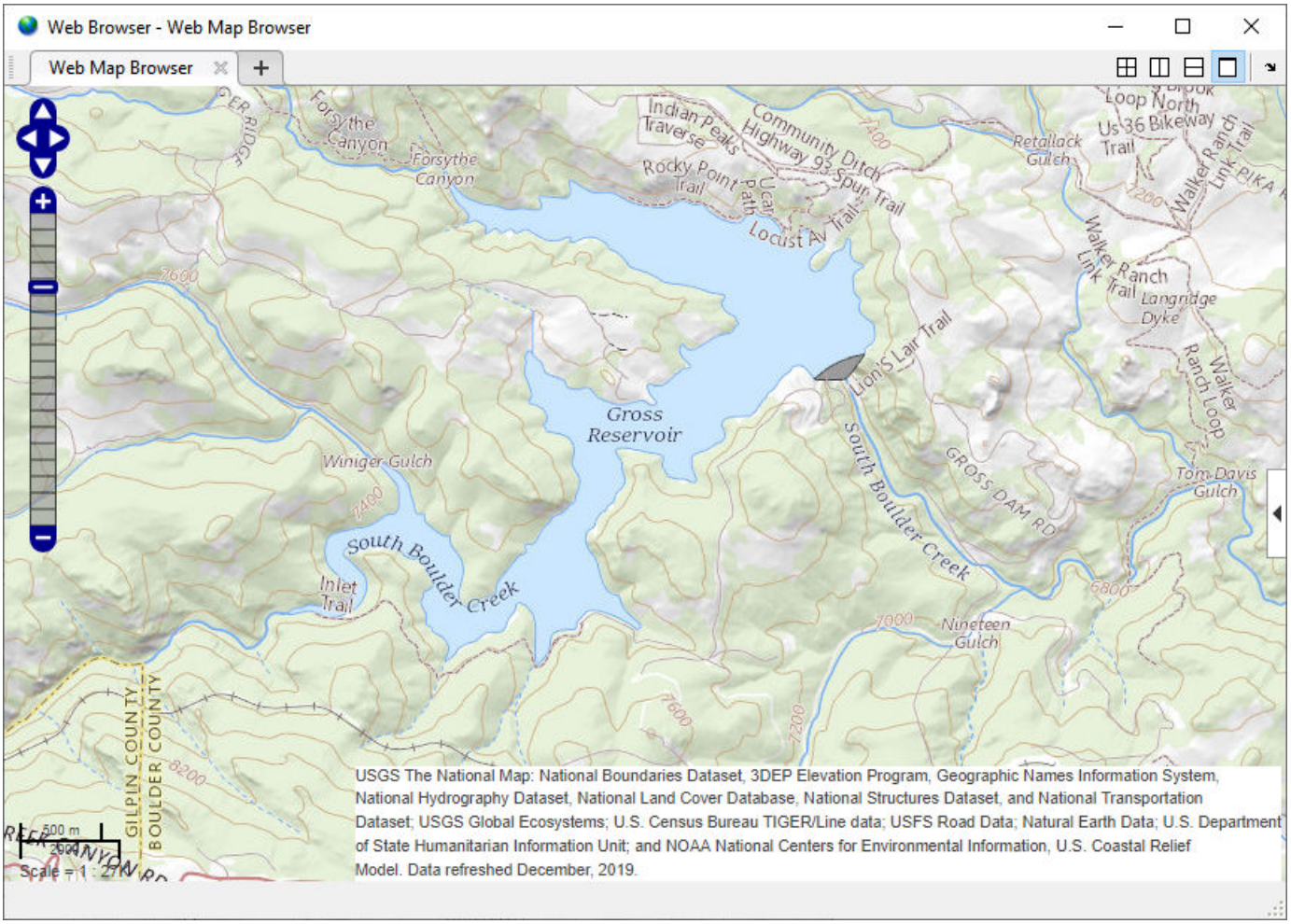
- “Basic Workflow for Displaying Web Maps” on page 9-66

Annotate a Web Map with Measurement Information

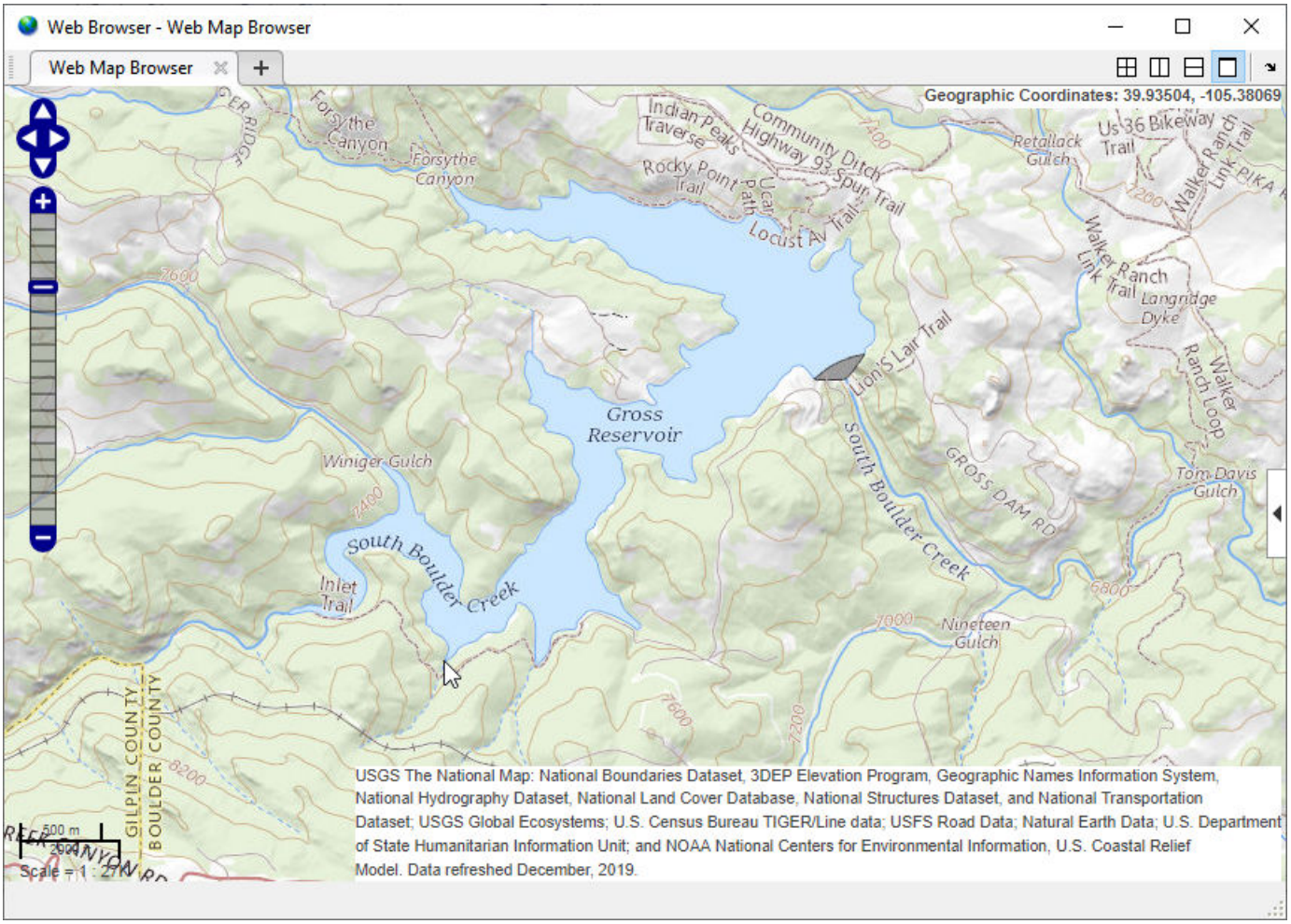
This example shows how to use a map to get information about a geographic feature. To illustrate, this example measures the length of the Gross Reservoir and adds some markers and a line overlay that act as annotations on the map.

Open a web map centered on the Gross Reservoir west of Boulder, Colorado. Use the USGS Shaded Topographic Map to get the level of topographical detail required for this measurement.

```
webmap('usgsshadedtopographicmap')
lat = 39.9428;
lon = -105.3691;
zoom = 14;
wmcenter(lat,lon,zoom)
```



Identify two points at opposite ends of the lake and get the latitude and longitude of these points. To get this information in a web map, move the mouse pointer over a location on the map. In the upper right corner, the Web Browser displays the geographic coordinates of the point.



Store the latitude and longitude information in a geoshape vector.

```
lat1 = 39.93504;
lon1 = -105.38069;
lat2 = 39.95226;
lon2 = -105.35892;
s = geoshape([lat1 lat2],[lon1 lon2])
```

```
s =

 1×1 geoshape vector with properties:

 Collection properties:
     Geometry: 'line'
     Metadata: [1×1 struct]
 Vertex properties:
     Latitude: [39.9350 39.9523]
    Longitude: [-105.3807 -105.3589]
```

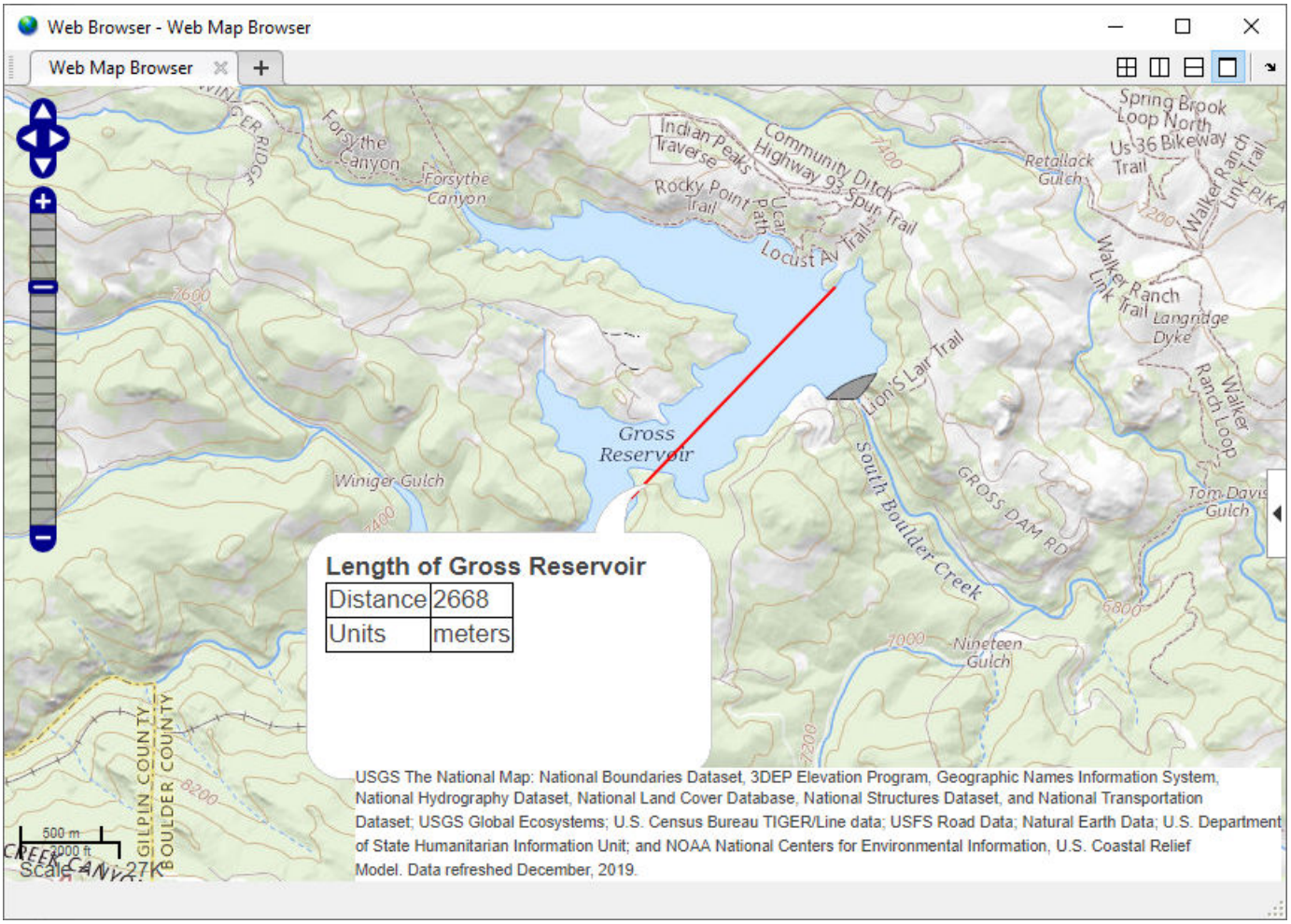
Calculate the distance between the two points to get the length of the reservoir. Use the `distance` function which calculates the distance between points on a sphere or ellipsoid.

```
d = distance(s.Latitude(1),s.Longitude(1),s.Latitude(2), ...
    s.Longitude(2),wgs84Ellipsoid)
```

```
d =

   2.6678e+03
```

Display a line between the two points. Include information about the length of the lake in the line's information balloon. Store the distance and information about units as two dynamic fields added to the geoshape vector.

```
s.Distance = round(d);
s.Units = 'meters';

wmline(s,'Color','red','FeatureName','Length of Gross Reservoir', ...
    'Overlayname','Transect');
```



See Also

geoshape | webmap | wmline | wmmarker | wmpolygon | wmremove | wmsfind | wmsupdate

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

Compositing and Animating Web Map Service (WMS) Meteorological Layers

This example shows how to composite and animate data from multiple Web Map Service (WMS) layers.

The base layer is from the NASA Goddard Space Flight Center's Scientific Visualization Studio (SVS) Image Server. The data in this layer shows satellite cloud data during Hurricane Katrina from August 23 through August 30, 2005. The layer consists of cloud data extracted from GOES-12 imagery and overlaid on a color image of the southeast United States.

Next-Generation Radar (NEXRAD) images, collected by the Iowa State University's Iowa Environmental Mesonet (IEM) Web map server, are composited with the cloud data at regular intervals of time.

In particular, this example will show you how to:

- Use the WMS database to find the Katrina and NEXRAD layers
- Retrieve the Katrina base map from a WMS server at a particular time-step
- Retrieve the NEXRAD map from a WMS server at the same time-step
- Composite the base map with the map containing the NEXRAD imagery
- View the composited map in a projected coordinate system
- Retrieve, composite, and animate multiple time sequences
- Create a video file and animated GIF file of the animation

Understanding Basic WMS Terminology

If you are new to WMS, several key concepts are important to understand and are listed here.

- *Web Map Service* --- The Open Geospatial Consortium (OGC) defines a Web Map Service (WMS) to be an entity that "produces maps of spatially referenced data dynamically from geographic information."
- *WMS server* --- A server that follows the guidelines of the OGC to render maps and return them to clients
- *map* --- The OGC definition for map is "a portrayal of geographic information as a digital image file suitable for display on a computer screen."
- *layer* --- A dataset of a specific type of geographic information, such as temperature, elevation, weather, orthophotos, boundaries, demographics, topography, transportation, environmental measurements, and various data from satellites
- *capabilities document* --- An XML document containing metadata describing the geographic content offered by a server

Source Function

The code shown in this example can be found in this function:

```
function mapexwmsanimate(useInternet,datadir)
```

Internet Access

Since WMS servers are located on the Internet, this example can be set to access the Internet to dynamically render and retrieve maps from WMS servers or it can be set to use data previously

retrieved from the Internet using the WMS capabilities but now stored in local files. You can use a variable, `useInternet`, to determine whether to read data from locally stored files, or retrieve the data from the Internet.

If the `useInternet` flag is set to true, then an Internet connection must be established to run the example. Note that the WMS servers may be unavailable, and several minutes may elapse before the maps are returned. One of the challenges of working with WMS servers is that sometimes you will encounter server errors. A function, such as `wmsread`, may time out if a server is unavailable. Often, this is a temporary problem and you will be able to connect to the server if you try again later. For a list of common problems and strategies for working around them, please see the Common Problems with WMS Servers section in the Mapping Toolbox™ User's Guide.

You can store the data locally the first time you run the example and then set the `useInternet` flag to false. If the `useInternet` flag is not defined, it is set to false.

```
if ~exist('useInternet', 'var')
    useInternet = false;
end
```

Setup: De ne a Data Directory and Filename Utility Function

This example writes data to files if `useInternet` is `true` or reads data from files if `useInternet` is `false`. It uses the variable `datadir` to denote the location of the folder containing the data files.

```
if ~exist('datadir','var')
    datadir = fullfile(matlabroot,'examples','map','data');
end
if ~exist(datadir,'dir')
    mkdir(datadir)
end
```

Define an anonymous function to prepend `datadir` to the input filename:

```
datafile = @(filename) fullfile(datadir,filename);
```

Step 1: Find Katrina Layers From Local Database

One of the more challenging aspects of using WMS is finding a WMS server and then finding the layer that is of interest to you. The process of finding a server that contains the data you need and constructing a specific and often complicated URL with all the relevant details can be very daunting.

The Mapping Toolbox™ simplifies the process of locating WMS servers and layers by providing a local, installed, and pre-qualified WMS database, that is searchable, using the function `wmsfind`. You can search the database for layers and servers that are of interest to you. Here is how you find layers containing the term `katrina` in either the `LayerName` or `LayerTitle` field of the database:

```
katrina = wmsfind('katrina');
whos katrina
```

```
  Name          Size            Bytes  Class       Attributes

  katrina      34x1             16754  WMSLayer
```

The search for the term `'katrina'` returned a `WMSLayer` array containing multiple layers. To inspect information about an individual layer, simply display it like this:

```
katrina(1)
```

```
ans =

  WMSLayer

  Properties:
          Index: 1
    ServerTitle: 'NASA SVS Image Server'
      ServerURL: 'https://svs.gsfc.nasa.gov/cgi-bin/wms?'
     LayerTitle: 'GOES-12 Imagery of Hurricane Katrina: Longwave Infrared Close-up (1024x1024 A
      LayerName: '3216_22510'
         Latlim: [15.0000 45.0000]
         Lonlim: [-100.0000 -70.0000]
```

If you type, `katrina`, in the command window, the entire contents of the array are displayed, with each element's index number included in the output. This display makes it easy for you to examine the entire array quickly, searching for a layer of interest. You can display only the `LayerTitle` property for each element by executing the command:

```
disp(katrina,'Properties','layertitle','Index','off','Label','off');
```

As you've discovered, a search for the generic word `'katrina'` returned results of many layers and you need to select only one layer. In general, a search may even return thousands of layers, which may be too large to review individually. Rather than searching the database again, you may refine your search by using the `refine` method of the `WMSLayer` class. Using the `refine` method is more efficient and returns results faster than `wmsfind` since the search has already been narrowed to a smaller set. Supplying the query string, `'goes-12*katrina*visible*close*up*animation'`, to the `refine` method returns a `WMSLayer` array whose elements contain a match of the query string in either the `LayerTitle` or `LayerName` properties. The `*` character indicates a wild-card search. If multiple entries are returned, select only the first one from the svs.gsfc.nasa.gov server.

```
katrina = refine(katrina,'goes-12*katrina*visible*close*up*animation');
katrina = refine(katrina,'svs.gsfc.nasa.gov','Searchfield','serverurl');
katrina = katrina(1);
whos katrina
```

```
  Name         Size            Bytes  Class       Attributes

  katrina      1x1               466  WMSLayer
```

Step 2: Synchronize WMSLayer Object with Server

The database only stores a subset of the layer information. For example, information from the layer's abstract, details about the layer's attributes and style information, and the coordinate reference system of the layer are not returned by `wmsfind`. To return all the information, you need to use the `wmsupdate` function. `wmsupdate` synchronizes the layer from the database with the server, filling in the missing properties of the layer.

Synchronize the first `katrina` layer with the server in order to obtain the abstract information. Since this action requires access to the Internet, call `wmsupdate` only if the `useInternet` flag is true.

```
cachefile = datafile('katrina.mat');
if useInternet
```

```
    katrina = wmsupdate(katrina);
    if ~exist(cachefile,'file')
        save(cachefile,'katrina')
    end
else
    cache = load(cachefile);
    katrina = cache.katrina;
end
```

Display the abstract information of the layer. Use `isspace` to help determine where to line wrap the text.

```
abstract = katrina.Abstract;
endOfLine = find(isstrprop(abstract,'cntrl'),1);
abstract = abstract(1:endOfLine);
numSpaces = 60;
while(~isempty(abstract))
    k = find(isspace(abstract));
    n = find(k > numSpaces,1);
    if ~isempty(n)
        fprintf('%s\n',abstract(1:k(n)))
        abstract(1:k(n)) = [];
    else
        fprintf('%s\n',abstract)
        abstract = '';
    end
end
```

```
The GOES-12 satellite sits at 75 degrees west longitude at an
altitude of 36,000 kilometers over the equator, in geosynchronous
orbit.  At this position its Imager instrument takes pictures
of cloud patterns in several wavelengths for all of North and
South America, a primary measurement used in weather forecasting.
 The Imager takes a pattern of pictures of parts of the Earth
in several wavelengths all day, measurements that are vital in
weather forecasting.  This animation shows a daily sequence of
GOES-12 images in the visible wavelengths, from 0.52 to 0.72
microns, during the period that Hurricane Katrina passed through
the Gulf of Mexico.  At one kilometer resolution, the visible
band measurement is the highest resolution data from the Imager,
which accounts for the very high level of detail in these images.
 For this animation, the cloud data was extracted from GOES image
and laid over a background color image of the southeast United
States.
```

Note that this abstract information, including any typographical issues and incomplete fragments, was obtained directly from the server.

Step 3: Explore Katrina Layer Details

You can find out more information about the `katrina` layer by exploring the `Details` property of the `katrina` layer. The `Details.Attributes` field informs you that the layer has fixed width and fixed height attributes, thus the size of the requested map cannot be modified.

```
katrina.Details.Attributes
```

```
ans =
```

```
  struct with fields:

      Queryable: 0
       Cascaded: 0
         Opaque: 1
      NoSubsets: 1
     FixedWidth: 1024
    FixedHeight: 1024
```

The `Details.Dimension` field informs you that the layer has a `time` dimension

```
katrina.Details.Dimension
```

```
ans =

  struct with fields:

              Name: 'time'
             Units: 'ISO8601'
        UnitSymbol: ''
           Default: '2005-08-30T17:45Z'
    MultipleValues: 0
      NearestValue: 0
           Current: 0
            Extent: '2005-08-23T17:45Z/2005-08-30T17:45Z/P1D'
```

with an extent from `2005-08-23T17:45Z` to `2005-08-30T17:45Z` with a period of `P1D` (one day), as shown in the `Details.Dimension.Extent` field.

```
katrina.Details.Dimension.Extent
```

```
ans =

    '2005-08-23T17:45Z/2005-08-30T17:45Z/P1D'
```

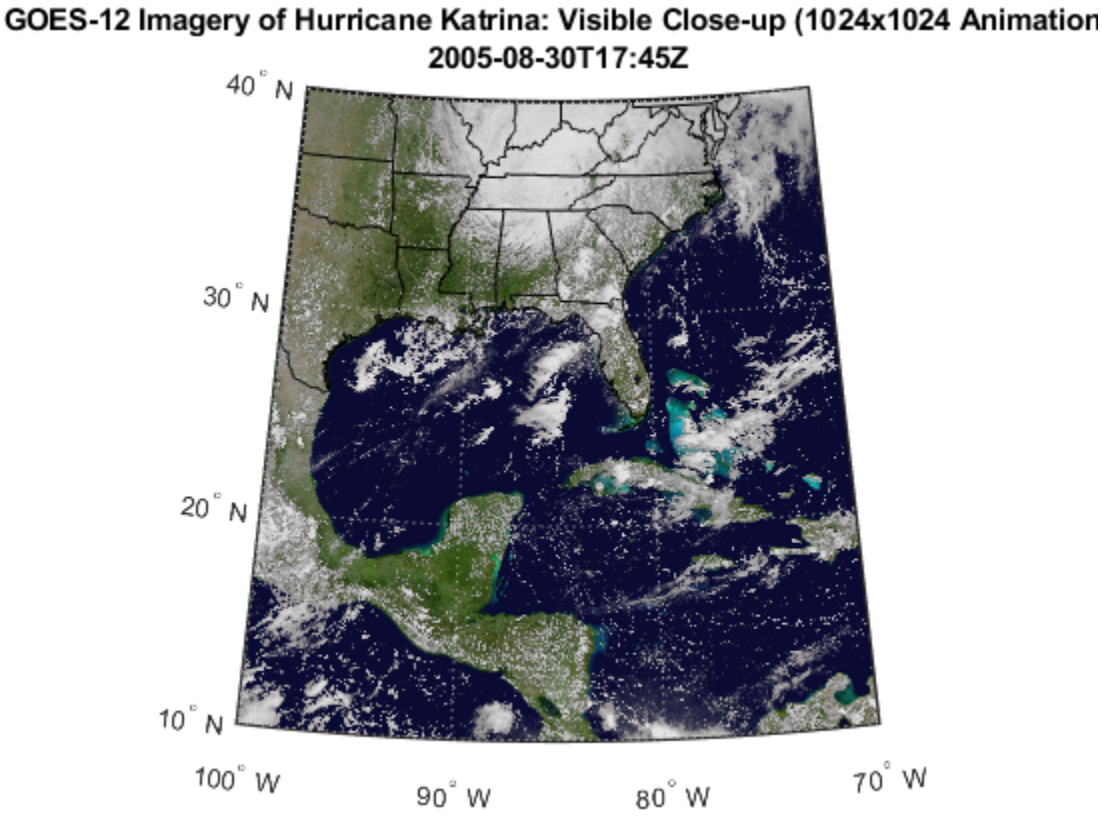
Step 4: Retrieve Katrina Map from Server

Now that you have found a layer of interest, you can retrieve the raster map using the function `wmsread` and display the map using the function `geoshow`. Since `Time` is not specified when reading the layer, the default time, `2005-08-30T17:45Z`, is retrieved as specified by the `Details.Dimension.Default` field. If the `useInternet` flag is set to true, then cache the image and referencing matrix in a GeoTIFF file.

```
cachefile = datafile('katrina.tif');
if useInternet
    [katrinaMap,R] = wmsread(katrina);
    if ~exist(cachefile,'file')
        geotiffwrite(cachefile,katrinaMap,R)
    end
else
    [katrinaMap,R] = readgeoraster(cachefile);
end
```

Display the `katrinaMap` and overlay the data from the `usastatehi.shp` file.

```
states = shaperead('usastatehi.shp','UseGeoCoords',true);
figure
usamap(katrina.Latlim, katrina.Lonlim)
geoshow(katrinaMap,R)
geoshow(states,'FaceColor','none')
title({katrina.LayerTitle, katrina.Details.Dimension.Default}, ...
    'Interpreter','none')
```



Step 5: Find NEXRAD Radar Layer

NEXRAD radar images for the United States are stored on the Iowa State University's IEM Web map server. The server conveniently stores NEXRAD images in five minute increments from `1995-01-01` to the present time. You can find the layer by first searching for the term `IEM WMS Service` in the `ServerTitle` field of the WMS database, then refining the search by requesting the layer of interest, `nexrad-n0r-wmst`.

```
iemLayers = wmsfind('IEM WMS Service','SearchField','servertitle');
nexrad = refine(iemLayers,'nexrad-n0r-wmst');
```

Synchronize the layer with the server.

```
cachefile = datafile('nexrad.mat');
if useInternet
    nexrad = wmsupdate(nexrad);
    if ~exist(cachefile,'file')
        save(cachefile,'nexrad')
```

```
    end
else
    cache = load(cachefile);
    nexrad = cache.nexrad;
end
```

Step 6: Obtain Extent Parameters

To composite the `nexrad` layer with the `katrina` layer, you need to obtain the `nexrad` layer at coincidental time periods, and concurrent geographic and image extents. The `Details.Dimension` field informs you that the layer has a time dimension,

```
nexrad.Details.Dimension
```

```
ans =

  struct with fields:

              Name: 'time'
             Units: 'ISO8601'
        UnitSymbol: ''
           Default: '2006-06-23T03:10:00Z'
    MultipleValues: 0
      NearestValue: 0
           Current: 0
            Extent: '1995-01-01/2011-12-31/PT5M'
```

and the `Details.Dimension.Default` field informs you that the layer's time extent includes seconds.

```
nexrad.Details.Dimension.Default
```

```
ans =

    '2006-06-23T03:10:00Z'
```

Obtain a time value coincidental with the `katrina` layer, and add seconds to the time specification.

```
nexradTime = [katrina.Details.Dimension.Default(1:end-1) ':00Z'];
```

Assign `latlim` and `lonlim` variables to specify the limits for the `nexrad` layer. Set the values to the limits of the `katrina` layer so that the geographic areas match. Note that the `nexrad` layer's southern latitude limit does not extend as far south as the `katrina` layer's southern latitude limit. The values that lie outside the geographic bounding quadrangle of the `nexrad` layer are set to the background color.

```
fprintf('%s%d\n','Southern latitude limit of NEXRAD  layer: ',nexrad.Latlim(1))
fprintf('%s%d\n','Southern latitude limit of Katrina layer: ',katrina.Latlim(1))
```

```
Southern latitude limit of NEXRAD  layer: 24
Southern latitude limit of Katrina layer: 10
```

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

Assign `imageHeight` and `imageWidth` variables.

```
imageHeight = katrina.Details.Attributes.FixedHeight;
imageWidth  = katrina.Details.Attributes.FixedWidth;
```

Step 7: Retrieve NEXRAD Radar Map from Server

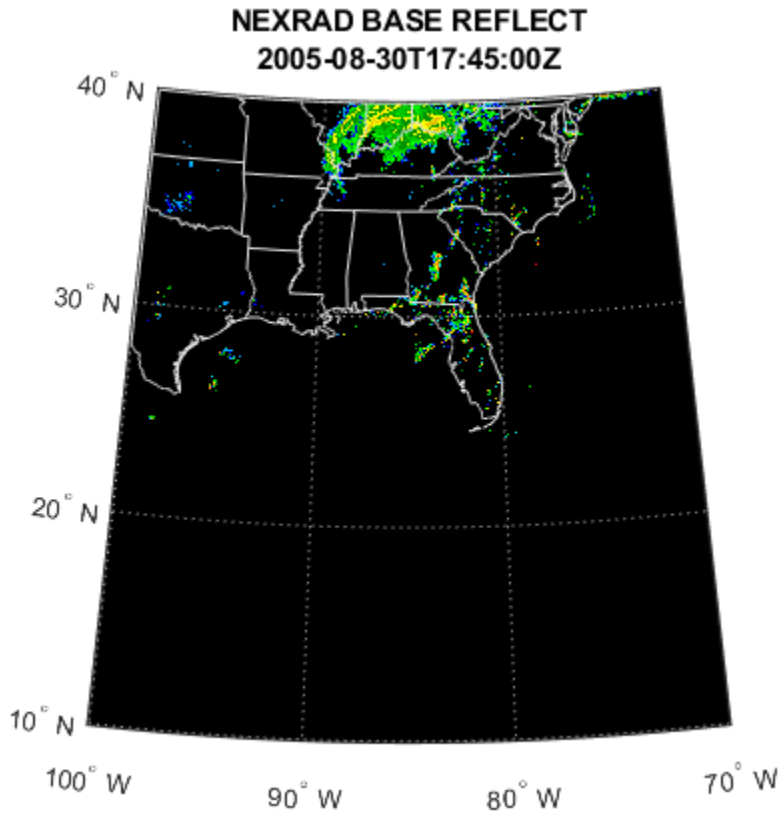
You can retrieve the `nexradMap` from the server, specified at the same time as the `katrinaMap` and for the same geographic and image extents, by supplying parameter/value pairs to the `wmsread` function. To accurately retrieve the radar signal from the map, set the `ImageFormat` parameter to the `image/png` format. In order to easily retrieve the signal from the background, set the background color to black (`[0 0 0]`).

Retrieve the `nexradMap`.

```
black = [0 0 0];
cachefile = datafile('nexrad.tif');
if useInternet
    [nexradMap,R] = wmsread(nexrad, ...
        'Latlim',latlim,'Lonlim',lonlim,'Time',nexradTime, ...
        'BackgroundColor',black,'ImageFormat','image/png', ...
        'ImageHeight',imageHeight,'ImageWidth',imageWidth);
    if ~exist(cachefile, 'file')
        geotiffwrite(cachefile,nexradMap,R)
    end
else
    [nexradMap,R] = readgeoraster(cachefile);
end
```

Display the `nexradMap`.

```
figure
usamap(latlim,lonlim)
geoshow(nexradMap,R)
geoshow(states,'FaceColor','none','EdgeColor',[0.9 0.9 0.9])
title({nexrad.LayerTitle, nexradTime},'Interpreter','none');
```



Step 8: Composite NEXRAD Radar Map with Katrina Map

To composite the `nexradMap` with a copy of the `katrinaMap`, you need to identify the non-background pixels in the `nexradMap`. The `nexradMap` data is returned as an image with class double, because of how this web map server handles `PNG` format, so you need convert it to `uint8` before merging.

Identify the pixels of the `nexradMap` image that do not contain the background color.

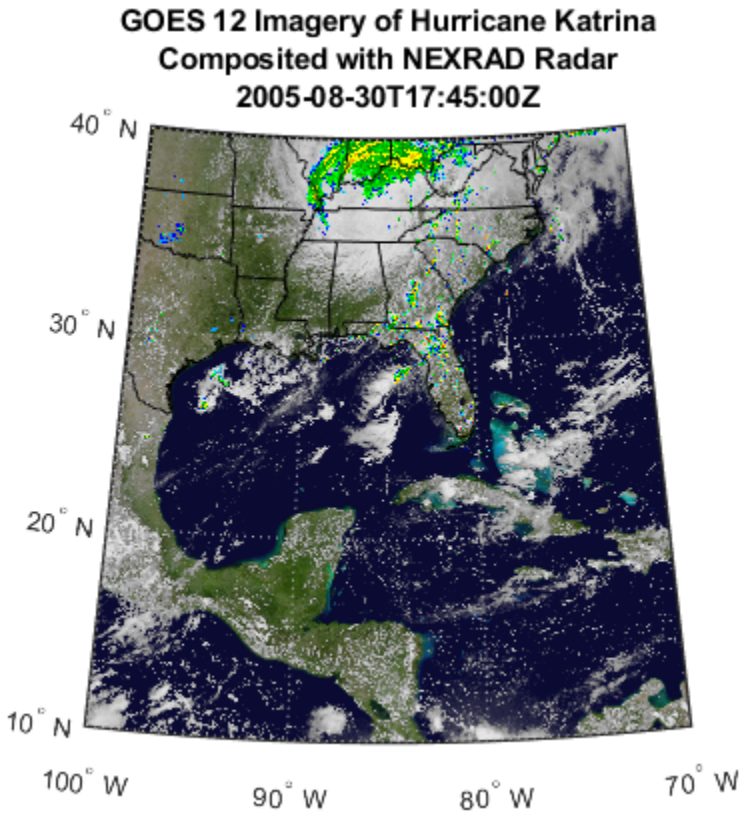
```
threshold = 0;
index = any(nexradMap > threshold, 3);
index = repmat(index,[1 1 3]);
```

Composite the `nexradMap` with the `katrinaMap`.

```
combination = katrinaMap;
combination(index) = uint8(nexradMap(index)*255);
```

Display the composited map.

```
figure
usamap(latlim,lonlim)
geoshow(combination,R)
geoshow(states,'FaceColor','none')
title({'GOES 12 Imagery of Hurricane Katrina', ...
    'Composited with NEXRAD Radar',nexradTime})
```



Step 9: Initialize Variables to Animate the Katrina and NEXRAD Maps

The next step is to initialize variables in order to animate the composited `katrina` and `nexrad` maps.

Create variables that contain the time extent of the `katrina` layer.

```
extent = katrina.Details.Dimension.Extent;
slash = '/';
slashIndex = strfind(extent,slash);
startTime = extent(1:slashIndex(1)-1);
endTime = extent(slashIndex(1)+1:slashIndex(2)-1);
```

Calculate numeric values for the start and end days. Note that the time extent is in `yyyy-mm-dd` format.

```
hyphen = '-';
hyphenIndex = strfind(startTime,hyphen);
dayIndex = [hyphenIndex(2) + 1, hyphenIndex(2) + 2];
startDay = str2double(startTime(dayIndex));
endDay = str2double(endTime(dayIndex));
```

Assign the initial `katrinaTime`.

```
katrinaTime = startTime;
```

Since multiple requests to a server are required for animation, it is more efficient to use the `WebMapServer` and `WMSMapRequest` classes.

Construct a `WebMapServer` object for each layer's server.

```
nasaServer = WebMapServer(katrina.ServerURL);
iemServer  = WebMapServer(nexrad.ServerURL);
```

Create `WMSMapRequest` objects.

```
katrinaRequest = WMSMapRequest(katrina, nasaServer);
nexradRequest  = WMSMapRequest(nexrad, iemServer);
```

Assign properties.

```
nexradRequest.Latlim = latlim;
nexradRequest.Lonlim = lonlim;
nexradRequest.BackgroundColor = black;
nexradRequest.ImageFormat = 'image/png';
nexradRequest.ImageHeight = imageHeight;
nexradRequest.ImageWidth  = imageWidth;
```

Step 10: Create Animation Files

An animation can be viewed in the browser when the browser opens an animated GIF file or an AVI video file. To create the animation frames of the WMS basemap and vector overlays, create a loop through each day, from `startDay` to `endDay`, and obtain the `katrinaMap` and the `nexradMap` for that day. Composite the maps into a single image, display the image, retrieve the frame, and store the results into a frame of an AVI file and a frame of an animated GIF file.

To share with others or to post to web video services, create an AVI video file containing all the frames using the `VideoWriter` class.

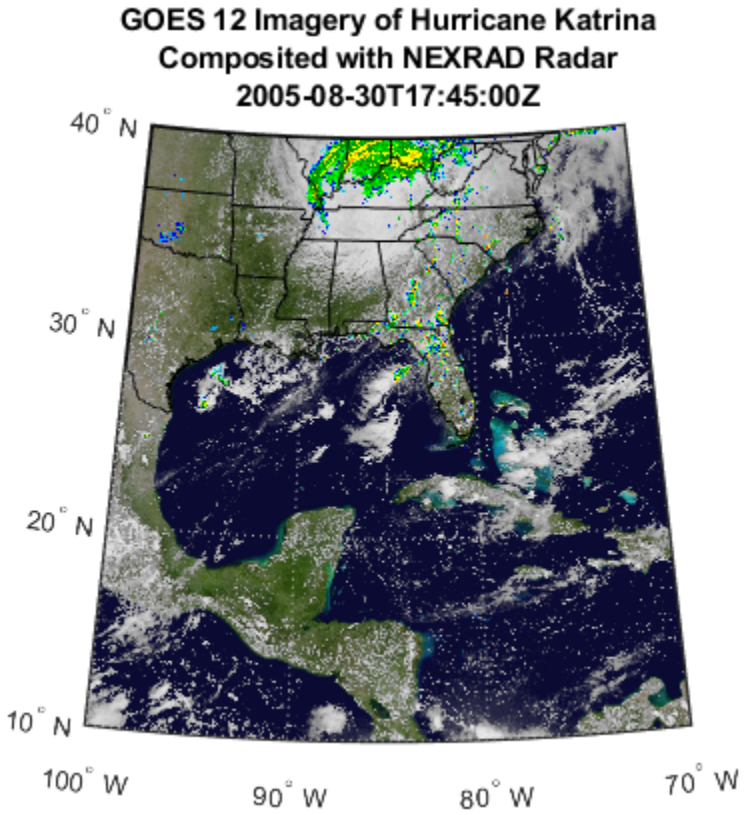
```
videoFilename = fullfile(pwd,'wmsanimated.avi');
if exist(videoFilename,'file')
    delete(videoFilename)
end
writer = VideoWriter(videoFilename);
writer.FrameRate = 1;
writer.Quality = 100;
open(writer)
```

The animation is viewed in a single map display. Outside the animation loop, create a map display. Initialize `hmap`, used in the loop as the return handle from the function `geoshow`, so it can be deleted on the first pass through the loop. Loop through each day, retrieve and display the WMS map, and save the frame.

```
fig = figure;
usamap(latlim,lonlim)
hstates = geoshow(states,'FaceColor','none');
hmap = [];

for k = startDay:endDay

    % Update the time values and assign the Time property for each server.
    currentDay = num2str(k);
    katrinaTime(dayIndex) = currentDay;
    nexradTime = [katrinaTime(1:end-1) ':00Z'];
    katrinaRequest.Time = katrinaTime;
    nexradRequest.Time = nexradTime;
```

```
% Retrieve the WMS map of Katrina from the server (or file)
% for this time period.
cachefile = datafile(['katrina_' num2str(currentDay) '.tif']);
if useInternet
    katrinaMap = getMap(nasaServer, katrinaRequest.RequestURL);
    if ~exist(cachefile, 'file')
        geotiffwrite(cachefile, katrinaMap, katrinaRequest.RasterRef)
    end
else
    katrinaMap = readgeoraster(cachefile);
end

% Retrieve the WMS map of the NEXRAD imagery from the server (or file)
% for this time period.
cachefile = datafile(['nexrad_' num2str(currentDay) '.tif']);
if useInternet
    nexradMap = getMap(iemServer, nexradRequest.RequestURL);
    if ~exist(cachefile, 'file')
        geotiffwrite(cachefile, nexradMap, nexradRequest.RasterRef)
    end
else
    nexradMap = readgeoraster(cachefile);
end

% Identify the pixels of the nexradMap image that do not contain the
% background color.
index = any(nexradMap > threshold, 3);
index = repmat(index,[1 1 3]);

% Composite nexradMap with katrinaMap.
combination = katrinaMap;
combination(index) = uint8(nexradMap(index)*255);

% Delete the old map and display the new composited map.
delete(hmap)
hmap = geoshow(combination, katrinaRequest.RasterRef);
uistack(hstates,'top')
title({'GOES 12 Imagery of Hurricane Katrina', ...
    'Composited with NEXRAD Radar',nexradTime})
drawnow

% Save the current frame as an RGB image.
currentFrame = getframe(fig);
RGB = currentFrame.cdata;

% Create an indexed image for each RGB frame in order to display an
% animated GIF.
if k == startDay
    % The first time through the loop, convert the RGB image to
    % an indexed image and save the colormap into the
    % variable, cmap. Use cmap to convert later frames.
    [frame,cmap] = rgb2ind(RGB,256,'nodither');

    % Use the size of the first frame and the total
    % number of frames to initialize animated with
    % a size large enough to contain all the frames.
    frameSize = size(frame);
    numFrames = endDay - startDay + 1;
```

```
        animated = zeros([frameSize 1 numFrames],'like',frame);
    else
        % Use the colormap from the first frame conversion and
        % convert this frame to an indexed image.
        frame = rgb2ind(RGB,cmap,'nodither');
    end

    % Store the frame into the animated array for the GIF file.
    frameCount = k - startDay + 1;
    animated(:,:,1,frameCount) = frame;

    % Write the RGB frame to the AVI file.
    writeVideo(writer,RGB);
end
```



Close the Figure window and the AVI file.

```
close(fig)
close(writer)
```

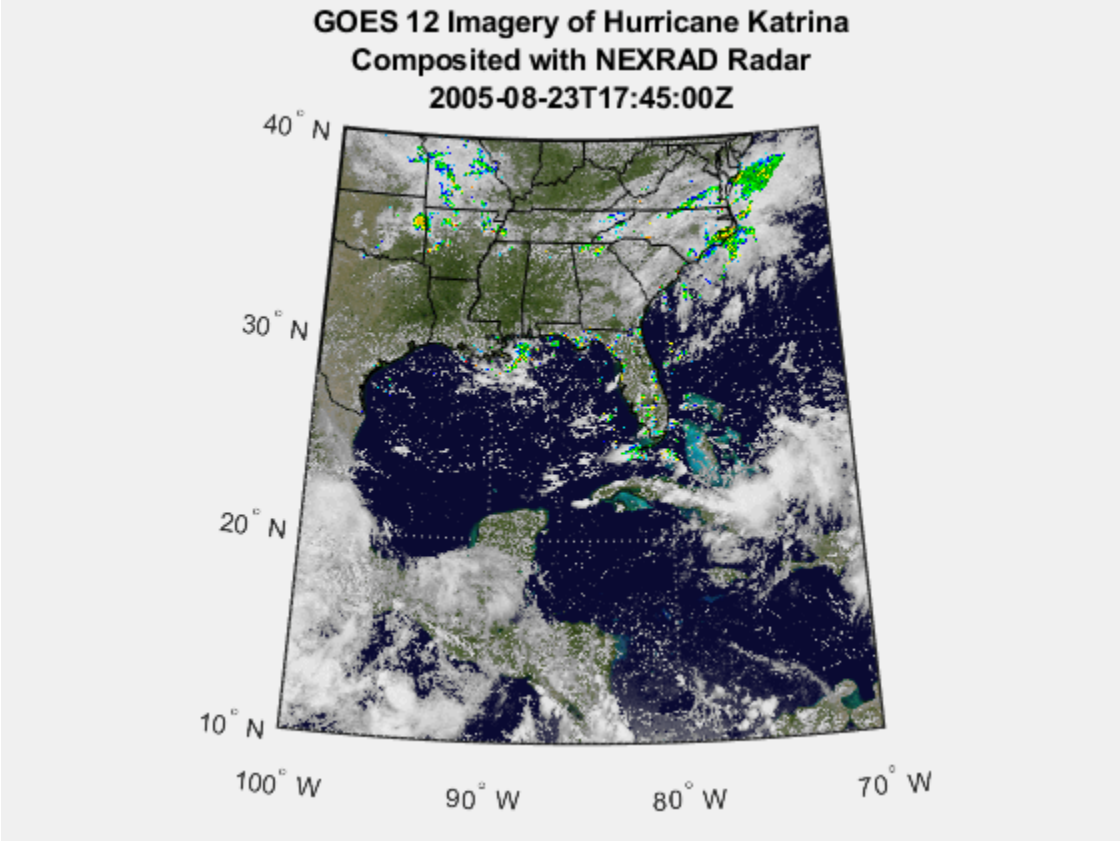
Write the animated GIF file.

```
filename = fullfile(pwd,'wmsanimated.gif');
if exist(filename,'file')
    delete(filename)
end
delayTime = 2.0;
loopCount = inf;
```

```
imwrite(animated,cmap,filename, ...
    'DelayTime',delayTime,'LoopCount',loopCount);
```

Step 11: View Animated GIF File

An animation can be viewed in the browser when the browser opens an animated GIF file.



Credits

Katrina Layer

The Katrina layer used in the example is from the NASA Goddard Space Flight Center's SVS Image Server and is maintained by the Scientific Visualization Studio.

For more information about this server, run:

```
>> wmsinfo('http://svs.gsfc.nasa.gov/cgi-bin/wms?')
```

NEXRAD Layer

The NEXRAD layer used in the example is from the Iowa State University's IEM WMS server and is a generated CONUS composite of National Weather Service (NWS) WSR-88D level III base reflectivity.

For more information about this server, run:

```
>> wmsinfo('http://mesonet.agron.iastate.edu/cgi-bin/wms/nexrad/n0r-t.cgi?')
```

See Also

WMSMapRequest | WebMapServer | geoshow | refine | usamap | wmsfind | wmsread | wmsupdate

Troubleshoot Common Problems with Web Maps

Why Does My Web Map Contain Empty Tiles?

If you create a web map and the display contains empty tiles, it can mean that the web map server is temporarily off line. To display web maps, the Mapping toolbox must create connections to web map providers over the Internet. Often, simply trying again after a few minutes solves the problem.

Why Does My Web Map Lose Detail When I Zoom In?

If you zoom in on a web map and certain details of the map disappear, it can mean that the map does not support that particular zoom level.

See Also

`webmap`

More About

- “Basic Workflow for Displaying Web Maps” on page 9-66

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 for Point Locations on a Sphere” on page 10-2
- “Equal-Areas in Geographic Statistics” on page 10-6
- “Navigation” on page 10-9
- “Fix Position” on page 10-11
- “Plan the Shortest Path” on page 10-20
- “Display Navigational Tracks” on page 10-23
- “Dead Reckoning” on page 10-26
- “Drift Correction” on page 10-30
- “Time Zones” on page 10-32

Geographic Statistics for Point Locations on a Sphere

Certain Mapping Toolbox functions compute basic geographical measures for spatial analysis and for filtering and conditioning data. Since MATLAB functions can compute statistics such as means, medians, and variances, why not use those functions in the toolbox? First of all, classical statistical formulas typically assume that data is one-dimensional (and, often, normally distributed). Because this is not true for geospatial data, spatial analysts have developed statistical measures that extend conventional statistics to higher dimensions.

Second, such formulas generally assume that data occupies a two-dimensional Cartesian coordinate system. Computing statistics for geospatial data with geographic coordinates as if it were in a Cartesian framework can give statistically inappropriate results. While this assumption can sometimes yield reasonable numerical approximations within small geographic regions, for larger areas it can lead to incorrect conclusions because of distance measures and area assumptions that are inappropriate for spheres and spheroids. Mapping Toolbox functions appropriately compute statistics for geospatial data, avoiding these potential pitfalls.

Geographic Means

Consider the problem of calculating the mean position of a collection of geographic points. Taking the arithmetical mean of the latitudes and longitudes using the standard MATLAB `mean` function may seem reasonable, but doing this could yield misleading results.

Take two points at the same latitude, 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°,0°), (0°,120°), and (0°,240°), 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° 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.

See Also

More About

- “Equal-Areas in Geographic Statistics” on page 10-6

Equal-Areas in Geographic Statistics

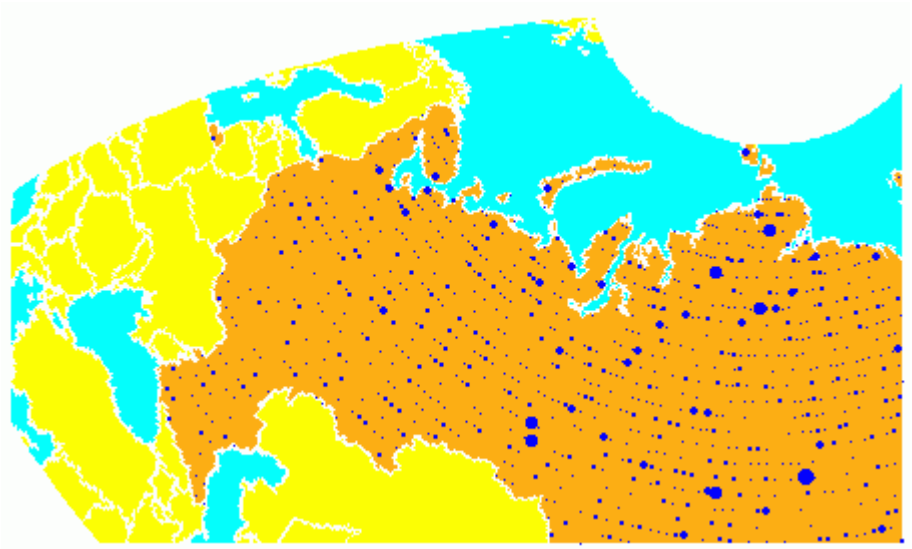
A common error in applying two-dimensional statistics to geographic data lies in ignoring equal-area treatment. It is often necessary to *bin* data to statistically analyze it. In a Cartesian plane, this is easily done by dividing the space into equal *x*-*y* squares. The geographic equivalent of this is to bin up the data in equal latitude-longitude *squares*. Since such squares at high latitudes cover smaller areas than their low-latitude counterparts, the observations in these regions are underemphasized. The result can be conclusions that are biased toward the equator.

Geographic Histograms

The geographic histogram function `histr` allows you to display *binned-up* geographic observations. The `histr` function results in equirectangular binning. Each bin has the same angular measurement in both latitude and longitude, with a default measurement of 1 degree. The center latitudes and longitudes of the bins are returned, as well as the number of observations per bin:

```
[binlat,binlon,num] = histr(lats,lons)
```

As previously noted, these equirectangular bins result in counting bias toward the equator. Here is a display of the one-degree-by-one-degree binning of approximately 5,000 random data points in Russia. The relative size of the circles indicates the number of observations per bin:

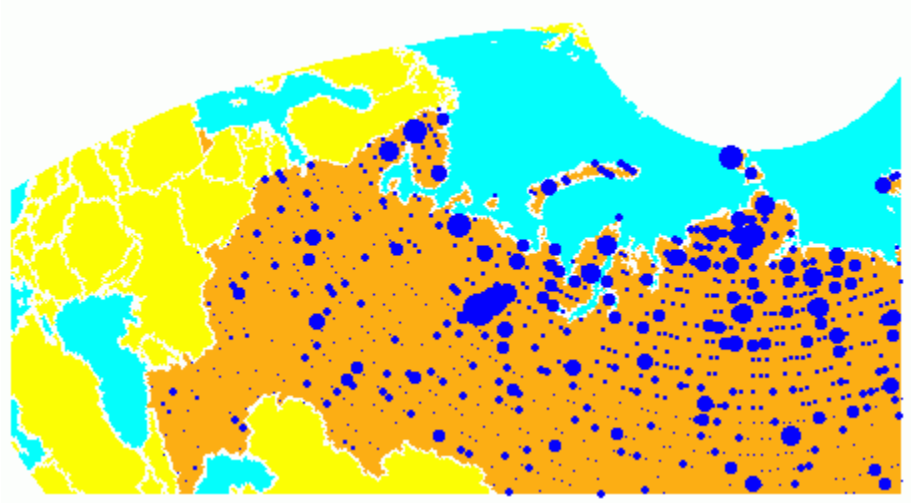


This is a portion of the whole map, displayed in an equal-area Bonne projection. The first step in creating data displays without area bias is to choose an equal-area projection. The proportionally sized symbols are a result of the specialized display function `scatterm`.

You can eliminate the area bias by adding a fourth output argument to `histr`, that will be used to weight each bin's observation by that bin's area:

```
[binlat,binlon,num,wnum] = histr(lats,lons)
```

The fourth output is the weighted observation count. Each bin's observation count is divided by its normalized area. Therefore, a high-latitude bin will have a larger weighted number than a low-latitude bin with the same number of actual observations. The same data and bins look much different when they are area-weighted:



Notice that there are larger symbols to the north in this display. The previous display suggested that the data was relatively uniformly distributed. When equal-area considerations are included, it is clear that the data is skewed to the north. In fact, the data used is northerly skewed, but a simple equirectangular handling failed to demonstrate this.

The `histr` function, therefore, does provide for the display of area-weighted data. However, the actual bins used are of varying areas. Remember, the one-degree-by-one-degree bin near a pole is much smaller than its counterpart near the equator.

The `hista` function provides for actual equal-area bins.

Converting to an Equal-Area Coordinate System

The actual data itself can be converted to an equal-area coordinate system for analysis with other statistical functions. It is easy to convert a collection of geographic latitude-longitude points to an equal-area *x*-*y* Cartesian coordinate system. The `grn2eqa` function applies the same transformation used in calculating the Equal-Area Cylindrical projection:

```
[x,y] = grn2eqa(lat,lon)
```

For each geographic `lat` - `lon` pair, an equal-area `x` - `y` is returned. The variables `x` and `y` can then be operated on under the equal-area assumption, using a variety of two-dimensional statistical techniques. Tools for such analysis can be found in the Statistics and Machine Learning Toolbox™ software and elsewhere. The results can then be converted back to geographic coordinates using the `eqa2grn` function:

```
[lat,lon] = eqa2grn(x, y)
```

Remember, when converting back and forth between systems, latitude corresponds to *y* and longitude corresponds to *x*.

See Also

More About

- “Geographic Statistics for Point Locations on a Sphere” on page 10-2

Navigation

What Is Navigation?

Navigation is the process of planning, recording, and controlling the movement of a craft or vehicle from one location to another. The word derives from the Latin roots *navis* ("ship") and *agere* ("to move or direct"). Geographic information—usually in the form of latitudes and longitudes—is at the core of navigation practice. The toolbox includes specialized functions for navigating across expanses of the globe, for which projected coordinates are of limited use.

Navigating on land, over water, and through the air can involve a variety of tasks:

- Establishing position, using known, fixed landmarks (piloting)
- Using the stars, sun, and moon (celestial navigation)
- Using technology to fix positions (inertial guidance, radio beacons, and satellite navigation, including GPS)
- Deducing net movement from a past known position (dead reckoning)

Another navigational task involves planning a voyage or flight, which includes determining an efficient route (usually by great circle approximation), weather avoidance (optimal track routing), and setting out a plan of intended movement (track laydown). Mapping Toolbox functions support these navigational activities as well.

Conventions for Navigational Functions

Units

You can use and convert among several angular and distance measurement units. The navigational support functions are

- `dreckon`
- `gcwaypts`
- `legs`
- `navfix`

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

- Angles are always in degrees.
- Distances are always in nautical miles.
- Speeds are always in knots (nautical miles per hour).

Related functions that *do not* carry this restriction include `rhxrh`, `scxsc`, `gcxgc`, `gcxsc`, `track`, `timezone`, and `crossfix`, because of their potential for application outside navigation.

Navigational Track Format

Navigational track format requires column-vector variables for the latitudes and longitudes of track waypoints. A *waypoint* is a point through which a track passes, usually corresponding to a course (or speed) change. Navigational tracks are made up of the line segments connecting these waypoints,

which are called *legs*. In this format, therefore, *n* legs are described using *n+1* waypoints, because an endpoint for the final leg must be defined. Mapping Toolbox navigation functions always presume angle units are always given in degrees.

Here, five track legs require six waypoints. In navigational track format, the waypoints are represented by two 6-by-1 vectors, one for the latitudes and one for the longitudes.

See Also

More About

- “Fix Position” on page 10-11
- “Plan the Shortest Path” on page 10-20
- “Display Navigational Tracks” on page 10-23
- “Dead Reckoning” on page 10-26
- “Drift Correction” on page 10-30
- “Time Zones” on page 10-32

Fix Position

The fundamental objective of navigation is to determine at a given moment how to proceed to your destination, avoiding hazards on the way. The first step in accomplishing this is to establish your current position. Early sailors kept within sight of land to facilitate this. Today, navigation within sight (or radar range) of land is called *piloting*. Positions are fixed by correlating the bearings and/or ranges of landmarks. In real-life piloting, all sighting bearings are treated as rhumb lines, while in fact they are actually great circles.

Over the distances involved with visual sightings (up to 20 or 30 nautical miles), this assumption causes no measurable error and it provides the significant advantage of allowing the navigator to plot all bearings as straight lines on a Mercator projection.

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

These assumptions also lead to computationally efficient methods for fixing positions with a computer. The toolbox includes the `navfix` function, which mimics the manual plotting and fixing process using these assumptions.

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

Some Possible Situations

In this imaginary coastal region, you take a visual bearing on the radio tower of 270°. At the same time, Gilligan's Lighthouse bears 0°. If you plot a 90°-270° line through the radio tower and a 0°-180° 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°. 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×(*n*-choose-2) possible intersections result. In this example, it is easy to tell which ones are *right*.

Bearing lines and arcs can be combined. If instead of reporting a third range, your radar watch had reported a bearing from the radar tower of 20°, the ambiguity could also have been resolved. Note, however, that in practice, lines of bearing for navigational fixing should only be taken visually, except in desperation. A radar's beam width can be a degree or more, leading to uncertainty.

As you begin to wonder whether this manual plotting process could be automated, your first officer shows up on the bridge with a laptop and Mapping Toolbox software.

Using nav x

The `navfix` function can be used to determine the points of intersection among any number of lines and arcs. Be warned, however, that due to the combinatorial nature of this process, the computation time grows rapidly with the number of objects. To illustrate this function, assign positions to the landmarks. Point A, Cape Jones, is at (`latA`,`lonA`). Point B, the radio tower, is at (`latB`,`lonB`). Point C, Gilligan's Lighthouse, is at (`latC`,`lonC`).

For the bearing-lines-only example, the syntax is:

```
[latfix,lonfix] = navfix([latA latB latC],[lonA lonB lonC],...
                         [300 270 0])
```

This defines the three points and their bearings as taken *from the ship*. The outputs would look something like this, with actual numbers, of course:

```
latfix =

    latfix1       NaN          % A intersecting B
    latfix2       NaN          % A intersecting C
    latfix3       NaN          % B intersecting C

lonfix =

    lonfix1       NaN          % A intersecting B
    lonfix2       NaN          % A intersecting C
    lonfix3       NaN          % B intersecting C
```

Notice that these are two-column matrices. The second column consists of `NaN`s because it is used only for the two-intersection ambiguity associated with arcs.

For the range-arcs-only example, the syntax is

```
[latfix,lonfix] = navfix([latA latB latC],[lonA lonB lonC],...
                         [16 14 15],[0 0 0])
```

This defines the three points and their ranges as taken from the ship. The final argument indicates that the three cases are all ranges.

The outputs have the following form:

```
latfix =

    latfix11  latfix12          % A intersecting B
    latfix21  latfix22          % A intersecting C
    latfix31  latfix32          % B intersecting C

lonfix =

    lonfix11  lonfix12          % A intersecting B
    lonfix21  lonfix22          % A intersecting C
    lonfix31  lonfix32          % B intersecting C
```

Here, the second column is used, because each pair of arcs has two potential intersections.

For the bearings and ranges example, the syntax requires the final input to indicate which objects are lines of bearing (indicated with a 1) and which are range arcs (indicated with a 0):

```
[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                         [20 14 15],[1 0 0])
```

The resulting output is mixed:

```
latfix =

    latfix11       NaN          % Line B intersecting Arc B
    latfix21  latfix22          % Line B intersecting Arc C
    latfix31  latfix32          % Arc B intersecting Arc C

lonfix =

    lonfix11       NaN          % Line B intersecting Arc B
    lonfix21  lonfix22          % Line B intersecting Arc C
    lonfix31  lonfix32          % Arc B intersecting Arc C
```

Only one intersection is returned for the line from B with the arc about B, since the line originates inside the circle and intersects it once. The same line intersects the other circle twice, and hence it returns two points. The two circles taken together also return two points.

Usually, you have an idea as to where you are before you take the fix. For example, you might have a dead reckoning position for the time of the fix (see below). If you provide `navfix` with this estimated position, it chooses from each pair of ambiguous intersections the point closest to the estimate. Here's what it might look like:

```
[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                         [20 14 15],[1 0 0],drlat,drlon)
```

```
latfix =

    latfix11                    % the only point
    latfix21                    % the closer point
    latfix31                    % the closer point

lonfix =

    lonfix11                    % the only point
    lonfix21                    % the closer point
    lonfix31                    % the closer point
```

A Numerical Example of Using navx

1 Define some specific points in the middle of the Atlantic Ocean. These are strictly arbitrary; perhaps they correspond to points in Atlantis:

```
lata = 3.1;  lona = -56.2;
latb = 2.95; lonb = -55.9;
latc = 3.15; lonc = -55.95;
```

2 Plot them on a Mercator projection:

```
axesm('MapProjection','mercator','Frame','on',...
    'MapLatLimit',[2.8 3.3],'MapLonLimit',[-56.3 -55.8])
plotm([lata latb latc],[lona lonb lonc],...
    'LineStyle','none','Marker','pentagram',...
    'MarkerEdgeColor','b','MarkerFaceColor','b',...
    'MarkerSize',12)
```

Here is what it looks like (with labeling and imaginary coastlines added after the fact for illustration):

3 Take three visual bearings: Point A bears 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 intersection points to the map:

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

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

5 What if instead, you had ranges from the three points, A, B, and C, of 13 nmi, 9 nmi, and 7.5 nmi, respectively?

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [13 9 7.5],[0 0 0])
```

```
newlat =
```

```
    3.0739    2.9434
    3.2413    3.0329
    3.0443    3.0880

newlong =

   -55.9846  -56.0501
   -56.0355  -55.9937
   -56.0168  -55.8413
```

Here's what these points look like:

Three of these points look reasonable, three do not.

6 What if, instead of a range from Point A, you had a bearing to it of 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,drlon)
```

```
newlat =

    3.0526
    3.0592
    3.0443

newlong =

   -56.0096
   -56.0360
   -56.0168
```

See Also

More About

- “Navigation” on page 10-9
- “Display Navigational Tracks” on page 10-23
- “Plan the Shortest Path” on page 10-20
- “Dead Reckoning” on page 10-26

Plan the Shortest Path

You know that the shortest path between two geographic points is a great circle. Sailors and aviators are interested in minimizing distance traveled, and hence time elapsed. You also know that the rhumb line is a path of constant heading, the *natural* means of traveling. In general, to follow a great circle path, you would have to continuously alter course. This is impractical. However, you can approximate a great circle path by rhumb line segments so that the added distance is minor and the number of course changes minimal.

Surprisingly, very few rhumb line *track legs* are required to closely approximate the distance of the great circle path.

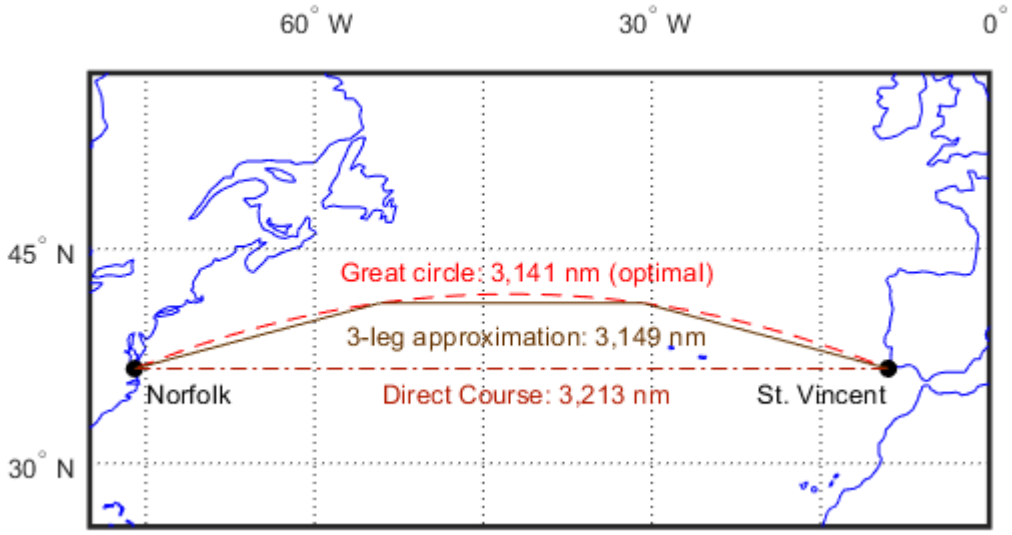
Consider the voyage from Norfolk, Virginia (37°N,76°W), to Cape St. Vincent, Portugal (37°N,9°W), one of the most heavily trafficked routes in the Atlantic. A due-east rhumb line track is 3,213 nautical miles, while the optimal great circle distance is 3,141 nautical miles.

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 coastlines;
hg = geoshow(coastlat,coastlon,'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')
text(-0.61,0.66,'Norfolk','HorizontalAlignment','left')
geoshow(stvincent(1),stvincent(2),'DisplayType','point',...
    'markeredgecolor','k','markerfacecolor','k','marker','o')
text(0.54,0.66,'St. Vincent','HorizontalAlignment','right')

% 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','--')
text(-0.02,0.85,'Great circle: 3,141 nm (optimal)',...
    'color','r','HorizontalAlignment','center')

% 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','-.')
text(-0.02,0.66,'Direct course: 3,213 nm',...
```

```
    'color',[.7 .1 0],'HorizontalAlignment','center')

% Compute and draw path along three waypoints
[latpts,lonpts] = gcwaypts(norfolk(1),norfolk(2),...
    stvincent(1),stvincent(2),3);
geoshow(latpts,lonpts,'DisplayType','line',...
    'color',[.4 .2 0],'linestyle','-')
text(-0.02,0.75,'3-leg approximation: 3,149 nm',...
    'color',[.4 .2 0],'HorizontalAlignment','center')
```

The resulting tracks and distances are shown below:



The Mapping Toolbox function `gcwaypts` calculates waypoints in navigation track format in order to approximate a great circle with rhumb line segments. It uses this syntax:

```
[latpts,lonpts] = gcwaypts(lat1,lon1,lat2,lon2,numlegs)
```

All the inputs for this function are scalars a (starting and an ending position). The `numlegs` input is the number of equal-length legs desired, which is 10 by default. The outputs are column vectors representing waypoints in navigational track format (`[heading distance]`). The size of each of these vectors is `[(numlegs+1) 1]`. Here are the points for this example:

```
[latpts,lonpts] = gcwaypts(norfolk(1),norfolk(2),...
stvincent(1),stvincent(2),3)   % Compute 3 waypoints

latpts =

   37.0000
   41.5076
   41.5076
   37.0000

lonpts =

  -76.0000
  -54.1777
  -30.8223
   -9.0000
```

These points represent waypoints along the great circle between which the approximating path follows rhumb lines. Four points are needed for three legs, because the final point at Cape St. Vincent must be included.

Now we can compute the distance in nautical miles (nm) along each track and via the waypoints:

```
drh = distance('rh',norfolk,stvincent); % Get rhumb line dist (deg)
dgc = distance('gc',norfolk,stvincent); % Get gt. circle dist (deg)
% Compute headings and distances for the waypoint legs
[course distnm] = legs(latpts,lonpts,'rh');
```

Finally, compare the distances:

```
distrhnm = deg2nm(drh)            % Nautical mi along rhumb line
distgcnm = deg2nm(dgc)            % Nautical mi along great circle
distlegsnm = sum(distnm)          % Total dist along the 3 legs
rhgcdiff = distrhnm - distgcnm    % Excess rhumb line distance
trgcdiff = distlegsnm - distgcnm  % Excess distance along legs
```

```
distrhnm =

  3.2127e+003

distgcnm =

  3.1407e+003

distlegsnm =

  3.1490e+003

rhgcdiff =

   71.9980

trgcdiff =

    8.3446
```

Following just three rhumb line legs reduces the distance travelled from 72 nm to 8.3 nm compared to a great circle course.

See Also

More About

- “Navigation” on page 10-9
- “Fix Position” on page 10-11
- “Display Navigational Tracks” on page 10-23
- “Dead Reckoning” on page 10-26

Display Navigational Tracks

Navigational tracks are most useful when graphically displayed. Traditionally, the navigator identifies and plots waypoints on a Mercator projection and then connects them with a straightedge, which on this projection results in rhumb line tracks. In the previous example, waypoints were chosen to approximate a great circle route, but they can be selected for a variety of other reasons.

Let's say that after arriving at Cape St. Vincent, your tanker must traverse the Straits of Gibraltar and then travel on to Port Said, the northern terminus of the Suez Canal. On the scale of the Mediterranean Sea, following great circle paths is of little concern compared to ensuring that the many straits and passages are safely transited. The navigator selects appropriate waypoints and plots them.

To accomplish this with Mapping Toolbox functions, you can display a map axes with a Mercator projection, select appropriate map latitude and longitude limits to isolate the area of interest, plot coastline data, and interactively mouse-select the waypoints with the `inputm` function. The `track` function will generate points to connect these waypoints, which can then be displayed with `plotm`.

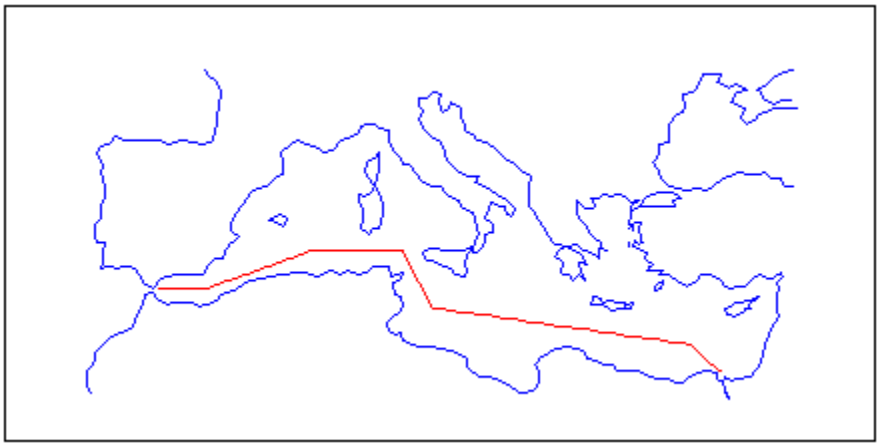
For illustration, assume that the waypoints are known (or were gathered using `inputm`). To learn about using `inputm`, see “Pick Locations Interactively” on page 4-125, or `inputm` in the Mapping Toolbox reference pages.

```
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 coastlines
axesm('MapProjection','mercator',...
'MapLatLimit',[30 47],'MapLonLimit',[-10 37])
framem
plotm(coastlat,coastlon)

[lttrk,lntrk] = track(waypoints);
plotm(lttrk,lntrk,'r')
```

Although these track segments are straight lines on the Mercator projection, they are curves on others:



The segments of a track like this are called *legs*. Each of these legs can be described in terms of course and distance. The function `legs` will take the waypoints in navigational track format and return the course and distance required for each leg. Remember, the order of the points in this format determines the direction of travel. Courses are therefore calculated from each waypoint to its successor, not the reverse.

```
[courses,distances] = legs(waypoints)
```

```
courses =

    90.0000
    70.3132
    90.0000
   151.8186
    98.0776
   131.5684

distances =

   145.6231
   356.2117
   283.6839
   204.2073
   854.0092
   135.6415
```

Since this is a navigation function, the courses are all in degrees and the distances are in nautical miles. From these distances, speeds required to arrive at Port Said at a given time can be calculated. Southbound traffic is allowed to enter the canal only once per day, so this information might be economically significant, since unnecessarily high speeds can lead to high fuel costs.

See Also

More About

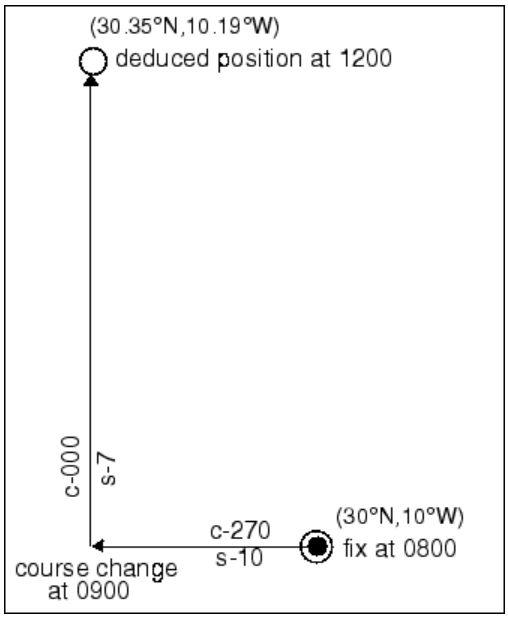
- “Navigation” on page 10-9
- “Fix Position” on page 10-11

- “Plan the Shortest Path” on page 10-20
- “Dead Reckoning” on page 10-26

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.

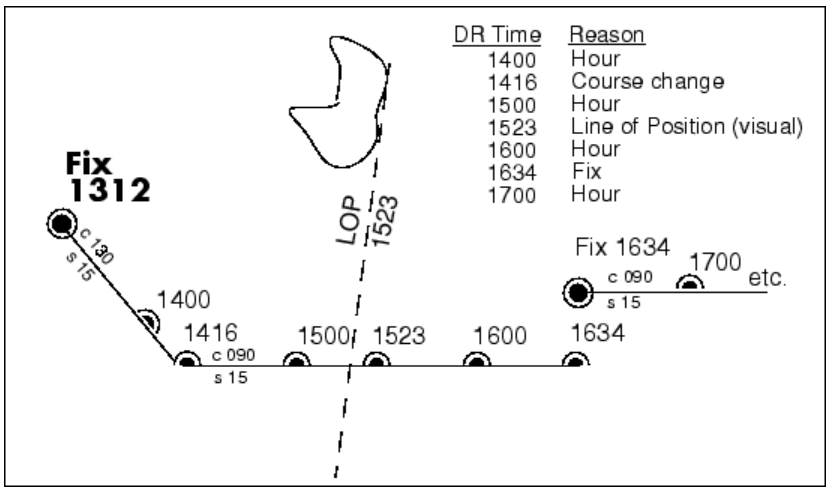
When navigators establish a fix from some source, be it from piloting, celestial, or satellite observations, they plot a dead reckoning (DR) track, which is a plot of the intended positions of the ship forward in time. In practice, dead reckoning is usually plotted for 3 hours in advance, or for the 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 *DR*s, are plotted according to the *Rules of DR*:

- DR at every course change
- DR at every speed change
- DR every hour on the hour
- DR every time a fix or running fix is obtained

- DR 3 hours ahead or for the next three expected fixes
- DR for every line of position (LOP), either visual or celestial

For example, the navigator plots these DRs:



Notice that the 1523 DR does not coincide with the LOP at 1523. Although note is taken of this variance, one line is insufficient to calculate a new fix.

Mapping Toolbox function `dreckon` calculates the DR positions for a given set of courses and speeds. The function provides DR positions for the first three rules of dead reckoning. The approach is to provide a set of waypoints in navigational track format corresponding to the plan of intended movement.

The time of the initial waypoint, or fix, is also needed, as well as the speeds to be employed along each leg. Alternatively, a set of speeds and the times for which each speed will apply can be provided. `dreckon` returns the positions and times required of these DRs:

- `dreckon` calculates the times for position of each course change, which will occur at the waypoints
- `dreckon` calculates the positions for each whole hour
- If times are provided for speed changes, `dreckon` calculates positions for these times if they do not occur at course changes

Imagine you have a fix at midnight at the point (10°N,0°):

```
waypoints(1,:) = [10 0]; fixtime = 0;
```

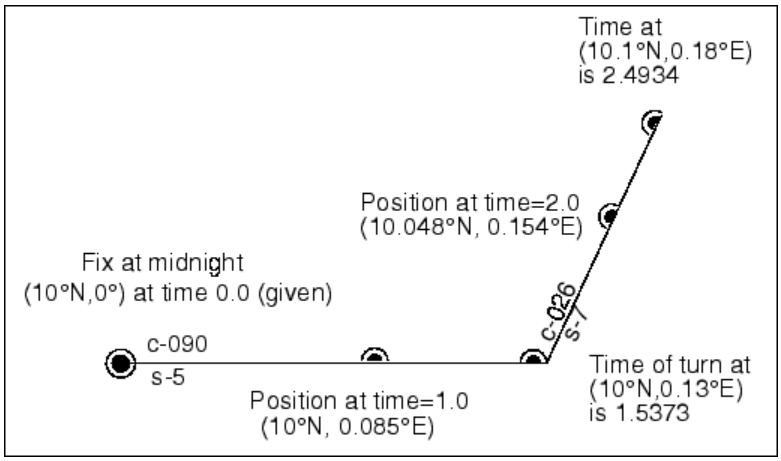
You intend to travel east and alter course at the point (10°N,0.13°E) and head for the point (10.1°N,0.18°E). On the first leg, you will travel at 5 knots, and on the second leg you will speed up to 7 knots.

```
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,drtime] = dreckon(waypoints,fixtime,speeds);
[drlat drlon drtime]

ans =

    10.0000    0.0846    1.0000     % Position at 1 am
    10.0000    0.1301    1.5373     % Time of course change
    10.0484    0.1543    2.0000     % Position at 2 am
    10.1001    0.1801    2.4934     % Time at final waypoint
```

Here is an illustration of this track and its DR points:

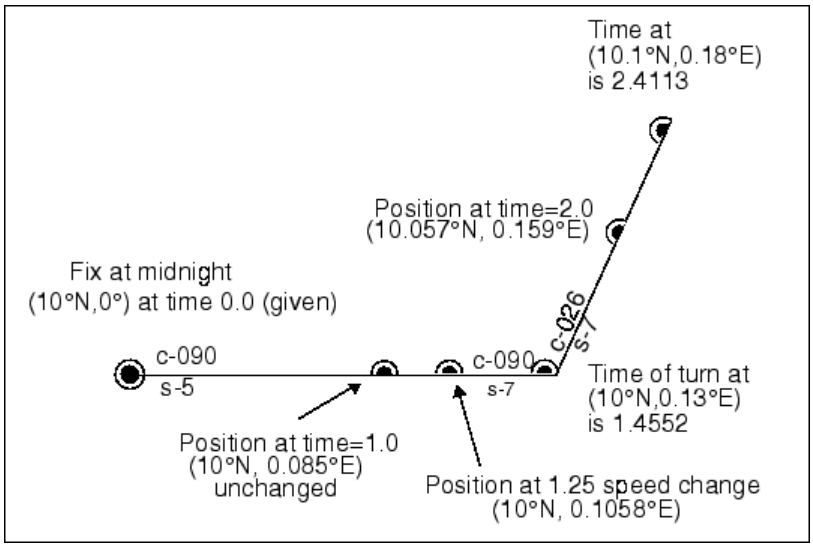


However, you would like to get to the final point a little earlier to make a rendezvous. You decide to recalculate your DRs based on speeding up to 7 knots a little earlier than planned. The first calculation tells you that you were going to increase speed at the turn, which would occur at a time 1.5373 hours after midnight, or 1:32 a.m. (at time 0132 in navigational time format). What time would you reach the rendezvous if you increased your speed to 7 knots at 1:15 a.m. (0115, or 1.25 hours after midnight)?

To indicate times for speed changes, another input is required, providing a time interval after the fix time at which each ordered speed is to end. The first speed, 5 knots, is to end 1.25 hours after midnight. Since you don't know when the rendezvous will be made under these circumstances, set the time for the second speed, 7 knots, to end at infinity. No DRs will be returned past the last waypoint.

```
spdtimes = [1.25; inf];
[drlat,drlon,drtime] = dreckon(waypoints,fixtime,...
                               speeds,spdtimes);
[drlat,drlon,drtime]

ans =

    10.0000    0.0846    1.0000   % Position at 1 am
    10.0000    0.1058    1.2500   % Position at speed change
    10.0000    0.1301    1.4552   % Time of course change
    10.0570    0.1586    2.0000   % Position at 2 am
    10.1001    0.1801    2.4113   % Time at final waypoint
```

This following illustration shows the difference:



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

See Also

More About

- “Navigation” on page 10-9
- “Fix Position” on page 10-11
- “Plan the Shortest Path” on page 10-20
- “Display Navigational Tracks” on page 10-23
- “Drift Correction” on page 10-30

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
```

See Also

More About

- “Navigation” on page 10-9
- “Fix Position” on page 10-11
- “Plan the Shortest Path” on page 10-20
- “Display Navigational Tracks” on page 10-23
- “Dead Reckoning” on page 10-26

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 there are no Mapping Toolbox functions designed specifically for celestial navigation, a simple example can be devised.

It is possible with a sextant to determine *local apparent noon*. This is the moment when the Sun is at its zenith from your point of view. At the exact center longitude of a time zone, the phenomenon occurs exactly at noon, local time. Since the Sun traverses a 15° time zone in 1 hour, it crosses one degree every 4 minutes. So if you observe local apparent noon at 11:54, you must be 1.5° east of your center longitude.

You must know what time zone you are in before you can even attempt a fix. This concept has been understood since the spherical nature of the Earth was first accepted, but early sailors had no ability to keep accurate time on ship, and so were unable to determine their longitude. The invention of accurate chronometers in the 18th century solved this problem.

The `timezone` function is quite simple. It returns `zd`, an integer for use in calculations, `zltr`, a character vector of the zone designator, and `zone`, a character vector 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.

See Also

More About

- “Navigation” on page 10-9

Map Projections Alphabetical List

aito

Aitoff projection

Classi cation

Modified Azimuthal

Identi er

`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.

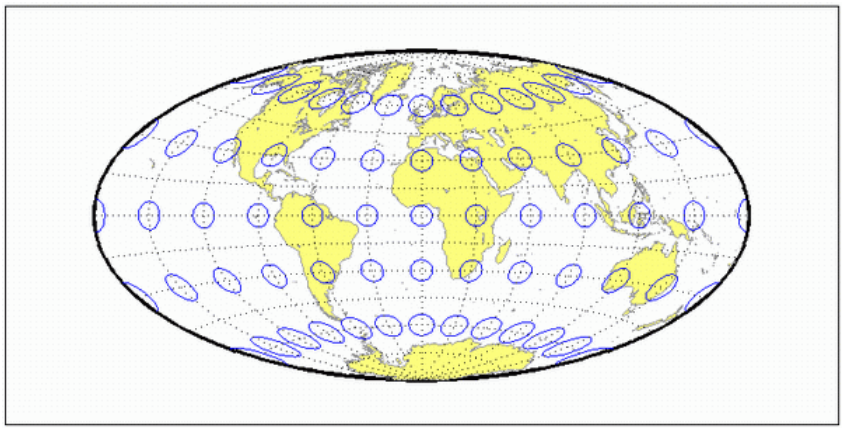
Remarks

- 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.
- This implementation of the Aitoff projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Aitoff projection using the ESRI authority code 54043. For example: `projcrs(54043,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('aitoff', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eqaconic

Albers Equal-Area Conic Projection

Classification

Conic

Identifier

`eqaconic`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the point of convergence. Spacing of parallels decreases away from the central latitudes.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

This is an equal-area projection. Scale is true along the one or two selected standard parallels. Scale is constant along any parallel; the scale factor of a meridian at any given point is the reciprocal of that along the parallel to preserve equal-area. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is neither conformal nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane and a Lambert Azimuthal Equal-Area projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Equal-Area Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Equal-Area Cylindrical projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

Remarks

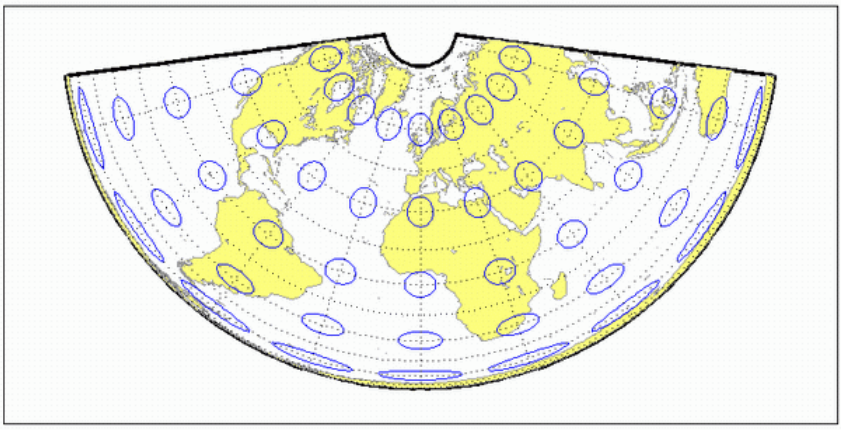
This projection was presented by Heinrich Christian Albers in 1805.

Limitations

Longitude data greater than 135º 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 on page 11-6

Introduced before R2006a

eqaconicstd

Albers Equal-Area Conic Projection — Standard

Classification

Conic

Identifier

`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` on page 11-4 for an alternative implementation based on rotating the authalic sphere.

This is an equal area projection. Scale is true along the one or two selected standard parallels. Scale is constant along any parallel; the scale factor of a meridian at any given point is the reciprocal of that along the parallel to preserve equal area. The projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is neither conformal nor equidistant.

Parallels

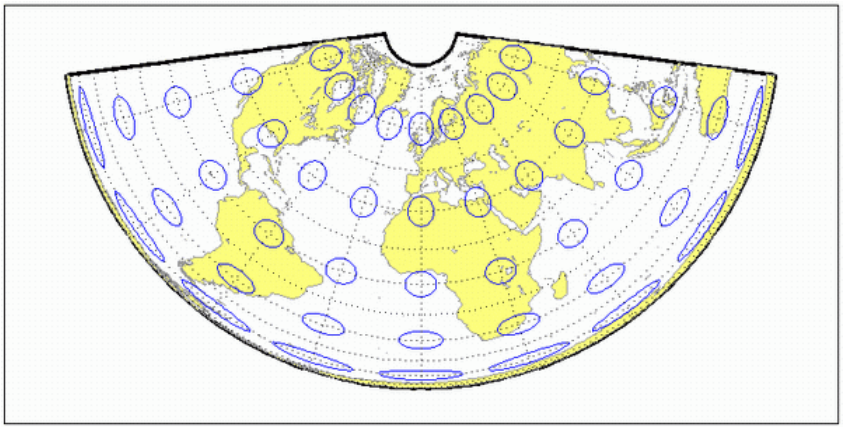
The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane and a Lambert Azimuthal Equal-Area projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Equal-Area Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Equal-Area Cylindrical projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other equal-area cylindrical projection is the result. Suggested parallels for maps of the conterminous U.S. are [29.5 45.5]. The default parallels are [15 75].

Remarks

- This projection was presented by Heinrich Christian Albers in 1805 and it is also known as a Conical Orthomorphic projection. The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Lambert Equal Area Conic projection is the result. If the Equator is chosen as a single parallel, the cone becomes a cylinder and a Lambert Cylindrical Equal Area Projection is the result. Finally, if two parallels equidistant from the Equator are chosen as the standard parallels, a Behrmann or other cylindrical equal area projection is the result.
- Mapping Toolbox uses a different implementation of the standard Albers equal-area conic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

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` on page 11-4

Introduced before R2006a

apianus

Apianus II Projection

Classification

Pseudocylindrical

Identifier

`apianus`

Graticule

Meridians: Equally spaced elliptical curves converging at the poles.

Parallels: Equally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

Scale is constant along any parallel or pair of parallels equidistant from the Equator, as well as along the central meridian. The Equator is free of angular distortion. This projection is not equal-area, equidistant, or conformal.

Parallels

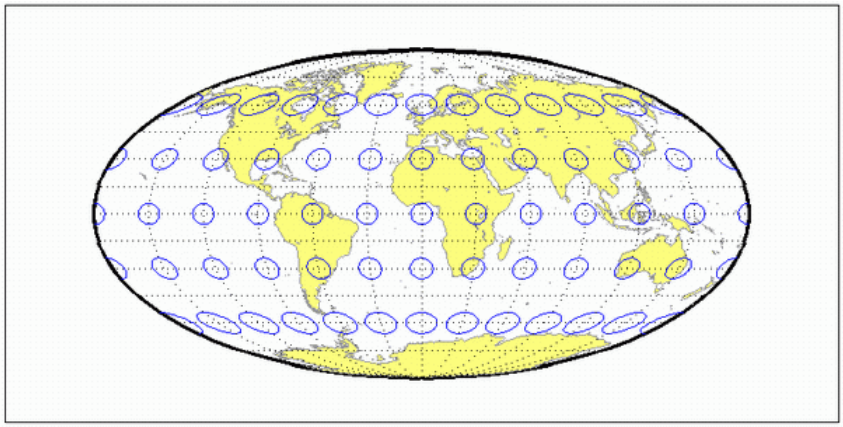
There is no standard parallel for this projection.

Remarks

- This projection was first described in 1524 by Peter Apian (or Bienewitz).
- This implementation of the Apianus II projection is applicable only for coordinates that are referenced to a sphere.

Example

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



Introduced before R2006a

balthsrt

Balthasart Cylindrical Projection

Classi cation

Cylindrical

Identi er

```
balthsrt
```

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 50º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Standard Parallels

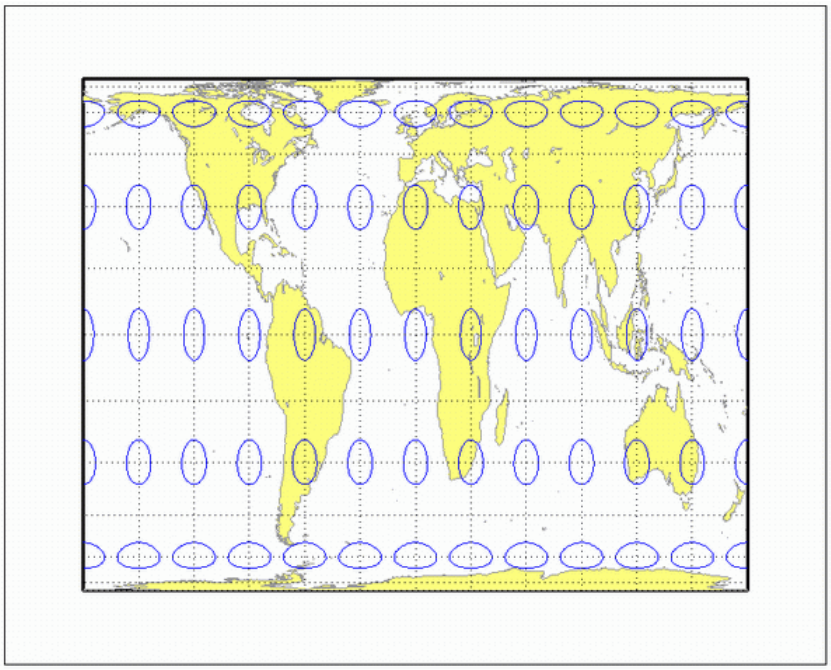
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 50º.

Remarks

The Balthasart Cylindrical projection was presented in 1935 and is a special form of the Equal-Area Cylindrical projection secant at 50ºN and S.

Example

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



Introduced before R2006a

behrmann

Behrmann Cylindrical Projection

Classification

Cylindrical

Identifier

behrmann

Graticule

Meridians: Equally spaced straight parallel lines 0.42 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 30º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

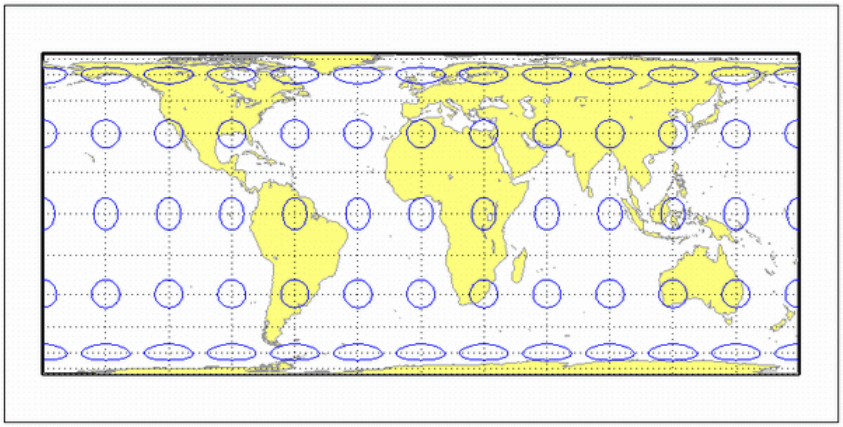
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 30º.

Remarks

This projection is named for Walter Behrmann, who presented it in 1910 and is a special form of the Equal-Area Cylindrical projection secant at 30ºN and S.

Example

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



Introduced before R2006a

bsam

Bolshoi Sovietskii Atlas Mira Projection

Classification

Cylindrical

Identifier

`bsam`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at the 30º parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 30º.

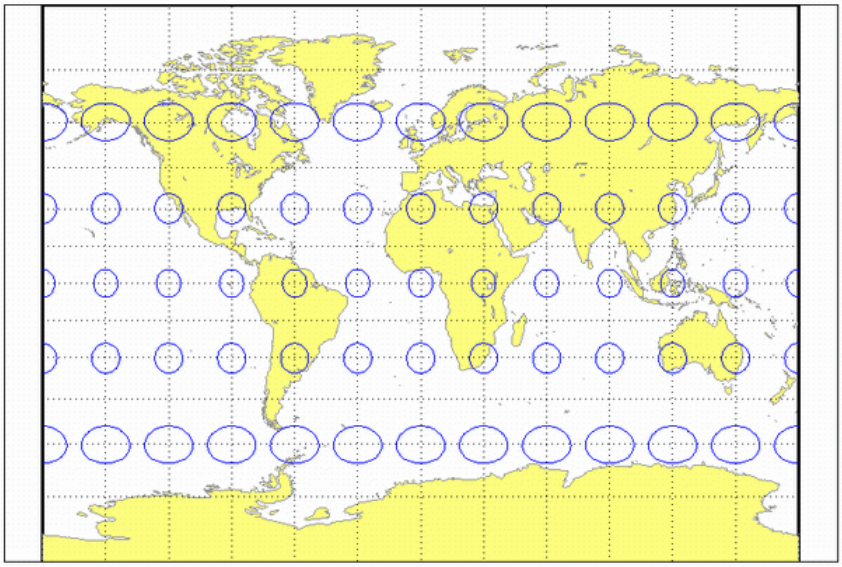
Remarks

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



Introduced before R2006a

bonne

Bonne Projection

Classification

Pseudoconic

Identifier

bonne

Graticule

Central Meridian: A straight line.

Meridians: Complex curves connecting points equally spaced along each parallel and concave toward the central meridian.

Parallels: Concentric circular arcs spaced at true distances along the central meridian.

Poles: Points.

Symmetry: About the central meridian.

Features

This is an equal-area projection. The curvature of the standard parallel is identical to that on a cone tangent at that latitude. The central meridian and the central parallel are free of distortion. This projection is not conformal.

Parallels

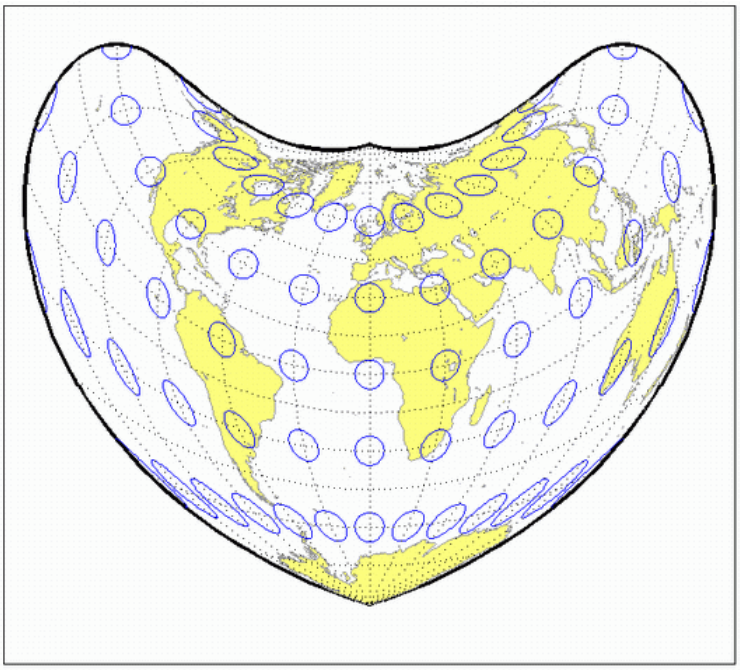
This projection has one standard parallel, which is 30ºN by default. It has two interesting limiting forms. If a pole is employed as the standard parallel, a Werner projection results; if the Equator is used, a Sinusoidal projection results.

Remarks

This projection dates in a rudimentary form back to Claudius Ptolemy (about A.D. 100). It was further developed by Bernardus Sylvanus in 1511. It derives its name from its considerable use by Rigobert Bonne, especially in 1752.

Example

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



Introduced before R2006a

braun

Braun Perspective Cylindrical Projection

Classification

Cylindrical

Identifier

`braun`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at standard parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude may be chosen; the default is arbitrarily set to 0º.

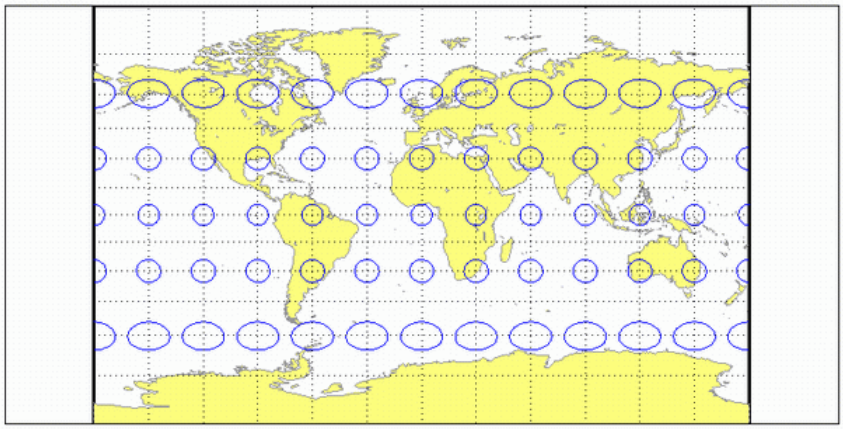
Remarks

- This projection was first described by Braun in 1867. It is less well known than the specific forms of it called the Gall Stereographic and the *Bolshoi Sovietskii Atlas Mira* projections.
- This implementation of the Braun perspective cylindrical projection is applicable only for coordinates that are referenced to a 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;
```



Introduced before R2006a

breusing

Breusing Harmonic Mean Projection

Classification

Azimuthal

Identifier

`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.

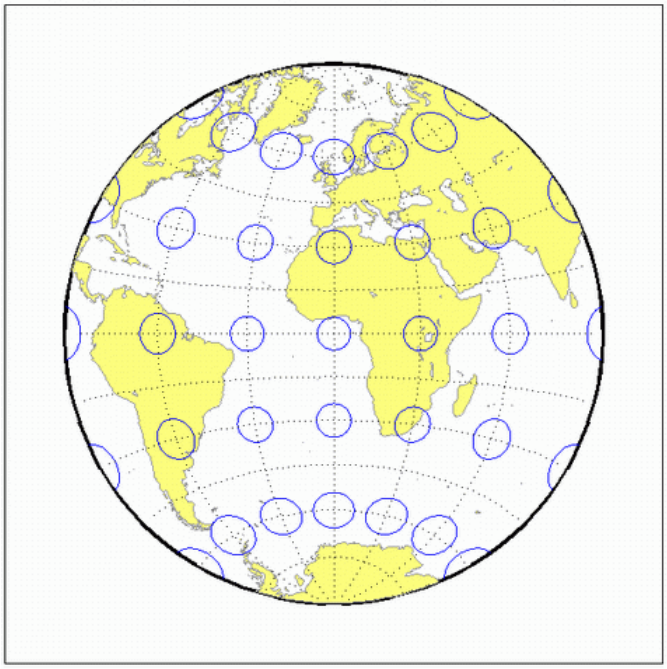
Remarks

- 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.
- This implementation of the Breusing harmonic mean projection is applicable only for coordinates that are referenced to a 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;
```



Introduced before R2006a

bries

Briesemeister Projection

Classi cation

Modified Azimuthal

Identi er

`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

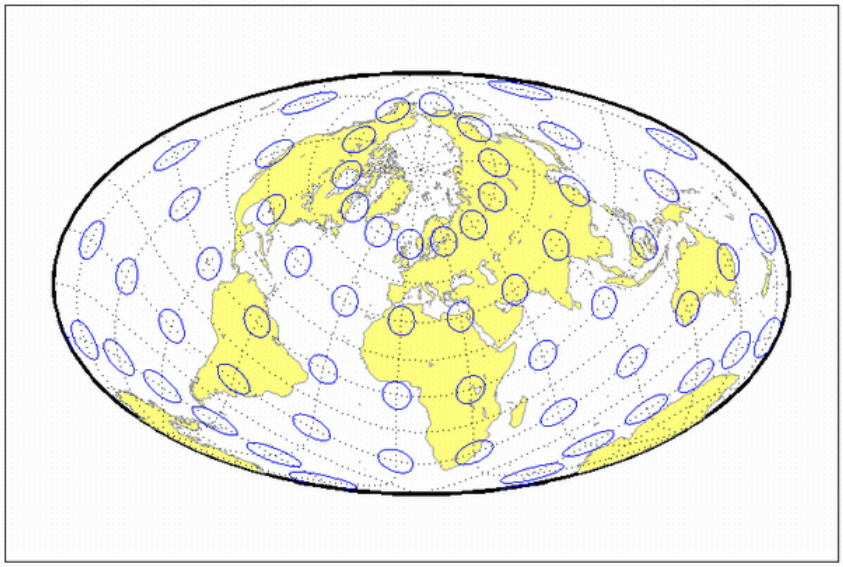
There is no standard parallel for this projection.

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.
- This implementation of the Briesemeister projection is applicable only for coordinates that are referenced to a sphere.

Example

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



Introduced before R2006a

cassini

Cassini Cylindrical Projection

Classi cation

Cylindrical

Identi er

`cassini`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90º from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the central meridian. Distortion of both shape and area are functions of distance from the central meridian. Scale is true along the central meridian and along any straight line perpendicular to the central meridian (i.e., it is equidistant).

Parallels

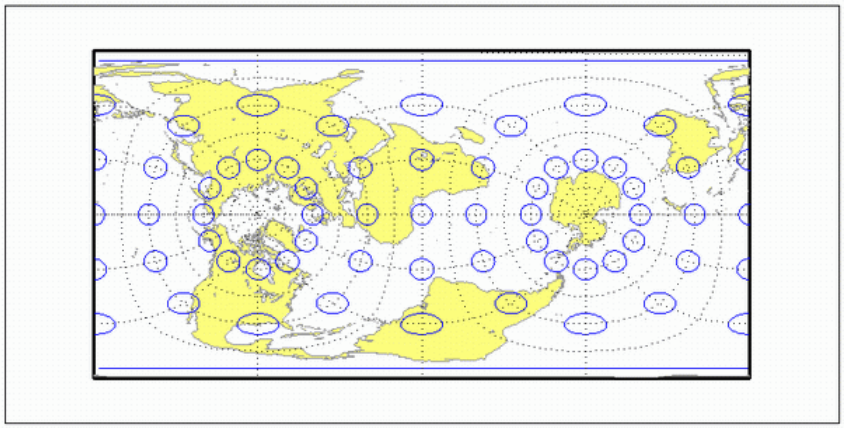
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel *of the base projection* is by definition fixed at 0º.

Remarks

- This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.
- This implementation of the Cassini cylindrical projection is applicable only for coordinates that are referenced to a sphere.

Example

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



See also

`cassinistd` on page 11-26

Introduced before R2006a

cassinistd

Cassini Cylindrical Projection — Standard

Identifier

`cassinistd`

Graticule

Central Meridian: Straight line (includes meridian opposite the central meridian in one continuous line).

Other Meridians: Straight lines if 90º from central meridian, complex curves concave toward the central meridian otherwise.

Parallels: Complex curves concave toward the nearest pole.

Poles: Points along the central meridian.

Symmetry: About any straight meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the central meridian. Distortion of both shape and area are functions of distance from the central meridian. Scale is true along the central meridian and along any straight line perpendicular to the central meridian (i.e., it is equidistant).

Parallels

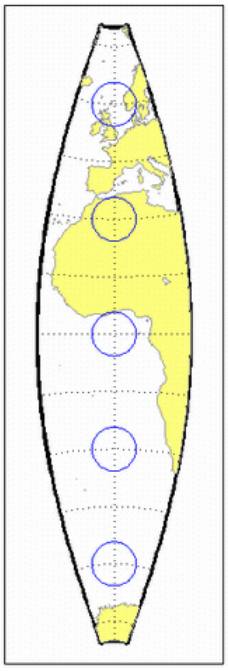
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel *of the base projection* is by definition fixed at 0º.

Remarks

- This projection is the transverse aspect of the Plate Carrée projection, developed by César François Cassini de Thury (1714–1784). It is still used for the topographic mapping of a few countries.
- `cassinistd` implements the Cassini projection directly on a sphere or reference ellipsoid, as opposed to using the equidistant cylindrical projection in transverse mode as in function `cassini` on page 11-24. 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.
- Mapping Toolbox uses a different implementation of the standard Cassini cylindrical projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

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



See also

`cassini` on page 11-24

Introduced before R2006a

ccylin

Central Cylindrical Projection

Classification

Cylindrical

Identifier

`ccylin`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles, more rapidly than that of the Mercator projection.

Poles: Cannot be shown.

Symmetry: About any meridian or the Equator.

Features

This is a perspective projection from the center of the Earth onto a cylinder tangent at the Equator. It is not equal-area, equidistant, or conformal. Scale is true along the Equator and constant between two parallels equidistant from the Equator. Scale becomes infinite at the poles. There is no distortion along the Equator, but it increases rapidly away from the Equator.

Parallels

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

Remarks

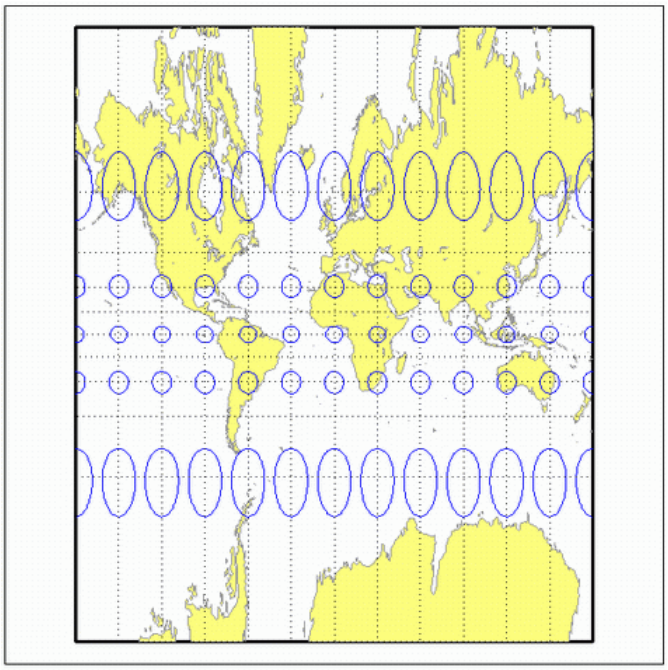
- The origin of this projection is unknown; it has little use beyond the educational aspects of its method of projection and as a comparison to the Mercator projection, which is not perspective. The transverse aspect of the central cylindrical is called the Wetch projection.
- This implementation of the central cylindrical projection is applicable only for coordinates that are referenced to a sphere.

Limitations

Data at latitudes greater than 75º is trimmed to prevent large values from dominating the display.

Example

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



Introduced before R2006a

collig

Collignon Projection

Classi cation

Pseudocylindrical

Identi er

```
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 15º51'N parallel, constant along any parallel, and *different* for any pair of parallels. Distortion is severe in many regions, and is only absent at 15º51'N on the central meridian. This projection is not conformal or equidistant.

Parallels

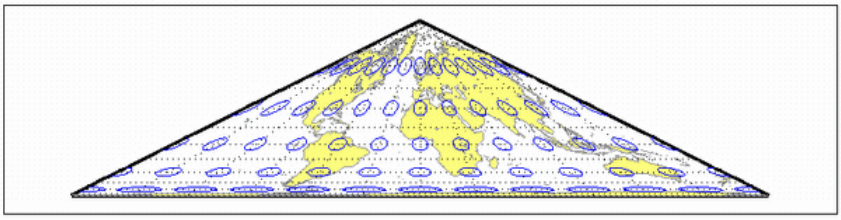
This projection has one standard parallel, which is by definition fixed at 15º51'.

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;
```



Introduced before R2006a

craster

Craster Parabolic Projection

Classification

Pseudocylindrical

Identifier

```
craster
```

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced parabolas intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes very gradually and is greatest near the Equator.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 36º46' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than the Sinusoidal projection. This projection is free of distortion only at the two points where the central meridian intersects the 36º46' parallels. This projection is not conformal or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 36º46'.

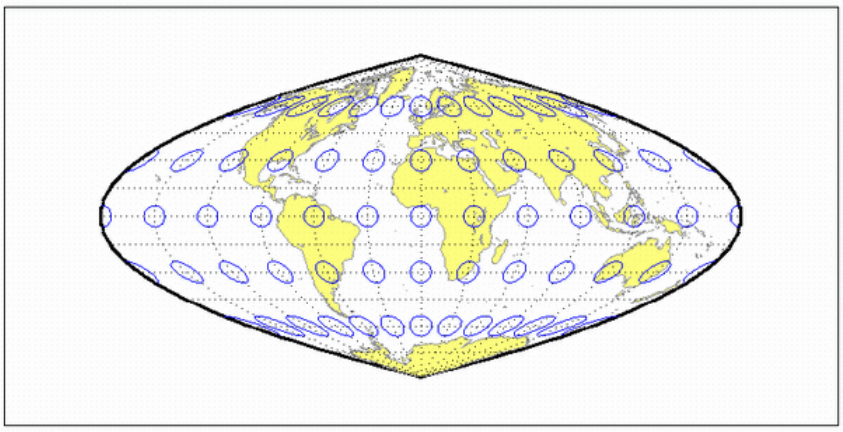
Remarks

This projection was developed by John Evelyn Edmund Craster in 1929; it was further developed by Charles H. Deetz and O.S. Adams in 1934. It was presented independently in 1934 by Putnins as his P_4 projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('craster', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eckert1

Eckert I Projection

Classi cation

Pseudocylindrical

Identi er

`eckert1`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced straight converging lines broken at the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 47º10' parallels and is constant along any parallel, between any pair of parallels equidistant from the Equator, and along any given meridian. It is not free of distortion at any point, and the break at the Equator introduces excessive distortion there; regardless of the appearance here, the Tissot indicatrices are of indeterminate shape along the Equator. This novelty projection is not equal-area or conformal.

Parallels

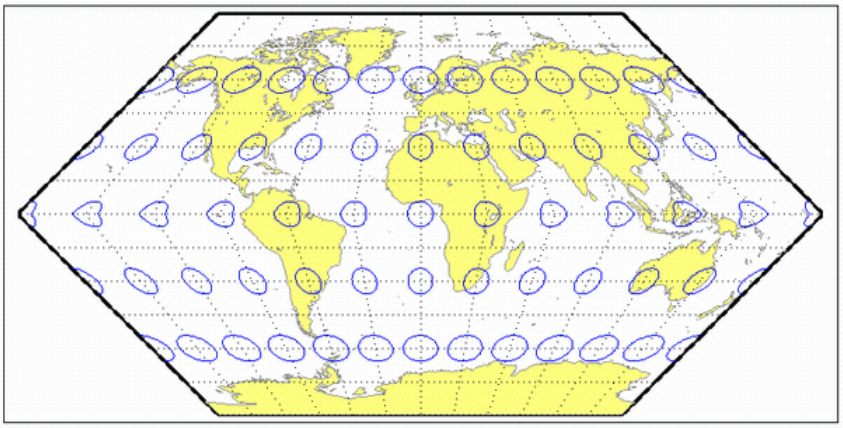
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 47º10'.

Remarks

- This projection was presented by Max Eckert in 1906.
- This implementation of the Eckert I projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Eckert I projection using the ESRI authority code 54015. For example: `projcrs(54015,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert1', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eckert2

Eckert ll Projection

Classi cation

Pseudocylindrical

Identi er

`eckert2`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced straight converging lines broken at the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 55º10' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is not free of distortion at any point except at 55º10'N and S along the central meridian; the break at the Equator introduces excessive distortion there. Regardless of the appearance here, the Tissot indicatrices are of indeterminate shape along the Equator. This novelty projection is not conformal or equidistant.

Parallels

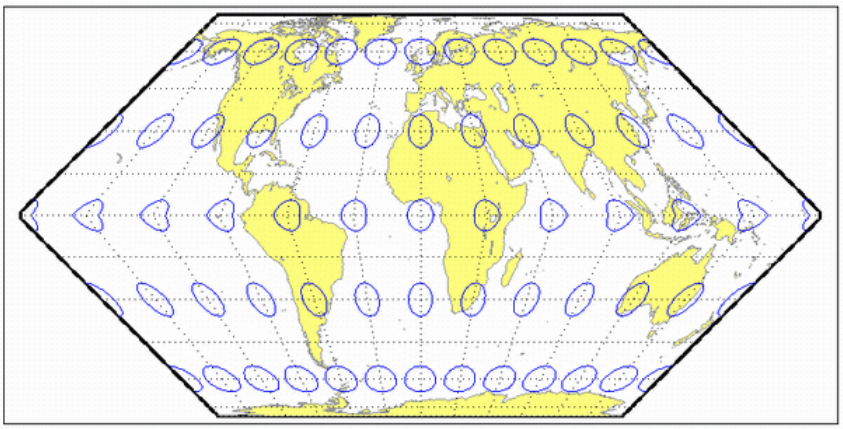
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 55º10'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert2', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eckert3

Eckert lll Projection

Classi cation

Pseudocylindrical

Identi er

`eckert3`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180º east and west of the central meridian, are semicircles.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 35º58' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. No point is free of all scale distortion, but the Equator is free of angular distortion. This projection is not equal-area, conformal, or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 35º58'.

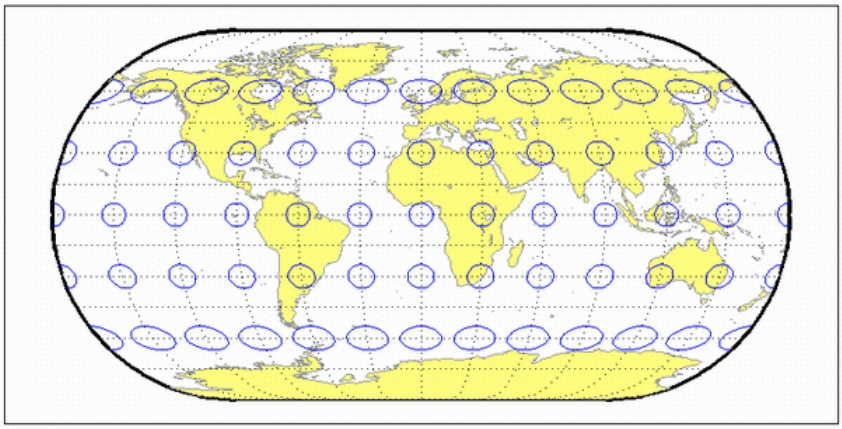
Remarks

- This projection was presented by Max Eckert in 1906.
- This implementation of the Eckert III projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Eckert III projection using the ESRI authority code 54013. For example: `projcrs(54013,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert3', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eckert4

Eckert IV Projection

Classification

Pseudocylindrical

Identifier

`eckert4`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced semiellipses concave toward the central meridian. The outer meridians, 180º east and west of the central meridian, are semicircles.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 40º30' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 40º30' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

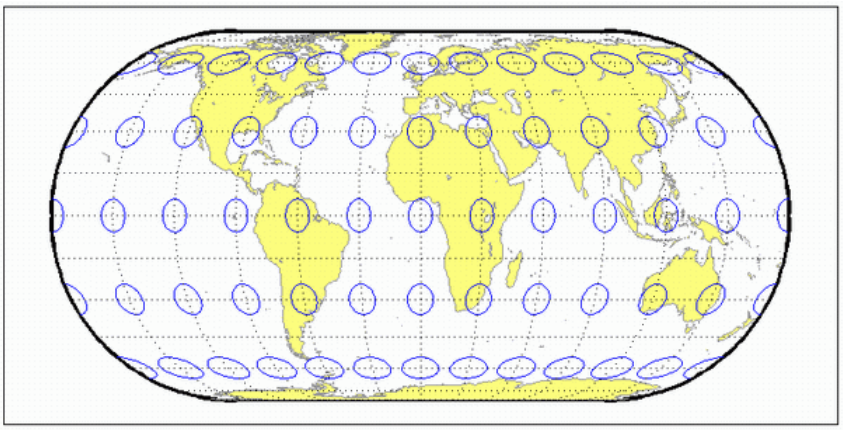
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 40º30'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert4', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eckert5

Eckert V Projection

Classification

Pseudocylindrical

Identifier

`eckert5`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This projection is an arithmetic average of the *x* and *y* coordinates of the Sinusoidal and Plate Carrée projections. Scale is true along latitudes 37º55'N and S, and is constant along any parallel and between any pair of parallels equidistant from the Equator. There is no point free of all distortion, but the Equator is free of angular distortion. This projection is not equal-area, conformal, or equidistant.

Parallels

This projection has one standard parallel, which is by definition fixed at 0º.

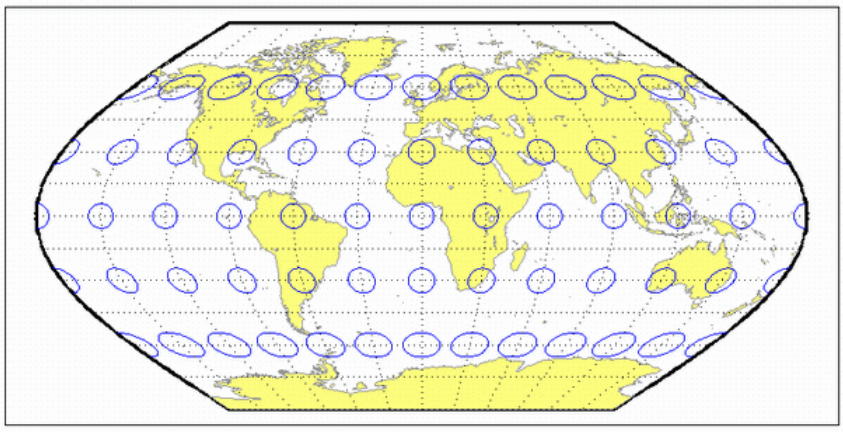
Remarks

- This projection was presented by Max Eckert in 1906.
- This implementation of the Eckert V projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Eckert V projection using the ESRI authority code 54011. For example: `projcrs(54011,'Authority','ESRI')`.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert5', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eckert6

Eckert VI Projection

Classi cation

Pseudocylindrical

Identi er

`eckert6`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 49º16' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 49º16' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

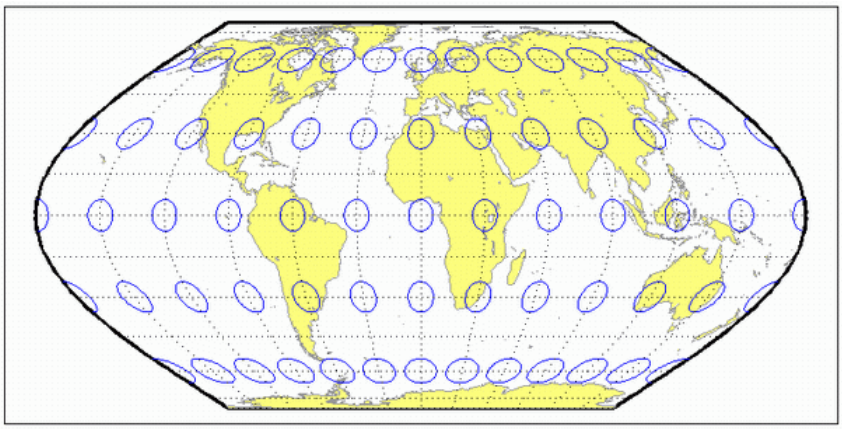
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 49º16'.

Remarks

This projection was presented by Max Eckert in 1906.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('eckert6', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eqacylin

Equal-Area Cylindrical Projection

Classification

Cylindrical

Identifier

`eqacylin`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the standard parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

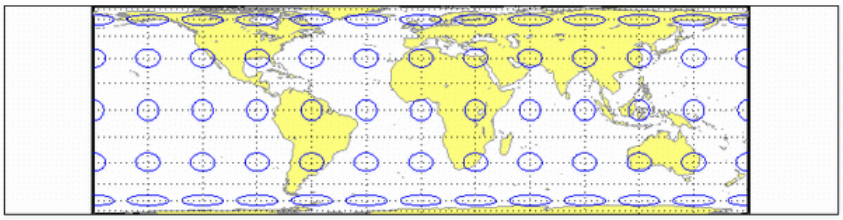
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude may be chosen; the default is arbitrarily set to 0º (the Lambert variation).

Remarks

- This projection was proposed by Johann Heinrich Lambert (1772), a prolific cartographer who proposed seven different important projections. The form of this projection tangent at the Equator is often called the Lambert Equal-Area Cylindrical projection. That and other special forms of this projection are included separately in this guide, including the Gall Orthographic, the Behrmann Cylindrical, the Balthasart Cylindrical, and the Trystan Edwards Cylindrical projections.
- Mapping Toolbox uses a different implementation of the equal-area cylindrical projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

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



Introduced before R2006a

eqdazim

Equidistant Azimuthal Projection

Classification

Azimuthal

Identifier

`eqdazim`

Graticule

The graticule described is for the polar aspect.

Meridians: Equally spaced straight lines intersecting at a central pole. The angles between them are the true angles.

Parallels: Equally spaced circles, centered on the central pole. The entire Earth may be shown.

Poles: Central pole is a point. The opposite pole is a bounding circle with a radius twice that of the Equator.

Symmetry: About any meridian.

Features

This is an equidistant projection. It is neither equal-area nor conformal. In the polar aspect, scale is true along any meridian. The projection is distortion free only at the center point. Distortion is moderate for the inner hemisphere, but it becomes extreme in the outer hemisphere.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

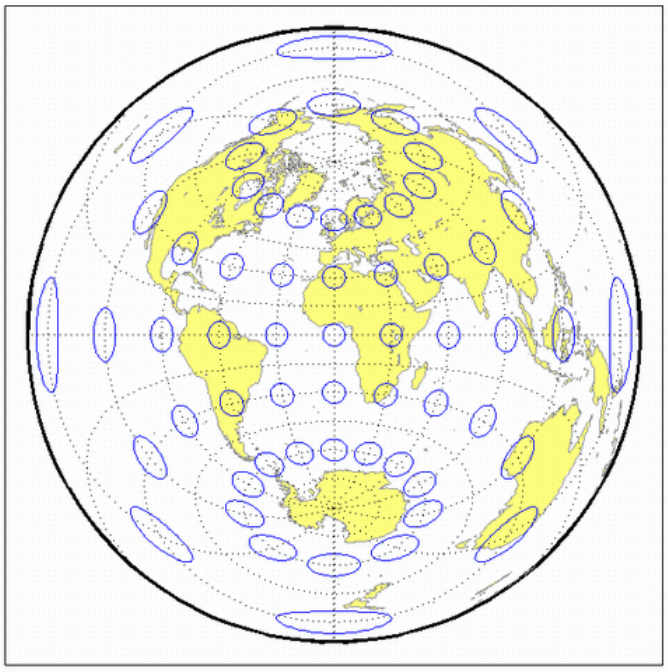
- This projection may have been first used by the ancient Egyptians for star charts. Several cartographers used it during the sixteenth century, including Guillaume Postel, who used it in 1581. Other names for this projection include Postel and Zenithal Equidistant.
- Mapping Toolbox uses a different implementation of the equidistant azimuthal projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.
- The implementation of the equidistant azimuthal projection for displaying coordinates on map axes is applicable only for coordinates that are referenced to a sphere. The implementation of the equidistant azimuthal projection for projecting coordinates using the `projfwd` or `projinv` function is applicable for coordinates referenced to either a sphere or an ellipsoid.

Limitations

This projection is available only on the sphere.

Example

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



Introduced before R2006a

eqdconic

Equidistant Conic Projection

Classification

Conic

Identifier

`eqdconic`

Graticule

Meridians: Equally spaced straight lines converging to a common point, usually beyond the pole. The angles between the meridians are less than the true angles.

Parallels: Equally spaced concentric circular arcs centered on the point of meridional convergence.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About any meridian.

Features

Scale is true along each meridian and the one or two selected standard parallels. Scale is constant along any parallel. This projection is free of distortion along the two standard parallels. Distortion is constant along any other parallel. This projection provides a compromise in distortion between conformal and equal-area conic projections, of which it is neither.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and an Equidistant Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then an Equidistant Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is so chosen, the cone becomes a cylinder and a Plate Carrée projection results. If two parallels equidistant from the Equator are chosen as the standard parallels, an Equidistant Cylindrical projection results. The default parallels are [15 75].

Remarks

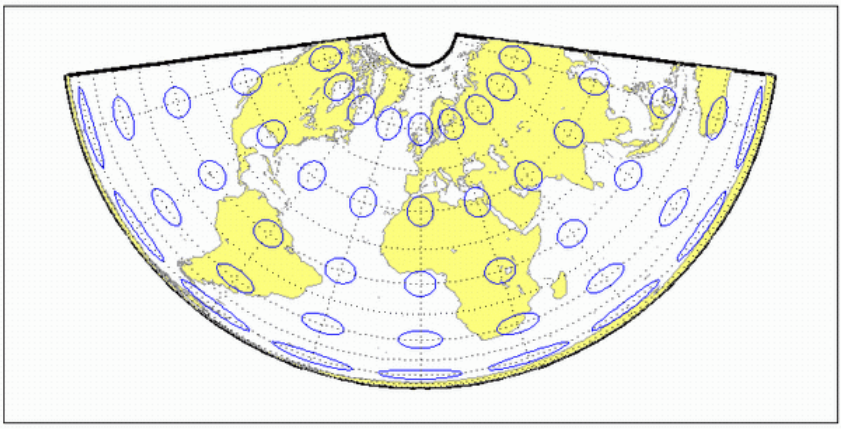
In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.

Limitations

Longitude data greater than 135º 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 on page 11-52

Introduced before R2006a

eqdconicstd

Equidistant Conic Projection — Standard

Identifier

`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` on page 11-50 for an alternative implementation based on rotating the rectifying sphere.

Scale is true along each meridian and the one or two selected standard parallels. Scale is constant along any parallel. This projection is free of distortion along the two standard parallels. Distortion is constant along any other parallel. This projection provides a compromise in distortion between conformal and equal-area conic projections, of which it is neither.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and an Equidistant Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then an Equidistant Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator is so chosen, the cone becomes a cylinder and a Plate Carrée projection results. If two parallels equidistant from the Equator are chosen as the standard parallels, an Equidistant Cylindrical projection results. The default parallels are [15 75].

Remarks

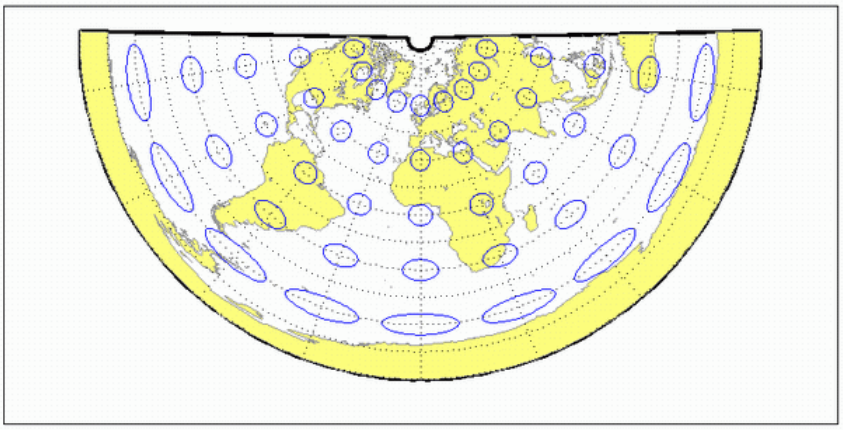
- In a rudimentary form, this projection dates back to Claudius Ptolemy, about A.D. 100. Improvements were developed by Johannes Ruysch in 1508, Gerardus Mercator in the late 16th century, and Nicolas de l'Isle in 1745. It is also known as the Simple Conic or Conic projection.
- Mapping Toolbox uses a different implementation of the standard equidistant conic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

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

Example

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



See Also

eqdconic on page 11-50

Introduced before R2006a

eqdcylin

Equidistant Cylindrical Projection

Classification

Cylindrical

Identifier

`eqdcylin`

Graticule

Meridians: Equally spaced straight parallel lines more than half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder secant at the standard parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the standard parallels and is constant along any parallel and along the parallel of opposite sign.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude can be chosen; the default is arbitrarily set to 30º.

Remarks

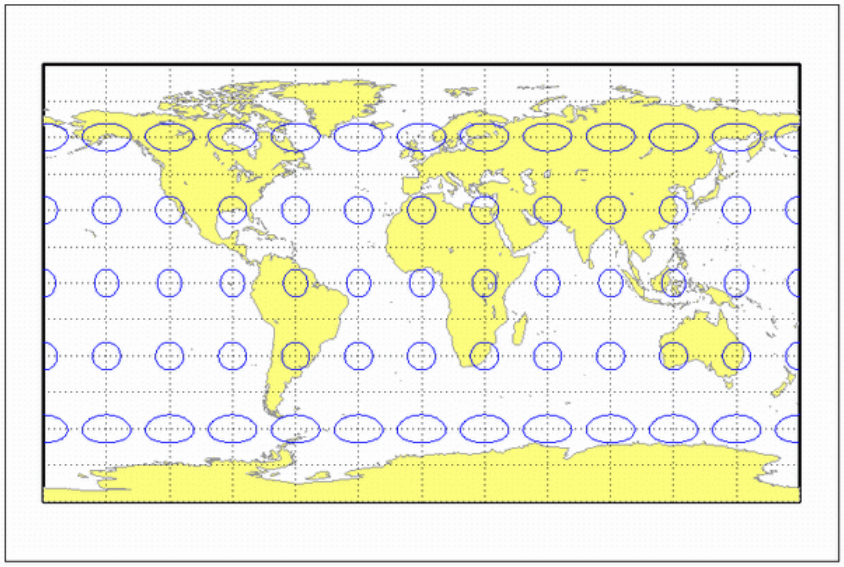
- This projection was first used by Marinus of Tyre about A.D. 100. Special forms of this projection are the Plate Carrée, with a standard parallel at 0º, and the Gall Isographic, with standard parallels at 45ºN and S. Other names for this projection include Equirectangular, Rectangular, Projection of Marinus, *La Carte Parallélogrammatique*, and *Die Rechteckige Plattkarte*.
- By default, the standard parallels are at +/- 30 degrees in geodetic latitude.
- When projecting a sphere, the origin vector is used to specify a triaxial rigid-body rotation.
- When projecting an ellipsoid:
  - The origin longitude (2nd element of the origin vector) determines which meridian maps to the line *x* == false easting

- The origin latitude (1st element of the origin vector) is used to shift the natural origin off the equator via a constant *y*-offset, in addition to any false northing that may be specified.
- The grid convergence is fixed at 0, even if the 3rd element of the origin vector is nonzero.

- Mapping Toolbox uses a different implementation of the equidistant cylindrical projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

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



Introduced before R2006a

fournier

Fournier Projection

Classification

Pseudocylindrical

Identifier

`fournier`

Graticule

Meridians: Equally spaced elliptical curves converging at the poles.

Parallels: Straight lines.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is equal-area. Scale is constant along any parallel or pair of parallels equidistant from the Equator. This projection is neither equidistant nor conformal.

Parallels

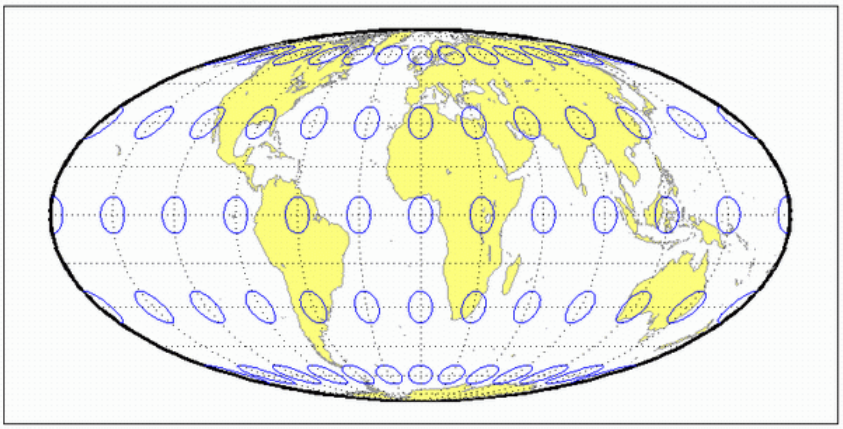
There is no standard parallel for this projection.

Remarks

This projection was first described in 1643 by Georges Fournier. This is actually his second projection, the Fournier II.

Example

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



Introduced before R2006a

giso

Gall Isographic Projection

Classification

Cylindrical

Identifier

`giso`

Graticule

Meridians: Equally spaced straight parallel lines more than half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having wider spacing than the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder secant at the 45º parallels. Distortion of both shape and area increase with distance from the standard parallels. Scale is true along all meridians (i.e., it is equidistant) and the two standard parallels, and is constant along any parallel and along the parallel of opposite sign.

Parallels

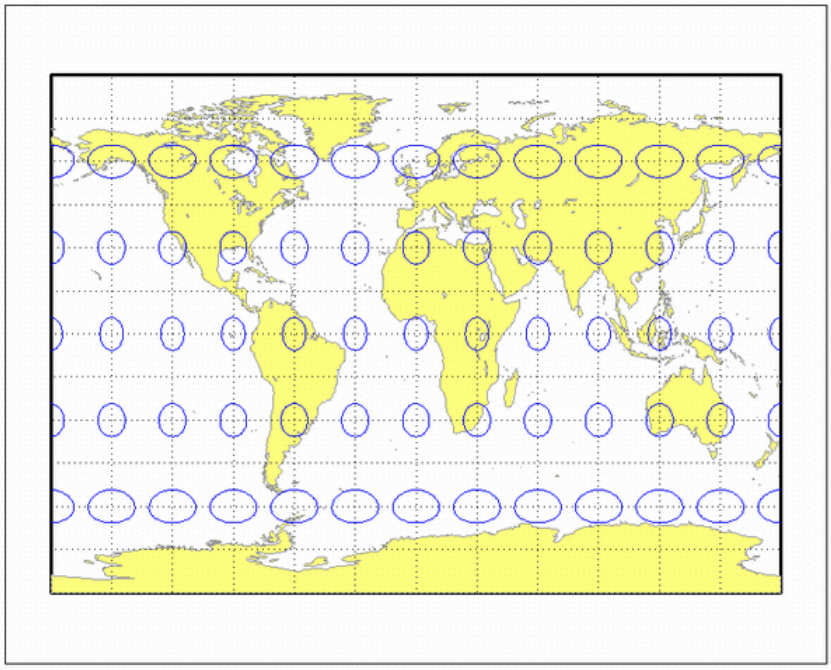
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

Remarks

- This projection is a specific case of the Equidistant Cylindrical projection, with standard parallels at 45ºN and S.
- On the sphere, this projection can have an arbitrary, oblique aspect, as controlled by the `Origin` property of the map axes. On the ellipsoid, only the equatorial aspect is supported.

Example

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



Introduced before R2006a

gortho

Gall Orthographic Projection

Classification

Cylindrical

Identifier

`gortho`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 45º parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

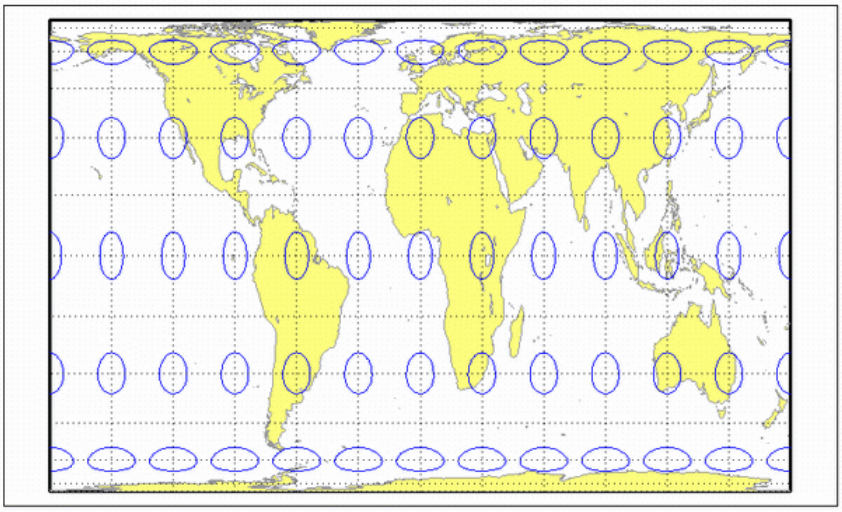
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

Remarks

This projection is named for James Gall, who originated it in 1855 and is a special form of the Equal-Area Cylindrical projection secant at 45ºN and S. This projection is also known as the Peters projection.

Example

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



Introduced before R2006a

gstereo

Gall Stereographic Projection

Classification

Cylindrical

Identifier

`gstereo`

Graticule

Meridians: Equally spaced straight parallel lines 0.77 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a perspective projection from a point on the Equator opposite a given meridian onto a cylinder secant at the 45º parallels. It is not equal-area, equidistant, or conformal. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. There is no distortion along the standard parallels, but it increases moderately away from these parallels, becoming severe at the poles.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 45º.

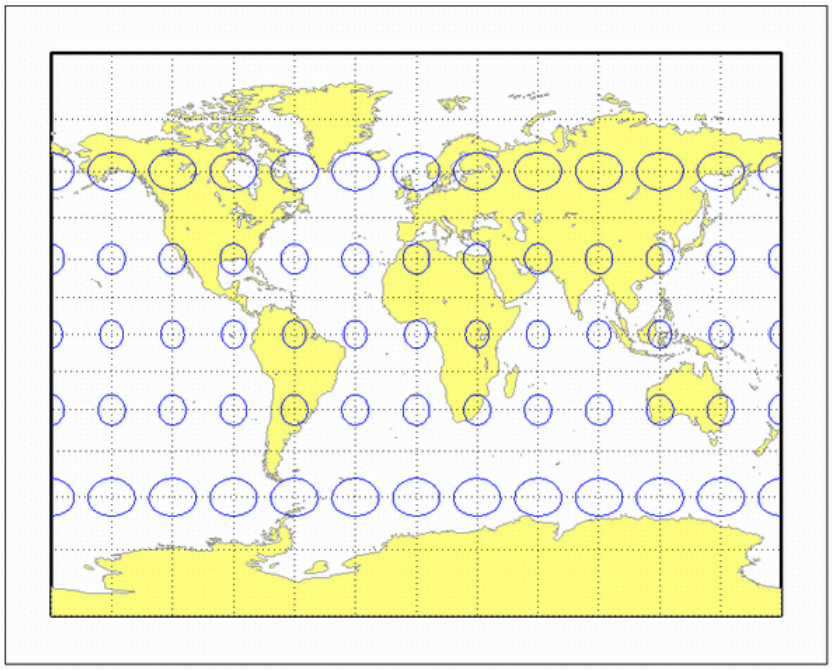
Remarks

- This projection was presented by James Gall in 1855. It is also known simply as the Gall projection. It is a special form of the Braun Perspective Cylindrical projection secant at 45ºN and S.
- This implementation of the Gall stereographic projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Gall stereographic projection using the ESRI authority code 54016. For example: `projcrs(54016,'Authority','ESRI')`.

- Mapping Toolbox uses a different implementation of the Gall stereographic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

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



Introduced before R2006a

globe

Frame for 3-D map display

Classification

Spherical

Identifier

`globe`

Graticule

This map display is based on a coordinate transformation and is not a true map projection. Meridians, parallels, and displayed map data appear in a 3-D view that depends on the view and camera settings of the map axes. Change the view interactively or by using the `view` function. Change the camera settings using the `camposm`, `camtargm`, and `camupm` functions.

Features

In the 3-D sense, `globe` is true in scale, equal-area, conformal, minimum error, and equidistant everywhere.

While `globe` has none of the distortions inherent in planar projections, it cannot be displayed without distortion or in its entirety. 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.

Parallels

The globe requires no standard parallels.

Remarks

The `globe` display allows you to visualize terrain relief or other data for an entire planet viewed from space. Its underlying transformation maps latitude, longitude, and elevation to a 3-D Cartesian frame. The `globe` display is different from other transformations because it can render relative relief of elevations above, below, or on a sphere.

When displayed, the globe looks like an orthographic azimuthal projection, provided that the `Projection` property of the map axes is set to `'orthographic'`.

Examples

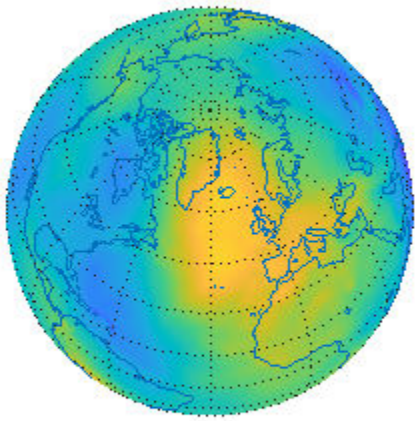
Display Geoid Heights on Globe

Display geoid heights from the EGM96 geoid model over a 3-D globe. First, get geoid heights and a geographic postings reference object. Load coastline latitude and longitude data.

```
[N,R] = egm96geoid;
load coastlines
```

Create a frame for the 3-D globe display using `axesm`. Set the line of sight for the globe using `view`. Turn off the axes background using `axis off`. Then, display the geoid heights and coastline data.

```
axesm('globe','Grid','on')
view(60,60)
axis off
meshm(N,R)
plotm(coastlat,coastlon)
```



Display Polygon on Globe

Display a polygon on a globe by converting the polygon to a data grid.

Create a sample polygon that contains a hole and rests on the surface of the globe. To do this, generate the vertices of its external and internal boundaries using the `outlinegeoquad` function. Specify the geographic limits as the first two arguments, and the vertex spacing in degrees as the

next two arguments. Reverse the order of the internal boundary vertices using the `flip` function, so they are in a counterclockwise order.

```
[latE,lonE] = outlinegeoquad([-35 35],[-30 30],0.25,0.25);
[latI,lonI] = outlinegeoquad([-15 15],[-15 15],0.25,0.25);
latI = flip(latI);
lonI = flip(lonI);
```

Combine the vertices into a single list by separating the boundaries with `NaN` values.

```
lat = [latE NaN latI];
lon = [lonE NaN lonI];
```

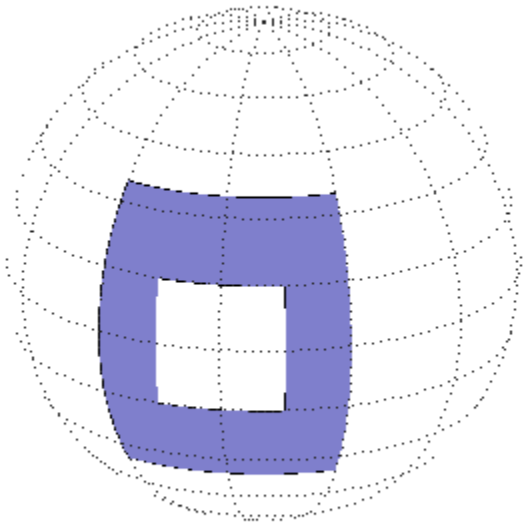
The vectors `lat` and `lon` represent the boundaries of a polygon that contain a hole. Display the boundaries on the globe as a filled polygon by converting the polygon to a data grid.

To do this, create a geographic cells reference object for the globe and a grid of ones. Replace elements of the grid with the polygon data using the `vec2mtx` function. The new grid contains 0s to indicate the inside region of the polygon, 1s to indicate the boundaries, and 2s to indicate the outside region of the polygon.

```
R = georefcells([-90 90],[-180 180],0.25,0.25);
V = ones(R.RasterSize);
[V,R] = vec2mtx(lat,lon,V,R,'filled');
```

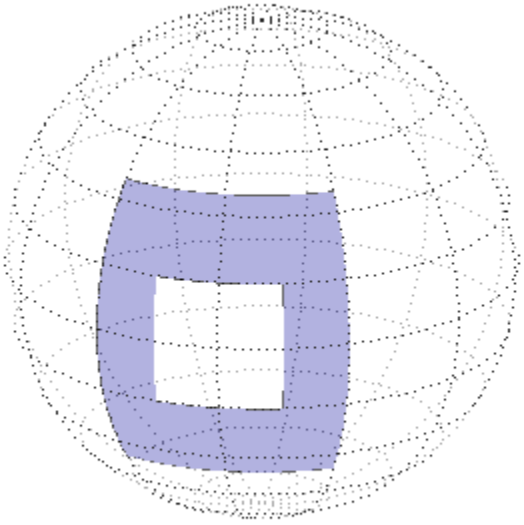
Create a globe using the `axesm` function. Display the data grid as an image using the `geoshow` function. Adjust the colormap so the inside region of the polygon is purple and the outside region is white. Change the camera line of sight using the `view` function, so the polygon is displayed on the near side of the globe.

```
axesm('globe','Grid','on')
geoshow(V,R,'DisplayType','texturemap')
colormap([0.5 0.5 0.8; 0 0 0; 1 1 1])
axis off
view(100,20)
```

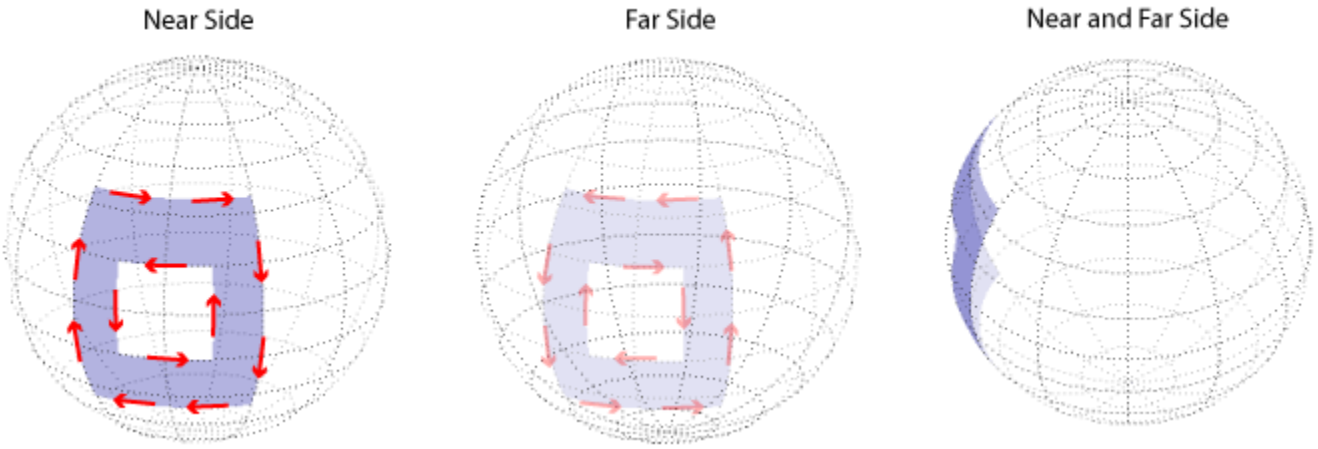


The appearance of polygons on the globe is dependent on the camera line of sight and the globe transparency. For example, make the globe slightly transparent using the `alpha` function.

```
alpha(0.6)
```



When you view the polygon from the near side of the globe, the external boundary vertices appear in a clockwise order. When you view the polygon from the far side of the globe, the external boundary vertices appear in a counterclockwise order. When you rotate the globe so the polygon appears on both the near side and far side, then the polygon appears to intersect itself.



See Also

`axesm` | `geoshow` | `outlinegeoquad` | `vec2mtx`

Topics
“Create and Display Polygons” on page 2-12
“The Globe Display Compared with the Orthographic Projection” on page 5-39

Introduced before R2006a

gnomonic

Gnomonic Projection

Classification

Azimuthal

Identifier

`gnomonic`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing increases rapidly away from this pole. The Equator and the opposite hemisphere cannot be shown

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection from the center of the globe on a plane tangent at the center point, which is a pole in the common polar aspect, but can be any point. Less than one hemisphere can be shown with this projection, regardless of its center point. The significant property of this projection is that all great circles are straight lines. This is useful in navigation, as a great circle is the shortest path between two points on the globe. Only the center point enjoys true scale and zero distortion. This projection is neither conformal nor equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

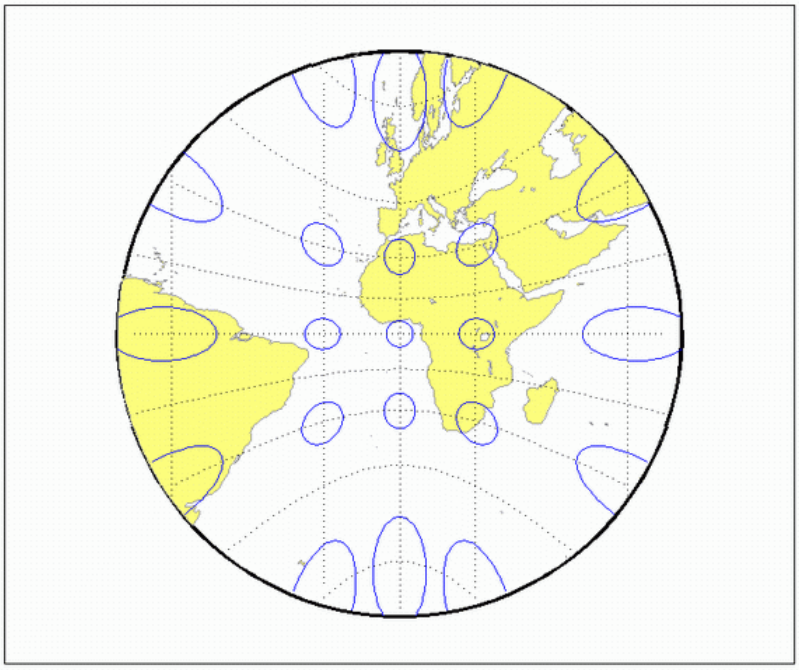
- This projection may have been first developed by Thales around 580 B.C. Its name is derived from the gnomon, the face of a sundial, since the meridians radiate like hour markings. This projection is also known as a gnomic or central projection.
- This implementation of the gnomonic projection is applicable only for coordinates that are referenced to a sphere.
- Mapping Toolbox uses a different implementation of the gnomonic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data greater than 65º distant from the center point is trimmed.

Example

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



Introduced before R2006a

goode

Goode Homolosine Projection

Classification

Pseudocylindrical

Identifier

goode

Graticule

Central Meridian: Straight line 0.44 as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves between the 40º44'11.8'' parallels and elliptical arcs elsewhere, all concave toward the central meridian. The result is a slight, visible bend in the meridians at 40º44'11.8'' N and S.

Parallels: Straight parallel lines, perpendicular to the central meridian. Equally spaced between the 40º44'11.8'' parallels, with gradually decreasing spacing outside these parallels.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along all parallels and the central meridian between 40º44'11.8'' N and S, and is constant along any parallel and between any pair of parallels equidistant from the Equator for all latitudes. Its distortion is identical to that of the Sinusoidal projection between 40º44'11.8'' N and S, and to that of the Mollweide projection elsewhere. This projection is not conformal or equidistant.

Parallels

This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

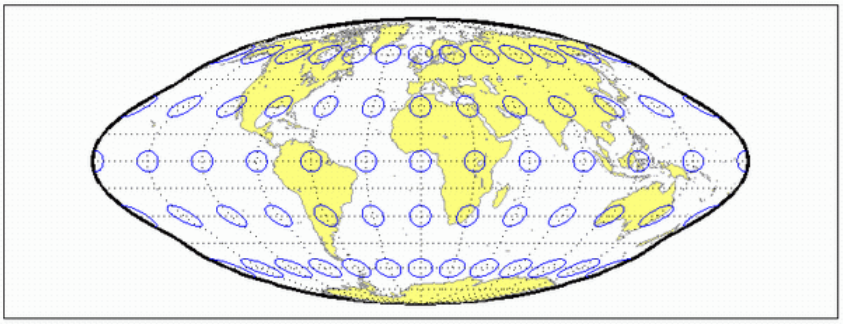
This projection was developed by J. Paul Goode in 1916. It is sometimes called simply the Homolosine projection, and it is usually used in an interrupted form. It is a merging of the Sinusoidal and Mollweide projections.

Limitations

This projection is available in an uninterrupted form only.

Example

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



Introduced before R2006a

hammer

Hammer Projection

Classification

Modified Azimuthal

Identifier

`hammer`

Graticule

Meridians: Central meridian is a straight line half the length of the Equator. Other meridians are complex curves, equally spaced along the Equator, and concave toward the central meridian.

Parallels: Equator is straight. Other parallels are complex curves, equally spaced along the central meridian, and concave toward the nearest pole.

Poles: Points.

Symmetry: About the Equator and central meridian.

Features

This projection is equal-area. The only point free of distortion is the center point. Distortion of shape is moderate throughout. This projection has less angular distortion on the outer meridians near the poles than pseudoazimuthal projections

Parallels

There is no standard parallel for this projection.

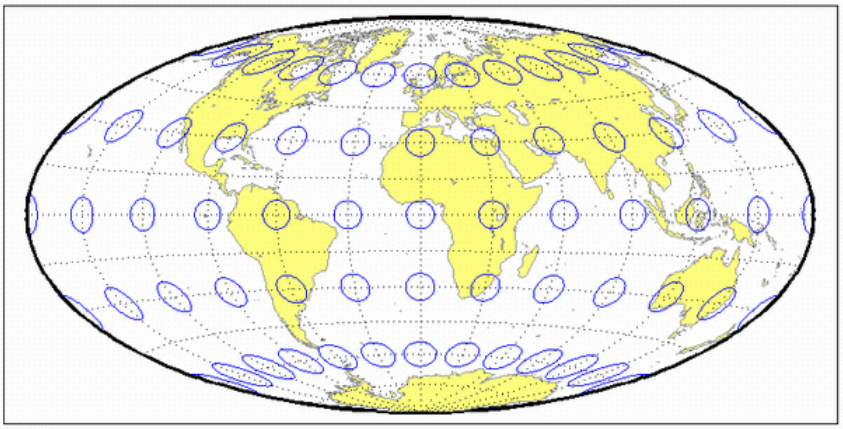
Remarks

- This projection was presented by H. H. Ernst von Hammer in 1892. It is a modification of the Lambert Azimuthal Equal Area projection. Inspired by Aitoff projection, it is also known as the Hammer-Aitoff. It in turn inspired the Briesemeister, a modified oblique Hammer projection. John Bartholomew's Nordic projection is an oblique Hammer centered on 45 degrees north and the Greenwich meridian. The Hammer projection is used in whole-world maps and astronomical maps in galactic coordinates.
- This implementation of the Hammer projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('hammer', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

hatano

Hatano Asymmetrical Equal-Area Projection

Classification

Pseudocylindrical

Identifier

`hatano`

Graticule

Central Meridian: Straight line 0.48 as long as the Equator.

Other Meridians: Equally spaced elliptical arcs concave toward the central meridian. The eccentricity of each ellipse changes at the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is not symmetrical about the Equator.

Poles: The North Pole is a line two-thirds the length of the Equator; the South Pole is a line three-fourths the length of the Equator.

Symmetry: About the central meridian but *not* the Equator.

Features

This is an equal-area projection. Scale is true along 40º42'N and 38º27'S, and is constant along any parallel but generally *not* between pairs of parallels equidistant from the Equator. It is free of distortion only along the central meridian at 40º42'N and 38º27'S. This projection is not conformal or equidistant.

Parallels

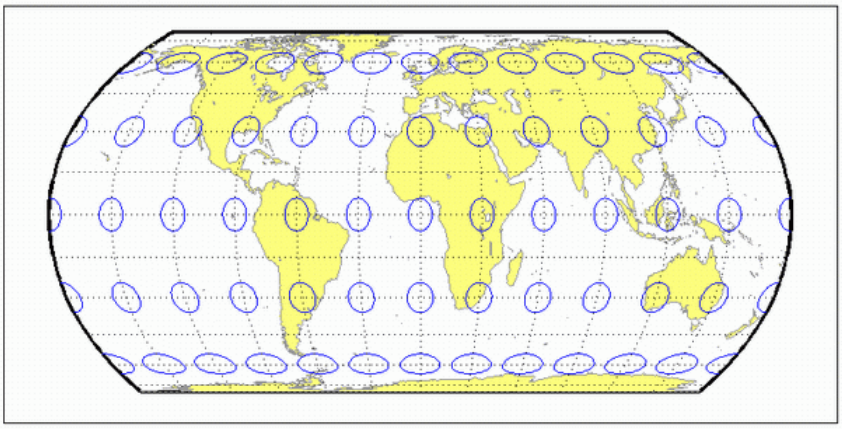
Because of the asymmetrical nature of this projection, two standard parallels must be specified. The standard parallels are by definition fixed at 40º42'N and 38º27'S.

Remarks

This projection was presented by Masataka Hatano in 1972.

Example

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



Introduced before R2006a

kavrsky5

Kavraisky V Projection

Classification

Pseudocylindrical

Identifier

`kavrsky5`

Graticule

Meridians: Complex curves converging at the poles. A sine function is used for *y*, but the meridians are not sine curves.

Parallels: Unequally spaced straight lines.

Poles: Points.

Symmetry: About the Equator and the central meridian.

Features

This is an equal-area projection. Scale is true along the fixed standard parallels at 35º, and 0.9 true along the Equator. This projection is neither conformal nor equidistant.

Parallels

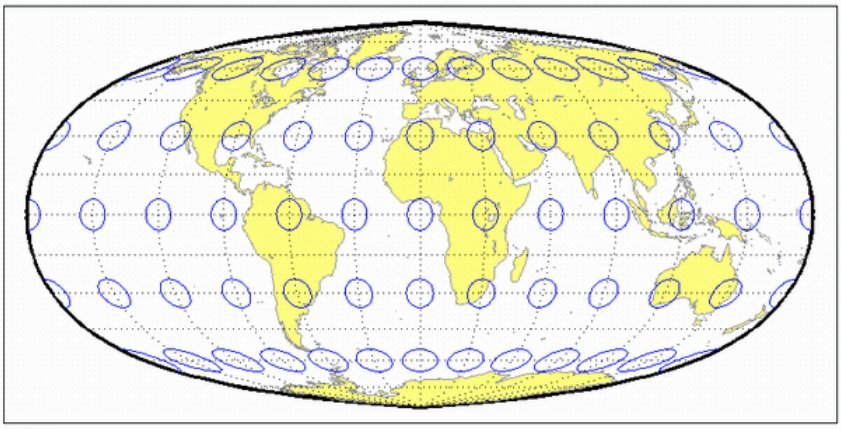
The fixed standard parallels are at 35º.

Remarks

This projection was described by V. V. Kavraisky in 1933.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('kavrsky5', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

kavrsky6

Kavraisky VI Projection

Classi cation

Pseudocylindrical

Identi er

`kavrsky6`

Graticule

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

Meridians: Sine curves (60º segments).

Parallels: Unequally spaced straight lines.

Poles: Straight lines half the length of the Equator.

Symmetry: About the Equator and the central meridian.

Features

This is an equal-area projection. Scale is constant along any parallel or pair of equidistant parallels. This projection is neither conformal nor equidistant.

Parallels

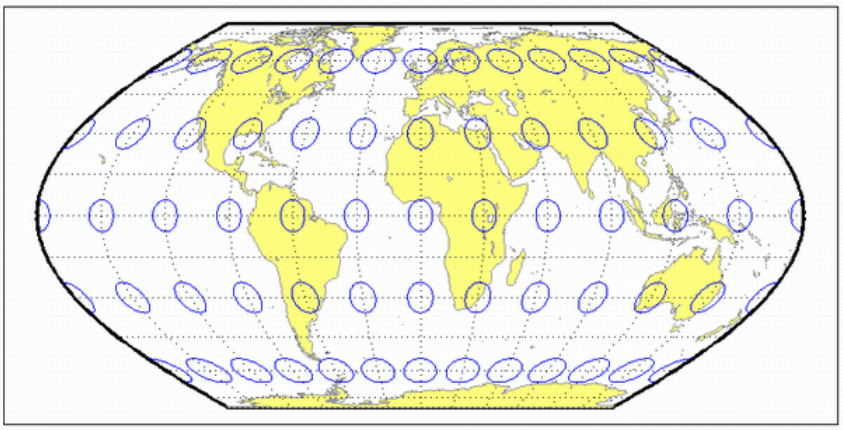
There are no standard parallels for this projection.

Remarks

This projection was described by V. V. Kavraisky in 1936. It is also called the Wagner I, for Karlheinz Wagner, who described it in 1932.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('kavrsky6', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

eqaazim

Lambert Azimuthal Equal-Area Projection

Classification

Azimuthal

Identifier

`eqaazim`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. The entire Earth can be shown. Spacing decreases away from the central pole.

Pole: The central pole is a point; the other pole is a bounding circle with 1.41 the radius of the Equator.

Symmetry: About any meridian.

Features

This nonperspective projection is equal-area. Only the center point is free of distortion, but distortion is moderate within 90º of this point. Scale is true only at the center point, increasing tangentially and decreasing radially with distance from the center point. This projection is neither conformal nor equidistant.

Parallels

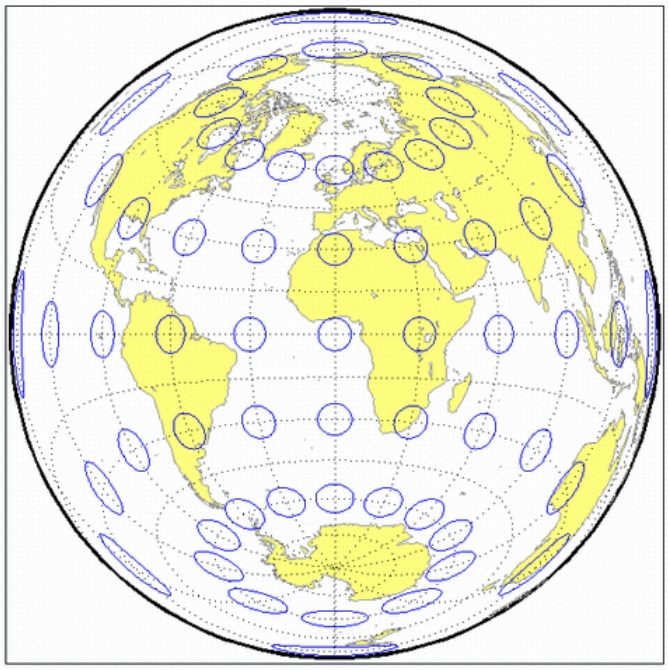
There are no standard parallels for azimuthal projections.

Remarks

- This projection was presented by Johann Heinrich Lambert in 1772. It is also known as the Zenithal Equal-Area and the Zenithal Equivalent projection, and the Lorgna projection in its polar aspect.
- Mapping Toolbox uses a different implementation of the Lambert azimuthal equal-area projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

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



lambert

Lambert Conformal Conic Projection

Classi cation

Conic

Identi er

`lambert`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.

Poles: The pole nearest a standard parallel is a point, the other cannot be shown.

Symmetry: About any meridian.

Features

Scale is true along the one or two selected standard parallels. Scale is constant along any parallel and is the same in every direction at any point. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is conformal everywhere but the poles; it is neither equal-area nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Stereographic Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Conformal Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator or two parallels equidistant from the Equator are chosen as the standard parallels, the cone becomes a cylinder, and a Mercator projection results. The default parallels are [15 75].

Remarks

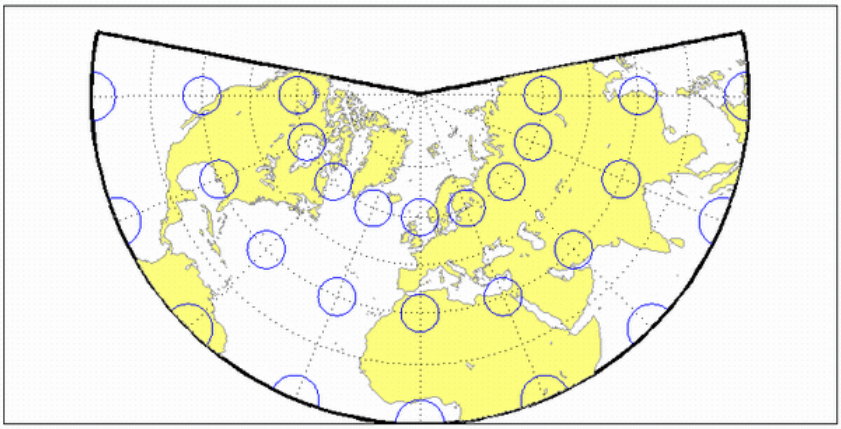
This projection was presented by Johann Heinrich Lambert in 1772 and is also known as a Conical Orthomorphic projection.

Limitations

Longitude data greater than 135º 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` on page 11-86

lambertstd

Lambert Conformal Conic Projection — Standard

Classification

Conic

Identifier

`lambertstd`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles. The angles between the meridians are less than the true angles.

Parallels: Unequally spaced concentric circular arcs centered on the pole of convergence. Spacing of parallels increases away from the central latitudes.

Poles: The pole nearest a standard parallel is a point, the other cannot be shown.

Symmetry: About any meridian.

Features

`lambertstd` implements the Lambert Conformal Conic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `lambert` on page 11-84 for an alternative implementation based on rotating the authalic sphere.

Scale is true along the one or two selected standard parallels. Scale is constant along any parallel and is the same in every direction at any point. This projection is free of distortion along the standard parallels. Distortion is constant along any other parallel. This projection is conformal everywhere but the poles; it is neither equal-area nor equidistant.

Parallels

The cone of projection has interesting limiting forms. If a pole is selected as a single standard parallel, the cone is a plane, and a Stereographic Azimuthal projection results. If two parallels are chosen, not symmetric about the Equator, then a Lambert Conformal Conic projection results. If a pole is selected as one of the standard parallels, then the projected pole is a point, otherwise the projected pole is an arc. If the Equator or two parallels equidistant from the Equator are chosen as the standard parallels, the cone becomes a cylinder, and a Mercator projection results. The default parallels are [15 75].

Remarks

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

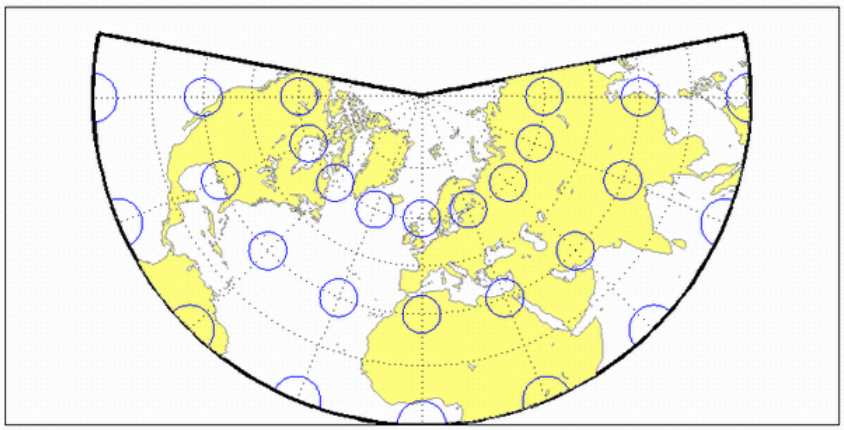
- Mapping Toolbox uses a different implementation of the standard Lambert conformal conic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Longitude data greater than 135º 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` on page 11-84

Introduced before R2006a

lambcyln

Lambert Equal-Area Cylindrical Projection

Classification

Cylindrical

Identifier

`lambcyln`

Graticule

Meridians: Equally spaced straight parallel lines 0.32 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder tangent at the Equator. It is equal-area, but distortion of shape increases with distance from the Equator. Scale is true along the Equator and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

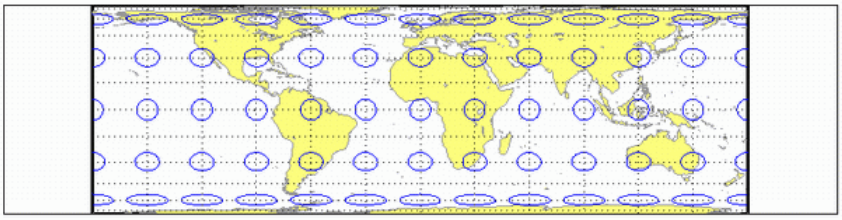
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks

This projection is named for Johann Heinrich Lambert and is a special form of the Equal-Area Cylindrical projection tangent at the Equator.

Example

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



Introduced before R2006a

loximuth

Loximuthal Projection

Classi cation

Pseudocylindrical

Identi er

`loximuth`

Graticule

Central Meridian: Straight line at least half as long as the Equator. Actual length depends on the choice of central latitude. Length is 0.5 when the central latitude is the Equator, for example, and 0.65 for central latitudes of 40º.

Other Meridians: Complex curves intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian. Symmetry about the Equator only when it is the central latitude.

Features

This projection has the special property that from the central point (the intersection of the central latitude with the central meridian), rhumb lines (loxodromes) are shown as straight, true to scale, and correct in azimuth from the center. This differs from the Mercator projection, in that rhumb lines are here shown in true scale and that unlike the Mercator, this projection does not maintain true azimuth for all points along the rhumb lines. Scale is true along the central meridian and is constant along any parallel, but not, generally, between parallels. It is free of distortion only at the central point and can be severely distorted in places. However, this projection is designed for its specific special property, in which distortion is not a concern.

Parallels

For this projection, only one standard parallel is specified: the central latitude described above. Specification of this central latitude defines the center of the loximuthal projection. The default value is 0º.

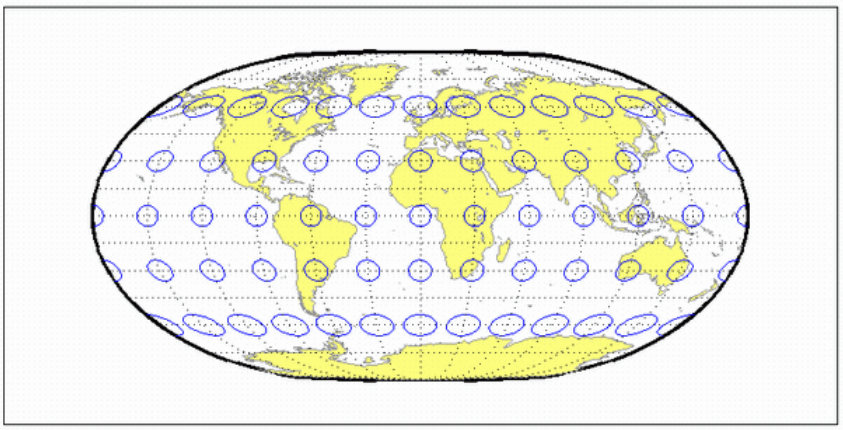
Remarks

- 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.

- This implementation of the loximuthal projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the loximuthal projection using the ESRI authority code 54023. For example: `projcrs(54023,'Authority','ESRI')`.

Example

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



Introduced before R2006a

atplrp

McBryde-Thomas Flat-Polar Parabolic Projection

Classification

Pseudocylindrical

Identifier

`flatplrp`

Graticule

Central Meridian: Straight line 0.48 as long as the Equator.

Other Meridians: Equally spaced parabolic curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 45º30' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the pointed-polar projections. It is free of distortion only at the two points where the central meridian intersects the 45º30' parallels. This projection is not conformal or equidistant.

Parallels

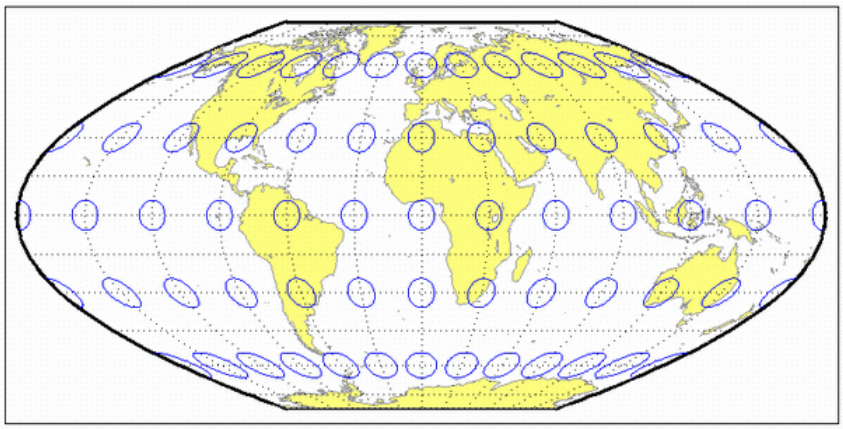
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 45º30'.

Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.

Example

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



Introduced before R2006a

atplrq

McBryde-Thomas Flat-Polar Quartic Projection

Classification

Pseudocylindrical

Identifier

`flatplrq`

Graticule

Central Meridian: Straight line 0.45 as long as the Equator.

Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 33º45' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the pointed-polar projections. It is free of distortion only at the two points where the central meridian intersects the 33º45' parallels. This projection is not conformal or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 33º45'.

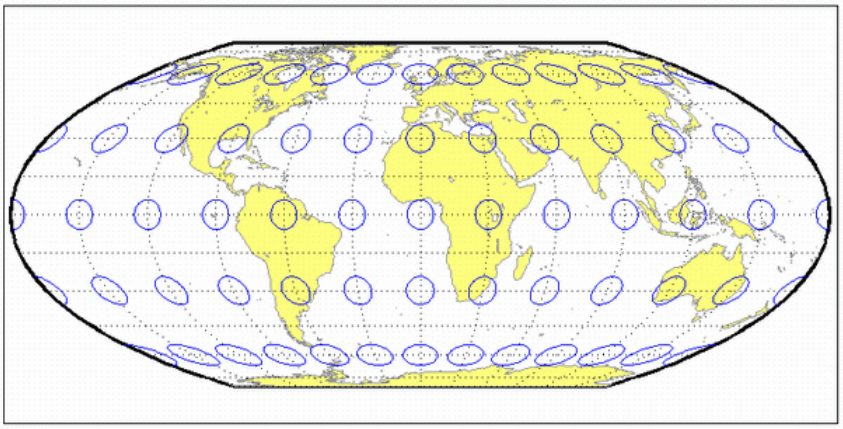
Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949, and is also known simply as the Flat-Polar Quartic projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('flatplrq', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

atplrs

McBryde-Thomas Flat-Polar Sinusoidal Projection

Classification

Pseudocylindrical

Identifier

`flatplrs`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is widest near the Equator.

Poles: Lines one-third as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This projection is equal-area. Scale is true along the 55º51' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the central meridian intersects the 55º51' parallels. This projection is not conformal or equidistant.

Parallels

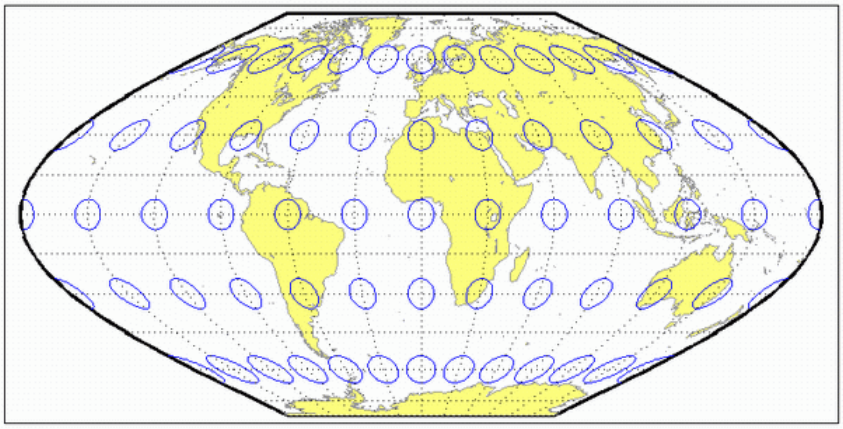
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 55º51'.

Remarks

This projection was presented by F. Webster McBryde and Paul D. Thomas in 1949.

Example

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



Introduced before R2006a

mercator

Mercator Projection

Classification

Cylindrical

Identifier

`mercator`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles.

Poles: Cannot be shown.

Symmetry: About any meridian or the Equator.

Features

This is a projection with parallel spacing calculated to maintain conformality. It is not equal-area, equidistant, or perspective. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. It is also constant in all directions near any given point. Scale becomes infinite at the poles. The appearance of the Mercator projection is unaffected by the selection of standard parallels; they serve only to define the latitude of true scale.

The Mercator, which may be the most famous of all projections, has the special feature that all rhumb lines, or loxodromes (lines that make equal angles with all meridians, i.e., lines of constant heading), are straight lines. This makes it an excellent projection for navigational purposes. However, the extreme area distortion makes it unsuitable for general maps of large areas.

Parallels

For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, any latitude less than 86º may be chosen; the default is arbitrarily set to 0º.

Remarks

- The Mercator projection is named for Gerardus Mercator, who presented it *for navigation* in 1569. It is now known to have been used for the Tunhuang star chart as early as 940 by Ch'ien Lo-Chih. It was first used in Europe by Erhard Etzlaub in 1511. It is also, but rarely, called the Wright projection, after Edward Wright, who developed the mathematics behind the projection in 1599.

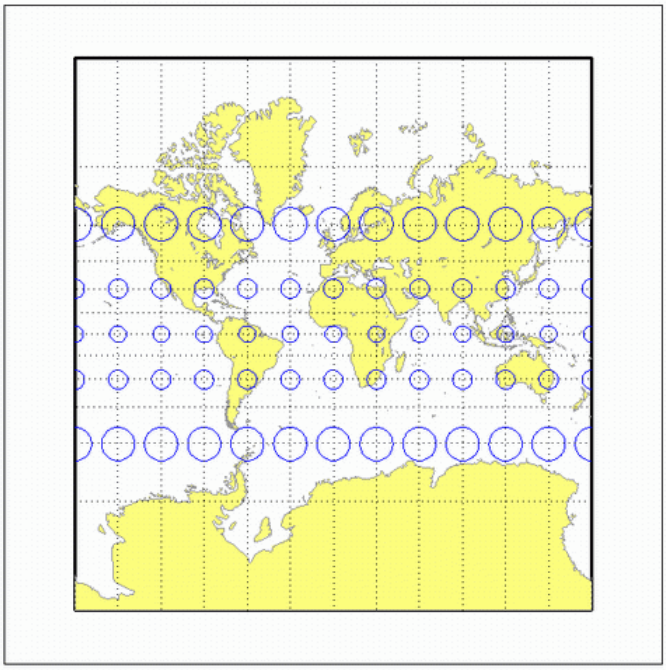
- Mapping Toolbox uses a different implementation of the Mercator projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data at latitudes greater than 86º is trimmed to prevent large *y*-values from dominating the display.

Example

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



Introduced before R2006a

miller

Miller Cylindrical Projection

Classification

Cylindrical

Identifier

`miller`

Graticule

Meridians: Equally spaced straight parallel lines 0.73 as long as the Equator.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing increases toward the poles, less rapidly than that of the Mercator projection.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection with parallel spacing calculated to maintain a look similar to the Mercator projection while reducing the distortion near the poles and allowing the poles to be displayed. It is not equal-area, equidistant, conformal, or perspective. Scale is true along the Equator and constant between two parallels equidistant from the Equator. There is no distortion near the Equator, and it increases moderately away from the Equator, but it becomes severe at the poles.

The Miller Cylindrical projection is derived from the Mercator projection; parallels are spaced from the Equator by calculating the distance on the Mercator for a parallel at 80% of the true latitude and dividing the result by 0.8. The result is that the two projections are almost identical near the Equator.

Parallels

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

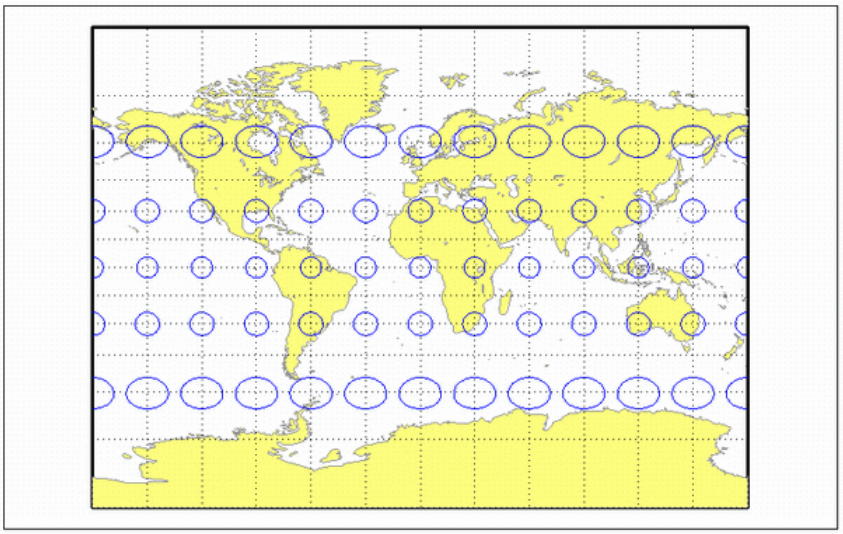
Remarks

- This projection was presented by Osborn Maitland Miller of the American Geographical Society in 1942. It is often used in place of the Mercator projection for atlas maps of the world, for which it is somewhat more appropriate.
- Mapping Toolbox uses a different implementation of the Miller projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

- The implementation of the Miller projection for displaying coordinates on map axes is applicable only for coordinates that are referenced to a sphere. The implementation of the Miller projection for projecting coordinates using the `projfwd` or `projinv` function is applicable for coordinates referenced to either a sphere or an ellipsoid.

Example

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



Introduced before R2006a

mollweid

Mollweide Projection

Classification

Pseudocylindrical

Identifier

`mollweid`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Meridians 90º east and west of the central meridian form a circle. The others are equally spaced semiellipses intersecting at the poles and concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator, but the spacing changes gradually.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 40º44' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. It is free of distortion only at the two points where the 40º44' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 40º44'.

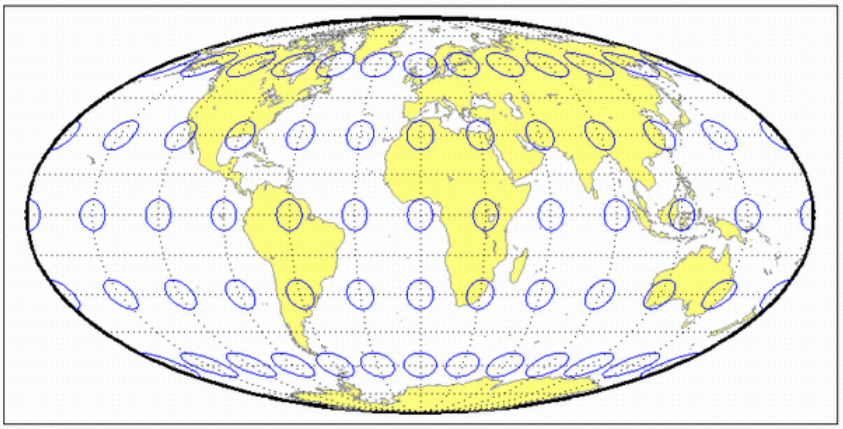
Remarks

This projection was presented by Carl B. Mollweide in 1805. Its other names include the Homolographic, the Homalographic, the Babinet, and the Elliptical projections. It is occasionally used for thematic world maps, and it is combined with the Sinusoidal to produce the Goode Homolosine projection.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('mollweid', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

murdoch1

Murdoch I Conic Projection

Classification

Conic

Identifier

`murdoch1`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles.

Parallels: Equally spaced concentric circular arcs.

Poles: Arcs, one of which might become a point in the limit.

Symmetry: About any meridian.

Features

This is an equidistant projection that is nearly minimum-error. Scale is true along any meridian and is constant along any parallel. Scale is also true along two standard parallels. These must be calculated, however (see remark on parallels below). The total area of the mapped area is correct, but it is not equal-area everywhere.

Parallels

The parallels for this projection are not standard parallels, but rather limiting parallels. The special feature of this map, correct total area, holds between these parallels. The default parallels are [15 75].

Remarks

- Described by Patrick Murdoch in 1758.
- This implementation of the Murdoch I conic projection is applicable only for coordinates that are referenced to a sphere.

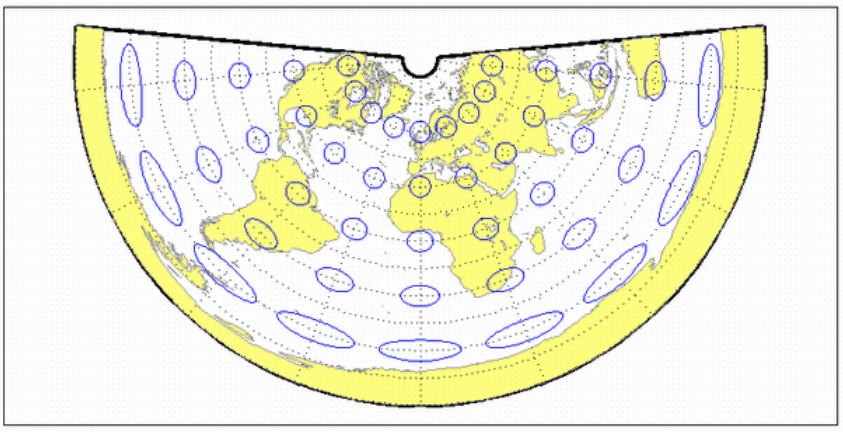
Limitations

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

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('murdoch1', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

murdoch3

Murdoch III Minimum Error Conic Projection

Classification

Conic

Identifier

`murdoch3`

Graticule

Meridians: Equally spaced straight lines converging at one of the poles.

Parallels: Equally spaced concentric circular arcs.

Poles: Arcs, one of which might become a point in the limit.

Symmetry: About any meridian.

Features

This is an equidistant projection that is minimum-error. Scale is true along any meridian and is constant along any parallel. Scale is also true along two standard parallels. These must be calculated, however (see remark on parallels below). The total area of the mapped area is correct, but it is not equal-area everywhere.

Parallels

The parallels for this projection are not standard parallels, but rather limiting parallels. The special feature of this map, correct total area, holds between these parallels. The default parallels are [15 75].

Remarks

- Described by Patrick Murdoch in 1758, with errors corrected by Everett in 1904.
- This implementation of the Murdoch III minimum error conic projection is applicable only for coordinates that are referenced to a sphere.

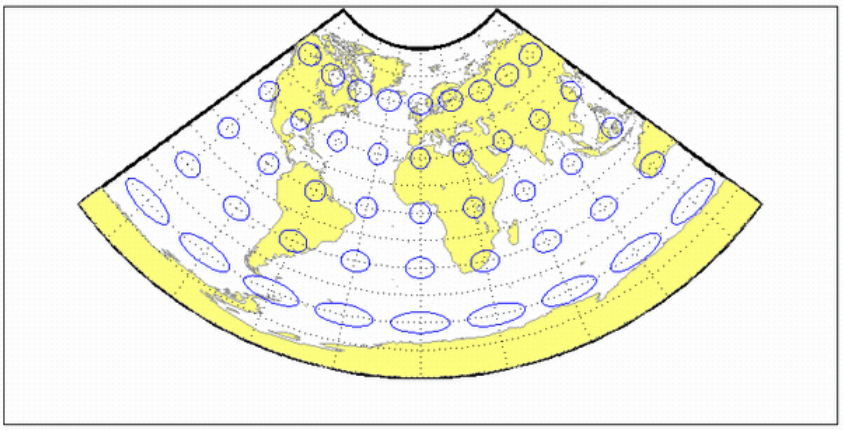
Limitations

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

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('murdoch3', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

ortho

Orthographic Projection

Classification

Azimuthal

Identifier

`ortho`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing decreases away from this pole. The opposite hemisphere cannot be shown.

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from an infinite distance (that is, orthogonally). The center point is a pole in the common polar aspect, but can be any point. This projection has two significant properties. It looks like a globe, providing views of the Earth resembling those seen from outer space. Additionally, all great and small circles are either straight lines or elliptical arcs on this projection. Scale is true only at the center point and is constant in the circumferential direction along any circle having the center point as its center. Distortion increases rapidly away from the center point, the only place that is distortion-free. This projection is neither conformal nor equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

- This projection appears to have been developed by the Egyptians and Greeks by the second century B.C.
- This implementation of the orthographic projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map

projection structure. You can create a `projcrs` object for the orthographic projection using the ESRI authority code 102037. For example: `projcrs(102037,'Authority','ESRI')`.

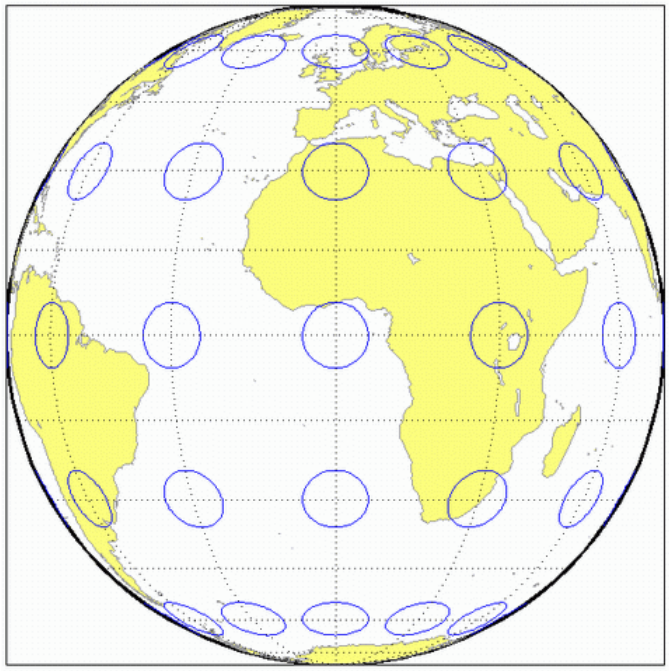
- Mapping Toolbox uses a different implementation of the orthographic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data greater than 89º distant from the center point is trimmed.

Example

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



Introduced before R2006a

pcarree

Plate Carree Projection

Classi cation

Cylindrical

Identi er

`pcarree`

Graticule

Meridians: Equally spaced straight parallel lines half as long as the Equator.

Parallels: Equally spaced straight parallel lines, perpendicular to and having the same spacing as the meridians.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is a projection onto a cylinder tangent at the Equator. Distortion of both shape and area increases with distance from the Equator. Scale is true along all meridians (i.e., it is equidistant) and the Equator and is constant along any parallel and along the parallel of opposite sign.

Parallels

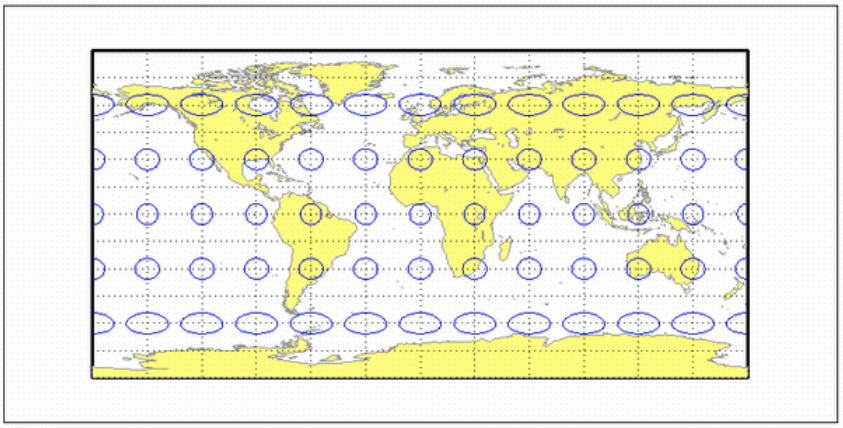
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 0º.

Remarks

- This projection, like the more general Equidistant Cylindrical, is credited to Marinus of Tyre, thought to have invented it about A.D. 100. It may, in fact, have been originated by Erastosthenes, who lived approximately 275–195 B.C. The Plate Carrée has the most simply constructed graticule of any projection. It was used frequently in the 15th and 16th centuries and is quite common today in very simple computer mapping programs. It is the simplest and limiting form of the Equidistant Cylindrical projection. Another name for the Plate Carrée projection is the Simple Cylindrical. Its transverse aspect is the Cassini projection.
- On the sphere, this projection can have an arbitrary, oblique aspect, as controlled by the `Origin` property of the map axes. On the ellipsoid, only the equatorial aspect is supported.

Example

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



Introduced before R2006a

polycon

Polyconic Projection

Classi cation

Polyconic

Identi er

`polycon`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About the Equator or the central meridian.

Features

For this projection, each parallel has a curvature identical to its curvature on a cone tangent at that latitude. Since each parallel has its own cone, this is a “polyconic” projection. Scale is true along the central meridian and along each parallel. This projection is free of distortion only along the central meridian; distortion can be severe at extreme longitudes. This projection is neither conformal nor equal-area.

Parallels

By definition, this projection has no standard parallels, since every parallel is a *standard parallel*.

Remarks

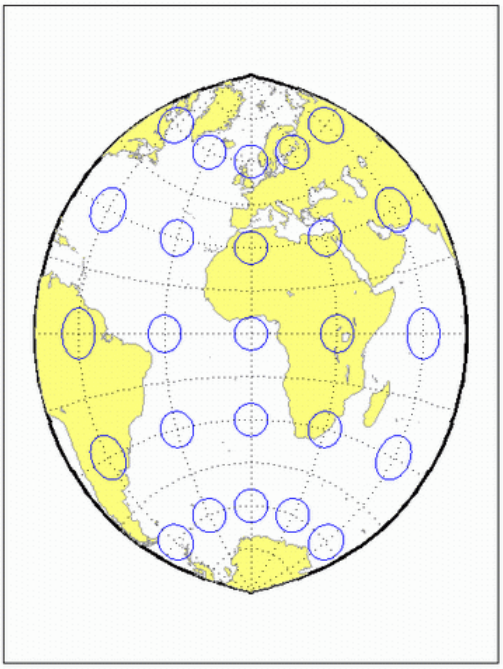
This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.

Limitations

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

Example

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



See Also

`polyconstd` on page 11-114

Introduced before R2006a

polyconstd

Polyconic Projection — Standard

Classification

Polyconic

Identifier

`polyconstd`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves spaced equally along the Equator and each parallel, and concave toward the central meridian.

Parallels: The Equator is a straight line. All other parallels are nonconcentric circular arcs spaced at true distances along the central meridian.

Poles: Normally circular arcs, enclosing the same angle as the displayed parallels.

Symmetry: About the Equator or the central meridian.

Features

`polyconstd` implements the Polyconic projection directly on a reference ellipsoid, consistent with the industry-standard definition of this projection. See `polycon` on page 11-112 for an alternative implementation based on rotating the rectifying sphere.

For this projection, each parallel has a curvature identical to its curvature on a cone tangent at that latitude. Since each parallel has its own cone, this is a “polyconic” projection. Scale is true along the central meridian and along each parallel. This projection is free of distortion only along the central meridian; distortion can be severe at extreme longitudes. This projection is neither conformal nor equal-area.

Parallels

By definition, this projection has no standard parallels, since every parallel is a *standard parallel*.

Remarks

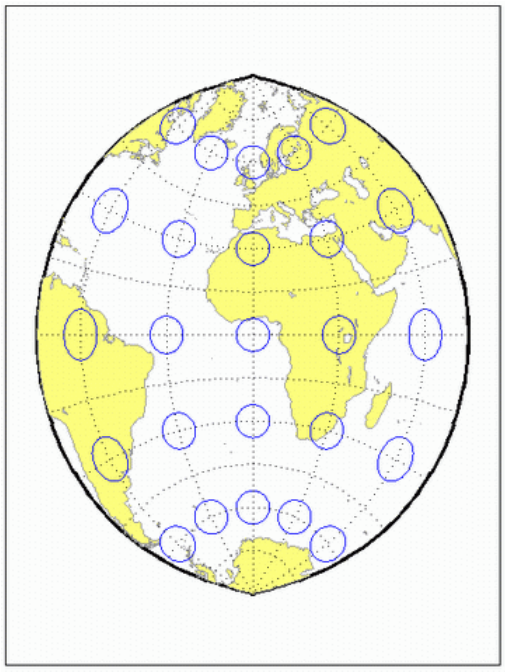
- This projection was apparently originated about 1820 by Ferdinand Rudolph Hassler. It is also known as the American Polyconic and the Ordinary Polyconic projection.
- Mapping Toolbox uses a different implementation of the standard polyconic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

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

Example

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



See Also

polycon on page 11-112

Introduced before R2006a

putnins5

Putnins P5 Projection

Classification

Pseudocylindrical

Identifier

`putnins5`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced portions of hyperbolas intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 21º14' parallels and is constant along any parallel, between any pair of parallels equidistant from the Equator, and along the central meridian. It is not free of distortion at any point. This projection is not equal-area, conformal, or equidistant.

Parallels

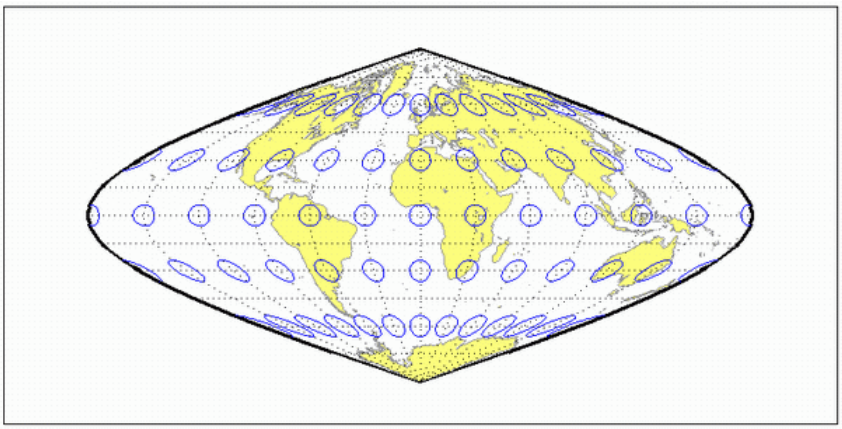
For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 21º14'.

Remarks

- This projection was presented by Reinholds V. Putnins in 1934.
- This implementation of the Putnins P5 projection is applicable only for coordinates that are referenced to a sphere.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('putnins5', 'Frame', 'on', 'Grid', 'on');
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

quartic

Quartic Authalic Projection

Classi cation

Pseudocylindrical

Identi er

`quartic`

Graticule

Central Meridian: Straight line 0.45 as long as the Equator.

Other Meridians: Equally spaced quartic (fourth-order equation) curves concave toward the central meridian.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing changes gradually and is greatest near the Equator.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the Equator and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is severe near the outer meridians at high latitudes, but less so than on the Sinusoidal projection. It is free of distortion along the Equator. This projection is not conformal or equidistant.

Parallels

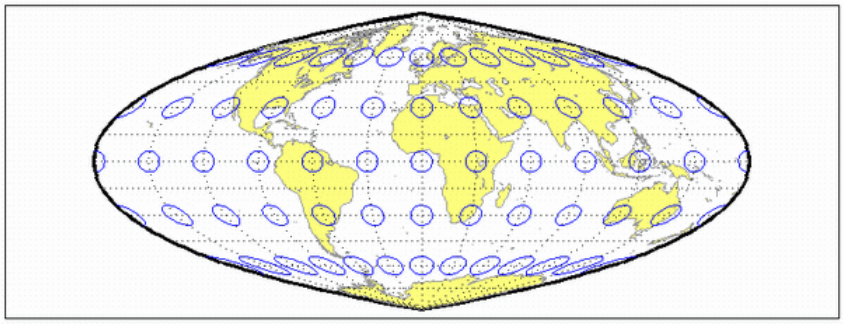
This projection has one standard parallel, which is by definition fixed at 0º.

Remarks

This projection was presented by Karl Siemon in 1937 and independently by Oscar Sherman Adams in 1945.

Example

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



Introduced before R2006a

robinson

Robinson Projection

Classification

Pseudocylindrical

Identifier

`robinson`

Graticule

Central Meridian: Straight line 0.51 as long as the Equator.

Other Meridians: Equally spaced, resemble elliptical arcs and are concave toward the central meridian.

Parallels: Straight parallel lines, perpendicular to the central meridian. Spacing is equal between the 38º parallels, decreasing outside these limits.

Poles: Lines 0.53 as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

Scale is true along the 38º parallels and is constant along any parallel or between any pair of parallels equidistant from the Equator. It is not free of distortion at any point, but distortion is very low within about 45º of the center and along the Equator. This projection is not equal-area, conformal, or equidistant; however, it is considered to *look right* for world maps, and hence is widely used by Rand McNally, the National Geographic Society, and others. This feature is achieved through the use of tabular coordinates rather than mathematical formulae for the graticules.

Parallels

For this projection, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. The standard parallel is by definition fixed at 38º.

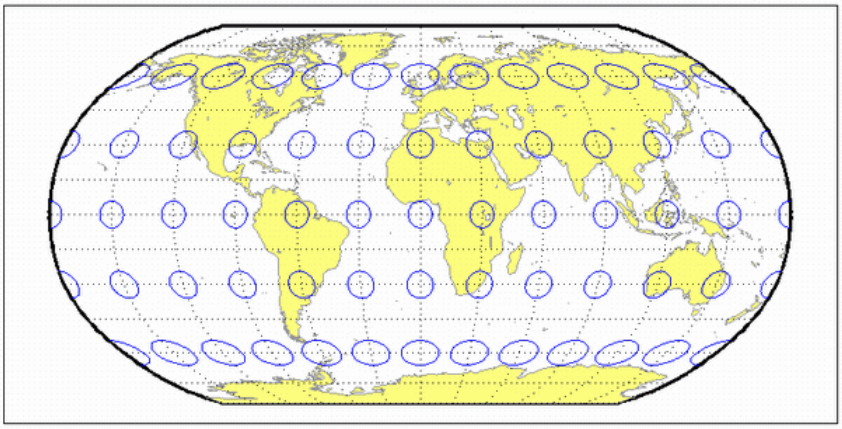
Remarks

- This projection was presented by Arthur H. Robinson in 1963, and is also called the Orthophanic projection, which means *right appearing*.
- This implementation of the Robinson projection is applicable only for coordinates that are referenced to a sphere. If you want to project coordinates that are referenced to an ellipsoid, using the `projfwd` or `projinv` functions, then create a `projcrs` object instead of a map projection structure. You can create a `projcrs` object for the Robinson projection using the ESRI authority code 54030. For example: `projcrs(54030,'Authority','ESRI')`.

- Mapping Toolbox uses a different implementation of the Robinson projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

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



Introduced before R2006a

sinusoid

Sinusoidal projection

Classification

Pseudocylindrical

Identifier

`sinusoid`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced sinusoidal curves intersecting at the poles and concave toward the central meridian.

Parallels: Equally spaced straight parallel lines, perpendicular to the central meridian.

Poles: Points.

Symmetry: About the central meridian or the Equator.

Features

This projection is equal-area. Scale is true along every parallel and along the central meridian. There is no distortion along the Equator or along the central meridian, but it becomes severe near the outer meridians at high latitudes.

Parallels

This projection has one standard parallel, which is by definition fixed at 0º.

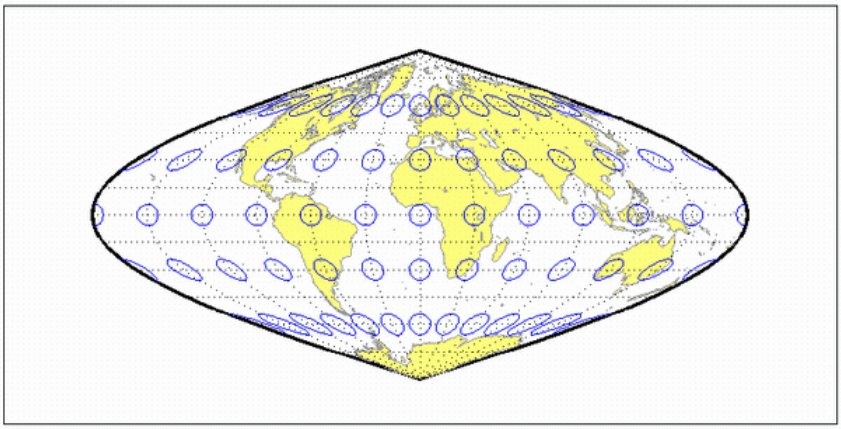
Remarks

- This projection was developed in the 16th century. It was used by Jean Cossin in 1570 and by Jodocus Hondius in Mercator atlases of the early 17th century. It is the oldest pseudocylindrical projection currently in use, and is sometimes called the Sanson-Flamsteed or the Mercator Equal-Area projection.
- Mapping Toolbox uses a different implementation of the sinusoidal projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('sinusoid', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

stereo

Stereographic Projection

Classification

Azimuthal

Identifier

`stereo`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing increases gradually away from this pole.

Pole: The central pole is a point; the other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from the point antipodal to the center point. The center point is a pole in the common polar aspect, but can be any point. This projection has two significant properties. It is conformal, being free from angular distortion. Additionally, all great and small circles are either straight lines or circular arcs on this projection. Scale is true only at the center point and is constant along any circle having the center point as its center. This projection is not equal-area.

Parallels

There are no standard parallels for azimuthal projections.

Remarks

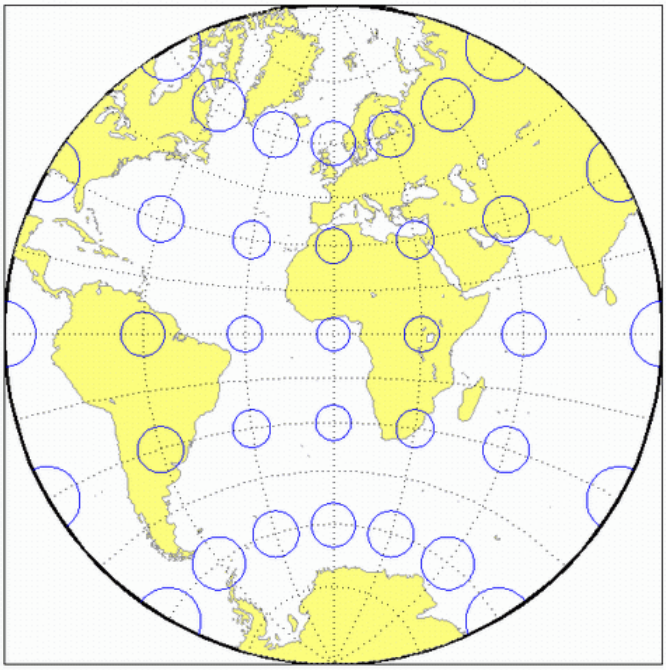
- The polar aspect of this projection appears to have been developed by the Egyptians and Greeks by the second century B.C.
- Mapping Toolbox uses a different implementation of the stereographic projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Limitations

Data greater than 90º distant from the center point is trimmed.

Example

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



Introduced before R2006a

modsine

Tissot Modified Sinusoidal Projection

Classification

Pseudocylindrical

Identifier

`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

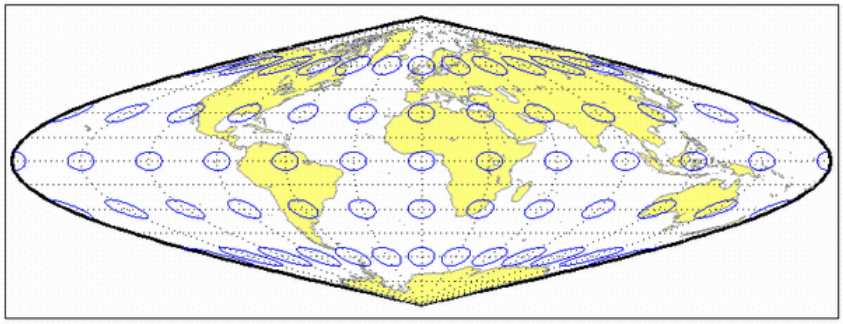
This projection was first described by N. A. Tissot in 1881

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;
```



Introduced before R2006a

tranmerc

Transverse Mercator Projection

Classification

Cylindrical

Identifier

`tranmerc`

Features

This conformal projection is the transverse form of the Mercator projection and is also known as the Gauss-Krueger projection. It is not equal area, equidistant, or perspective.

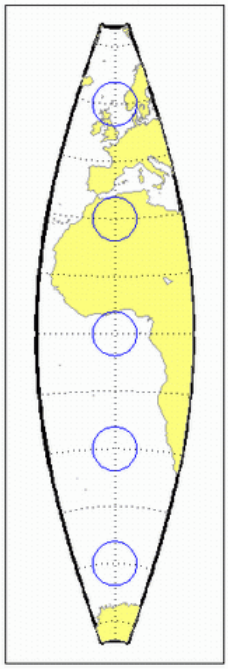
The scale is constant along the central meridian, and increases to the east and west. The scale at the central meridian can be set true to scale, or reduced slightly to render the mean scale of the overall map more correctly.

Remarks

- The uniformity of scale along its central meridian makes transverse Mercator an excellent choice for mapping areas that are elongated north-to-south. Its best known application is the definition of Universal Transverse Mercator (UTM) coordinates. Each UTM zone spans only 6 degrees of longitude, but the northern half extends from the equator all the way to 84 degrees north and the southern half extends from 80 degrees south to the equator. Other map grids based on transverse Mercator include many of the state plane zones in the U.S. and the U.K. National Grid.
- Mapping Toolbox uses a different implementation of the transverse Mercator projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Example

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



Introduced before R2006a

trystan

Trystan Edwards Cylindrical Projection

Classification

Cylindrical

Identifier

`trystan`

Graticule

Meridians: Equally spaced straight parallel lines.

Parallels: Unequally spaced straight parallel lines, perpendicular to the meridians. Spacing is closest near the poles.

Poles: Straight lines equal in length to the Equator.

Symmetry: About any meridian or the Equator.

Features

This is an orthographic projection onto a cylinder secant at the 37º24' parallels. It is equal-area, but distortion of shape increases with distance from the standard parallels. Scale is true along the standard parallels and constant between two parallels equidistant from the Equator. This projection is not equidistant.

Parallels

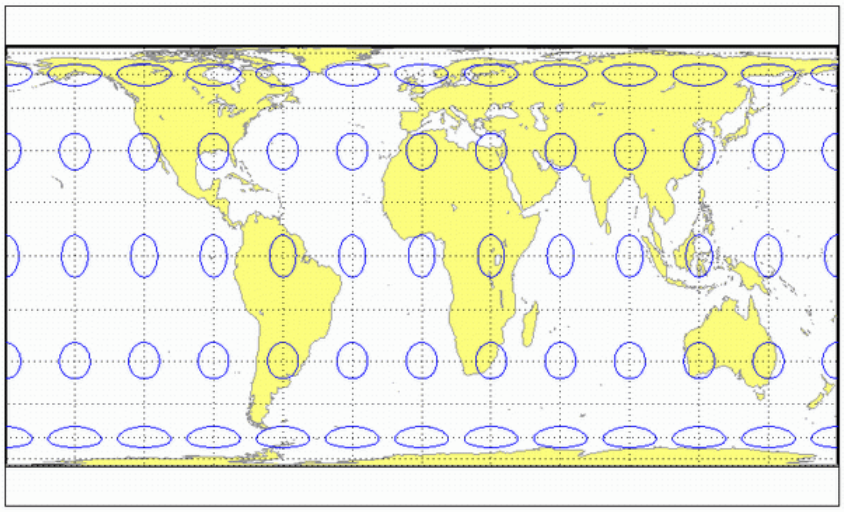
For cylindrical projections, only one standard parallel is specified. The other standard parallel is the same latitude with the opposite sign. For this projection, the standard parallel is by definition fixed at 37º24'.

Remarks

This projection is named for Trystan Edwards, who presented it in 1953. It is a special form of the Equal-Area Cylindrical projection secant at 37º24'N and S.

Example

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



Introduced before R2006a

ups

Universal Polar Stereographic System

Classification

Azimuthal

Identifier

ups

Graticule

The graticule described is for the southern zone.

Meridians: Equally spaced straight lines centered on the South Pole. The angles displayed are the true angles between meridians.

Parallels: Unequally spaced circles centered on the South Pole. Spacing increases gradually away from the circle of true scale along latitude 87 degrees, 7 minutes N. The opposite pole cannot be shown.

Poles: The South Pole is a point. The North Pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent to either the North or South Pole. It is conformal, being free from angular distortion. Additionally, all great and small circles are either straight lines or circular arcs on this projection. Scale is true along latitudes 87 degrees, 7 minutes N or S, and is constant along any other parallel. This projection is not equal area.

Parallels

The parallels 87 degrees, 7 minutes N and S are lines of true scale by virtue of the scale factor. There are no standard parallels for azimuthal projections.

Remarks

- This projection is a special case of the stereographic projection in the polar aspect. It is used as part of the Universal Transverse Mercator (UTM) system to extend coverage to the poles. This projection has two zones: “North” for latitudes 84º N to 90º N, and “South” for latitudes 80º S to 90º S. The defaults for this projection are: scale factor is 0.994, false easting and northing are 2,000,000 meters. The international ellipsoid in units of meters is used as the geoid model.
- Mapping Toolbox uses a different implementation of the UPS projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.

Introduced in R2006a

utm

Universal Transverse Mercator System

Classification

Cylindrical

Identifier

`utm`

Graticule

Meridians: Complex curves concave toward the central meridian.

Parallels: Complex curves concave toward the nearest pole.

Poles: Not shown.

Symmetry: About the central meridian or the Equator.

Features

This is a conformal projection with parameters chosen to minimize distortion over a defined set of small areas. It is not equal area, equidistant, or perspective. Scale is true along two straight lines on the map approximately 180 kilometers east and west of the central meridian, and is constant along other straight lines equidistant from the central meridian. Scale is less than true between, and greater than true outside the lines of true scale.

Parallels

There are no standard parallels for this projection. There are two lines of zero distortion by virtue of the scale factor.

Remarks

The UTM system divides the world between 80º S and 84º degrees N into a set of quadrangles called zones. Zones generally cover 6 degrees of longitude and 8 degrees of latitude. Each zone has a set of defined projection parameters, including central meridian, false eastings and northings and the reference ellipsoid. The projection equations are the Gauss-Krüger versions of the Transverse Mercator. The projected coordinates form a grid system, in which a location is specified by the zone, easting and northing.

The UTM system was introduced in the 1940s by the U.S. Army. It is widely used in topographic and military mapping.

Introduced in R2006a

vgrint1

Van der Grinten I Projection

Classification

Polyconic

Identifier

`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

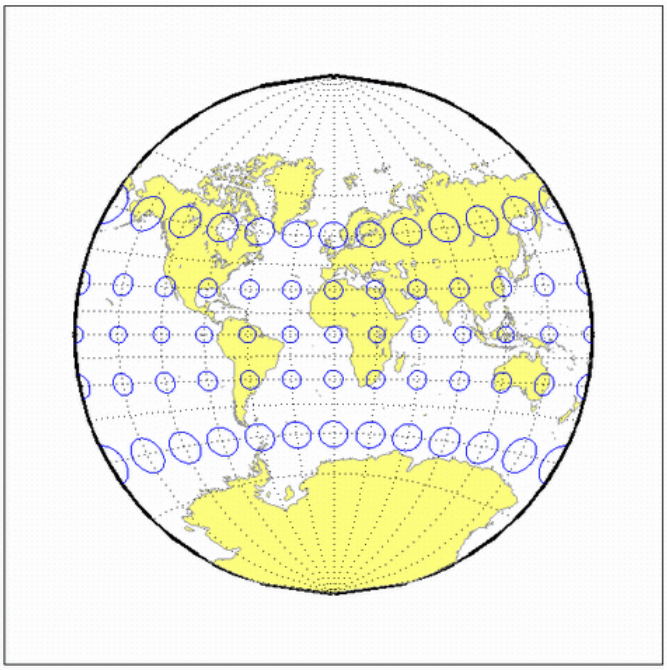
- This projection was presented by Alphons J. Van der Grinten in 1898. He obtained a U.S. patent for it in 1904. It is also known simply as the Van der Grinten projection (without the “I”).
- Mapping Toolbox uses a different implementation of the Van der Grinten I projection for displaying coordinates on map axes than for projecting coordinates using the `projfwd` or `projinv` function. These implementations may produce differing results.
- The implementation of the Van der Grinten I projection for displaying coordinates on map axes is applicable only for coordinates that are referenced to a sphere. The implementation of the Van der Grinten I projection for projecting coordinates using the `projfwd` or `projinv` function is applicable for coordinates referenced to either a sphere or an ellipsoid.

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;
```



Introduced before R2006a

vperspec

Vertical Perspective Azimuthal Projection

Classification

Azimuthal

Identifier

`vperspec`

Graticule

The graticule described is for a polar aspect.

Meridians: Equally spaced straight lines intersecting at the central pole. The angles displayed are true angles between meridians.

Parallels: Unequally spaced circles centered on the central pole. Spacing decreases away from this pole. The opposite hemisphere cannot be shown, nor can distant parts of the closer hemisphere. The limit of visibility depends on the observation altitude.

Poles: The central pole is a point. The other pole is not shown.

Symmetry: About any meridian.

Features

This is a perspective projection on a plane tangent at the center point from a finite distance. Scale is true only at the center point, and is constant in the circumferential direction along any circle having the center point as its center. Distortion increases rapidly away from the center point, the only point which is distortion free. This projection is neither conformal nor equal area.

Remarks

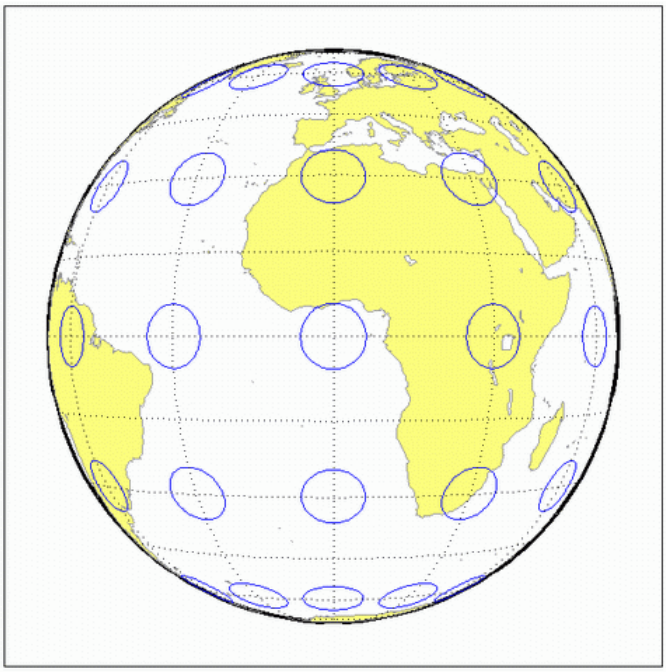
- This projection provides views of the globe resembling those seen from a spacecraft in orbit. The Orthographic projection is a limiting form with the observer at an infinite distance.
- This projection requires a view altitude parameter, which specifies the observer's altitude above the origin point. Because this parameter is unique to this projection and because the projection does not need any standard parallels, a special workaround is used. Rather than add an extra map axes property just for `vperspec`, the `MapParallels` property is repurposed instead. You should assign the desired view altitude value to the `MapParallels` property. Provide a scalar value for length in the same units as the earth radius or semi-major axis length used in the map axes reference ellipsoid (`'Geoid'`) property.
- This implementation of the vertical perspective azimuthal projection is applicable only for coordinates that are referenced to a sphere.

Limitations

Data more distant than the limit of visibility is trimmed.

Example

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



Introduced before R2006a

wagner4

Wagner IV Projection

Classification

Pseudocylindrical

Identifier

`wagner4`

Graticule

Central Meridian: Straight line half as long as the Equator.

Other Meridians: Equally spaced portions of ellipses concave toward the central meridian. The meridians 103º55' east and west of the central meridian are circular arcs.

Parallels: Unequally spaced straight parallel lines, perpendicular to the central meridian. Spacing is greatest toward the Equator.

Poles: Lines half as long as the Equator.

Symmetry: About the central meridian or the Equator.

Features

This is an equal-area projection. Scale is true along the 42º59' parallels and is constant along any parallel and between any pair of parallels equidistant from the Equator. Distortion is not as extreme near the outer meridians at high latitudes as for pointed-polar pseudocylindrical projections, but there is considerable distortion throughout the polar regions. It is free of distortion only at the two points where the 42º59' parallels intersect the central meridian. This projection is not conformal or equidistant.

Parallels

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

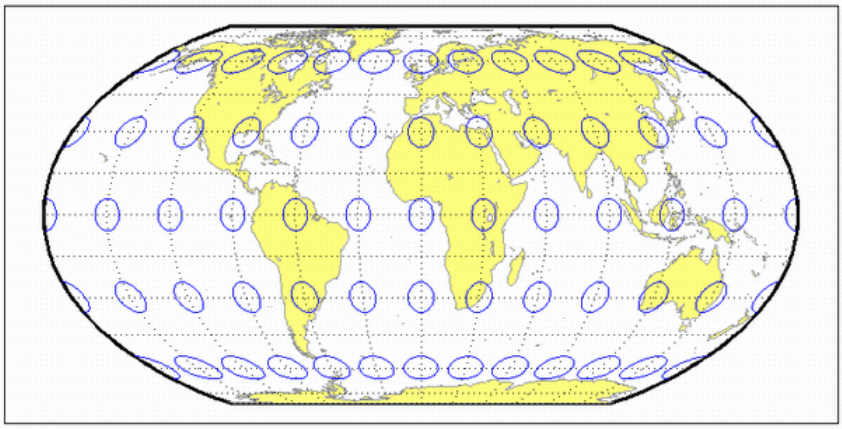
Remarks

This projection was presented by Karlheinz Wagner in 1932.

Example

```
landareas = shaperead('landareas.shp','UseGeoCoords',true);
axesm ('wagner4', 'Frame', 'on', 'Grid', 'on');
```

```
geoshow(landareas,'FaceColor',[1 1 .5],'EdgeColor',[.6 .6 .6]);
tissot;
```



Introduced before R2006a

werner

Werner Projection

Classification

Pseudoconic

Identifier

`werner`

Graticule

Central Meridian: A straight line.

Meridians: Complex curves connecting points equally spaced along each parallel and concave toward the central meridian.

Parallels: Concentric circular arcs spaced at true distances along the central meridian, centered on one of the poles.

Poles: Points.

Symmetry: About the central meridian.

Features

This is an equal-area projection. It is a Bonne projection with one of the poles as its standard parallel. The central meridian is free of distortion. This projection is not conformal. Its heart shape gives it the additional descriptor *cordiform*.

Parallels

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

Remarks

This projection was developed by Johannes Stabius (Stab) about 1500 and was promoted by Johannes Werner in 1514. It is also called the Stab-Werner projection.

Example

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