Navigation

On this page…

What Is Navigation?

Navigation is the process of planning, recording, and controlling the movement of a craft or vehicle from one location to another. The word derives from the Latin roots *navis* ("ship") and *agere* ("to move or direct"). Geographic information—usually in the form of latitudes and longitudes—is at the core of navigation practice. The toolbox includes specialized functions for navigating across expanses of the globe, for which projected coordinates are of limited use.

Navigating on land, over water, and through the air can involve a variety of tasks:

- Establishing position, using known, fixed landmarks (piloting)
- Using the stars, sun, and moon (celestial navigation)
- Using technology to fix positions (inertial guidance, radio beacons, and satellite navigation, including GPS)
- Deducing net movement from a past known position (dead reckoning)

Another navigational task involves planning a voyage or flight, which includes determining an efficient route (usually by great circle approximation), weather avoidance (optimal track routing), and setting out a plan of intended movement (track laydown). Mapping Toolbox™ functions support these navigational activities as well.

Conventions for Navigational Functions

Units

You can use and convert among several angular and distance measurement units. The navigational support functions are

- `dreckon`
- `gcwaypts`
- `legs`
- `navfix`

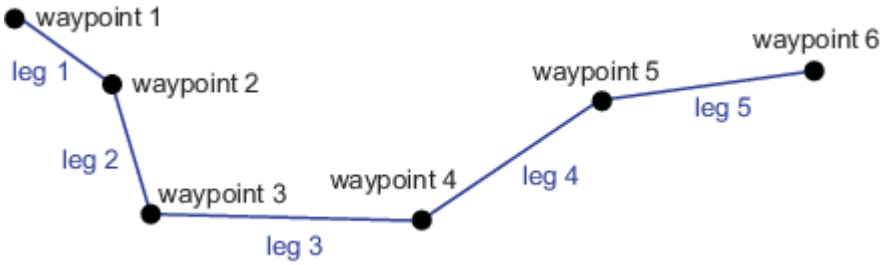
To make these functions easy to use, and to conform to common navigational practice, *for these specific functions only*, certain conventions are used:

- Angles are always in degrees.
- Distances are always in nautical miles.
- Speeds are always in knots (nautical miles per hour).

Related functions that *do not* carry this restriction include **`rhxrh`**, **`scxsc`**, **`gcxgc`**, **`gcxsc`**, **`track`**, **`timezone`**, and **`crossfix`**, because of their potential for application outside navigation.

Navigational Track Format

Navigational track format requires column-vector variables for the latitudes and longitudes of track waypoints. A *waypoint* is a point through which a track passes, usually corresponding to a course (or speed) change. Navigational tracks are made up of the line segments connecting these waypoints, which are called *legs*. In this format, therefore, *n* legs are described using *n+1* waypoints, because an endpoint for the final leg must be defined. Mapping Toolbox navigation functions always presume angle units are always given in degrees.



Here, five track legs require six waypoints. In navigational track format, the waypoints are represented by two 6-by-1 vectors, one for the latitudes and one for the longitudes.

Fixing Position

The fundamental objective of navigation is to determine at a given moment how to proceed to your destination, avoiding hazards on the way. The first step in accomplishing this is to establish your current position. Early sailors kept within sight of land to facilitate this. Today, navigation within sight (or radar range) of land is called *piloting*. Positions are fixed by correlating the bearings and/or ranges of landmarks. In real-life piloting, all sighting bearings are treated as rhumb lines, while in fact they are actually great circles.

Over the distances involved with visual sightings (up to 20 or 30 nautical miles), this assumption causes no measurable error and it provides the significant advantage of allowing the navigator to plot all bearings as straight lines on a Mercator projection.

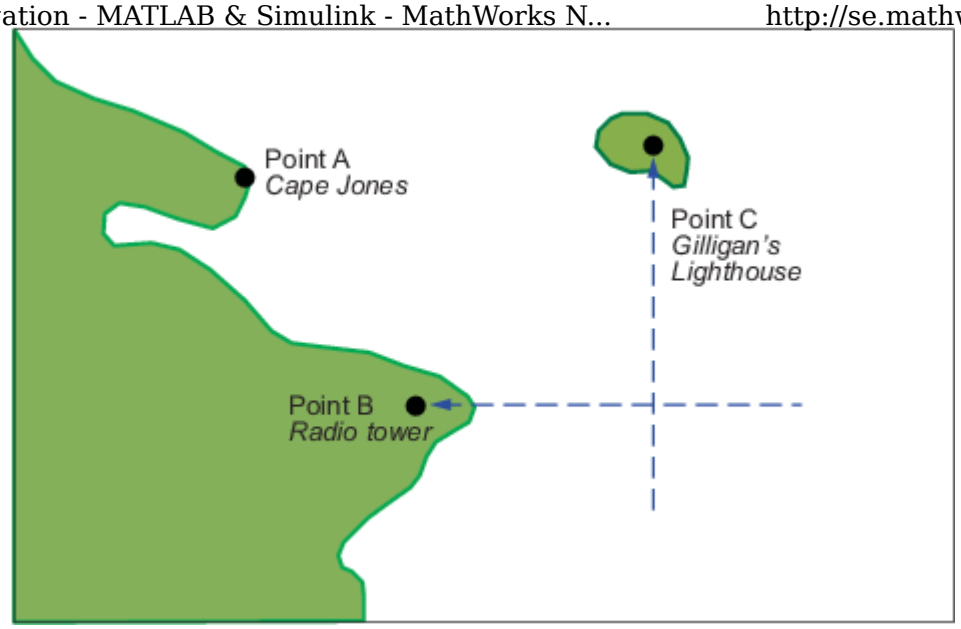
The Mercator was designed exactly for this purpose. Range circles, which might be determined with a radar, are assumed to plot as true circles on a Mercator chart. This allows the navigator to manually draw the range arc with a compass.

These assumptions also lead to computationally efficient methods for fixing positions with a computer. The toolbox includes the `navfix` function, which mimics the manual plotting and fixing process using these assumptions.

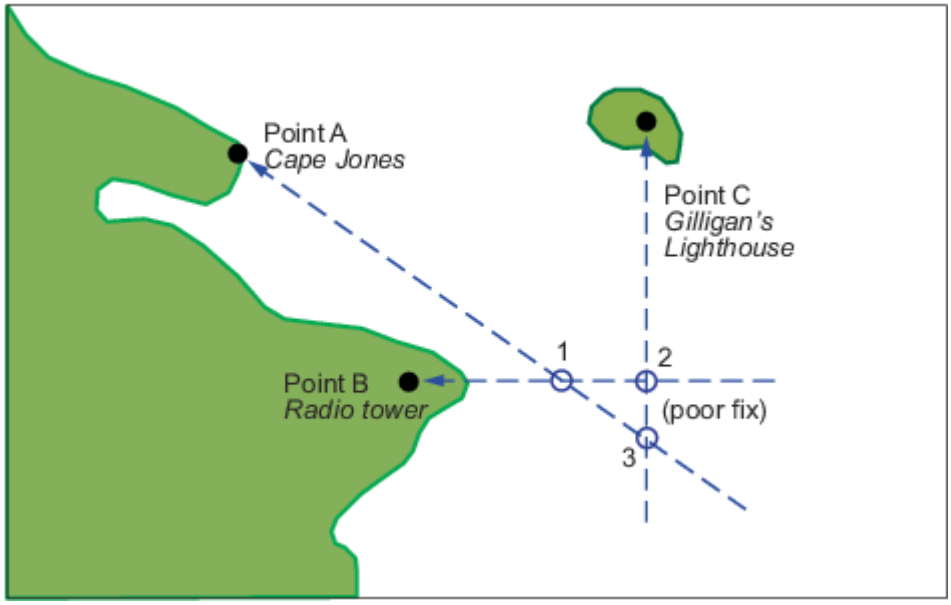
To obtain a good navigational fix, your relationship to at least three known points is considered necessary. A questionable or poor fix can be obtained with two known points.

Some Possible Situations

In this imaginary coastal region, you take a visual bearing on the radio tower of 270º. At the same time, Gilligan's Lighthouse bears 0º. If you plot a 90º-270º line through the radio tower and a 0º-180º line through the lighthouse on your Mercator chart, the point at which the lines cross is a fix. Since you have used only two lines, however, its quality is questionable.



But wait; your port lookout says he took a bearing on Cape Jones of 300º. If that line exactly crosses the point of intersection of the first two lines, you will have a perfect fix.

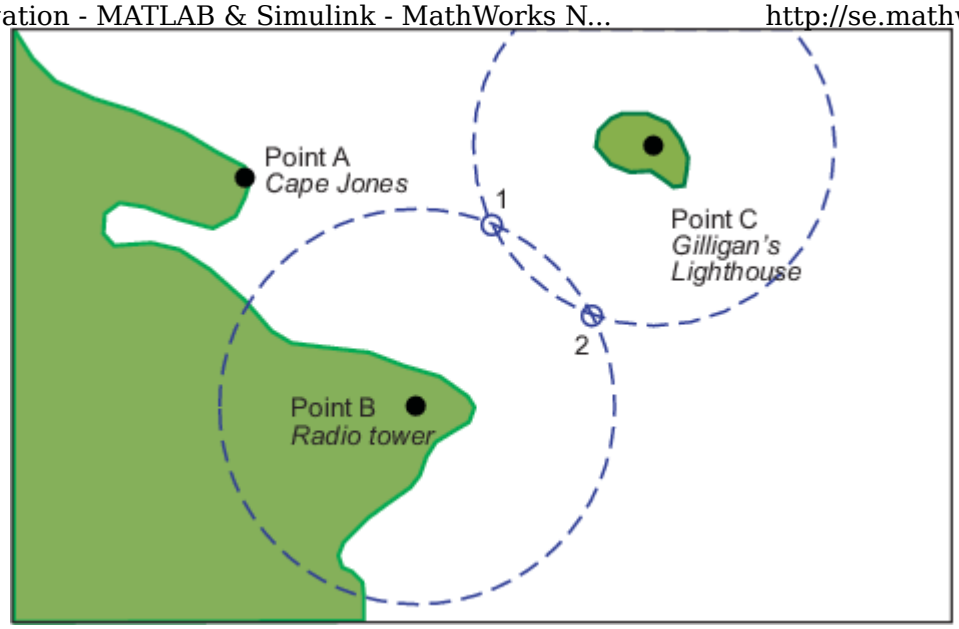


Whoops. What happened? Is your lookout in error? Possibly, but perhaps one or both of your bearings was slightly in error. This happens all the time. Which point, 1, 2, or 3, is correct? As far as you know, they are all equally valid.

In practice, the little triangle is plotted, and the fix position is taken as either the center of the triangle or the vertex closest to a danger (like shoal water). If the triangle is large, the quality is reported as *poor*, or even as *no fix*. If a fourth line of bearing is available, it can be plotted to try to resolve the ambiguity. When all three lines appear to cross at exactly the same point, the quality is reported as *excellent* or *perfect*.

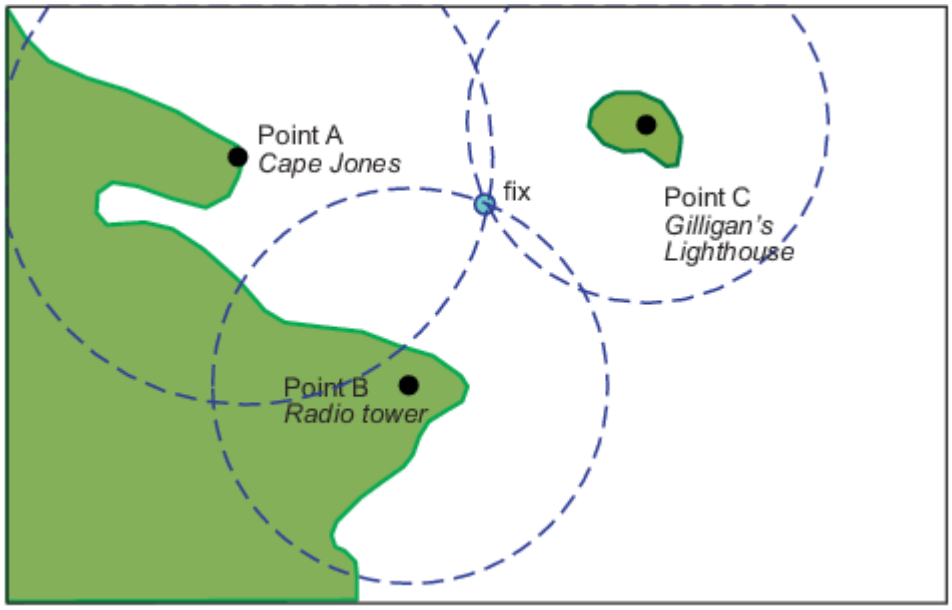
Notice that three lines resulted in three intersection points. Four lines would return six intersection points. This is a case of combinatorial counting. Each intersection corresponds to choosing two lines to intersect from among *n* lines.

The next time you traverse these straits, it is a very foggy morning. You can't see any landmarks, but luckily, your navigational radar is operating. Each of these landmarks has a good radar signature, so you're not worried. You get a range from the radio tower of 14 nautical miles and a range from the lighthouse of 15 nautical miles.



Now what? You took ranges from only two objects, and yet you have two possible positions. This ambiguity arises from the fact that circles can intersect twice.

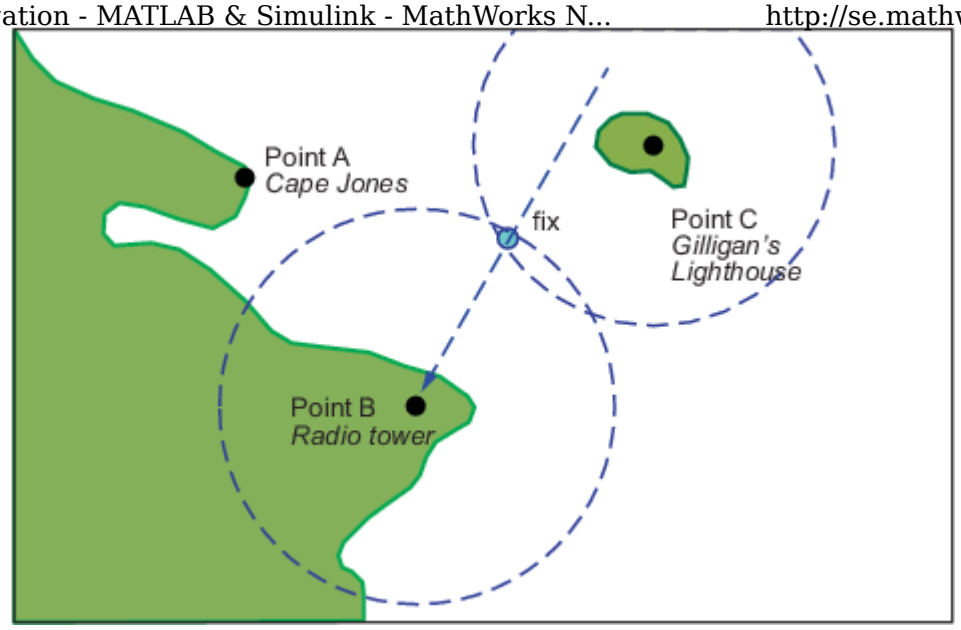
Luckily, your radar watch reports that he has Cape Jones at 18 nautical miles. This should resolve everything.



You were lucky this time. The third range resolved the ambiguity and gave you an excellent fix. Three intersections practically coincide. Sometimes the ambiguity is resolved, but the fix is still poor because the three closest intersections form a sort of circular triangle.

Sometimes the third range only adds to the confusion, either by bisecting the original two choices, or by failing to intersect one or both of the other arcs at all. In general, when *n* arcs are used, 2x(*n*-choose-2) possible intersections result. In this example, it is easy to tell which ones are *right*.

Bearing lines and arcs can be combined. If instead of reporting a third range, your radar watch had reported a bearing from the radar tower of 20º, the ambiguity could also have been resolved. Note, however, that in practice, lines of bearing for navigational fixing should only be taken visually, except in desperation. A radar's beam width can be a degree or more, leading to uncertainty.



As you begin to wonder whether this manual plotting process could be automated, your first officer shows up on the bridge with a laptop and Mapping Toolbox software.

Using navfix

The `navfix` function can be used to determine the points of intersection among any number of lines and arcs. Be warned, however, that due to the combinatorial nature of this process, the computation time grows rapidly with the number of objects. To illustrate this function, assign positions to the landmarks. Point A, Cape Jones, is at (`latA`,`lonA`). Point B, the radio tower, is at (`latB`,`lonB`). Point C, Gilligan's Lighthouse, is at (`latC`,`lonC`).

For the bearing-lines-only example, the syntax is:

```
[latfix,lonfix] = navfix([latA latB latC],[lonA lonB lonC],...
                         [300 270 0])
```

This defines the three points and their bearings as taken *from the ship*. The outputs would look something like this, with actual numbers, of course:

```
latfix =
        latfix1       NaN          % A intersecting B
        latfix2       NaN          % A intersecting C
        latfix3       NaN          % B intersecting C
lonfix =
        lonfix1       NaN          % A intersecting B
        lonfix2       NaN          % A intersecting C
        lonfix3       NaN          % B intersecting C
```

Notice that these are two-column matrices. The second column consists of `NaN`s because it is used only for the two-intersection ambiguity associated with arcs.

For the range-arcs-only example, the syntax is

```
[latfix,lonfix] = navfix([latA latB latC],[lonA lonB lonC],...
                         [16 14 15],[0 0 0])
```

This defines the three points and their ranges as taken from the ship. The final argument indicates that the three cases are all ranges.

The outputs have the following form:

```
latfix =
        latfix11  latfix12          % A intersecting B
        latfix21  latfix22          % A intersecting C
        latfix31  latfix32          % B intersecting C
lonfix =
        lonfix11  lonfix12          % A intersecting B
        lonfix21  lonfix22          % A intersecting C
        lonfix31  lonfix32          % B intersecting C
```

Here, the second column is used, because each pair of arcs has two potential intersections.

For the bearings and ranges example, the syntax requires the final input to indicate which objects are lines of bearing (indicated with a 1) and which are range arcs (indicated with a 0):

```
[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                         [20 14 15],[1 0 0])
```

The resulting output is mixed:

```
latfix =
        latfix11       NaN          % Line B intersecting Arc B
        latfix21  latfix22          % Line B intersecting Arc C
        latfix31  latfix32          % Arc B intersecting Arc C
lonfix =
        lonfix11       NaN          % Line B intersecting Arc B
        lonfix21  lonfix22          % Line B intersecting Arc C
        lonfix31  lonfix32          % Arc B intersecting Arc C
```

Only one intersection is returned for the line from B with the arc about B, since the line originates inside the circle and intersects it once. The same line intersects the other circle twice, and hence it returns two points. The two circles taken together also return two points.

Usually, you have an idea as to where you are before you take the fix. For example, you might have a dead reckoning position for the time of the fix (see below). If you provide `navfix` with this estimated position, it chooses from each pair of ambiguous intersections the point closest to the estimate. Here's what it might look like:

```
[latfix,lonfix] = navfix([latB latB latC],[lonB lonB lonC],...
                         [20 14 15],[1 0 0],drlat,drlon)
latfix =
        latfix11                    % the only point
        latfix21                    % the closer point
        latfix31                    % the closer point
lonfix =
        lonfix11                    % the only point
        lonfix21                    % the closer point
        lonfix31                    % the closer point
```

A Numerical Example of Using navfix

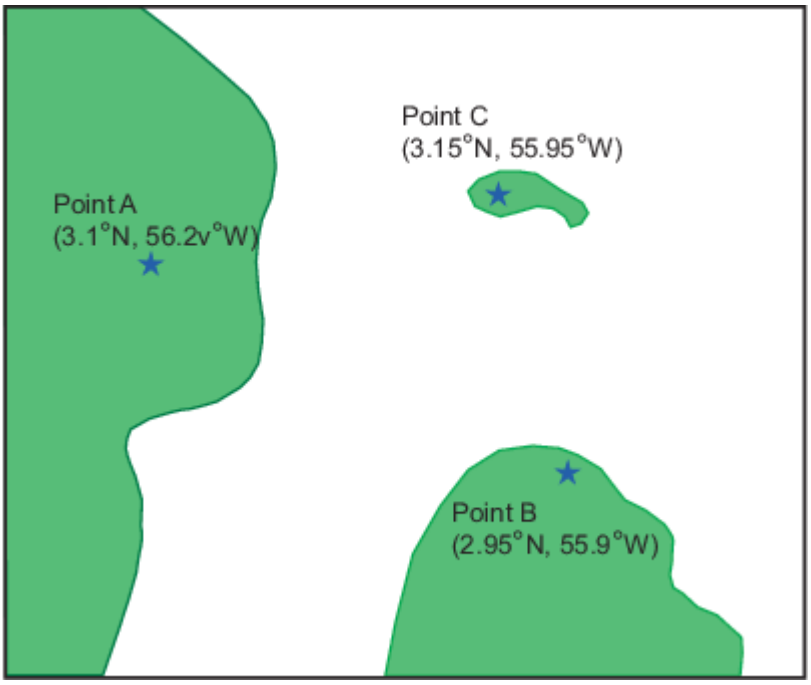
1. Define some specific points in the middle of the Atlantic Ocean. These are strictly arbitrary; perhaps they correspond to points in Atlantis:

```
lata = 3.1;  lona = -56.2;
latb = 2.95; lonb = -55.9;
latc = 3.15; lonc = -55.95;
```

2. Plot them on a Mercator projection:

```
axesm('MapProjection','mercator','Frame','on',...
        'MapLatLimit',[2.8 3.3],'MapLonLimit',[-56.3 -55.8])
plotm([lata latb latc],[lona lonb lonc],...
        'LineStyle','none','Marker','pentagram',...
        'MarkerEdgeColor','b','MarkerFaceColor','b',...
        'MarkerSize',12)
```

Here is what it looks like (with labeling and imaginary coastlines added after the fact for illustration):

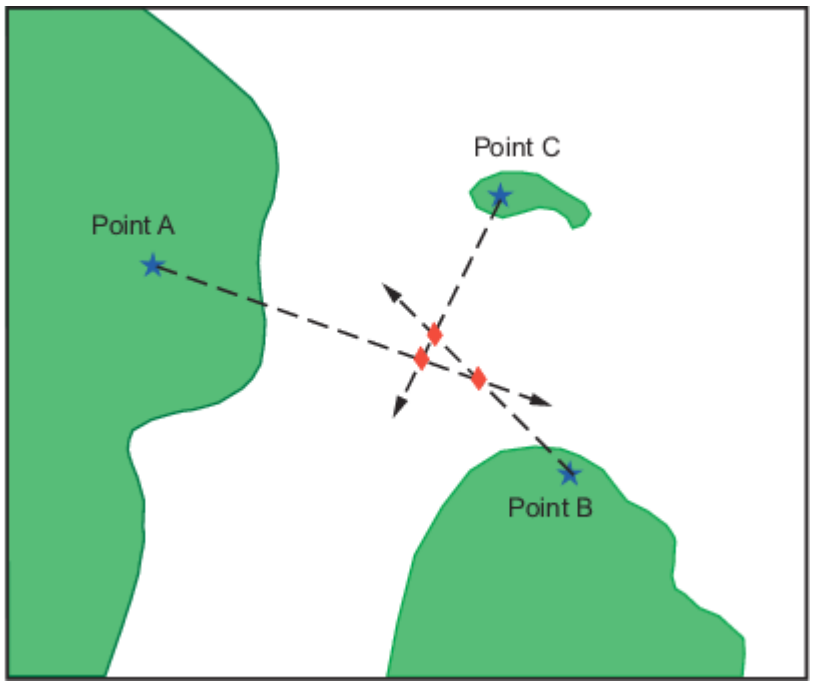


3. Take three visual bearings: Point A bears 289º, Point B bears 135º, and Point C bears 026.5º. Calculate the intersections:

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [289 135 26.5],[1 1 1])
newlat =
        3.0214       NaN
        3.0340       NaN
        3.0499       NaN
newlong =
        -55.9715       NaN
        -56.0079       NaN
        -56.0000       NaN
```

4. Add the intersection points to the map:

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

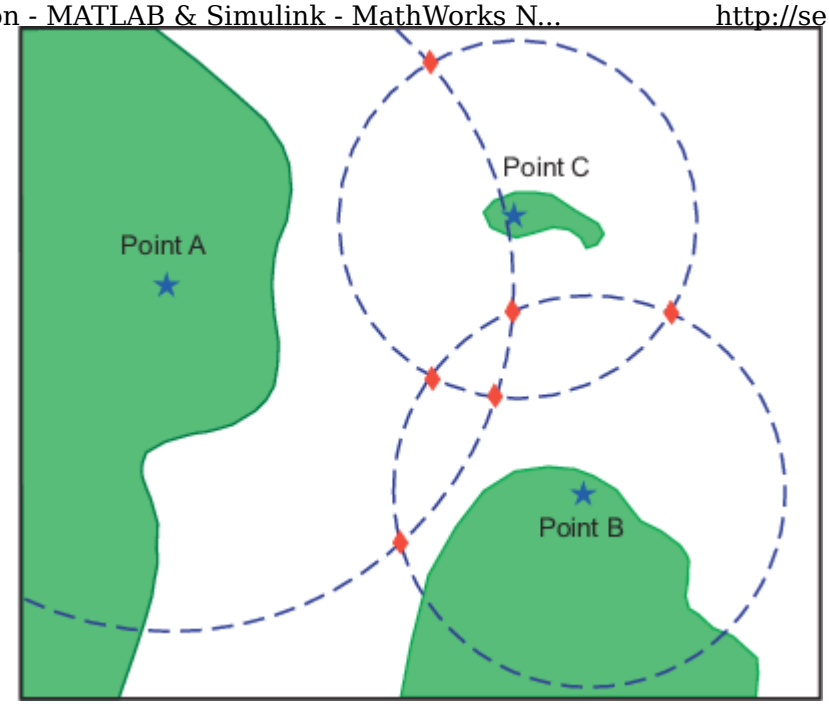


Bearing lines have been added to the map for illustration purposes. Notice that each pair of objects results in only one intersection, since all are lines of bearing.

5. What if instead, you had ranges from the three points, A, B, and C, of 13 nmi, 9 nmi, and 7.5 nmi, respectively?

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [13 9 7.5],[0 0 0])
newlat =
        3.0739    2.9434
        3.2413    3.0329
        3.0443    3.0880
newlong =
        -55.9846  -56.0501
        -56.0355  -55.9937
        -56.0168  -55.8413
```

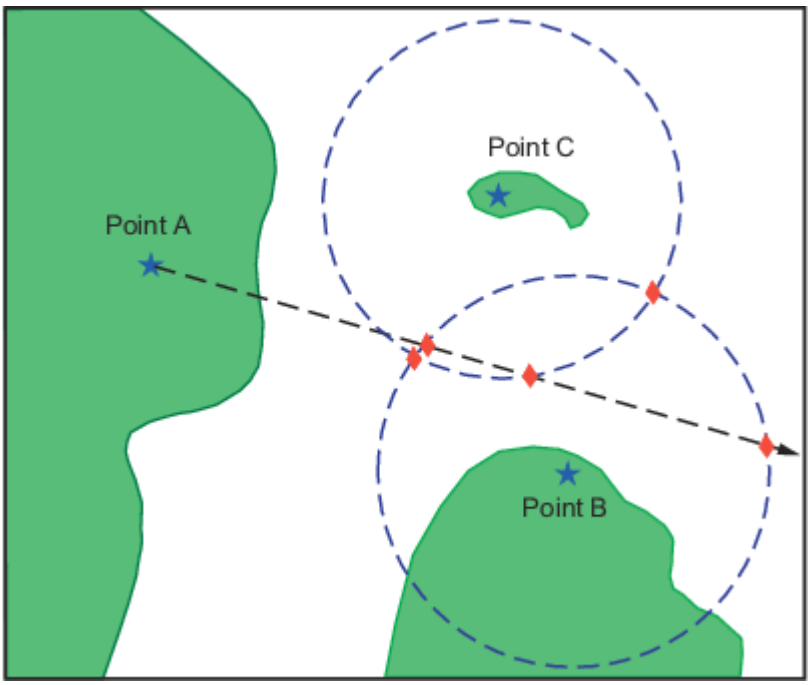
Here's what these points look like:



Three of these points look reasonable, three do not.

6. What if, instead of a range from Point A, you had a bearing to it of 284º?

```
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [284 9 7.5],[1 0 0])
newlat =
        3.0526    2.9892
        3.0592    3.0295
        3.0443    3.0880
newlong =
       -56.0096  -55.7550
       -56.0360  -55.9168
       -56.0168  -55.8413
```



Again, visual inspection of the results indicates which three of the six possible points seem like *reasonable* positions.

7. When using the dead reckoning position (3.05ºN,56.0ºW), the closer, more reasonable candidate from each pair of intersecting objects is chosen:

```
drlat = 3.05; drlon = -56;
[newlat,newlong] = navfix([lata latb latc],[lona lonb lonc],...
                          [284 9 7.5],[1 0 0],drlat,drlon)
newlat =
        3.0526
        3.0592
        3.0443
newlong =
        -56.0096
        -56.0360
        -56.0168
```

Planning the Shortest Path

You know that the shortest path between two geographic points is a great circle. Sailors and aviators are interested in minimizing distance traveled, and hence time elapsed. You also know that the rhumb line is a path of constant heading, the *natural* means of traveling. In general, to follow a great circle path, you would have to continuously alter course. This is impractical. However, you can approximate a great circle path by rhumb line segments so that the added distance is minor and the number of course changes minimal.

Surprisingly, very few rhumb line *track legs* are required to closely approximate the distance of the great circle path.

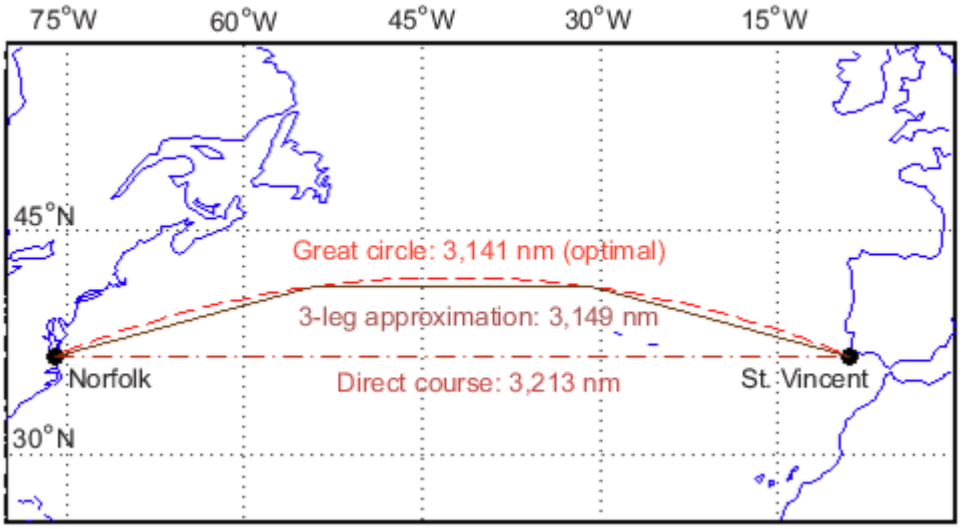
Consider the voyage from Norfolk, Virginia (37ºN,76ºW), to Cape St. Vincent, Portugal (37ºN,9ºW), one of the most heavily trafficked routes in the Atlantic. A due-east rhumb line track is 3,213 nautical miles, while the optimal great circle distance is 3,141 nautical miles.

Although the rhumb line path is only a little more than 2% longer, this is an additional 72 miles over the course of the trip. For a 12-knot tanker, this results in a 6-hour delay, and in shipping, time is money. If just three rhumb line segments are used to approximate the great circle, the total distance of the trip is 3,147 nautical miles. Our tanker would suffer only a half-hour delay compared to a continuous rhumb line course. Here is the code for computing the three types of tracks between Norfolk and St. Vincent:

```
figure('color','w');
ha = axesm('mapproj','mercator',...
           'maplatlim',[25 55],'maplonlim',[-80 0]);
axis off, gridm on, framem on;
setm(ha,'MLineLocation',15,'PLineLocation',15);
mlabel on, plabel on;
load coast;
hg = geoshow(lat,long,'displaytype','line','color','b');

% Define point locs for Norfolk, VA and St. Vincent Portugal
norfolk = [37,-76];
stvincent = [37, -9];
geoshow(norfolk(1),norfolk(2),'DisplayType','point',...
```

```
    'markeredgecolor','k','markerfacecolor','k','marker','o')
geoshow(stvincent(1),stvincent(2),'DisplayType','point',...
    'markeredgecolor','k','markerfacecolor','k','marker','o')

% Compute and draw 100 points for great circle
gcpts = track2('gc',norfolk(1),norfolk(2),...
               stvincent(1),stvincent(2));
geoshow(gcpts(:,1),gcpts(:,2),'DisplayType','line',...
    'color','red','linestyle','--')

% Compute and draw 100 points for rhumb line
rhpts = track2('rh',norfolk(1),norfolk(2),...
                stvincent(1),stvincent(2));
geoshow(rhpts(:,1),rhpts(:,2),'DisplayType','line',...
    'color',[.7 .1 0],'linestyle','-.')
[latpts,lonpts] = gcwaypts(norfolk(1),norfolk(2),...
    stvincent(1),stvincent(2),3);   % Compute 3 waypoints
geoshow(latpts,lonpts,'DisplayType','line',...
    'color',[.4 .2 0],'linestyle','-')
```

The resulting tracks and distances are shown below:



The Mapping Toolbox function `gcwaypts` calculates waypoints in navigation track format in order to approximate a great circle with rhumb line segments. It uses this syntax:

```
[latpts,lonpts] = gcwaypts(lat1,lon1,lat2,lon2,numlegs)
```

All the inputs for this function are scalars a (starting and an ending position). The `numlegs` input is the number of equal-length legs desired, which is 10 by default. The outputs are column vectors representing waypoints in navigational track format (`[heading distance]`). The size of each of these vectors is `[(numlegs+1) 1]`. Here are the points for this example:

```
[latpts,lonpts] = gcwaypts(norfolk(1),norfolk(2),...
stvincent(1),stvincent(2),3)   % Compute 3 waypoints
```

```
latpts =
   37.0000
   41.5076
   41.5076
   37.0000

lonpts =
  -76.0000
  -54.1777
  -30.8223
   -9.0000
```

These points represent waypoints along the great circle between which the approximating path follows rhumb lines. Four points are needed for three legs, because the final point at Cape St. Vincent must be included.

Now we can compute the distance in nautical miles (nm) along each track and via the waypoints:

```
drh = distance('rh',norfolk,stvincent); % Get rhumb line dist (deg)
dgc = distance('gc',norfolk,stvincent); % Get gt. circle dist (deg)
% Compute headings and distances for the waypoint legs
[course distnm] = legs(latpts,lonpts,'rh');
```

Finally, compare the distances:

```
distrhnm = deg2nm(drh)            % Nautical mi along rhumb line
distgcnm = deg2nm(dgc)            % Nautical mi along great circle
distlegsnm = sum(distnm)          % Total dist along the 3 legs
rhgcdiff = distrhnm - distgcnm    % Excess rhumb line distance
trgcdiff = distlegsnm - distgcnm  % Excess distance along legs

distrhnm =
  3.2127e+003

distgcnm =
  3.1407e+003

distlegsnm =
  3.1490e+003

rhgcdiff =
   71.9980

trgcdiff =
    8.3446
```

Following just three rhumb line legs reduces the distance travelled from 72 nm to 8.3 nm compared to a great circle course.

Track Laydown – Displaying Navigational Tracks

Navigational tracks are most useful when graphically displayed. Traditionally, the navigator identifies and plots waypoints on a Mercator projection and then connects them with a straightedge, which on this projection results in rhumb line tracks. In the previous example, waypoints were chosen to approximate a great circle route, but they can be selected for a variety of other reasons.

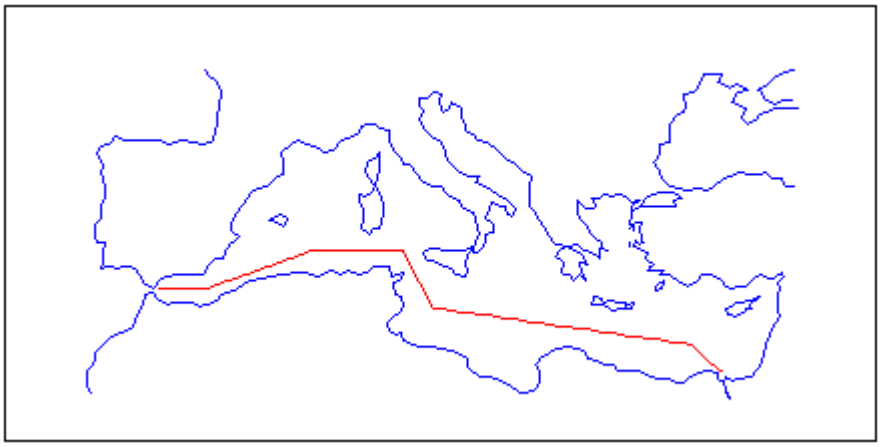
Let's say that after arriving at Cape St. Vincent, your tanker must traverse the Straits of Gibraltar and then travel on to Port Said, the northern terminus of the Suez Canal. On the scale of the Mediterranean Sea, following great circle paths is of little concern compared to ensuring that the many straits and passages are safely transited. The navigator selects appropriate waypoints and plots them.

To accomplish this with Mapping Toolbox functions, you can display a map axes with a Mercator projection, select appropriate map latitude and longitude limits to isolate the area of interest, plot coastline data, and interactively mouse-select the waypoints with the `inputm` function. The `track` function will generate points to connect these waypoints, which can then be displayed with `plotm`.

For illustration, assume that the waypoints are known (or were gathered using `inputm`). To learn about using **`inputm`**, see **Picking Locations Interactively**, or **`inputm`** in the Mapping Toolbox reference pages.

```
waypoints = [36 -5; 36 -2; 38 5; 38 11; 35 13; 33 30; 31.5 32]
waypoints =
        36.0000   -5.0000
        36.0000   -2.0000
        38.0000    5.0000
        38.0000   11.0000
        35.0000   13.0000
        33.0000   30.0000
        31.5000   32.0000
load coast
axesm('MapProjection','mercator',...
'MapLatLimit',[30 47],'MapLonLimit',[-10 37])
framem
plotm(lat,long)

[lttrk,lntrk] = track(waypoints);
plotm(lttrk,lntrk,'r')
```

Although these track segments are straight lines on the Mercator projection, they are curves on others:



The segments of a track like this are called *legs*. Each of these legs can be described in terms of course and distance. The function `legs` will take the waypoints in navigational track format and return the course and distance required for each leg. Remember, the order of the points in this format determines the direction of travel. Courses are therefore calculated from each waypoint to its successor, not the reverse.

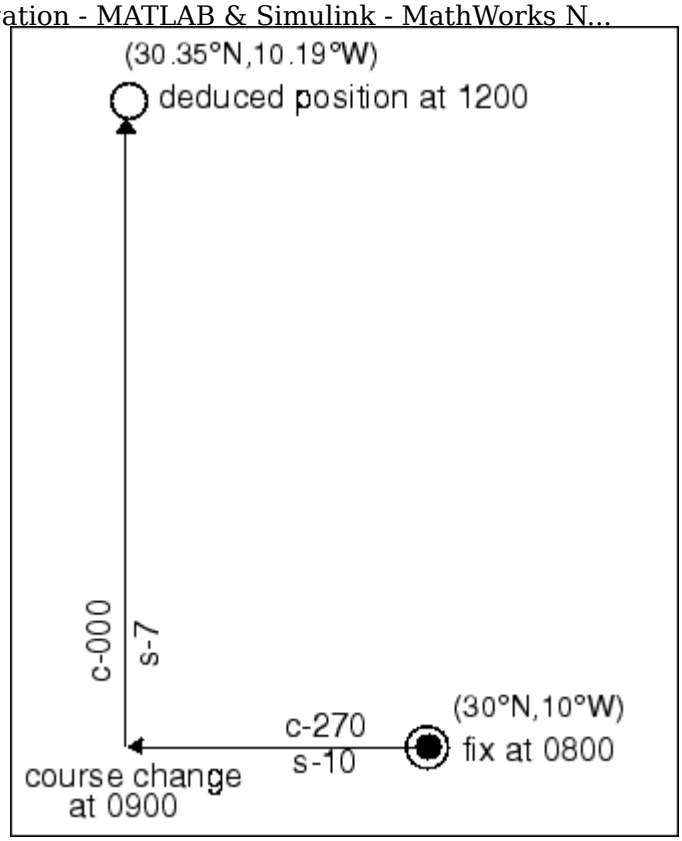
```
[courses,distances] = legs(waypoints)
courses =
        90.0000
        70.3132
        90.0000
       151.8186
        98.0776
       131.5684
distances =
       145.6231
       356.2117
       283.6839
       204.2073
       854.0092
       135.6415
```

Since this is a navigation function, the courses are all in degrees and the distances are in nautical miles. From these distances, speeds required to arrive at Port Said at a given time can be calculated. Southbound traffic is allowed to enter the canal only once per day, so this information might be economically significant, since unnecessarily high speeds can lead to high fuel costs.

Dead Reckoning

When sailors first ventured out of sight of land, they faced a daunting dilemma. How could they find their way home if they didn't know where they were? The practice of *dead reckoning* is an attempt to deal with this problem. The term is derived from *deduced reckoning*.

Briefly, dead reckoning is vector addition plotted on a chart. For example, if you have a fix at (30ºN,10ºW) at 0800, and you proceed due west for 1 hour at 10 knots, and then you turn north and sail for 3 hours at 7 knots, you should be at (30.35ºN,10.19ºW) at 1200.



However, a sailor *shoots the sun* at local apparent noon and discovers that the ship's latitude is actually 30.29ºN. What's worse, he lives before the invention of a reliable chronometer, and so he cannot calculate his longitude at all from this sighting. What happened?

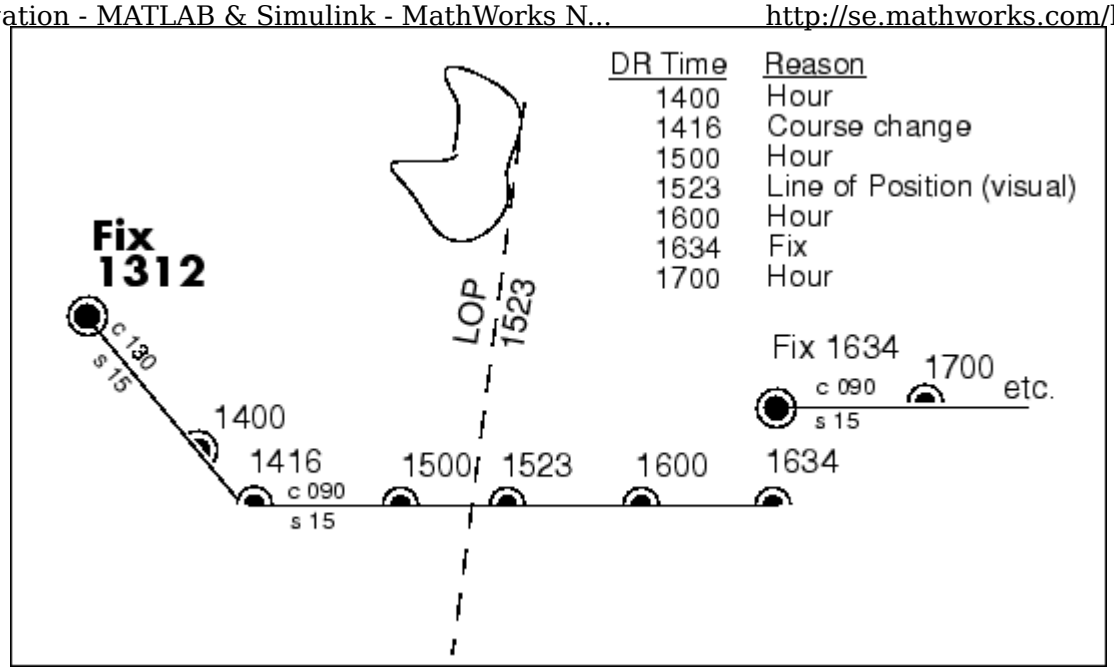
Leaving aside the difficulties in speed determination and the need to tack off course, even modern craft have to contend with winds and currents. However, despite these limitations, dead reckoning is still used for determining position between fixes and for forecasting future positions. This is because dead reckoning provides a certainty of assumptions that estimations of wind and current drift cannot.

When navigators establish a fix from some source, be it from piloting, celestial, or satellite observations, they plot a dead reckoning (DR) track, which is a plot of the intended positions of the ship forward in time. In practice, dead reckoning is usually plotted for 3 hours in advance, or for the time period covered by the next three expected fixes. In open ocean conditions, hourly fixes are sufficient; in coastal pilotage, three-minute fixes are common.

Specific DR positions, which are sometimes called *DR*s, are plotted according to the *Rules of DR*:

- DR at every course change
- DR at every speed change
- DR every hour on the hour
- DR every time a fix or running fix is obtained
- DR 3 hours ahead or for the next three expected fixes
- DR for every line of position (LOP), either visual or celestial

For example, the navigator plots these DRs:



Notice that the 1523 DR does not coincide with the LOP at 1523. Although note is taken of this variance, one line is insufficient to calculate a new fix.

Mapping Toolbox function `dreckon` calculates the DR positions for a given set of courses and speeds. The function provides DR positions for the first three rules of dead reckoning. The approach is to provide a set of waypoints in navigational track format corresponding to the plan of intended movement.

The time of the initial waypoint, or fix, is also needed, as well as the speeds to be employed along each leg. Alternatively, a set of speeds and the times for which each speed will apply can be provided. `dreckon` returns the positions and times required of these DRs:

- `dreckon` calculates the times for position of each course change, which will occur at the waypoints
- `dreckon` calculates the positions for each whole hour
- If times are provided for speed changes, `dreckon` calculates positions for these times if they do not occur at course changes

Imagine you have a fix at midnight at the point (10ºN,0º):

```
waypoints(1,:) = [10 0]; fixtime = 0;
```

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

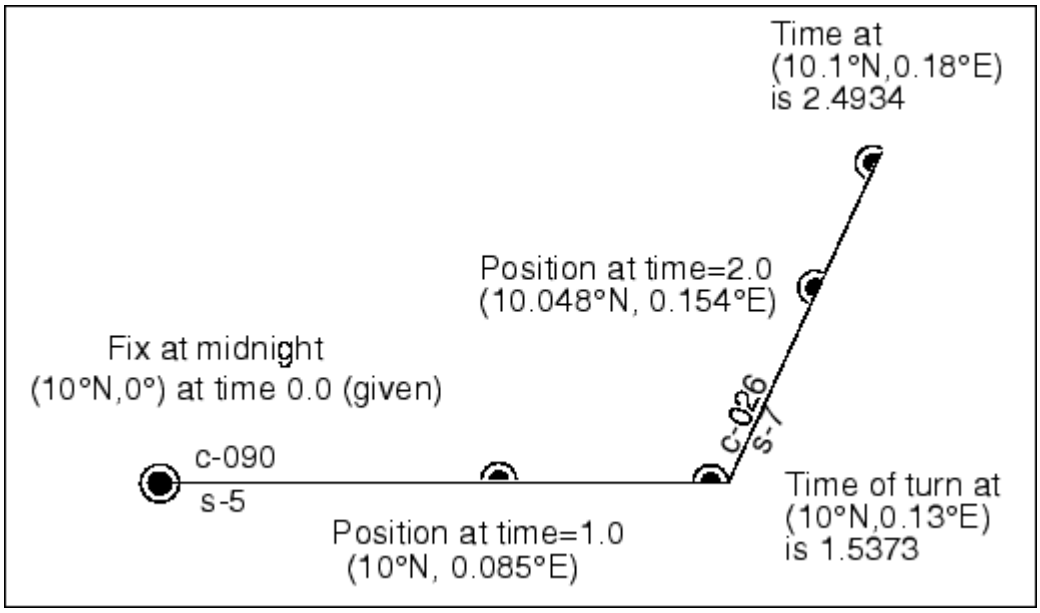
```
waypoints(2,:) = [10 .13];
waypoints(3,:) = [10.1 .18];
speeds = [5;7];
```

To determine the DR points and times for this plan, use `dreckon`:

```
[drlat,drlon,drtime] = dreckon(waypoints,fixtime,speeds);
[drlat drlon drtime]
ans =
       10.0000    0.0846    1.0000     % Position at 1 am
       10.0000    0.1301    1.5373     % Time of course change
       10.0484    0.1543    2.0000     % Position at 2 am
```

```
        10.1001    0.1801    2.4934     % Time at final waypoint
```

Here is an illustration of this track and its DR points:

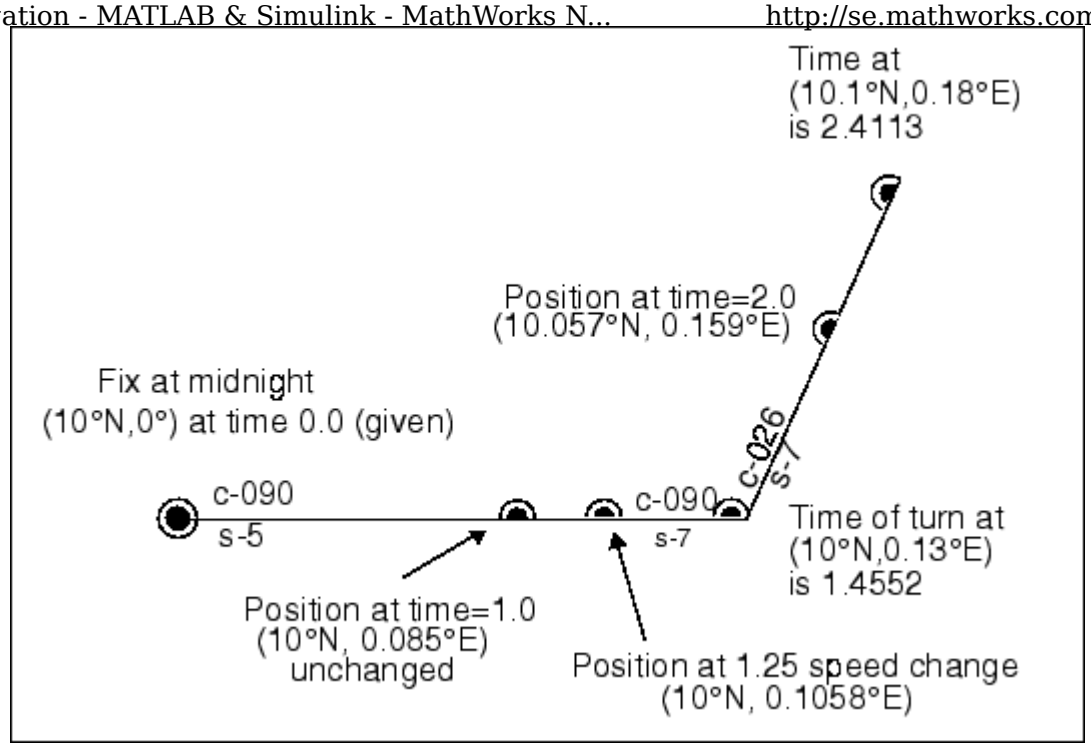


However, you would like to get to the final point a little earlier to make a rendezvous. You decide to recalculate your DRs based on speeding up to 7 knots a little earlier than planned. The first calculation tells you that you were going to increase speed at the turn, which would occur at a time 1.5373 hours after midnight, or 1:32 a.m. (at time 0132 in navigational time format). What time would you reach the rendezvous if you increased your speed to 7 knots at 1:15 a.m. (0115, or 1.25 hours after midnight)?

To indicate times for speed changes, another input is required, providing a time interval after the fix time at which each ordered speed is to end. The first speed, 5 knots, is to end 1.25 hours after midnight. Since you don't know when the rendezvous will be made under these circumstances, set the time for the second speed, 7 knots, to end at infinity. No DRs will be returned past the last waypoint.

```
spdtimes = [1.25; inf];
[drlat,drlon,drtime] = dreckon(waypoints,fixtime,...
                               speeds,spdtimes);
[drlat,drlon,drtime]
ans =
        10.0000    0.0846    1.0000   % Position at 1 am
        10.0000    0.1058    1.2500   % Position at speed change
        10.0000    0.1301    1.4552   % Time of course change
        10.0570    0.1586    2.0000   % Position at 2 am
        10.1001    0.1801    2.4113   % Time at final waypoint
```

This following illustration shows the difference:

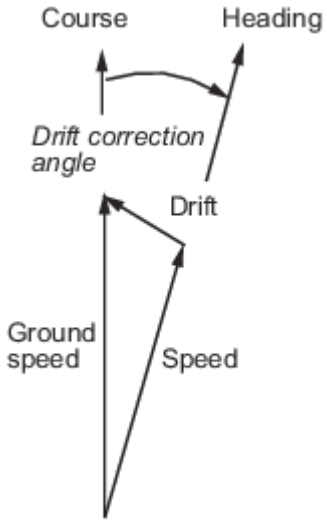


The times at planned positions after the speed change are a little earlier; the position at the known time (2 a.m.) is a little farther along. With this plan, you will arrive at the rendezvous about 4 1/2 minutes earlier, so you may want to consider a greater speed change.

Drift Correction

Dead reckoning is a reasonably accurate method for predicting position if the vehicle is able to maintain the planned course. Aircraft and ships can be pushed off the planned course by winds and current. An important step in navigational planning is to calculate the required drift correction.

In the standard drift correction problem, the desired course and wind are known, but the heading needed to stay on course is unknown. This problem is well suited to vector analysis. The wind velocity is a vector of known magnitude and direction. The vehicle's speed relative to the moving air mass is a vector of known magnitude, but unknown direction. This heading must be chosen so that the sum of the vehicle and wind velocities gives a resultant in the specified course direction. The ground speed can be larger or smaller than the air speed because of headwind or tailwind components. A navigator would like to know the required heading, the associated wind correction angle, and the resulting ground speed.



What heading puts an aircraft on a course of 250° when the wind is 38 knots from 285°? The aircraft flies at an

airspeed of 145 knots.

```
course = 250; airspeed = 145; windfrom = 285; windspeed = 38;
[heading,groundspeed,windcorrangle] = ...
driftcorr(course,airspeed,windfrom,windspeed)

heading =
        258.65

groundspeed =
        112.22

windcorrangle =
          8.65
```

The required heading is about 9° to the right of the course. There is a 33-knot headwind component.

A related problem is the calculation of the wind speed and direction from observed heading and course. The wind velocity is just the vector difference of the ground speed and the velocity relative to the air mass.

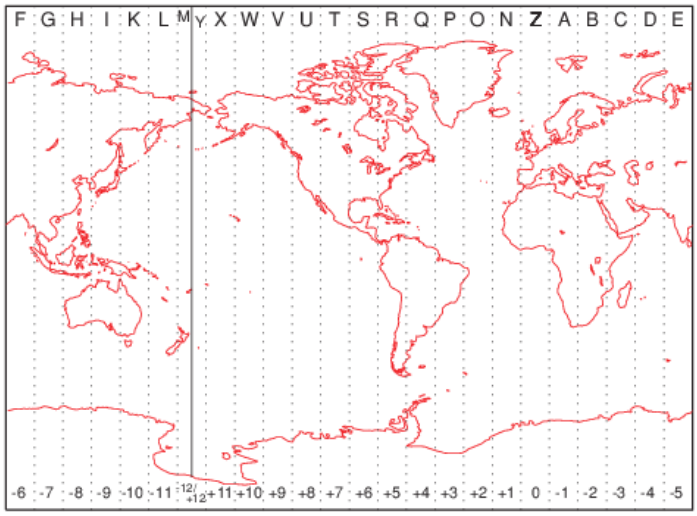
```
[windfrom,windspeed] = ...
driftvel(course,groundspeed,heading,airspeed)

windfrom =
        285.00

windspeed =
         38.00
```

Time Zones

Time zones used for navigation are uniform 15º extents of longitude. The `timezone` function returns a navigational time zone, that is, one based solely on longitude with no regard for statutory divisions. So, for example, Chicago, Illinois, lies in the statutory U.S. Central time zone, which has irregular boundaries devised for political or convenience reasons. However, from a navigational standpoint, Chicago's longitude places it in the *S* (Sierra) time zone. The zone's *description* is +6, which indicates that 6 hours must be added to local time to get Greenwich, or *Z* (Zulu) time. So, if it is noon, standard time in Chicago, it is 12+6, or 6 p.m., at Greenwich.

Each 15º navigational time zone has a distinct description and designating letter. The exceptions to this are the two zones on either side of the date line, *M* and *Y* (Mike and Yankee). These zones are only 7-1/2º wide, since on one side of the date line, the description is +12, and on the other, it is -12.

Navigational time zones are very important for celestial navigation calculations. Although there are no Mapping Toolbox functions designed specifically for celestial navigation, a simple example can be devised.



It is possible with a sextant to determine *local apparent noon*. This is the moment when the Sun is at its zenith from your point of view. At the exact center longitude of a time zone, the phenomenon occurs exactly at noon, local time. Since the Sun traverses a 15º time zone in 1 hour, it crosses one degree every 4 minutes. So if you observe local apparent noon at 11:54, you must be 1.5º east of your center longitude.

You must know what time zone you are in before you can even attempt a fix. This concept has been understood since the spherical nature of the Earth was first accepted, but early sailors had no ability to keep accurate time on ship, and so were unable to determine their longitude. The invention of accurate chronometers in the 18th century solved this problem.

The `timezone` function is quite simple. It returns the description, `zd`, an integer for use in calculations, a string, `zltr`, of the zone designator, and a string fully naming the `zone`. For example, the information for a longitude 123ºE is the following:

```
[zd,zltr,zone] = timezone(123)
zd =
        -8
zltr =
H
zone =
        -8 H
```

Returning to the simple celestial navigation example, the center longitude of this zone is:

```
-(zd*15)
ans =
        120
```

This means that at our longitude, 123ºE, we should experience local apparent noon at 11:48 a.m., 12 minutes early.