Aerospace Toolbox

User's Guide

MATLAB®

R2019b

MathWorks®

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

Aerospace Toolbox User's Guide

Revision History

September 2006	Online only	New for Version 1.0 (Release 2006b)
March 2007	Online only	Revised for Version 1.1 (Release 2007a)
September 2007	First printing	Revised for Version 2.0 (Release 2007b)
March 2008	Online only	Revised for Version 2.1 (Release 2008a)
October 2008	Online only	Revised for Version 2.2 (Release 2008b)
March 2009	Online only	Revised for Version 2.3 (Release 2009a)
September 2009	Online only	Revised for Version 2.4 (Release 2009b)
March 2010	Online only	Revised for Version 2.5 (Release 2010a)
September 2010	Online only	Revised for Version 2.6 (Release 2010b)
April 2011	Online only	Revised for Version 2.7 (Release 2011a)
September 2011	Online only	Revised for Version 2.8 (Release 2011b)
March 2012	Online only	Revised for Version 2.9 (Release 2012a)
September 2012	Online only	Revised for Version 2.10 (Release 2012b)
March 2013	Online only	Revised for Version 2.11 (Release 2013a)
September 2013	Online only	Revised for Version 2.12 (Release 2013b)
March 2014	Online only	Revised for Version 2.13 (Release 2014a)
October 2014	Online only	Revised for Version 2.14 (Release 2014b)
March 2015	Online only	Revised for Version 2.15 (Release 2015a)
September 2015	Online only	Revised for Version 2.16 (Release 2015b)
March 2016	Online only	Revised for Version 2.17 (Release 2016a)
September 2016	Online only	Revised for Version 2.18 (Release 2016b)
March 2017	Online only	Revised for Version 2.19 (Release 2017a)
September 2017	Online only	Revised for Version 2.20 (Release 2017b)
March 2018	Online only	Revised for Version 2.21 (Release 2018a)
September 2018	Online only	Revised for Version 3.0 (Release 2018b)
March 2019	Online only	Revised for Version 3.1 (Release 2019a)
September 2019	Online only	Revised for Version 3.2 (Release 2019b)

Contents

4 Alphabetical List

5 Aerospace Toolbox Examples

Getting Started

Aerospace Toolbox Product Description

Analyze and visualize aerospace vehicle motion using reference standards and models

Aerospace Toolbox provides tools and functions for analyzing the navigation and environment of aerospace vehicles and visualizing their flight using standard cockpit instruments or a flight simulator. It lets you import Data Compendium (Datcom) files directly into MATLAB® to represent vehicle aerodynamics and incorporate validated environment models for atmosphere, gravity, wind, geoid height, and magnetic field. You can evaluate vehicle motion and orientation using built-in aerospace math operations and coordinate system and spatial transformations. You can visualize the vehicle in flight directly from MATLAB with standard cockpit instruments and using the pre-built FlightGear Flight Simulator interface.

Key Features

- Built-in axis transformations for body, wind, and geodetic coordinates, including earth-centered / earth-fixed (ECEF) and earth-centered inertial (ECI)
- Validated environment models, including gravity, atmosphere, wind, geoid height, magnetic field, and planetary ephemerides
- 3D animations and flight instruments for visualizing flight data
- Gas dynamics functions for calculating flow ratios, relations, and Prandtl-Meyer expansion waves
- Aerodynamic coefficients from Data Compendium (Datcom) files

Aerospace Toolbox and Aerospace Blockset

The Aerospace product family includes the Aerospace Toolbox and Aerospace Blockset products. The toolbox provides static data analysis capabilities, while the blockset provides an environment for dynamic modeling and vehicle component modeling and simulation. The Aerospace Blockset software uses part of the functionality of the toolbox as an engine. Use these products together to model aerospace systems in the MATLAB and Simulink® environments.

Using Aerospace Toolbox

Fundamental Coordinate System Concepts

Coordinate systems allow you to track an aircraft or spacecraft position and orientation in space. The Aerospace Toolbox coordinate systems are based on these underlying concepts from geodesy, astronomy, and physics.

Definitions

The Aerospace Toolbox software uses right-handed (RH) Cartesian coordinate systems. The rightmost rule establishes the *x*-*y*-*z* sequence of coordinate axes.

An inertial frame is a nonaccelerating motion reference frame. Loosely speaking, acceleration is defined with respect to the distant cosmos. In an inertial frame, Newton's second law (force = mass X acceleration) holds.

Strictly defined, an inertial frame is a member of the set of all frames not accelerating relative to one another. A noninertial frame is any frame accelerating relative to an inertial frame. Its acceleration, in general, includes both translational and rotational components, resulting in pseudoforces (pseudogravity, as well as Coriolis and centrifugal forces).

The toolbox models the Earth shape (the geoid) as an oblate spheroid, a special type of ellipsoid with two longer axes equal (defining the equatorial plane) and a third, slightly shorter (geopolar) axis of symmetry. The equator is the intersection of the equatorial plane and the Earth surface. The geographic poles are the intersection of the Earth surface and the geopolar axis. In general, the Earth geopolar and rotation axes are not identical.

Latitudes parallel the equator. Longitudes parallel the geopolar axis. The zero longitude or prime meridian passes through Greenwich, England.

Approximations

The Aerospace Toolbox software makes three standard approximations in defining coordinate systems relative to the Earth.

- The Earth surface or geoid is an oblate spheroid, defined by its longer equatorial, and defined by its longer equatorial and shorter geopolar axes. In reality, the Earth is slightly deformed with respect to the standard geoid.

- The Earth rotation axis and equatorial plane are perpendicular, so that the rotation and geopolar axes are identical. In reality, these axes are slightly misaligned, and the equatorial plane wobbles as the Earth rotates. This effect is negligible in most applications.
- The only noninertial effect in Earth-fixed coordinates is due to the Earth rotation about its axis. This is a *rotating*, *geocentric* system. The toolbox ignores the Earth motion around the Sun, the Sun motion in the Galaxy, and the Galaxy's motion through cosmos. In most applications, only the Earth rotation matters.

  This approximation must be changed for spacecraft sent into deep space, that is, outside the Earth-Moon system, and a heliocentric system is preferred.

Motion with Respect to Other Planets

The Aerospace Toolbox software uses the standard WGS-84 geoid to model the Earth. You can change the equatorial axis length, the flattening, and the rotation rate.

You can represent the motion of spacecraft with respect to any celestial body that is well approximated by an oblate spheroid by changing the spheroid size, flattening, and rotation rate. If the celestial body is rotating westward (retrogradely), make the rotation rate negative.

References

Recommended Practice for Atmospheric and Space Flight Vehicle Coordinate Systems, R-004-1992, ANSI/AIAA, February 1992.

Mapping Toolbox™ documentation, The MathWorks, Inc., Natick, Massachusetts. “Mapping Toolbox”.

Rogers, R. M., *Applied Mathematics in Integrated Navigation Systems*, AIAA, Reston, Virginia, 2000.

Stevens, B. L., and F. L. Lewis, *Aircraft Control, and Simulation*, 2nd ed., Wiley-Interscience, New York, 2003.

Thomson, W. T., *Introduction to Space Dynamics*, John Wiley & Sons, New York, 1961/ Dover Publications, Mineola, New York, 1986.

World Geodetic System 1984 (WGS 84), `http://earth-info.nga.mil/GandG/wgs84`.

Coordinate Systems for Modeling

Modeling aircraft and spacecraft are simplest if you use a coordinate system fixed in the body itself. In the case of aircraft, the forward direction is modified by the presence of wind, and the craft's motion through the air is not the same as its motion relative to the ground.

Body Coordinates

The noninertial body coordinate system is fixed in both origin and orientation to the moving craft. The craft is assumed to be rigid.

The orientation of the body coordinate axes is fixed in the shape of body.

- The *x*-axis points through the nose of the craft.
- The *y*-axis points to the right of the *x*-axis (facing in the pilot's direction of view), perpendicular to the *x*-axis.
- The *z*-axis points down through the bottom of the craft, perpendicular to the *x*-*y* plane and satisfying the RH rule.

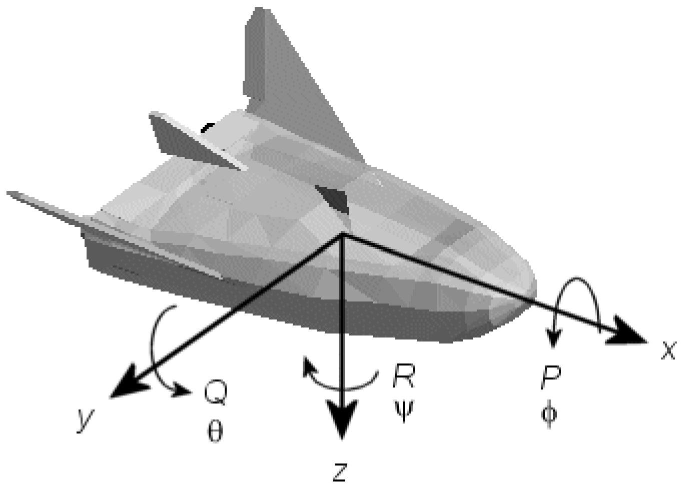
Translational Degrees of Freedom

Translations are defined by moving along these axes by distances *x*, *y*, and *z* from the origin.

Rotational Degrees of Freedom

Rotations are defined by the Euler angles *P*, *Q*, *R* or Φ, Θ, Ψ. They are

- *P* or Φ: Roll about the *x*-axis
- *Q* or Θ: Pitch about the *y*-axis
- *R* or Ψ: Yaw about the *z*-axis



Unless otherwise specified, by default the software uses ZYX rotation order for Euler angles.

Wind Coordinates

The noninertial wind coordinate system has its origin fixed in the rigid aircraft. The coordinate system orientation is defined relative to the craft's velocity V.

The orientation of the wind coordinate axes is fixed by the velocity V.

- The *x*-axis points in the direction of V.
- The *y*-axis points to the right of the *x*-axis (facing in the direction of V), perpendicular to the *x*-axis.
- The *z*-axis points perpendicular to the *x*-*y* plane in whatever way needed to satisfy the RH rule with respect to the *x*- and *y*-axes.

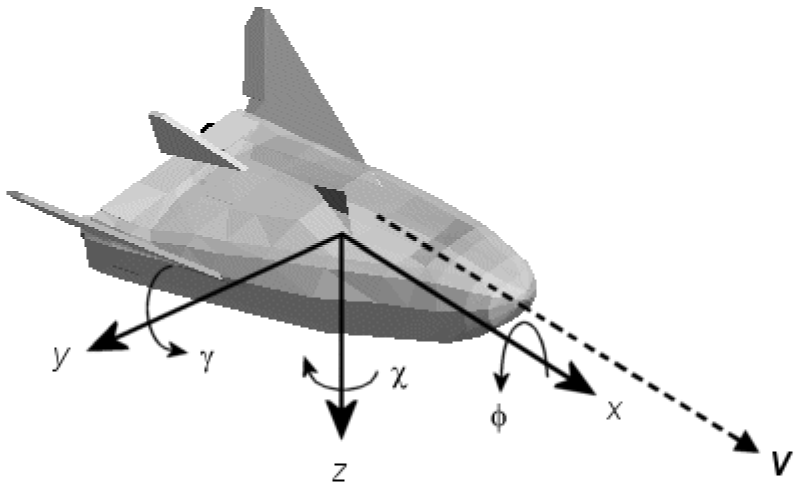
Translational Degrees of Freedom

Translations are defined by moving along these axes by distances *x*, *y*, and *z* from the origin.

Rotational Degrees of Freedom

Rotations are defined by the Euler angles Φ, γ, χ. They are

- Φ: Bank angle about the *x*-axis
- γ: Flight path about the *y*-axis
- χ: Heading angle about the *z*-axis



Unless otherwise specified, by default the software uses ZYX rotation order for Euler angles.

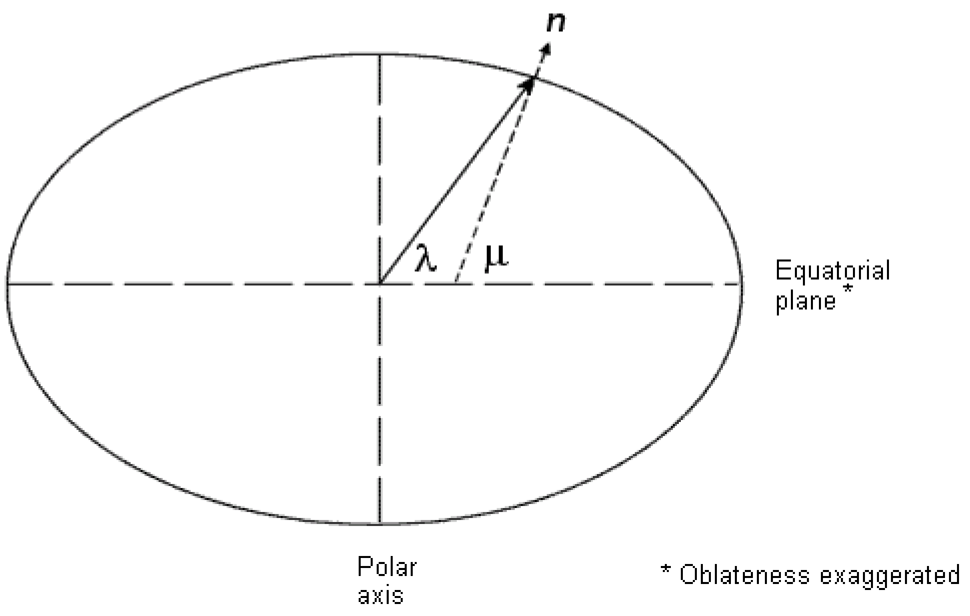
Coordinate Systems for Navigation

Modeling aerospace trajectories requires positioning and orienting the aircraft or spacecraft with respect to the rotating Earth. Navigation coordinates are defined with respect to the center and surface of the Earth.

Geocentric and Geodetic Latitudes

The geocentric latitude λ on the Earth surface is defined by the angle subtended by the radius vector from the Earth center to the surface point with the equatorial plane.

The geodetic latitude μ on the Earth surface is defined by the angle subtended by the surface normal vector *n* and the equatorial plane.

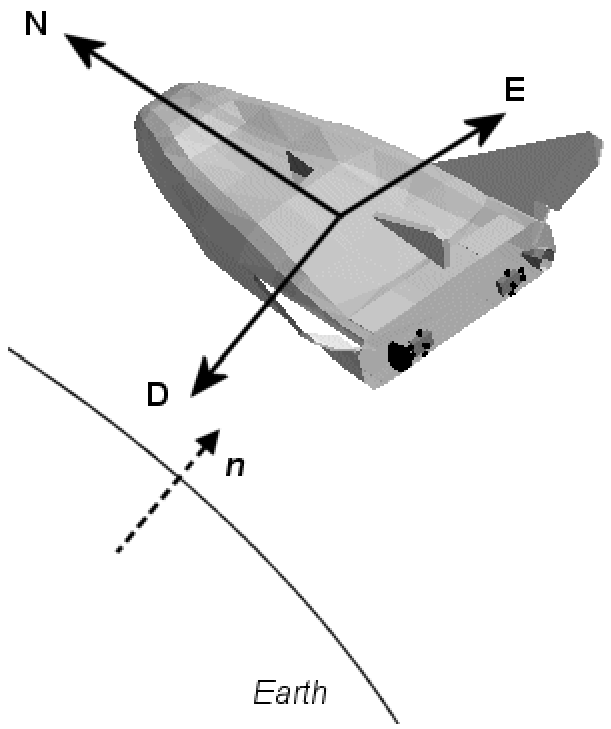


NED Coordinates

The north-east-down (NED) system is a noninertial system with its origin fixed at the aircraft or spacecraft's center of gravity. Its axes are oriented along the geodetic directions defined by the Earth surface.

- The *x*-axis points north parallel to the geoid surface, in the polar direction.
- The *y*-axis points east parallel to the geoid surface, along a latitude curve.
- The *z*-axis points downward, toward the Earth surface, antiparallel to the surface's outward normal *n*.

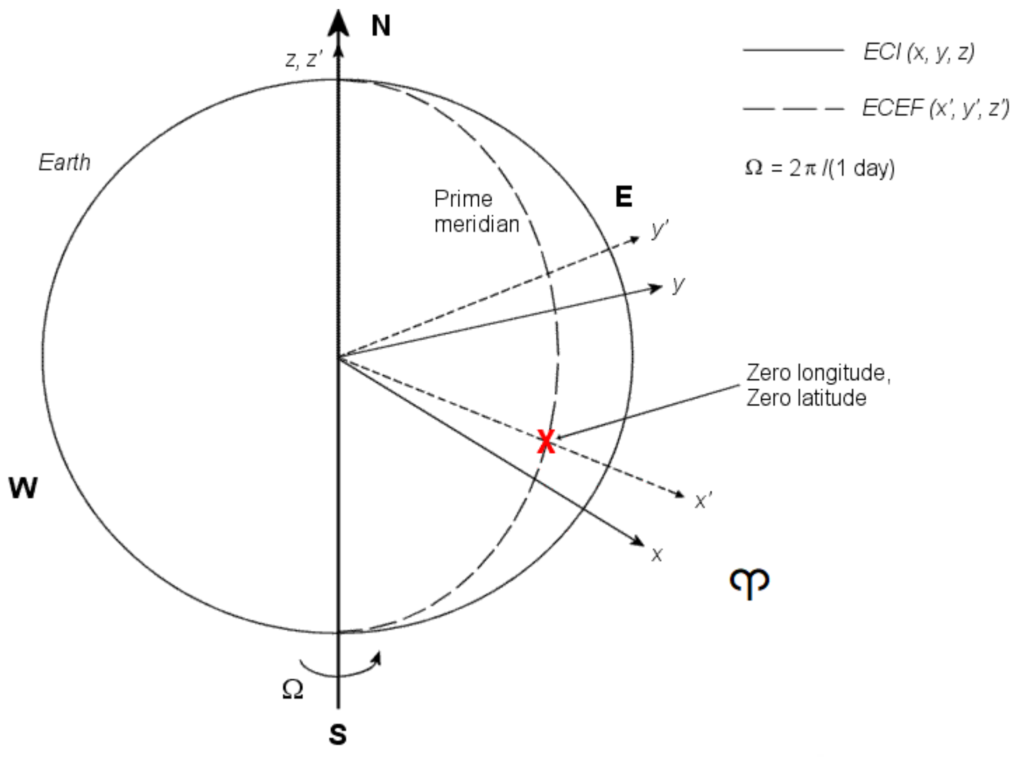
  Flying at a constant altitude means flying at a constant *z* above the Earth's surface.



ECI Coordinates

The Earth-centered inertial (ECI) system is non-rotating. For most applications, assume this frame to be inertial, although the equinox and equatorial plane move very slightly over time. The ECI system is considered to be truly inertial for high-precision orbit calculations when the equator and equinox are defined at a particular epoch (e.g. J2000). Aerospace functions and blocks that use a particular realization of the ECI coordinate system provide that information in their documentation. The ECI system origin is fixed at the center of the Earth (see figure).

- The *x*-axis points towards the vernal equinox (First Point of Aries ♈).
- The *y*-axis points 90 degrees to the east of the *x*-axis in the equatorial plane.
- The *z*-axis points northward along the Earth rotation axis.



Earth-Centered Coordinates

ECEF Coordinates

The Earth-center, Earth-fixed (ECEF) system is a noninertial system that rotates with the Earth. Its origin is fixed at the center of the Earth.

- The *z*-axis points northward along the Earth's rotation axis.
- The *x*-axis points outward along the intersection of the Earth's equatorial plane and prime meridian.

- The *y*-axis points into the eastward quadrant, perpendicular to the *x*-*z* plane so as to satisfy the RH rule.

Coordinate Systems for Display

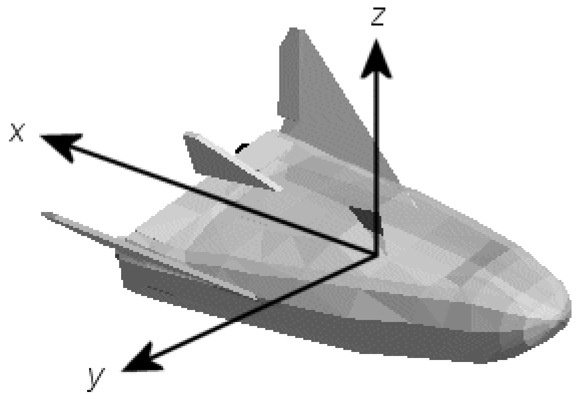
The Aerospace Toolbox software lets you use FlightGear coordinates for rendering motion.

FlightGear is an open-source, third-party flight simulator with an interface supported by the Aerospace Toolbox product.

- "Flight Simulator Interface Example" on page 2-57 discusses the toolbox interface to FlightGear.
- See the FlightGear documentation at www.flightgear.org for complete information about this flight simulator.

The FlightGear coordinates form a special body-fixed system, rotated from the standard body coordinate system about the *y*-axis by -180 degrees:

- The *x*-axis is positive toward the back of the vehicle.
- The *y*-axis is positive toward the right of the vehicle.
- The *z*-axis is positive upward, e.g., wheels typically have the lowest *z* values.



Aerospace Units

The Aerospace Toolbox functions support standard measurement systems. The Unit Conversion functions provide means for converting common measurement units from one system to another, such as converting velocity from feet per second to meters per second and vice versa.

The unit conversion functions support all units listed in this table.

Quantity	MKS (SI)	English
Acceleration	meters/second^2 (m/s^2), kilometers/second^2 (km/s^2), (kilometers/hour)/second (km/h-s), g-unit (*g*)	inches/second^2 (in/s^2), feet/second^2 (ft/s^2), (miles/hour)/second (mph/s), g-unit (*g*)
Angle	radian (rad), degree (deg), revolution	radian (rad), degree (deg), revolution
Angular acceleration	radians/second^2 (rad/s^2), degrees/second^2 (deg/s^2)	radians/second^2 (rad/s^2), degrees/second^2 (deg/s^2)
Angular velocity	radians/second (rad/s), degrees/second (deg/s), revolutions/minute (rpm), revolutions/second (rps)	radians/second (rad/s), degrees/second (deg/s), revolutions/minute (rpm), revolutions/second (rps)
Density	kilogram/meter^3 (kg/m^3)	pound mass/foot^3 (lbm/ft^3), slug/foot^3 (slug/ft^3), pound mass/inch^3 (lbm/in^3)
Force	newton (N)	pound (lb)
Length	meter (m)	inch (in), foot (ft), mile (mi), nautical mile (nm)
Mass	kilogram (kg)	slug (slug), pound mass (lbm)
Pressure	pascal (Pa)	pound/inch^2 (psi), pound/foot^2 (psf), atmosphere (atm)
Temperature	kelvin (K), degrees Celsius (^{o}C)	degrees Fahrenheit (^{o}F), degrees Rankine (^{o}R)

Quantity	MKS (SI)	English
Velocity	meters/second (m/s), kilometers/second (km/s), kilometers/hour (km/h)	inches/second (in/sec), feet/second (ft/sec), feet/minute (ft/min), miles/hour (mph), knots

Digital DATCOM Data

In this section...

Digital DATCOM Data Overview

The Aerospace Toolbox product enables bringing United States Air Force (USAF) Digital DATCOM files into the MATLAB environment by using the `datcomimport` function. For more information, see the `datcomimport` function reference page. This section explains how to import data from a USAF Digital DATCOM file.

The example used in the following topics is available as an Aerospace Toolbox example. You can run the example by entering `astimportddatcom` in the MATLAB Command Window.

USAF Digital DATCOM File

The following is a sample input file for USAF Digital DATCOM for a wing-body-horizontal tail-vertical tail configuration running over five alphas, two Mach numbers, and two altitudes and calculating static and dynamic derivatives. You can also view this file by entering `type astdatcom.in` in the MATLAB Command Window.

```
 $FLTCON NMACH=2.0,MACH(1)=0.1,0.2$
 $FLTCON NALT=2.0,ALT(1)=5000.0,8000.0$
 $FLTCON NALPHA=5.,ALSCHD(1)=-2.0,0.0,2.0,
  ALSCHD(4)=4.0,8.0,LOOP=2.0$
 $OPTINS SREF=225.8,CBARR=5.75,BLREF=41.15$
 $SYNTHS XCG=7.08,ZCG=0.0,XW=6.1,ZW=-1.4,ALIW=1.1,XH=20.2,
   ZH=0.4,ALIH=0.0,XV=21.3,ZV=0.0,VERTUP=.TRUE.$
 $BODY NX=10.0,
   X(1)=-4.9,0.0,3.0,6.1,9.1,13.3,20.2,23.5,25.9,
   R(1)=0.0,1.0,1.75,2.6,2.6,2.6,2.0,1.0,0.0$
 $WGPLNF CHRDTP=4.0,SSPNE=18.7,SSPN=20.6,CHRDR=7.2,SAVSI=0.0,CHSTAT=0.25,
   TWISTA=-1.1,SSPNDD=0.0,DHDADI=3.0,DHDADO=3.0,TYPE=1.0$
NACA-W-6-64A412
 $HTPLNF CHRDTP=2.3,SSPNE=5.7,SSPN=6.625,CHRDR=0.25,SAVSI=11.0,
   CHSTAT=1.0,TWISTA=0.0,TYPE=1.0$
NACA-H-4-0012
 $VTPLNF CHRDTP=2.7,SSPNE=5.0,SSPN=5.2,CHRDR=5.3,SAVSI=31.3,
   CHSTAT=0.25,TWISTA=0.0,TYPE=1.0$
NACA-V-4-0012
CASEID SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG
DAMP
NEXT CASE
```

The output file generated by USAF Digital DATCOM for the same wing-body-horizontal tail-vertical tail configuration running over five alphas, two Mach numbers, and two altitudes can be viewed by entering `type astdatcom.out` in the MATLAB Command Window.

Data from DATCOM Files

Use the `datcomimport` function to bring the Digital DATCOM data into the MATLAB environment.

```
alldata = datcomimport('astdatcom.out', true, 0);
```

Imported DATCOM Data

The `datcomimport` function creates a cell array of structures containing the data from the Digital DATCOM output file.

```
data = alldata{1}
data =
 struct with fields:

        case: 'SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG'
        mach: [0.1000 0.2000]
         alt: [5000 8000]
       alpha: [-2 0 2 4 8]
       nmach: 2
        nalt: 2
      nalpha: 5
       rnnub: []
      hypers: 0
        loop: 2
        sref: 225.8000
        cbar: 5.7500
       blref: 41.1500
         dim: 'ft'
       deriv: 'deg'
      stmach: 0.6000
      tsmach: 1.4000
        save: 0
       stype: []
        trim: 0
        damp: 1
       build: 1
        part: 0
     highsym: 0
     highasy: 0
     highcon: 0
        tjet: 0
      hypeff: 0
          lb: 0
         pwr: 0
        grnd: 0
       wsspn: 18.7000
       hsspn: 5.7000
      ndelta: 0
       delta: []
      deltal: []
      deltar: []
         ngh: 0
      grndht: []
      config: [1x1 struct]
          cd: [5x2x2 double]
          cl: [5x2x2 double]
          cm: [5x2x2 double]
          cn: [5x2x2 double]
          ca: [5x2x2 double]
         xcp: [5x2x2 double]
         cla: [5x2x2 double]
         cma: [5x2x2 double]
         cyb: [5x2x2 double]
         cnb: [5x2x2 double]
         clb: [5x2x2 double]
       qqinf: [5x2x2 double]
         eps: [5x2x2 double]
    depsdalp: [5x2x2 double]
         clq: [5x2x2 double]
         cmq: [5x2x2 double]
        clad: [5x2x2 double]
        cmad: [5x2x2 double]
         clp: [5x2x2 double]
         cyp: [5x2x2 double]
         cnp: [5x2x2 double]
         cnr: [5x2x2 double]
         clr: [5x2x2 double]
```

Missing DATCOM Data

By default, missing data points are set to 99999 and data points are set to NaN where no DATCOM methods exist or where the method is not applicable.

It can be seen in the Digital DATCOM output file and examining the imported data that $C_{Y\beta}$, $C_{n\beta}$, C_{Lq}, and C_{mq} have data only in the first alpha value. Here are the imported data values.

```
data.cyb
ans(:,:,1) =

  1.0e+004 *

   -0.0000   -0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999


ans(:,:,2) =

  1.0e+004 *

   -0.0000   -0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999

data.cnb
ans(:,:,1) =

  1.0e+004 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999


ans(:,:,2) =

  1.0e+004 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999

data.clq
ans(:,:,1) =

  1.0e+004 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999


ans(:,:,2) =

  1.0e+004 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999

data.cmq
ans(:,:,1) =

  1.0e+004 *
```

```
   -0.0000   -0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999


ans(:,:,2) =

  1.0e+004 *

   -0.0000   -0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
```

The missing data points will be filled with the values for the first alpha, since these data points are meant to be used for all alpha values.

```
aerotab = {'cyb' 'cnb' 'clq' 'cmq'};

for k = 1:length(aerotab)
    for m = 1:data.nmach
        for h = 1:data.nalt
            data.(aerotab{k})(:,m,h) = data.(aerotab{k})(1,m,h);
        end
    end
end
```

Here are the updated imported data values.

```
data.cyb
ans(:,:,1) =

   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035


ans(:,:,2) =

   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035

data.cnb
ans(:,:,1) =

  1.0e-003 *

    0.9142    0.8781
    0.9142    0.8781
    0.9142    0.8781
    0.9142    0.8781
    0.9142    0.8781
```

```
ans(:,:,2) =

  1.0e-003 *

    0.9190    0.8829
    0.9190    0.8829
    0.9190    0.8829
    0.9190    0.8829
    0.9190    0.8829

data.clq
ans(:,:,1) =

    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984


ans(:,:,2) =

    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984

data.cmq
ans(:,:,1) =

   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899


ans(:,:,2) =

   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
```

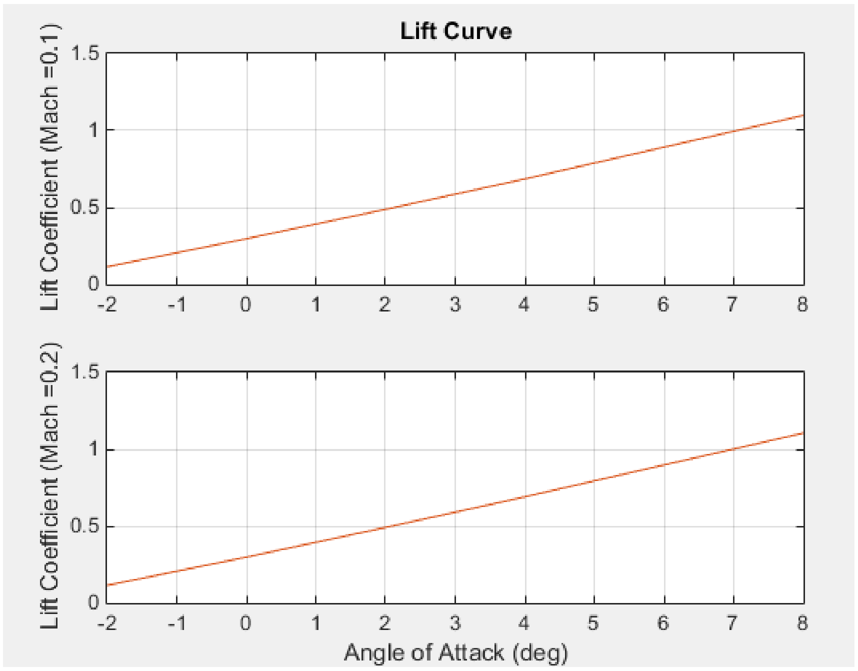
Aerodynamic Coefficients

You can now plot the aerodynamic coefficients:

- “Plotting Lift Curve Moments” on page 2-20
- “Plotting Drag Polar Moments” on page 2-20
- “Plotting Pitching Moments” on page 2-21

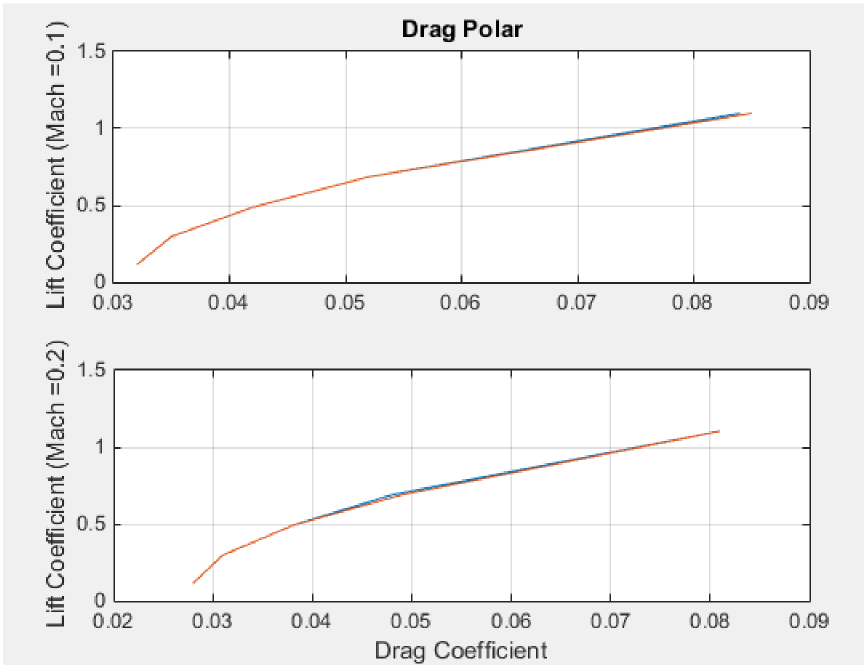
Plotting Lift Curve Moments

```
h1 = figure;
figtitle = {'Lift Curve' ''};
for k=1:2
    subplot(2,1,k)
    plot(data.alpha,permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach =' num2str(data.mach(k)) ')'])
    title(figtitle{k});
end
xlabel('Angle of Attack (deg)')
```



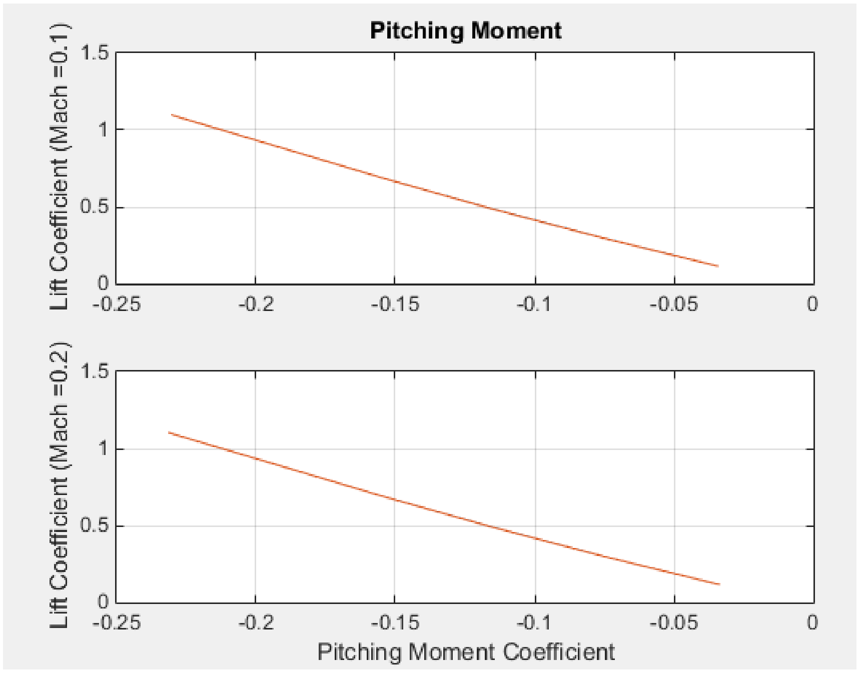
Plotting Drag Polar Moments

```
h2 = figure;
figtitle = {'Drag Polar' ''};
for k=1:2
    subplot(2,1,k)
    plot(permute(data.cd(:,k,:),[1 3 2]),permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach =' num2str(data.mach(k)) ')'])
    title(figtitle{k})
end
xlabel('Drag Coefficient')
```



Plotting Pitching Moments

```
h3 = figure;
figtitle = {'Pitching Moment' ''};
for k=1:2
    subplot(2,1,k)
    plot(permute(data.cm(:,k,:),[1 3 2]),permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach =' num2str(data.mach(k)) ')'])
    title(figtitle{k})
end
xlabel('Pitching Moment Coefficient')
```


Pitching Moment
Lift Coefficient (Mach =0.1)
Lift Coefficient (Mach =0.2)
Pitching Moment Coefficient
1.5
1
0.5
0
-0.25
-0.2
-0.15
-0.1
-0.05
0

Aerospace Toolbox Animation Objects

To visualize flight data in the Aerospace Toolbox environment, you can use the following animation objects and their associated methods. These animation objects use the MATLAB time series object, `timeseries` to visualize flight data.

- `Aero.Animation` — Visualize flight data without any other tool or toolbox. The following objects support this object.

  - `Aero.Body`
  - `Aero.Camera`
  - `Aero.Geometry`

  For more information, see “Aero.Animation Objects” on page 2-24.
- `Aero.VirtualRealityAnimation` — Visualize flight data with the Simulink 3D Animation™ product. The following objects support this object.

  - `Aero.Node`
  - `Aero.Viewpoint`

  For more information, see “Aero.VirtualRealityAnimation Objects” on page 2-34.
- `Aero.FlightGearAnimation` — Visualize flight data with the FlightGear simulator. For more information, see “Aero.FlightGearAnimation Objects” on page 2-53.

Aero.Animation Objects

The toolbox interface to animation objects uses the Handle Graphics capability. The Overlaying Simulated and Actual Flight Data (`astmlanim`) example visually compares simulated and actual flight trajectory data by creating animation objects, creating bodies for those objects, and loading the flight trajectory data.

- Create and configure an animation object.
- Load recorded data for flight trajectories.
- Display body geometries in a figure window.
- Play back flight trajectories using the animation object.
- Manipulate the camera.
- Move and reposition bodies.
- Create a transparency in the first body.
- Change the color of the second body.
- Turn off the landing gear of the second body.

Running the Example

1. Start the MATLAB software.
2. Enter `astmlanim` in the MATLAB Command Window.

   While running, the example performs several steps by issuing a series of commands.

Simulated and Actual Flight Data Using Aero.Animation Objects

Creating and Configuring an Animation Object

This series of commands creates an animation object and configures the object.

1. Create an animation object.

   ```
   h = Aero.Animation;
   ```

2. Configure the animation object to set the number of frames per second (`FramesPerSecond`) property. This configuration controls the rate at which frames are displayed in the figure window.

   ```
   h.FramesPerSecond = 10;
   ```

3. Configure the animation object to set the seconds of animation data per second time scaling (`TimeScaling`) property.

   ```
   h.TimeScaling = 5;
   ```

   The combination of `FramesPerSecond` and `TimeScaling` property determine the time step of the simulation. These settings result in a time step of approximately 0.5 s.

4. Create and load bodies for the animation object. This example uses these bodies to work with and display the simulated and actual flight trajectories. The first body is orange; it represents simulated data. The second body is blue; it represents the actual flight data.

   ```
   idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
   idx2 = h.createBody('pa24-250_blue.ac','Ac3d');
   ```

   Both bodies are AC3D format files. AC3D is one of several file formats that the animation objects support. FlightGear uses the same file format. The animation object reads in the bodies in the AC3D format and stores them as patches in the geometry object within the animation object.

Loading Recorded Data for Flight Trajectories

This series of commands loads the recorded flight trajectory data, which is contained in files in the *`matlabroot`*`\toolbox\aero\astdemos` folder.

- `simdata` – Contains simulated flight trajectory data, which is set up as a 6DoF array.
- `fltdata` – Contains actual flight trajectory data which is set up in a custom format. To access this custom format data, the example must set the body object **TimeSeriesSourceType** parameter to `Custom` and then specify a custom read function.

1 Load the flight trajectory data.

```
load simdata
load fltdata
```

2 Set the time series data for the two bodies.

```
h.Bodies{1}.TimeSeriesSource = simdata;
h.Bodies{2}.TimeSeriesSource = fltdata;
```

3 Identify the time series for the second body as custom.

```
h.Bodies{2}.TimeSeriesSourceType = 'Custom';
```

4 Specify the custom read function to access the data in `fltdata` for the second body. The example provides the custom read function in `*matlabroot*\toolbox\aero\astdemos\CustomReadBodyTSData.m`.

```
h.Bodies{2}.TimeseriesReadFcn = @CustomReadBodyTSData;
```

Displaying Body Geometries in a Figure Window

This command creates a figure object for the animation object.

```
h.show();
```

Recording Animation Files

Enable recording of the playback of flight trajectories using the animation object on page 2-27.

```
h.VideoRecord = 'on';
h.VideoQuality = 50;
h.VideoCompression = 'Motion JPEG AVI'
h.VideoFilename = 'astMotion_JPEG';
```

Enable animation recording at any point that you want to preserve an animation sequence.

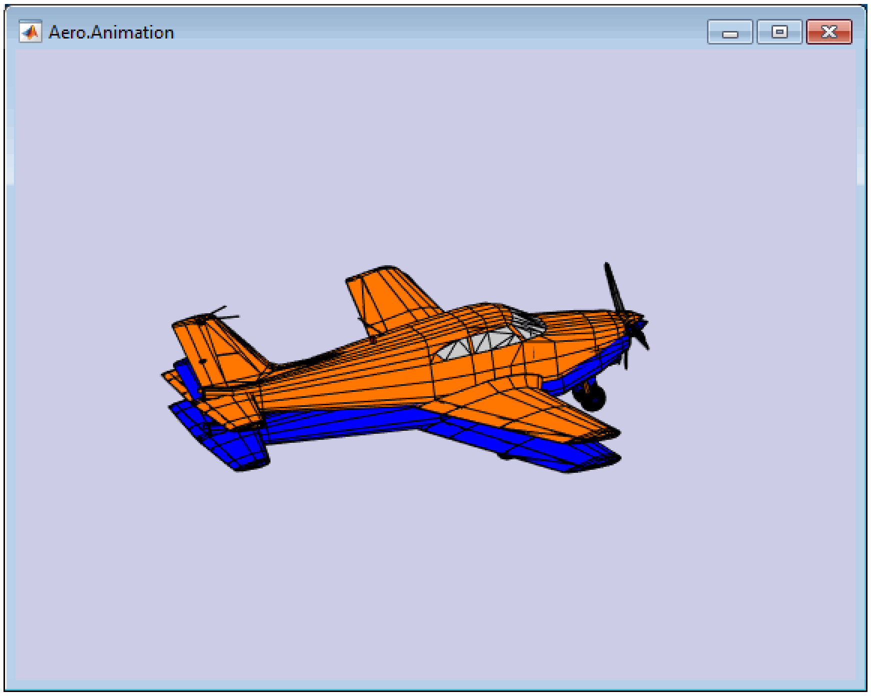
Note When choosing the video compression type, keep in mind that you will need the corresponding viewer software. For example, if you create an AVI format, you need a viewer such as Windows Media® Player to view the file.

After you play the animation as described in “Playing Back Flight Trajectories Using the Animation Object” on page 2-27, `astMotion_JPEG` contains a recording of the playback.

Playing Back Flight Trajectories Using the Animation Object

This command plays back the animation bodies for the duration of the time series data. This playback shows the differences between the simulated and actual flight data.

```
h.play();
```



If you used the `Video` properties to store the recording, see “Viewing Recorded Animation Files” on page 2-28 for a description of how to view the files.

Viewing Recorded Animation Files

If you do not have an animation file to view, see “Recording Animation Files” on page 2-26.

1. Open the folder that contains the animation file you want to view.
2. View the animation file with an application of your choice.

   If your animation file is not yet running, start it now from the application.
3. To prevent other `h.play` commands from overwriting the contents of the animation file, disable the recording after you are satisfied with the contents.

   ```
   h.VideoRecord = 'off';
   ```

Manipulating the Camera

This command series shows how you can manipulate the camera on the two bodies and redisplay the animation. The `PositionFcn` property of a camera object controls the camera position relative to the bodies in the animation. In “Playing Back Flight Trajectories Using the Animation Object” on page 2-27, the camera object uses a default value for the `PositionFcn` property. In this command series, the example references a custom `PositionFcn` function that uses a static position based on the position of the bodies. No dynamics are involved.

Note The custom `PositionFcn` function is located in the `*matlabroot*\toolbox\aero\astdemos` folder.

1. Set the camera `PositionFcn` to the custom function `staticCameraPosition`.

   ```
   h.Camera.PositionFcn = @staticCameraPosition;
   ```
2. Run the animation again.

   ```
   h.play();
   ```

Moving and Repositioning Bodies

This series of commands illustrates how to move and reposition bodies.

1. Set the starting time to 0.

   ```
   t = 0;
   ```

2. Move the body to the starting position that is based on the time series data. Use the Aero.Animation object `updateBodies` method.

   ```
   h.updateBodies(t);
   ```

3. Update the camera position using the custom `PositionFcn` function set in the previous section. Use the Aero.Animation object `updateCamera` method.

   ```
   h.updateCamera(t);
   ```

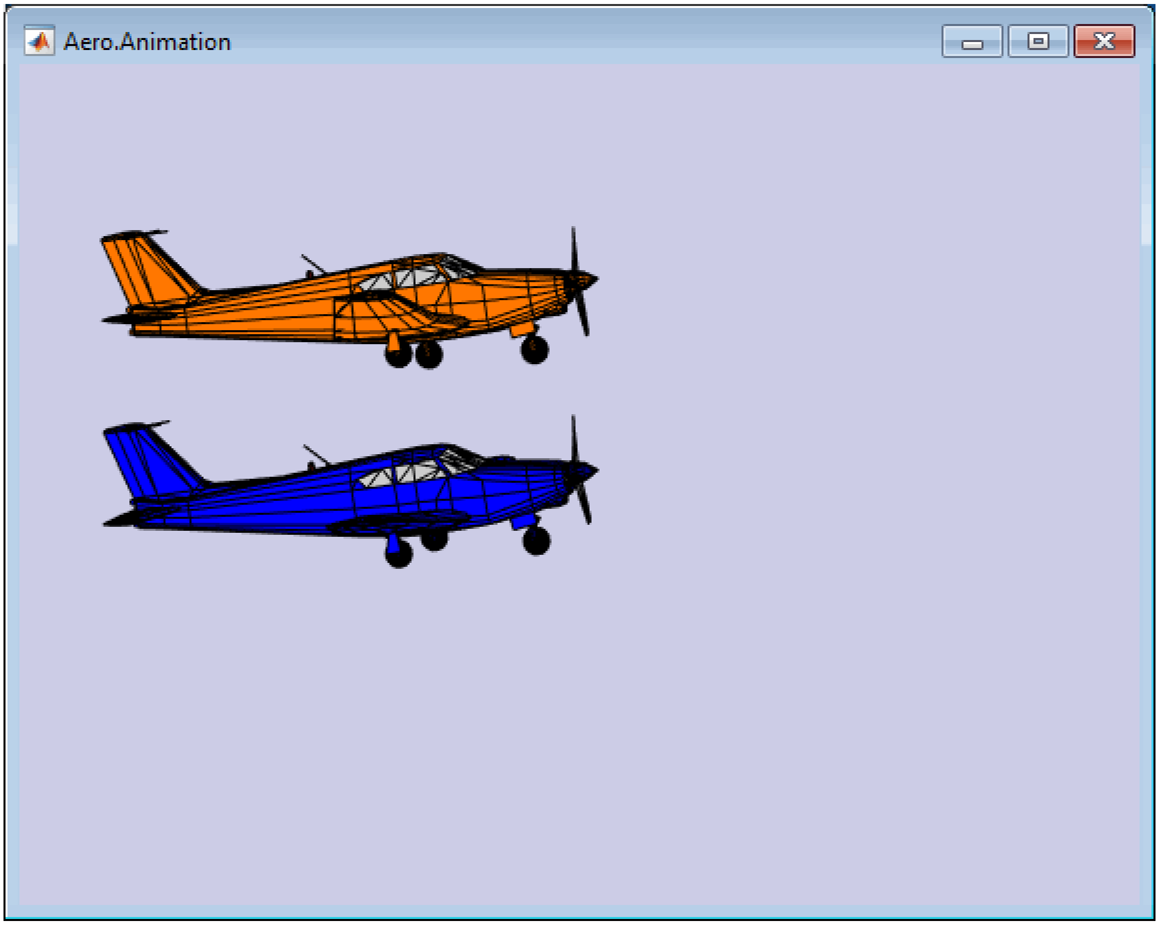
4. Reposition the bodies by first getting the current body position, then separating the bodies.

   a. Get the current body positions and rotations from the objects of both bodies.

      ```
      pos1 = h.Bodies{1}.Position;
      rot1 = h.Bodies{1}.Rotation;
      pos2 = h.Bodies{2}.Position;
      rot2 = h.Bodies{2}.Rotation;
      ```

   b. Separate and reposition the bodies by moving them to new positions.

      ```
      h.moveBody(1,pos1 + [0 0 -3],rot1);
      h.moveBody(2,pos1 + [0 0  0],rot2);
      ```



Creating a Transparency in the First Body

This series of commands illustrates how to create and attach a transparency to a body. The animation object stores the body geometry as patches. This example manipulates the transparency properties of these patches (see Patch Properties).

Note The use of transparencies might decrease animation speed on platforms that use software OpenGL® rendering (see `opengl`).

1 Change the body patch properties. Use the Aero.Body `PatchHandles` property to get the patch handles for the first body.

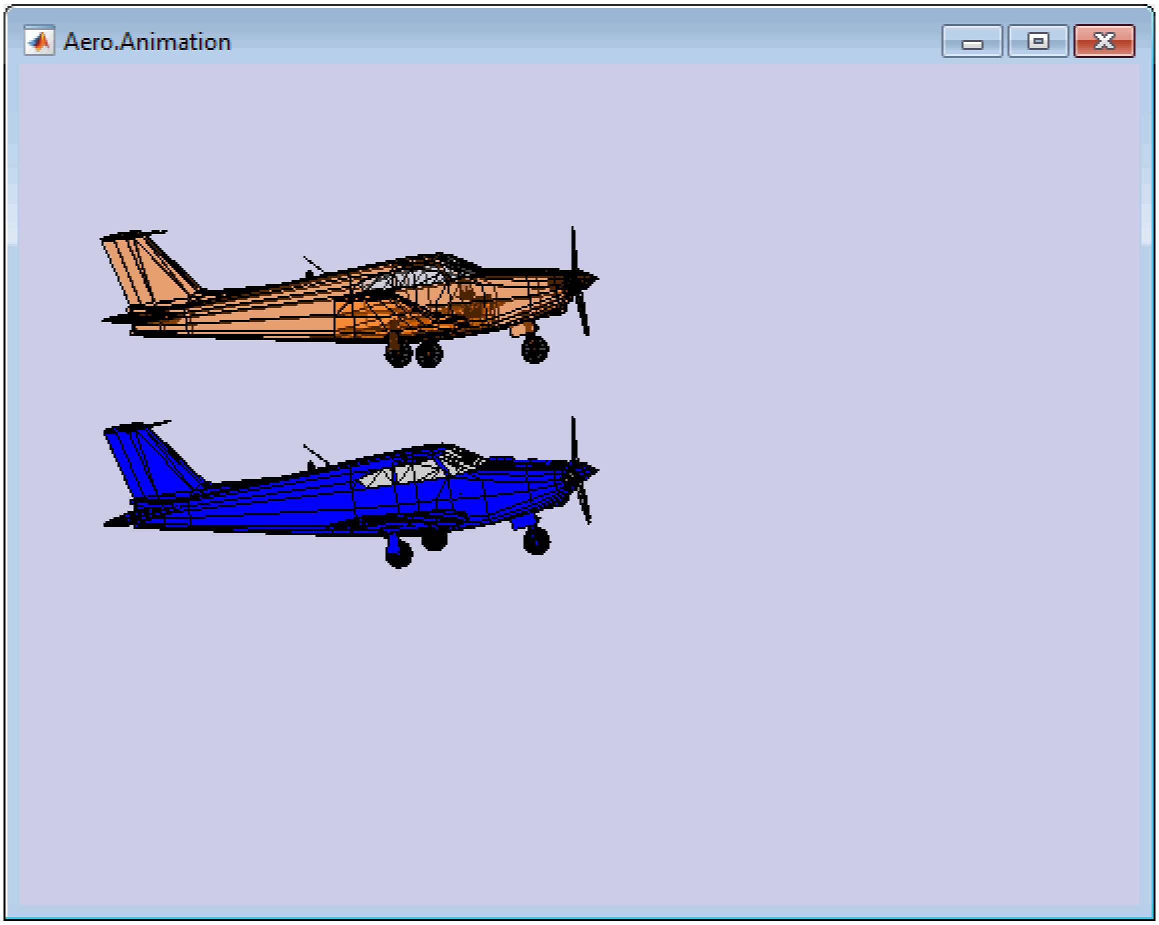
```
patchHandles2 = h.Bodies{1}.PatchHandles;
```

2 Set the face and edge alpha values that you want for the transparency.

```
desiredFaceTransparency = .3;
desiredEdgeTransparency = 1;
```

3 Get the current face and edge alpha data and change all values to the alpha values that you want. In the figure, the first body now has a transparency.

```
for k = 1:size(patchHandles2,1)
    tempFaceAlpha = get(patchHandles2(k),'FaceVertexAlphaData');
    tempEdgeAlpha = get(patchHandles2(k),'EdgeAlpha');
       set(patchHandles2(k),...
        'FaceVertexAlphaData',repmat(desiredFaceTransparency,size(tempFaceAlpha)));
    set(patchHandles2(k),...
        'EdgeAlpha',repmat(desiredEdgeTransparency,size(tempEdgeAlpha)));
end
```



Changing the Color of the Second Body

This series of commands illustrates how to change the color of a body. The animation object stores the body geometry as patches. This example manipulates the `FaceVertexColorData` property of these patches.

1. Change the body patch properties. Use the Aero.Body `PatchHandles` property to get the patch handles for the first body.

   ```
   patchHandles3 = h.Bodies{2}.PatchHandles;
   ```

2 Set the patch color to red.

```
desiredColor = [1 0 0];
```

3 Get the current face color and data and propagate the new patch color, red, to the face.

- The `if` condition prevents the windows from being colored.
- The `name` property is stored in the body geometry data (`h.Bodies{2}.Geometry.FaceVertexColorData(k).name`).
- The code changes only the indices in `patchHandles3` with nonwindow counterparts in the body geometry data.

Note If you cannot access the `name` property to determine the parts of the vehicle to color, you must use an alternative way to selectively color your vehicle.

```
for k = 1:size(patchHandles3,1)
    tempFaceColor = get(patchHandles3(k),'FaceVertexCData');
    tempName = h.Bodies{2}.Geometry.FaceVertexColorData(k).name;
    if isempty(strfind(tempName,'Windshield')) &&...
       isempty(strfind(tempName,'front-windows')) &&...
       isempty(strfind(tempName,'rear-windows'))
    set(patchHandles3(k),...
        'FaceVertexCData',repmat(desiredColor,[size(tempFaceColor,1),1]));
    end
end
```

Turning Off the Landing Gear of the Second Body

This command series illustrates how to turn off the landing gear on the second body by turning off the visibility of all the vehicle parts associated with the landing gear.

Note The indices into the `patchHandles3` vector are determined from the `name` property. If you cannot access the `name` property to determine the indices, you must use an alternative way to determine the indices that correspond to the geometry parts.

```
for k = [1:8,11:14,52:57]
    set(patchHandles3(k),'Visible','off')
end
```

Aero.VirtualRealityAnimation Objects

The Aerospace Toolbox interface to virtual reality animation objects uses the Simulink 3D Animation software. For more information, see `Aero.VirtualRealityAnimation`, `Aero.Node`, and `Aero.Viewpoint`.

- Create, configure, and initialize an animation object.
- Enable the tracking of changes to virtual worlds.
- Load the animation world.
- Load time series data for simulation.
- Set coordination information for the object.
- Add a chase helicopter to the object.
- Load time series data for chase helicopter simulation.
- Set coordination information for the new object.
- Add a new viewpoint for the helicopter.
- Play the animation.
- Create a new viewpoint.
- Add a route.
- Add another helicopter.
- Remove bodies.
- Revert to the original world.

Running the Example

1. Start the MATLAB software.
2. Enter `astvranim` in the MATLAB Command Window.

   While running, the example performs several steps by issuing a series of commands.

Example of Visualize Aircraft Takeoff and Chase Helicopter with the Virtual Reality Animation Object

Visualize Aircraft Takeoff via Virtual Reality Animation Object

This example shows how to visualize aircraft takeoff and chase helicopter with the virtual reality animation object. In this example, you can use the Aero.VirtualRealityAnimation object to set up a virtual reality animation based on the asttkoff.wrl file. The scene simulates an aircraft takeoff. The example adds a chase vehicle to the simulation and a chase viewpoint associated with the new vehicle.

Create the Animation Object

This code creates an instance of the `Aero.VirtualRealityAnimation` object.

```
h = Aero.VirtualRealityAnimation;
```

Set the Animation Object Properties

This code sets the number of frames per second and the seconds of animation data per second time scaling. `'FramesPerSecond'` controls the rate at which frames are displayed in the figure window. `'TimeScaling'` is the seconds of animation data per second time scaling.

The `'TimeScaling'` and `'FramesPerSecond'` properties determine the time step of the simulation. The settings in this example result in a time step of approximately 0.5s. The equation is:

(1/FramesPerSecond)*TimeScaling + extra terms to handle for sub-second precision.

```
h.FramesPerSecond = 10;
h.TimeScaling = 5;
```

This code sets the .wrl file to be used in the virtual reality animation.

```
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
```

Change Directory

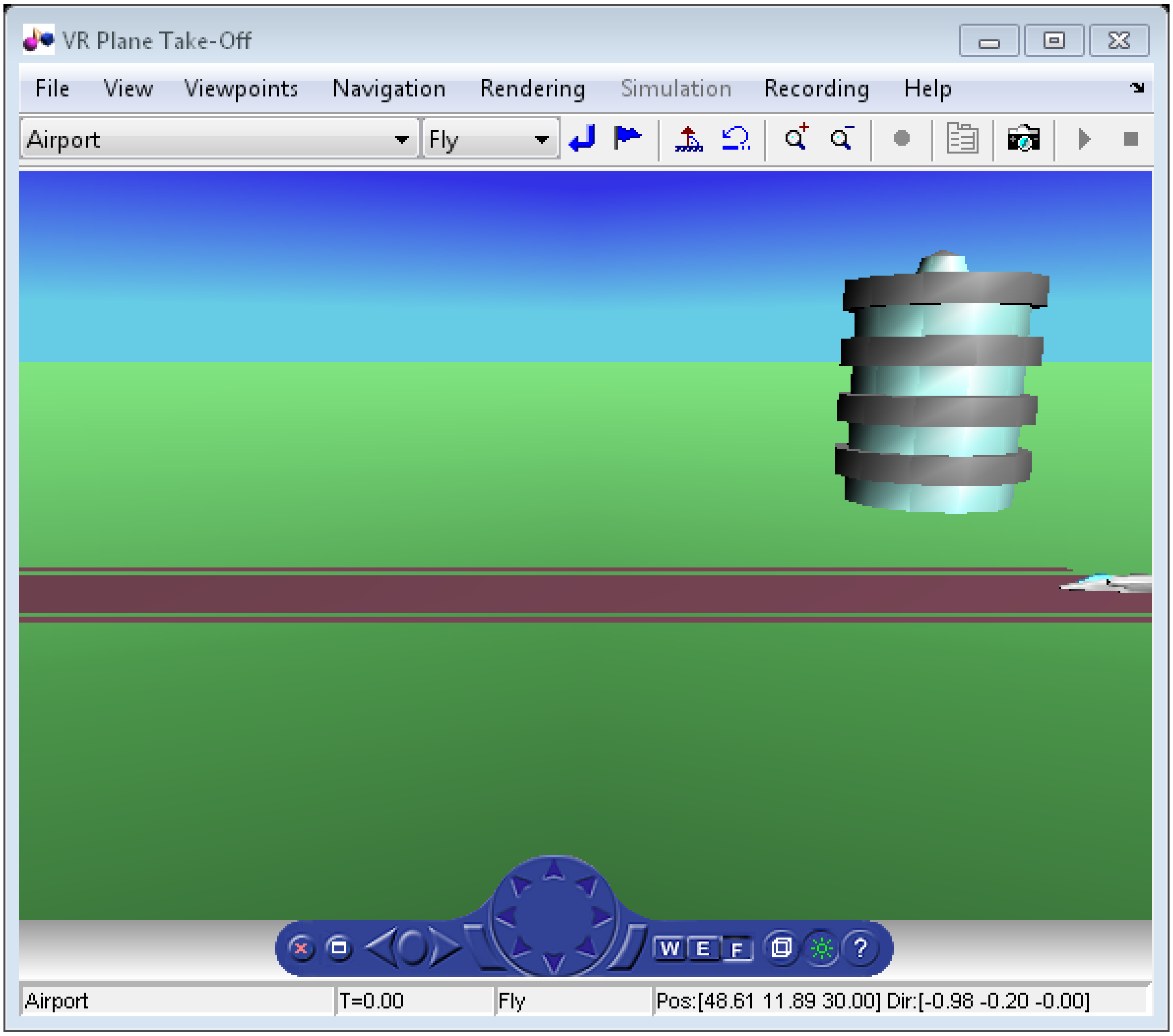
The VirtualRealityAnimation object methods use temporary .wrl files to keep track of changes to the world. This requires the directory containing the original .wrl file to be writable. This code runs the example from a temporary directory to ensure there are no issues with directory permissions. Note, a license for Simulink® 3D Animation™ is required to run this example.

```
% Copy file to temporary directory
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
% Set world filename to the copied .wrl file.
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
```

Initialize the Virtual Reality Animation Object

The `initialize` method loads the animation world described in the `'VRWorldFilename'` field of the animation object. When parsing the world, node objects are created for existing nodes with DEF names. The `initialize` method also opens the Simulink 3D Animation viewer.

```
h.initialize();
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F

Airport T=0.00 Fly Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.00]

Set Additional Node Information

This code sets simulation timeseries data. `takeoffData.mat` contains logged simulated data. `takeoffData` is set up as a `'StructureWithTime'`, which is one of the default data formats.

```
load takeoffData
[~, idxPlane] = find(strcmp('Plane', h.nodeInfo));
h.Nodes{idxPlane}.TimeseriesSource = takeoffData;
h.Nodes{idxPlane}.TimeseriesSourceType = 'StructureWithTime';
```

Set Coordinate Transform Function

The virtual reality animation object expects positions and rotations in aerospace body coordinates. If the input data is different, you must create a coordinate transformation function in order to correctly line up the position and rotation data with the surrounding objects in the virtual world. This code sets the coordinate transformation function for the virtual reality animation.

In this particular case, if the input translation coordinates are [x1,y1,z1], they must be adjusted as follows: [X,Y,Z] = -[y1,x1,z1]. The custom transform function can be seen here: `matlabroot/toolbox/aero/astdemos/vranimCustomTransform.m`

```
h.Nodes{idxPlane}.CoordTransformFcn = @vranimCustomTransform;
```

Add a Chase Helicopter

This code shows how to add a chase helicopter to the animation object.

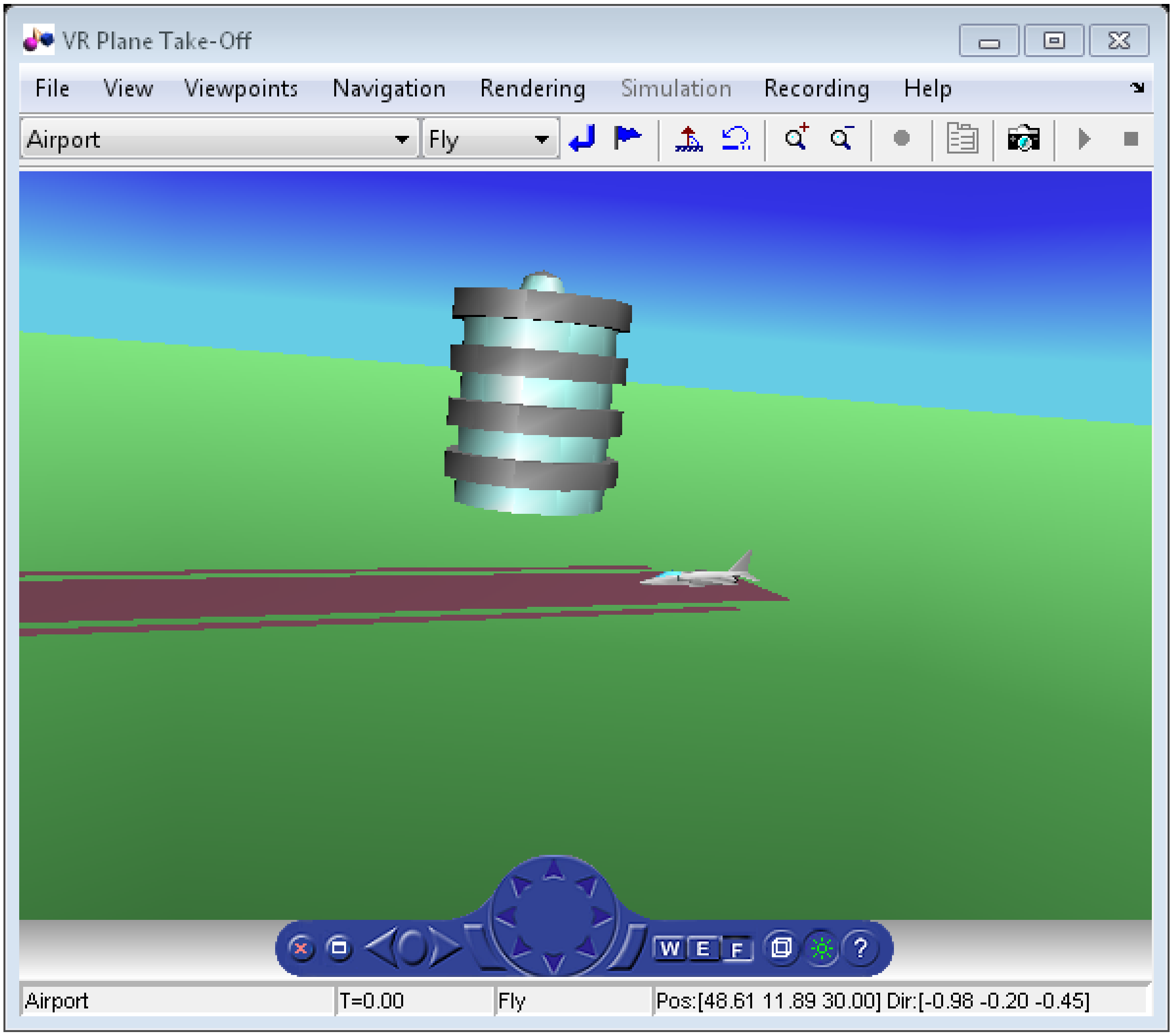
You can view all the nodes currently in the virtual reality animation object by using the `nodeInfo` method. When called with no output argument, this method prints the node information to the command window. With an output argument, the method sets node information to that argument.

```
h.nodeInfo;
```

```
Node Information
1    Camera1
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
```

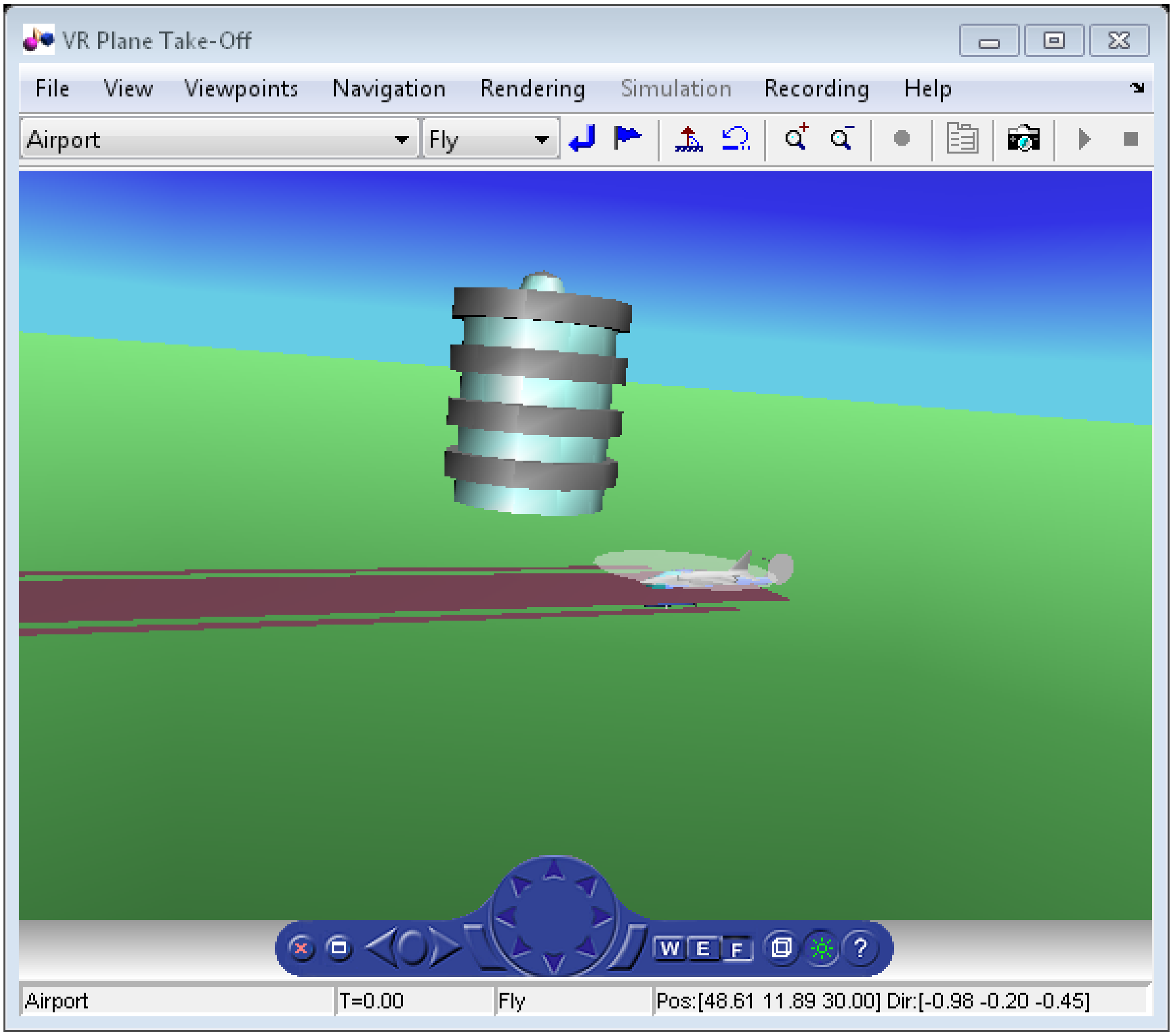
This code moves the camera angle of the virtual reality figure to view the aircraft.

```
set(h.VRFigure,'CameraDirection',[0.45 0 -1]);
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F
Airport
T=0.00
Fly
Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.45]

Use the addNode method to add another node to the object. By default, each time you add or remove a node or route, or when you call the saveas method, Aerospace Toolbox displays a message about the current .wrl file location. To disable this message, set the 'ShowSaveWarning' property in the VirtualRealityAnimation object.

```
h.ShowSaveWarning = false;
h.addNode('Lynx',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
```



Airport | T=0.00 | Fly | Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.45]

Another call to `nodeInfo` shows the newly added Node objects.

```
h.nodeInfo
```

```
Node Information
1    Camera1
```

```
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
9    Lynx
10    Lynx_Inline
```

Adjust newly added helicopter to sit on runway.

```
[~, idxLynx] = find(strcmp('Lynx',h.nodeInfo));
h.Node{idxLynx}.VRNode.translation = [0 1.5 0];
```

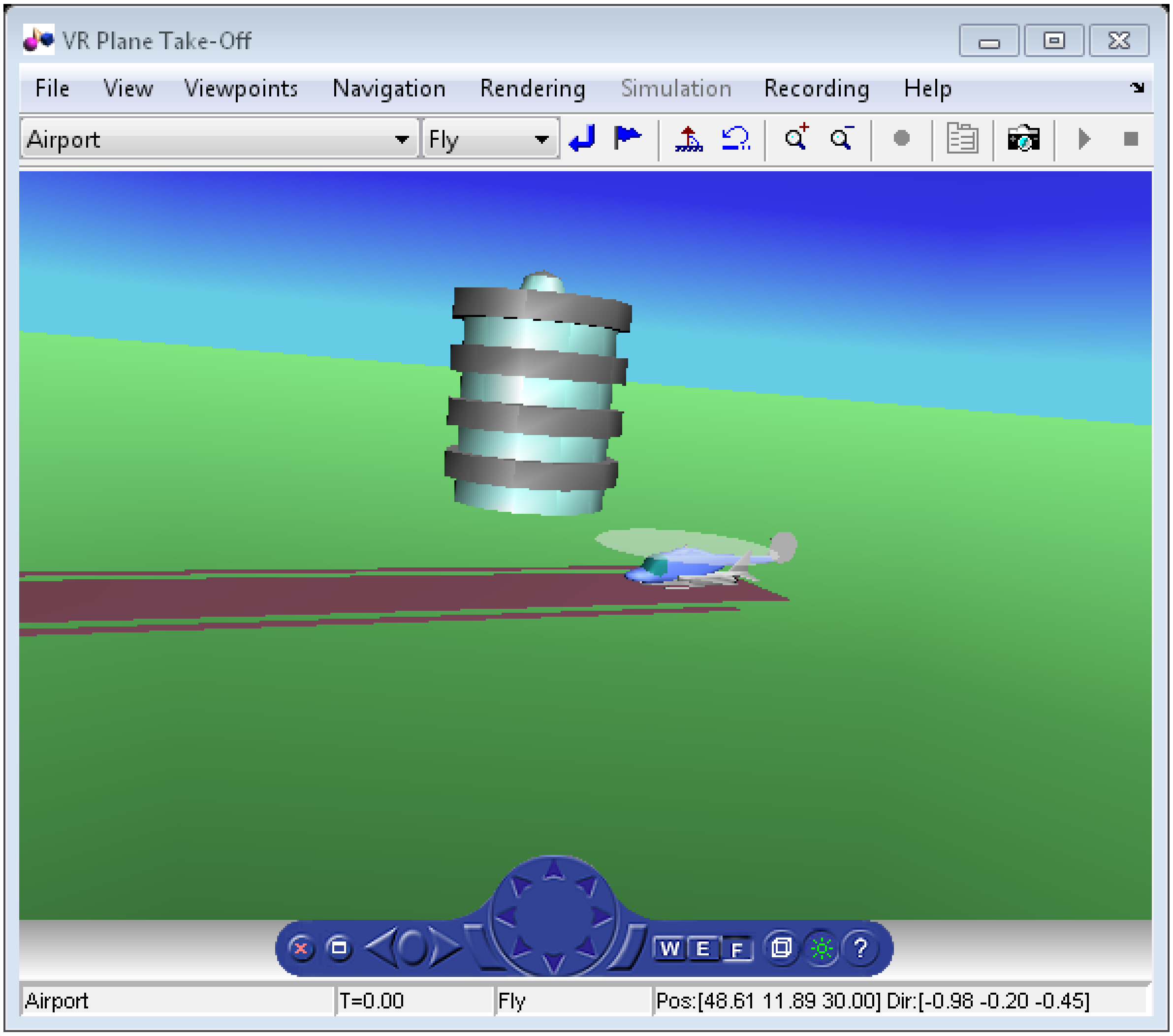
This code sets data properties for the chase helicopter. The `'TimeseriesSourceType'` is the default `'Array6DoF'`, so no additional property changes are needed. The same coordinate transform function (`vranimCustomTransform`) is used for this node as the preceding node. The previous call to `nodeInfo` returned the node index (2).

```
load chaseData
h.Nodes{idxLynx}.TimeseriesSource = chaseData;
h.Nodes{idxLynx}.CoordTransformFcn = @vranimCustomTransform;
```

Create New Viewpoint

This code uses the `addViewpoint` method to create a new viewpoint named 'chaseView'. The new viewpoint will appear in the viewpoint pulldown menu in the virtual reality window as "View From Helicopter". Another call to `nodeInfo` shows the newly added node objects. The node is created as a child of the chase helicopter.

```
h.addViewpoint(h.Nodes{idxLynx}.VRNode,'children','chaseView','View From Helicopter');
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F
Airport
T=0.00
Fly
Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.45]

Play Animation

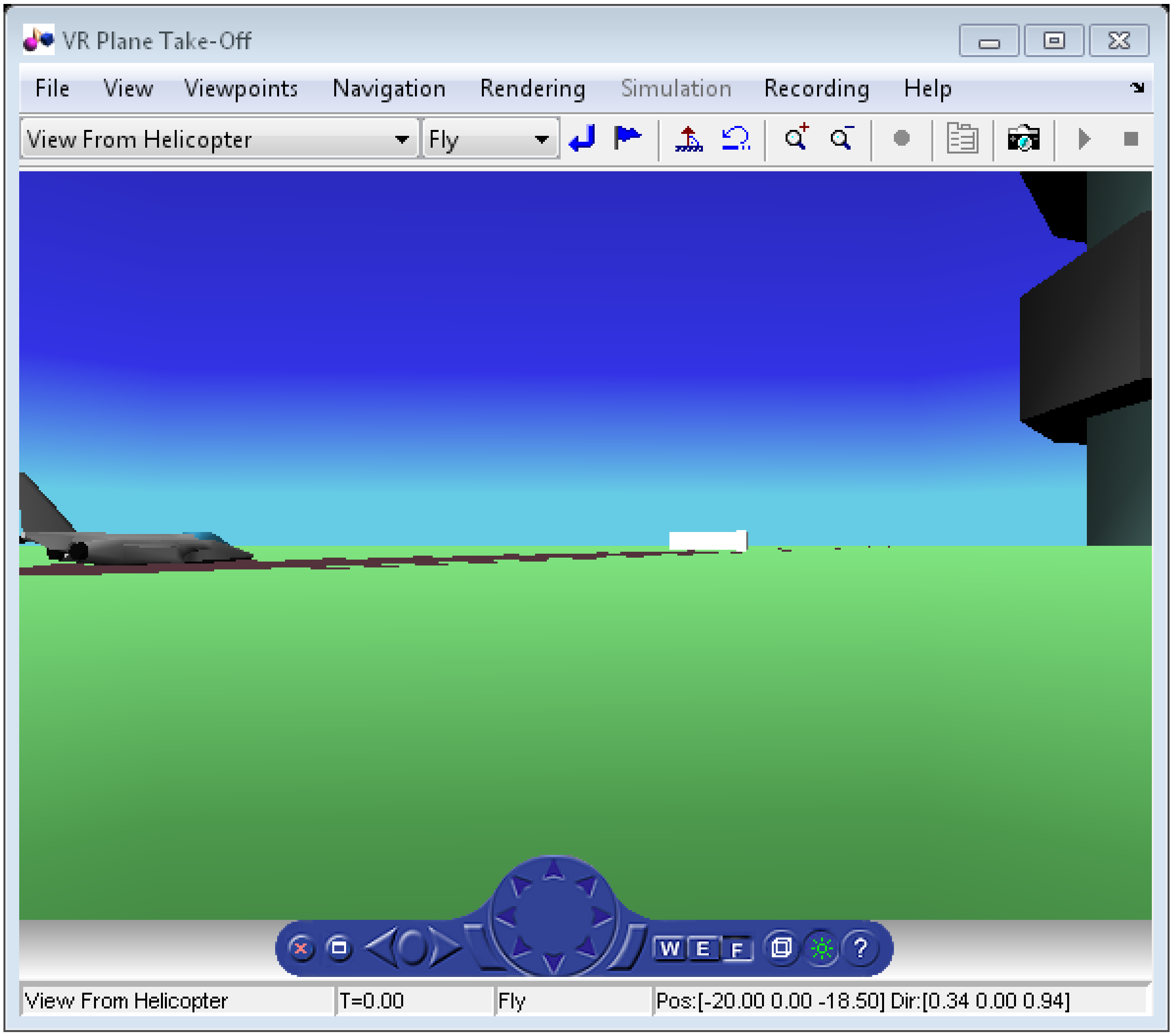
The `play` method runs the simulation for the specified timeseries data.

```
h.play();
```

Play Animation From Helicopter

This code sets the orientation of the viewpoint via the vrnode object associated with the node object for the viewpoint. In this case, it will change the viewpoint to look out the left side of the helicopter at the plane.

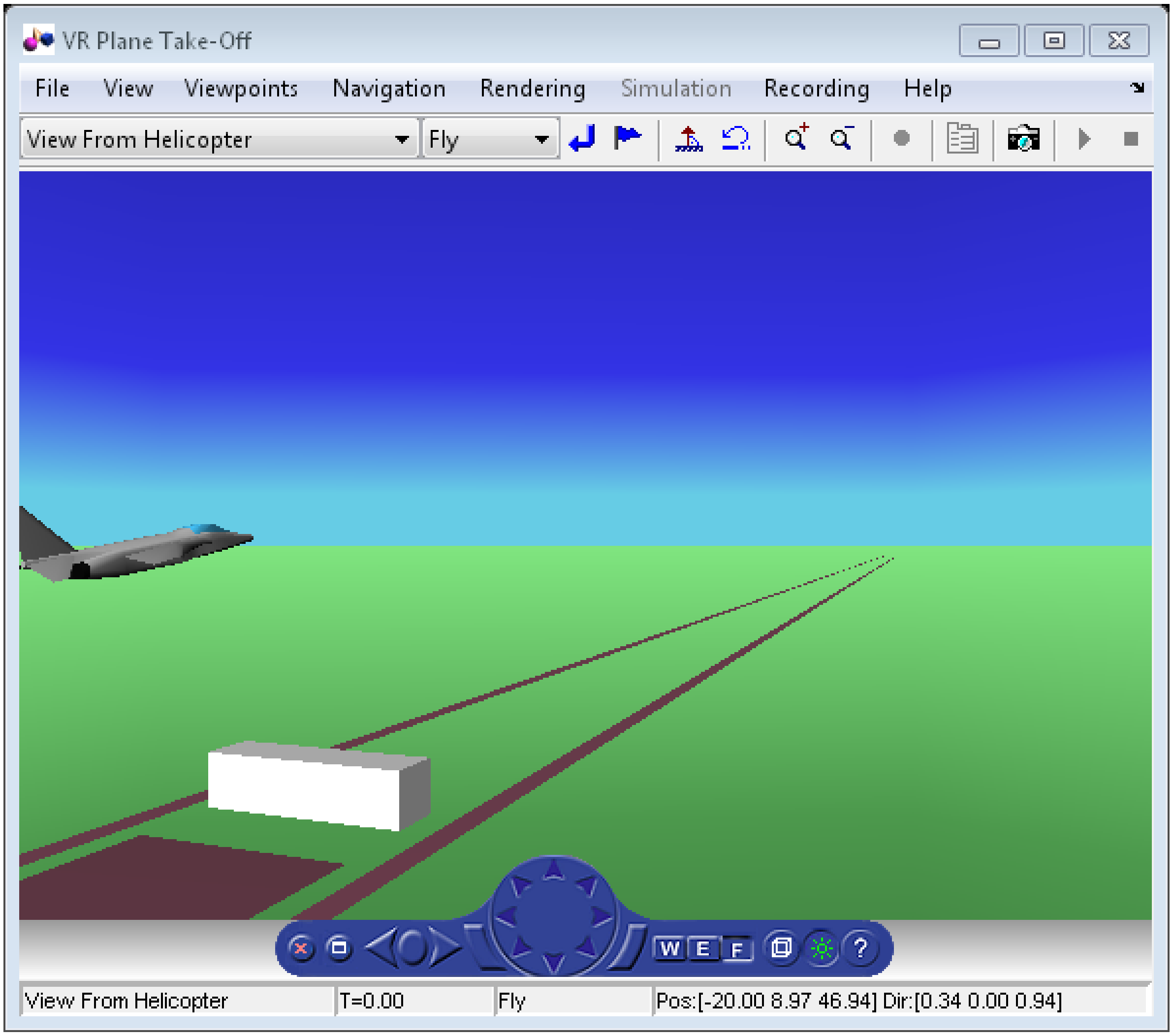
```
[~, idxChaseView] = find(strcmp('chaseView',h.nodeInfo));
h.Nodes{idxChaseView}.VRNode.orientation = [0 1 0 convang(200,'deg','rad')];
set(h.VRFigure,'Viewpoint','View From Helicopter');
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
View From Helicopter
Fly
W E F
View From Helicopter
T=0.00
Fly
Pos:[-20.00 0.00 -18.50] Dir:[0.34 0.00 0.94]

Add ROUTE

This code calls the `addRoute` method to add a ROUTE command to connect the plane position to the Camera1 node. This will allow for the "Ride on the Plane" viewpoint to function as intended.

```
h.addRoute('Plane','translation','Camera1','translation');
```

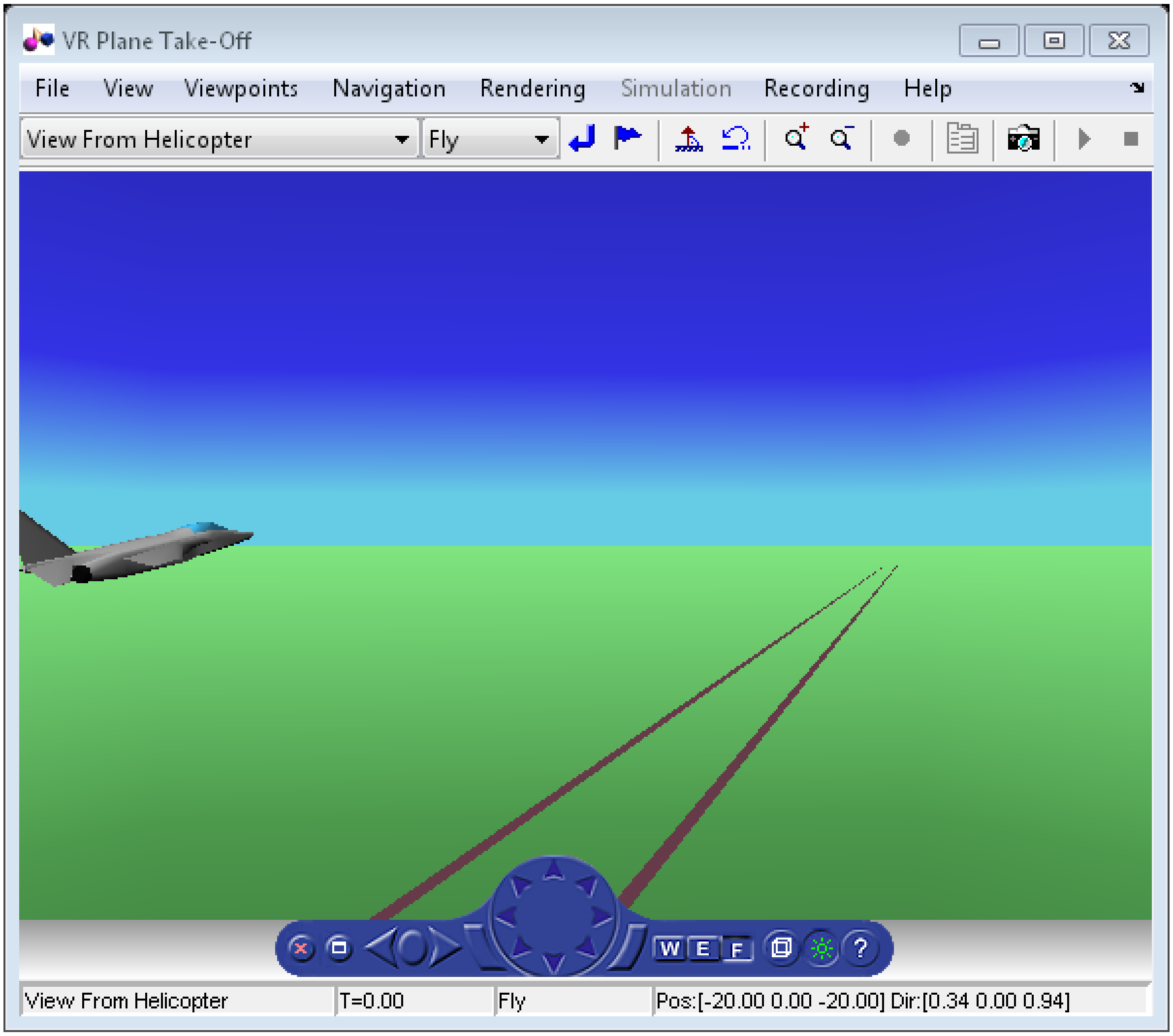


View From Helicopter | T=0.00 | Fly | Pos:[-20.00 8.97 46.94] Dir:[0.34 0.00 0.94]

The scene from the helicopter viewpoint

This code plays the animation.

```
h.play();
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
View From Helicopter
Fly
W E F

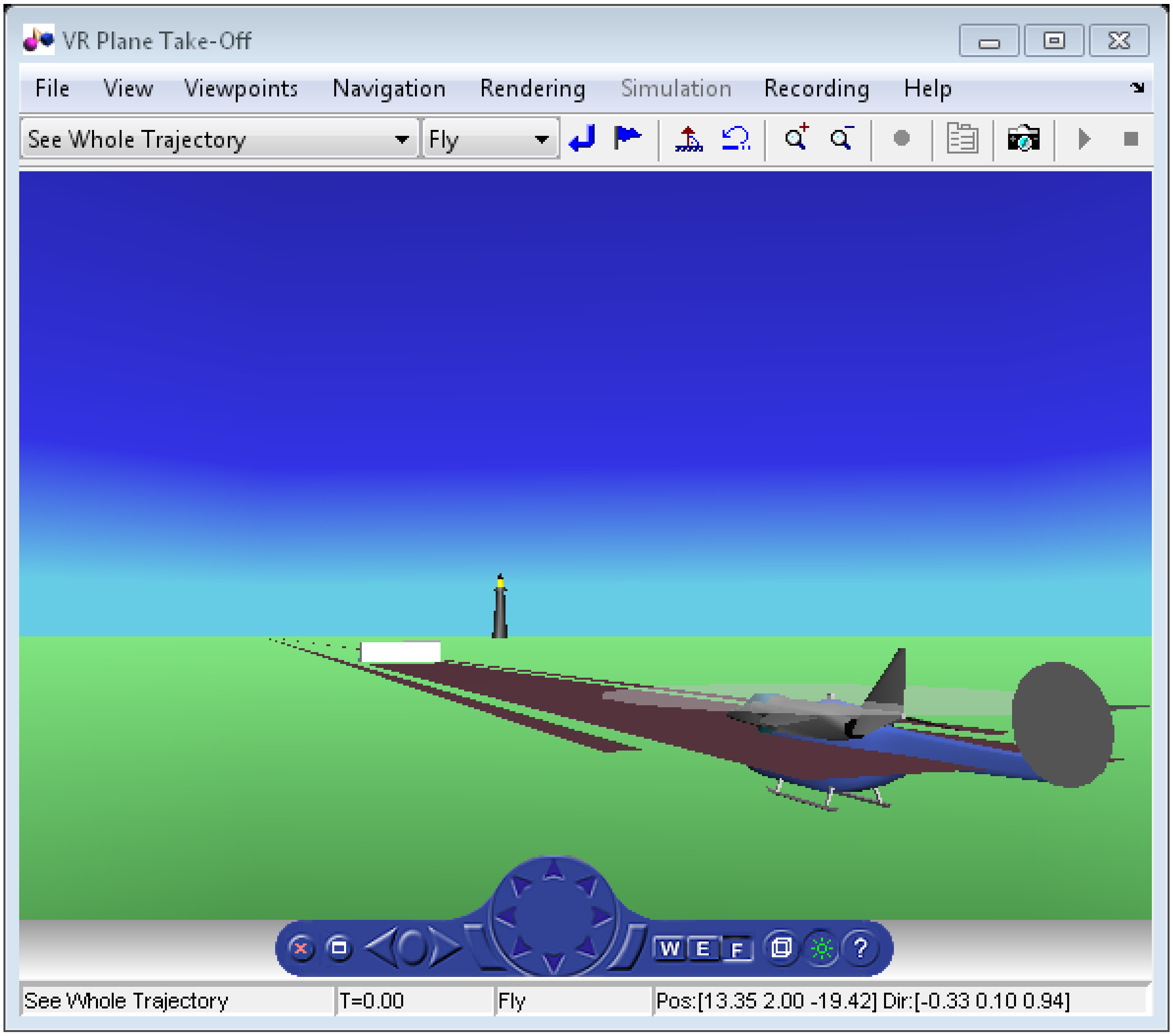
View From Helicopter T=0.00 Fly Pos:[-20.00 0.00 -20.00] Dir:[0.34 0.00 0.94]

Add Another Body

This code adds another helicopter to the scene. It also changes to another viewpoint to view all three bodies in the scene at once.

```
set(h.VRFigure,'Viewpoint','See Whole Trajectory');
h.addNode('Lynx1',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
h.nodeInfo
```

```
Node Information
1    Camera1
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
9    Lynx
10    Lynx_Inline
11    chaseView
12    Lynx1
13    Lynx1_Inline
```



See Whole Trajectory | T=0.00 | Fly | Pos:[13.35 2.00 -19.42] Dir:[-0.33 0.10 0.94]

Adjust newly added helicopter to sit above runway.

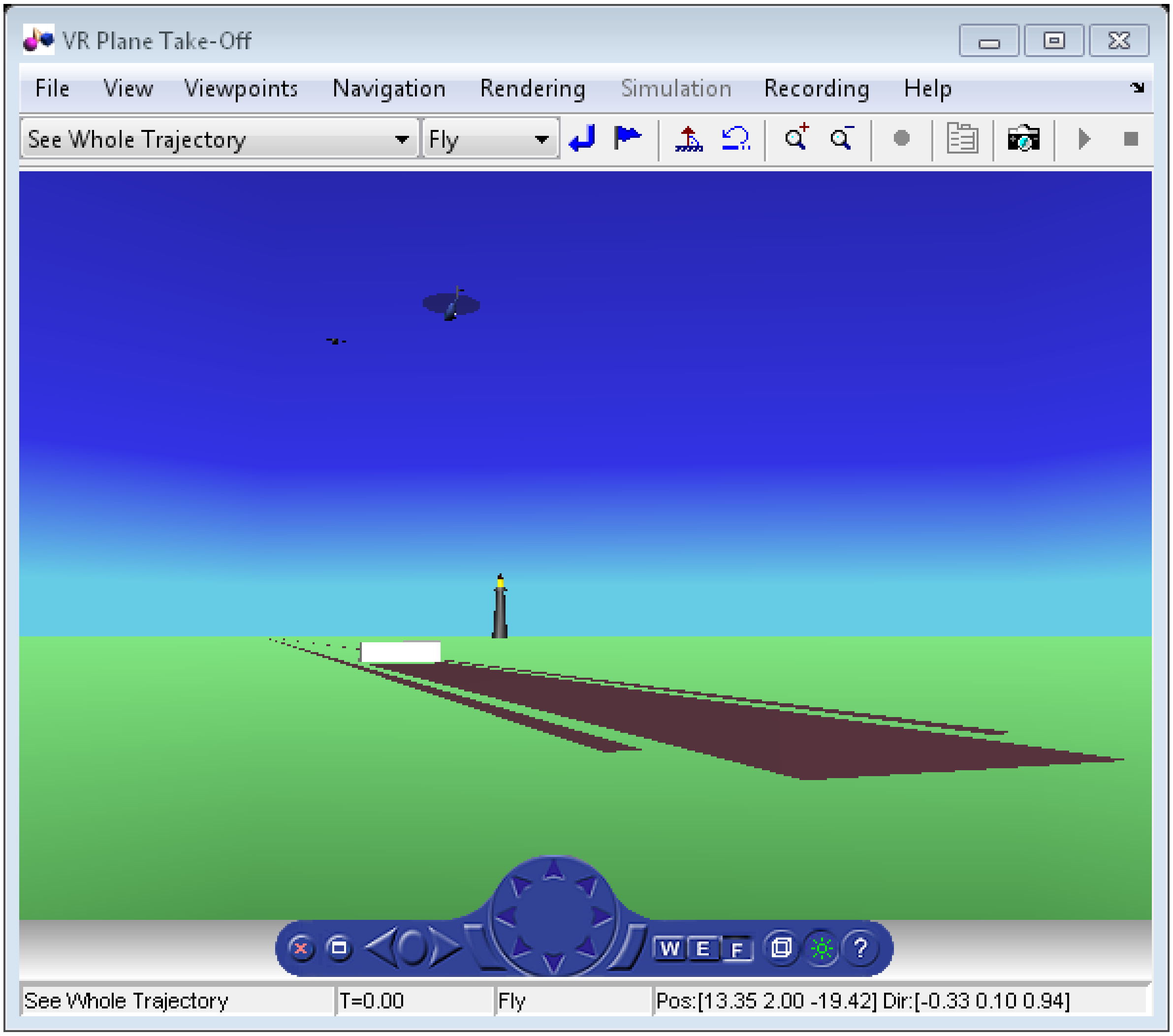
```
[~, idxLynx1] = find(strcmp('Lynx1',h.nodeInfo));
h.Node{idxLynx1}.VRNode.translation = [0 1.3 0];
```

Remove Body

This code uses the `removeNode` method to remove the second helicopter. `removeNode` takes either the node name or node index (as obtained from `nodeInfo`). The associated inline node is removed as well.

```
h.removeNode('Lynx1');
h.nodeInfo
```

```
Node Information
1    Camera1
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
9    Lynx
10    Lynx_Inline
11    chaseView
```

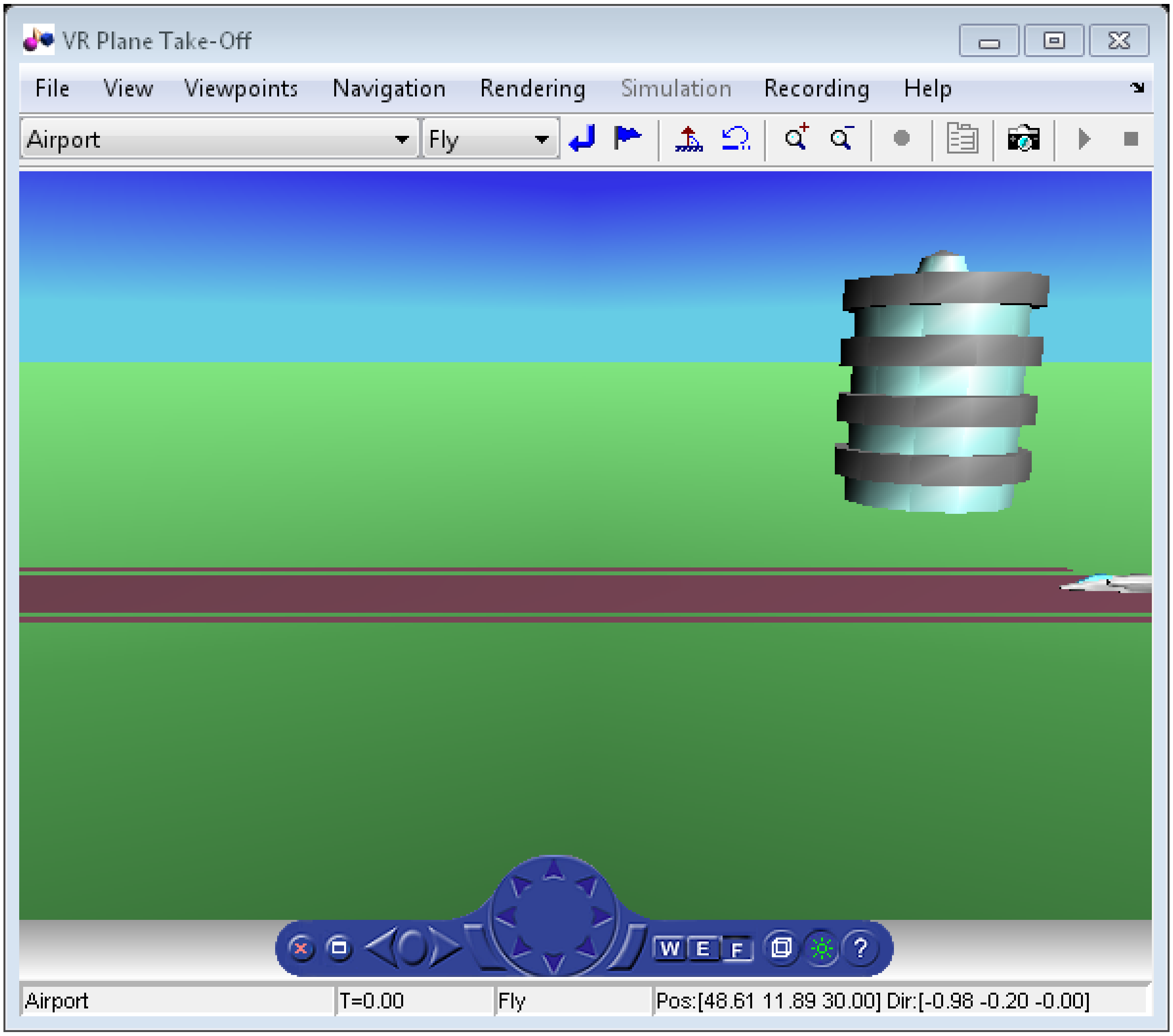


See Whole Trajectory | T=0.00 | Fly | Pos:[13.35 2.00 -19.42] Dir:[-0.33 0.10 0.94]

Revert To Original World

The original filename is stored in the `'VRWorldOldFilename'` property of the animation object. To bring up the original world, set `'VRWorldFilename'` to the original name and reinitializing it.

```
h.VRWorldFilename = h.VRWorldOldFilename{1};
h.initialize();
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F

Airport T=0.00 Fly Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.00]

Close and Delete World

To close and `delete`

```
h.delete();
```

Aero.FlightGearAnimation Objects

The Aerospace Toolbox interface to the FlightGear flight simulator enables you to visualize flight data in a three-dimensional environment. The third-party FlightGear simulator is an open source software package available through a GNU® General Public License (GPL). This section describes how to obtain and install the third-party FlightGear flight simulator. It also describes how to play back 3-D flight data by using a FlightGear example, provided with your Aerospace Toolbox software.

About the FlightGear Interface

The FlightGear flight simulator interface included with the Aerospace Toolbox product is a unidirectional transmission link from the MATLAB software to FlightGear. It uses FlightGear's published `net_fdm` binary data exchange protocol. Data is transmitted via UDP network packets to a running instance of FlightGear. The toolbox supports multiple standard binary distributions of FlightGear. For interface details, see “Flight Simulator Interface Example” on page 2-57.

FlightGear is a separate software entity that is not created, owned, or maintained by MathWorks.

- To report bugs in or request enhancements to the Aerospace Toolbox FlightGear interface, contact MathWorks technical support at https://www.mathworks.com/support/.
- To report bugs or request enhancements to FlightGear itself, go to `www.flightgear.org` and use the contact page.

Supported FlightGear Versions

The Aerospace Toolbox product supports the following FlightGear versions:

- `v2018.3`
- `v2018.2`
- `v2018.1`
- `v2017.3`
- `v2017.1`
- `v2016.3`
- `v2016.1`

- `v3.4`
- `v3.2`
- `v3.0`
- `v2.12`
- `v2.10`
- `v2.8`
- `v2.6`
- `v2.4`
- `v2.0`

Obtaining FlightGear Software

You can obtain FlightGear software from `www.flightgear.org` in the download area or by ordering CDs from FlightGear. The download area contains extensive documentation for installation and configuration. Because FlightGear is an open source project, source downloads are also available for customization and porting to custom environments.

Configuring Your Computer for FlightGear

You must have a high-performance graphics card with stable drivers to use FlightGear. For more information, see the FlightGear CD distribution or the hardware requirements and documentation areas of the FlightGear website, `www.flightgear.org`.

Setup on Linux, Mac OS X, and Other Platforms

FlightGear distributions are available for Linux®, Mac OS X, and other UNIX® platforms from the FlightGear website, `www.flightgear.org`. Installation on these platforms, like Windows®, requires careful configuration of graphics cards and drivers. Consult the documentation and hardware requirements sections at the FlightGear website.

FlightGear and Video Cards in Windows Systems

Your computer built-in video card, such as NVIDIA®cards, can have issues working with FlightGear shaders. Consider this workaround:

- Disable the FlightGear shaders by specifying the `DisableShaders` property of the `Aero.FlightGearAnimation` object to the `GenerateRunScript (Aero.FlightGearAnimation)` method.

Install and Start FlightGear

The extensive FlightGear documentation guides you through the installation. For complete installation instructions, consult the documentation section of the FlightGear website `www.flightgear.org`.

Note:

- Generous central processor speed, system and video RAM, and virtual memory are essential for good flight simulator performance.

  For more information, see `http://wiki.flightgear.org/Hardware_recommendations`.
- Have sufficient disk space for the FlightGear download and installation.
- Before you install FlightGear, configure your computer graphics card. See the preceding section, “Configuring Your Computer for FlightGear” on page 2-54.
- Before installing FlightGear, shut down all running applications (including the MATLAB software).
- Install FlightGear in a folder path name composed of ASCII characters.
- The operational stability of FlightGear is especially sensitive during startup. It is best to not move, resize, mouse over, overlap, or cover up the FlightGear window until the initial simulation scene appears after the startup splash screen fades out.
- The current releases of FlightGear are optimized for flight visualization at altitudes below 100,000 feet. FlightGear does not work well or at all with very high altitude and orbital views.

The Aerospace Toolbox product supports FlightGear on a number of platforms (System Requirements). The following table lists the properties to be aware of before you start using FlightGear.

FlightGear Property	Folder Description	Platforms	Typical Location
`FlightGearBase-Directory`	FlightGear installation folder.	Windows	`C:\Program Files\FlightGear` (default)
		Linux	Directory into which you installed FlightGear

FlightGear Property	Folder Description	Platforms	Typical Location
		Mac	`/Applications` (folder into which you dragged the FlightGear icon)
`GeometryModelName`	Model geometry folder	Windows	`C:\Program Files\FlightGear\data\Aircraft\HL20` (default)
		Linux	`$FlightGearBaseDirectory/-data/Aircraft/HL20`
		Mac	`$FlightGearBaseDirectory/-FlightGear.app/Contents/-Resources/data/Aircraft/-HL20`

Installing Additional FlightGear Scenery

When you install the FlightGear software, the installation provides a basic level of scenery files. The FlightGear documentation guides you through installing scenery as part the general FlightGear installation.

If you need to install more FlightGear scenery files, see the instructions at `http://www.flightgear.org`. Those instructions describe how to install the additional scenery in a default location.

If you install additional scenery in a non-standard location, you may need to update the `FG_SCENERY` environment variable in the script output from the `GenerateRunScript` function to include the new path. For a description of the `FG_SCENERY` variable, see the documentation at `http://www.flightgear.org`.

If you do not download scenery in advance, you can direct FlightGear to download it automatically during simulation using the `InstallScenery` property of the `Aero.FlightGearAnimation` object for the `GenerateRunScript (Aero.FlightGearAnimation)` method.

For Windows systems, you may encounter an error message while launching FlightGear with the `InstallScenery` option enabled:

```
Error creating directory: No such file or directory
```

This error likely indicates that your default FlightGear download folder is not writeable, the path cannot be resolved, or the path contains UNC path names. To work around the issue, edit the `runfg.bat` file to specify a new folder path to store the scenery data:

1. Edit `runfg.bat`.
2. To the list of command options, append `--download-dir=` and specify a folder to which to download the scenery data. For example:

   ```
   --download-dir=C:\Users\user1\Documents\FlightGear
   ```

   All data downloaded during this FlightGear session is saved to the specified directory. To avoid downloading duplicate scenery data, use the same directory in succeeding FlightGear sessions
3. To open FlightGear, run `runfg.bat`.

Note Each time that you run the `GenerateRunScript` function, it creates a new script. It overwrites any edits that you have added.

Flight Simulator Interface Example

The Aerospace Toolbox product provides an example named Displaying Flight Trajectory Data. This example shows you how you can visualize flight trajectories with FlightGear Animation object. The example is intended to be modified depending on the particulars of your FlightGear installation. Use this example to play back your own 3-D flight data with FlightGear.

Before attempting to simulate this model, you must have FlightGear installed and configured. See “About the FlightGear Interface” on page 2-53.

To run the example:

- Import the aircraft geometry into FlightGear.
- Run the example. The example performs the following steps:
  - Loads recorded trajectory data.
  - Creates a time series object from trajectory data.
  - Creates a FlightGearAnimation object.
- Modify the animation object properties, if needed.

- Create a run script for launching the FlightGear flight simulator.
- Start the FlightGear flight simulator.
- Play back the flight trajectory.

Import the Aircraft Geometry into FlightGear

Before running the example, copy the aircraft geometry model into FlightGear. From the following procedures, choose the one appropriate for your platform. This section assumes that you have read “Install and Start FlightGear” on page 2-55.

If your platform is Windows:

1. Go to your installed FlightGear folder. Open the `data` folder, and then the `Aircraft` folder: `*FlightGear*\data\Aircraft\`.
2. If you have previously run the Aerospace Blockset NASA HL-20 with FlightGear Interface example, you might already have an `HL20` subfolder there.

   Otherwise, copy the `HL20` folder from the *`matlabroot`*`\toolbox\aero\astdemos\` folder to the *`FlightGear`*`\data\Aircraft\` folder. This folder contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file *`matlabroot`*`\toolbox\aero\aerodemos\HL20\models\HL20.xml` defines the geometry.

If your platform is Linux:

1. Go to your installed FlightGear folder. Open the `data` folder, then the `Aircraft` folder: *`$FlightGearBaseDirectory`*`/data/Aircraft/`.
2. If you have previously run the Aerospace Blockset NASA HL-20 with FlightGear Interface example, you might already have an `HL20` subfolder there. If that is the case, you do not have to do anything, because you can use the existing geometry model.

   Otherwise, copy the `HL20` folder from the *`matlabroot`*`/toolbox/aero/aerodemos/` folder to the *`$FlightGearBaseDirectory`*`/data/Aircraft/` folder. This folder contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file *`matlabroot`*`/toolbox/aero/aerodemos/HL20/models/HL20.xml` defines the geometry.

If your platform is Mac:

1 Open a terminal.
2 List the contents of the Aircraft folder. For example, type:

```
ls $FlightGearBaseDirectory/data/Aircraft/
```

3 If you have previously run the Aerospace Blockset NASA HL-20 with FlightGear Interface example, you might already have an `HL20` subfolder there. In this case, you do not have to do anything, because you can use the existing geometry model. Continue to “Running the Example” on page 2-24.

   Otherwise, copy the `HL20` folder from the

   ```
   matlabroot/toolbox/aero/aerodemos/
   ```

   folder to the

   ```
   $FlightGearBaseDirectory/FlightGear.app/Contents/Resources/data/Aircraft/
   ```

   folder. This folder contains the preconfigured geometries for the HL-20 simulation and `HL20-set.xml`. The file *`matlabroot`*`/toolbox/aero/aerodemos/HL20/models/HL20.xml` defines the geometry.

Running the Example

1 Start the MATLAB software.
2 Enter `astfganim` in the MATLAB Command Window.

   While running, the example performs several steps by issuing a series of commands.

Flight Trajectory Data

Loading Recorded Flight Trajectory Data

The flight trajectory data for this example is stored in a comma separated value formatted file. Using `dlmread`, the data is read from the file starting at row 1 and column 0, which omits the header information.

```
tdata = dlmread('asthl20log.csv',',',1,0);
```

Creating a Time Series Object from Trajectory Data

The time series object, `ts`, is created from the latitude, longitude, altitude, Euler angle data, and the time array in `tdata` using the MATLAB `timeseries` command. Latitude, longitude, and Euler angles are also converted from degrees to radians using the `convang` function.

```
ts = timeseries([convang(tdata(:,[3 2]),'deg','rad') ...
        tdata(:,4) convang(tdata(:,5:7),'deg','rad')],tdata(:,1));
```

Creating a FlightGearAnimation Object

This series of commands creates a FlightGearAnimation object:

1. Open a FlightGearAnimation object.

   ```
   h = fganimation;
   ```
2. Set FlightGearAnimation object properties for the time series.

   ```
   h.TimeseriesSourceType = 'Timeseries';
   h.TimeseriesSource = ts;
   ```
3. Set FlightGearAnimation object properties relating to FlightGear. These properties include the path to the installation folder, the version number, the aircraft geometry model, and the network information for the FlightGear flight simulator.

   ```
   h.FlightGearBaseDirectory = 'C:\Program Files\FlightGear20183';
   h.FlightGearVersion = '2018.3';
   h.GeometryModelName = 'HL20';
   h.DestinationIpAddress = '127.0.0.1';
   h.DestinationPort = '5502';
   ```

4 Set the initial conditions (location and orientation) for the FlightGear flight simulator.

```
h.AirportId = 'KSFO';
h.RunwayId = '10L';
h.InitialAltitude = 7224;
h.InitialHeading = 113;
h.OffsetDistance = 4.72;
h.OffsetAzimuth = 0;
```

5 Set the seconds of animation data per second of wall-clock time.

```
h.TimeScaling = 5;
```

6 Check the FlightGearAnimation object properties and their values.

```
get(h)
```

The example stops running and returns the FlightGearAnimation object, `h`:

```
          TimeseriesSource: [1x1 timeseries]
      TimeseriesSourceType: 'Timeseries'
         TimeseriesReadFcn: @TimeseriesRead
               TimeScaling: 5
           FramesPerSecond: 12
         FlightGearVersion: '2018.3'
            OutputFileName: 'runfg.bat'
   FlightGearBaseDirectory: 'C:\Program Files\FlightGear20183'
         GeometryModelName: 'HL20'
      DestinationIpAddress: '127.0.0.1'
           DestinationPort: '5502'
                 AirportId: 'KSFO'
                  RunwayId: '10L'
           InitialAltitude: 7224
            InitialHeading: 113
            OffsetDistance: 4.7200
             OffsetAzimuth: 0
                    TStart: NaN
                    TFinal: NaN
              Architecture: 'Default'
```

You can now set the object properties for data playback (see “Modifying the FlightGearAnimation Object Properties” on page 2-62).

Modifying the FlightGearAnimation Object Properties

Modify the FlightGearAnimation object properties as needed. If your FlightGear installation folder is other than the one in the example (for example, `FlightGear`), modify the `FlightGearBaseDirectory` property by issuing the following command:

```
h.FlightGearBaseDirectory = 'C:\Program Files\FlightGear';
```

Similarly, if you want to use a particular file name for the run script, modify the `OutputFileName` property.

Verify the FlightGearAnimation object properties:

```
get(h)
```

You can now generate the run script (see “Generating the Run Script” on page 2-62).

Generating the Run Script

To start FlightGear with the initial conditions (location, date, time, weather, operating modes) that you want, create a run script by using the `GenerateRunScript` command:

```
GenerateRunScript(h)
```

By default, `GenerateRunScript` saves the run script as a text file named `runfg.bat`. You can specify a different name by modifying the `OutputFileName` property of the FlightGearAnimation object, as described in the previous step.

You do not need to generate the file each time the data is viewed, only when the initial conditions or FlightGear information changes.

You are now ready to start FlightGear (see “Starting the FlightGear Flight Simulator” on page 2-63).

Note The `FlightGearBaseDirectory` and `OutputFileName` properties must be composed of ASCII characters.

Starting the FlightGear Flight Simulator

To start FlightGear from the MATLAB command prompt, use the `system` command to execute the run script. Provide the name of the output file created by GenerateRunScript as the argument:

```
system('runfg.bat &');
```

FlightGear starts in a separate window.

Tip With the FlightGear window in focus, press the **V** key to alternate between the different aircraft views: cockpit, helicopter, chase, and so on.

You are now ready to play back data (see “Playing Back the Flight Trajectory” on page 2-63). If you cannot view scenes, see “Installing Additional FlightGear Scenery” on page 2-56.

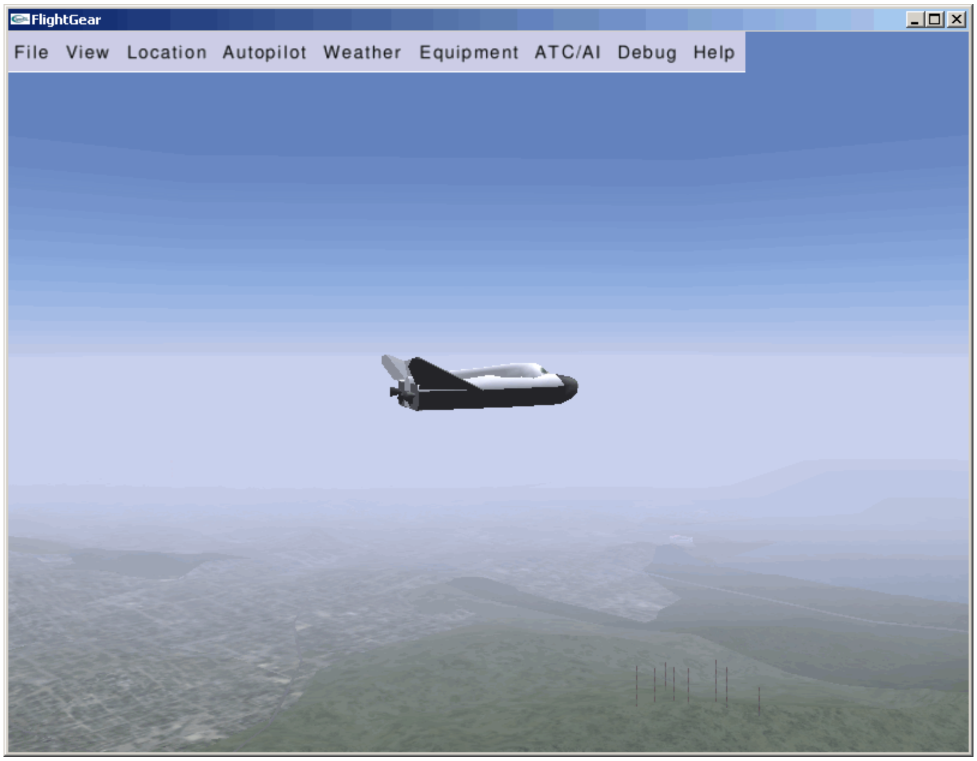
Tip If FlightGear uses more computer resources than you want, you can change its scheduling priority to a lesser one. For example, see commands like Windows `start` and Linux `nice` or their equivalents.

Playing Back the Flight Trajectory

Once FlightGear is running, the FlightGearAnimation object can start to communicate with FlightGear. To animate the flight trajectory data, use the play command:

```
play(h)
```

The following illustration shows a snapshot of flight data playback in tower view without yaw.


FlightGear
File View Location Autopilot Weather Equipment ATC/AI Debug Help

Create and Configure Flight Instrument Component and an Animation Object

You can display flight data using any of the standard flight instrument components:

- Airspeed indicator
- Altimeter
- Climb indicator
- Exhaust gas temperature (EGT) indicator
- Heading indicator
- Artificial horizon
- Revolutions per minute (RPM) indicator
- Turn coordinator

As a general workflow:

1. Load simulation data.
2. Create an animation object.
3. Create a figure window.
4. Create a flight control panel to contain the flight instrument components.
5. Create the flight instrument components.
6. Trigger a display of the animation in the instrument panel.

Note Use Aerospace Toolbox flight instruments only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

Load and Visualize Data

To load and visualize data, consider this workflow:

1. Load simulation data. For example, the `simdata` variable contains logged simulated flight trajectory data.

   ```
   load simdata
   ```

2 To visualize animation data, create an animation object. For example:

   a Create an `Aero.Animation` object.

   ```
   h = Aero.Animation;
   ```

   b Create a body using the `pa24-250_orange.ac` AC3D file and its associated patches.

   ```
   h.createBody('pa24-250_orange.ac','Ac3d');
   ```

   c Set up the bodies of the animation object `h`. Set the `TimeSeriesSource` property to the loaded `simdata`.

   ```
   h.Bodies{1}.TimeSeriesSource = simdata;
   ```

   d Set up the camera and figure positions.

   ```
   h.Camera.PositionFcn = @staticCameraPosition;
   h.Figure.Position(1) = h.Figure.Position(1) + 572/2;
   ```

   e Create and show the figure graphics object for `h`.

   ```
   h.updateBodies(simdata(1,1));
   h.updateCamera(simdata(1,1));
   h.show();
   ```

To create the flight instrument components, see “Create Flight Instrument Components” on page 2-66

Create Flight Instrument Components

This workflow assumes that you have loaded data and created an animation object as described in “Load and Visualize Data” on page 2-65.

1 Create a `uifigure` figure window. This example creates `fig`, to contain the flight instrument for `h`.

```
fig = uifigure('Name','Flight Instruments',...
'Position',[h.Figure.Position(1)-572 h.Figure.Position(2)+h.Figure.Position(4)-502 572 502],...
'Color',[0.2667 0.2706 0.2784],'Resize','off');
```

2 Create a flight instrument panel image for the flight instruments and save it as a graphic file, such as a PNG file.

3 Read the flight instrument panel image into MATLAB and create and load it into UI axes in App Designer using the `uiaxes` function. To display the flight instrument panel image in the current axes, use the `image` function. For example:

```
imgPanel = imread('astFlightInstrumentPanel.png');
ax = uiaxes('Parent',fig,'Visible','off','Position',[10 30 530 460],...
'BackgroundColor',[0.2667 0.2706 0.2784]);
image(ax,imgPanel);
```

4. Create a flight instruments component. For example, create an artificial horizon component. Specify the parent object as the `uifigure` and the position and size of the artificial horizon.

   ```
   hor = uiaerohorizon('Parent',fig,'Position',[212 299 144 144]);
   ```

5. To trigger a display of the animation in the instrument panel, you must input a time step. For example, connect a time input device such as a slider or knob that can change the time. As you change the time on the time input device, the flight instrument component updates to show the result. This example uses the `uislider` function to create a slider component.

   ```
   sl = uislider('Parent',fig,'Limits',[simdata(1,1),...
   simdata(end,1)],'FontColor','white');
   sl.Position = [50 60 450 3];
   ```

6. The slider component has a `ValueChangingFcn` callback, which executes when you move the slider thumb. To update the flight instruments and animation figure, assign the `ValueChangingFcn` callback to a helper function. This example uses the `astHelperFlightInstrumentsAnimation` helper function.

   ```
   sl.ValueChangingFcn = @(sl,event)
   astHelperFlightInstrumentsAnimation(sl,fig,simdata,h);
   ```

7. To display the time selected in the slider, use the `uilabel` function to create a label component. This code creates the label text in white and places the label at position [230 10 90 30].

   ```
   lbl = uilabel('Parent',fig,'Text',['Time: ' num2str(sl.Value,4) ' sec'],'FontColor','white');
   lbl.Position = [230 10 90 30];
   ```

For a complete example, see Display Flight Trajectory Data Using Flight Instruments and Flight Animation.

See Also

Functions
`imread` | `uiaeroairspeed` | `uiaeroaltimeter` | `uiaeroclimb` | `uiaeroegt` | `uiaeroheading` | `uiaerohorizon` | `uiaerorpm` | `uiaeroturn` | `uiaxes` | `uifigure` | `uilabel` | `uislider`

Properties
AirspeedIndicator Properties | Altimeter Properties | ArtificialHorizon Properties | ClimbIndicator Properties | EGTIndicator Properties | HeadingIndicator Properties | RPMIndicator Properties | TurnCoordinator Properties

More About

- “Flight Instruments”
- “Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145
- “Flight Instrument Components in App Designer” on page 2-69

Flight Instrument Components in App Designer

Create aerospace-specific applications in App Designer using common aircraft flight instruments. App Designer is a rich development environment that provides layout and code views, a fully integrated version of the MATLAB editor, and a large set of interactive components. For more information on App Designer, see “App Designer” (MATLAB). To use the flight instrument components in App Designer, you must have an Aerospace Toolbox license.

For a simple flight instruments app example that uses the App Designer, see the Getting Started examples when you first start App Designer. To create an app to visualize saved flight data for a Piper PA-24 Comanche, use this workflow.

1. Start App Designer by typing `appdesigner` at the command line, and then select **Blank App** on the Getting Started page.
2. Drag aerospace components from the **Component Library** to the app canvas.
3. To load simulation data, add a `startup` function to the app, and then create an animation object.
4. Enter the callbacks, functions, and properties for the components to the app. Also add associated code.
5. Trigger a display of the animation in the instrument app.
6. Save and run the app.

The following topics contain more detailed steps for this workflow as an example. This example uses an `Aero.Animation` object.

Start App Designer and Create a New App

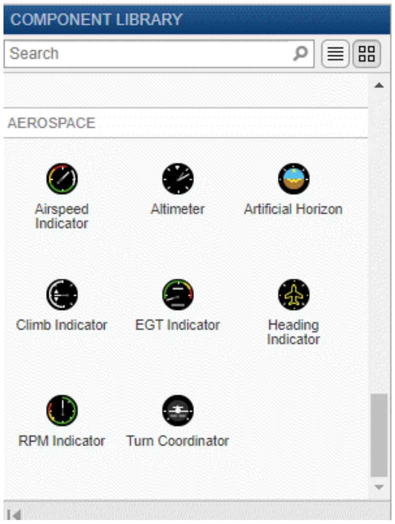
1. Start App Designer. In the MATLAB Command Window, type:

   ```
   appdesigner
   ```
2. In the App Designer welcome window, click **Blank App**. App Designer displays with a blank canvas.
3. To look at the blank app template, click **Code View**. Notice that the app contains a template with sections for app component properties, component initialization, and app creation and deletion.
4. To return to the canvas view, click **Design View**.

Drag Aerospace Components into the App

To add components to the blank canvas:

1. In the **Component Library**, navigate to Aerospace.



2. From the library, drag these aerospace components to the canvas:

   - Airspeed Indicator
   - Artificial Horizon
   - Turn Coordinator
   - Heading Indicator
   - Climb Indicator
   - Altimeter
3. This example uses an `Aero.Animation` object to visualize the flight status of an aircraft over time. To set the current time, add a time input device such as a slider or knob. As you change the time on the time input device, the flight instrument components and the animation window update to show the results. The example code provides further details.

   For this example:

- Add a Slider component as a time input device.
- To display the current time from the slider, edit the label of the slider. For example:
  - Change the label to `Time: 00.0 sec`.
  - Change the upper limit to `50`.

4 Click **Code View** and note that the properties and component initialization sections now contain definitions for the new components. The code managed by App Designer is noneditable (grayed out).

5 In the Property Inspector section on the right of the canvas, rename these components:

- UIfigure component to `FlightInstrumentsFlightDataPlaybackUIFigure`
- Slider component to `Time000secSlider`

Add Code to Load and Visualize Data for the App

This workflow assumes that you have started App Designer, created a blank app, and added aerospace components to the app.

1 In the code view for the app, place the cursor after the properties section and, in the Insert section, click **Callback**.

   The Add Callback Function dialog box displays.

2 In the Add Callback Function dialog box:

   a From the **Callback** list, select `StartupFcn`.

   b In the Name parameter, enter a name for the startup function, for example `startupFcn`.

   A callbacks section is added.

3 Add additional properties to the class for the simulation data and the animation object. Place your cursor just after the component properties section and, in the Insert section, click **Property** > **Public Property**. In the new properties template, add code so that it looks like this:

```
simdata % Saved flight data [time X Y Z phi theta psi]
animObj % Aero.Animation object
```

`simdata` is the saved flight data. `animObj` is the `Aero.Animation` object for the figure window.

4 To the startupFcn section, add code to the `startup` function that loads simulation data. For example, the `simdata.mat` file contains logged simulated flight trajectory data.

```
% Code that executes after component creation
function startupFcn(app)

       % Load saved flight status data
       savedData = load(fullfile(matlabroot, 'toolbox', 'aero', 'astdemos', 'simdata.mat'), 'simdata');
       yaw = savedData.simdata(:,7);
       yaw(yaw<0) = yaw(yaw<0)+2*pi; % Unwrap yaw angles
       savedData.simdata(:,7) = yaw;
       app.simdata = savedData.simdata;  % Load saved flight status data
```

5 To visualize animation data, create an animation object. For example, after loading the simulation data:

   a Create an `Aero.Animation` object.

   ```
   app.animObj = Aero.Animation;
   ```

   b Use the piper pa-24 comanche geometry for the animation object.

   ```
   app.animObj.createBody('pa24-250_orange.ac','Ac3d'); % Piper PA-24 Comanche geometry
   ```

   c Use the data loaded previously, `app.simdata`, as the source for the animation object.

   ```
   app.animObj.Bodies{1}.TimeseriesSourceType = 'Array6DoF'; % [time X Y Z phi the
   app.animObj.Bodies{1}.TimeSeriesSource = app.simdata;
   ```

   d Initialize the camera and figure positions.

   ```
   app.animObj.Camera.PositionFcn = @staticCameraPosition;
   app.animObj.Figure.Position = [app.FlightInstrumentsFlightDataPlaybackUIFigure.Position(1)+625,...
       app.FlightInstrumentsFlightDataPlaybackUIFigure.Position(2),...
       app.FlightInstrumentsFlightDataPlaybackUIFigure.Position(3),...
       app.FlightInstrumentsFlightDataPlaybackUIFigure.Position(4)];
   app.animObj.updateBodies(app.simdata(1,1)); % Initialize animation window at t=0
   app.animObj.updateCamera(app.simdata(1,1));
   ```

   e Create and show the figure graphics object.

   ```
   app.animObj.show();
   ```

Add Code to Trigger a Display of the Animation Object

This workflow assumes that you have added a startup function to the app to load simulation data and create an animation object. To trigger an update of the animation object and flight instruments:

1. In the code view for the app, add a callback for the slider. For example, navigate to the Property Inspector section and select `app.Time000secSlider`.
2. Enter a name for **valueChangingFcn**, for example, `Time000secSliderValueChanging`, and press **Enter**.

   In the code view, App Designer adds a callback function `Time000secSliderValueChanging`.
3. Add code to display the current time in the slider label `Time000secSliderLabel`, for example:

   ```
   % Display current time in slider component
   t = event.Value;
   app.Time000secSliderLabel.Text = sprintf('Time: %.1f sec', t);
   ```
4. Add code to compute data values for each flight instrument component corresponding with the selected time on the slider, for example:

   ```
   % Find corresponding time data entry
   k = find(app.simdata(:,1)<=t);
   k = k(end);

          app.Altimeter.Altitude = convlength(-app.simdata(k,4), 'm', 'ft');
          app.HeadingIndicator.Heading = convang(app.simdata(k,7),'rad','deg');
          app.ArtificialHorizon.Roll = convang(app.simdata(k,5),'rad','deg');
          app.ArtificialHorizon.Pitch = convang(app.simdata(k,6),'rad','deg');

          if k>1
              % Estimate velocity and angular rates
              Vel = (app.simdata(k,2:4)-app.simdata(k-1,2:4))/(app.simdata(k,1)-app.simdata(k-1,1));
              rates = (app.simdata(k,5:7)-app.simdata(k-1,5:7))/(app.simdata(k,1)-app.simdata(k-1,1));

              app.AirspeedIndicator.Airspeed = convvel(sqrt(sum(Vel.^2)),'m/s','kts');
              app.ClimbIndicator.ClimbRate = convvel(-Vel(3),'m/s','ft/min');

              % Estimate turn rate and slip behavior
              app.TurnCoordinator.Turn = convangvel(rates(1)*sind(30) + rates(3)*cosd(30),'rad/s','deg/s');
              app.TurnCoordinator.Slip = 1/(2*pi)*convang(atan(rates(3)*sqrt(sum(Vel.^2))/9.81)-app.simdata(k,5),'rad',
          else
              % time = 0
              app.ClimbIndicator.ClimbRate = 0;
              app.AirspeedIndicator.Airspeed = 0;
              app.TurnCoordinator.Slip = 0;
              app.TurnCoordinator.Turn = 0;
          end
   ```
5. Add code to update the animation window display, for example:

   ```
   %% Update animation window display
   app.animObj.updateBodies(app.simdata(k,1));
   app.animObj.updateCamera(app.simdata(k,1));
   ```

Add Code to Close the Animation Window with UIfigure Window

This workflow assumes that you are ready to define the close function for the `FlightInstrumentsFlightDataPlaybackUIFigure` figure window.

1 Add a `CloseRequestFcn` function. In the code view for the app, place the cursor after the properties section for `FlightInstrumentsFlightDataPlaybackUIFigure` and, in the Insert section, click **Callback**.

   The Add Callback Function dialog box displays.

2 In the Add Callback Function dialog box:

   a From the **Callback** list, select `CloseRequestFcn`.

   b In the Name parameter, enter a name for the close function, for example `FlightInstrumentsFlightDataPlaybackUIFigureCloseRequest`.

      A callbacks section is added.

3 In the new callback template, add code to delete the animation object, such as:

```
% Close animation figure with app
delete(app.animObj);
delete(app);
```

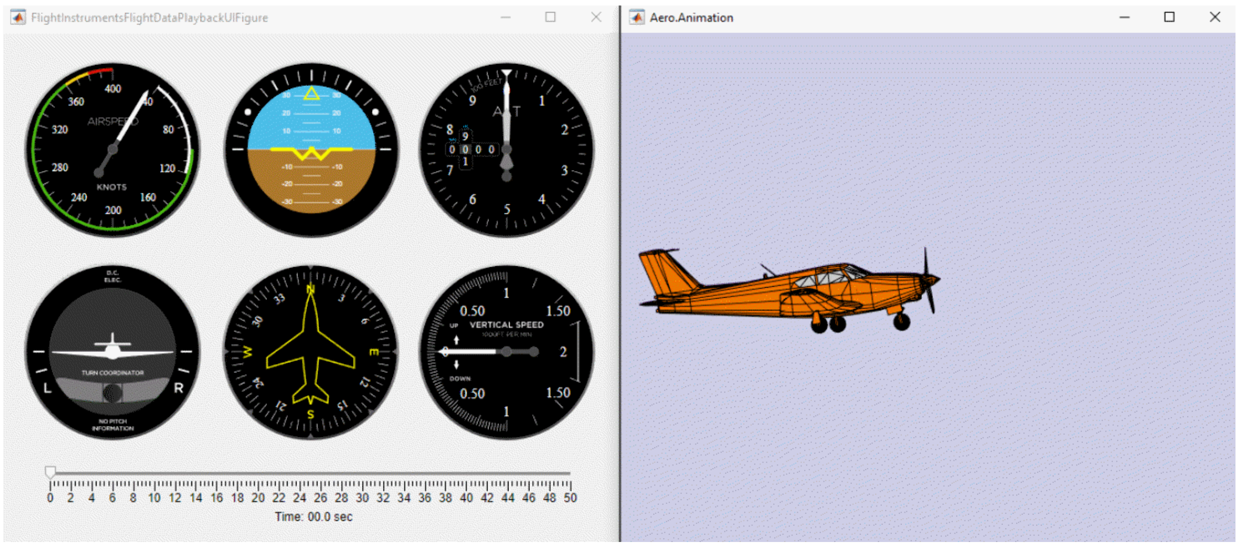
Save and Run the App

This workflow assumes that you have added code to close the uifigure window. To save and run the app:

1 Save the app with the file name `myFlightInstrumentsExample`. Note that this name is applied to the `classdef`.

2 Click **Run**.

   After saving your changes, you can run the app from the App Designer window, or by typing its name (without the `.mlapp` extension) at the MATLAB Command Window. When you run the app from the command prompt, the file must be in the current folder or on the MATLAB path.



3 To visualize the saved flight data, change the slider position. Observe the flight instruments as the aircraft changes orientation in the animation window.

For a complete example, see “Aerospace Flight Instruments in App Designer” on page 5-149.

See Also

Functions

`uiaeroairspeed` | `uiaeroaltimeter` | `uiaeroclimb` | `uiaeroegt` | `uiaeroheading` | `uiaerohorizon` | `uiaerorpm` | `uiaeroturn` | `uiaxes` | `uifigure` | `uilabel` | `uislider`

Properties

AirspeedIndicator Properties | Altimeter Properties | ArtificialHorizon Properties | ClimbIndicator Properties | EGTIndicator Properties | HeadingIndicator Properties | RPMIndicator Properties | TurnCoordinator Properties

More About

- “Create and Configure Flight Instrument Component and an Animation Object” on page 2-65

- “Aerospace Flight Instruments in App Designer” on page 5-149
- “Flight Instruments”
- “App Designer” (MATLAB)

Add-On for Ephemeris and Geoid Data Support

Add Ephemeris and Geoid Data for Aerospace Products

Add ephemeris and/or geoid data to use it with the Aerospace Toolbox functions and Aerospace Blockset blocks. You can add data for these functions and blocks.

Aerospace Toolbox Functions	Aerospace Blockset Blocks
`geoidheight` **Note** Only for the EGM2008 Geopotential Model. Aerospace Toolbox provides EGM96 Geopotential Model data.	Geoid Height **Note** Only for the EGM2008 Geopotential Model. Aerospace Toolbox provides EGM96 Geopotential Model data.
`earthNutation`	Earth Nutation
`moonLibration`	Moon Libration
`planetEphemeris`	Planetary Ephemeris

To add ephemeris and geoid data for these functions and blocks.

1 In a MATLAB Command Window, type:

```
aeroDataPackage
```

The Add-On Explorer starts.

2 Select the data you want to add, for example:

- `Geoid Data for Aerospace Toolbox`
- `Ephemeris Data for Aerospace Toolbox`

3 On the data page, click the **Install** button.

Note You must have write privileges for the folder to which you are adding data.

To check for updates, repeat this process when a new version of MATLAB software is released. You can also check for updates between releases using this process.

4

Alphabetical List

addBody

Class: `Aero.Animation`
Package: `Aero`

Add loaded body to animation object and generate its patches

Syntax

```
idx = addBody(h,b)
idx = h.addBody(b)
```

Description

`idx = addBody(h,b)` and `idx = h.addBody(b)` add a loaded body, `b`, to the animation object `h` and generates its patches. `idx` is the index of the body to be added.

Input Arguments

`h`	Animation object.
`b`	Loaded body.

Output Arguments

`idx`	Index of the body to be added.

Examples

Add a second body to the list that is a pointer to the first body. This means that if you change the properties of one body, the properties of the other body change correspondingly.

```
h = Aero.Animation;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
b = h.Bodies{1};
idx2 = h.addBody(b);
```

addNode (Aero.VirtualRealityAnimation)

Add existing node to current virtual reality world

Syntax

```
addNode(h, node_name, wrl_file)
h.addNode(node_name, wrl_file)
```

Description

`addNode(h, node_name, wrl_file)` and `h.addNode(node_name, wrl_file)` add an existing node, `node_name`, to the current virtual reality world. The `wrl_file` is the file from which the new node is taken. `addNode` adds a new node named `node_name`, which contains (or points to) the `wrl_file`. `node_name` must be unique from other node names in the same `.wrl` file. `wrl_file` must contain the node to be added. You must specify the full path for this file. The vrnode object associated with the node object must be defined using a `DEF` statement in the `.wrl` file. This method creates a node object on the world of type Transform.

When you use the `addNode` method to add a node, all the objects in the `.wrl` file will be added to the virtual reality animation object under one node. If you want to add separate nodes for the objects in the `.wrl` file, place each node in a separate `.wrl` file.

Examples

Add node to world defined in `chaseHelicopter.wrl`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
h.addNode('Lynx',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
```

See Also

`Aero.Node` | `Aero.VirtualRealityAnimation` | `move` | `removeNode` | `updateNodes`

Introduced in R2007b

addRoute (Aero.VirtualRealityAnimation)

Add VRML ROUTE statement to virtual reality animation

Syntax

```
addRoute(h, nodeOut, eventOut, nodeIn, eventIn)
h.addRoute(nodeOut, eventOut, nodeIn, eventIn)
```

Description

`addRoute(h, nodeOut, eventOut, nodeIn, eventIn)` and `h.addRoute(nodeOut, eventOut, nodeIn, eventIn)` add a VRML ROUTE statement to the virtual reality animation, where `nodeOut` is the node from which information is routed, `eventOut` is the event (property), `nodeIn` is the node to which information is routed, and `eventIn` is the receiving event (property).

Examples

Add a ROUTE command to connect the Plane position to the Camera1 node.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
h.addNode('Lynx',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
h.addRoute('Plane','translation','Camera1','translation');
```

See Also

`addViewpoint`

Introduced in R2007b

addViewpoint (Aero.VirtualRealityAnimation)

Add viewpoint for virtual reality animation

Syntax

```
addViewpoint(h, parent_node, parent_field, node_name)
h.addViewpoint(parent_node, parent_field, node_name)
addViewpoint(h, parent_node, parent_field, node_name, description)
h.addViewpoint(parent_node, parent_field, node_name, description)
addViewpoint(h, parent_node, parent_field, node_name, description,
position)
h.addViewpoint(parent_node, parent_field, node_name, description,
position)
addViewpoint(h, parent_node, parent_field, node_name, description,
position, orientation)
h.addViewpoint(parent_node, parent_field, node_name, description,
position, orientation)
```

Description

`addViewpoint(h, parent_node, parent_field, node_name)` and `h.addViewpoint(parent_node, parent_field, node_name)` add a viewpoint named `node_name` whose `parent_node` is the parent node field of the vrnode object and whose `parent_field` is a valid parent field of the vrnode object to the virtual world animation object, `h`.

`addViewpoint(h, parent_node, parent_field, node_name, description)` and `h.addViewpoint(parent_node, parent_field, node_name, description)` add a viewpoint named `node_name` whose `parent_node` is the parent node field of the vrnode object and whose `parent_field` is a valid parent field of the vrnode object to the virtual world animation object, `h`. `description` is the character vector or string you want to describe the viewpoint.

`addViewpoint(h, parent_node, parent_field, node_name, description, position)` and `h.addViewpoint(parent_node, parent_field, node_name,` `description, position)` add a viewpoint named `node_name` whose `parent_node` is the parent node field of the vrnode object and whose `parent_field` is a valid parent field of the vrnode object to the virtual world animation object, `h`. `description` is the character vector or string you want to describe the viewpoint and `position` is the position of the viewpoint. Specify `position` using VRML coordinates (*x y z*).

`addViewpoint(h, parent_node, parent_field, node_name, description, position, orientation)` and `h.addViewpoint(parent_node, parent_field, node_name, description, position, orientation)` add a viewpoint named `node_name` whose `parent_node` is the parent node field of the vrnode object and whose `parent_field` is a valid parent field of the vrnode object to the virtual world animation object, `h`. `description` is the character vector or string you want to describe the viewpoint, `position` is the position of the viewpoint, and `orientation` is the orientation of the viewpoint. Specify `position` using VRML coordinates (*x y z*). Specify `orientation` in a VRML axes angle format (*x y z* Θ).

Note If you call `addViewpoint` with only the description argument, you must set the position and orientation of the viewpoint with the Simulink 3D Animation `vrnode/setfield` function. This requires you to use VRML coordinates.

Examples

Add a viewpoint named chaseView.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
h.addViewpoint(h.Nodes{2}.VRNode,'children','chaseView','View From Helicopter');
```

See Also

`addRoute` | `removeViewpoint`

Introduced in R2007b

Aero.Animation class

Package: `Aero`

Visualize aerospace animation

Description

Use the Aero.Animation class to visualize flight data without any other tool or toolbox. You only need the Aerospace Toolbox to visualize this data.

Construction

Aero.Animation	Construct animation object

Methods

addBody	Add loaded body to animation object and generate its patches
createBody	Create body and its associated patches in animation
delete	Destroy animation object
hide	Hide animation figure
initialize	Create animation object figure and axes and build patches for bodies
initIfNeeded	Initialize animation graphics if needed
moveBody	Move body in animation object
play	Animate `Aero.Animation` object given position/angle time series
removeBody	Remove one body from animation
show	Show animation object figure
updateBodies	Update bodies of animation object
updateCamera	Update camera in animation object

Properties

Bodies	Specify name of animation object
Camera	Specify camera that animation object contains
Figure	Specify name of figure object
FigureCustomizationFcn	Specify figure customization function
FramesPerSecond	Animation rate
Name	Specify name of animation object
TCurrent	Current time
TFinal	End time
TimeScaling	Scaling time
TStart	Start time
VideoCompression	Video recording compression file type
VideoFileName	Video recording file name
VideoQuality	Video recording quality
VideoRecord	Video recording
VideoTFinal	Video recording stop time for scheduled recording
VideoTStart	Video recording start time for scheduled recording

See Also

`Aero.FlightGearAnimation` | `Aero.VirtualRealityAnimation`

Topics

"Aero.Animation Objects" on page 2-24

Aero.Animation

Class: `Aero.Animation`
Package: `Aero`

Construct animation object

Syntax

```
h = Aero.Animation
```

Description

`h = Aero.Animation` constructs an animation object. The animation object is returned to `h`.

Note The `Aero.Animation` constructor does not retain the properties of previously created animation objects, even those that you have saved to a MAT-file. This means that subsequent calls to the animation object constructor always create animation objects with default properties.

Examples

```
h=Aero.Animation
```

Aero.Body

Create body object for use with animation object

Syntax

```
h = Aero.Body
```

Description

`h = Aero.Body` constructs a body for an animation object. The animation object is returned in `h`. To use the Aero.Body object, you typically:

1. Create the animation body.
2. Configure or customize the body object.
3. Load the body.
4. Generate patches for the body (requires an axes from a figure).
5. Set time series data source.
6. Move or update the body.

By default, an Aero.Body object natively uses aircraft *x*-*y*-*z* coordinates for the body geometry and the time series data. It expects the rotation order *z*-*y*-*x* (psi, theta, phi).

Convert time series data from other coordinate systems on the fly by registering a different `CoordTransformFcn` function.

Constructor Summary

Constructor	Description
`Body`	Construct body object for use with animation object.

Method Summary

Method	Description
`findstartstoptimes`	Return start and stop times of time series data.
`generatePatches`	Generate patches for body with loaded face, vertex, and color data.
`load`	Get geometry data from source.
`move`	Change Aero.Body position and orientation.
`update`	Changes body position and orientation versus time data.

Property Summary

Property	Description	Values
`CoordTransformFcn`	Specify a function that controls the coordinate transformation.	Character vector | string
`Name`	Specify name of body.	
`Position`	Specify position of body.	MATLAB array
`Rotation`	Specify rotation of body.	MATLAB array
`Geometry`	Specify geometry of body.	handle
`PatchGenerationFcn`	Specify patch generation function.	MATLAB array
`PatchHandles`	Specify patch handles.	MATLAB array
`ViewingTransform`	Specify viewing transform.	MATLAB array
`TimeseriesSource`	Specify time series source.	MATLAB array

Property	Description	Values
`TimeseriesSource-Type`	Specify the type of time series data stored in `'TimeseriesSource'`. Five values are available. They are listed in TimeseriesSourceType Properties. The default value is `'Array6DoF'`.	Character vector | string
`TimeseriesReadFcn`	Specify time series read function.	MATLAB array

The time series data, stored in the property `'TimeseriesSource'`, is interpreted according to the `'TimeseriesSourceType'` property, which can be one of:

TimeseriesSourceType Properties

Property	Description
`'Timeseries'`	MATLAB time series data with six values per time: `x y z phi theta psi` The values are resampled.
`'StructureWithTime'`	Simulink struct with time (for example, Simulink root outport logging `'Structure with time'`): • `signals(1).values: x y z` • `signals(2).values: phi theta psi` Signals are linearly interpolated vs. time using `interp1`.
`'Array6DoF'`	A double-precision array in `n` rows and 7 columns for 6-DoF data: `time x y z phi theta psi`. If a double-precision array of 8 or more columns is in `'TimeseriesSource'`, the first 7 columns are used as 6-DoF data.
`'Array3DoF'`	A double-precision array in `n` rows and 4 columns for 3-DoF data: `time x z theta`. If a double-precision array of 5 or more columns is in `'TimeseriesSource'`, the first 4 columns are used as 3-DoF data.
`'Custom'`	Position and angle data is retrieved from `'TimeseriesSource'` by the currently registered `'TimeseriesReadFcn'`.

See Also

`Aero.Geometry`

Introduced in R2007a

Aero.Camera

Construct camera object for use with animation object

Syntax

```
h = Aero.Camera
```

Description

`h = Aero.Camera` constructs a camera object `h` for use with an animation object. The camera object uses the registered coordinate transform. By default, this is an aerospace body coordinate system. Axes of custom coordinate systems must be orthogonal.

By default, an `Aero.Body` object natively uses aircraft *`x-y-z`* coordinates for the body geometry and the time series data. Convert time series data from other coordinate systems on the fly by registering a different `CoordTransformFcn` function.

For more information, see:

- “Overlaying Simulated and Actual Flight Data” on page 5-40
- “Camera Graphics Terminology” (MATLAB)
- “Low-Level Camera Properties” (MATLAB)

Constructor Summary

Constructor	Description
`Camera`	Construct camera object for use with animation object.

Method Summary

Method	Description
update	Update camera position based on time and position of other Aero.Body objects.

Property Summary

Property	Description	Values
CoordTransformFcn	Specify a function that controls the coordinate transformation.	MATLAB array
PositionFcn	Specify a function that controls the position of a camera relative to an animation body.	MATLAB array
Position	Specify position of camera.	MATLAB array [-150,-50,0]
Offset	Specify offset of camera.	MATLAB array [-150,-50,0]
AimPoint	Specify aim point of camera.	MATLAB array [0,0,0]
UpVector	Specify up vector of camera.	MATLAB array [0,0,-1]
ViewAngle	Specify view angle of camera.	MATLAB array {3}
ViewExtent	Specify view extent of camera.	MATLAB array {[-50,50]}
xlim	Specify *x*-axis limit of camera.	MATLAB array {[-50,50]}
ylim	Specify *y*-axis limit of camera.	MATLAB array {[-50,50]}
zlim	Specify *z*-axis limit of camera.	MATLAB array {[-50,50]}
PrevTime	Specify previous time of camera.	MATLAB array {0}
UserData	Specify custom data.	MATLAB array {[]}

See Also

`Aero.Geometry`

Introduced in R2007a

aeroDataPackage

Start Add-On Explorer to download, install, or uninstall aerospace-specific data

Syntax

`aeroDataPackage`

Description

`aeroDataPackage` opens Add-On Explorer. To see a list of available data, run Add-On Explorer and select the data you want.

Examples

Start Add-On Explorer

Start Add-On Explorer to add data.

```
aeroDataPackage
```

The Add-On Explorer starts. Follow the instructions to download your data.

Limitations

The `aeroDataPackage` function is not available for Aerospace Toolbox Online.

See Also

Topics

“Add Ephemeris and Geoid Data for Aerospace Products” on page 3-2

Introduced in R2014a

Aero.FlightGearAnimation

Construct FlightGear animation object

Syntax

`h = Aero.FlightGearAnimation`

Description

`h = Aero.FlightGearAnimation` constructs a FlightGear animation object. The FlightGear animation object is returned to `h`.

Limitations

These capabilities are not available for Aerospace Toolbox Online:

- The `Aero.FlightGearAnimation` object
- The related example, “Create a Flight Animation from Trajectory Data” on page 5-21

Constructor

Method	Description
`fganimation`	Construct FlightGear animation object.

Method Summary

Method	Description
`ClearTimer`	Clear and delete timer for animation of FlightGear flight simulator.
`delete`	Destroy FlightGear animation object.

Method	Description
GenerateRunScript	Generate run script for FlightGear flight simulator.
initialize	Set up FlightGear animation object.
play	Animate FlightGear flight simulator using given position/angle time series.
SetTimer	Set name of timer for animation of FlightGear flight simulator.
update	Update position data to FlightGear animation object.

Property Summary

Properties	Description
TimeseriesSource	Specify variable that contains the time series data.
TimeseriesSource-Type	Specify the type of time series data stored in 'TimeseriesSource'. Five values are available. They are listed in TimeseriesSourceType Properties. The default value is 'Array6DoF'.
TimeseriesReadFcn	Specify a function to read the time series data if 'TimeseriesSourceType' is 'Custom'.
TimeScaling	Specify the seconds of animation data per second of wall-clock time. The default ratio is 1.
FramesPerSecond	Specify the number of frames per second used to animate the 'TimeseriesSource'. The default value is 12 frames per second.

Properties	Description
`FlightGearVersion`	Select your FlightGear software version: `v2018.1`, `v2017.3`, `v2017.1`, `v2016.3`, `v2016.1`, `v3.4`, `v3.2`, `v3.0`, `v2.12`, `v2.10`, `v2.8`, `v2.6`, `v2.4`, or `'2.0'`. The default version is the latest version. **Note** If you are using a FlightGear version older than 2.0, the software returns a warning when you use the `initialize` method. Consider upgrading your FlightGear version. For more information, see "Supported FlightGear Versions" on page 2-53.
`OutputFileName`	Specify the name of the output file. The file name is the name of the command you will use to start FlightGear with these initial parameters. The default value is `'runfg.bat'`. **Note** The run script file name must be composed of ASCII characters.
`FlightGearBase-Directory`	Specify the name of your FlightGear installation folder. The default value is `'D:\Applications\FlightGear'`. **Note** FlightGear must be installed in a folder path name composed of ASCII characters.
`GeometryModelName`	Specify the name of the folder containing the desired model geometry in the `*FlightGear*\data\Aircraft` folder. The default value is `'HL20'`.
`DestinationIp-Address`	Specify your destination IP address. The default value is `'127.0.0.1'`.
`DestinationPort`	Specify your network flight dynamics model (fdm) port. This destination port should be an unused port that you can use when you launch FlightGear. The default value is `'5502'`.
`AirportId`	Specify the airport ID. The list of supported airports is available in the FlightGear interface, under **Location**. The default value is `'KSFO'`.
`RunwayId`	Specify the runway ID. The default value is `'10L'`.

Properties	Description
`InitialAltitude`	Specify the initial altitude of the aircraft, in feet. The default value is `7224` feet.
`InitialHeading`	Specify the initial heading of the aircraft, in degrees. The default value is `113` degrees.
`OffsetDistance`	Specify the offset distance of the aircraft from the airport, in miles. The default value is `4.72` miles.
`OffsetAzimuth`	Specify the offset azimuth of the aircraft, in degrees. The default value is `0` degrees.
`TStart`	Specify start time as a double.
`TFinal`	Specify end time as a double.
`Architecture`	Specify the architecture the FlightGear software is running on. `GenerateRunScript` takes this setting into account when generating the bash run script to start FlightGear. The platforms are listed in Architecture Properties. The default value is `'Default'`.

The time series data, stored in the property `'TimeseriesSource'`, is interpreted according to the `'TimeseriesSourceType'` property, which can be one of:

TimeseriesSourceType Properties

Property	Description
`'Timeseries'`	MATLAB time series data with six values per time: `lat lon alt phi theta psi` The values are resampled.
`'StructureWithTime'`	Simulink struct with time (for example, Simulink root outport logging `'Structure with time'`): • `signals(1).values: lat lon alt` • `signals(2).values: phi theta psi` Signals are linearly interpolated vs. time using `interp1`.
`'Array6DoF'`	A double-precision array in `n` rows and 7 columns for 6-DoF data: `time lat lon alt phi theta psi`. If a double-precision array of 8 or more columns is in `'TimeseriesSource'`, the first 7 columns are used as 6-DoF data.
`'Array3DoF'`	A double-precision array in `n` rows and 4 columns for 3-DoF data: `time lat alt theta`. If a double-precision array of 5 or more columns is in `'TimeseriesSource'`, the first 4 columns are used as 3-DoF data.
`'Custom'`	Position and angle data is retrieved from `'TimeseriesSource'` by the currently registered `'TimeseriesReadFcn'`.

Specify one of these values for the `Architecture` property:

Architecture Properties

Property	Description
`'Default'`	Architecture the MATLAB software is currently running on. If the property has this value, `GenerateRunScript` creates a bash file that can work in the architecture that MATLAB is currently running on.
`'Win64'`	Windows (64-bit) architecture.
`'Mac'`	Mac OS X (64-bit) architecture.
`'Linux'`	Linux (64-bit) architecture.

`'Default'`	Architecture the MATLAB software is currently running on. If the property has this value, `GenerateRunScript` creates a bash file that can work in the architecture that MATLAB is currently running on.
`'Win64'`	Windows (64-bit) architecture.
`'Mac'`	Mac OS X (64-bit) architecture.
`'Linux'`	Linux (64-bit) architecture.

Examples

Construct a FlightGear animation object, `h`:

```
h = fganimation
```

See Also

`fganimation` | `generaterunscript` | `play`

Introduced in R2007a

aeroReadIERSData

File containing current International Astronomical Union (IAU) 2000A Earth orientation data

Syntax

```
file=aeroReadIERSData(foldername)
file=aeroReadIERSData(foldername,'url',urladdress)
```

Description

`file=aeroReadIERSData(foldername)` creates a MAT-file, `file`, based on IAU 2000A Earth orientation data from the International Earth Rotation and Reference Systems Service (IERS). It saves the file to `foldername`. `file` name has the format `aeroiersdata`*`YYYYMMDD`*`.mat`, where:

- YYYY - Year
- MM - Month
- DD - Day

`file=aeroReadIERSData(foldername,'url',urladdress)` creates the MAT-file based on Earth orientation data from a specific website or data file.

Examples

Create File for Current Day

Create the Earth orientation data file for the current day, in the current folder, using data from the default website https://maia.usno.navy.mil/ser7/finals2000A.data.

```
aeroReadIERSData(pwd)
```

Create File from Specified Website

Create the Earth orientation file for the current day, in the current folder, using data from the alternate website https://datacenter.iers.org/data/latestVersion/10_FINALS.DATA_IAU2000_V2013_0110.txt.

```
aeroReadIERSData(pwd,'url','https://datacenter.iers.org/data/latestVersion/10_FINALS.DATA_IAU2000_V2013_0110.txt')
```

Create File from Specified File

Create the Earth orientation file for the current day, in the current folder, using data from a specified file `file:///C:\Documents\final2000A.data`.

```
aeroReadIERSData(pwd,'url','file:///C:\Documents\finals2000A.data')
```

Input Arguments

`foldername` — Folder for IERS data file
folder name

Folder for IERS data file, specified as a character array or string. Before running this function, create *foldername* with write permission.

Data Types: `char` | `string`

`'url',urladdress` — Optional website or Earth orientation data file
https://maia.usno.navy.mil/ser7/finals2000A.data (default) | website address | file name

Optional website or file containing the IAU 2000A Earth orientation data, specified as a website address or file name.

Example: `https://datacenter.iers.org/data/latestVersion/10_FINALS.DATA_IAU2000_V2013_0110.txt`

Data Types: `char` | `string`

Output Arguments

`file` — Location of Earth orientation data MAT-file
character array

Location of Earth orientation data MAT-file, specified as a character array.

More About

International Astronomical Union (IAU) 2000A Earth Orientation Data Format

The function expects the International Astronomical Union (IAU) 2000A Earth orientation data to use the format referenced here https://maia.usno.navy.mil/ser7/readme.finals2000A:

Column	Description
1 to 2	Year (to get true calendar year, add 1900 for MJD<=51543 or add 2000 for MJD>=51544)
3 to 4	Month number
5 to 6	Day of month
7	Blank
8 to 15	Fractional modified Julian date (MJD UTC)
16	Blank
17	IERS (I) or Prediction (P) flag for Bull. A polar motion values*
18	Blank
19 to 27	Bull. A PM-x (sec. of arc)*
28-36	Error in PM-x (sec. of arc)*
37	Blank
38-46	Bull. A PM-y (sec. of arc)*
47-55	Error in PM-y (sec. of arc)*
56-57	Blank
58	IERS (I) or Prediction (P) flag for Bulletin A UT1-UTC values*

Column	Description
59-68	Bull. A UT1-UTC (sec. of time)*
69-78	Error in UT1-UTC (sec. of time)*
79	Blank
80-86	Bull. A LOD (msec. of time) -- not always filled*
87-93	Error in LOD (msec. of time) -- not always filled*
94-95	Blank
96	IERS (I) or Prediction (P) flag for Bull. A nutation values*
97	Blank
98-106	Bull. A dX wrt IAU2000A nutation (msec. of arc), free core nutation not removed*
107-115	Error in dX (msec. of arc)
116	Blank
117-125	Bull. A dY wrt IAU2000A nutation (msec. of arc), free core nutation not removed*
126-134	Error in dY (msec. of arc)
135-144	Bull. B PM-x (sec. of arc)*
145-154	Bull. B PM-x (sec. of arc)*
155-165	Bull. B UT1-UTC (sec. of time)*
166-175	Bull. B dX wrt IAU2000A nutation (msec. of arc)*
176-185	Bull. B dY wrt IAU2000A nutation (msec. of arc)*

* Abbreviated terms:

- Bull. — Bulletin
- LOD — Length of day
- wrt — With regard to

- pm — Polar motion

See Also

`dcmeci2ecef` | `deltaUT1` | `eci2aer` | `eci2lla` | `lla2eci` | `mjuliandate`

External Websites

https://maia.usno.navy.mil/ser7/finals2000A.data
https://maia.usno.navy.mil/ser7/readme.finals2000A

Introduced in R2017b

Aero.Geometry

Construct 3-D geometry for use with animation object

Syntax

```
h = Aero.Geometry
```

Description

`h = Aero.Geometry` defines a 3-D geometry for use with an animation object.

This object supports the attachment of transparency data from an Ac3d file to patch generation.

Constructor Summary

Constructor	Description
`Geometry`	Construct 3-D geometry for use with animation object.

Method Summary

Method	Description
`read`	Read geometry data using current reader.

Property Summary

Property	Description	Values
`Name`	Specify name of geometry.	Character vector | string

Property	Description	Values	
`Source`	Specify geometry data source.	`{['Auto'], 'Variable', 'MatFile', 'Ac3dFile', 'Custom'}`	
`Reader`	Specify geometry reader.	MATLAB array	
`FaceVertexColorData`	Specify the color of the geometry face vertex.	MATLAB structure with the following fields	
		`name`	Character vector or string that contains the name of the geometry being loaded.
		`faces`	See Faces.
		`vertices`	See Vertices.
		`cdata`	See CData.
		`alpha`	See FaceVertexAlphaData.

See Also

`read`

Introduced in R2007a

Aero.Node

Create node object for use with virtual reality animation

Syntax

```
h = Aero.Node
```

Description

`h = Aero.Node` creates a node object for use with virtual reality animation. Typically, you do not need to create a node object with this method. This is because the `.wrl` file stores the information for a virtual reality scene. During the initialization of the virtual reality animation object, any node with a `DEF` statement in the specified `.wrl` file has a node object created.

When working with nodes, consider the translation and rotation. Translation is a 1-by-3 matrix in the aerospace body coordinate system defined for the VirtualRealityAnimation object or another coordinate system. In the latter case, you can use the `CoordTransformFcn` function to move it into the defined aerospace body coordinate system. The defined aerospace body coordinate system is defined relative to the screen as *x*-left, *y*-in, *z*-down.

Rotation is a 1-by-3 matrix, in radians, that specifies the rotations about the right-hand *x*-*y*-*z* sequence of coordinate axes. The order of application of the rotation is *z*-*y*-*x* (*r*-*q*-*p*). This function uses the `CoordTransformFcn` to apply the translation and rotation from the input coordinate system to the defined aerospace body coordinate system. The function then moves the translation and rotation from the defined aerospace body coordinate system to the defined VRML *x*-*y*-*z* coordinates for the VirtualRealityAnimation object. The defined VRML coordinate system is defined relative to the screen as *x*-right, *y*-up, *z*-out.

Constructor Summary

Constructor	Description
`Node`	Create node object for use with virtual reality animation.

Method Summary

Method	Description
`findstart-stoptimes`	Return start and stop times for time series data.
`move`	Change node translation and rotation.
`update`	Change node position and orientation versus time data.

Property Summary

Property	Description	Values
`Name`	Specify name of the node object.	Character vector | string
`VRNode`	Return the handle to the `vrnode` object associated with the node object.	MATLAB array
`CoordtransformFcn`	Specify a function that controls the coordinate transformation.	MATLAB array
`TimeseriesSource`	Specify time series source.	MATLAB array
`Timeseries-SourceType`	Specify the type of time series data stored in `'TimeseriesSource'`. Five values are available. They are listed in TimeseriesSourceType Properties. The default value is `'Array6DoF'`.	Character vector | string

Property	Description	Values
`TimeseriesReadFcn`	Specify time series read function.	MATLAB array

The time series data, stored in the property `'TimeseriesSource'`, is interpreted according to the `'TimeseriesSourceType'` property, which can be one of:

TimeseriesSourceType Properties

Property	Description
`'Timeseries'`	MATLAB time series data with six values per time: `lat lon alt phi theta psi` The values are resampled.
`'StructureWithTime'`	Simulink struct with time (for example, Simulink root outport logging `'Structure with time'`): • `signals(1).values: lat lon alt` • `signals(2).values: phi theta psi` Signals are linearly interpolated vs. time using `interp1`.
`'Array6DoF'`	A double-precision array in `n` rows and 7 columns for 6-DoF data: `time lat lon alt phi theta psi`. If a double-precision array of 8 or more columns is in `'TimeseriesSource'`, the first 7 columns are used as 6-DoF data.
`'Array3DoF'`	A double-precision array in `n` rows and 4 columns for 3-DoF data: `time lat alt theta`. If a double-precision array of 5 or more columns is in `'TimeseriesSource'`, the first 4 columns are used as 3-DoF data.
`'Custom'`	Position and angle data is retrieved from `'TimeseriesSource'` by the currently registered `'TimeseriesReadFcn'`.

Introduced in R2007b

Aero.Viewpoint

Create viewpoint object for use in virtual reality animation

Syntax

```
h = Aero.Viewpoint
```

Description

`h = Aero.Viewpoint` creates a viewpoint object for use with virtual reality animation.

Constructor Summary

Constructor	Description
`Viewpoint`	Create node object for use with virtual reality animation.

Property Summary

Property	Description	Values
`Name`	Specify name of the node object.	Character vector | string
`Node`	Specify node object that contains the viewpoint node.	MATLAB array

Introduced in R2007b

Aero.VirtualRealityAnimation

Construct virtual reality animation object

Syntax

`h = Aero.VirtualRealityAnimation`

Description

`h = Aero.VirtualRealityAnimation` constructs a virtual reality animation object. The animation object is returned to `h`. The animation object has the following methods and properties.

Limitations

The `Aero.VirtualRealityAnimation` object is not available for Aerospace Toolbox Online.

Constructor Summary

Constructor	Description
`VirtualReality-Animation`	Construct virtual reality animation object.

Method Summary

Method	Description
`addNode`	Add existing node to current virtual reality world.
`addRoute`	Add VRML ROUTE statement to virtual reality animation.

Method	Description
`addViewpoint`	Add viewpoint for virtual reality animation.
`delete`	Destroy virtual reality animation object.
`initialize`	Create and populate virtual reality animation object.
`nodeInfo`	Create list of nodes associated with virtual reality animation object.
`play`	Animate virtual reality world for given position and angle in time series data.
`removeNode`	Remove node from virtual reality animation object.
`removeViewpoint`	Remove viewpoint node from virtual reality animation.
`saveas`	Save virtual reality world associated with virtual reality animation object.
`updateNodes`	Set new translation and rotation of moveable items in virtual reality animation.

Notes on Aero.VirtualRealityAnimation Methods

Aero.VirtualRealityAnimation methods that change the current virtual reality world use a temporary `.wrl` file to manage those changes. These methods include:

- `addNode`
- `removeNode`
- `addViewpoint`
- `removeViewpoint`
- `addRoute`

Be aware of the following behavior:

- After the methods make the changes, they reinitialize the world, using the information stored in the temporary `.wrl` file.
- When you delete the virtual reality animation object, this action deletes the temporary file.
- Use the `saveas` method to save the temporary `.wrl` file.
- These methods do not affect user-created `.wrl` files.

Property Summary

Property	Description	Values
`Name`	Specify name of the animation object.	Character vector | string
`VRWorld`	Returns the vrworld object associated with the animation object.	MATLAB array
`VRWorldFilename`	Specify the `.wrl` file for the vrworld.	Character vector | string
`VRWorldOldFilename`	Specify the old `.wrl` files for the vrworld.	MATLAB array
`VRWorldTempFilename`	Specify the temporary `.wrl` file for the animation object.	Character vector | string
`VRFigure`	Returns the vrfigure object associated with the animation object.	MATLAB array
`Nodes`	Specify the nodes that the animation object contains.	MATLAB array
`Viewpoints`	Specify the viewpoints that the animation object contains.	MATLAB array
`TimeScaling`	Specify the time scaling, in seconds.	double
`TStart`	Specify the recording start time, in seconds.	double
`TFinal`	Specify end time, in seconds.	double
`TCurrent`	Specify current time, in seconds.	double
`FramesPerSecond`	Specify rate, in frames per second.	double
`ShowSaveWarning`	Specify save warning display setting.	double • `0` — No warning is displayed. • Non-zero — Warning is displayed.

Property	Description	Values
`VideoFileName`	Specify video recording file name.	Character vector | string
`VideoCompression`	Specify video recording compression file type. For more information on video compression, see `VideoWriter`.	• `'Archival'` Create Motion JPEG 2000 format file with lossless compression. • `'Motion JPEG AVI'` Create compressed AVI format file using Motion JPEG codec. • `'Motion JPEG 2000'` Create compressed Motion JPEG 2000 format file. • `'MPEG-4'` Create compressed MPEG-4 format file with H.264 encoding (Windows 7 systems only). • `'Uncompressed AVI'` Create uncompressed AVI format file with RGB24 video. `Aero.VideoProfileTypeEnum` **Default:** `'Archival'`
`VideoQuality`	Specify video recording quality. For more information on video quality, see the `Quality` property of `VideoWriter`.	Value between `0` and `100`. double **Default:** `75`

Property	Description	Values
`VideoRecord`	Enable video recording.	• `'on'` Enable video recording. • `'off'` Disable video recording. • `'scheduled'` Schedule video recording. Use this property with the `VideoTStart` and `VideoTFinal` properties. **Default:** `'off'`
`VideoTStart`	Specify video recording start time for scheduled recording.	Value between `TStart` and `TFinal`. double **Default:** `NaN`, which uses the value of `TStart`.
`VideoTFinal`	Specify video recording stop time for scheduled recording.	Value between `TStart` and `TFinal`. double **Default:** `NaN`, which uses the value of `TFinal`.

Examples

Record Virtual Reality Animation Object Simulation

This example shows how to record virtual reality animation of an object simulation.

- Record the simulation of a virtual reality animation object
- Simulate and record flight data

- Create an animation object

```
h = Aero.VirtualRealityAnimation;
% Control the frame display rate.

h.FramesPerSecond = 10;

% Configure the animation object to set the seconds of animation data per
% second time scaling (TimeScaling) property.

h.TimeScaling = 5;

% The combination of FramesPerSecond and TimeScaling property determine the
% time step of the simulation. These settings result in a time step of
% approximately 0.5 s.
% This code sets the .wrl file to use in the virtual reality animation.

h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];

% Copy the .wrl file to a temporary directory and set the world file name
% to the copied .wrl file.

copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];

% Load the animation world described in the 'VRWorldFilename' field of the
% animation object.

h.initialize();

% Set simulation timeseries data. takeoffData.mat contains logged simulated
% data. takeoffData is set up as a 'StructureWithTime', which is one of the
% default data formats.

load takeoffData
[~, idxPlane] = find(strcmp('Plane', h.nodeInfo));
h.Nodes{idxPlane}.TimeseriesSource = takeoffData;
h.Nodes{idxPlane}.TimeseriesSourceType = 'StructureWithTime';

% Use the example custom function vranimCustomTransform to correctly line
% up the position and rotation data with the surrounding objects in the
% virtual world. This code sets the coordinate transformation function for
% the virtual reality animation.

h.Nodes{idxPlane}.CoordTransformFcn = @vranimCustomTransform;

% Set up recording properties.

h.VideoRecord = 'on';
h.VideoQuality = 50;
h.VideoCompression = 'Motion JPEG AVI'
h.VideoFilename = 'astMotion_JPEG_VR';

% Play the animation.
```

```
h.play();

% Verify that a file named astMotion_JPEG_VR.avi was created in the current folder.
% Disable recording to preserve the file.

h.VideoRecord = 'off';
```

Record Virtual Reality Animation for Four Seconds

This example shows how to simulate flight data for four seconds.

```
% Create an animation object.
h = Aero.VirtualRealityAnimation;

% Control the frame display rate.
h.FramesPerSecond = 10;

% Configure the animation object to set the seconds of animation data per
% second time scaling (TimeScaling) property.
h.TimeScaling = 5;

% The combination of FramesPerSecond and TimeScaling properties determines
% the time step of the simulation. These settings result in a time step of
% approximately 0.5 s.
% This code sets the .wrl file to use in the virtual reality animation.
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];

% Copy the .wrl file to a temporary directory and set the world file name
% to the copied .wrl file.
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];

% Load the animation world described in the 'VRWorldFilename' field of the
% animation object.
h.initialize();

% Set simulation timeseries data. takeoffData.mat contains logged simulated
% data. takeoffData is set up as a 'StructureWithTime', which is one of the
% default data formats.
load takeoffData
[~, idxPlane] = find(strcmp('Plane', h.nodeInfo));
h.Nodes{idxPlane}.TimeseriesSource = takeoffData;
h.Nodes{idxPlane}.TimeseriesSourceType = 'StructureWithTime';

% Use the example custom function vranimCustomTransform to correctly line
% up the position and rotation data with the surrounding objects in the
% virtual world. This code sets the coordinate transformation function for
% the virtual reality animation.
h.Nodes{idxPlane}.CoordTransformFcn = @vranimCustomTransform;

% Set up recording properties.
h.VideoRecord = 'on';
h.VideoQuality = 50;
h.VideoCompression = 'Motion JPEG AVI'
h.VideoFilename = 'astMotion_JPEG';
```

```
% Play the animation from TFinal to TStart.
h.TSTart = 1;
h.TFinal = 5;
h.play();

% Verify that a file named astMotion_JPEG_VR.avi was created in the
% current folder. When you rerun the recording, notice that the play time
% is faster than when you record for the length of the simulation time.

% Disable recording to preserve the file.
h.VideoRecord = 'off';
```

Schedule Three Second Recording of Virtual Reality Object Simulation

This example shows how to schedule a three second recording a virtual reality object animation simulation.

```
% Create an animation object.
h = Aero.VirtualRealityAnimation;

% Control the frame display rate.
h.FramesPerSecond = 10;

% Configure the animation object to set the seconds of animation data per
% second time scaling (TimeScaling) property.
h.TimeScaling = 5;

% The combination of FramesPerSecond and TimeScaling properties determines
% the time step of the simulation. These settings result in a time step of
% approximately 0.5 s.
% This code sets the .wrl file to use in the virtual reality animation.
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];

% Copy the .wrl file to a temporary directory and set the world file name
% to the copied .wrl file.
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];

% Load the animation world described in the 'VRWorldFilename' field of the
% animation object.
h.initialize();

% Set simulation timeseries data. takeoffData.mat contains logged
% simulated data. takeoffData is set up as a 'StructureWithTime', which is
% one of the default data formats.
load takeoffData
[~, idxPlane] = find(strcmp('Plane', h.nodeInfo));
h.Nodes{idxPlane}.TimeseriesSource = takeoffData;
h.Nodes{idxPlane}.TimeseriesSourceType = 'StructureWithTime';

% Use the example custom function vranimCustomTransform to correctly line
% up the position and rotation data with the surrounding objects in the
% virtual world. This code sets the coordinate transformation function for
% the virtual reality animation.
h.Nodes{idxPlane}.CoordTransformFcn = @vranimCustomTransform;

% Set up recording properties.
h.VideoQuality = 50;
h.VideoCompression = 'Motion JPEG AVI';
h.VideoFilename = 'astMotion_JPEG';

% Set up simulation time from TFinal to TStart.
h.TSTart = 1;
h.TFinal = 5;

% Set up to record between two and four seconds of the four second
% simulation.
h.VideoRecord='scheduled';
h.VideoTSTart = 2;
h.VideoTFinal = 4;

% Play the animation.
h.play();

% Verify that a file named astMotion_JPEG_VR.avi was created in the
% current folder. When you rerun the recording, notice that the play time
% is faster than when you record for the length of the simulation time.
% Disable recording to preserve the file.
h.VideoRecord = 'off';
```

Introduced in R2007b

airspeed

Airspeed from velocity

Syntax

`airspeed = airspeed(`*`velocities`*`)`

Description

`airspeed = airspeed(`*`velocities`*`)` computes `m` airspeeds, `airspeed`, from an *m*-by-3 array of velocities, *`velocities`*.

Examples

Determine the airspeed for velocity one array:

```
as = airspeed([84.3905  33.7562  10.1269])

as =

   91.4538
```

Determine the airspeed for velocity for multiple arrays:

```
as = airspeed([50 20 6; 5 0.5 2])

as =

   54.1849
    5.4083
```

See Also

`alphabeta` | `correctairspeed` | `dpressure` | `machnumber`

Introduced in R2006b

AirspeedIndicator Properties

Control airspeed indicator appearance and behavior

Description

Airspeed indicators are components that represent an airspeed indicator. Properties control the appearance and behavior of an airspeed indicator. Use dot notation to refer to a particular object and property:

```
f = uifigure;
airspeed = uiaeroairspeed(f);
airspeed.Airspeed = 100;
```

By default, minor ticks represent 10-knot increments and major ticks represent 40-knot increments. The parameters **Minimum** and **Maximum** determine the minimum and maximum values on the gauge. The number and distribution of ticks is fixed, which means that the first and last tick display the minimum and maximum values. The ticks in between distribute evenly between the minimum and maximum values. For major ticks, the distribution of ticks is (**Maximum**-**Minimum**)/9. For minor ticks, the distribution of ticks is (**Maximum**-**Minimum**)/36.

The airspeed indicator has scale color bars that allow for overlapping for the first bar, displayed at a different radius. This different radius lets the block represent maximum speed with flap extended (V_{FE}) and stall speed with flap extended (V_{SO}) accurately for aircraft airspeed and stall speed.

Properties

Airspeed Indicator

`Airspeed` — Airspeed

`0` (default) | finite, real, and scalar numeric

Airspeed value, specified as a finite, real, and scalar numeric, in knots. The airspeed value determines the airspeed of the aircraft.

- If the value is less than the minimum `Limits` property value, then the needle points to a location immediately before the beginning of the scale.
- If the value is more than the maximum `Limits` property value, then the needle points to a location immediately after the end of the scale.

Example: 100

`Limits` — Minimum and maximum airspeed indicator scale values
[40 400] (default) | two-element finite and real numeric array

Minimum and maximum gauge scale values, specified as a two-element numeric array. The first value in the array must be less than the second value, in knots.

If you change `Limits` such that the `Value` property is less than the new lower limit, or more than the new upper limit, then the gauge needle points to a location off the scale.

For example, suppose `Limits` is `[0 100]` and the `Value` property is 20. If the `Limits` changes to `[50 100]`, then the needle points to a location off the scale, slightly less than 50.

`ScaleColors` — Scale colors
[ ] (default) | n-by-3 array of RGB triplets | cell array

Scale colors, specified as one of the following arrays:

- An n-by-3 array of RGB triplets
- A cell array containing RGB triplets, any of the color options listed in the table below, or a combination of both.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`

Option	Description	Equivalent RGB Triplet
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]` (not recommended)

Each color of the `ScaleColors` array corresponds to a colored section of the gauge. Set the `ScaleColorLimits` property to map the colors to specific sections of the gauge.

If you do not set the `ScaleColorLimits` property, MATLAB distributes the colors equally over the range of the gauge.

`ScaleColorLimits` — Scale color limits
[ ] (default) | n-by-2 array

Scale color limits, specified as an n-by-2 array of numeric values. For every row in the array, the first element must be less than the second element. The first `ScaleColorLimits` value can overlap (see Display Flight Trajectory Data Using Flight Instruments and Flight Animation).

When applying colors to the gauge, MATLAB applies the colors starting with the first color in the `ScaleColors` array. Therefore, if two rows in `ScaleColorLimits` array overlap, then the color applied later takes precedence.

The gauge does not display any portion of the `ScaleColorLimits` that falls outside of the `Limits` property.

If the `ScaleColors` and `ScaleColorLimits` property values are different sizes, then the gauge shows only the colors that have matching limits. For example, if the `ScaleColors` array has three colors, but the `ScaleColorLimits` has only two rows, then the gauge displays the first two color/limit pairs only.

Value — Airspeed
`0` (default) | finite, real, and scalar numeric

Airspeed value, specified as a finite, real, and scalar numeric. The airspeed value determines the airspeed of the aircraft.

- If the value is less than the minimum `Limits` property value, then the needle points to a location immediately before the beginning of the scale.
- If the value is more than the maximum `Limits` property value, then the needle points to a location immediately after the end of the scale.

Example: 100

Interactivity

`Visible` — Visibility of airspeed indicator

`'on'` (default) | `'off'`

Visibility of the airspeed indicator, specified as `'on'` or `'off'`. The `Visible` property determines whether the airspeed indicator is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire airspeed indicator is hidden, but you can still specify and access its properties.

`Enable` — Operational state of airspeed indicator

`'on'` (default) | `'off'`

Operational state of airspeed indicator, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the indicator indicates that the indicator is operational.
- If you set this property to `'off'`, then the appearance of the indicator appears dimmed, indicating that the indicator is not operational.

Position

`Position` — Location and size of airspeed indicator

`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the airspeed indicator relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the airspeed indicator

Element	Description
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the airspeed indicator
`width`	Distance between the right and left outer edges of the airspeed indicator
`height`	Distance between the top and bottom outer edges of the airspeed indicator

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[200 120 120 120]`

`InnerPosition` — Inner location and size of airspeed indicator
`[100 100 120 120]` (default) | `[left bottom width height]`

Inner location and size of the airspeed indicator, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of airspeed indicator
`[100 100 120 120]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the airspeed indicator returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an airspeed indicator in the third row and second column of its parent grid.

```
g = ui([4 3]);
gauge = uiaeroairspeed(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the airspeed indicator span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this airspeed indicator spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.

HandleVisibility Value	Description
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Identifiers

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

Type — Type of graphics object
`'uiaeroairspeed'`

This property is read-only.

Type of graphics object, returned as `'uiaeroairspeed'`.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also

`uiaeroairspeed`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

alphabeta

Incidence and sideslip angles

Syntax

`[incidence sideslip] = alphabeta(velocities)`

Description

`[incidence sideslip] = alphabeta(velocities)` computes *m* incidence and sideslip angles, *incidence* and *sideslip* , between the velocity vector and the body. *velocities* is an `m`-by-3 array of velocities in body axes. *incidence* and *sideslip* are in radians.

Examples

Determine the incidence and sideslip angles for velocity for one array:

```
[alpha, beta] = alphabeta([84.3905  33.7562  10.1269])

alpha =

    0.1194


beta =

    0.3780
```

Determine the incidence and sideslip angles for velocity for two arrays:

```
[alpha, beta] = alphabeta([50 20 6; 5 0.5 2])

alpha =

    0.1194
    0.3805

beta =

    0.3780
    0.0926
```

See Also

`airspeed` | `machnumber`

Introduced in R2006b

Altimeter Properties

Control altimeter appearance and behavior

Description

Altimeters are components that represent an altimeter. Properties control the appearance and behavior of an altimeter. Use dot notation to refer to a particular object and property:

```
f = uifigure;
altimeter = uiaeroaltimeter(f);
altimeter.Altitude = 100;
```

The altimeter displays the altitude above sea level in feet, also known as the pressure altitude. It displays the altitude value with needles on a gauge and a numeric indicator.

- The gauge has 10 major ticks. Within each major tick are five minor ticks. This gauge has three needles. Using the needles, the altimeter can display accurately only altitudes between 0 and 100,000 feet.
  - For the longest needle, an increment of a small tick represents 20 feet and a major tick represents 100 feet.
  - For the second longest needle, a minor tick represents 200 feet and a major tick represents 1,000 feet.
  - For the shortest needle a minor tick represents 2,000 feet and a major tick represents 10,000 feet.
- For the numeric display, the gauge shows values as numeric characters between 0 and 9,999 feet. When the numeric display value reaches 10,000 feet, the gauge displays the value as the remaining values below 10,000 feet. For example, 12,345 feet displays as 2,345 feet. When a value is less than 0 (below sea level), the gauge displays `0`. The needles show the appropriate value except for when the value is below sea level or over 100000 feet. Below sea level, the needles set to `0`, over 100,000, the needles stay set at 100,000.

Properties

Altimeter

`Altitude` — Altitude of aircraft
`0` (default) | finite, real, and scalar numeric

Altitude of the aircraft, specified as any finite and scalar numeric, in feet.

Example: `60`

Dependencies

Specifying this value changes the value of `Value`.

Data Types: `double`

`Value` — Location of aircraft heading
`0` (default) | finite, real, and scalar numeric

Location of the aircraft altitude, specified as a finite and scalar numeric, in feet.

- Changing the value changes the direction of the heading in 5-degree increments.

Example: `60`

Dependencies

Specifying this value changes the value of the `Altitude` value.

Data Types: `double`

Interactivity

`Visible` — Visibility of altimeter
`'on'` (default) | `'off'`

Visibility of the altimeter, specified as `'on'` or `'off'`. The `Visible` property determines whether the altimeter is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire altimeter is hidden, but you can still specify and access its properties.

`Enable` — Operational state of altimeter
`'on'` (default) | `'off'`

Operational state of altimeter, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the altimeter indicates that the altimeter is operational.
- If you set this property to `'off'`, then the appearance of the altimeter appears dimmed, indicating that the altimeter is not operational.

Position

`Position` — Location and size of altimeter

`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the altimeter relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the altimeter
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the altimeter
`width`	Distance between the right and left outer edges of the altimeter
`height`	Distance between the top and bottom outer edges of the altimeter

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[200 120 120 120]`

`InnerPosition` — Inner location and size of altimeter

`[100 100 120 120]` (default) | `[left bottom width height]`

Inner location and size of the altimeter, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of altimeter
`[100 100 120 120]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the altimeter returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an altimeter in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
gauge = uiaeroaltimeter(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the altimeter span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this altimeter spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function
`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.

- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function
`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.

- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Identifiers

Type — Type of graphics object

`'uiaeroaltimeter'`

This property is read-only.

Type of graphics object, returned as `'uiaeroaltimeter'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also

`uiaeroaltimeter`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

angle2dcm

Convert rotation angles to direction cosine matrix

Syntax

```
dcm = angle2dcm(rotationAng1,rotationAng2,rotationAng3)
dcm = angle2dcm( ___ ,rotationSequence)
```

Description

`dcm = angle2dcm(rotationAng1,rotationAng2,rotationAng3)` calculates the direction cosine matrix given three sets of rotation angles specifying yaw, pitch, and roll.

`dcm = angle2dcm( ___ ,rotationSequence)` calculates the direction cosine matrix given three sets of rotation angles.

Examples

Direction Cosine Matrix from Angles

Calculate direction cosine matrix from rotation angles.

```
yaw = 0.7854;
pitch = 0.1;
roll = 0;
dcm = angle2dcm( yaw, pitch, roll )
```

```
dcm = 3×3

    0.7036    0.7036   -0.0998
   -0.7071    0.7071         0
    0.0706    0.0706    0.9950
```

Direction Cosine Matrix from Rotation Angle and Sequence

Calculate direction cosine matrix from rotation angles and rotation sequence.

```
yaw = [0.7854 0.5];
pitch = [0.1 0.3];
roll = [0 0.1];
dcm = angle2dcm( pitch, roll, yaw, 'YXZ' )
```

```
dcm =
dcm(:,:,1) =

    0.7036    0.7071   -0.0706
   -0.7036    0.7071    0.0706
    0.0998         0    0.9950


dcm(:,:,2) =

    0.8525    0.4770   -0.2136
   -0.4321    0.8732    0.2254
    0.2940   -0.0998    0.9506
```

Input Arguments

`rotationAng1` — First rotation angles

m-by-1 array

First rotation angles, specified as an *m*-by-1 array, in rads.

Data Types: `double` | `single`

`rotationAng2` — Second rotation angles

m-by-1 array

Second rotation angles, specified as an *m*-by-1 array, in rads.

Data Types: `double` | `single`

`rotationAng3` — Third rotation angles
m-by-1 array

Third rotation angles, specified as an *m*-by-1 array, in rads.

Data Types: `double` | `single`

`rotationSequence` — Rotation sequence
'*ZYX*' (default) | '*ZYX*' | '*ZYZ*' | '*ZXY*' | '*ZXZ*' | '*YXZ*' | '*YXY*' | '*YZX*' | '*YZY*' | '*XYZ*' | '*XZY*' | '*XYX*' | '*XZX*'

Rotation sequence, specified as a scalar.

Data Types: `char` | `string`

Output Arguments

`dcm` — Direction cosine matrices
3-by-3-by-*m* matrix

Direction cosine matrices, specified as a 3-by-3-by-*m* matrix, where *m* is the number of direction cosine matrices.

See Also

`angle2quat` | `dcm2angle` | `dcm2quat` | `quat2angle` | `quat2dcm`

Introduced in R2006b

angle2quat

Convert rotation angles to quaternion

Syntax

`quaternion` `= angle2quat(`*`rotationAng1`*`,`*`rotationAng2`*`,`*`rotationAng3`*`)`
`quaternion` `=`
`angle2quat(`*`rotationAng1`*`,`*`rotationAng2`*`,`*`rotationAng3`*`,`*`rotationSequence`*`)`

Description

`quaternion` `= angle2quat(`*`rotationAng1`*`,`*`rotationAng2`*`,`*`rotationAng3`*`)` calculates the quaternion for three rotation angles.

`quaternion` `=`
`angle2quat(`*`rotationAng1`*`,`*`rotationAng2`*`,`*`rotationAng3`*`,`*`rotationSequence`*`)` calculates the quaternion using a rotation sequence.

Input Arguments

`rotationAng1`

m-by-1 array of first rotation angles, in radians.

`rotationAng2`

m-by-1 array of second rotation angles, in radians.

`rotationAng3`

m-by-1 array of third rotation angles, in radians.

`rotationSequence`

Rotation sequence. For example, the default `'ZYX'` represents a sequence where *`rotationAng1`* is *z*-axis rotation, *`rotationAng2`* is *y*-axis rotation, and *`rotationAng3`* is *x*-axis rotation.
`'ZYX'`
`'ZYZ'`
`'ZXY'`
`'ZXZ'`
`'YXZ'`
`'YXY'`
`'YZX'`
`'YZY'`
`'XYZ'`
`'XZY'`
`'XYX'`
`'XZX'`
`'ZYX'` (default)

Output Arguments

`quaternion`

`m`-by-4 matrix containing *`m`* quaternions. *`quaternion`* has its scalar number as the first column.

Examples

Determine the quaternion from rotation angles:

```
yaw = 0.7854;
pitch = 0.1;
roll = 0;
q = angle2quat(yaw, pitch, roll)
q =
    0.9227   -0.0191    0.0462    0.3822
```

Determine the quaternion from rotation angles and rotation sequence:

```
yaw = [0.7854 0.5];
pitch = [0.1 0.3];
roll = [0 0.1];
q = angle2quat(pitch, roll, yaw, 'YXZ')
q =
    0.9227    0.0191    0.0462    0.3822
    0.9587    0.0848    0.1324    0.2371
```

See Also

`angle2dcm` | `dcm2angle` | `dcm2quat` | `quat2angle` | `quat2dcm`

Introduced in R2007b

angle2rod

Convert rotation angles to Euler-Rodrigues vector

Syntax

```
rod=angle2rod(R1,R2,R3)
rod=angle2rod(R1,R2,R3,S)
```

Description

`rod=angle2rod(R1,R2,R3)` function converts the rotation described by the three rotation angles, `R1`, `R2`, and `R3`, into an *M*-by-3 Euler-Rodrigues matrix, `rod`.

`rod=angle2rod(R1,R2,R3,S)` function converts the rotation described by the three rotation angles and a rotation sequence, S, into an *M*-by-3 Euler-Rodrigues array, `rod`, that contains the *M* Rodrigues vector.

Examples

Determine the Rodrigues Vector from One Rotation Angle

Determine the Rodrigues vector from rotation angles.

```
yaw = 0.7854;
pitch = 0.1;
roll = 0;
r = angle2rod(yaw,pitch,roll)
```

```
r =

   -0.0207    0.0500    0.4142
```

Determine Rodrigues Vectors from Multiple Rotation Angles

Determine the Rodrigues vectors from multiple rotation angles.

```
yaw = [0.7854 0.5];
pitch = [0.1 0.3];
roll = [0 0.1];
r = angle2rod(pitch,roll,yaw,'YXZ')
```

```
r =

    0.0207    0.0500    0.4142
    0.0885    0.1381    0.2473
```

Input Arguments

R1 — First rotation angle
M-by-1 array

First rotation angle, in radians, from which to determine Euler-Rodrigues vector. Values must be real.

Data Types: `double` | `single`

R2 — Second rotation angle
M-by-1 array

Second rotation angle, in radians, from which to determine Euler-Rodrigues vector. Values must be real.

Data Types: `double` | `single`

R3 — Third rotation angle
M-by-1 array

Third rotation angle, in radians, from which to determine Euler-Rodrigues vector. Values must be real.

Data Types: `double` | `single`

S — Rotation sequence
ZYX (default) | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Rotation sequence. For the default rotation sequence, ZYX, the rotation angle order is:

- R1 — *z*-axis rotation
- R2 — *y*-axis rotation
- R3 — *x*-axis rotation

Data Types: `char` | `string`

Output Arguments

`rod` — Euler-Rodrigues vector
3-element vector

Euler-Rodrigues vector determined from rotation angles.

Algorithms

An Euler-Rodrigues vector $\overrightarrow{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\overrightarrow{b} = [b_x \; b_y \; b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\overrightarrow{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

See Also

`dcm2rod` | `quat2rod` | `rod2angle` | `rod2dcm` | `rod2quat`

Introduced in R2017a

ArtificialHorizon Properties

Control artificial horizon appearance and behavior

Description

Artificial horizons are components that represent an artificial horizon. Properties control the appearance and behavior of an artificial horizon. Use dot notation to refer to a particular object and property:

```
f = uifigure;
artificialhorizon = uiaerohorizon(f);
artificialhorizon.Value = [100 20];
```

The artificial horizon represents aircraft attitude relative to horizon and displays roll and pitch in degrees:

- Values for roll cannot exceed +/– 90 degrees.
- Values for pitch cannot exceed +/– 30 degrees.

If the values exceed the maximum values, the gauge maximum and minimum values do not change.

Changes in roll value affect the gauge semicircles and the ticks located on the black arc turn accordingly. Changes in pitch value affect the scales and the distribution of the semicircles.

Properties

Artificial Horizon

`Pitch` — Pitch

`0` (default) | finite, real, and scalar numeric

Pitch value, specified as any finite and scalar numeric. The pitch value determines the movement of the aircraft around the transverse axis, in degrees.

Example: 10

Dependencies

Specifying this value changes the second element of the `Value` vector. Conversely, changing the second element of the `Value` vector changes the `Pitch` value.

Data Types: `double`

`Roll` — Roll

`0` (default) | finite, real, and scalar numeric

Roll value, specified as any finite and scalar numeric. The roll value determines the rotation of the aircraft around the longitudinal axis, in degrees.

Example: 10

Dependencies

Specifying this value changes the first element of the `Value` vector. Conversely, changing the first element of the `Value` vector changes the `Roll` value.

Data Types: `double`

`Value` — Roll and pitch

`[0 0]` (default) | two-element vector of finite, real, and scalar numerics

Roll and pitch values, specified as a vector ([`Roll Pitch`]).

- The roll value determines the rotation of the aircraft around the longitudinal axis.
- The pitch value determines the movement of the aircraft around the transverse axis.

Example: [100 -200]

Dependencies

- Specifying the `Roll` value changes the first element of the `Value` vector. Conversely, changing the first element of the `Value` vector changes the `Roll` value.
- Specifying the `Pitch` value changes the second element of the `Value` vector. Conversely, changing the second element of the `Value` vector changes the `Pitch` value.

Data Types: `double`

Interactivity

`Visible` — Visibility of artificial horizon
`'on'` (default) | `'off'`

Visibility of the artificial horizon, specified as `'on'` or `'off'`. The `Visible` property determines whether the artificial horizon, is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire artificial horizon is hidden, but you can still specify and access its properties.

`Enable` — Operational state of artificial horizon
`'on'` (default) | `'off'`

Operational state of artificial horizon, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the artificial horizon indicates that the artificial horizon is operational.
- If you set this property to `'off'`, then the appearance of the artificial horizon appears dimmed, indicating that the artificial horizon is not operational.

Position

`Position` — Location and size of artificial horizon
`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the artificial horizon relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the artificial horizon
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the artificial horizon
`width`	Distance between the right and left outer edges of the artificial horizon
`height`	Distance between the top and bottom outer edges of the artificial horizon

All measurements are in pixel units.

The Position values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: [200 120 120 120]

InnerPosition — Inner location and size of artificial horizon
[100 100 120 120] (default) | [left bottom width height]

Inner location and size of the artificial horizon, specified as [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

OuterPosition — Outer location and size of artificial horizon
[100 100 120 120]] (default) | [left bottom width height]

This property is read-only.

Outer location and size of the artificial horizon returned as [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

Layout — Layout options
empty LayoutOptions array (default) | GridLayoutOptions object

Layout options, specified as a GridLayoutOptions object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the Row and Column properties on the GridLayoutOptions object.

For example, this code places an artificial horizon in the third row and second column of its parent grid.

```
g = ui([4 3]);
gauge = uiaerohorizon(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the artificial horizon span multiple rows or columns, specify the Row or Column property as a two-element vector. For example, this artificial horizon spans columns 2 through 3:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

`Interruptible` — Callback interruption
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

Parent/Child

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Identifiers

Type — Type of graphics object
'uiaerohorizon'

This property is read-only.

Type of graphics object, returned as `'uiaerohorizon'`.

Tag — Object identifier
'' (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
[] (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also
`uiaerohorizon`

Topics
“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

atmoscoesa

Use 1976 COESA model

Syntax

`[T, a, P, Rho] = atmoscoesa(height, action)`

Description

Committee on Extension to the Standard Atmosphere has the acronym COESA. `[T, a, P, Rho] = atmoscoesa(height, action)` implements the mathematical representation of the 1976 COESA United States standard lower atmospheric values. These values are absolute temperature, pressure, density, and speed of sound for the input geopotential altitude.

Below the geopotential altitude of 0 m (0 feet) and above the geopotential altitude of 84,852 m (approximately 278,386 feet), the function extrapolates values. It extrapolates temperature values linearly and pressure values logarithmically.

Input Arguments

`height`

Array of *m*-by-1 geopotential heights, in meters.

`action`

Action for out-of-range input. Specify one:
`'Error'`
`'Warning'` (default)
`'None'`

Output Arguments

`T`

Array of *m*-by-1 temperatures, in kelvin.

`a`

Array of *m*-by-1 speeds of sound, in meters per second. The function calculates speed of sound using a perfect gas relationship.

`P`

Array of *m*-by-1 pressures, in pascal.

`Rho`

Array of *m*-by-1 densities, in kilograms per meter cubed. The function calculates density using a perfect gas relationship.

Examples

Calculate the COESA model at 1000 m with warnings for out-of-range inputs:

```
[T, a, P, rho] = atmoscoesa(1000)

T =

  281.6500

a =

  336.4341

P =

  8.9875e+004

rho =

    1.1116
```

Calculate the COESA model at 1000, 11,000, and 20,000 m with errors for out-of-range inputs:

```
[T, a, P, rho] = atmoscoesa([1000 11000 20000], 'Error')

T =

  281.6500  216.6500  216.6500

a =

  336.4341  295.0696  295.0696

P =

  1.0e+004 *

    8.9875    2.2632    0.5475

rho =

    1.1116    0.3639    0.0880
```

References

U.S. Standard Atmosphere, 1976, U.S. Government Printing Office, Washington, D.C.

See Also

`atmoscira` | `atmosisa` | `atmoslapse` | `atmosnonstd` | `atmospalt`

Introduced in R2006b

atmoscira

Use COSPAR International Reference Atmosphere 1986 model

Syntax

`[`*`T altitude zonalWind`*`] = atmoscira(`*`latitude`*`, `*`ctype`*`, `*`coord`*`, `*`mtype`*`, `*`month`*`, `*`action`*`)`

Description

`[`*`T altitude zonalWind`*`] = atmoscira(`*`latitude`*`, `*`ctype`*`, `*`coord`*`, `*`mtype`*`, `*`month`*`, `*`action`*`)` implements the mathematical representation of the Committee on Space Research (COSPAR) International Reference Atmosphere (CIRA) from 1986 model. The CIRA 1986 model provides a mean climatology. The mean climatology consists of temperature, zonal wind, and geopotential height or pressure. It encompasses nearly pole-to-pole coverage (80 degrees S to 80 degrees N) for 0 km to 120 km. This provision also encompasses the troposphere, middle atmosphere, and lower thermosphere. Use this mathematical representation as a function of pressure or geopotential height.

This function uses a corrected version of the CIRA data files provided by J. Barnett in July 1990 in ASCII format.

This function has the limitations of the CIRA 1986 model and limits the values for the CIRA 1986 model.

The CIRA 1986 model limits values to the regions of 80 degrees S to 80 degrees N on Earth. It also limits geopotential heights from 0 km to 120 km. In each monthly mean data set, the model omits values at 80 degrees S for 101,300 pascal or 0 m. It omits these values because these levels are within the Antarctic land mass. For zonal mean pressure in constant altitude coordinates, pressure data is not available below 20 km. Therefore, this value is the bottom level of the CIRA climatology.

Input Arguments

`latitude`

Array of *m* geopotential heights, in meters.

`ctype`

Representation of coordinate type. Specify:

`'Pressure'`	Pressure in pascal
`'GPHeight'`	Geopotential height in meters

`coord`

Depending on the value of `ctype`, this argument specifies one of the following arrays:

m	Pressures in pascal
m	Geopotential height in meters

`mtype`

Mean value type:

`'Monthly'` (default)	Monthly values.
`'Annual'`	Annual values. Valid when `ctype` has a value of `'Pressure'`.

`month`

Scalar value that selects the month in which the model takes the mean values. This argument applies only when *`mtype`* has a value of `'Monthly'`.

1 (default)	January
2	February
3	March
4	April
5	May
6	June

`7`	July
`8`	August
`9`	September
`10`	October
`11`	November
`12`	December

`action`

Action for out-of-range input. Specify one:
`'Error'`
`'Warning'` (default)
`'None'`

Output Arguments

`T`

Array of temperatures:

If *`m`* is `'Monthly'`	Array of *`m`* temperatures, in kelvin
If *`mtype`* is `'Annual'`	Array of *`m`*-by-7 values: • Annual mean temperature in kelvin • Annual temperature cycle amplitude in kelvin • Annual temperature cycle phase in month of maximum • Semiannual temperature cycle amplitude in kelvin • Semiannual temperature cycle phase in month of maximum • Terannual temperature cycle amplitude in kelvin • Terannual temperature cycle phase in month of maximum

`altitude`

If *`mtype`* is `'Monthly'`, an array of *`m`* geopotential heights or *`m`* air pressures:

If *`ctype`* is `'Pressure'`	Array *`m`* geopotential heights
If *`ctype`* is `'GPHeight'`	Array *`m`* air pressure

If *`mtype`* is `'Annual'`, an array of *`m`*-by-7 values for geopotential heights. The function defines this array only for the northern hemisphere (*`latitude`* is greater than `0`).

- Annual mean geopotential heights in meters
- Annual geopotential heights cycle amplitude in meters
- Annual geopotential heights cycle phase in month of maximum
- Semiannual geopotential heights cycle amplitude in meters
- Semiannual geopotential heights cycle phase in month of maximum
- Terannual geopotential heights cycle amplitude in meters
- Terannual geopotential heights cycle phase in month of maximum

`zonalWind`

Array of zonal winds:

If *`mtype`* is `'Monthly'`	Array in meters per second.

If *mtype* is `'Annual'`	Array of *m*-by-7 values: • Annual mean zonal winds in meters per second • Annual zonal winds cycle amplitude in meters per second • Annual zonal winds cycle phase in month of maximum • Semiannual zonal winds cycle amplitude in meters per second • Semiannual zonal winds cycle phase in month of maximum • Terannual zonal winds cycle amplitude in meters per second • Terannual zonal winds cycle phase in month of maximum

Examples

Using the CIRA 1986 model at 45 degrees latitude and 101,300 pascal for January with out-of-range actions generating warnings, calculate the mean monthly values. Calculate values for temperature (`T`), geopotential height (*alt*), and zonal wind (*zwind*).

```
[T, alt, zwind] = atmoscira( 45, 'Pressure', 101300 )
T =
  280.6000
alt =
   -18
zwind =
    3.3000
```

Using the CIRA 1986 model at 45 degrees latitude and 20,000 m for October with out-of-range actions generating warnings, calculate the mean monthly values. Calculate values for temperature (`T`), pressure (*pres*), and zonal wind (*zwind*).

```
[T, pres, zwind] = atmoscira( 45, 'GPHeight', 20000, 'Monthly', 10)
T =
  215.8500
pres =
  5.5227e+003
zwind =
    9.5000
```

Use the CIRA 1986 model at 45 and –30 degrees latitude and 20,000 m for October with out-of-range actions generating errors. Calculate values for temperature (*T*), pressure (*pres*), and zonal wind (*zwind*).

```
[T, pres, zwind] = atmoscira( [45 -30], 'GPHeight', 20000, 10, 'error')
T =
  215.8500  213.9000
pres =
  1.0e+003 *
    5.5227    5.6550
zwind =
    9.5000    4.3000
```

For September, with out-of-range actions generating warnings, use the CIRA 1986 model at 45 degrees latitude and –30 degrees latitude. Also use the model at 2000 pascal and 101,300 pascal. Calculate mean monthly values for temperature (*T*), geopotential height (*alt*), and zonal wind (*zwind*).

```
[T, alt, zwind] = atmoscira( [45 -30], 'Pressure', [2000 101300], 9)
T =
  223.5395  290.9000
alt =
  1.0e+004 *
    2.6692    0.0058
zwind =
    0.6300   -1.1000
```

Using the CIRA 1986 model at 45 degrees latitude and 2000 pascal with out-of-range actions generating warnings, calculate annual values. Calculate values for temperature (*T*), geopotential height (*alt*), and zonal wind (*zwind*).

```
[T, alt, zwind] = atmoscira( 45, 'Pressure', 2000, 'Annual' )
T =
  221.9596    5.0998    6.5300    1.9499    1.3000    1.0499    1.3000
alt =
  1.0e+004 *
    2.6465    0.0417    0.0007    0.0087    0.0001    0.0015    0.0002
zwind =
    4.6099   14.7496    0.6000    1.6499    4.6000    0.5300    1.4000
```

References

Fleming, E. L., Chandra, S., Shoeberl, M. R., Barnett, J. J., *Monthly Mean Global Climatology of Temperature, Wind, Geopotential Height and Pressure for 0-120 k*m, NASA TM100697, February 1988

`https://ccmc.gsfc.nasa.gov/modelweb/atmos/cospar1.html`

See Also

`atmoscoesa` | `atmosisa` | `atmoslapse` | `atmosnonstd` | `atmosnrlmsise00` | `atmospalt`

Introduced in R2007b

atmoshwm

Implement horizontal wind model

Syntax

```
wind = atmoshwm(latitude,longitude,altitude)
```

```
wind = atmoshwm(latitude,longitude,altitude,Name,Value)
```

Description

`wind = atmoshwm(latitude,longitude,altitude)` implements the U.S. Naval Research Laboratory Horizontal Wind Model (HWM™) routine to calculate the meridional and zonal components of the wind for one or more sets of geophysical data: `latitude`, `longitude`, and `altitude`.

`wind = atmoshwm(latitude,longitude,altitude,Name,Value)` uses additional options specified by one or more `Name,Value` pair arguments.

Examples

Calculate the Total Horizontal Wind Model

dCalculate the total horizontal wind model for a latitude of 45 degrees south, longitude of 85 degrees west, and altitude of 25,000 m above mean sea level (msl). The date is the 150th day of the year, at 11 am UTC, using an Ap index of 80. The horizontal model version is 14.

```
w = atmoshwm(-45,-85,25000,'day',150,'seconds',39600,'apindex',80,'model','total', 'version', '14')
```

```
w =

    3.2874   25.8735
```

Calculate the Quiet Horizontal Wind Model

Calculate the quiet horizontal wind model for a latitude of 50 degrees north, two altitudes of 100,000 m and 150,000 m above msl, and a longitude of 20 degrees west. The date is midnight UTC of January 30. The default horizontal model version is 14.

```
w = atmoshwm([50;50],[-20;-20],[100000;150000],'day',[30;30])
```

```
w =

  -42.9350  -40.3693
   29.1106    0.6253
```

Calculate a Disturbed Horizontal Wind Model

Calculate the disturbed horizontal wind model for an altitude of 150,000 m above msl at latitude 70 degrees north, longitude 65 degrees west. The date is midnight UTC of June 15. The default horizontal model version is 14.

```
dw = atmoshwm(70,-65,150000,'day',166,'model','disturbance')
```

```
dw =
    1.7954   -1.7130
```

Input Arguments

`latitude` — Geodetic latitude
scalar | *M*-by-1 array

Geodetic latitudes, in degrees, specified as a scalar or *M*-by-1 array, where *M* is one or more sets of geophysical data.

Example: -45

Data Types: `double`

`longitude` — Geodetic longitude
scalar | *M*-by-1 array

Geodetic longitudes, in degrees, specified as a scalar or *M*-by-1 array, where *M* is one or more sets of geophysical data.

Example: -85

Data Types: `double`

`altitude` — Geopotential height
scalar | *M*-by-1 array

Geopotential heights, in meters, within the range of 0 to 500 km, specified as a scalar or *M*-by-1 array. *M* is one or more sets of geophysical data.

Example: 25000

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'apindex',80,'model','total'` specifies that the total horizontal wind model be calculated for an Ap index of 80.

apindex — Ap index
M-by-1 array of zeroes (default) | scalar | *M*-by-1 array

Ap index for the Universal Coordinated Time (UTC) at which `atmoshwm` evaluates the model, specified as an *M*-by-1 array of zeroes, a scalar, or an *M*-by-1 array. *M* is one or more sets of geophysical data. Select the index from the NOAA National Geophysical Data Center, which contains three-hour interval geomagnetic disturbance index values. If the Ap index value is greater than zero, the model evaluation accounts for magnetic effects.

Specify the Ap index as a value from 0 through 400. Specify an Ap index value for only the disturbance or total wind model type.

Data Types: `double`

day — Day of year
M-by-1 array of ones (default) | scalar | *M*-by-1 array

Day of year in UTC. Specify the day as a value from 1 through 366 (for a leap year), specified as an *M*-by-1 array of zeroes, a scalar, or an *M*-by-1 array. *M* is one or more sets of geophysical data.

Data Types: `double`

`seconds` — Elapsed seconds
M-by-1 array of zeroes (default) | scalar | *M*-by-1 array

Elapsed seconds since midnight for the selected day, in UTC, specified as specified as an *M*-by-1 array of zeroes, a scalar, or an *M*-by-1 array. *M* is one or more sets of geophysical data.

Specify the seconds as a value from 0 through 86,400.

Data Types: `double`

`model` — Horizontal wind model type
`'quiet'` (default) | `'disturbance'` | `'total'`

Horizontal wind model type for which to calculate the wind components. This setting applies to all the sets of geophysical data in *M*.

- `'quiet'`

  Calculates the horizontal wind model without the magnetic disturbances. Quiet model types do not account for Ap index values. For this model type, do not specify an Ap index value when using this model type.
- `'disturbance'`

  Calculates the effect of only magnetic disturbances in the wind. For this model type, specify Ap index values greater than or equal to zero.
- `'total'`

  Calculates the combined effect of the quiet and magnetic disturbances. for this model type, specify Ap index values greater than or equal to zero.

Data Types: `double`

`action` — Function behavior
`'Error'` (default) | `'None'` | `'Warning'`

Function behavior when inputs are out of range, specified as one of these values. This type applies to all the sets of geophysical data in *M*.

Value	Description
`'None'`	No action.

Value	Description
`'Warning'`	Warning in the MATLAB Command Window, model simulation continues.
`'Error'`	MATLAB returns an exception, model simulation stops.

Data Types: `double`

`version` — Horizontal wind model version
`'14'` (default) | `'07'`

Implements specified horizontal wind model type.

- `'14'`

  Horizontal Wind Model 14.
- `'07'`

  Horizontal Wind Model 07.

Data Types: `double`

Output Arguments

`wind` — Meridional and zonal wind components
M-by-2 array

Meridional and zonal wind components of the horizontal wind model, returned as an *M*-by-2 array, in m/s.

See Also

`atmoscira` | `atmoscoesa` | `atmosnrlmsise00`

External Websites

NOAA National Geophysical Data Center
An empirical model of the Earth's horizontal wind fields: HWM07
An update to the Horizontal Wind Model (HWM): The quiet time thermosphere

Introduced in R2016b

atmosisa

Use International Standard Atmosphere model

Syntax

```
[T, a, P, rho] = atmosisa(height)
```

Description

`[T, a, P, rho] = atmosisa(height)` implements the mathematical representation of the International Standard Atmosphere values for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude.

This function assumes that temperature and pressure values are held constant for both:

- Below the geopotential altitude of 0 km
- Above the geopotential altitude of the tropopause (at 20 km)

Examples

Calculate International Standard Atmosphere at One Height

Calculate the International Standard Atmosphere at 1000 m.

```
[T, a, P, rho] = atmosisa(1000)
```

```
T = 281.6500

a = 336.4341

P = 8.9875e+04

rho = 1.1116
```

Calculate International Standard Atmosphere at Multiple Heights

Calculate the International Standard Atmosphere at 1000, 11,000, and 20,000 m.

```
[T, a, P, rho] = atmosisa([1000 11000 20000])
```

```
T = 1×3

  281.6500  216.6500  216.6500


a = 1×3

  336.4341  295.0696  295.0696


P = 1×3
10^4 ×

    8.9875    2.2632    0.5475


rho = 1×3

    1.1116    0.3639    0.0880
```

Input Arguments

`height` — Geopotential heights
m-by-1*m* array

Geopotential heights, specified as an *m*-by-1*m* array.

Data Types: `double`

Output Arguments

T — Temperatures
m-element array

Temperatures, returned as an *m*-element array, in kelvin.

a — Speeds of sound
m-element array

Speeds of sound, returned as an *m*-element array, in meters per second. The function calculates speed of sound using a perfect gas relationship.

P — Pressures
m-element array

Pressures, returned as an *m*-element array, in pascal.

rho — Densities
m-element array

Densities, returned as an *m*-element array, in kilograms per meter cubed. The function calculates density using a perfect gas relationship.

References

[1] *U.S. Standard Atmosphere, 1976*. U.S. Government Printing Office, Washington, D.C.

See Also

`atmoscira` | `atmoscoesa` | `atmoslapse` | `atmosnonstd` | `atmospalt`

Introduced in R2006b

atmoslapse

Use Lapse Rate Atmosphere model

Syntax

[*T*, *a*, *P*, *rho*] = atmoslapse(*height*, *g*, *heatRatio*, *characteristicGasConstant*, *lapseRate*, *heightTroposphere*, *heightTropopause*, *density0*, *pressure0*, *temperature0*)
[*T*, *a*, *P*, *rho*] = atmoslapse(*height*, *g*, *heatRatio*, *characteristicGasConstant*, *lapseRate*, *heightTroposphere*, *heightTropopause*, *density0*, *pressure0*, *temperature0*, *height0*)

Description

[*T*, *a*, *P*, *rho*] = atmoslapse(*height*, *g*, *heatRatio*, *characteristicGasConstant*, *lapseRate*, *heightTroposphere*, *heightTropopause*, *density0*, *pressure0*, *temperature0*) implements the mathematical representation of the lapse rate atmospheric equations for ambient temperature, pressure, density, and speed of sound for the input geopotential altitude. To customize this atmospheric model, specify the atmospheric properties in the function input.

The function holds temperature and pressure values below the geopotential altitude of 0 km and above the geopotential altitude of the tropopause. The function calculates the density and speed of sound using a perfect gas relationship.

[*T*, *a*, *P*, *rho*] = atmoslapse(*height*, *g*, *heatRatio*, *characteristicGasConstant*, *lapseRate*, *heightTroposphere*, *heightTropopause*, *density0*, *pressure0*, *temperature0*, *height0*) indicates that the values for ambient temperature, pressure, density, and speed of sound are for below mean sea level geopotential altitudes.

The function holds temperature and pressure values below the geopotential altitude of `height0` and above the geopotential altitude of the tropopause. The function calculates the density and speed of sound using a perfect gas relationship.

Input Arguments

`height`

Array of *m*-by-1 geopotential heights, in meters.

`g`

Scalar of acceleration due to gravity, in meters per second squared.

`heatRatio`

Scalar of specific heat ratio.

`characteristicGasConstant`

Scalar of characteristic gas constant, in joule per kilogram-kelvin.

`lapseRate`

Scalar of lapse rate, in kelvin per meter.

`heightTroposphere`

Scalar of height of troposphere, in meters.

`heightTropopause`

Scalar of height of tropopause, in meters.

`density0`

Scalar of air density at mean sea level, in kilograms per meter cubed.

`pressure0`

Scalar of static pressure at mean sea level, in pascal.

`temperature0`

Scalar of absolute temperature at mean sea level, in kelvin.

height0

Scalar of minimum sea level altitude, in meters.

Output Arguments

T

Array of *m*-by-1 temperatures, in kelvin.

a

Array of *m*-by-1 speeds of sound, in meters per second. The function calculates speed of sound using a perfect gas relationship.

P

Array of *m*-by-1 pressures, in pascal.

rho

Array of *m*-by-1 densities, in kilograms per meter cubed. The function calculates density using a perfect gas relationship.

Examples

Calculate the atmosphere at 1000 m with the International Standard Atmosphere input values:

```
[T, a, P, rho] = atmoslapse(1000, 9.80665, 1.4, 287.0531, 0.0065, ...
    11000, 20000, 1.225, 101325, 288.15 )

T =

  281.6500

a =

  336.4341

P =

  8.9875e+004
```

```
rho =

    1.1116
```

References

U.S. Standard Atmosphere, 1976, U.S. Government Printing Office, Washington, D.C.

See Also

`atmoscira` | `atmoscoesa` | `atmosisa` | `atmosnonstd` | `atmospalt`

Introduced in R2006b

atmosnonstd

Use climatic data from MIL-STD-210 or MIL-HDBK-310

Syntax

`[T, a, P, rho] = atmosnonstd(height, atmosphericType, extremeParameter, frequency, extremeAltitude, action, specification)`

Description

`[T, a, P, rho] = atmosnonstd(height, atmosphericType, extremeParameter, frequency, extremeAltitude, action, specification)` implements a portion of the climatic data of the MIL-STD-210C or MIL-HDBK-310 worldwide air environment to 80 km geometric (or approximately 262,000 feet geometric). This implementation provides absolute temperature, pressure, density, and speed of sound for the input geopotential altitude.

This function holds all values below the geometric altitude of 0 m (0 feet) and above the geometric altitude of 80,000 m (approximately 262,000 feet). The envelope atmospheric model has exceptions where values are held below the geometric altitude of 1 km (approximately 3281 feet). It also has exceptions above the geometric altitude of 30,000 m (approximately 98,425 feet). These exceptions are due to lack of data in MIL-STD-210 or MIL-HDBK-310 for these conditions.

In general, this function interpolates temperature values linearly and density values logarithmically. It calculates pressure and speed of sound using a perfect gas relationship. The envelope atmospheric model has exceptions where the extreme value is the only value provided as an output. In these cases, the function interpolates pressure logarithmically. These envelope atmospheric model exceptions apply to all cases of high and low pressure, high and low temperature, and high and low density. These exceptions exclude the extreme values and 1% frequency of occurrence. These exceptions are due to lack of data in MIL-STD-210 or MIL-HDBK-310 for these conditions.

A limitation is that MIL-STD-210 and MIL-HDBK-310 exclude from consideration climatic data for the region south of 60 degrees S latitude.

This function uses the metric version of data from the MIL-STD-210 or MIL-HDBK-310 specifications. A limitation is some inconsistent data between the metric and English data. Locations where these inconsistencies occur are within the envelope data for low density, low temperature, high temperature, low pressure, and high pressure. The most noticeable differences occur in the following values:

- For low density envelope data with 5% frequency, the density values in metric units are inconsistent at 4 km and 18 km. In addition, the density values in English units are inconsistent at 14 km.
- For low density envelope data with 10% frequency, the density values in metric units are inconsistent at 18 km. In addition, the density values in English units are inconsistent at 14 km.
- For low density envelope data with 20% frequency, the density values in English units are inconsistent at 14 km.
- For high-pressure envelope data with 10% frequency, the pressure values at 8 km are inconsistent.

Input Arguments

`height`

Array of *m*-by-1 geopotential heights, in meters.

`atmosphericType`

Representation of `'Profile'` or `'Envelope'` for the atmospheric data:

`'Profile'`	Is the realistic atmospheric profiles associated with extremes at specified altitudes. Use `'Profile'` for simulation of vehicles vertically traversing the atmosphere, or when you need the total influence of the atmosphere.
`'Envelope'`	Uses extreme atmospheric values at each altitude. Use `'Envelope'` for vehicles traversing the atmosphere horizontally, without much change in altitude.

`extremeParameter`

Atmospheric parameter that is the extreme value. Atmospheric parameters that you can specify are:

'High temperature'
'Low temperature'
'High density'
'Low density'
'High pressure', available only if *atmosphericType* is 'Envelope'
'Low pressure', available only if *atmosphericType* is 'Envelope'

frequency

Percent of time that extreme values would occur. When using *atmosphericType* of 'Envelope' and *frequency* of '5%', '10%', and '20%', only the *extreme** parameter that you specify (temperature, density, or pressure) has a valid output. All other parameter outputs are zero.
'Extreme values', available only if *atmosphericType* is 'Envelope'
'1%'
'5%', available only if *atmosphericType* is 'Envelope'
'10%
'20%', available only if *atmosphericType* is 'Envelope'

extremeAltitude

Scalar value, in kilometers, selecting geometric altitude at which the extreme values occur. *extremeAltitude* applies only when *atmosphericType* is 'Profile'.

5	16404 ft
10	32808 ft
20	65617 ft
30	98425 ft
40	131234 ft

action

Action for out-of-range input:
'Error'
'Warning' (default)
'None'

specification

Atmosphere model:

`'210c'`	MIL-STD-210C
`'310'`	MIL-HDBK-310 (default)

Output Arguments

`T`

Array of *m*-by-1 temperatures, in kelvin. This function interpolates temperature values linearly.

`a`

Array of *m*-by-1 speeds of sound, in meters per second. This function calculates speed of sound using a perfect gas relationship.

`P`

Array of *m*-by-1 pressures, in pascal. This function calculates pressure using a perfect gas relationship.

`rho`

Array of *m*-by-1 densities, in kilograms per meter cubed. This function interpolates density values logarithmically.

Examples

Calculate the nonstandard atmosphere profile. Use high density occurring 1% of the time at 5 km from MIL-HDBK-310 at 1000 m with warnings for out-of-range inputs:

```
[T, a, P, rho] = atmosnonstd( 1000,'Profile','High density','1%',5 )

T =

  248.1455

a =

  315.7900

P =
  8.9893e+004

rho =

    1.2620
```

Calculate the nonstandard atmosphere envelope with high pressure. Assume that high pressure occurs 20% of the time from MIL-STD-210C at 1000, 11,000, and 20,000 m with errors for out-of-range inputs:

```
[T, a, P, rho] = atmosnonstd([1000 11000 20000],'Envelope', ...
                              'High pressure','20%','Error','210c' )

T =

  0     0     0

a =

  0     0     0

P =

  1.0e+004 *
    9.1598    2.5309    0.6129

rho =

    0     0     0
```

References

Global Climatic Data for Developing Military Products (MIL-STD-210C), 9 January 1987, Department of Defense, Washington, D.C.

Global Climatic Data for Developing Military Products (MIL-HDBK-310), 23 June 1997, Department of Defense, Washington, D.C.

See Also

`atmoscira` | `atmoscoesa` | `atmosisa` | `atmoslapse` | `atmospalt`

Introduced in R2006b

atmosnrlmsise00

Implement mathematical representation of 2001 United States Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere

Syntax

```
[T rho] = atmosnrlmsise00(altitude, latitude, longitude, year,
dayOfYear, UTseconds)
[T rho] = atmosnrlmsise00(altitude, latitude, longitude, year,
dayOfYear, UTseconds, localApparentSolarTime)
[T rho] = atmosnrlmsise00(altitude, latitude, longitude, year,
dayOfYear, UTseconds, f107Average, f107Daily, magneticIndex)
[T rho] = atmosnrlmsise00(altitude, latitude, longitude, year,
dayOfYear, UTseconds, flags)
[T rho] = atmosnrlmsise00(altitude, latitude, longitude, year,
dayOfYear, UTseconds, otype)
[T rho] = atmosnrlmsise00(altitude, latitude, longitude, year,
dayOfYear, UTseconds, action)
```

Description

`[`*`T rho`*`] = atmosnrlmsise00(`*`altitude`*`, `*`latitude`*`, `*`longitude`*`, `*`year`*`, `*`dayOfYear`*`, `*`UTseconds`*`)` implements the mathematical representation of the 2001 United States Naval Research Laboratory Mass Spectrometer and Incoherent Scatter Radar Exosphere (NRLMSISE-00) of the MSIS® class model. NRLMSISE-00 calculates the neutral atmosphere empirical model from the surface to lower exosphere (0 m to 1,000,000 m). Optionally, it performs this calculation including contributions from anomalous oxygen that can affect satellite drag above 500,000 m.

`[`*`T rho`*`] = atmosnrlmsise00(`*`altitude`*`, `*`latitude`*`, `*`longitude`*`, `*`year`*`, `*`dayOfYear`*`, `*`UTseconds`*`, `*`localApparentSolarTime`*`)` specifies an array of `m` local apparent solar time (hours).

`[`*`T rho`*`] = atmosnrlmsise00(`*`altitude`*`, `*`latitude`*`, `*`longitude`*`, `*`year`*`, `*`dayOfYear`*`, `*`UTseconds`*`, `*`f107Average`*`, `*`f107Daily`*`, `*`magneticIndex`*`)` specifies arrays of *m* 81 day average of F10.7 flux (centered on `doy`), *m*-by-1 daily F10.7 flux for previous day, and `m`-by-7 of magnetic index information.

`[`*`T rho`*`] = atmosnrlmsise00(`*`altitude`*`, `*`latitude`*`, `*`longitude`*`, `*`year`*`, `*`dayOfYear`*`, `*`UTseconds`*`, `*`flags`*`)` specifies an array of 23 to enable or disable particular variations for the outputs.

`[`*`T rho`*`] = atmosnrlmsise00(`*`altitude`*`, `*`latitude`*`, `*`longitude`*`, `*`year`*`, `*`dayOfYear`*`, `*`UTseconds`*`, `*`otype`*`)` specifies a character vector or string for total mass density output.

`[`*`T rho`*`] = atmosnrlmsise00(`*`altitude`*`, `*`latitude`*`, `*`longitude`*`, `*`year`*`, `*`dayOfYear`*`, `*`UTseconds`*`, `*`action`*`)` specifies out-of-range input action.

This function has the limitations of the NRLMSISE-00 model. For more information, see the NRLMSISE-00 model documentation.

The NRLMSISE-00 model uses *`UTseconds`*, *`localApparentSolarTime`*, and *`longitude`* independently. These arguments are not of equal importance for every situation. For the most physically realistic calculation, choose these three variables to be consistent by default:

```
localApparentSolarTime = UTseconds/3600 + longitude/15
```

If available, you can include departures from this equation for *`localApparentSolarTime`*, but they are of minor importance.

Input Arguments

`action`

Action for out-of-range input. Specify one:
`'Error'`
`'Warning'` (default)
`'None'`

`altitude`

Array of *m*-by-1 altitudes, in meters.

dayOfYear

Array *m*-by-1 day of year.

f107Average

Array of *m*-by-1 81 day average of F10.7 flux (centered on day of year (*dayOfYear*)). If you specify*f107Average*, you must also specify *f107Daily* and *magneticIndex*. The effects of *f107Average* are not large or established below 80,000 m; therefore, the default value is 150.

These *f107Average* values correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. The *f107Average* values do not correspond to the radio flux at 1 AU. The following site provides both classes of values: `ftp://ftp.ngdc.noaa.gov/STP/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/`

See the limitations in “Description” on page 4-120 for more information.

f107Daily

Array of *m*-by-1 daily F10.7 flux for previous day. If you specify *f107Daily*, you must also specify *f107Average* and *magneticIndex*. The effects of *f107Daily* are not large or established below 80,000 m; therefore, the default value is 150.

These *f107Daily* values correspond to the 10.7 cm radio flux at the actual distance of the Earth from the Sun. The *f107Daily* values do not correspond to the radio flux at 1 AU. The following site provides both classes of values: `ftp://ftp.ngdc.noaa.gov/STP/space-weather/solar-data/solar-features/solar-radio/noontime-flux/penticton/`

See the limitations in “Description” on page 4-120 for more information.

flags

Array of 23 to enable or disable particular variations for the outputs. If *flags* array length, *m*, is 23 and you have not specified all available inputs, this function assumes that *flags* is set.

The flags, associated with the *flags* input, enable or disable particular variations for the outputs:

Field	Description
`Flags(1)`	F10.7 effect on mean
`Flags(2)`	Independent of time
`Flags(3)`	Symmetrical annual
`Flags(4)`	Symmetrical semiannual
`Flags(5)`	Asymmetrical annual
`Flags(6)`	Asymmetrical semiannual
`Flags(7)`	Diurnal
`Flags(8)`	Semidiurnal
`Flags(9)`	Daily AP. If you set this field to -1, the function uses the entire matrix of magnetic index information (APH) instead of `APH(:,1)`.
`Flags(10)`	All UT, longitudinal effects
`Flags(11)`	Longitudinal
`Flags(12)`	UT and mixed UT, longitudinal
`Flags(13)`	Mixed AP, UT, longitudinal
`Flags(14)`	Terdiurnal
`Flags(15)`	Departures from diffusive equilibrium
`Flags(16)`	All exospheric temperature variations
`Flags(17)`	All variations from 120,000 meter temperature (TLB)
`Flags(18)`	All lower thermosphere (TN1) temperature variations
`Flags(19)`	All 120,000 meter gradient (S) variations
`Flags(20)`	All upper stratosphere (TN2) temperature variations
`Flags(21)`	All variations from 120,000 meter values (ZLB)
`Flags(22)`	All lower mesosphere temperature (TN3) variations
`Flags(23)`	Turbopause scale height variations

`latitude`

Array of *m*-by-1 geodetic latitudes, in degrees.

`longitude`

Array of *m*-by-1 longitudes, in degrees.

localApparentSolarTime

Array of *m*-by-1 local apparent solar time (hours). To obtain a physically realistic value, the function sets *localApparentSolarTime* to (`sec/3600 + lon/15`) by default. See “Description” on page 4-120 for more information.

magneticIndex

An array of *m*-by-7 of magnetic index information. If you specify *magneticIndex*, you must also specify *f107Average* and *f107Daily*. This information consists of:
Daily magnetic index (AP)
3 hour AP for current time
3 hour AP for 3 hours before current time
3 hour AP for 6 hours before current time
3 hour AP for 9 hours before current time
Average of eight 3 hour AP indices from 12 to 33 hours before current time
Average of eight 3 hour AP indices from 36 to 57 hours before current time

The effects of daily magnetic index are not large or established below 80,000 m. As a result, the function sets the default value to 4. See the limitations in “Description” on page 4-120 for more information.

otype

Total mass density output:

`'Oxygen'`	Total mass density outputs include anomalous oxygen number density.
`'NoOxygen'`	Total mass density outputs do not include anomalous oxygen number density.

UTseconds

Array of *m*-by-1 seconds in day in universal time (UT)

year

This function ignores the value of *year*.

Output Arguments

`T`

Array of *N*-by-2 values of temperature, in kelvin. The first column is exospheric temperature, in kelvin. The second column is temperature at altitude, in kelvin.

`rho`

An array of *N*-by-9 values of densities (kg/m^3 or 1/m^3) in selected density units. The column order is:
Density of He, in 1/m^3
Density of O, in 1/m^3
Density of N2, in 1/m^3
Density of O2, in 1/m^3
Density of Ar, in 1/m^3
Total mass density, in kg/m^3
Density of H, in 1/m^3
Density of N, in 1/m^3
Anomalous oxygen number density, in 1/m^3

`density(6)`, total mass density, is the sum of the mass densities of He, O, N2, O2, Ar, H, and N. Optionally, `density(6)` can include the mass density of anomalous oxygen making `density(6)`, the effective total mass density for drag.

Examples

Calculate the temperatures, densities not including anomalous oxygen using the NRLMSISE-00 model at 10,000 m, 45 degrees latitude, -50 degrees longitude. This calculation uses the date January 4, 2007 at 0 UT. It uses default values for flux, magnetic index data, and local solar time with out-of-range actions generating warnings:

```
[T, rho] = atmosnrlmsise00( 10000, 45, -50, 2007, 4, 0)

T =
   1.0e+03 *

    1.0273    0.2212

rho =

   1.0e+24 *

    0.0000         0    6.6824    1.7927    0.0799    0.0000         0         0         0
```

Calculate the temperatures, densities not including anomalous oxygen using the NRLMSISE-00 model. Use the model at 10,000 m, 45 degrees latitude, –50 degrees longitude and 25,000 m, 47 degrees latitude, –55 degrees longitude.

This calculation uses the date January 4, 2007 at 0 UT. It uses default values for flux, magnetic index data, and local solar time with out-of-range actions generating warnings:

```
[T, rho] = atmosnrlmsise00( [10000; 25000], [45; 47], ...
[-50; -55], [2007; 2007], [4; 4], [0; 0])

T =

  1.0e+003 *

    1.0273    0.2212
    1.0273    0.2116


rho =

  1.0e+024 *

    0.0000     0    6.6824    1.7927    0.0799    0.0000     0     0     0
    0.0000     0    0.6347    0.1703    0.0076    0.0000     0     0     0
```

Calculate the temperatures, densities including anomalous oxygen using the NRLMSISE-00 model at 10,000 m, 45 degrees latitude, –50 degrees longitude. This calculation uses the date January 4, 2007 at 0 UT. It uses default values for flux, magnetic index data, and local solar time with out-of-range actions generating errors:

```
[T, rho] = atmosnrlmsise00( 10000, 45, -50, 2007, ...
4, 0, 'Oxygen', 'Error')

T =

  1.0e+003 *

    1.0273    0.2212


rho =

  1.0e+024 *

    0.0000     0    6.6824    1.7927    0.0799    0.0000     0     0     0
```

Calculate the temperatures, densities including anomalous oxygen using the NRLMSISE-00 model at 100,000 m, 45 degrees latitude, –50 degrees longitude. This calculation uses the date January 4, 2007 at 0 UT. It uses defined values for flux, and magnetic index data, and default local solar time. It specifies that the out-of-range action is to generate no message:

```
aph = [17.375 15 20 15 27 (32+22+15+22+9+18+12+15)/8 (39+27+9+32+39+9+7+12)/8]
f107 = 87.7
nov_6days  = [ 78.6 78.2 82.4 85.5 85.0 84.1]
dec_31daymean = 84.5
jan_31daymean = 83.5
feb_13days = [ 89.9 90.3 87.3 83.7 83.0 81.9 82.0 78.4 76.7 75.9 74.7 73.6 72.7]
f107a = (sum(nov_6days) + sum(feb_13days) + (dec_31daymean + jan_31daymean)*31)/81
flags = ones(1,23)
flags(9) = -1
[T, rho] = atmosnrlmsise00( 100000, 45, -50, 2007, 4, 0, f107a, f107, ...
aph, flags, 'Oxygen', 'None')
aph =

   17.3750   15.0000   20.0000   15.0000   27.0000   18.1250   21.7500


f107 =

   87.7000


nov_6days =

   78.6000   78.2000   82.4000   85.5000   85.0000   84.1000


dec_31daymean =

   84.5000


jan_31daymean =

   83.5000


feb_13days =

  Columns 1 through 10

   89.9000 90.3000 87.3000 83.7000 83.0000 81.9000 82.0000 78.4000 76.7000 75.9000

  Columns 11 through 13

   74.7000   73.6000   72.7000


f107a =

   83.3568


flags =

  Columns 1 through 17

     1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

  Columns 18 through 23

     1  1  1  1  1  1
```

```
flags =

  Columns 1 through 17

     1  1  1  1  1  1  1  1 -1  1  1  1  1  1  1  1  1

  Columns 18 through 23

     1  1  1  1  1  1


T =

  1.0e+003 *

    1.0273    0.1917


rho =

  1.0e+018 *

    0.0001  0.4241  7.8432  1.9721  0.0808  0.0000  0.0000  0.0000  0.0000
```

References

`https://ccmc.gsfc.nasa.gov/modelweb/atmos/nrlmsise00.html`

See Also

`atmoscira`

Introduced in R2007b

atmospalt

Calculate pressure altitude based on ambient pressure

Syntax

`pressureAltitude = atmospalt(pressure, action)`

Description

`pressureAltitude = atmospalt(pressure, action)` computes the pressure altitude based on ambient pressure. Pressure altitude is the altitude with specified ambient pressure in the 1976 Committee on Extension to the Standard Atmosphere (COESA) United States standard. Pressure altitude is the same as the mean sea level (MSL) altitude.

This function extrapolates altitude values logarithmically below the pressure of 0.3961 Pa (approximately 0.00006 psi) and above the pressure of 101,325 Pa (approximately 14.7 psi).

This function assumes that air is dry and an ideal gas.

Input Arguments

`pressure`

Array of *m*-by-1 ambient pressures, in pascal.

`action`

Action for out-of-range input. Specify one:
`'Error'`
`'Warning'` (default)
`'None'`

Output Arguments

`pressureAltitude`

Array of *m*-by-1 pressure altitudes or MSL altitudes, in meters.

Examples

Calculate the pressure altitude at a static pressure of 101,325 Pa with warnings for out-of-range inputs:

```
h = atmospalt(101325)


h =

     0
```

Calculate the pressure altitude at static pressures of 101,325 Pa and 26,436 Pa with errors for out-of-range inputs:

```
h = atmospalt([101325 26436], 'Error' )


h =

  1.0e+004 *

         0    1.0000
```

References

U.S. Standard Atmosphere, 1976, U.S. Government Printing Office, Washington, D.C.

See Also

`atmoscira` | `atmoscoesa` | `atmosisa` | `atmoslapse` | `atmosnonstd`

Introduced in R2006b

Body (Aero.Body)

Construct body object for use with animation object

Syntax

```
h = Aero.Body
```

Description

`h = Aero.Body` constructs a body for an animation object. The animation object is returned in `h`. To use the Aero.Body object, you typically:

1. Create the animation body.
2. Configure or customize the body object.
3. Load the body.
4. Generate patches for the body (requires an axes from a figure).
5. Set the source for the time series data.
6. Move or update the body.

The animation object has the following properties:

By default, an `Aero.Body` object natively uses aerospace body coordinates for the body geometry and the time series data. Convert time series data from other coordinate systems on the fly by registering a different `CoordTransformFcn` function.

See `Aero.Body` for further details.

See Also

`Aero.Body`

Introduced in R2007a

Camera (Aero.Camera)

Construct camera object for use with animation object

Syntax

```
h = Aero.Camera
```

Description

`h = Aero.Camera` constructs a camera object `h` for use with an animation object. The camera object uses the registered coordinate transform. By default, this is an aerospace body coordinate system. Axes of custom coordinate systems must be orthogonal.

The animation object has the following properties:

By default, an `Aero.Body` object natively uses aerospace body coordinates for the body geometry and the time series data. Convert time series data from other coordinate systems on the fly by registering a different `CoordTransformFcn` function.

See `Aero.Camera` for further details.

See Also

`Aero.Camera`

Introduced in R2007a

ClearTimer (Aero.FlightGearAnimation)

Clear and delete timer for animation of FlightGear flight simulator

Syntax

```
ClearTimer(h)
h.ClearTimer
```

Description

`ClearTimer(h)` and `h.ClearTimer` clear and delete the MATLAB timer for the animation of the FlightGear flight simulator.

Examples

Clear and delete the MATLAB timer for animation of the FlightGear animation object, `h`:

```
h = Aero.FlightGearAnimation
h.SetTimer
h.ClearTimer
h.SetTimer('FGTimer')
```

See Also

`SetTimer`

Introduced in R2008b

ClimbIndicator Properties

Control climb indicator appearance and behavior

Description

Climb indicators are components that represent a climb indicator. Properties control the appearance and behavior of a climb indicator. Use dot notation to refer to a particular object and property.

```
f = uifigure;
climbindicator = uiaeroclimb(f);
climbindicator.ClimbRate = 100;
```

The climb rate indicator displays measurements for an aircraft climb rate in ft/min.

The needle covers the top semicircle, if the velocity is positive, and the lower semicircle, if the climb rate is negative. The range of the indicator is from –**Maximum** feet per minute to **Maximum** feet per minute. Major ticks indicate **Maximum**/4. Minor ticks indicate **Maximum**/8 and **Maximum**/80.

Properties

Climb Indicator

`ClimbRate` — Climb rate of aircraft

`0` (default) | finite, real, and scalar numeric

Climb rate of the aircraft, specified as a finite, real, and scalar numeric, in ft/min.

Example: `60`

Dependencies

Specifying this value changes the value of `Value`.

Data Types: `double`

`MaximumRate` — Maximum gauge scale values

`0` (default) | finite, real, positive, and scalar numeric

Maximum gauge scale values, specified as a finite, real, positive, and scalar numeric, representing the +/- maximum climb rate, in ft/min.

Example: `100`

Data Types: `double`

`Value` — Climb rate of aircraft
`0` (default) | finite, real, and scalar numeric

Climb rate of the aircraft, specified as a finite, real, and scalar numeric, in ft/min.

Example: `60`

Dependencies

Specifying this value changes the value of `ClimbRate`.

Data Types: `double`

Interactivity

`Visible` — Visibility of climb indicator
`'on'` (default) | `'off'`

Visibility of the climb indicator, specified as `'on'` or `'off'`. The `Visible` property determines whether the climb indicator is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire climb indicator is hidden, but you can still specify and access its properties.

`Enable` — Operational state of climb indicator
`'on'` (default) | `'off'`

Operational state of climb indicator, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the climb indicator indicates that the climb indicator is operational.
- If you set this property to `'off'`, then the appearance of the climb indicator appears dimmed, indicating that the climb indicator is not operational.

Position

`Position` — Location and size of climb indicator
`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the climb indicator relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the climb indicator
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the climb indicator
`width`	Distance between the right and left outer edges of the climb indicator
`height`	Distance between the top and bottom outer edges of the climb indicator

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[200 120 120 120]`

`InnerPosition` — Inner location and size of climb indicator
`[100 100 120 120]` (default) | `[left bottom width height]`

Inner location and size of the climb indicator, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of climb indicator
`[100 100 120 120]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the climb indicator returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an climb rate indicator in the third row and second column of its parent grid.

```
g = ui([4 3]);
gauge = uiaeroclimb(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the climb rate indicator span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this climb rate indicator spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

BusyAction — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Identifiers

Type — Type of graphics object
`'uiaeroclimb'`

This property is read-only.

Type of graphics object, returned as `'uiaerohorizon'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also

`uiaeroclimb`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

convacc

Convert from acceleration units to desired acceleration units

Syntax

```
convertedValues = convacc(valuesToConvert, inputAccelUnits,
outputAccelUnits)
```

Description

`convertedValues = convacc(valuesToConvert, inputAccelUnits, outputAccelUnits)` computes the conversion factor from specified input acceleration units to specified output acceleration units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

`valuesToConvert`

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *`inputAccelUnits`* to *`outputAccelUnits`*.

`inputAccelUnits`

Specified input acceleration units. Supported units are:

`'ft/s^2'`	Feet per second squared
`'m/s^2'`	Meters per second squared
`'km/s^2'`	Kilometers per second squared
`'in/s^2'`	Inches per second squared
`'km/h-s'`	Kilometers per hour per second
`'mph/s'`	Miles per hour per second

`'G''s'`	g-units

`outputAccelUnits`

Specified output acceleration units. Supported units are:

`'ft/s^2'`	Feet per second squared
`'m/s^2'`	Meters per second squared
`'km/s^2'`	Kilometers per second squared
`'in/s^2'`	Inches per second squared
`'km/h-s'`	Kilometers per hour per second
`'mph/s'`	Miles per hour per second
`'G''s'`	g-units

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three accelerations from feet per second squared to meters per second squared:

```
a = convacc([3 10 20],'ft/s^2','m/s^2')

a =

    0.9144    3.0480    6.0960
```

See Also

`convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convang

Convert from angle units to desired angle units

Syntax

convertedValues = convang(*valuesToConvert*, *inputAngleUnits*, *outputAngleUnits*)

Description

convertedValues = convang(*valuesToConvert*, *inputAngleUnits*, *outputAngleUnits*) computes the conversion factor from specified input angle units to specified output angle units. It then applies the conversion factor to the input to produce the output in the desired units. *inputAngleUnits* and *outputAngleUnits* are character vectors or strings.

Input Arguments

valuesToConvert

Floating-point array of size *m*-by-*n* values the function is to convert. All values must have the same unit conversions from *inputAngleUnits* to *outputAngleUnits*.

inputAngleUnits

Specified input angle units. Supported units are:

`'deg'`	Degrees
`'rad'`	Radians
`'rev'`	Revolutions

outputAngleUnits

Specified output angle units. Supported units are:

`'deg'`	Degrees
`'rad'`	Radians
`'rev'`	Revolutions

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three angles from degrees to radians:

```
a = convang([3 10 20],'deg','rad')

a =

    0.0524    0.1745    0.3491
```

See Also

`convacc` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convangacc

Convert from angular acceleration units to desired angular acceleration units

Syntax

`convertedValues = convangacc(valuesToConvert, inputAngularUnits, outputAngularUnits)`

Description

`convertedValues` = `convangacc(`*`valuesToConvert`*`, `*`inputAngularUnits`*`, `*`outputAngularUnits`*`)` computes the conversion factor from specified input angular acceleration units to specified output angular acceleration units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

`valuesToConvert`

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *`inputAngularUnits`* to *`outputAngularUnits`*.

`inputAngularUnits`

Specified input angular acceleration units. Supported units are:

`'deg/s^2'`	Degrees per second squared
`'rad/s^2'`	Radians per second squared
`'rpm/s'`	Revolutions per minute per second

`outputAngularUnits`

Specified output angular acceleration units. Supported units are:

`'deg/s^2'`	Degrees per second squared
`'rad/s^2'`	Radians per second squared
`'rpm/s'`	Revolutions per minute per second

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three angular accelerations from degrees per second squared to radians per second squared:

```
a = convangacc([0.3 0.1 0.5],'deg/s^2','rad/s^2')

a =

    0.0052    0.0017    0.0087
```

See Also

`convacc` | `convang` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convangvel

Convert from angular velocity units to desired angular velocity units

Syntax

`convertedValues = convangvel(valuesToConvert, inputAngularVelocityUnits, outputAngularVelocityUnits)`

Description

`convertedValues = convangvel(valuesToConvert, inputAngularVelocityUnits, outputAngularVelocityUnits)` computes the conversion factor from specified input angular velocity units to specified output angular velocity units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

`valuesToConvert`

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *`inputAngularVelocityUnits`* to *`outputAngularVelocityUnits`*.

`inputAngularVelocityUnits`

Specified input angular velocity units. Supported units are:

`'deg/s'`	Degrees per second
`'rad/s'`	Radians per second
`'rpm'`	Revolutions per minute

outputAngularVelocityUnits

Specified output angular velocity units. Supported units are:

`'deg/s'`	Degrees per second
`'rad/s'`	Radians per second
`'rpm'`	Revolutions per minute

Output Arguments

convertedValues

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three angular velocities from degrees per second to radians per second:

```
a = convangvel([0.3 0.1 0.5],'deg/s','rad/s')

a =

    0.0052    0.0017    0.0087
```

See Also

`convacc` | `convang` | `convangacc` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convdensity

Convert from density units to desired density units

Syntax

convertedValues = `convdensity(`*valuesToConvert*, *inputDensityUnits*, *outputDensityUnits*`)`

Description

convertedValues = `convdensity(`*valuesToConvert*, *inputDensityUnits*, *outputDensityUnits*`)` computes the conversion factor from specified input density units to specified output density units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

`valuesToConvert`

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *inputDensityUnits* to *outputDensityUnits*.

`inputDensityUnits`

Specified input density units. Supported units are:

`'lbm/ft^3'`	Pound mass per feet cubed
`'kg/m^3'`	Kilograms per meters cubed
`'slug/ft^3'`	Slugs per feet cubed
`'lbm/in^3'`	Pound mass per inch cubed

`outputDensityUnits`

Specified output density units. Supported units are:

`'lbm/ft^3'`	Pound mass per feet cubed
`'kg/m^3'`	Kilograms per meters cubed
`'slug/ft^3'`	Slugs per feet cubed
`'lbm/in^3'`	Pound mass per inch cubed

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three densities from pound mass per feet cubed to kilograms per meters cubed:

```
a = convdensity([0.3 0.1 0.5],'lbm/ft^3','kg/m^3')

a =

    4.8055    1.6018    8.0092
```

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convforce

Convert from force units to desired force units

Syntax

`convertedValues` `= convforce(`*`valuesToConvert`*`,` *`inputForceUnits`*`,` *`outputForceUnits`*`)`

Description

`convertedValues` `= convforce(`*`valuesToConvert`*`,` *`inputForceUnits`*`,` *`outputForceUnits`*`)` computes the conversion factor from specified input force units to specified output force units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

`valuesToConvert`

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *`inputForceUnits`* to *`outputForceUnits`*.

`inputForceUnits`

Specified input force units. Supported units are:

`'lbf'`	Pound force
`'N'`	Newton

`outputForceUnits`

Specified output force units. Supported units are:

`'lbf'`	Pound force

`'N'`	Newton

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three forces from pound force to newtons:

```
a = convforce([120 1 5],'lbf','N')

a =

  533.7866    4.4482   22.2411
```

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convlength` | `convmass` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convlength

Convert from length units to desired length units

Syntax

```
convertedValues = convlength(valuesToConvert,inputLengthUnits,
outputLengthUnits)
```

Description

`convertedValues = convlength(valuesToConvert,inputLengthUnits,`
`outputLengthUnits)` converts `valuesToConvert` from original units to desired units using computed conversion factor.

Examples

Convert Lengths

Convert three lengths from feet to meters.

```
a = convlength([3 10 20],'ft','m')
```

```
a = 1×3

    0.9144    3.0480    6.0960
```

Input Arguments

`valuesToConvert` — Input lengths to convert
floating-point array of *m*-by-*n* values

Input lengths to convert, specified as a floating-point array of *m*-by-*n* values, in original units. All values must have the same units.

Data Types: `double`

`inputLengthUnits` — Original unit
`'ft'` | `'m'` | `'km'` | `'in'` | `'naut mi'`

Original unit of input lengths, specified as:

`'ft'`	Feet
`'m'`	Meters
`'km'`	Kilometers
`'in'`	Inches
`'mi'`	Miles
`'naut mi'`	Nautical miles

Data Types: `char` | `string`

`outputLengthUnits` — Units to convert to
`'ft'` | `'m'` | `'km'` | `'in'` | `'naut mi'`

New unit to convert to, specified as:

`'ft'`	Feet
`'m'`	Meters
`'km'`	Kilometers
`'in'`	Inches
`'mi'`	Miles
`'naut mi'`	Nautical miles

Data Types: `char` | `string`

Output Arguments

`convertedValues` — Converted lengths
floating-point array of size *m*-by-*n* values

Converted lengths, returned in new units.

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convmass` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convmass

Convert from mass units to desired mass units

Syntax

convertedValues = convmass(*valuesToConvert*, *inputMassUnits*, *outputMassUnits*)

Description

convertedValues = convmass(*valuesToConvert*, *inputMassUnits*, *outputMassUnits*) computes the conversion factor from specified input mass units to specified output mass units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

valuesToConvert

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *inputMassUnits* to *outputMassUnits*.

inputMassUnits

Specified input mass units. Supported units are:

'lbm'	Pound mass
'kg'	Kilograms
'slug'	Slugs

outputMassUnits

Specified output mass units. Supported units are:

`'lbm'`	Pound mass
`'kg'`	Kilograms
`'slug'`	Slugs

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three masses from pound mass to kilograms:

```
a = convmass([3 1 5],'lbm','kg')

a =

    1.3608    0.4536    2.2680
```

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convpres` | `convtemp` | `convvel`

Introduced in R2006b

convpres

Convert from pressure units to desired pressure units

Syntax

`convertedValues``= convpres(`*`valuesToConvert`*`, `*`inputPressureUnits`*`, `*`outputPressureUnits`*`)`

Description

`convertedValues``= convpres(`*`valuesToConvert`*`, `*`inputPressureUnits`*`, `*`outputPressureUnits`*`)` computes the conversion factor from specified input pressure units to specified output pressure units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

`valuesToConvert`

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *`inputPressureUnits`* to *`outputPressureUnits`*.

`inputPressureUnits`

Specified input pressure units. Supported units are:

`'psi'`	Pound force per square inch
`'Pa'`	Pascal
`'psf'`	Pound force per square foot
`'atm'`	Atmosphere

`outputPressureUnits`

Specified output pressure units. Supported units are:

`'psi'`	Pound force per square inch
`'Pa'`	Pascal
`'psf'`	Pound force per square foot
`'atm'`	Atmosphere

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert two pressures from pound force per square inch to atmospheres:

```
a = convpres([14.696  35],'psi','atm')

a =

    1.0000    2.3816
```

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convtemp` | `convvel`

Introduced in R2006b

convtemp

Convert to desired temperature units

Syntax

```
convertedValues = convtemp(valuesToConvert,inputTemperatureUnits,
outputTemperatureUnits)
```

Description

`convertedValues = convtemp(valuesToConvert,inputTemperatureUnits,
outputTemperatureUnits)` computes the conversion factor from specified input temperature units (`inputTemperatureUnits`) to specified output temperature units (`outputTemperatureUnits`). The function then applies the conversion factor to the `valuesToConvert`.

Examples

Convert Temperatures

Convert temperatures

Convert three temperatures from degrees Celsius to degrees Fahrenheit.

```
a = convtemp([0 100 15],'C','F')
```

```
a = 1×3
```

```
   32.0000  212.0000   59.0000
```

Input Arguments

`valuesToConvert` — Temperatures to convert

m-by-*n* floating-point array

Temperatures to convert, specified as an *m*-by-*n* floating-point array. All values must have the same units to be converted.

Data Types: `double` | `single`

`inputTemperatureUnits` — Unit of temperature to convert

`'K'` | `'F'` | `'C'` | `'R'`

Unit of temperature to convert:

`'K'`	Kelvin
`'F'`	Degree Fahrenheit
`'C'`	Degree Celsius
`'R'`	Degree Rankine

Data Types: `char` | `string`

`outputTemperatureUnits` — Unit of temperature to convert to

`'K'` | `'F'` | `'C'` | `'R'`

Unit of temperature to convert to:

`'K'`	Kelvin
`'F'`	Degree Fahrenheit
`'C'`	Degree Celsius
`'R'`	Degree Rankine

Data Types: `char` | `string`

Output Arguments

`convertedValues` — Converted temperature
m-by-*n* floating-point array

Converted temperature, output as a floating-point array.

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convvel`

Topics

"Floating-Point Numbers" (MATLAB)

Introduced in R2006b

convvel

Convert from velocity units to desired velocity units

Syntax

`convertedValues = convvel(valuesToConvert, inputVelocityUnits, outputVelocityUnits)`

Description

`convertedValues` `= convvel(`*`valuesToConvert`*`,` *`inputVelocityUnits`*`,` *`outputVelocityUnits`*`)` computes the conversion factor from specified input velocity units to specified output velocity units. It then applies the conversion factor to the input to produce the output in the desired units.

Input Arguments

valuesToConvert

Floating-point array of size *m*-by-*n* values that the function is to convert. All values must have the same unit conversions from *`inputVelocityUnits`* to *`outputVelocityUnits`*.

inputVelocityUnits

Specified input velocity units. Supported units are:

`'ft/s'`	Feet per second
`'m/s'`	Meters per second
`'km/s'`	Kilometers per second
`'in/s'`	Inches per second
`'km/h'`	Kilometers per hour
`'mph'`	Miles per hour

`'kts'`	Knots
`'ft/min'`	Feet per minute

`outputVelocityUnits`

Specified output velocity units. Supported units are:

`'ft/s'`	Feet per second
`'m/s'`	Meters per second
`'km/s'`	Kilometers per second
`'in/s'`	Inches per second
`'km/h'`	Kilometers per hour
`'mph'`	Miles per hour
`'kts'`	Knots
`'ft/min'`	Feet per minute

Output Arguments

`convertedValues`

Floating-point array of size *m*-by-*n* values that the function has converted.

Examples

Convert three velocities from feet per minute to meters per second:

```
a = convvel([30 100 250],'ft/min','m/s')

a =

    0.1524    0.5080    1.2700
```

See Also

`convacc` | `convang` | `convangacc` | `convangvel` | `convdensity` | `convforce` | `convlength` | `convmass` | `convpres` | `convtemp`

Introduced in R2006b

correctairspeed

Convert from one of other two airspeeds to equivalent airspeed (EAS), calibrated airspeed (CAS), or true airspeed (TAS)

Syntax

outputAirpseed = `correctairspeed(`*inputAirspeed*, *speedOfSound*, *pressure0*, *inputAirspeedType*, *outputAirspeedType*`)`
outputAirpseed = `correctairspeed(`*inputAirspeed*, *speedOfSound*, *pressure0*, *inputAirspeedType*, *outputAirspeedType*, *method*`)`

Description

outputAirpseed = `correctairspeed(`*inputAirspeed*, *speedOfSound*, *pressure0*, *inputAirspeedType*, *outputAirspeedType*`)` computes the conversion factor from specified input airspeed to specified output airspeed using speed of sound and static pressure. The function applies the conversion factor to the input airspeed to produce the output in the desired airspeed.

outputAirpseed = `correctairspeed(`*inputAirspeed*, *speedOfSound*, *pressure0*, *inputAirspeedType*, *outputAirspeedType*, *method*`)` uses the specified method to compute the conversion factor.

Input Arguments

inputAirspeed

Floating-point array of size *m*-by-1 of airspeeds in meters per second. All values must have the same airspeed conversions from *inputAirspeedType* to *outputAirspeedType*.

speedOfSound

Floating-point array of size *m*-by-1 of speeds of sound, in meters per second.

`pressure0`

Floating-point array of size *m*-by-1 of static air pressures, in pascal.

`inputAirspeedType`

Input airspeed, specified as a string. Supported airspeeds are:

`'TAS'`	True airspeed
`'CAS'`	Calibrated airspeed
`'EAS'`	Equivalent airspeed

`outputAirspeedType`

Output airspeed, specified as a string. Supported airspeeds are:

`'TAS'`	True airspeed
`'CAS'`	Calibrated airspeed
`'EAS'`	Equivalent airspeed

`method`

Specify a method, as a string, for computing the conversion factor.

`'TableLookup'`	(Default) Generate output airspeed by looking up or estimating table values based on inputs *`inputAirspeed`*, *`speedOfSound`*, and *`pressure0`*. The `'TableLookup'` method is not recommended for either of these instances: • *`speedOfSound`* less than 200 m/s or greater than 350 m/s. • *`pressure0`* less than 1000 Pa or greater than 106,500 Pa. Using the `'TableLookup'` method in these instances causes inaccuracies.

`'Equation'`	Compute output airspeed directly using input values *`inputAirspeed`*, *`speedOfSound`*, and *`pressure0`*. Calculations involving supersonic airspeeds (greater than Mach 1) require an iterative computation. If the function does not conclude within 30 iterations, it displays an error message.

The `correctairspeed` function automatically uses the `'Equation'` method for any of these instances:

- Conversion with *`inputAirspeedType`* set to `'TAS'` and *`outputAirspeedType`* set to `'EAS'`.
- Conversion with *`inputAirspeedType`* set to `'EAS'` and *`outputAirspeedType`* set to `'TAS'`.
- Conversion with *`inputAirspeed`* is greater than five times the speed of sound at sea level (approximately 1700 m/s).

Output Arguments

`outputAirpseed`

Floating-point array of size *m*-by-1 of airspeeds in meters per second.

Examples

Convert three airspeeds from true airspeed (`'TAS'`) to equivalent airspeed (`'EAS'`) at 1,000 ms using the `'TableLookup'` method:

```
ain = [25.7222; 10.2889; 3.0867];
as = correctairspeed(ain,336.4,89874.6,'TAS','CAS','TableLookup')
as =

   24.5077
    9.8024
    2.9407
```

Convert airspeeds from calibrated airspeed (`'CAS'`) to equivalent airspeed (`'EAS'`) at 1,000 m and 0 m using the `'Equation'` method:

```
ain = [25.7222; 10.2889; 3.0867];
sos = [336.4; 340.3; 340.3];
```

```
P0 = [ 89874.6; 101325; 101325];
as = correctairspeed(ain,sos,P0,'CAS','EAS','Equation')
as =

   25.7199
   10.2889
    3.0867
```

Convert airspeed from true airspeed (`'TAS'`) to equivalent airspeed (`'EAS'`) at 15,000 meters. Use the `atmoscoesa` function to first calculate the speed of sound (*`sos`*) and static air pressure (*`P0`*):

```
ain = 376.25;
[~, sos, P0, ~] = atmoscoesa(15000);
as = correctairspeed( ain, sos, P0, 'EAS', 'TAS')
as =

  149.6042
```

Limitations

This function assumes that air flow is compressible dry air with constant specific heat ratio (gamma).

References

Lowry, J.T., *Performance of Light Aircraft*, AIAA Education Series, Washington, D.C., 1999

Aeronautical Vestpocket Handbook, United Technologies Pratt & Whitney, August1986

Gracey, William, *Measurement of Aircraft Speed and Altitude*, NASA Reference Publication 1046, 1980.

See Also

`airspeed` | `atmoscoesa` | `atmosisa` | `atmoslapse` | `atmosnonstd`

Introduced in R2006b

createBody

Class: `Aero.Animation`
Package: `Aero`

Create body and its associated patches in animation

Syntax

```
idx = createBody(h,bodyDataSrc)
idx = h.createBody(bodyDataSrc)
idx = createBody(h,bodyDataSrc,geometrysource)
idx = h.createBody(bodyDataSrc,geometrysource)
```

Description

`idx = createBody(h,bodyDataSrc)` and `idx = h.createBody(bodyDataSrc)` create a new body using the `bodyDataSrc`, makes its patches, and adds it to the animation object `h`. This command assumes a default geometry source type set to `Auto`.

`idx = createBody(h,bodyDataSrc,geometrysource)` and `idx = h.createBody(bodyDataSrc,geometrysource)` create a new body using the `bodyDataSrc` file, makes its patches, and adds it to the animation object `h`. `geometrysource` is the geometry source type for the body.

Input Arguments

`bodyDataSrc`	Source of data for body.

`geometrysource`	Geometry source type for body: • `Auto` — Recognizes `.mat` extensions as MAT-files, `.ac` extensions as Ac3d files, and structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables. Default. • `Variable` — Recognizes structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables. • `MatFile` — Recognizes `.mat` extensions as MAT-files. • `Ac3d` — Recognizes `.ac` extensions as Ac3d files. • `Custom` — Recognizes custom extensions.

Output Arguments

`idx`	Index of the body to be created.

Examples

Create a body for the animation object, `h`. Use the Ac3d format data source `pa24-250_orange.ac`, for the body.

```
h = Aero.Animation;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
```

datcomimport

Bring DATCOM file into MATLAB environment

Syntax

```
aero = datcomimport(file)
aero = datcomimport(file,usenan)
aero = datcomimport(file,usenan,verbose)
aero = datcomimport( ___ ,filetype)
```

Description

`aero = datcomimport(file)` imports aerodynamic data from `file` into `aero`. Before reading the United States Air Force Digital DATCOM file, `datcomimport` initializes values to 99999 when there is not a full set of data for the DATCOM case.

`aero = datcomimport(file,usenan)` replaces data points with NaN or zero where no DATCOM methods exist or where the method is not applicable.

`aero = datcomimport(file,usenan,verbose)` displays the status of the DATCOM file being read in the MATLAB Command Window.

`aero = datcomimport( ___ ,filetype)` imports a DATCOM of a particular USAF Digital DATCOM file type. Specify any of the input argument combinations in the previous syntaxes, followed by the file type.

Examples

Read 1976 Version of Digital DATCOM File

Read the 1976 version Digital DATCOM output file `astdatcom.out`.

```
aero = datcomimport('astdatcom.out')
aero =

  1×1 cell array

    {1×1 struct}
```

Read USAF Digital DATCOM Output File Replacing Data Points with Zeroes

Read the 1976 Digital DATCOM output file `astdatcom.out` using zeros to replace data points where no DATCOM methods exist. Use *usenanvar* variable to set `usenan` argument to `false`.

```
usenanvar = false;
aero = datcomimport('astdatcom.out',usenanvar)
```

```
aero =

  1×1 cell array

    {1×1 struct}
```

Read USAF Digital DATCOM Output File Replacing Data Points with Zeroes Specifying Verbose Settings

Read the 1976 Digital DATCOM output file `astdatcom.out` using zeros to replace data points where no DATCOM methods exist and displaying status information in the MATLAB Command Window. Use *usenanvar* variable to set `usenan` argument to `false`.

```
usenanvar = false;
aero = datcomimport('astdatcom.out',usenanvar,1)
```

```
Loading file 'astdatcom.out'.
Reading input data from file 'astdatcom.out'.
Reading output data from file 'astdatcom.out'.
aero =

  1×1 cell array

    {1×1 struct}
```

Read USAF Digital DATCOM Output File Replacing Data Points with Zeroes and Specifying Verbose Settings and DATCOM File Type

Read the 1976 Digital DATCOM output file `astdatcom.out` using NaNs to replace data points where no DATCOM methods exist, displaying status information in the MATLAB Command Window, and specifying the DATCOM output file type. Use *usenanvar* variable to set `usenan` argument to `true`.

```
usenanvar = true;
aero = datcomimport('astdatcom.out',usenanvar,1,6)
```

```
Loading file 'astdatcom.out'.
Reading input data from file 'astdatcom.out'.
Reading output data from file 'astdatcom.out'.
aero =

  1×1 cell array

    {1×1 struct}
```

Input Arguments

`file` — DATCOM file

character vector | cell array of file names

Digital DATCOM output file name, specified as a character vector or cell array of file names. This file is generated from USAF Digital DATCOM files.

The `datcomimport` supports only these USAF Digital DATCOM files. You can rename the output files before importing them.

Output File from DATCOM	File Type Versions
`for006.dat` by all DATCOM versions	1976, 1999, 2007, 2008, 2011, and 2014
`for021.dat` by DATCOM 2007, DATCOM 2008, DATCOM 2011, and DATCOM 2014	2007, 2008, 2011, and 2014
`for042.csv` by DATCOM 2008, DATCOM 2011, and DATCOM 2014	2008, 2011, and 2014

Example: `for006.dat`

Dependencies

The `datcomimport` function accepts DATCOM files of the type specified by the `filetype` argument. By default, the file type is 6 (`for006.dat`, output by all DATCOM versions).

Data Types: `char` | `string`

`usenan` — Replace data points
`true` (default) | `false`

While importing the DATCOM file, replace data points with NaNs (`true`) or zeroes (`false`) where no DATCOM methods exist or where methods are not applicable.

Data Types: `char` | `string`

`verbose` — Read status
`2` (default) | `0` | `1`

Read status of import of DATCOM file, specified as:

- `0` — No status information.
- `1` — Display a status information as a progress bar.
- `2` — Display status information in the MATLAB Command Window.

Data Types: `double`

`filetype` — DATCOM file type
`6` (default) | `2142`

DATCOM file type, specified as `6`, `21`, or `42`.

Depending on the file type, the `datcomimport` function expects the imported DATCOM files to contain the fields listed in the **Expected Fields** column.

`filetype`	Output File from DATCOM	File Type Versions	Expected Fields
6	`for006.dat` by all DATCOM versions	1976, 1999, 2007, 2008, 2011, and 2014	• "Fields for 1976 Version (File Type 6)" on page 4-181 • "Fields for 1999 Version (File Type 6)" on page 4-198 • "Fields for 2007, 2008, 2011, and 2014 Versions (File Type 6)" on page 4-202
21	`for021.dat` by DATCOM 2007, DATCOM 2008, DATCOM 2011, and DATCOM 2014	2007, 2008, 2011, and 2014	• "Fields for 2007, 2008, 2011, and 2014 Versions (File Type 21)" on page 4-207
42	`for042.csv` by DATCOM 2008, DATCOM 2011, and DATCOM 2014	2008, 2011, and 2014	• "Fields for 2008, 2011, and 2014 Version (File Type 42)" on page 4-212

Note If `filetype` is `21`, the function collates the breakpoints and data from all the cases and appends them as the last entry of `aero`.

Data Types: `double`

Output Arguments

`aero` — DATCOM structures
cell array of structures

DATCOM structures, returned as a cell array of structures.

Limitations

- The operational limitations of the 1976 version DATCOM apply to the data contained in AERO. For more information on DATCOM limitations, see “References” on page 4-217, section 2.4.5.
- USAF Digital DATCOM data for wing section, horizontal tail section, vertical tail section, and ventral fin section are not read.

More About

Fields for 1976 Version (File Type 6)

1976 version of file type 6 DATCOM files must contain these fields.

Common Fields for the 1976 Version (File Type 6)

Field	Description	Default
`case`	Character vector containing the case ID.	`[]`
`mach`	Array of Mach numbers.	`[]`
`alt`	Array of altitudes.	`[]`
`alpha`	Array of angles of attack.	`[]`
`nmach`	Number of Mach numbers.	`0`
`nalt`	Number of altitudes.	`0`
`nalpha`	Number of angles of attack.	`0`
`rnnub`	Array of Reynolds numbers.	`[]`
`hypers`	Logical denoting, when `true`, that `mach` numbers above `tsmach` are hypersonic. Default values are supersonic.	`false`
`loop`	Scalar denoting the type of looping done to generate the DATCOM file. When `loop` is `1`, `mach` and `alt` are varied together. When `loop` is `2`, `mach` varies while `alt` is fixed. Altitude is then updated and Mach numbers are cycled through again. When `loop` is `3`, `mach` is fixed while `alt` varies. `mach` is then updated and altitudes are cycled through again.	`1`
`sref`	Scalar denoting the reference area for the case.	`[]`
`cbar`	Scalar denoting the longitudinal reference length.	`[]`
`blref`	Scalar denoting the lateral reference length.	`[]`
`dim`	Character vector denoting the specified system of units for the case.	`'ft'`
`deriv`	Character vector denoting the specified angle units for the case.	`'deg'`

Field	Description	Default
stmach	Scalar value setting the upper limit of subsonic Mach numbers.	0.6
tsmach	Scalar value setting the lower limit of supersonic Mach numbers.	1.4
save	Logical denoting whether the input values for this case are used in the next case.	false
stype	Scalar denoting the type of asymmetric flap for the case.	[]
trim	Logical denoting the reading of trim data for the case. When trim runs are read, this value is set to true.	false
damp	Logical denoting the reading of dynamic derivative data for the case. When dynamic derivative runs are read, this value is set to true.	false
build	Scalar denoting the reading of build data for the case. When build runs are read, this value is set to 10.	1
part	Logical denoting the reading of partial data for the case. When partial runs are written for each Mach number, this value is set to true.	false
highsym	Logical denoting the reading of symmetric flap high-lift data for the case. When symmetric flap runs are read, this value is set to true.	false
highasy	Logical denoting the reading of asymmetric flap high-lift data for the case. When asymmetric flap runs are read, this value is set to true.	false
highcon	Logical denoting the reading of control/ trim tab high-lift data for the case. When control/trim tab runs are read, this value is set to true.	false

Field	Description	Default
tjet	Logical denoting the reading of transverse-jet control data for the case. When transverse-jet control runs are read, this value is set to true.	false
hypeff	Logical denoting the reading of hypersonic flap effectiveness data for the case. When hypersonic flap effectiveness runs are read, this value is set to true.	false
lb	Logical denoting the reading of low aspect ratio wing or lifting body data for the case. When low aspect ratio wing or lifting body runs are read, this value is set to true.	false
pwr	Logical denoting the reading of power effects data for the case. When power effects runs are read, this value is set to true.	false
grnd	Logical denoting the reading of ground effects data for the case. When ground effects runs are read, this value is set to true.	false
wsspn	Scalar denoting the semi-span theoretical panel for wing. This value is used to determine if the configuration contains a canard.	1
hsspn	Scalar denoting the semi-span theoretical panel for horizontal tail. This value is used to determine if the configuration contains a canard.	1
ndelta	Number of control surface deflections: delta, deltal, or deltar.	0
delta	Array of control-surface streamwise deflection angles.	[]

Field	Description	Default
`deltal`	Array of left lifting surface streamwise control deflection angles, which are defined positive for trailing-edge down.	`[]`
`deltar`	Array of right lifting surface streamwise control deflection angles, which are defined positive for trailing-edge down.	`[]`
`ngh`	Scalar denoting the number of ground altitudes.	`0`
`grndht`	Array of ground heights.	`[]`

`config`	Structure of logicals denoting whether the case contains horizontal tails.	`false`, as follows.
		
		`config.downwash = false;`
		`config.body = false;`
		`config.wing = false;`
		`config.htail = false;`
		`config.vtail = false;`
		`config.vfin = false;`

`version`	Version of DATCOM file.	`1976`

Static Longitude and Lateral Stability Fields Available for the 1976 Version (File Type 6)

Field	Matrix of...	Function of...
`cd`	Drag coefficients, which are defined positive for an aft-acting load.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`cl`	Lift coefficients, which are defined positive for an up-acting load.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`cm`	Pitching-moment coefficients, which are defined positive for a nose-up rotation.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`cn`	Normal-force coefficients, which are defined positive for a normal force in the +Z direction.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`ca`	Axial-force coefficients, which are defined positive for a normal force in the +X direction.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`xcp`	Distances between moment reference center and the center of pressure divided by the longitudinal reference length. Distances are defined positive for a location forward of the center of gravity.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`cla`	Derivatives of lift coefficients relative to `alpha`.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`cma`	Derivatives of pitching-moment coefficients relative to `alpha`.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`cyb`	Derivatives of side-force coefficients relative to sideslip angle.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`cnb`	Derivatives of yawing-moment coefficients relative to sideslip angle.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`clb`	Derivatives of rolling-moment coefficients relative to sideslip angle.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`qqinf`	Ratios of dynamic pressure at the horizontal tail to the freestream value.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`
`eps`	Downwash angle at horizontal tail in degrees.	`alpha`, `mach`, `alt`, `build`, `grndht`, `delta`

Field	Matrix of...	Function of...
depsdalp	Downwash angle relative to angle of attack.	alpha, mach, alt, build, grndht, delta

Dynamic Derivative Fields for the 1976 Version (File Type 6)

Field	Matrix of...	Function of...
clq	Lift force derivatives due to pitch rate.	alpha, mach, alt, build
cmq	Pitching-moment derivatives due to pitch rate.	alpha, mach, alt, build
clad	Lift-force derivatives due to rate of angle of attack.	alpha, mach, alt, build
cmad	Pitching-moment derivatives due to rate of angle of attack.	alpha, mach, alt, build
clp	Rolling-moment derivatives due to roll rate.	alpha, mach, alt, build
cyp	Lateral-force derivatives due to roll rate.	alpha, mach, alt, build
cnp	Yawing-moment derivatives due to roll rate.	alpha, mach, alt, build
cnr	Yawing-moment derivatives due to yaw rate.	alpha, mach, alt, build
clr	Rolling-moment derivatives due to yaw rate.	alpha, mach, alt, build

High-Lift and Control Fields for Symmetric Flaps for the 1976 Version (File Type 6)

Field	Matrix of...	Function of...
`dcl_sym`	Incremental lift coefficients due to deflection of control surface, valid in the linear-lift angle of attack range.	`delta`, `mach`, `alt`
`dcm_sym`	Incremental pitching-moment coefficients due to deflection of control surface, valid in the linear-lift angle of attack range.	`delta`, `mach`, `alt`
`dclmax_sym`	Incremental maximum lift coefficients.	`delta`, `mach`, `alt`
`dcdmin_sym`	Incremental minimum drag coefficients due to control or flap deflection.	`delta`, `mach`, `alt`
`clad_sym`	Lift-curve slope of the deflected, translated surface.	`delta`, `mach`, `alt`
`cha_sym`	Control-surface hinge-moment derivatives due to angle of attack. These derivatives, when defined positive, tend to rotate the flap trailing edge down.	`delta`, `mach`, `alt`
`chd_sym`	Control-surface hinge-moment derivatives due to control deflection. When defined positive, these derivatives tend to rotate the flap trailing edge down.	`delta`, `mach`, `alt`
`dcdi_sym`	Incremental induced drag coefficients due to flap detection.	`alpha`, `delta`, `mach`, `alt`

High-Lift and Control Fields Available for Asymmetric Flaps for the 1976 Version (File Type 6)

Field	Matrix of...	Function of...
`xsc`	Streamwise distances from wing leading edge to spoiler tip.	`delta`, `mach`, `alt`
`hsc`	Projected height of spoiler measured from normal to airfoil meanline.	`delta`, `mach`, `alt`
`ddc`	Projected height of deflector for spoiler-slot-deflector control.	`delta`, `mach`, `alt`
`dsc`	Projected height of spoiler control.	`delta`, `mach`, `alt`
`clroll`	Incremental rolling-moment coefficients due to asymmetrical deflection of control surface. The coefficients are defined positive when right wing is down.	`delta`, `mach`, and `alt`, or `alpha`, `delta`, `mach`, and `alt` for differential horizontal stabilizer
`cn_asy`	Incremental yawing-moment coefficients due to asymmetrical deflection of control surface. The coefficients are defined positive when nose is right.	`delta`, `mach`, and `alt`, or `alpha`, `delta`, `mach`, and `alt` for plain flaps

High-Lift and Control Fields Available for Control/Trim Tabs for the 1976 Version (File Type 6)

Field	Matrix of...	Function of...
`fc_con`	Stick forces or stick force coefficients.	`alpha`, `delta`, `mach`, `alt`
`fhmcoeff_free`	Flap-hinge moment coefficients tab free.	`alpha`, `delta`, `mach`, `alt`
`fhmcoeff_lock`	Flap-hinge moment coefficients tab locked.	`alpha`, `delta`, `mach`, `alt`
`fhmcoeff_gear`	Flap-hinge moment coefficients due to gearing.	`alpha`, `delta`, `mach`, `alt`
`ttab_def`	Trim-tab deflections for zero stick force.	`alpha`, `delta`, `mach`, `alt`

High-Lift and Control Fields Available for Trim for the 1976 Version (File Type 6)

Field	Matrix of...	Function of...
`cl_utrim`	Untrimmed lift coefficients, which are defined positive for an up-acting load.	`alpha`, `mach`, `alt`
`cd_utrim`	Untrimmed drag coefficients, which are defined positive for an aft-acting load.	`alpha`, `mach`, `alt`
`cm_utrim`	Untrimmed pitching-moment coefficients, which are defined positive for a nose-up rotation.	`alpha`, `mach`, `alt`
`delt_trim`	Trimmed control-surface streamwise deflection angles.	`alpha`, `mach`, `alt`
`dcl_trim`	Trimmed incremental lift coefficients in the linear-lift angle of attack range due to deflection of control surface.	`alpha`, `mach`, `alt`
`dclmax_trim`	Trimmed incremental maximum lift coefficients.	`alpha`, `mach`, `alt`
`dcdi_trim`	Trimmed incremental induced drag coefficients due to flap deflection.	`alpha`, `mach`, `alt`
`dcdmin_trim`	Trimmed incremental minimum drag coefficients due to control or flap deflection.	`alpha`, `mach`, `alt`
`cha_trim`	Trimmed control-surface hinge-moment derivatives due to angle of attack.	`alpha`, `mach`, `alt`
`chd_trim`	Trimmed control-surface hinge-moment derivatives due to control deflection.	`alpha`, `mach`, `alt`
`cl_tailutrim`	Untrimmed stabilizer lift coefficients, which are defined positive for an up-acting load.	`alpha`, `mach`, `alt`
`cd_tailutrim`	Untrimmed stabilizer drag coefficients, which are defined positive for an aft-acting load.	`alpha`, `mach`, `alt`
`cm_tailutrim`	Untrimmed stabilizer pitching-moment coefficients, which are defined positive for a nose-up rotation.	`alpha`, `mach`, `alt`
`hm_tailutrim`	Untrimmed stabilizer hinge-moment coefficients, which are defined positive for a stabilizer rotation with leading edge up and trailing edge down.	`alpha`, `mach`, `alt`

Field	Matrix of...	Function of...
aliht_tailtrim	Stabilizer incidence required to trim.	alpha, mach, alt
cl_tailtrim	Trimmed stabilizer lift coefficients, which are defined positive for an up-acting load.	alpha, mach, alt
cd_tailtrim	Trimmed stabilizer drag coefficients, which are defined positive for an aft-acting load.	alpha, mach, alt
cm_tailtrim	Trimmed stabilizer pitching-moment coefficients, which are defined positive for a nose-up rotation.	alpha, mach, alt
hm_tailtrim	Trimmed stabilizer hinge-moment coefficients, which are defined positive for a stabilizer rotation with leading edge up and trailing edge down.	alpha, mach, alt
cl_trimi	Lift coefficients at trim incidence. These coefficients are defined positive for an up-acting load.	alpha, mach, alt
cd_trimi	Drag coefficients at trim incidence. These coefficients are defined positive for an aft-acting load.	alpha, mach, alt

Transverse Jet Control Fields for the 1976 Version (File Type 6)

Field	Description	Stored with Indices of...
`time`	Matrix of times.	`mach`, `alt`, `alpha`
`ctrlfrc`	Matrix of control forces.	`mach`, `alt`, `alpha`
`locmach`	Matrix of local Mach numbers.	`mach`, `alt`, `alpha`
`reynum`	Matrix of Reynolds numbers.	`mach`, `alt`, `alpha`
`locpres`	Matrix of local pressures.	`mach`, `alt`, `alpha`
`dynpres`	Matrix of dynamic pressures.	`mach`, `alt`, `alpha`
`blayer`	Cell array of character vectors containing the state of the boundary layer.	`mach`, `alt`, `alpha`
`ctrlcoeff`	Matrix of control force coefficients.	`mach`, `alt`, `alpha`
`corrcoeff`	Matrix of corrected force coefficients.	`mach`, `alt`, `alpha`
`sonicamp`	Matrix of sonic amplification factors.	`mach`, `alt`, `alpha`
`ampfact`	Matrix of amplification factors.	`mach`, `alt`, `alpha`
`vacthr`	Matrix of vacuum thrusts.	`mach`, `alt`, `alpha`
`minpres`	Matrix of minimum pressure ratios.	`mach`, `alt`, `alpha`
`minjet`	Matrix of minimum jet pressures.	`mach`, `alt`, `alpha`
`jetpres`	Matrix of jet pressures.	`mach`, `alt`, `alpha`
`massflow`	Matrix of mass flow rates.	`mach`, `alt`, `alpha`
`propelwt`	Matrix of propellant weights.	`mach`, `alt`, `alpha`

Hypersonic Fields for the 1976 Version (File Type 6)

Field	Matrix of...	Stored with Indices of...
`df_normal`	Increments in normal force per spanwise foot of control.	`alpha`, `delta`, `mach`
`df_axial`	Increments in axial force per spanwise foot of control.	`alpha`, `delta`, `mach`
`cm_normal`	Increments in pitching moment due to normal force per spanwise foot of control.	`alpha`, `delta`, `mach`
`cm_axial`	Increments in pitching moment due to axial force per spanwise foot of control.	`alpha`, `delta`, `mach`
`cp_normal`	Center of pressure locations of normal force.	`alpha`, `delta`, `mach`
`cp_axial`	Center of pressure locations of axial force.	`alpha`, `delta`, `mach`

Auxiliary and Partial Fields Available for the 1976 Version (File Type 6)

Field	Matrix of...	Stored with Indices of...
wetarea_b	Body wetted area.	mach, alt, number of runs
xcg_b	Longitudinal locations of the center of gravity.	mach, alt, number of runs (normally 1, 2 for hypers = true)
zcg_b	Vertical locations of the center of gravity.	mach, alt, number of runs (normally 1, 2 for hypers = true)
basearea_b	Body base area.	mach, alt, number of runs (normally 1, 2 for hypers = true)
cd0_b	Body zero lift drags.	mach, alt, number of runs (normally 1, 2 for hypers = true)
basedrag_b	Body base drags.	mach, alt, number of runs (normally 1, 2 for hypers = true)
fricdrag_b	Body friction drags.	mach, alt, number of runs (normally 1, 2 for hypers = true)
presdrag_b	Body pressure drags.	mach, alt, number of runs (normally 1, 2 for hypers = true)
lemac	Leading edge mean aerodynamic chords.	mach, alt
sidewash	sidewash	mach, alt
hiv_b_w	iv-b(w)	alpha, mach, alt
hiv_w_h	iv-w(h)	alpha, mach, alt
hiv_b_h	iv-b(h)	alpha, mach, alt
gamma	gamma*2*pi*alpha*v*r	alpha, mach, alt
gamma2pialpvr	gamma*(2*pi*alpha*v*r)t	alpha, mach, alt
clpgammacl0	clp(gamma=cl=0)	mach, alt
clpgammaclp	clp(gamma)/cl (gamma=0)	mach, alt
cnptheta	cnp/theta	mach, alt
cypgamma	cyp/gamma	mach, alt
cypcl	cyp/cl (cl=0)	mach, alt
clbgamma	clb/gamma	mach, alt

Field	Matrix of...	Stored with Indices of...
cmothetaw	(cmo/theta)w	mach, alt
cmothetah	(cmo/theta)h	mach, alt
espeff	(epsoln)eff	alpha, mach, and alt
despdalpeff	d(epsoln)/d(alpha) eff	alpha, mach, alt
dragdiv	drag divergence mach number	mach, alt
cd0mach	Four Mach numbers for the zero lift drag.	index, mach, alt
cd0	Four zero lift drags.	index, mach, alt
clbclmfb_****	(clb/cl)mfb, where **** is either wb (wing-body) or bht (body-horizontal tail).	mach, alt.
cnam14_****	(cna)m=1.4, where **** is either wb (wing-body) or bht (body-horizontal tail).	mach,alt
area_*_**	Areas, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
taperratio_*_**	Taper ratios, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
aspectratio_*_**	Aspect ratios, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)

Field	Matrix of...	Stored with Indices of...
qcsweep_*_**	Quarter chord sweeps, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
mac_*_**	Mean aerodynamic chords, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
qcmac_*_**	Quarter chord x(mac), where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
ymac_*_**	y(mac), where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cd0_*_**	Zero lift drags, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)

Field	Matrix of...	Stored with Indices of...
friccoeff_*_**	Friction coefficients, where * is either w (wing), ht (horizontal tail), vt (vertical tail), or vf (ventral fin) and ** is either tt (total theoretical), ti (theoretical inboard), te (total exposed), ei (exposed inboard), or o (outboard).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cla_b_***	cla-b(***), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cla_***_b	cla-***(b), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
k_b_***	k-b(***), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
k_***_b	k-***(b), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
xacc_b_***	xac/c-b(***), where *** is either w (wing) or ht (stabilizer).	mach, alt, number of runs (normally 1, 2 for hypers = true)
cdlcl2_***	cdl/cl^2, where *** is either w (wing) or ht (stabilizer).	mach, alt
clbcl_***	clb/cl, where *** is either w (wing) or ht (stabilizer).	mach, alt
fmach0_***	Force break Mach numbers with zero sweep, where *** is either w (wing) or ht (stabilizer).	mach, alt
fmach_***	Force break Mach numbers with sweep, where *** is either w (wing) or ht (stabilizer).	mach, alt
macha_***	mach(a), where *** is either w (wing) or ht (stabilizer).	mach, alt
machb_***	mach(b), where *** is either w (wing) or ht (stabilizer).	mach, alt
claa_***	cla(a), where *** is either w (wing) or ht (stabilizer).	mach, alt

Field	Matrix of...	Stored with Indices of...
clab_***	cla(b), where *** is either w (wing) or ht (stabilizer).	mach, alt
clbm06_***	(clb/cl)m=0.6, where *** is either w (wing) or ht (stabilizer).	mach, alt
clbm14_***	(clb/cl)m=1.4, where *** is either w (wing) or ht (stabilizer).	mach, alt
clalpmach_***	Five Mach numbers for the lift curve slope, where *** is either w (wing) or ht (stabilizer).	index, mach, alt
clalp_***	Five lift-curve slope values, where *** is either w (wing) or ht (stabilizer).	index, mach, alt

Fields for 1999 Version (File Type 6)

1999 version of file type 6 DATCOM files must contain these fields.

Common Fields for the 1999 Version (File Type 6)

Field	Description	Default
`case`	Character vector containing the case ID.	`[]`
`mach`	Array of Mach numbers.	`[]`
`alt`	Array of altitudes.	`[]`
`alpha`	Array of angles of attack.	`[]`
`nmach`	Number of Mach numbers.	`0`
`nalt`	Number of altitudes.	`1`
`nalpha`	Number of angles of attack.	`0`
`rnnub`	Array of Reynolds numbers.	`[]`
`beta`	Scalar containing sideslip angle.	`0`
`phi`	Scalar containing aerodynamic roll angle.	`0`
`loop`	Scalar denoting the type of looping performed to generate the DATCOM file. When `loop` is `1`, `mach` and `alt` are varied together. The only loop option for the 1999 version of DATCOM is `loop` is equal to `1`.	`1`
`sref`	Scalar denoting the reference area for the case.	`[]`
`cbar`	Scalar denoting the longitudinal reference length.	`[]`
`blref`	Scalar denoting the lateral reference length.	`[]`
`dim`	Character vector denoting the specified system of units for the case.	`'ft'`
`deriv`	Character vector denoting the specified angle units for the case.	`'deg'`
`save`	Logical denoting whether the input values for this case are used in the next case.	`false`
`stype`	Scalar denoting the type of asymmetric flap for the case.	`[]`

Field	Description	Default
`trim`	Logical denoting the reading of trim data for the case. When trim runs are read, this value is set to `true`.	`false`
`damp`	Logical denoting the reading of dynamic derivative data for the case. When dynamic derivative runs are read, this value is set to `true`.	`false`
`build`	Scalar denoting the reading of build data for the case. When build runs are read, this value is set to the number of build runs depending on the vehicle configuration.	`1`
`part`	Logical denoting the reading of partial data for the case. When partial runs are written for each Mach number, this value is set to `true`.	`false`
`hypeff`	Logical denoting the reading of hypersonic data for the case. When hypersonic data is read, this value is set to `true`.	`false`
`ngh`	Scalar denoting the number of ground altitudes.	`0`
`nolat`	Logical denoting the calculation of the lateral-direction derivatives is inhibited.	`false`

`config`	Structure of logicals and structures detailing the case configuration and fin deflections.	`config.body = false`
		`config.fin1.avail = false;`
		`config.fin1.npanel = [];`
		`config.fin1.delta = [];`
		`config.fin2.avail = false;`
		`config.fin2.npanel = [];`
		`config.fin2.delta = [];`
		`config.fin3.avail = false;`
		`config.fin3.npanel = [];`
		`config.fin3.delta = [];`
		`config.fin4.avail = false;`
		`config.fin4.npanel = [];`
		`config.fin4.delta = [];`

`version`	Version of DATCOM file.	`1999`

Static Longitude and Lateral Stability Fields Available for the 1999 Version (File Type 6)

Field	Matrix of...	Function of...
cd	Drag coefficients, which are defined positive for an aft-acting load.	alpha, mach, alt, build
cl	Lift coefficients, which are defined positive for an up-acting load.	alpha, mach, alt, build
cm	Pitching-moment coefficients, which are defined positive for a nose-up rotation.	alpha, machalt, build
cn	Normal-force coefficients, which are defined positive for a normal force in the +Z direction.	alpha, mach, alt, build
ca	Axial-force coefficients, which are defined positive for a normal force in the +X direction.	alpha, mach, alt, build
xcp	Distances between moment reference center and the center of pressure divided by the longitudinal reference length. These distances are defined positive for a location forward of the center of gravity.	alpha, mach, alt, build
cna	Derivatives of normal-force coefficients relative to alpha.	alpha, mach, alt, build
cma	Derivatives of pitching-moment coefficients relative to alpha.	alpha, mach, alt, build
cyb	Derivatives of side-force coefficients relative to sideslip angle.	alpha, mach, alt, build
cnb	Derivatives of yawing-moment coefficients relative to sideslip angle.	alpha, mach, alt, build
clb	Derivatives of rolling-moment coefficients relative to sideslip angle.	alpha, mach, alt, build
clod	Ratios of lift coefficient to drag coefficient.	alpha, mach, alt, build
cy	Side-force coefficients.	alpha, mach, alt, build
cln	Yawing-moment coefficient in body-axis.	alpha, mach, alt, build
cll	Rolling-moment coefficient in body-axis.	alpha, mach, alt, build

Dynamic Derivative Fields for the 1999 Version (File Type 6)

Field	Matrix of...	Function of...
`cnq`	Normal-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cmq`	Pitching-moment derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`caq`	Axial-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cnad`	Normal-force derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `build`
`cmad`	Pitching-moment derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `build`
`clp`	Rolling-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cyp`	Lateral force derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cnp`	Yawing-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cnr`	Yawing-moment derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`
`clr`	Rolling-moment derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`
`cyr`	Side force derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`

Fields for 2007, 2008, 2011, and 2014 Versions (File Type 6)

2007, 2008, 2011, and 2014 versions of file type 6 DATCOM files must contain these fields.

Common Fields for the 2007, 2008, 2011, and 2014 Versions (File Type 6)

Field	Description	Default
`case`	Character vector containing the case ID.	`[]`
`mach`	Array of Mach numbers.	`[]`
`alt`	Array of altitudes.	`[]`
`alpha`	Array of angles of attack.	`[]`
`nmach`	Number of Mach numbers.	`0`
`nalt`	Number of altitudes.	`1`
`nalpha`	Number of angles of attack.	`0`
`rnnub`	Array of Reynolds numbers.	`[]`
`beta`	Scalar containing sideslip angle. **Note** This value does not appear correctly for the 2014 version. It always displays 0.	`0`
`phi`	Scalar containing aerodynamic roll angle.	`0`
`loop`	Scalar denoting the type of looping performed to generate the DATCOM file. When `loop` is `1`, `mach` and `alt` are varied together. The only loop option for the 2007 version of DATCOM is `loop`, equal to `1`.	`1`
`sref`	Scalar denoting the reference area for the case.	`[]`
`cbar`	Scalar denoting the longitudinal reference length.	`[]`
`blref`	Scalar denoting the lateral reference length.	`[]`
`dim`	Character vector denoting the specified system of units for the case.	`'ft'`
`deriv`	Character vector denoting the specified angle units for the case.	`'deg'`
`save`	Logical denoting whether the input values for this case are used in the next case.	`false`

Field	Description	Default
`stype`	Scalar denoting the type of asymmetric flap for the case.	`[]`
`trim`	Logical denoting the reading of trim data for the case. When trim runs are read, this value is set to `true`.	`false`
`damp`	Logical denoting the reading of dynamic derivative data for the case. When dynamic derivative runs are read, this value is set to `true`.	`false`
`build`	Scalar denoting the reading of build data for the case. When build runs are read, this value is set to the number of build runs depending on the vehicle configuration.	`1`
`part`	Logical denoting the reading of partial data for the case. When partial runs are written for each Mach number, this value is set to `true`.	`false`
`hypeff`	Logical denoting the reading of hypersonic data for the case. When hypersonic data is read, this value is set to `true`.	`false`
`ngh`	Scalar denoting the number of ground altitudes.	`0`
`nolat`	Logical denoting the calculation of the lateral-direction derivatives is inhibited.	`false`

Field	Description	Default

`config`	Structure of logicals and structures detailing the case configuration and fin deflections.	`config.body = false;`
		`config.fin1.avail = false;`
		`config.fin1.npanel = [];`
		`config.fin1.delta = [];`
		`config.fin2.avail = false;`
		`config.fin2.npanel = [];`
		`config.fin2.delta = [];`
		`config.fin3.avail = false;`
		`config.fin3.npanel = [];`
		`config.fin3.delta = [];`
		`config.fin4.avail = false;`
		`config.fin4.npanel = [];`
		`config.fin4.delta = [];`

`nolat_-namelist`	Logical denoting the calculation of the lateral-direction derivatives is inhibited in the DATCOM input case.	`false`
`version`	Version of DATCOM file.	`2007`

Static Longitude and Lateral Stability Fields Available for the 2007, 2008, 2011, and 2014 Versions (File Type 6)

Field	Matrix of...	Function of...
`cd`	Drag coefficients, which are defined positive for an aft-acting load.	`alpha`, `mach`, `alt`, `build`
`cl`	Lift coefficients, which are defined positive for an up-acting load.	`alpha`, `mach`, `alt`, `build`
`cm`	Pitching-moment coefficients, which are defined positive for a nose-up rotation.	`alpha`, `machalt`, `build`
`cn`	Normal-force coefficients, which are defined positive for a normal force in the +Z direction.	`alpha`, `mach`, `alt`, `build`
`ca`	Axial-force coefficients, which are defined positive for a normal force in the +X direction.	`alpha`, `mach`, `alt`, `build`
`xcp`	Distances between moment reference center and the center of pressure divided by the longitudinal reference length. These distances are defined positive for a location forward of the center of gravity.	`alpha`, `mach`, `alt`, `build`
`cna`	Derivatives of normal-force coefficients relative to `alpha`.	`alpha`, `mach`, `alt`, `build`
`cma`	Derivatives of pitching-moment coefficients relative to `alpha`.	`alpha`, `mach`, `alt`, `build`
`cyb`	Derivatives of side-force coefficients relative to sideslip angle.	`alpha`, `mach`, `alt`, `build`
`cnb`	Derivatives of yawing-moment coefficients relative to sideslip angle.	`alpha`, `mach`, `alt`, `build`
`clb`	Derivatives of rolling-moment coefficients relative to sideslip angle.	`alpha`, `mach`, `alt`, `build`
`clod`	Ratios of lift coefficient to drag coefficient.	`alpha`, `mach`, `alt`, `build`
`cy`	Side-force coefficients.	`alpha`, `mach`, `alt`, `build`
`cln`	Yawing-moment coefficient in body-axis.	`alpha`, `mach`, `alt`, `build`
`cll`	Rolling-moment coefficient in body-axis.	`alpha`, `mach`, `alt`, `build`

Dynamic Derivative Fields for the 2007, 2008, 2011, and 2014 Versions (File Type 6)

Field	Matrix of...	Function of...
`cnq`	Normal-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cmq`	Pitching-moment derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`caq`	Axial-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cnad`	Normal-force derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `build`
`cmad`	Pitching-moment derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `build`
`clp`	Rolling-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cyp`	Lateral-force derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cnp`	Yawing-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cnr`	Yawing-moment derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`
`clr`	Rolling-moment derivatives due to yaw rate	`alpha`, `mach`, `alt`, `build`
`cyr`	Side-force derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`

Fields for 2007, 2008, 2011, and 2014 Versions (File Type 21)

2007, 2008, 2011, and 2014 versions of file type 21 DATCOM files must contain these fields.

Common Fields for the 2007, 2008, 2011, and 2014 Versions (File Type 21)

Field	Description	Default
`mach`	Array of Mach numbers.	`[]`
`alt`	Array of altitudes.	`[]`
`alpha`	Array of angles of attack.	`[]`
`nalpha`	Number of angles of attack.	`0`
`beta`	Scalar containing sideslip angle. **Note** This value does not appear correctly for the 2014 version. It always displays 0.	`0`
`total_col`	Scalar denoting the type of looping performed to generate the DATCOM file. When `loop` is 1, `mach` and `alt` are varied together. The only loop option for the 2007, 2008, 2011, and 2014 versions of DATCOM is `loop` equal to `1`.	`[]`
`deriv_col`	Logical denoting the calculation of the lateral-direction derivatives is inhibited.	`0`
`config`	Structure of logicals and structures detailing the case configuration and fin deflections.	`config.fin1.delta = zeros(1,8);` `config.fin2.delta = zeros(1,8);` `config.fin3.delta = zeros(1,8);` `config.fin4.delta = zeros(1,8);`
`version`	Version of DATCOM file.	`2007`

Static Longitude and Lateral Stability Fields Available for the 2007, 2008, 2011, and 2014 Versions (File Type 21)

Field	Matrix of...	Function of...
`cn`	Normal-force coefficients, which are defined positive for a normal force in the +Z direction.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cm`	Pitching-moment coefficients, which are defined positive for a nose-up rotation.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`ca`	Axial-force coefficients, which are defined positive for a normal force in the +X direction.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cy`	Side-force coefficients.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cln`	Yawing-moment coefficient in body-axis.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cll`	Rolling-moment coefficient in body-axis.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`

Dynamic Derivative Fields for the 2007, 2008, 2011, and 2014 Versions (File Type 21)

Field	Matrix of...	Function of...
`cnad`	Normal-force derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cmad`	Pitching-moment derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cnq`	Normal-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cmq`	Pitching-moment derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`caq`	Axial-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cyq`	Side-force due to pitch rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`

Field	Matrix of...	Function of...
clnq	Yawing-moment due to pitch rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cllq	Rolling-moment due to pitch rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cnp	Yawing-moment derivatives due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cap	Axial-force due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cyp	Lateral-force derivatives due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
clnp	Yawing-moment due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta
cllp	Rolling-moment due to roll rate.	alpha, mach, alt, beta, config.fin1.delta, config.fin2.delta, config.fin3.delta, config.fin4.delta

Field	Matrix of...	Function of...
`cnr`	Yawing-moment derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`car`	Axial-force due to yaw rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cyr`	Side-force derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`clnr`	Yawing-moment due to yaw rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`
`cllr`	Rolling-moment due to yaw rate.	`alpha`, `mach`, `alt`, `beta`, `config.fin1.delta`, `config.fin2.delta`, `config.fin3.delta`, `config.fin4.delta`

Fields for 2008, 2011, and 2014 Version (File Type 42)

2008, 2011, and 2014 versions of file type 42 DATCOM files must contain these fields.

Fields for the 2008, 2011, and 2014 Version (File Type 42)

Field	Description	Default
`case`	Character vector containing the case ID.	`[]`
`totalCol`	Scalar containing number of columns of data in file.	`[]`
`mach`	Array of Mach numbers.	`[]`
`alt`	Array of altitudes.	`[]`
`alpha`	Array of angles of attack.	`[]`
`nmach`	Number of Mach numbers.	`0`
`nalpha`	Number of angles of attack.	`0`
`rnnub`	Array of Reynolds numbers.	`[]`
`q`	Dynamic pressure.	`[]`
`beta`	Scalar containing sideslip angle. **Note** This value does not appear correctly for the 2014 version. It always displays 0.	`0`
`phi`	Scalar containing aerodynamic roll angle.	`0`
`sref`	Scalar denoting the reference area for the case.	`[]`
`cbar`	Scalar denoting the longitudinal reference length.	`[]`
`blref`	Scalar denoting the lateral reference length.	`[]`
`xcg`	Distance from nose to center of gravity.	`[]`
`xmrp`	Distance from nose to center of gravity, measured in calibers.	`[]`
`deriv`	Character vector denoting the specified angle units for the case.	`'deg'`

Field	Description	Default
`trim`	Logical denoting the reading of trim data for the case. When trim runs are read, this value is set to `true`.	`false`
`damp`	Logical denoting the reading of dynamic derivative data for the case. When dynamic derivative runs are read, this value is set to `true`.	`false`
`build`	Scalar denoting the reading of partial data for the case. This value is set to the number of partial runs depending on the vehicle configuration.	`1`
`part`	Logical denoting the reading of partial data for the case. When partial runs are written for each Mach number, this value is set to `true`.	`false`
`nolat`	Logical denoting the calculation of the lateral-direction derivatives is inhibited.	`true`

`config`	Structure of logicals and structures detailing the case configuration and fin deflections.	`config.body = false;`
		`config.fin1.avail = false;`
		`config.fin1.npanel = [];`
		`config.fin1.delta = [];`
		`config.fin2.avail = false;`
		`config.fin2.npanel = [];`
		`config.fin2.delta = [];`
		`config.fin3.avail = false;`
		`config.fin3.npanel = [];`
		`config.fin3.delta = [];`
		`config.fin4.avail = false;`
		`config.fin4.npanel = [];`

`version`	Version of DATCOM file.	`2008`

Static Longitude and Lateral Stability Fields Available for the 2008, 2011, and 2014 Versions (File Type 42)

Field	Matrix of...	Function of...
`delta`	Trim deflection angles.	`alpha`, `mach`
`cd`	Drag coefficients, which are defined positive for an aft-acting load.	`alpha`, `mach`, `build`
`cl`	Lift coefficients, which are defined positive for an up-acting load.	`alpha`, `mach`, `build`
`cm`	Pitching-moment coefficients, which are defined positive for a nose-up rotation.	`alpha`, `mach`, `build`
`cn`	Normal-force coefficients, which are defined positive for a normal force in the +Z direction.	`alpha`, `mach`, `build`
`ca`	Axial-force coefficients, which are defined positive for a normal force in the +X direction.	`alpha`, `mach`, `build`
`caZeroBase`	Axial-force coefficient with no base drag included.	`alpha`, `mach`, `build`
`caFullBase`	Axial-force coefficient with full base drag included.	`alpha`, `mach`, `build`
`xcp`	Distance from nose to center of pressure.	`alpha`, `mach`, `build`
`cna`	Derivatives of normal-force coefficients relative to `alpha`.	`alpha`, `mach`, `build`
`cma`	Derivatives of pitching-moment coefficients relative to `alpha`.	`alpha`, `mach`, `build`
`cyb`	Derivatives of side-force coefficients relative to sideslip angle.	`alpha`, `mach`, `build`
`cnb`	Derivatives of yawing-moment coefficients relative to sideslip angle.	`alpha`, `mach`, `build`
`clb`	Derivatives of rolling-moment coefficients relative to sideslip angle.	`alpha`, `mach`, `build`
`clod`	Ratios of lift coefficient to drag coefficient.	`alpha`, `mach`, `build`

Field	Matrix of...	Function of...
`cy`	Side-force coefficient.	`alpha`, `mach`, `build`
`cln`	Yawing-moment coefficient.	`alpha`, `mach`, `build`
`cll`	Rolling-moment coefficient.	`alpha`, `mach`, `build`

Dynamic Derivative Fields for the 2008, 2011, and 2014 Version (File Type 42)

Field	Matrix of...	Function of...
`cnq`	Normal-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cmq`	Pitching-moment derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`caq`	Axial-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cnad`	Normal-force derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `build`
`cmad`	Pitching-moment derivatives due to rate of angle of attack.	`alpha`, `mach`, `alt`, `build`
`cyq`	Lateral-force derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`clnq`	Yawing-moment derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cllq`	Rolling-moment derivatives due to pitch rate.	`alpha`, `mach`, `alt`, `build`
`cyr`	Side-force derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`
`clnr`	Yawing-moment derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`
`cllr`	Rolling-moment derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`
`cyp`	Lateral-force derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`clnp`	Yawing-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cllp`	Rolling-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cnp`	Normal-force derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cmp`	Pitching-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cap`	Axial-force derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`cnr`	Normal-force derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`
`cmr`	Pitching-moment derivatives due to roll rate.	`alpha`, `mach`, `alt`, `build`
`car`	Axial-force derivatives due to yaw rate.	`alpha`, `mach`, `alt`, `build`

References

[1] AFFDL-TR-79-3032: *The USAF Stability and Control DATCOM*, Volume 1, User's Manual

[2] AFRL-VA-WP-TR-1998-3009: *MISSILE DATCOM*, User's Manual – 1997 FORTRAN 90 Revision

[3] AFRL-RB-WP-TR-2009-3015: *MISSILE DATCOM*, User's Manual – 2008 Revision

[4] AFRL-RB-WP-TR-2011-3071: *MISSILE DATCOM*, User's Manual – 2011 Revision

[5] AFRL-RQ-WP-TR-2014-3999: *MISSILE DATCOM*, Users Manual – 2014 Revision

See Also

Topics
“Importing from USAF Digital DATCOM Files” on page 5-2

Introduced in R2006b

dcm2alphabeta

Convert direction cosine matrix to angle of attack and sideslip angle

Syntax

```
[a b] = dcm2alphabeta(n)
[a b] = dcm2alplhabeta(n,action)
[a b] = dcm2angle(n,action,tolerance)
```

Description

`[a b] = dcm2alphabeta(n)` calculates the angle of attack and sideslip angle, `a` and `b`, for a given direction cosine matrix, `n`. `n` is a 3-by-3-by-`m` matrix containing `m` orthogonal direction cosine matrices. `a` is an `m` array of angles of attack. `b` is an `m` array of sideslip angles. `n` performs the coordinate transformation of a vector in body-axes into a vector in wind-axes. Angles of attack and sideslip angles are output in radians.

`[a b] = dcm2alplhabeta(n,action)` performs `action` if the direction cosine matrix is invalid (not orthogonal).

- Warning — Displays warning and indicates that the direction cosine matrix is .
- Error — Displays error and indicates that the direction cosine matrix is .
- None — Does not display warning or error (default).

`[a b] = dcm2angle(n,action,tolerance)` uses a `tolerance` level to evaluate if the direction cosine matrix, `n`, is valid (orthogonal). `tolerance` is a scalar whose default is `eps(2)` (`4.4409e-16`). The function considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`)
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Examples

Determine the angle of attack and sideslip angle from direction cosine matrix:

```
dcm = [ 0.8926    0.1736    0.4162; ...
       -0.1574    0.9848   -0.0734; ...
       -0.4226         0    0.9063];
[alpha, beta] = dcm2alphabeta(dcm)

alpha =

    0.4363


beta =

    0.1745
```

Determine the angle of attack and sideslip angle from multiple direction cosine matrices:

```
dcm = [ 0.8926    0.1736    0.4162; ...
       -0.1574    0.9848   -0.0734; ...
       -0.4226         0    0.9063];
dcm(:,:,2) = [ 0.9811    0.0872    0.1730; ...
              -0.0859    0.9962   -0.0151; ...
              -0.1736         0    0.9848];
[alpha, beta] = dcm2alphabeta(dcm)


alpha =

    0.4363
    0.1745


beta =

    0.1745
    0.0873
```

Determine the angle of attack and sideslip angle from multiple direction cosine matrices validated within tolerance:

```
dcm = [ 0.8926    0.1736    0.4162; ...
       -0.1574    0.9848   -0.0734; ...
       -0.4226         0    0.9063];
dcm(:,:,2) = [ 0.9811    0.0872    0.1730; ...
              -0.0859    0.9962   -0.0151; ...
              -0.1736         0    0.9848];
[alpha, beta] = dcm2alphabeta(dcm,'Warning',0.1)

alpha =

    0.4363
    0.1745


beta =

    0.1745
    0.0873
```

See Also

angle2dcm | dcm2angle | dcmbody2wind

Introduced in R2006b

dcm2angle

Create rotation angles from direction cosine matrix

Syntax

```
[r1 r2 r3] = dcm2angle(n)
[r1 r2 r3] = dcm2angle(n,s)
[r1 r2 r3] = dcm2angle(n,s,lim)
[r1 r2 r3] = dcm2angle(n,s,lim,action)
[r1 r2 r3] = dcm2angle(n,s,lim,action,tolerance)
```

Description

`[r1 r2 r3] = dcm2angle(n)` calculates the set of rotation angles, `r1`, `r2`, `r3`, for a given direction cosine matrix, `n`. `n` is a 3-by-3-by-`m` matrix containing `m` direction cosine matrices. `r1` returns an `m` array of first rotation angles. `r2` returns an `m` array of second rotation angles. `r3` returns an `m` array of third rotation angles. Rotation angles are output in radians. This function applies only to direction cosine matrices that are orthogonal with determinant +1.

`[r1 r2 r3] = dcm2angle(n,s)` calculates the set of rotation angles, `r1`, `r2`, `r3`, for a given direction cosine matrix, `n`, and a specified rotation sequence, `s`.

The default rotation sequence is `'ZYX'`, where `r1` is *z*-axis rotation, `r2` is *y*-axis rotation, and `r3` is *x*-axis rotation.

Supported rotation sequences are `'ZYX'`, `'ZYZ'`, `'ZXY'`, `'ZXZ'`, `'YXZ'`, `'YXY'`, `'YZX'`, `'YZY'`, `'XYZ'`, `'XYX'`, `'XZY'`, and `'XZX'`.

`[r1 r2 r3] = dcm2angle(n,s,lim)` calculates the set of rotation angles, `r1`, `r2`, `r3`, for a given direction cosine matrix, `n`, a specified rotation sequence, `s`, and a specified angle constraint, `lim`. `lim` specifies either `'Default'` or `'ZeroR3'`. See “Assumptions and Limitations” on page 4-223 for full definitions of angle constraints.

`[r1 r2 r3] = dcm2angle(n,s,lim,action)` performs `action` if the direction cosine matrix is invalid (not orthogonal).

- Warning — Displays warning and indicates that the direction cosine matrix is invalid.
- Error — Displays error and indicates that the direction cosine matrix is invalid.
- None — Does not display warning or error (default).

`[r1 r2 r3] = dcm2angle(n,s,lim,action,tolerance)` uses a `tolerance` level to evaluate if the direction cosine matrix, `n`, is valid (orthogonal). `tolerance` is a scalar whose default is `eps(2)` (`4.4409e-16`). The function considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance `tolerance` (`transpose(n)*n == 1±tolerance`)
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Examples

Determine the rotation angles from direction cosine matrix:

```
dcm = [1 0 0; 0 1 0; 0 0 1];
[yaw, pitch, roll] = dcm2angle(dcm)
yaw =

     0

pitch =

     0

roll =

     0
```

Determine the rotation angles from multiple direction cosine matrices:

```
dcm        = [ 1 0 0; 0 1 0; 0 0 1];
dcm(:,:,2) = [ 0.85253103550038   0.47703040785184  -0.21361840626067; ...
              -0.43212157513194   0.87319830445628   0.22537893734811; ...
               0.29404383655186  -0.09983341664683   0.95056378592206];
[pitch, roll, yaw] = dcm2angle(dcm,'YXZ')

pitch =

         0
```

```
    0.3000

roll =

         0
    0.1000

yaw =

         0
    0.5000
```

Determine the rotation angles from direction matrices validated within tolerance:

```
dcm        = [ 1 0 0; 0 1 0; 0 0 1];
dcm(:,:,2) = [ 0.85253103550038   0.47703040785184  -0.21361840626067; ...
              -0.43212157513194   0.87319830445628   0.22537893734811; ...
               0.29404383655186  -0.09983341664683   0.95056378592206];
[pitch, roll, yaw] = dcm2angle(dcm,'YXZ','Default','None',0.1)

pitch =
         0
    0.3000

roll =
         0
    0.1000

yaw =
         0
    0.5000
```

Assumptions and Limitations

The `'Default'` limitations for the `'ZYX'`, `'ZXY'`, `'YXZ'`, `'YZX'`, `'XYZ'`, and `'XZY'` implementations generate an `r2` angle that lies between ±90 degrees, and `r1` and `r3` angles that lie between ±180 degrees.

The `'Default'` limitations for the `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` implementations generate an `r2` angle that lies 0–180 degrees, and `r1` and `r3` angles that lie between ±180 degrees.

The `'ZeroR3'` limitations for the `'ZYX'`, `'ZXY'`, `'YXZ'`, `'YZX'`, `'XYZ'`, and `'XZY'` implementations generate an `r2` angle that lies between ±90 degrees, and `r1` and `r3` angles that lie between ±180 degrees. However, when `r2` is ±90 degrees, `r3` is set to 0 degrees.

The `'ZeroR3'` limitations for the `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` implementations generate an `r2` angle that lies 0–180 degrees, and `r1` and `r3` angles that lie between ±180 degrees. However, when `r2` is 0 or ±180 degrees, `r3` is set to 0 degrees.

See Also

`angle2dcm` | `dcm2quat` | `quat2angle` | `quat2dcm`

Introduced in R2006b

dcm2latlon

Convert direction cosine matrix to geodetic latitude and longitude

Syntax

```
[lat lon] = dcm2latlon(n)
[lat lon] = dcm2latlon(n,action)
[lat lon] = dcm2latlon(n,action,tolerance)
```

Description

`[lat lon] = dcm2latlon(n)` calculates the geodetic latitude and longitude, `lat` and `lon`, for a given direction cosine matrix, `n`. `n` is a 3-by-3-by-`m` matrix containing `m` orthogonal direction cosine matrices. `lat` is an `m` array of geodetic latitudes. `lon` is an `m` array of longitudes. `n` performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes into a vector in north-east-down (NED) axes. Geodetic latitudes and longitudes are output in degrees.

`[lat lon] = dcm2latlon(n,action)` performs `action` if the direction cosine matrix is invalid (not orthogonal).

- Warning — Displays warning and indicates that the direction cosine matrix is invalid.
- Error — Displays error and indicates that the direction cosine matrix is invalid.
- None — Does not display warning or error (default).

`[lat lon] = dcm2latlon(n,action,tolerance)` uses a `tolerance` level to evaluate if the direction cosine matrix, `n`, is valid (orthogonal). `tolerance` is a scalar whose default is `eps(2)` (`4.4409e-16`). The function considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`)
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Examples

Determine the geodetic latitude and longitude from direction cosine matrix:

```
dcm = [ 0.3747    0.5997    0.7071; ...
        0.8480   -0.5299         0; ...
        0.3747    0.5997   -0.7071];
[lat, lon] = dcm2latlon(dcm)

lat =

   44.9995


lon =

 -122.0005
```

Determine the geodetic latitude and longitude from multiple direction cosine matrices:

```
dcm = [ 0.3747    0.5997    0.7071; ...
        0.8480   -0.5299         0; ...
        0.3747    0.5997   -0.7071];
dcm(:,:,2) = [-0.0531    0.6064    0.7934; ...
               0.9962    0.0872         0; ...
              -0.0691    0.7903   -0.6088];
[lat, lon] = dcm2latlon(dcm)


lat =

   44.9995
   37.5028


lon =

 -122.0005
  -84.9975
```

Determine the geodetic latitude and longitude from multiple direction cosine matrices validated within tolerance:

```
dcm = [ 0.3747    0.5997    0.7071; ...
        0.8480   -0.5299         0; ...
        0.3747    0.5997   -0.7071];
dcm(:,:,2) = [-0.0531    0.6064    0.7934; ...
               0.9962    0.0872         0; ...
              -0.0691    0.7903   -0.6088];
[lat, lon] = dcm2latlon(dcm,'Warning',0.1)

lat =
   44.9995
   37.5028
lon =
 -122.0005
  -84.9975
```

See Also

angle2dcm | dcm2angle | dcmecef2ned

Introduced in R2006b

dcm2quat

Convert direction cosine matrix to quaternion

Syntax

```
q = dcm2quat(n)
q = dcm2quat(n,action)
q = dcm2quat(n,action,tolerance)
```

Description

`q = dcm2quat(n)` calculates the quaternion, `q`, for a given direction cosine matrix, `n`. Input `n` is a 3-by-3-by-`m` matrix of orthogonal direction cosine matrices. The direction cosine matrix performs the coordinate transformation of a vector in inertial axes to a vector in body axes. `q` returns an `m`-by-4 matrix containing `m` quaternions. `q` has its scalar number as the first column.

This function applies only to direction cosine matrices that are orthogonal with determinant +1.

`q = dcm2quat(n,action)` performs `action` if the direction cosine matrix is invalid (not orthogonal).

- Warning — Displays warning and indicates that the direction cosine matrix is invalid.
- Error — Displays error and indicates that the direction cosine matrix is invalid.
- None — Does not display warning or error (default).

`q = dcm2quat(n,action,tolerance)` uses a `tolerance` level to evaluate if the direction cosine matrix, `n`, is valid (orthogonal). `tolerance` is a scalar whose default is `eps(2)` (`4.4409e-16`). The function considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`)

- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Examples

Determine the quaternion from direction cosine matrix:

```
dcm = [0 1 0; 1 0 0; 0 0 -1];
q = dcm2quat(dcm)
q =

         0    0.7071    0.7071         0
```

Determine the quaternions from multiple direction cosine matrices:

```
dcm        = [ 0 1 0; 1 0 0; 0 0 -1];
dcm(:,:,2) = [ 0.4330    0.2500   -0.8660; ...
               0.1768    0.9186    0.3536; ...
               0.8839   -0.3062    0.3536];
q = dcm2quat(dcm)
q =

         0    0.7071    0.7071         0
    0.8224    0.2006    0.5320    0.0223
```

Determine the quaternion from a direction cosine matrix validated within tolerance:

```
dcm = [0 1 0; 1 0 0; 0 0 -1];
q = dcm2quat(dcm,'Warning',0.1)

q =

         0    0.7071    0.7071         0
```

See Also

`angle2dcm` | `angle2quat` | `dcm2angle` | `quat2angle` | `quat2dcm`

Introduced in R2006b

dcm2rod

Convert direction cosine matrix to Euler-Rodrigues vector

Syntax

```
R = dcm2rod(dcm)
R = dcm2rod(dcm,action)
R = dcm2rod(dcm,action,tolerance)
```

Description

`R = dcm2rod(dcm)` function calculates the Euler-Rodrigues vector (`R`) from the direction cosine matrix. This function applies only to direction cosine matrices that are orthogonal with determinant +1.

`R = dcm2rod(dcm,action)` performs `action` if the direction cosine matrix is invalid (not orthogonal).

`R = dcm2rod(dcm,action,tolerance)` uses a `tolerance` level to evaluate if the direction cosine matrix, `n`, is valid (orthogonal).

Examples

Determine Rodrigues Vector from Direction Cosine Matrix

Determine the Rodrigues vector from the direction cosine matrix.

```
DCM = [0.433 0.75 0.5;-0.25 -0.433 0.866;0.866 -0.5 0.0];
r = dcm2rod(DCM)
```

```
r =

    1.3660    0.3660    1.0000
```

Determine Rodrigues Vector from Direction Cosine Matrix Validated within Tolerance:

Determine the Rodrigues vector from the direction cosine matrix validated within tolerance.

```
DCM = [0.433 0.75 0.5;-0.25 -0.433 0.866;0.866 -0.5 0.0];
r = dcm2rod(DCM,'Warning',0.1)
```

```
r =
    1.3660    0.3660    1.0000
```

Input Arguments

`dcm` — Direction cosine matrix

3-by-3-by-*M* matrix

3-by-3-by-*M* containing *M* direction cosine matrices.

Data Types: `double`

`action` — Function behavior

`'None'` (default) | `'Error'` | `'Warning'`

Function behavior when direction cosine matrix is invalid (not orthogonal).

- Warning — Displays warning and indicates that the direction cosine matrix is invalid.
- Error — Displays error and indicates that the direction cosine matrix is invalid.
- None — Does not display warning or error (default).

Data Types: `char` | `string`

`tolerance` — Tolerance

`eps(2)` (`4.4409e-16`) (default) | scalar

Tolerance of direction cosine matrix validity, specified as a scalar. The function considers the direction cosine matrix valid if these conditions are true:

- The transpose of the direction cosine matrix times itself equals `1` within the specified tolerance (`transpose(n)*n == 1±tolerance`)
- The determinant of the direction cosine matrix equals `1` within the specified tolerance (`det(n) == 1±tolerance`).

Data Types: `double`

Output Arguments

R — Rodrigues vector
M-by-3 matrix

M-by-3 matrix containing *M* Rodrigues vectors.

Data Types: `double`

Algorithms

An Euler-Rodrigues vector $\overrightarrow{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\overrightarrow{b} = [b_x \; b_y \; b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\overrightarrow{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

See Also

`angle2rod` | `quat2rod` | `rod2angle` | `rod2dcm` | `rod2quat`

Introduced in R2017a

dcmbody2wind

Convert angle of attack and sideslip angle to direction cosine matrix

Syntax

```
n = dcmbody2wind(a, b)
```

Description

`n = dcmbody2wind(a, b)` calculates the direction cosine matrix, `n`, for given angle of attack and sideslip angle, `a`, `b`. `a` is an `m` array of angles of attack. `b` is an `m` array of sideslip angles. `n` returns a 3-by-3-by-`m` matrix containing `m` direction cosine matrices. `n` performs the coordinate transformation of a vector in body-axes into a vector in wind-axes. Angles of attack and sideslip angles are input in radians.

Examples

Determine the direction cosine matrix from angle of attack and sideslip angle:

```
alpha = 0.4363;
beta = 0.1745;
dcm = dcmbody2wind(alpha, beta)

dcm =

    0.8926    0.1736    0.4162
   -0.1574    0.9848   -0.0734
   -0.4226         0    0.9063
```

Determine the direction cosine matrix from multiple angles of attack and sideslip angles:

```
alpha = [0.4363 0.1745];
beta = [0.1745 0.0873];
dcm = dcmbody2wind(alpha, beta)

dcm(:,:,1) =
    0.8926    0.1736    0.4162
   -0.1574    0.9848   -0.0734
   -0.4226         0    0.9063


dcm(:,:,2) =

    0.9811    0.0872    0.1730
   -0.0859    0.9962   -0.0151
   -0.1736         0    0.9848
```

See Also

`angle2dcm` | `dcm2alphabeta` | `dcm2angle`

Introduced in R2006b

dcmecef2ned

Convert geodetic latitude and longitude to direction cosine matrix

Syntax

```
n = dcmecef2ned(lat, lon)
```

Description

`n = dcmecef2ned(lat, lon)` calculates the direction cosine matrix, `n`, for a given set of geodetic latitude and longitude, `lat`, `lon`. `lat` is an `m` array of geodetic latitudes. `lon` is an `m` array of longitudes. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles. `n` returns a 3-by-3-by-`m` matrix containing `m` direction cosine matrices. `n` performs the coordinate transformation of a vector in Earth-centered Earth-fixed (ECEF) axes into a vector in north-east-down (NED) axes. Geodetic latitudes and longitudes are input in degrees.

Examples

Determine the direction cosine matrix from geodetic latitude and longitude:

```
lat = 45;
lon = -122;
dcm = dcmecef2ned(lat, lon)

dcm =

    0.3747    0.5997    0.7071
    0.8480   -0.5299         0
    0.3747    0.5997   -0.7071
```

Determine the direction cosine matrix from multiple geodetic latitudes and longitudes:

```
lat = [45 37.5];
lon = [-122 -85];
```

```
dcm = dcmecef2ned(lat, lon)

dcm(:,:,1) =

    0.3747    0.5997    0.7071
    0.8480   -0.5299         0
    0.3747    0.5997   -0.7071


dcm(:,:,2) =

   -0.0531    0.6064    0.7934
    0.9962    0.0872         0
   -0.0691    0.7903   -0.6088
```

See Also

`angle2dcm` | `dcm2angle` | `dcm2latlon`

Introduced in R2006b

dcmeci2ecef

Convert Earth-centered inertial (ECI) to Earth-centered Earth-fixed (ECEF) coordinates

Syntax

```
dcm=dcmeci2ecef(reduction,utc)
```

```
dcm=dcmeci2ecef(reduction,utc,deltaAT)
dcm=dcmeci2ecef(reduction,utc,deltaAT,deltaUT1)
dcm=dcmeci2ecef(reduction,utc,deltaAT,deltaUT1,polarmotion)
dcm=dcmeci2ecef(reduction,utc,deltaAT,deltaUT1,polarmotion,
Name,Value)
```

Description

`dcm=dcmeci2ecef(reduction,utc)` calculates the position direction cosine matrix (ECI to ECEF) as a 3-by-3-by-*M* array. The calculation is based on the specified reduction method and Universal Coordinated Time (UTC).

`dcm=dcmeci2ecef(reduction,utc,deltaAT)` uses the difference between International Atomic Time and UTC to calculate the position direction cosine matrix.

`dcm=dcmeci2ecef(reduction,utc,deltaAT,deltaUT1)` uses the difference between UTC and Universal Time (UT1).

`dcm=dcmeci2ecef(reduction,utc,deltaAT,deltaUT1,polarmotion)` uses the polar displacement.

`dcm=dcmeci2ecef(reduction,utc,deltaAT,deltaUT1,polarmotion, Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Examples

Convert using IAU-2000/2006 reduction

Convert Earth-centered inertial (ECI) to Earth-centered Earth-fixed (ECEF) coordinates for January 12, 2000 at 4 hours, 52 minutes, 12.4 seconds and January 12, 2000 at 4 hours, 52 minutes, and 13 seconds. Use the IAU-2000/2006 reduction. Specify only the reduction method and UTC.

```
dcm = dcmeci2ecef('IAU-2000/2006',[2000 1 12 4 52 12.4;2000 1 12 4 52 13])
```

```
dcm(:,:,1) =

   -0.9975   -0.0708   -0.0000
    0.0708   -0.9975   -0.0000
   -0.0000   -0.0000    1.0000

dcm(:,:,2) =

   -0.9975   -0.0709   -0.0000
    0.0709   -0.9975   -0.0000
   -0.0000   -0.0000    1.0000
```

Convert using IAU-76/FK5 reduction

Convert Earth-centered inertial (ECI) to Earth-centered Earth-fixed (ECEF) coordinates for January 12, 2000 at 4 hours, 52 minutes, 12.4 seconds. Use the IAU-76/FK5 reduction. Specify all arguments, including optional ones such as polar motion.

```
dcm = dcmeci2ecef('IAU-76/FK5',[2000 1 12 4 52 12.4],32,0.234,[-0.0682e-5 ...
0.1616e-5],'dNutation',[-0.2530e-6 -0.0188e-6])
```

```
dcm =

   -0.9975   -0.0708   -0.0000
    0.0708   -0.9975   -0.0000
   -0.0000   -0.0000    1.0000
```

Input Arguments

reduction — Reduction method

'IAU-76/FK5' | 'IAU-2000/2006'

Reduction method to calculate the direction cosine matrix, specified as one of the following:

- IAU-76/FK5

  Reduce the calculation using the International Astronomical Union (IAU)-76/Fifth Fundamental Catalogue (FK5) (IAU-76/FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5. You can use the `'dNutation'` Name,Value pair with this reduction.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, `dcmeci2ecef` calculates the transformation matrix rather than the direction cosine matrix.
- IAU-2000/2006

  Reduce the calculation using the International Astronomical Union (IAU)-2000/2005 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation. You can use the `'dCIP'` Name,Value pair with this reduction.

`utc` — Universal Coordinated Time
1-by-6 array | *M*-by-6 matrix

Universal Coordinated Time (UTC) in the order year, month, day, hour, minutes, and seconds, for which the function calculates the direction cosine matrix, specified as one of the following.

- For the year value, enter a double value that is a whole number greater than 1, such as `2013`.
- For the month value, enter a double value that is a whole number greater than 0, within the range `1` to `12`.
- For the day value, enter a double value that is a whole number greater than 0, within the range `1` to `31`.
- For the hour value, enter a double value that is a whole number greater than 0, within the range `1` to `24`.
- For the minute and second values, enter a double value that is a whole number greater than 0, within the range `1` to `60`.

Specify these values in one of the following formats:

- 1-by-6 array

  Specify a 1-row-by-6-column array of UTC values to calculate one direction cosine or transformation matrix.
- *M*-by-6 matrix

  Specify an *M*-by-6 array of UTC values, where *M* is the number of direction cosine or transformation matrices to calculate. Each row corresponds to one set of UTC values.

Example: `[2000 1 12 4 52 12.4]`

This is a one row-by-6 column array of UTC values.

Example: `[2000 1 12 4 52 12.4;2010 6 5 7 22 0]`

This is an *M*-by-6 array of UTC values, where *M* is 2.

Data Types: `double`

`deltaAT` — Difference between International Atomic Time and UTC
scalar | one-dimensional array

Difference between International Atomic Time (IAT) and UTC, in seconds, for which the function calculates the direction cosine or transformation matrix. By default, the function assumes an *M*-by-1 array of zeroes.

- scalar

  Specify one difference-time value to calculate one direction cosine or transformation matrix.
- one-dimensional array

  Specify a one-dimensional array with *M* elements, where *M* is the number of direction cosine or transformation matrices to calculate. Each row corresponds to one set of UTC values.

Example: `32`

Specify `32` seconds as the difference between IAT and UTC.

Data Types: `double`

`deltaUT1` — Difference between UTC and Universal Time (UT1)
scalar | one-dimensional array

Difference between UTC and Universal Time (UT1) in seconds, for which the function calculates the direction cosine or transformation matrix. By default, the function assumes an *M*-by-1 array of zeroes.

- scalar

  Specify one difference-time value to calculate one direction cosine or transformation matrix.
- one-dimensional array

  Specify a one-dimensional array with *M* elements of difference time values, where *M* is the number of direction cosine or transformation matrices to be calculated. Each row corresponds to one set of UTC values.

Example: `0.234`

Specify `0.234` seconds as the difference between UTC and UT1.

Data Types: `double`

`polarmotion` — Polar displacement
1-by-2 array | *M*-by-2 array

Polar displacement of the Earth, in radians, from the motion of the Earth crust, along the *x*- and *y*-axes. By default, the function assumes an *M*-by-2 array of zeroes.

- 1-by-2 array

  Specify a 1-by-2 array of the polar displacement values to convert one direction cosine or transformation matrix.
- *M*-by-2 array

  Specify an *M*-by-2 array of polar displacement values, where *M* is the number of direction cosine or transformation matrices to convert. Each row corresponds to one set of UTC values.

Example: `[-0.0682e-5 0.1616e-5]`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[-0.2530e-6 -0.0188e-6]`

`dNutation` — Adjustment to longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*)

M-by-2 array

Adjustment to the longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*), in radians, as the comma-separated pair consisting of `dNutation` and an *M*-by-2 array. Use this Name,Value pair with the IAU-76/FK5 reduction. By default, the function assumes an *M*-by-2 array of zeroes.

For historical values, see the International Earth Rotation and Reference Systems Service Web site (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/Products page.

- *M*-by-2 array

  Specify *M*-by-2 array of adjustment values, where *M* is the number of direction cosine or transformation matrices to be converted. Each row corresponds to one set of longitude and obliquity values.

Data Types: `double`

`dCIP` — Adjustment to the location of the Celestial Intermediate Pole (CIP)

M-by-2 array

Adjustment to the location of the Celestial Intermediate Pole (CIP), in radians, specified as the comma-separated pair consisting of `dCIP` and an *M*-by-2 array. This location (*dDeltaX*, *dDeltaY*) is along the *x*- and *y*- axes. Use this argument with the IAU-200/2006 reduction. By default, this function assumes an *M*-by-2 array of zeroes.

For historical values, see the International Earth Rotation and Reference Systems Service Web site (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/Products page.

- *M*-by-2 array

Specify *M*-by-2 array of location adjustment values, where *M* is the number of direction cosine or transformation matrices to be converted. Each row corresponds to one set of *dDeltaX* and *dDeltaY* values.

Example: `[-0.2530e-6 -0.0188e-6]`

Data Types: `double`

Output Arguments

`dcm` — Direction cosine or transformation matrix
3-by-3-*M* array

Direction cosine or transformation matrix, returned as a 3-by-3-*M* array.

See Also

`ecef2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef`

Topics

https://www.iers.org

Introduced in R2013b

decyear

Decimal year calculator

Syntax

```
dy = decyear(v)
dy = decyear(s,f)
dy = decyear(y,mo,d)
dy = decyear([y,mo,d])
dy = decyear(y,mo,d,h,mi,s)
dy = decyear([y,mo,d,h,mi,s])
```

Description

`dy = decyear(v)` converts one or more date vectors, `v`, into decimal year, `dy`. Input `v` can be an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. `decyear` returns a column vector of `m` decimal years.

A date vector contains six elements, specifying year, month, day, hour, minute, and second. A partial date vector has three elements, specifying year, month, and day. Each element of `v` must be a positive double-precision number.

`dy = decyear(s,f)` converts one or more date character vectors, `s`, to decimal year, `dy`, using format `f`. `s` can be a character array, where each row corresponds to one date, or a one-dimensional cell array of character vectors. `decyear` returns a column vector of `m` decimal years, where `m` is the number of character vectors in `s`.

All of the dates in `s` must have the same format `f`, composed of the same date format symbols as the `datestr` function. `decyear` does not accept formats containing the letter *`Q`*.

If a format does not contain enough information to compute a date number, then:

- Hours, minutes, and seconds default to 0.
- Days default to 1.

- Months default to January.
- Years default to the current year.

Dates with two-character years are interpreted to be within 100 years of the current year.

`dy = decyear(y,mo,d)` and `dy = decyear([y,mo,d])` return the decimal year for corresponding elements of the `y,mo,d` (year,month,day) arrays. Specify `y`, `mo`, and `d` as one-dimensional arrays of the same length or scalar values.

`dy = decyear(y,mo,d,h,mi,s)` and `dy = decyear([y,mo,d,h,mi,s])` return the decimal year for corresponding elements of the `y,mo,d,h,mi,s` (year,month,day,hour,minute,second) arrays. Specify the six arguments as one-dimensional arrays of the same length or scalar values.

Examples

Calculate decimal year for May 24, 2005:

```
dy = decyear('24-May-2005','dd-mmm-yyyy')

dy =

  2.0054e+003
```

Calculate decimal year for December 19, 2006:

```
dy = decyear(2006,12,19)

dy =

  2.0070e+003
```

Calculate decimal year for October 10, 2004, at 12:21:00 p.m.:

```
dy = decyear(2004,10,10,12,21,0)

dy =

  2.0048e+003
```

Assumptions and Limitations

The calculation of decimal year does not take into account leap seconds.

See Also

`juliandate` | `leapyear` | `mjuliandate`

Introduced in R2006b

delete

Class: `Aero.Animation`
Package: `Aero`

Destroy animation object

Syntax

```
delete(h)
h.delete
```

Description

`delete(h)` and `h.delete` destroy the animation object `h`. This function also destroys the animation object figure, and any objects that the animation object contained (for example, bodies, camera, and geometry).

Input Arguments

`h`	Animation object.

Examples

Delete the animation object, `h`.

```
h=Aero.Animation;
h.delete;
```

delete (Aero.FlightGearAnimation)

Destroy FlightGear animation object

Syntax

```
delete(h)
h.delete
```

Description

`delete(h)` and `h.delete` destroy the FlightGear animation object `h`. This function also destroys the animation object timer, and closes the socket that the FlightGear animation object contains.

Examples

Delete the FlightGear animation object, `h`.

```
h=Aero.FlightGearAnimation;
h.delete;
```

See Also

`initialize`

Introduced in R2007a

delete (Aero.VirtualRealityAnimation)

Destroy virtual reality animation object

Syntax

```
delete(h)
h.delete
```

Description

`delete(h)` and `h.delete` destroy the virtual reality animation object `h`. This function also destroys the temporary file, if it exists, cleans up the vrfigure object, the animation object timer, and closes the vrworld object.

Examples

Delete the virtual reality animation object, h.

```
h=Aero.VirtualRealityAnimation;
h.delete;
```

See Also

`initialize`

Introduced in R2007b

deltaCIP

Calculate Celestial Intermediate Pole (CIP) location adjustment

Syntax

```
DCIP=deltaCIP(utc)
[DCIP,DCIPError]=deltaCIP(utc)
```

```
DCIP=deltaCIP(utc,Name,Value)
[DCIP,DCIPError]=deltaCIP(utc,Name,Value)
```

Description

`DCIP=deltaCIP(utc)` calculates the adjustment to location of the Celestial Intermediate Pole (CIP) for a specific Universal Coordinated Time (UTC), specified as a modified Julian date. By default, this function uses a prepopulated list of IAU 2000A Earth orientation (IERS) data. This list contains measured and calculated (predicted) data supplied by the IERS. The IERS measures and calculates this data for a set of predetermined dates.

`[DCIP,DCIPError]=deltaCIP(utc)` returns the error for the adjustment to location of the CIP.

`DCIP=deltaCIP(utc,Name,Value)` calculates the location of the CIP using additional options specified by one or more `Name,Value` pair arguments.

`[DCIP,DCIPError]=deltaCIP(utc,Name,Value)` returns the error for the adjustment to location of the CIP.

Examples

Calculate CIP Location Adjustment

Calculate the CIP adjustment for December 28, 2015. Use the `mjuliandate` function to calculate the date as a modified Julian date.

```
mjd = mjuliandate(2015,12,28)
dCIP = deltaCIP(mjd)

mjd =
       57384
dCIP =
   1.0e-09 *
   -0.3927    0.0145
```

Calculate CIP Location Adjustment and Error Using IERS Data

Calculate the CIP adjustment and CIP adjustment error for December 28, 2015 and January 10, 2016 using the `aeroiersdata.mat` file. Use the `mjuliandate` function to calculate the date as a modified Julian date.

```
mjd = mjuliandate([2015 12 28;2016 1 10])
[dCIP,dCIPErr] = deltaCIP(mjd,'Source','aeroiersdata.mat')

mjd =
       57384
       57397
dCIP =

   1.0e-08 *

   -0.0393    0.0015
   -0.0087   -0.1110

dCIPErr =
   1.0e-09 *
    0.5769    0.1842
    0.2376    0.4121
```

Input Arguments

`utc` — Principal Universal Time (UT1) for UTC
M-by-1 array

Array of UTC dates, specified as an *M*-by-1 array, represented as modified Julian dates. Use the `mjuliandate` function to convert the UTC date to a modified Julian date.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Source','aeroiersdata.mat'

`Source` — Custom list of Earth orientation data

`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

`action` — Out-of-range action

Warning (default) | action

Out-of-range action, specified as a string.

Action to take in case of out-of-range or predicted value dates, specified as a string:

- Warning — Displays warning and indicates that the dates were out-of-range or predicted values.
- Error — Displays error and indicates that the dates were out-of-range or predicted values.
- None — Does not display warning or error.

Data Types: `string`

Output Arguments

`DCIP` — Adjustment to location of the CIP

M-by-2 array

Adjustment (`[dDeltaX,dDeltaY]`) to location of the Celestial Intermediate Pole (CIP), specified as an *M*-by-2 array, in radians.

`DCIPError` — Error for adjustment to location of the CIP
M-by-2 array

Error for adjustment to location of the CIP, specified as an *M*-by-2 array, in radians.

See Also

`aeroReadIERSData` | `dcmeci2ecef` | `deltaUT1` | `eci2aer` | `eci2lla` | `lla2eci` | `mjuliandate` | `polarMotion`

Topics

“Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-141

Introduced in R2018b

deltaUT1

Calculate difference between Coordinated Universal Time (UTC) and Principal Universal Time (UT1)

Syntax

```
DUT1=deltaUT1(utc)
[DUT1,DUT1Error]=deltaUT1(utc)

DUT1=deltaUT1(utc,Name,Value)
[DUT1,DUT1Error]=deltaUT1(utc,Name,Value)
```

Description

`DUT1=deltaUT1(utc)` calculates the difference between Coordinated Universal Time (UTC) and Principal Universal Time (UT1) for UTC, specified as a modified Julian date (MJD). By default, this function uses a prepopulated list of International Astronomical Union (IAU) 2000A Earth orientation (IERS) data. This list contains measured and calculated (predicted) data supplied by the IERS. The IERS measures and calculates this data for a set of predetermined dates. For dates after those listed in the prepopulated list, `deltaUT1` calculates the data by using this equation, limiting the values to +/- .9s:

```
UT1-UTC=0.5309-0.00123(MJD-57808)-(UT2-UT1)
```

`[DUT1,DUT1Error]=deltaUT1(utc)` returns the error for the difference between Coordinated Universal Time (UTC) and Principal Universal Time (UT1) for UTC.

`DUT1=deltaUT1(utc,Name,Value)` calculates the difference between UTC and UT1 using additional options specified by one or more `Name,Value` pair arguments.

`[DUT1,DUT1Error]=deltaUT1(utc,Name,Value)` returns the error for the difference between Coordinated Universal Time (UTC) and Principal Universal Time (UT1) for UTC.

Examples

Calculate Difference Value for December 28, 2015

Calculate the difference between UT1 and UTC values for December 28, 2015.

```
mjd = mjuliandate(2015,12,28)
dUT1 = deltaUT1(mjd)
```

```
mjd =
       57384

dUT1 =
    0.0886
```

Calculate Difference Value for December 28, 2015 and January 10, 2016

Calculate the difference between UT1 and UTC values for December 28, 2015 and January 10, 2016 using the custom file, aeroiersdata20170101.mat.

```
mjd = mjuliandate([2015 12 28;2016 1 10])
dUT1 = deltaUT1(mjd,'Source','aeroiersdata20170101.mat')
```

```
mjd =
       57384
       57397

dUT1 =
    0.0886
    0.0648
```

Calculate Difference Value and Error Using IERS Data

Calculate the difference between UT1-UTC values for December 28, 2015 and January 10, 2016 using the custom file, `aeroiersdata.mat`.

```
mjd = mjuliandate([2015 12 28;2016 1 10])
[dUT1,dUT1Err] = deltaUT1(mjd,'Source','aeroiersdata.mat')
```

```
mjd =
       57384
       57397


dUT1 =
    0.0886
    0.0648


dUT1Err =
   1.0e-05 *

    0.3900
    0.7300
```

Input Arguments

`utc` — Principal Universal Time (UT1) for UTC
M-by-1 array

Array of UTC dates, specified as an *M*-by-1 array, represented as modified Julian dates. Use the `mjuliandate` function to convert the UTC date to a modified Julian date.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Source','aeroiersdata.mat'

`Source` — Custom list of Earth orientation data
`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

`action` — Out-of-range action
Warning (default) | action

Out-of-range action, specified as a string.

Action to take in case of out-of-range or predicted value dates, specified as a string:

- Warning — Displays warning and indicates that the dates were out-of-range or predicted values.
- Error — Displays error and indicates that the dates were out-of-range or predicted values.
- None — Does not display warning or error.

Data Types: `string`

Output Arguments

`DUT1` — Difference between UT1 and UTC
M-by-1 array

Difference between UT1 and UTC, specified as an *M*-by-1 array.

`DUT1Error` — Error for difference between UT1 and UTC
M-by-1 array

Error for difference between UT1 and UTC (UT1-UTC), according to the International Astronomical Union (IAU) 2000A resolutions, specified as an *M*-by-1 array, in seconds.

See Also

`aeroReadIERSData` | `dcmeci2ecef` | `eci2aer` | `eci2lla` | `lla2eci` | `mjuliandate`

Topics

"Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation" on page 5-141

Introduced in R2017b

dpressure

Compute dynamic pressure using velocity and density

Syntax

```
q = dpressure(v, r)
```

Description

`q = dpressure(v, r)` computes `m` dynamic pressures, `q`, from an `m`-by-3 array of velocities, `v`, and an array of `m` densities, `r`. `v` and `r` must have the same length units.

Examples

Determine dynamic pressure for velocity in feet per second and density in slugs per feet cubed:

```
q = dpressure([84.3905  33.7562  10.1269], 0.0024)


q =

   10.0365
```

Determine dynamic pressure for velocity in meters per second and density in kilograms per meters cubed:

```
q = dpressure([25.7222 10.2889 3.0867], [1.225  0.3639])


q =

  475.9252
  141.3789
```

Determine dynamic pressure for velocity in meters per second and density in kilograms per meters cubed:

```
q = dpressure([50 20 6; 5 0.5 2], [1.225  0.3639])


q =

  1.0e+003 *

    1.7983
    0.0053
```

See Also

`airspeed` | `machnumber`

Introduced in R2006b

earthNutation

Implement Earth nutation

Syntax

```
angles= earthNutation(ephemerisTime)
angles= earthNutation(ephemerisTime,ephemerisModel)
angles= earthNutation(ephemerisTime,ephemerisModel,action)
```

```
[angles,rates] = earthNutation( ___ )
```

Description

`angles= earthNutation(ephemerisTime)` implements the International Astronomical Union (IAU) 1980 nutation series for ephemerisTime, expressed in Julian days. It returns angles.

The function uses the Chebyshev coefficients that the NASA Jet Propulsion Laboratory provides.

This function requires that you download ephemeris data with the Add-On Explorer. For more information, see `aeroDataPackage`.

`angles= earthNutation(ephemerisTime,ephemerisModel)` uses the `ephemerisModel` coefficients to implement these values.

`angles= earthNutation(ephemerisTime,ephemerisModel,action)` uses `action` to determine error reporting.

`[angles,rates] = earthNutation( ___ )` implements the International Astronomical Union (IAU) 1980 nutation series using any combination of the input arguments in the previous syntaxes. It returns angles and angular rates.

Examples

Implement Earth Nutation Angles

Implement Earth nutation angles for December 1, 1990. Because no ephemerides model is specified, the default, DE405, is used. Use the `juliandate` function to specify the Julian date.

```
angles = earthNutation(juliandate(1990,12,1))
```

```
angles =
   1.0e-04 *
    0.6448    0.2083
```

Implement Earth Nutation Angles and Angular Rates

Implement Earth nutation angles and angular rates for noon on January 1, 2000 using DE421:

```
[angles,rates] = earthNutation([2451544.5 0.5],'421')
```

```
angles =
   1.0e-04 *
   -0.6750   -0.2799

rates =
   1.0e-07 *
    0.3687   -0.9937
```

Input Arguments

`ephemerisTime` — Julian date
scalar | 2-element vector | column vector | *M*-by-2 matrix

Julian dates for which the positions are calculated, specified as one of the following:

- Scalar

  Specify one fixed Julian date.
- 2-element vector

  Specify the Julian date in multiple parts. The first element is the Julian date for a specific epoch that is the most recent midnight at or before the interpolation epoch.

The second element is the fractional part of a day elapsed between the first element and epoch. The second element must be positive. The value of the first element plus the second element cannot exceed the maximum Julian date.

- Column vector

  Specify a column vector with *M* elements, where *M* is the number of fixed Julian dates.
- *M*-by-2 matrix

  Specify a matrix, where *M* is the number of Julian dates and the second column contains the elapsed days (Julian epoch date/elapsed day pairs).

Data Types: `double`

`ephemerisModel` — Ephemerides coefficients
`'405'` (default) | `'421'` | `'423'` | `'430'`

Ephemerides coefficients, specified as one of these ephemerides defined by the Jet Propulsion Laboratory:

- `'405'`

  Released in 1998. This ephemerides takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
- `'421'`

  Released in 2008. This ephemerides takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
- `'423'`

  Released in 2010. This ephemerides takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

- `'430'`

  Released in 2013. This ephemerides takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650).

  This function implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Data Types: `char`

`action` — Function behavior
`'Error'` (default) | `'None'` | `'Warning'`

Function behavior when inputs are out of range, specified as one of these values:

Value	Description
`'None'`	No action.
`'Warning'`	Warning in the MATLAB Command Window, model simulation continues.
`'Error'`	MATLAB returns an exception, model simulation stops.

Data Types: `char`

Output Arguments

`angles` — Earth nutation angles
M-by-2 vector

Earth nutation angles, returned as an *M*-by-2 vector, where *M* is the number of Julian dates. The 2 vector contains the *d(psi)* and *d(epsilon)* angles, in radians. The input arguments include multiple Julian dates or epochs. The vector has the same number of rows as the `ephemerisTime` input.

`rates` — Earth nutation angular rates
M-by-2 vector

Earth nutation angular rates, returned as an *M*-by-2 vector, where *M* is the number of Julian dates. The 2 vector contains the *d(psi)* and *d(epsilon)* angular rate, in radians/day. The input arguments include multiple Julian dates or epochs. The vector has the same number of rows as the `ephemerisTime` input.

References

[1] Folkner, W. M., J. G. Williams, D. H. Boggs, “The Planetary and Lunar Ephemeris DE 421,” *JPL Interplanetary Network Progress Report 24-178*, 2009.

[2] Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, McGraw-Hill, New York, 1997.

See Also

`juliandate` | `moonLibration` | `planetEphemeris`

External Websites

`https://ssd.jpl.nasa.gov/?planet_eph_export`

Introduced in R2013a

ecef2lla

Convert Earth-centered Earth-fixed (ECEF) coordinates to geodetic coordinates

Syntax

```
lla = ecef2lla(p)
lla = ecef2lla(p, model)
lla = ecef2lla(p, f, Re)
```

Description

`lla = ecef2lla(p)` converts the `m`-by-3 array of ECEF coordinates, `p`, to an `m`-by-3 array of geodetic coordinates (latitude, longitude and altitude), `lla`. `lla` is in [degrees degrees meters]. `p` is in meters. The default ellipsoid planet is WGS84.

`lla = ecef2lla(p, model)` is an alternate method for converting the coordinates for a specific ellipsoid planet. Currently only `'WGS84'` is supported for `model`.

`lla = ecef2lla(p, f, Re)` is another alternate method for converting the coordinates for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters.

Examples

Determine latitude, longitude, and altitude at a coordinate:

```
lla = ecef2lla([4510731 4510731 0])


lla =

         0   45.0000  999.9564
```

Determine latitude, longitude, and altitude at multiple coordinates, specifying WGS84 ellipsoid model:

```
lla = ecef2lla([4510731 4510731 0; 0 4507609 4498719], 'WGS84')


lla =

         0   45.0000  999.9564
   45.1358   90.0000  999.8659
```

Determine latitude, longitude, and altitude at multiple coordinates, specifying custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
lla = ecef2lla([4510731 4510731 0; 0 4507609 4498719], f, Re)


lla =

  1.0e+006 *

         0    0.0000    2.9821
    0.0000    0.0001    2.9801
```

See Also

`geoc2geod` | `geod2geoc` | `lla2ecef`

Introduced in R2006b

eci2aer

Convert Earth-centered inertial (ECI) coordinates to azimuth, elevation, slant range (AER) coordinates

Syntax

```
aer = eci2aer(position,utc,lla0)

aer = eci2aer(position,utc,lla0,reduction)
aer = eci2aer(position,utc,lla0,reduction,deltaAT)
aer = eci2aer(position,utc,lla0,reduction,deltaAT,deltaUT1)
aer = eci2aer(position,utc,lla0,reduction,deltaAT,deltaUT1,
polarmotion)
aer = eci2aer(position,utc,lla0,reduction,deltaAT,deltaUT1,
polarmotion,Name,Value)
```

Description

`aer = eci2aer(position,utc,lla0)` converts Earth-centered inertial coordinates, specified by position, to azimuth, elevation, and slant range (AER) coordinates, based on the geodetic position (latitude, longitude, and altitude). The conversion is based on the Universal Coordinated Time (UTC) you specify.

- Azimuth (A) — Angle measured clockwise from true north. It ranges from 0 to 360 degrees.
- Elevation (E) — Angle between a plane perpendicular to the ellipsoid and the line that goes from the local reference to the object position. It ranges from –90 to 90 degrees.
- Slant range (R) — Straight line distance between the local reference and the object, meters.

`aer = eci2aer(position,utc,lla0,reduction)` converts Earth-centered inertial coordinates, specified by `position`, to azimuth, elevation, and slant range coordinates. The conversion is based on the specified reduction method and the Universal Coordinated Time you specify.

`aer = eci2aer(position,utc,lla0,reduction,deltaAT)` uses the difference between International Atomic Time and UTC that you specify as `deltaAT` to calculate the AER coordinates.

`aer = eci2aer(position,utc,lla0,reduction,deltaAT,deltaUT1)` uses the difference between UTC and Universal Time (UT1), which you specify as `deltaUT1`, in the calculation.

`aer = eci2aer(position,utc,lla0,reduction,deltaAT,deltaUT1, polarmotion)` uses the polar displacement, `polarmotion`, in the calculation.

`aer = eci2aer(position,utc,lla0,reduction,deltaAT,deltaUT1, polarmotion,Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Examples

Convert Position to AER Coordinates Using UTC

Convert the position to AER coordinates from ECI coordinates 1e08*[-3.8454 -0.5099 -0.3255] meters for the date 1969/7/20 21:17:40 UTC at 28.4 degrees north, 80.5 degrees west and 2.7 meters altitude.

```
aer = eci2aer(1e08*[-3.8454,-0.5099,-0.3255],...
[1969,7,20,21,17,40], [28.4,-80.5,2.7])
```

```
aer =

   1.0e+08 *

    0.0000    0.0000    3.8401
```

Convert Position to AER Coordinates Using UTC and Reduction Method IAU-76/FK5

Convert the position to AER coordinates from ECI coordinates 1e08*[-3.8454 -0.5099 -0.3255] meters for the date 1969/7/20 21:17:40 UTC at 28.4 degrees north, 80.5 degrees west and 2.7 meters altitude. For an ellipsoid with a flattening of 1/290 and an equatorial radius of 60000 meters, use the IAU-76/FK5 reduction, polar motion [-0.0682e-5 0.1616e-5] radians, and nutation angles [-0.2530e-6 -0.0188e-6].

```
aer = eci2aer(1e08*[-3.8454,-0.5099,-0.3255],...
[1969,7,20,21,17,40],[28.4,-80.5,2.7],...
'IAU-76/FK5',32,0.234,[-0.0682e-5 0.1616e-5],...
'dNutation',[-0.2530e-6 -0.0188e-6],...
'flattening',1/290,'RE',60000)
```

```
aer =

   1.0e+08 *

    0.0000    0.0000    3.8922
```

Input Arguments

`position` — ECI coordinates

M-by-3 array

ECI coordinates in meters, specified as an *M*-by-3 array.

`utc` — Universal Coordinated Time

1-by-6 array of whole numbers | *M*-by-6 matrix of whole numbers

Universal Coordinated Time (UTC), in the order year, month, day, hour, minutes, and seconds, for which the function calculates the conversion, specified as one of the following:

- For the year value, enter a double value that is a whole number greater than 1, such as `2013`.
- For the month value, enter a double value that is a whole number greater than 0, within the range `1` to `12`.
- For the hour value, enter a double value that is a whole number greater than 0, within the range `1` to `24`.
- For the hour value, enter a double value that is a whole number greater than 0, within the range `1` to `60`.
- For the minute and second values, enter a double value that is a whole number greater than 0, within the range `1` to `60`.

Specify these values in one of the following formats:

- 1-by-6 array

  Specify a 1-row-by-6-column array of UTC values.
- *M*-by-6 matrix

  Specify an *M*-by-6 array of UTC values, where *M* is the number of transformation matrices to calculate. Each row corresponds to one set of UTC values.

This example is a one-row-by-6-column array of UTC values.

Example: `[2000 1 12 4 52 12.4]`

This example is an *M*-by-6 array of UTC values, where *M* is 2.

Example: `[2000 1 12 4 52 12.4;2010 6 5 7 22 0]`

Data Types: `double`

`lla0` — Geodetic coordinates
M-by-3 array

Geodetic coordinates of the local reference (latitude, longitude, and ellipsoidal altitude), in degrees, degrees, and meters. Latitude and longitude values can be any value. However, latitude values of +90 and –90 can return unexpected values because of singularity at the poles.

`reduction` — Reduction method
`'IAU-2000/2006'` (default) | `'IAU-76/FK5'`

Reduction method to calculate the coordinate conversion, specified as one of the following:

- `'IAU-76/FK5'`

  Reduce the calculation using the International Astronomical Union (IAU)-76/Fifth Fundamental Catalogue (FK5) (IAU-76/FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5. You can use the `'dNutation'` Name,Value pair with this reduction.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, `eci2aer` performs a coordinate conversion that is not orthogonal due to the polar motion approximation.

- `'IAU-2000/2006'`

  Reduce the calculation using the International Astronomical Union (IAU)-2000/2005 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation. You can use the `'dCIP'` Name,Value pair with this reduction.

`deltaAT` — Difference between International Atomic Time and UTC
scalar | one-dimensional array

Difference between International Atomic Time (IAT) and UTC, in seconds, for which the function calculates the direction cosine or transformation matrix. By default, the function assumes an *M*-by-1 array of zeroes.

- scalar

  Specify one difference-time value to calculate one direction cosine or transformation matrix.
- one-dimensional array

  Specify a one-dimensional array with *M* elements, where *M* is the number of direction cosine or transformation matrices to calculate. Each row corresponds to one set of UTC values.

Example: `32`

Specify `32` seconds as the difference between IAT and UTC.

Data Types: `double`

`deltaUT1` — Difference between UTC and Universal Time (UT1)
scalar | one-dimensional array

Difference between UTC and Universal Time (UT1) in seconds, for which the function calculates the direction cosine or transformation matrix. By default, the function assumes an *M*-by-1 array of zeroes.

- scalar

Specify one difference-time value to calculate one direction cosine or transformation matrix.

- one-dimensional array

  Specify a one-dimensional array with *M* elements of difference time values, where *M* is the number of direction cosine or transformation matrices to be calculated. Each row corresponds to one set of UTC values.

Example: `0.234`

Specify `0.234` seconds as the difference between UTC and UT1.

Data Types: `double`

`polarmotion` — Polar displacement
1-by-2 array | *M*-by-2 array

Polar displacement of the Earth, in radians, from the motion of the Earth crust, along the *x*- and *y*-axes. By default, the function assumes an *M*-by-2 array of zeroes.

- 1-by-2 array

  Specify a 1-by-2 array of the polar displacement values to convert one direction cosine or transformation matrix.
- *M*-by-2 array

  Specify an *M*-by-2 array of polar displacement values, where *M* is the number of direction cosine or transformation matrices to convert. Each row corresponds to one set of UTC values.

Example: `[-0.0682e-5 0.1616e-5]`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'dNutation',[-0.2530e-6 -0.0188e-6]`

`dNutation` — Adjustment to longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*)
M-by-2 array of zeroes (default) | *M*-by-2 array

Adjustment to the longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*), in radians, specified as the comma-separated pair consisting of `'dNutation'` and an *M*-by-2 array. You can use this Name,Value pair with the IAU-76/FK5 reduction.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/ Products page.

- *M*-by-2 array

  Specify *M*-by-2 array of adjustment values, where *M* is the number of LLA coordinates to be converted. Each row corresponds to one set of longitude and obliquity values.

Data Types: `double`

`dCIP` — Adjustment to the location of the celestial intermediate pole (CIP)
M-by-2 array of zeroes (default)

Adjustment to the location of the celestial intermediate pole (CIP), in radians, specified as the comma-separated pair consisting of `'dCIP'` and an *M*-by-2 array. This location (*dDeltaX*, *dDeltaY*) is along the *x*- and *y*- axes. You can use this argument with the IAU-200/2006 reduction.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/ Products page.

- *M*-by-2 array

  Specify *M*-by-2 array of location adjustment values, where *M* is the number of LLA coordinates to convert. Each row corresponds to one set of *dDeltaX* and *dDeltaY* values.

Example: `'dCIP',[-0.2530e-6 -0.0188e-6]`

Data Types: `double`

`flattening` — Custom ellipsoid planet
1-by-1 array

Custom ellipsoid planet defined by flattening, specified as the comma-separated pair consisting of `'flattening'` and a 1-by-1 array.

Example: `1/290`

Data Types: `double`

`re` — Custom planet ellipsoid radius
1-by-1 array

Custom planet ellipsoid radius, in meters, specified as the comma-separated pair consisting of `'re'` and a 1-by-1 array.

Example: `60000`

Data Types: `double`

See Also

`dcmeci2ecef` | `ecef2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef` | `lla2eci`

Introduced in R2015a

eci2lla

Convert Earth-centered inertial (ECI) coordinates to latitude, longitude, altitude (LLA) geodetic coordinates

Syntax

```
lla = eci2lla(position,utc)

lla = eci2lla(position,utc,reduction)
lla = eci2lla(position,utc,reduction,deltaAT)
lla = eci2lla(position,utc,reduction,deltaAT,deltaUT1)
lla = eci2lla(position,utc,reduction,deltaAT,deltaUT1,polarmotion)
lla = eci2lla(position,utc,reduction,deltaAT,deltaUT1,polarmotion,
Name,Value)
```

Description

`lla = eci2lla(position,utc)` converts Earth-centered inertial (ECI) coordinates, specified by position, to latitude, longitude, altitude (LLA) geodetic coordinates. The conversion is based on the Universal Coordinated Time (UTC) you specify.

`lla = eci2lla(position,utc,reduction)` converts Earth-centered inertial (ECI) coordinates, specified by `position`, to latitude, longitude, altitude (LLA) geodetic coordinates. The conversion is based on the specified reduction method and the Universal Coordinated Time (UTC) you specify.

`lla = eci2lla(position,utc,reduction,deltaAT)` uses the difference between International Atomic Time and UTC that you specify as `deltaAT` to calculate the ECI coordinates.

`lla = eci2lla(position,utc,reduction,deltaAT,deltaUT1)` uses the difference between UTC and Universal Time (UT1), which you specify as `deltaUT1`, in the calculation.

`lla = eci2lla(position,utc,reduction,deltaAT,deltaUT1,polarmotion)` uses the polar displacement, `polarmotion`, in the calculation.

`lla = eci2lla(position,utc,reduction,deltaAT,deltaUT1,polarmotion, Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Examples

Convert Position to LLA Coordinates Using UTC

Convert the position to LLA coordinates from ECI coordinates [-6.07 -1.28 0.66]*1e6 at 01/17/2010 10:20:36 UTC.

```
lla = eci2lla([-6.07 -1.28 0.66]*1e6,[2010 1 17 10 20 36])
```

```
lla =

   1.0e+05 *

    0.0001   -0.0008   -1.3940
```

Convert Position to LLA Coordinates Using UTC and Reduction Method IAU-76/FK5

Convert the position to LLA coordinates from ECI coordinates [-1.1 3.2 -4.9]*1e4 at 01/12/2000 4:52:12.4 UTC, with a difference of 32 seconds between TAI and UTC, and 0.234 seconds between UTC and UT1. For an ellipsoid with a flattening of 1/290 and an equatorial radius of 60000 meters, use the IAU-76/FK5 reduction, polar motion [-0.0682e-5 0.1616e-5] radians, and nutation angles [-0.2530e-6 -0.0188e-6].

```
lla = eci2lla([-1.1 3.2 -4.9]*1e4,[2000 1 12 4 52 12.4],...
'IAU-76/FK5',32,0.234,[-0.0682e-5 0.1616e-5],'dNutation'...
,[-0.2530e-6 -0.0188e-6],...
'flattening',1/290,'RE',60000)
```

```
lla =

  -55.5592  -75.0892 -311.3709
```

Input Arguments

`position` — ECI coordinates

M-by-3 array

ECI coordinates in meters, specified as an *M*-by-3 array.

`utc` — Universal Coordinated Time

1-by-6 array | *M*-by-6 matrix

Universal Coordinated Time (UTC), in the order year, month, day, hour, minutes, and seconds, for which the function calculates the conversion, specified as one of the following:

- For the year value, enter a double value that is a whole number greater than 1, such as `2013`.
- For the month value, enter a double value that is a whole number greater than 0, within the range `1` to `12`.
- For the hour value, enter a double value that is a whole number greater than 0, within the range `1` to `24`.
- For the hour value, enter a double value that is a whole number greater than 0, within the range `1` to `60`.
- For the minute and second values, enter a double value that is a whole number greater than 0, within the range `1` to `60`.

Specify these values in one of the following formats:

- 1-by-6 array

  Specify a 1-row-by-6-column array of UTC values.
- *M*-by-6 matrix

  Specify an *M*-by-6 array of UTC values, where *M* is the number of transformation matrices to calculate. Each row corresponds to one set of UTC values.

This is a one row-by-6 column array of UTC values.

Example: `[2000 1 12 4 52 12.4]`

This is an *M*-by-6 array of UTC values, where *M* is 2.

Example: `[2000 1 12 4 52 12.4;2010 6 5 7 22 0]`

Data Types: `double`

`reduction` — Reduction method
`'IAU-2000/2006'` (default) | `'IAU-76/FK5'`

Reduction method to calculate the coordinate conversion, specified as one of the following:

- `'IAU-76/FK5'`

  Reduce the calculation using the International Astronomical Union (IAU)-76/Fifth Fundamental Catalogue (FK5) (IAU-76/FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5. You can use the `'dNutation'` Name,Value pair with this reduction.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, `eci2lla` performs a coordinate conversion that is not orthogonal because of the polar motion approximation.

- `'IAU-2000/2006'`

  Reduce the calculation using the International Astronomical Union (IAU)-2000/2005 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation. You can use the `'dCIP'` Name,Value pair with this reduction.

`deltaAT` — Difference between International Atomic Time and UTC
M-by-1 array of zeroes (default) | scalar | one-dimensional array

Difference between International Atomic Time (IAT) and UTC, in seconds, for which the function calculates the coordinate conversion.

- scalar

Specify difference-time value to calculate one direction cosine or transformation matrix.

- one-dimensional array

  Specify a one-dimensional array with *M* elements, where *M* is the number of ECI coordinates. Each row corresponds to one set of ECI coordinates.

Specify `32` seconds as the difference between IAT and UTC.

Example: `32`

Data Types: `double`

`deltaUT1` — Difference between UTC and Universal Time (UT1)
M-by-1 array of zeroes (default) | scalar | one-dimensional array

Difference between UTC and Universal Time (UT1), in seconds, for which the function calculates the coordinate conversion.

- scalar

  Specify difference-time value to calculate ECI coordinates.
- one-dimensional array

  Specify a one-dimensional array with *M* elements of difference time values, where *M* is the number of ECI coordinates. Each row corresponds to one set of ECI coordinates.

Specify `0.234` seconds as the difference between UTC and UT1.

Example: `0.234`

Data Types: `double`

`polarmotion` — Polar displacement
M-by-2 array of zeroes (default) | 1-by-2 array | *M*-by-2 array

Polar displacement of the Earth, in radians, from the motion of the Earth crust, along the *x*- and *y*-axes.

- 1-by-2 array

  Specify a 1-by-2 array of the polar displacement values to convert one ECI coordinate.
- *M*-by-2 array

Specify an *M*-by-2 array of polar displacement values, where *M* is the number of ECI coordinates to convert. Each row corresponds to one set of UTC values.

Example: `[-0.0682e-5 0.1616e-5]`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'dNutation',[-0.2530e-6 -0.0188e-6]`

`dNutation` — Adjustment to longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*)
M-by-2 array of zeroes (default) | *M*-by-2 array

Adjustment to the longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*), in radians, specified as the comma-separated pair consisting of `dNutation` and an *M*-by-2 array. You can use this Name,Value pair with the IAU-76/FK5 reduction.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/Products page.

- *M*-by-2 array

  Specify *M*-by-2 array of adjustment values, where *M* is the number of LLA coordinates to be converted. Each row corresponds to one set of longitude and obliquity values.

Data Types: `double`

`dCIP` — Adjustment to the location of the celestial intermediate pole (CIP)
M-by-2 array of zeroes (default)

Adjustment to the location of the celestial intermediate pole (CIP), in radians, specified as the comma-separated pair consisting of `dCIP` and an *M*-by-2 array. This location (*dDeltaX*, *dDeltaY*) is along the *x*- and *y*- axes. You can use this argument with the IAU-200/2006 reduction.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/ Products page.

- *M*-by-2 array

  Specify *M*-by-2 array of location adjustment values, where *M* is the number of LLA coordinates to convert. Each row corresponds to one set of *dDeltaX* and *dDeltaY* values.

Example: `'dcip',[-0.2530e-5 -0.0188e-4]`

Data Types: `double`

`flattening` — Custom ellipsoid planet
1-by-1 array

Custom ellipsoid planet defined by flattening.

Example: `1/290`

Data Types: `double`

`re` — Custom planet ellipsoid radius
1-by-1 array

Custom planet ellipsoid radius, in meters.

Example: `60000`

Data Types: `double`

See Also

`dcmeci2ecef` | `ecef2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef` | `lla2eci`

Introduced in R2014a

EGTIndicator Properties

Control exhaust gas temperature (EGT) indicator appearance and behavior

Description

EGT indicators are components that represent an EGT indicator. Properties control the appearance and behavior of an EGT indicator. Use dot notation to refer to a particular object and property:

```
f = uifigure;
egtindicator = uiaeroegt(f);
egtindicator.Value = 100;
```

The EGT indicator displays temperature measurements for engine exhaust gas temperature (EGT) in Celsius.

This gauge displays values using both:

- A needle on a gauge. A major tick is (*Maximum*-*Minimum*)/1,000 degrees, a minor tick is (*Maximum*-*Minimum*)/200 degrees Celsius.
- A numeric indicator. The operating range for the indicator goes from *Minimum* to *Maximum* degrees Celsius.

If the value of the signal is under *Minimum*, the needle displays 5 degrees under the *Minimum* value, the numeric display shows the *Minimum* value. If the value exceeds the *Maximum* value, the needle displays 5 degrees over the maximum tick, and the numeric displays the *Maximum* value.

Properties

EGT Indicator

`Limits` — Minimum and maximum indicator scale values

[0 1000] (default) | two-element finite, real, and scalar numeric vector

Minimum and maximum indicator scale values, specified as a two-element numeric vector. The first value in the vector must be less than the second value, in degrees Celsius.

If you change `Limits` such that the `Value` property is less than the new lower limit, or more than the new upper limit, then the indicator needle points to a location off the scale.

For example, suppose `Limits` is `[0 100]` and the `Value` property is 20. If the `Limits` changes to `[50 100]`, then the needle points to a location off the scale, slightly less than 50.

`ScaleColors` — Scale colors
[ ] (default) | n-by-3 array of RGB triplets | cell array

Scale colors, specified one of the following ways:

- An n-by-3 array of RGB triplets
- A cell array containing RGB triplets, any of the color options listed in the table below, or a combination of both.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`
`'black'` or `'k'`	Black	`[0 0 0]`

Each color of the `ScaleColors` array corresponds to a colored section of the indicator. Set the `ScaleColorLimits` property to map the colors to specific sections of the indicator.

If you do not set the `ScaleColorLimits` property, MATLAB distributes the colors equally over the range of the indicator.

Data Types: `double`

`ScaleColorLimits` — Scale color limits

[ ] (default) | n-by-2 array

Scale color limits, specified as an n-by-2 array of numeric values. For every row in the array, the first element must be less than the second element.

When applying colors to the indicator, MATLAB applies the colors starting with the first color in the `ScaleColors` array. Therefore, if two rows in `ScaleColorLimits` array overlap, then the color applied later takes precedence.

The indicator does not display any portion of the `ScaleColorLimits` that falls outside of the `Limits` property.

If the `ScaleColors` and `ScaleColorLimits` property values are different sizes, then the indicator shows only the colors that have matching limits. For example, if the `ScaleColors` array has three colors, but the `ScaleColorLimits` has only two rows, then the indicator displays the first two color/limit pairs only.

`Temperature` — Temperature value

`0` (default) | finite, real, and scalar numeric

Temperature value, specified as any finite and scalar numeric, in degrees Celsius

Example: 10

Dependencies

Specifying this value changes the value of `Value`. Conversely, changing `Value` changes the `Temperature` value.

`Value` — Temperature value

`0` (default) | finite, real, and scalar numeric

Temperature value, specified as any finite and scalar numeric, in degrees Celsius.

Example: 10

Dependencies

Specifying this value changes the value of `Temperature`. Conversely, changing `Temperature` changes the `Value` value.

Interactivity

`Visible` — Visibility of EGT indicator

`'on'` (default) | `'off'`

Visibility of the EGT indicator, specified as `'on'` or `'off'`. The `Visible` property determines whether the EGT indicator, is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire EGT indicator is hidden, but you can still specify and access its properties.

`Enable` — Operational state of EGT indicator

`'on'` (default) | `'off'`

Operational state of EGT indicator, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the EGT indicator indicates that the EGT indicator is operational.
- If you set this property to `'off'`, then the appearance of the EGT indicator appears dimmed, indicating that the EGT indicator is not operational.

Position

`Position` — Location and size of EGT indicator

`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the EGT indicator relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the EGT indicator
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the EGT indicator
`width`	Distance between the right and left outer edges of the EGT indicator
`height`	Distance between the top and bottom outer edges of the EGT indicator

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[200 120 120 120]`

`InnerPosition` — Inner location and size of EGT indicator
`[100 100 120 120]` (default) | `[left bottom width height]`

Inner location and size of the EGT indicator, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of EGT indicator
`[100 100 120 120]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the EGT indicator returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an EGT indicator in the third row and second column of its parent grid.

```
g = ui([4 3]);
gauge = uiaeroegt(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the EGT indicator span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this EGT indicator spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function
`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function
`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

BusyAction — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

BeingDeleted — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uiaeroegt'`

This property is read-only.

Type of graphics object, returned as `'uiaeroegt'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also
`uiaeroegt`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

fganimation (Aero.FlightGearAnimation)

Construct FlightGear animation object

Syntax

```
h = fganimation
h = Aero.FlightGearAnimation
```

Description

`h = fganimation` and `h = Aero.FlightGearAnimation` construct a FlightGear animation object. The FlightGear animation object is returned to `h`.

Examples

Construct a FlightGear animation object, `h`:

```
h = fganimation
```

See Also

`Aero.FlightGearAnimation`

Introduced in R2007a

findstartstoptimes (Aero.Body)

Return start and stop times of time series data

Syntax

```
[tstart,tstop] = findstartstoptimes(h,tsdata)
[tstart,stop] = h.findstartstoptimes(tsdata)
```

Description

`[tstart,tstop] = findstartstoptimes(h,tsdata)` and `[tstart,stop] = h.findstartstoptimes(tsdata)` return the start and stop times of time series data `tsdata` for the animation body object `h`.

Examples

Find the start and stop times of the time series data, *`tsdata`*.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
tsdata = [ ...
    0,  1,1,1, 0,0,0; ...
    10  2,2,2, 1,1,1; ];
 b.TimeSeriesSource = tsdata;
[tstart,tstop] = findstartstoptimes(b,tsdata);
```

See Also

`load`

Introduced in R2007a

findstartstoptimes (Aero.Node)

Return start and stop times for time series data

Syntax

```
[tstart,tstop] = findstartstoptimes(h,tsdata)
[tstart,stop] = h.findstartstoptimes(tsdata)
```

Description

`[tstart,tstop] = findstartstoptimes(h,tsdata)` and `[tstart,stop] = h.findstartstoptimes(tsdata)` return the start and stop times of time series data `tsdata` for the virtual reality animation object `h`.

Examples

Find the start and stop times of the time series data, `takeoffData`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
load takeoffData;
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWith Time';
[tstart,stop]=h.Nodes{7}.findstartstoptimes;
```

Introduced in R2007b

flat2lla

Convert from flat Earth position to array of geodetic latitude, longitude, and altitude coordinates

Syntax

```
lla = flat2lla(flatearth_pos, llo, psio, href)
lla = flat2lla(flatearth_pos, llo, psio, href, ellipsoidModel)
lla = flat2lla(flatearth_pos, llo, psio, href, flattening,
equatorialRadius)
```

Description

`lla = flat2lla(flatearth_pos, llo, psio, href)` estimates an array of geodetic coordinates, *`lla`*, from an array of flat Earth coordinates, *`flatearth_pos`*. This function estimates the `lla` value with respect to a reference location that `llo`, `psio`, and `href` define.

`lla = flat2lla(flatearth_pos, llo, psio, href, ellipsoidModel)` estimates the coordinates for a specific ellipsoid planet.

`lla = flat2lla(flatearth_pos, llo, psio, href, flattening, equatorialRadius)` estimates the coordinates for a custom ellipsoid planet defined by `flattening` and `equatorialRadius`.

Input Arguments

`flatearth_pos`

Flat Earth position coordinates, in meters.

`llo`

Reference location, in degrees, of latitude and longitude, for the origin of the estimation and the origin of the flat Earth coordinate system.

psio

Angular direction of flat Earth *x*-axis (degrees clockwise from north), which is the angle in degrees used for converting flat Earth *x* and *y* coordinates to North and East coordinates.

href

Reference height from the surface of the Earth to the flat Earth frame with regard to the flat Earth frame, in meters.

ellipsoidModel

Specific ellipsoid planet model. This function supports only `'WGS84'`.

Default: WGS84

flattening

Custom ellipsoid planet defined by flattening.

equatorialRadius

Planetary equatorial radius, in meters.

Output Arguments

lla

m-by-3 array of geodetic coordinates (latitude, longitude, and altitude), in [degrees, degrees, meters].

Default:

Examples

Estimate latitude, longitude, and altitude at a specified coordinate:

```
lla = flat2lla( [ 4731 4511 120 ], [0 45], 5, -100)
```

```
lla =
    0.0391   45.0441  -20.0000
```

Estimate latitudes, longitudes, and altitudes at multiple coordinates, specifying the WGS84 ellipsoid model:

```
lla = flat2lla( [ 4731 4511 120; 0 5074 4498 ], [0 45], 5, -100, 'WGS84' )

lla =

  1.0e+003 *

    0.0000    0.0450   -0.0200
   -0.0000    0.0450   -4.3980
```

Estimate latitudes, longitudes, and altitudes at multiple coordinates, specifying a custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
lla = flat2lla( [ 4731 4511 120; 0 5074 4498 ], [0 45], 5, -100,  f, Re )

lla =

  1.0e+003 *

    0.0001    0.0451   -0.0200
   -0.0000    0.0451   -4.3980
```

Algorithms

The estimation begins by transforming the flat Earth *x* and *y* coordinates to North and East coordinates. The transformation has the form of

$$\begin{bmatrix} N \\ E \end{bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix}$$

where($\overline{\psi}$) is the angle in degrees clockwise between the *x*-axis and north.

To convert the North and East coordinates to geodetic latitude and longitude, the estimation uses the radius of curvature in the prime vertical (R_N) and the radius of curvature in the meridian (R_M). (R_N) and (R_M) are defined by the following relationships:

$$R_N = \frac{R}{\sqrt{1-(2f-f^2)\sin^2\mu_0}}$$

$$R_M = R_N\frac{1-(2f-f^2)}{1-(2f-f^2)\sin^2\mu_0}$$

where (R) is the equatorial radius of the planet and$\left(\bar{f}\right)$ is the flattening of the planet.

Small changes in the latitude and longitude are approximated from small changes in the North and East positions by

$$d\mu = \mathrm{atan}\left(\frac{1}{R_M}\right)dN$$

$$d\iota = \mathrm{atan}\left(\frac{1}{R_N\cos\mu}\right)dE$$

The output latitude and longitude are the initial latitude and longitude plus the small changes in latitude and longitude.

$$\mu = \mu_0 + d\mu$$

$$\iota = \iota_0 + d\iota$$

The altitude is the negative flat Earth z-axis value minus the reference height (h_{ref}).

$$h = -p_z - h_{ref}$$

References

Etkin, B., *Dynamics of Atmospheric Flight*. New York: John Wiley & Sons, 1972.

Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, 2nd ed. New York: John Wiley & Sons, 2003.

See Also

`lla2flat`

Introduced in R2011a

flowfanno

Fanno line flow relations

Syntax

[*mach*, *T*, *P*, *rho*, *velocity*, *P0*, *fanno*] = flowfanno(*gamma*, *fanno_flow*, *mtype*)

Description

[*mach*, *T*, *P*, *rho*, *velocity*, *P0*, *fanno*] = flowfanno(*gamma*, *fanno_flow*, *mtype*) returns an array for each Fanno line flow relation. This function calculates the arrays for a given set of specific heat ratios (*gamma*), and any one of the Fanno flow types. You select the Fanno flow type with *mtype*.

This function uses Fanno variables given by the following. *F* is the Fanno parameter given by F = f*L/D. *f* is the friction coefficient. *L* is the length of constant area duct required to achieve sonic flow. *D* is the hydraulic diameter of the duct.

This function assumes that variables vary in one dimension only. It also assumes that the main mechanism for the change of flow variables is the change of cross-sectional area of the flow stream tubes.

If the temperature experiences large fluctuations, the perfect gas assumption might be invalid. If the stagnation temperature is above 1500 K, do not assume constant specific heats. In this case, the medium ceases to be a calorically perfect gas. Consider it a thermally perfect gas. See 2 for thermally perfect gas correction factors. If the temperature is so high that molecules dissociate and ionize (static temperature 5000 K for air), you cannot assume a perfect gas.

Input Arguments

gamma

Array of *N* specific heat ratios. *gamma* must be either a scalar or an array of *N* real numbers greater than 1. *gamma* must be a real, finite scalar greater than 1 for the following input modes: subsonic total pressure ratio, supersonic total pressure ratio, subsonic Fanno parameter, and supersonic Fanno parameter.

`fanno_flow`

Array of real numerical values for one Fanno flow. This argument can be one of the following:

- Array of Mach numbers. *`flow_fanno`* must be a scalar or an array of *`N`* real numbers greater than or equal to 0. If *`flow_fanno`* and *`gamma`* are arrays, they must be the same size.

  Use *`flow_fanno`* with the *`mtype`* value `'mach'`. Because *`'mach'`* is the default of *`mtype`*, *`mtype`* is optional when this array is the input mode.
- Array of temperature ratios. The temperature ratio is the local static temperature over the reference static temperature for sonic flow. This array must be a scalar or array of *`N`* real numbers:

  - Greater than or equal to 0 (as the Mach number approaches infinity)
  - Less than or equal to (*`gamma`*+1)/2 (at Mach number equal 0)

  Use *`flow_fanno`* with *`mtype`* value *`'temp'`*.
- Array of pressure ratios. The pressure ratio is the local static pressure over the reference static pressure for sonic flow. *`flow_fanno`* must be a scalar or array of real numbers greater than or equal to 0. If *`flow_fanno`* and *`gamma`* are arrays, they must be the same size.

  Use *`flow_fanno`* with *`mtype`* value *`'pres'`*.
- Array of density ratios. The density ratio is the local density over the reference density for sonic flow. *`flow_fanno`* must be a scalar or array of real numbers. These numbers must be greater than or equal to:

  sqrt((*`gamma`*-1)/(*`gamma`*+1)) (as the Mach number approaches infinity).

  If *`flow_fanno`* and *`gamma`* are arrays, they must be the same size.

  Use *`flow_fanno`* with *`mtype`* value *`'dens'`*.
- Array of velocity ratios. The velocity ratio is the local velocity over the reference velocity for sonic flow. *`flow_fanno`* must be a scalar or an array of *`N`* of real numbers:

  - Greater than or equal to 0
  - Less than or equal to sqrt((*gamma*+1)/(*gamma*-1)) (as the Mach number approaches infinity)

  If *flow_fanno* and *gamma* are both arrays, they must be the same size.

  Use *flow_fanno* with *mtype* value '*velo*'.

- Scalar value of total pressure ratio. The total pressure ratio is the local total pressure over the reference total pressure for sonic flow. *flow_fanno* must be greater than or equal to 1.

  Use *flow_fanno* with *mtype* values `'totalp'` and `'totalpsup'`.

- Scalar value for Fanno parameter. The Fanno parameter is flow_fanno= f*L/D. *f* is the friction coefficient. *L* is the length of constant area duct required to achieve sonic flow. *D* is the hydraulic diameter of the duct. In subsonic mode, *flow_fanno* must be greater than or equal to 0. In supersonic mode, *flow_fanno* must be:

  - Greater than or equal to 0 (at Mach number equal 1)
  - Less than or equal to (*gamma*+1)/(2**gamma*)*log((*gamma*+1)/(*gamma*-1))-1/*gamma* (as Mach number approaches infinity)

  Use *flow_fanno* with *mtype* values '*fannosub*' and '*fannosup*'.

mtype

Input mode for the type of Fanno flow in *fanno_flow*.

Type	Description
`'mach'`	Default Mach number
`'temp'`	Temperature ratio
`'pres'`	Pressure ratio
`'dens'`	Density ratio
`'velo'`	Velocity ratio
`'totalpsub'`	Subsonic total pressure ratio
`'totalpsup'`	Supersonic total pressure ratio
`'fannosub'`	Subsonic Fanno parameter

Type	Description
`'fannosup'`	Supersonic Fanno parameter

Output Arguments

All outputs are the same size as the array inputs. If there are no array inputs, all outputs are scalars.

`mach`

Array of Mach numbers.

`T`

Array of temperature ratios. The temperature ratio is the local static temperature over the reference static temperature for sonic flow.

`P`

Array of pressure ratios. The pressure ratio is the local static pressure over the reference static pressure for sonic flow.

`rho`

Array of density ratio. The density ratio is the local density over the reference density for sonic flow.

`velocity`

Array of velocity ratios. The velocity ratio is the local velocity over the reference velocity for sonic flow.

`P0`

Array of stagnation (total) pressure ratio. The total pressure ratio is the local total pressure over the reference total pressure for sonic flow.

`fanno`

Array of Fanno parameters. The Fanno parameter is F = f*L/D. *`f`* is the friction coefficient. *`L`* is the length of constant area duct required to achieve sonic flow. *`D`* is the hydraulic diameter of the duct.

Examples

Calculate the Fanno line flow relations for air (`gamma` = 1.4) for subsonic Fanno parameter 1.2. The following returns scalar values for *`mach`*, *`T`*, *`P`*, *`rho`*, *`velocity`*, *`P0`*, and *`fanno`*.

```
[mach, T, P, rho, velocity, P0, fanno] = flowfanno(1.4, 1.2, 'fannosub')
```

Calculate the Fanno line flow relations for gases with specific heat ratios given in the following 1 x 4 row array for the Mach number 0.5. The following yields a 1 x 4 row array for *`mach`*, *`T`*, *`P`*, *`rho`*, *`velocity`*, *`P0`*, and *`fanno`*.

```
gamma = [1.3, 1.33, 1.4, 1.67];
[mach, T, P, rho, velocity, P0, fanno] = flowfanno(gamma, 0.5)
```

Calculate the Fanno line flow relations for a specific heat ratio of 1.4 and range of temperature ratios from 0.40 to 0.70 in increments of 0.10. The following returns a 4 x 1 column array for *`mach`*, *`T`*, *`P`*, *`rho`*, *`velocity`*, *`P0`*, and *`fanno`*.

```
[mach, T, P, rho, velocity, P0, fanno] = flowfanno(1.4, [1.1 1.2], 'temp')
```

Calculate the Fanno line flow relations for gases with specific heat ratio and velocity ratio combinations as shown. The following returns a 1 x 2 array for *`mach`*, *`T`*, *`P`*, *`rho`*, *`velocity`*, *`P0`*, and *`fanno`* each. The elements of each array correspond to the inputs element-wise.

```
gamma = [1.3, 1.4];
V = [0.53, 0.49];
[MACH, T, P, RHO, V, P0, F] = flowfanno(gamma, V, 'velo')
```

References

1. James, J. E. A., *Gas Dynamics, Second Edition*, Allyn and Bacon, Inc, Boston, 1984.

2. *NACA Technical Report 1135*, 1953, National Advisory Committee on Aeronautics, Ames Research Staff, Moffett Field, Calif. Pages 667–671.

See Also

`flowisentropic` | `flownormalshock` | `flowprandtlmeyer` | `flowrayleigh`

Introduced in R2010a

flowisentropic

Isentropic flow ratios

Syntax

`[mach, T, P, rho, area] = flowisentropic(gamma, flow, mtype)`

Description

`[mach, T, P, rho, area] = flowisentropic(gamma, flow, mtype)` returns an array. This array contains an isentropic flow Mach number (*mach*), temperature ratio (*T*), pressure ratio (*P*), density ratio (*rho*), and area ratio (*area*). This function calculates these arrays given a set of specific heat ratios (*gamma*), and any one of the isentropic flow types. You select the isentropic flow with *mtype*.

This function assumes that variables vary in one dimension only. It also assumes that the main mechanism for the change of flow variables is the change of cross-sectional area of the flow stream tubes.

This function assumes that the environment is a perfect gas. In the following instances, the function cannot assume a perfect gas environment. If there is a large change in either temperature or pressure without a proportionally large change in the other, the function cannot assume a perfect gas environment. . If the stagnation temperature is above 1500 K, do not assume that constant specific heats. In this case, the medium ceases to be a calorically perfect gas. Consider it a thermally perfect gas. See 2 for thermally perfect gas correction factors. If the temperature is so high that molecules dissociate and ionize (static temperature 5000 K for air), you cannot assume a calorically or thermally perfect gas.

Input Arguments

`gamma`

Array of *`N`* specific heat ratios. *`gamma`* must be a scalar or array of *`N`* real numbers greater than 1. For subsonic area ratio input mode and supersonic area ratio input mode, *`gamma`* must be a real, finite scalar greater than 1.

`flow`

Array of real numerical values for one of the isentropic flow relations. This argument can be one of the following:

- Array of Mach numbers. *`flow`* must be a scalar or an array of *`N`* real numbers greater than or equal to 0. If *`flow`* and *`gamma`* are arrays, they must be the same size.

  Use *`flow`* with the *`mtype`* value `'mach'`. Because *`'mach'`* is the default of *`mtype`*, *`mtype`* is optional when this array is the input mode.
- Array of temperature ratios. The temperature ratio is the local static temperature over the stagnation temperature. *`flow`* must be a scalar or an array of real numbers:

  - Greater than or equal to 0 (as the Mach number approaches infinity)
  - Less than or equal to 1 (at Mach number equal 0)

  If *`flow`* and *`gamma`* are both arrays, they must be the same size.

  Use *`flow`* with *`mtype`* value *`'temp'`*.
- Array of pressure ratios. The pressure ratio is the local static pressure over the stagnation pressure. *`flow`* must be a scalar or an array of real numbers:

  - Greater than or equal to 0 (as the Mach number approaches infinity)
  - Less than or equal to 1 (at Mach number equal 0)

  If *`flow`* and *`gamma`* are both arrays, they must be the same size.

  Use *`flow`* with *`mtype`* value *`'pres'`*.
- Array of density ratios. The density ratio is the local density over the stagnation density. *`flow`* must be a scalar or an array of real numbers:

  - Greater than or equal to 0 (as the Mach number approaches infinity)

  - Less than or equal to 1 (at Mach number equal 0)

  If *`flow`* and *`gamma`* are both arrays, they must be the same size.

  Use *`flow`* with *`mtype`* value `'`*`dens`*`'`.

- Scalar value of area ratio. *`flow`* must be a real value greater than or equal to 1.

  Use *`flow`* with *`mtype`* value `'`*`sup`*`'`.

`mtype`

Input mode for the isentropic flow in *`flow`*.

Type	Description
`'mach'`	Default. Mach number.
`'temp'`	Temperature ratio.
`'pres'`	Pressure ratio.
`'dens'`	Density ratio.
`'sub'`	Subsonic area ratio. The subsonic area ratio is the local subsonic stream tube area over the reference stream tube area for sonic conditions.
`'sup'`	Supersonic area ratio. The supersonic area ratio is the local supersonic stream tube area over the reference stream tube area for sonic conditions.

Output Arguments

All outputs are the same size as the array inputs. If there are no array inputs, all outputs are scalars.

`mach`

Array of Mach numbers.

`T`

Array of temperature ratios. The temperature ratio is the local static temperature over the stagnation temperature.

P

Array of pressure ratios. The pressure ratio is the local static pressure over the stagnation pressure.

rho

Array of density ratios. The density ratio is the local density over the stagnation density.

area

Array of area ratios. The area ratio is the local stream tube area over the reference stream tube area for sonic conditions.

Examples

Calculate the isentropic flow relations for air (*gamma* = 1.4) for a design subsonic area ratio of 1.255. This example returns scalar values for *mach*, *T*, *P*, *rho*, and *area*.

```
  [mach, T, P, rho, area] = flowisentropic(1.4, 1.255, 'sub')
```

Calculate the isentropic flow relations for gases with specific heat ratios given in the following 1 x 4 row array for the Mach number 0.5. This example following returns a 1 x 4 row array for *mach*, *T*, *P*, *rho*, and *area*.

```
gamma = [1.3, 1.33, 1.4, 1.67];
[mach, T, P, rho, area] = flowisentropic(gamma, 0.5)
```

Calculate the isentropic flow relations for a specific heat ratio of 1.4. Also calculate range of temperature ratios from 0.40 to 0.70 in increments of 0.10. This example returns a 4 x 1 column array for *mach*, *T*, *P*, *rho*, and *area*.

```
[mach, T, P, rho, area] = flowisentropic(1.4, (0.40:0.10:0.70)', 'temp')
```

Calculate the isentropic flow relations for gases with provided specific heat ratio and density ratio combinations. This example returns a 1 x 2 array for *mach*, *T*, *P*, *rho*, and *area* each. The elements of each vector correspond to the inputs element-wise.

```
gamma = [1.3, 1.4];
rho = [0.13, 0.9];
[mach, T, P, rho, area] = flowisentropic(gamma, rho , 'dens')
```

References

1. James, J. E. A., *Gas Dynamics, Second Edition*, Allyn and Bacon, Inc, Boston, 1984.

2. *NACA Technical Report 1135*, 1953, National Advisory Committee on Aeronautics, Ames Research Staff, Moffett Field, Calif. Pages 667–671.

See Also

`flowfanno` | `flownormalshock` | `flowprandtlmeyer` | `flowrayleigh`

Introduced in R2010a

flownormalshock

Normal shock relations

Syntax

`[`*`mach`*`, `*`T`*`, `*`P`*`, `*`rho`*`, `*`downstream_mach`*`, `*`P0`*`, `*`P1`*`] = flownormalshock(`*`gamma`*`, `*`normal_shock_relations`*`, `*`mtype`*`)`

Description

`[`*`mach`*`, `*`T`*`, `*`P`*`, `*`rho`*`, `*`downstream_mach`*`, `*`P0`*`, `*`P1`*`] = flownormalshock(`*`gamma`*`, `*`normal_shock_relations`*`, `*`mtype`*`)` produces an array for each normal shock relation (*`normal_shock_relations`*). This function calculates these arrays for a given set of specific heat ratios (*`gamma`*) and any one of the normal shock relations (*`normal_shock_relations`*). *`mtype`* selects the normal shock relations that *`normal_shock_relations`* represents. All ratios are downstream value over upstream value. Consider upstream to be before or ahead of the shock; downstream is after or behind the shock.

This function assumes that the medium is a calorically perfect gas. It assumes that the flow is frictionless and adiabatic. It assumes that the flow variables vary in one dimension only. It assumes that the main mechanism for the change of flow variables is the change of cross-sectional area of the flow stream tubes.

If the temperature experiences large fluctuations, the perfect gas assumption might be invalid. If the stagnation temperature is above 1500 K, do not assume constant specific heats. In this case, the medium ceases to be a calorically perfect gas. You must then consider it a thermally perfect gas. See 2 for thermally perfect gas correction factors. If the temperature is so high that molecules dissociate and ionize (static temperature 5000 K for air), you cannot assume a perfect gas.

Input Arguments

`gamma`

Array of *N* specific heat ratios. *`gamma`* must be either a scalar or an array of *N* real numbers greater than 1. For temperature ratio, total pressure ratio, and Rayleigh-Pitot ratio input modes, *`gamma`* must be a real, finite scalar greater than 1.

`normal_shock_relations`

Array of real numerical values for one of the normal shock relations. This argument can be one of the following:

- Array of upstream Mach numbers. This array must be a scalar or an array of *N* real numbers greater than or equal to 1. If *`normal_shock_relations`* and *`gamma`* are arrays, they must be the same size.

  Use *`normal_shock_relations`* with *`mtype`* value `'mach'`. Because `'`*`mach`*`'` is the default of *`mtype`*, *`mtype`* is optional when this array is the input mode.
- Scalar value of temperature ratio. The temperature ratio is the static temperature downstream of the shock over the static temperature upstream of the shock. *`normal_shock_relations`* must be a real scalar greater than or equal to 1.

  Use *`normal_shock_relations`* with *`mtype`* value `'`*`temp`*`'`.
- Array of pressure ratios. The pressure ratio is the static pressure downstream of the shock over the static pressure upstream of the shock. *`normal_shock_relations`* must be a scalar or array of real numbers greater than or equal to 1. If *`normal_shock_relations`* and *`gamma`* are arrays, they must be the same size.

  Use *`normal_shock_relations`* with *`mtype`* value `'`*`pres`*`'`.
- Array of density ratios. The density ratio is the density of the fluid downstream of the shock over the density upstream of the shock. *`normal_shock_relations`* must a scalar or array of real numbers be:

  - Greater than or equal to 1 (at Mach number equal 1)
  - Less than or equal to (*`gamma`*+1)/(*`gamma`*-1) (as the Mach number approaches infinity)

  If *`normal_shock_relations`* and *`gamma`* are arrays, they must be the same size. Use *`normal_shock_relations`* with *`mtype`* value `'`*`dens`*`'`.

- Array of downstream Mach numbers. *`normal_shock_relations`* must be scalar or array of real numbers:

  - Greater than or equal to 0 (as the Mach number approaches infinity)
  - Less than or equal to sqrt((*gamma*-1)/(2**gamma*)) (at Mach number equal 1)

  If *`normal_shock_relations`* and *`gamma`* are arrays, they must be the same size. Use *`normal_shock_relations`* with *`mtype`* value *`'down'`*.
- Scalar value of total pressure ratio. The total pressure ratio is the total pressure downstream of the shock over the total pressure upstream of the shock. *`normal_shock_relations`* must be:

  - Greater than or equal to 0 (as the Mach number approaches infinity)
  - Less than or equal to 1 (at Mach number equal 1)

  If *`normal_shock_relations`* and *`gamma`* are both arrays, they must be the same size. Use *`normal_shock_relations`* with *`mtype`* value `'totalp'`.
- Scalar value of Rayleigh-Pitot ratio. The Rayleigh-Pitot ratio is the static pressure upstream of the shock over the total pressure downstream of the shock. *`normal_shock_relations`* must be:

  - Real scalar greater than or equal to 0 (as the Mach number approaches infinity)
  - Less than or equal to ((*gamma*+1)/2)^(-*gamma*/(*gamma*-1)) (at Mach number equal 1)

  If *`normal_shock_relations`* and *`gamma`* are both arrays, they must be the same size. Use *`normal_shock_relations`* with *`mtype`* value `'pito'`.

`mtype`

Input mode for the normal shock relations in *`normal_shock_relations`*, specified as a character vector or string.

Type	**Description**
`'mach'`	Default. Mach number.
`'temp'`	Temperature ratio.
`'pres'`	Pressure ratio.
`'dens'`	Density ratio.

Type	Description
`'down'`	Downstream Mach number.
`'totalp'`	Total pressure ratio.
`'pito'`	Rayleigh-Pitot ratio.

Output Arguments

`mach`

Array of upstream Mach numbers.

`P`

Array of pressure ratios. The pressure ratio is the static pressure downstream of the shock over the static pressure upstream of the shock.

`T`

Array of temperature ratios. The temperature ratio is the static temperature downstream of the shock over the static temperature upstream of the shock.

`rho`

Array of density ratios. The density ratio is the density of the fluid downstream of the shock over the density upstream of the shock.

`downstream_mach`

Array of downstream Mach numbers.

`P0`

Array of total pressure ratios. The total pressure ratio is the total pressure downstream of the shock over the total pressure upstream of the shock.

`P1`

Array of Rayleigh-Pitot ratios. The Rayleigh-Pitot ratio is the static pressure upstream of the shock over the total pressure downstream of the shock.

Examples

Calculate the normal shock relations for air (*gamma* = 1.4) for total pressure ratio of 0.61. The following returns scalar values for *mach*, *T*, *P*, *rho*, *downstream_mach*, *P0*, and *P1*.

```
[mach, T, P, rho, downstream_mach, P0, P1] = flownormalshock(1.4, 0.61, 'totalp')
```

Calculate the normal shock relations for gases with specific heat ratios given in the following 1 x 4 row array for upstream Mach number 1.5. The follow yields a 1 x 4 array for *mach*, *T*, *P*, *rho*, *downstream_mach*, *P0*, and *P1*.

```
gamma = [1.3, 1.33, 1.4, 1.67];
[mach, T, P, rho, downstream_mach, P0, P1] = flownormalshock(gamma, 1.5)
```

Calculate the normal shock relations for a specific heat ratio of 1.4 and range of density ratios from 2.40 to 2.70 in increments of 0.10. The following returns a 4 x 1 column array for *mach*, *T*, *P*, *rho*, *downstream_mach*, *P0*, and *P1*.

```
[mach, T, P, rho, downstream_mach, P0, P1] = flownormalshock(1.4,...
 (2.4:.1:2.7)', 'dens')
```

Calculate the normal shock relations for gases with specific heat ratio and downstream Mach number combinations as shown. The following example returns a 1 x 2 array for *mach*, *T*, *P*, *rho*, *downstream_mach*, *P0*, and *P1* each, where the elements of each vector corresponds to the inputs element-wise.

```
gamma = [1.3, 1.4];
downstream_mach = [.34, .49];
[mach, T, P, rho, downstream_mach, P0, P1] = flownormalshock(gamma,...
 downstream_mach, 'down')
```

References

1. James, J. E. A., *Gas Dynamics, Second Edition*, Allyn and Bacon, Inc, Boston, 1984.

2. *NACA Technical Report 1135*, 1953, National Advisory Committee on Aeronautics, Ames Research Staff, Moffett Field, Calif. Pages 667–671.

See Also

`flowfanno` | `flowisentropic` | `flowprandtlmeyer` | `flowrayleigh`

Introduced in R2010a

flowprandtlmeyer

Calculate Prandtl-Meyer functions for expansion waves

Syntax

```
[mach, nu, mu] = flowprandtlmeyer(gamma, prandtlmeyer_array, mtype)
```

Description

`[mach, nu, mu] = flowprandtlmeyer(gamma, prandtlmeyer_array, mtype)` calculates the following: array of Mach numbers, *`mach`*, Prandtl-Meyer angles (*`nu`* in degrees) and Mach angles (*`mu`* in degrees). `flowprandtlmeyer` calculates these arrays for a given set of specific heat ratios, *`gamma`*, and any one of the Prandtl-Meyer types. You select the Prandtl-Meyer type with *`mtype`*.

The function assumes that the flow is two-dimensional. The function also assumes a smooth and gradual change in flow properties through the expansion fan.

Note, this function assumes that the environment is a perfect gas. In the following instances, it cannot assume a perfect gas environment. If there is a large change in either temperature or pressure without a proportionally large change in the other, it cannot assume a perfect gas environment. If the stagnation temperature is above 1500 K, the function cannot assume constant specific heats. In this case, you must consider it a thermally perfect gas. See 2 for thermally perfect gas correction factors. The local static temperature might be so high that molecules dissociate and ionize (static temperature 5000 K for air). In this case, you cannot assume a calorically or thermally perfect gas.

Input Arguments

gamma

Array of *`N`* specific heat ratios. *`gamma`* must be a scalar or array of *`N`* real numbers greater than 1. For subsonic area ratio input mode and supersonic area ratio input mode, *`gamma`* must be a real, finite scalar greater than 1.

`prandtlmeyer_array`

Array of real numerical values for one of the Prandtl-Meyer types. This argument can be one of the following:

- Array of Mach numbers. This array must be a scalar or an array of *`N`* real numbers greater than or equal to 0. If *`prandtlmeyer_array`* and *`gamma`* are arrays, they must be the same size.

  Use *`prandtlmeyer_array`* with *`mtype`* value `'mach'`. Note, because `'mach'` is the default of *`mtype`*, *`mtype`* is optional when this array is the input mode.
- Scalar value for Prandtl-Meyer angle in degrees. This value is the angle change required for a Mach 1 flow to achieve a given Mach number after expansion. *`prandtlmeyer_array`* must be:

  - Real scalar greater than or equal to 0 (at Mach number equal 1)
  - Less than or equal to 90 * (sqrt((*gamma*+1)/(*gamma*-1)) - 1) (as the Mach number approaches infinity).

  Use *`prandtlmeyer_array`* with *`mtype`* value `'nu'`.
- Array of Mach angles in degrees. These values are the angles between the flow direction and the lines of pressure disturbance caused by supersonic motion. The Mach angle is a function of Mach number only. *`prandtlmeyer_array`* must be a scalar or array of *`N`* real numbers that are:

  - Greater than or equal to 0 (as the Mach number approaches infinity).
  - Less than or equal to 90 (at Mach number equal 1).

  Use *`prandtlmeyer_array`* with *`mtype`* value `'mu'`.

`mtype`

Isentropic flow variable represented by *`prandtlmeyer_array`*.

Type	Description
`'mach'`	Default. Mach number..
`'nu'`	Prandtl-Meyer angle
`'mu'`	Mach angle.

Output Arguments

`mach`

Array of Mach numbers. In Prandtl-Meyer angle input mode, *`mach`* outputs are the same size as the array input or array inputs. If there are no array inputs, *`mach`* is a scalar.

`nu`

Array of Prandtl-Meyer angles. The Prandtl-Meyer angle is the angle change required for a Mach 1 flow to achieve a given Mach number after expansion.

`mu`

Array of Mach angles. The Mach angle is between the flow direction and the lines of pressure disturbance caused by supersonic motion.

Examples

Calculate the Prandtl-Meyer relations for air (*`gamma`* = 1.4) for Prandtl-Meyer angle 61 degrees. The following returns a scalar for *`mach`*, *`nu`*, and *`mu`*.

```
[mach, nu, mu] = flowprandtlmeyer(1.4, 61, 'nu')
```

Calculate the Prandtl-Meyer functions for gases with specific heat ratios. The following yields a 1 x 4 array for *`nu`*, but only a scalar for *`mach`* and *`mu`*.

```
gamma = [1.3, 1.33, 1.4, 1.67];
[mach, nu, mu] = flowprandtlmeyer(gamma, 1.5)
```

Calculate the Prandtl-Meyer angles for a specific heat ratio of 1.4 and range of Mach angles from 40 degrees to 70 degrees. This example uses increments of 10 degrees. The following returns a 4 x 1 column array for *`mach`*, *`nu`*, and *`mu`*.

```
[mach, nu, mu] = flowprandtlmeyer(1.4, (40:10:70)', 'mu')
```

Calculate the Prandtl-Meyer relations for gases with specific heat ratio and Mach number combinations as shown. The following returns a 1 x 2 array for *`nu`* and *`mu`* each, where the elements of each vector correspond to the inputs element-wise.

```
gamma = [1.3, 1.4];
prandtlmeyer_array = [1.13, 9];
[mach, nu, mu] = flowprandtlmeyer(gamma,prandtlmeyer_array)
```

References

1. James, J. E. A., *Gas Dynamics, Second Edition*, Allyn and Bacon, Inc, Boston, 1984.

2. *NACA Technical Report 1135*, 1953, National Advisory Committee on Aeronautics, Ames Research Staff, Moffett Field, Calif. Pages 667–671.

See Also

`flowfanno` | `flowisentropic` | `flownormalshock` | `flowrayleigh`

Introduced in R2010a

flowrayleigh

Rayleigh line flow relations

Syntax

```
[mach, T, P, rho, velocity, T0, P0] = flowrayleigh(gamma,
rayleigh_flow, mtype)
```

Description

`[mach, T, P, rho, velocity, T0, P0] = flowrayleigh(gamma, rayleigh_flow, mtype)` returns an array for each Rayleigh line flow relation. This function calculates these arrays for a given set of specific heat ratios (*gamma*), and any one of the Rayleigh line flow types. You select the Rayleigh flow type with *mtype*.

This function assumes that the medium is a calorically perfect gas in a constant area duct. It assumes that the flow is steady, frictionless, and one dimensional. It also assumes that the main mechanism for the change of flow variables is heat transfer.

This function assumes that the environment is a perfect gas. In the following instances, it cannot assume a perfect gas environment. If there is a large change in either temperature or pressure without a proportionally large change in the other, it cannot assume a perfect gas environment. If the stagnation temperature is above 1500 K, do not assume constant specific heats. In this case, the medium ceases to be a calorically perfect gas; you must then consider it a thermally perfect gas. See 2 for thermally perfect gas correction factors. The local static temperature might be so high that molecules dissociate and ionize (static temperature 5000 K for air). In this case, you cannot assume a calorically or thermally perfect gas.

Input Arguments

gamma

Array of *N* specific heat ratios. *gamma* must be either a scalar or an array of *N* real numbers greater than 1. *gamma* must be a real, finite scalar greater than 1 for the following input modes: low speed temperature ratio, high speed temperature ratio, subsonic total temperature, supersonic total temperature, subsonic total pressure, and supersonic total pressure.

`rayleigh_flow`

Array of real numerical values for one Rayleigh line flow. This argument can be one of the following:

- Array of Mach numbers. This array must be a scalar or an array of *`N`* real numbers greater than or equal to 0. If *`rayleigh_flow`* and *`gamma`* are arrays, they must be the same size.

  Use *`rayleigh_flow`* with *`mtype`* value `'mach'`. Because `'`*`mach`*`'` is the default of *`mtype`*, *`mtype`* is optional when this array is the input mode.
- Scalar value of temperature ratio. The temperature ratio is the local static temperature over the reference static temperature for sonic flow. *`rayleigh_flow`* must be a real scalar:

  - Greater than or equal to 0 (at the Mach number equal 0 for low speeds or as Mach number approaches infinity for high speeds)
  - Less than or equal to 1/4*(*`gamma`*+1/*`gamma`*)+1/2 (at *`mach`* = 1/sqrt(*`gamma`*))

  Use *`rayleigh_flow`* with *`mtype`* values `'`*`templo`*`'` and `'`*`temphi`*`'`.
- Array of pressure ratios. The pressure ratio is the local static pressure over the reference static pressure for sonic flow. *`rayleigh_flow`* must be a scalar or array of real numbers less than or equal to *`gamma`*+1 (at the Mach number equal 0). If *`rayleigh_flow`* and *`gamma`* are arrays, they must be the same size.

  Use *`rayleigh_flow`* with *`mtype`* value `'`*`pres`*`'`.
- Array of density ratios. The density ratio is the local density over the reference density for sonic flow. *`rayleigh_flow`* must be a scalar or array of real numbers. These numbers must be greater than or equal to:

  `gamma`/(*`gamma`*+1) (as Mach number approaches infinity)

  If *`rayleigh_flow`* and *`gamma`* are arrays, they must be the same size.

  Use *`rayleigh_flow`* with *`mtype`* value `'`*`dens`*`'`.

- Array of velocity ratios. The velocity ratio is the local velocity over the reference velocity for sonic flow. *rayleigh_flow* must be a scalar or an array of *N* real numbers:

  - Greater than or equal to 0
  - Less than or equal to (*gamma*+1)/*gamma* (as Mach number approaches infinity)

  If *rayleigh_flow* and *gamma* are both arrays, they must be the same size.

  Use *rayleigh_flow* with *mtype* value `'velo'`.
- Scalar value of total temperature ratio. The total temperature ratio is the local stagnation temperature over the reference stagnation temperature for sonic flow. In subsonic mode, *rayleigh_flow* must be a real scalar:

  - Greater than or equal to 0 (at the Mach number equal 0)
  - Less than or equal to 1 (at the Mach number equal 1)

  In supersonic mode, *rayleigh_flow* must be a real scalar:

  - Greater than or equal to (*gamma*+1)^2*(*gamma*-1)/2/(*gamma*^2*(1+(*gamma*-1)/2))) (as Mach number approaches infinity)
  - Less than or equal to 1 (at the Mach number equal 1)

  Use *rayleigh_flow* with the *mtype* values `'totaltsub'` and `'totaltsup'`.
- Scalar value of total pressure ratio. The total pressure ratio is the local stagnation pressure over the reference stagnation pressure for sonic flow. In subsonic mode, *rayleigh_flow* must be a real scalar.

  - Greater than or equal to 1 (at the Mach number equal 1)
  - Less than or equal to (1+*gamma*)*(1+(*gamma*-1)/2)^(-*gamma*/(*gamma*-1)) (at Mach number equal 0)

  In supersonic mode, *rayleigh_flow* must be a real scalar greater than or equal to 1.

  Use *rayleigh_flow* with *mtype* values `'totalpsub'` and `'totalpsup'`.

`mtype`

Input mode for the Rayleigh flow in *rayleigh_flow*.

Type	Description
`'mach'`	Default. Mach number.
`'templo'`	Low speed static temperature ratio. The low speed temperature ratio is the local static temperature over the reference sonic temperature. This ratio for when the Mach number of the upstream flow is less than the critical Mach number of 1/sqrt(*gamma*).
`'temphi'`	High speed static temperature ratio. The high speed temperature ratio is the local static temperature over the reference sonic temperature. This ratio is for when the Mach number of the upstream flow is greater than the critical Mach number of 1/sqrt(*gamma*).
`'pres'`	Pressure ratio.
`'dens'`	Density ratio.
`'velo'`	Velocity ratio.
`'totaltsub'`	Subsonic total temperature ratio.
`'totaltsup'`	Supersonic total temperature ratio.
`'totalpsub'`	Subsonic total pressure ratio.
`'totalpsup'`	Supersonic total pressure ratio.

Output Arguments

All output ratios are static conditions over the sonic conditions. All outputs are the same size as the array inputs. If there are no array inputs, all outputs are scalars.

`mach`

Array of Mach numbers.

`T`

Array of temperature ratios. The temperature ratio is the local static temperature over the reference static temperature for sonic flow.

P

Array of pressure ratios. The pressure ratio is the local static pressure over the reference static pressure for sonic flow.

rho

Array of density ratio. The density ratio is the local density over the reference density for sonic flow.

velocity

Array of velocity ratios. The velocity ratio is the local velocity over the reference velocity for sonic flow.

T0

Array of total temperature ratios. The temperature ratio is the local stagnation temperature over the reference stagnation temperature for sonic flow.

P0

Array of total pressure ratios. The total pressure ratio is the local stagnation pressure over the reference stagnation pressure for sonic flow.

Examples

Calculate Rayleigh Line Flow Relations Given Air

Calculate the Rayleigh line flow relations for air (*gamma* = 1.4) for supersonic total pressure ratio 1.2.

```
[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(1.4,1.2,'totalpsup')
```

```
mach =

    1.6397

T =

    0.6823
```

```
P =

    0.5038

rho =

    0.7383

velocity =

    1.3545

T0 =

    0.8744

P0 =

    1.2000
```

This example returns scalar values for *`mach`*, *`T`*, *`P`*, *`rho`*, *`velocity`*, *`T0`*, and *`P0`*.

Calculate Rayleigh Line Flow Relations for Specific Heat Ratios in Array

Calculate the Rayleigh line flow relations for gases with specific heat ratios given in the following 1 x 4 row array for the Mach number 0.5.

```
gamma = [1.3,1.33,1.4,1.67];
[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(gamma,0.5)
```

```
mach =

    0.5000    0.5000    0.5000    0.5000

T =

    0.7533    0.7644    0.7901    0.8870

P =

    1.7358    1.7486    1.7778    1.8836
```

```
rho =

    2.3043    2.2876    2.2500    2.1236

velocity =

    0.4340    0.4371    0.4444    0.4709

T0 =

    0.6796    0.6832    0.6914    0.7201

P0 =

    1.1111    1.1121    1.1141    1.1202
```

This example returns a 1 x 4 row array for *`mach`*, *`T`*, *`P`*, *`rho`*, *`velocity`*, *`T0`*, and *`P0`*.

Calculate Rayleigh Line Flow Relations for Specific Heat Ratios and High Speed Temperature

Calculate the Rayleigh line flow relations for a specific heat ratio of 1.4 and high speed temperature ratio 0.70.

```
[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(1.4,0.70,'temphi')
```

```
mach =

    1.6035

T =

    0.7000

P =

    0.5218

rho =

    0.7454

velocity =
    1.3416

T0 =

    0.8833

P0 =

    1.1777
```

This example returns scalar values for *mach*, *T*, *P*, *rho*, *velocity*, *T0*, and *P0*.

Calculate Rayleigh Line Flow Relations for Gases with Specific Heat Ratio and Static Pressure

Calculate the Rayleigh line flow relations for gases with specific heat ratio and static pressure ratio combinations as shown.

```
gamma = [1.3,1.4];
P = [0.13,1.7778];
[mach,T,P,rho,velocity,T0,P0] = flowrayleigh(gamma,P,'pres')
```

```
mach =

    3.5833    0.5000

T =

    0.2170    0.7901

P =

    0.1300    1.7778

rho =

    0.5991    2.2501

velocity =

    1.6692    0.4444
```

```
T0 =

    0.5521    0.6913

P0 =

    7.4381    1.1141
```

This example returns a 1 x 2 array for *`mach`*, *`T`*, *`P`*, *`rho`*, *`velocity`*, *`T0`*, and *`P0`* each. The elements of each array correspond to the inputs element-wise.

References

1. James, J. E. A., *Gas Dynamics, Second Edition*, Allyn and Bacon, Inc, Boston, 1984.

2. *NACA Technical Report 1135*, 1953, National Advisory Committee on Aeronautics, Ames Research Staff, Moffett Field, Calif. Pages 667–671.

See Also

`flowfanno` | `flowisentropic` | `flownormalshock` | `flowprandtlmeyer`

Introduced in R2010a

generatePatches (Aero.Body)

Generate patches for body with loaded face, vertex, and color data

Syntax

```
generatePatches(h, ax)
h.generatePatches(ax)
```

Description

`generatePatches(h, ax)` and `h.generatePatches(ax)` generate patches for the animation body object `h` using the loaded face, vertex, and color data in `ax`.

Examples

Generate patches for `b` using the axes, *ax*.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
f = figure;
ax = axes;
b.generatePatches(ax);
```

See Also

`load`

Introduced in R2007a

GenerateRunScript (Aero.FlightGearAnimation)

Generate run script for FlightGear flight simulator

Syntax

```
GenerateRunScript(h)
h.GenerateRunScript
```

Description

`GenerateRunScript(h)` and `h.GenerateRunScript` generate a run script for FlightGear flight simulator using the following FlightGear animation object properties:

`OutputFileName`	Specify the name of the output file. The file name is the name of the command you will use to start FlightGear with these initial parameters. The default value is `'runfg.bat'`.
`FlightGearBaseDirectory`	Specify the name of your FlightGear installation folder. The default value is `'C:\Applications\FlightGear'`.
`GeometryModelName`	Specify the name of the folder containing the desired model geometry in the *`FlightGear`*`\data\Aircraft` folder. The default value is `'HL20'`.
`DestinationIpAddress`	Specify your destination IP address. The default value is `'127.0.0.1'`.
`DestinationPort`	Specify your network flight dynamics model (fdm) port. This destination port should be an unused port that you can use when you launch FlightGear. The default value is `'5502'`.
`AirportId`	Specify the airport ID. The list of supported airports is available in the FlightGear interface, under **Location**. The default value is `'KSFO'`.

`RunwayId`	Specify the runway ID. The default value is `'10L'`.
`InitialAltitude`	Specify the initial altitude of the aircraft, in feet. The default value is `7224` feet.
`InitialHeading`	Specify the initial heading of the aircraft, in degrees. The default value is `113` degrees.
`OffsetDistance`	Specify the offset distance of the aircraft from the airport, in miles. The default value is `4.72` miles.
`OffsetAzimuth`	Specify the offset azimuth of the aircraft, in degrees. The default value is `0` degrees.
`InstallScenery`	Direct FlightGear to automatically install required scenery while the simulator is running. This property requires a steady Internet connection. For Windows systems, you may encounter an error message while launching FlightGear with this option enabled. For more information, see “Installing Additional FlightGear Scenery” on page 2-56.
`DisableShaders`	Disable FlightGear shader options. Your computer built-in video card, such as NVIDIA cards, can conflict with FlightGear shaders. Consider using this property if you have this conflict.
`FlightGearVersion`	Specify the version of your FlightGear installation. The default is `2018.3`.
`Architecture`	Specify the architecture on which the FlightGear software is running.

Examples

Create a run script, `runfg.bat`, to start FlightGear flight simulator using the default object settings:

```
h = fganimation
GenerateRunScript(h)
```

Create a run script, `myscript.bat`, to start FlightGear flight simulator using the default object settings:

```
h = fganimation
h.OutputFileName = 'myscript.bat'
GenerateRunScript(h)
```

See Also

`initialize` | `play` | `update`

Introduced in R2007a

geoc2geod

Convert geocentric latitude to geodetic latitude

Syntax

`geodeticLatitude` `= geoc2geod(`*`geocentricLatitude`*`,` *`radii`*`)`
`geodeticLatitude` `= geoc2geod(`*`geocentricLatitude`*`,` *`radii`*`,` *`model`*`)`
`geodeticLatitude` `= geoc2geod(`*`geocentricLatitude`*`,` *`radii`*`,` *`flattening`*`,` *`equatorialRadius`*`)`

Description

`geodeticLatitude` `= geoc2geod(`*`geocentricLatitude`*`,` *`radii`*`)` converts an array of *m*-by-1 geocentric latitudes and an array of radii from the center of the planet into an array of *m*-by-1 geodetic latitudes.

`geodeticLatitude` `= geoc2geod(`*`geocentricLatitude`*`,` *`radii`*`,` *`model`*`)` converts for a specific ellipsoid planet.

`geodeticLatitude` `= geoc2geod(`*`geocentricLatitude`*`,` *`radii`*`,` *`flattening`*`,` *`equatorialRadius`*`)` converts for a custom ellipsoid planet defined by flattening and the equatorial radius.

The function uses geometric relationships to calculate the geodetic latitude in this noniterative method.

This function has the limitation that this implementation generates a geodetic latitude that lies between ±90 degrees.

Input Arguments

`geocentricLatitude`

Array of *m*-by-1 geocentric latitudes, in degrees. Latitude values can be any value. However, values of +90 and -90 may return unexpected values because of singularity at the poles.

`radii`

Array of radii from the center of the planet, in meters.

`model`

Specific ellipsoid planet. This function supports only `'WGS84'`.

`flattening`

Custom ellipsoid planet defined by flattening.

`equatorialRadius`

Equatorial radius, in meters.

Output Arguments

`geodeticLatitude`

Array of *m*-by-1 geodetic latitudes, in degrees.

Examples

Determine geodetic latitude given a geocentric latitude and radius:

```
gd = geoc2geod(45, 6379136)


gd =

   45.1921
```

Determine geodetic latitude at multiple geocentric latitudes, given a radius, and specifying WGS84 ellipsoid model:

```
gd = geoc2geod([0 45 90], 6379136, 'WGS84')


gd =

         0   45.1921   90.0000
```

Determine geodetic latitude at multiple geocentric latitudes, given a radius, and specifying custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
gd = geoc2geod([0 45 90], 6379136, f, Re)


gd =

         0   45.1550   90.0000
```

References

Jackson, E.B., *Manual for a Workstation-based Generic Flight Simulation Program (LaRCsim) Version 1.4*, NASA TM 110164, April 1995

Hedgley, D. R., Jr., *An Exact Transformation from Geocentric to Geodetic Coordinates for Nonzero Altitudes*, NASA TR R-458, March, 1976

Clynch, J. R., *Radius of the Earth — Radii Used in Geodesy*, Naval Postgraduate School, 2002, `https://core.ac.uk/download/pdf/36732690.pdf`

Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, John Wiley & Sons, New York, NY, 1992

Edwards, C. H., and D. E. Penny, *Calculus and Analytical Geometry*, 2nd Edition, Prentice-Hall, Englewood Cliffs, NJ, 1986

See Also

`ecef2lla` | `geod2geoc` | `lla2ecef`

Introduced in R2006b

geocradius

Convert from geocentric latitude to radius of ellipsoid planet

Syntax

```
r = geocradius(lambda)
r = geocradius(lambda, model)
r = geocradius(lambda, f, Re)
```

Description

`r = geocradius(lambda)` estimates the radius, `r`, of an ellipsoid planet at a particular geocentric latitude, `lambda`. `lambda` is in degrees. `r` is in meters. The default ellipsoid planet is WGS84.

`r = geocradius(lambda, model)` is an alternate method for estimating the radius for a specific ellipsoid planet. Currently only `'WGS84'` is supported for `model`.

`r = geocradius(lambda, f, Re)` is another alternate method for estimating the radius for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters.

Examples

Determine radius at 45 degrees latitude:

```
r = geocradius(45)


r =

  6.3674e+006
```

Determine radius at multiple latitudes:

```
r = geocradius([0 45 90])


r =

  1.0e+006 *

    6.3781    6.3674    6.3568
```

Determine radius at multiple latitudes, specifying WGS84 ellipsoid model:

```
r = geocradius([0 45 90], 'WGS84')


r =

  1.0e+006 *

    6.3781    6.3674    6.3568
```

Determine radius at multiple latitudes, specifying custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
r = geocradius([0 45 90], f, Re)


r =

  1.0e+006 *

    3.3970    3.3883    3.3797
```

References

Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, John Wiley & Sons, New York, NY, 1992

Zipfel, P. H., and D. E. Penny, *Modeling and Simulation of Aerospace Vehicle Dynamics*, AIAA Education Series, Reston, VA, 2000

See Also

geoc2geod | geod2geoc

Introduced in R2006b

geod2geoc

Convert geodetic latitude to geocentric latitude

Syntax

```
gc = geod2geoc(gd, h)
gc = geod2geoc(gd, h, model)
gc = geod2geoc(gd, h, f, Re)
```

Description

`gc = geod2geoc(gd, h)` converts an array of `m` geodetic latitudes, `gd`, and an array of mean sea level altitudes, `h`, into an array of `m` geocentric latitudes, `gc`. `h` is in meters. Latitude values can be any value. However, values of +90 and -90 may return unexpected values because of singularity at the poles.

`gc = geod2geoc(gd, h, model)` is an alternate method for converting from geodetic to geocentric latitude for a specific ellipsoid planet. Currently only `'WGS84'` is supported for `model`. Latitude values can be any value. However, values of +90 and -90 may return unexpected values because of singularity at the poles.

`gc = geod2geoc(gd, h, f, Re)` is another alternate method for converting from geodetic to geocentric latitude for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters. Latitude values can be any value. However, values of +90 and -90 may return unexpected values because of singularity at the poles.

Examples

Determine geocentric latitude given a geodetic latitude and altitude:

```
gc = geod2geoc(45, 1000)


gc =
   44.8076
```

Determine geocentric latitude at multiple geodetic latitudes and altitudes, specifying WGS84 ellipsoid model:

```
gc = geod2geoc([0 45 90], [1000 0 2000], 'WGS84')


gc =

         0
   44.8076
   90.0000
```

Determine geocentric latitude at multiple geodetic latitudes, given an altitude and specifying custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
gc = geod2geoc([0 45 90], 2000, f, Re)


gc =

         0
   44.7084
   90.0000
```

Assumptions and Limitations

This implementation generates a geocentric latitude that lies between ±90 degrees.

References

Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, John Wiley & Sons, New York, NY, 1992

See Also

`ecef2lla` | `geoc2geod` | `lla2ecef`

Introduced in R2006b

geoidegm96

Calculate geoid height as determined from EGM96 Geopotential Model

Note `geoidegm96` will be removed in a future version. Use `geoidheight` instead.

Syntax

```
N = geoidegm96(lat, long)
N = geoidegm96(lat, long, action)
```

Description

`N = geoidegm96(lat, long)` calculates the geoid height as determined from the EGM96 Geopotential Model. It calculates geoid heights to 0.01 meters. This function interpolates geoid heights from a 15-minute grid of point values in the tide-free system, using the EGM96 Geopotential Model to the degree and order 360. The geoid undulations are relative to the WGS84 ellipsoid.

`N = geoidegm96(lat, long, action)` calculates the geoid height as determined from the EGM96 Geopotential Model. This function performs `action` if `latitude` or `longitude` are out of range.

Inputs required by `geoidegm96`:

`lat`	An array of `m` geocentric latitudes, in degrees, where north latitude is positive and south latitude is negative. `lat` must be of type single or double. If `lat` is not within the range -90 to 90, inclusive, this function wraps the value to be within the range.

`long`	An array of `m` geocentric longitudes, in degrees, where east longitude is positive and west longitude is negative. `long` must be of type single or double. If `long` is not within the range 0 to 360 inclusive, this function wraps the value to be within the range.
`action`	Action for out-of-range input. Specify if out-of-range input invokes a `'Warning'`, `'Error'`, or no action (`'None'`). The default is `'Warning'`.

Examples

Calculate the geoid height at 42.4 degrees N latitude and 71.0 degrees E longitude.

```
N = geoidegm96( 42.4, 71.0)
```

Calculate the geoid height at two different locations, with out-of-range actions generating warnings.

```
N = geoidegm96( [39.3,33.4], [-77.2, 36.5])
```

Calculate the geoid height with latitude wrapping, with out-of-range actions displaying no warnings.

```
N = geoidegm96(100,150,'None')
```

Limitations

This function has the limitations of the 1996 Earth Geopotential Model. For more information, see `https://earth-info.nga.mil/GandG/wgs84/gravitymod/egm96/egm96.html`.

The WGS84 EGM96 geoid undulations have an error range of +/-0.5 to +/-1.0 meters worldwide.

References

NIMA TR8350.2: “Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems.”

NASA/TP-1998-206861: “The Development of the Joint NASA GSFC and NIMA Geopotential Model EGM96”

National Geospatial-Intelligence Agency Website: http://earth-info.nga.mil/GandG/wgs84/gravitymod/egm96/egm96.html

See Also

`gravitywgs84`

Introduced in R2007b

geoidheight

Calculate geoid height

Syntax

```
N = geoidheight(latitude,longitude)
N = geoidheight(latitude, longitude, modelname)
N = geoidheight(latitude, longitude, action)
N = geoidheight(latitude, longitude, modelname, action)
N = geoidheight(latitude, longitude, 'Custom', datafile)
N = geoidheight(latitude, longitude, 'Custom', datafile, action)
```

Description

`N = geoidheight(latitude,longitude)` calculates the geoid height using the EGM96 Geopotential Model. For this model, it calculates these geoid heights to an accuracy of 0.01 m. It interpolates an array of *m* geoid heights at *m* geocentric latitudes, *latitude*, and *m* geocentric longitudes, *longitude*.

`N = geoidheight(latitude, longitude, modelname)` calculates the geoid height using the model, *modelname*.

`N = geoidheight(latitude, longitude, action)` calculates the geoid height using the EGM96 Geopotential Model. This function performs *action* if *latitude* or *longitude* are out of range.

`N = geoidheight(latitude, longitude, modelname, action)` calculates the geoid height using *modelname*.

`N = geoidheight(latitude, longitude, 'Custom', datafile)` calculates the geoid height using a custom model that *datafile* defines.

`N = geoidheight(latitude, longitude, 'Custom', datafile, action)` calculates the geoid height using the custom model. This function performs *action* if *latitude* or `longitude` are out of range.

Input Arguments

`latitude`

An array of *`m`* geocentric latitudes, in degrees, where north latitude is positive and south latitude is negative. *`latitude`* must be of type single or double. If *`latitude`* is not within the range –90 to 90, inclusive, this function wraps the value to be within the range.

Default:

`longitude`

An array of *`m`* geocentric longitudes, in degrees, where east longitude is positive and west longitude is negative. *`longitude`* must be of type single or double. If *`longitude`* is not within the range 0 to 360 inclusive, this function wraps the value to be within the range.

`modelname`

Geopotential model.

Geopotential Model	Description
`'EGM96'`	EGM96 Geopotential Model to degree and order 360. This model uses a 15-minute grid of point values in the tide-free system. This function calculates geoid heights to an accuracy of 0.01 m for this model.
`'EGM2008'`	EGM2008 Geopotential Model to degree and order 2159. This model uses a 2.5-minute grid of point values in the tide-free system. This function calculates geoid heights to an accuracy of 0.001 m for this model. **Note** This function requires that you download EGM2008 Geopotential Model data with the Add-On Explorer. For more information, see `aeroDataPackage`.

Geopotential Model	Description

`'Custom'`	Custom geopotential model that you define in *`datafile`*. This function calculates geoid heights to an accuracy of 0.01 m for custom models.
	
	Note To deploy a custom geopotential model, explicitly include the custom data and reader files to the MATLAB Compiler™ (`mcc`) command at compilation. For example:
	
	`mcc -m mycustomsgeoidheightfunction...`
	`-a customDataFile`
	
	For other geopotential models, use the MATLAB Compiler as usual.

Default: EGM96

`datafile`

Optional file that contains definitions for a custom geopotential model. Provide this file only if you specify `'Custom'` for the `modelname` argument.

This file must contain the following variables.

Variable	Description
`'latbp'`	Array of geocentric latitude breakpoints.
`'lonbp'`	Array of geocentric longitude breakpoints.
`'grid'`	Table of geoid height values.
`'windowSize'`	Even integer scalar greater than 2 for the number of interpolation points.

`action`

Action for out-of-range input. Specify one:
`'Error'`
`'Warning'`
`'None'`

Default: `Warning`

Output Arguments

N

An array of *M* geoid heights in meters. The values in this array have the same data type as *latitude*.

Examples

Calculate the EGM96 geoid height at 42.4 degrees N latitude and 71.0 degrees W longitude with warning actions:

```
N = geoidheight( 42.4, -71.0 )
```

Calculate the EGM2008 geoid height at two different locations with error actions.

```
N = geoidheight( [39.3, 33.4], [77.2, 36.5], 'egm2008', ...
'error')
```

Calculate a custom geoid height at two different locations with no actions.

```
N = geoidheight( [39.3, 33.4], [-77.2, 36.5], 'custom', ...
'geoidegm96grid','none')
```

Tips

- This function interpolates geoid heights from a grid of point values in the tide-free system.
- When using the EGM96 Model, this function has the limitations of the 1996 Earth Geopotential Model.
- When using the EGM2008 Model, this function has the limitations of the 2008 Earth Geopotential Model.
- The interpolation scheme wraps over the poles to allow for geoid height calculations at and near pole locations.
- The geoid undulations for the EGM96 and EGM2008 models are relative to the WGS84 ellipsoid.
- The WGS84 EGM96 geoid undulations have an error range of +/– 0.5 to +/– 1.0 m worldwide.

References

Vallado, D. A. “Fundamentals of Astrodynamics and Applications.” McGraw-Hill, New York, 1997.

NIMA TR8350.2: "Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems."

See Also

`gravitysphericalharmonic` | `gravitywgs84`

External Websites

National Geospatial-Intelligence Agency Web site: http://earth-info.nga.mil/GandG/publications/vertdatum.html

Introduced in R2010b

Geometry (Aero.Geometry)

Construct 3-D geometry for use with animation object

Syntax

```
h = Aero.Geometry
```

Description

`h = Aero.Geometry` defines a 3-D geometry for use with an animation object.

See `Aero.Geometry` for further details.

See Also

`Aero.Geometry`

Introduced in R2007a

gravitycentrifugal

Implement centrifugal effect of planetary gravity

Syntax

```
[gx gy gz] = gravitycentrifugal(planet_coordinates)
[gx gy gz] = gravitycentrifugal(planet_coordinates, model )
[gx gy gz] = gravitycentrifugal(planet_coordinates, 'Custom',
rotational_rate)
```

Description

`[gx gy gz] = gravitycentrifugal(planet_coordinates)` implements the mathematical representation of centrifugal effect for planetary gravity based on planetary rotation rate. This function calculates arrays of *N* gravity values in the *x*-axis, *y*-axis, and *z*-axis of the Planet-Centered Planet-Fixed coordinates for the planet. It performs these calculations using *`planet_coordinates`*, an *M*-by-3 array of Planet-Centered Planet-Fixed coordinates. You use centrifugal force in rotating or noninertial coordinate systems. Gravity centrifugal effect values are greatest at the equator of a planet.

`[gx gy gz] = gravitycentrifugal(planet_coordinates, model )` implements the mathematical representation of centrifugal effect based on planetary gravitational potential for the planetary model, *`model`*.

`[gx gy gz] = gravitycentrifugal(planet_coordinates, 'Custom', rotational_rate)` implements the mathematical representation of centrifugal effect based on planetary gravitational potential using the custom rotational rate, *`rotational_rate`*.

Input Arguments

`planet_coordinates`

`M`-by-3 array of Planet-Centered Planet-Fixed coordinates in meters. The *`z`*-axis is positive toward the North Pole. If *`model`* is `'Earth'`, the planet coordinates are ECEF coordinates.

`model`

Planetary model. Default is `'Earth'`. Specify one:

- `'Mercury'`
- `'Venus'`
- `'Earth'`
- `'Moon'`
- `'Mars'`
- `'Jupiter'`
- `'Saturn'`
- `'Uranus'`
- `'Neptune'`
- `'Custom'`

`'Custom'` requires that you specify your own planetary model using the *`rotational_rate`* parameter.

`rotational_rate`

Scalar value that specifies the planetary rotational rate in radians per second. Specify this parameter only if *`model`* has the value `'Custom'`.

Output Arguments

`gx`

Array of *`M`* gravity values in the *`x`*-axis of the Planet-Centered Planet-Fixed coordinates in meters per second squared (m/s^2).

gy

Array of *M* gravity values in the *y*-axis of the Planet-Centered Planet-Fixed coordinates in meters per second squared (m/s^2).

gz

Array of *M* gravity values in the *z*-axis of the Planet-Centered Planet-Fixed coordinates in meters per second squared (m/s^2).

Examples

Calculate the centrifugal effect of Earth gravity in the *x*-axis at the equator on the surface of Earth:

```
gx = gravitycentrifugal( [-6378.1363e3 0 0] )
```

Calculate the centrifugal effect of Mars gravity at 15000 m over the equator and 11000 m over the North Pole:

```
p  = [2412.648e3 -2412.648e3 0; 0 0 3376.2e3]
[gx, gy, gz] = gravitycentrifugal( p, 'Mars' )
```

Calculate the precessing centrifugal effect of gravity for Earth at 15000 m over the equator and 11000 m over the North Pole. This example uses a custom planetary model at Julian date 2451545:

```
p       = [2412.648e3 -2412.648e3 0; 0 0 3376e3]
% Set julian date to January 1, 2000 at noon GMT
JD      = 2451545
% Calculate precession rate in right ascension in meters
pres_RA = 7.086e-12 + 4.3e-15*(JD - 2451545)/36525
% Calculate the rotational rate in a precessing reference
% frame
Omega   = 7.2921151467e-5 + pres_RA
[gx, gy, gz] = gravitycentrifugal( p, 'custom', Omega )
```

See Also

gravitysphericalharmonic | gravitywgs84 | gravityzonal

Introduced in R2010a

gravitysphericalharmonic

Implement spherical harmonic representation of planetary gravity

Syntax

```
[gx gy gz] = gravitysphericalharmonic(planet_coordinates)
[gx gy gz] = gravitysphericalharmonic(planet_coordinates, model)
[gx gy gz] = gravitysphericalharmonic(planet_coordinates, degree)
[gx gy gz] = gravitysphericalharmonic(planet_coordinates, model,
degree)
[gx gy gz] = gravitysphericalharmonic(planet_coordinates, model,
degree, action)
[gx gy gz] = gravitysphericalharmonic(planet_coordinates, 'Custom',
degree, {datafile dfreader}, action)
```

Description

`[gx gy gz] = gravitysphericalharmonic(planet_coordinates)` implements the mathematical representation of spherical harmonic planetary gravity based on planetary gravitational potential. This function calculates arrays of *N* gravity values in the *x*-axis, *y*-axis, and *z*-axis of the Planet-Centered Planet-Fixed coordinates for the planet. It performs these calculations using *`planet_coordinates`*, an *M*-by-3 array of Planet-Centered Planet-Fixed coordinates. By default, this function assumes 120th degree and order spherical coefficients for the `'EGM2008'` (Earth) planetary model.

`[gx gy gz] = gravitysphericalharmonic(planet_coordinates, model)` implements the mathematical representation for the planetary model, *`model`*.

`[gx gy gz] = gravitysphericalharmonic(planet_coordinates, degree)` uses the degree and order that *`degree`* specifies.

`[gx gy gz] = gravitysphericalharmonic(planet_coordinates, model, degree)` uses the degree and order that *`degree`* specifies. *`model`* specifies the planetary model.

`[gx gy gz] = gravitysphericalharmonic(`*`planet_coordinates`*`, `*`model`*`, `*`degree`*`, `*`action`*`)` uses the specified *`action`* when input is out of range.

`[gx gy gz] = gravitysphericalharmonic(`*`planet_coordinates`*`, 'Custom', `*`degree`*`, {`*`datafile dfreader`*`}, `*`action`*`)` implements the mathematical representation for a custom model planet. *`datafile`* defines the planetary model. *`dfreader`* specifies the reader for *`datafile`*.

This function has the following limitations:

- The function excludes the centrifugal effects of planetary rotation, and the effects of a precessing reference frame.
- Spherical harmonic gravity model is valid for radial positions greater than the planet equatorial radius. Minor errors might occur for radial positions near or at the planetary surface. The spherical harmonic gravity model is not valid for radial positions less than planetary surface.

Input Arguments

`planet_coordinates`

M-by-3 array of Planet-Centered Planet-Fixed coordinates in meters. The *z*-axis is positive toward the North Pole. If *`model`* is `'EGM2008'` or `'EGM96'` (Earth), the planet coordinates are ECEF coordinates.

`model`

Planetary model. Default is `'EGM2008'`. Specify one:

Planetary Model	Planet
`'EGM2008'`	Earth Gravitational Model 2008. Planet coordinates are ECEF (WGS84).
`'EGM96'`	Earth Gravitational Model 1996. Planet coordinates are ECEF (WGS84).
`'LP100K'`	100th degree Moon model.
`'LP165P'`	165th degree Moon model.
`'GMM2B'`	Goddard Mars model 2B.

Planetary Model	Planet

`'Custom'`	Custom planetary model that you define in *`datafile`*.
	
	Note To deploy a custom planetary model, explicitly include the custom data and reader files to the MATLAB Compiler (`mcc`) command at compilation. For example:
	
	`mcc -m mycustomsphericalgravityfunction...`
	`-a customDataFile -a customReaderFile`
	
	For other planetary models, use the MATLAB Compiler as usual.

`'EIGENGL04C'`	Combined Earth gravity field model EIGEN-GL04C.

When inputting a large PCPF array and a high degree value, you might receive an out-of-memory error. For more information about avoiding out-of-memory errors in the MATLAB environment, see “Resolve “Out of Memory” Errors” (MATLAB).

When inputting a large PCPF array, you might receive a maximum matrix size limitation. To determine the largest matrix or array that you can create in the MATLAB environment for your platform, see “Performance and Memory” (MATLAB).

`degree`

Scalar value that specifies the degree and order of the harmonic gravity model.

Planetary Model	Degree and Order
`'EGM2008'`	Maximum degree and order is 2159. Default degree and order are 120.
`'EGM96'`	Maximum degree and order is 360. Default degree and order are 70.
`'LP100K'`	Maximum degree and order is 100. Default degree and order are 60.
`'LP165P'`	Maximum degree and order is 165. Default degree and order are 60.

Planetary Model	Degree and Order
'GMM2B'	Maximum degree and order is 80. Default degree and order are 60.
'Custom'	Maximum degree is default degree and order.
'EIGENGL04C'	Maximum degree and order is 360. Default degree and order are 70.

When inputting a large PCPF array and a high degree value, you might receive an out-of-memory error. For more information about avoiding out-of-memory errors in the MATLAB environment, see “Performance and Memory” (MATLAB).

When inputting a large PCPF array, you might receive a maximum matrix size limitation. To determine the largest matrix or array that you can create in the MATLAB environment for your platform, see “Performance and Memory” (MATLAB).

action

Action for out-of-range input. Specify one:
'Error'
'Warning' (default)
'None'

'Custom'

Character vector or string that specifies that *datafile* contains definitions for a custom planetary model.

datafile

File that contains definitions for a custom planetary model. For an example of file content, see aerogmm2b.mat.

This file must contain the following variables.

Variable	Description
Re	Scalar of planet equatorial radius in meters (m)

Variable	Description
GM	Scalar of planetary gravitational parameter in meters cubed per second squared (m^3/s^2)
degree	Scalar of maximum degree
C	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *C*
S	(*degree*+1)-by-(*degree*+1) matrix containing normalized spherical harmonic coefficients matrix, *S*

This parameter requires that you specify a program in the *`dfreader`* parameter to read the data file.

`dfreader`

Specify a MATLAB function to read `datafile`. The reader file that you specify depends on the file type of `datafile`.

Data File Type	Description
MATLAB file	Specify the MATLAB `load` function, for example, `@load`.
Other file type	Specify a custom MATLAB reader function. For examples of custom reader functions, see `astReadSHAFile.m` and `astReadEGMFile.m`. Note the output variable order in these files.

Output Arguments

`gx`

Array of *N* gravity values in the *x*-axis of the Planet-Centered Planet-Fixed coordinates in meters per second squared (m/s^2).

`gy`

Array of *N* gravity values in the *y*-axis of the Planet-Centered Planet-Fixed coordinates in meters per second squared (m/s^2).

gz

Array of *N* gravity values in the *z*-axis of the Planet-Centered Planet-Fixed coordinates in meters per second squared (m/s^2).

Examples

Calculate the gravity in the *x*-axis at the equator on the surface of Earth. This example uses the default 120 degree model of EGM2008 with default warning actions:

```
gx = gravitysphericalharmonic( [-6378.1363e3 0 0] )
```

Calculate the gravity at 25000 m over the south pole of Earth. This example uses the 70 degree model of EGM96 with error actions:

```
[gx, gy, gz] = gravitysphericalharmonic( [0 0 -6381.751e3], 'EGM96', 'Error' )
```

Calculate the gravity at 15000 m over the equator and 11000 m over the North Pole. This example uses a 30th order GMM2B Mars model with warning actions:

```
p  = [2412.648e3 -2412.648e3 0; 0 0 3397.2e3]
[gx, gy, gz] = gravitysphericalharmonic( p, 'GMM2B', 30, 'Warning' )
```

Calculate the gravity at 15000 m over the equator and 11000 m over the North Pole. This example uses a 60th degree custom planetary model with no actions:

```
p       = [2412.648e3 -2412.648e3 0; 0 0 3397e3]
[gx, gy, gz] = gravitysphericalharmonic( p, 'custom', 60, ...
{'GMM2BC80_SHA.txt' @astReadSHAFile}, 'None' )
```

Calculate the gravity at 25000 meters over the south pole of Earth using a 120th order EIGEN-GL04C Earth model with warning actions:

```
p  = [0 0 -6381.751e3]
[gx, gy, gz] = gravitysphericalharmonic( p, 'EIGENGL04C', ...
120, 'Warning' )
```

Tips

- When inputting a large PCPF array and a high degree value, you might receive an out-of-memory error. For more information about avoiding out-of-memory errors in the MATLAB environment, see “Performance and Memory” (MATLAB).

- When inputting a large PCPF array, you might receive a maximum matrix size limitation. To determine the largest matrix or array that you can create in the MATLAB environment for your platform, see “Performance and Memory” (MATLAB).

References

[1] Gottlieb, R. G., “Fast Gravity, Gravity Partials, Normalized Gravity, Gravity Gradient Torque and Magnetic Field: Derivation, Code and Data,” *Technical Report NASA Contractor Report 188243*, NASA Lyndon B. Johnson Space Center, Houston, Texas, February 1993.

[2] Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, McGraw-Hill, New York, 1997.

[3] “NIMA TR8350.2: Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems”.

[4] Konopliv, A. S., S. W. Asmar, E. Carranza, W. L. Sjogen, D. N. Yuan., “Recent Gravity Models as a Result of the Lunar Prospector Mission, Icarus”, Vol. 150, no. 1, pp 1–18, 2001.

[5] Lemoine, F. G., D. E. Smith, D.D. Rowlands, M.T. Zuber, G. A. Neumann, and D. S. Chinn, “An improved solution of the gravity field of Mars (GMM-2B) from Mars Global Surveyor”, *Journal Of Geophysical Research*, Vol. 106, No. E10, pp 23359-23376, October 25, 2001.

[6] Kenyon S., J. Factor, N. Pavlis, and S. Holmes, “Towards the Next Earth Gravitational Model”, Society of Exploration Geophysicists 77th Annual Meeting, San Antonio, Texas, September 23–28, 2007.

[7] Pavlis, N.K., S.A. Holmes, S.C. Kenyon, and J.K. Factor, “An Earth Gravitational Model to Degree 2160: EGM2008”, presented at the 2008 General Assembly of the European Geosciences Union, Vienna, Austria, April 13–18, 2008.

[8] Grueber, T., and A. Köhl, “Validation of the EGM2008 Gravity Field with GPS-Leveling and Oceanographic Analyses”, presented at the IAG International Symposium on Gravity, Geoid & Earth Observation 2008, Chania, Greece, June 23–27, 2008.

[9] Förste, C., Flechtner, F., Schmidt, R., König, R., Meyer, U., Stubenvoll, R., Rothacher, M., Barthelmes, F., Neumayer, H., Biancale, R., Bruinsma, S., Lemoine, J.M., Loyer, S., “A Mean Global Gravity Field Model From the Combination of Satellite

Mission and Altimetry/Gravmetry Surface Data - EIGEN-GL04C”, *Geophysical Research Abstracts*, Vol. 8, 03462, 2006.

[10] Hill, K. A. (2007). Autonomous Navigation in Libration Point Orbits. Doctoral dissertation, University of Colorado, Boulder.

[11] Colombo, Oscar L., “Numerical Methods for Harmonic Analysis on the Sphere”, Reports of the department of Geodetic Science, Report No. 310, The Ohio State University, Columbus, OH., March 1981.

[12] Colombo, Oscar L., “The Global Mapping of Gravity with Two Satellites", Nederlands Geodetic Commission, vol. 7 No. 3, Delft, The Nederlands, 1984., Reports of the department of Geodetic Science, Report No. 310, The Ohio State University, Columbus, OH., March 1981.

[13] Jones, Brandon A. (2010). Efficient Models for the Evaluation and Estimation of the Gravity Field. Doctoral dissertation, University of Colorado, Boulder.

See Also

`geoidegm96` | `gravitycentrifugal` | `gravitywgs84` | `gravityzonal`

Introduced in R2010a

gravitywgs84

Implement 1984 World Geodetic System (WGS84) representation of Earth's gravity

Syntax

```
g = gravitywgs84(h, lat)
g = gravitywgs84(h, lat, lon, method, [noatm, nocent, prec, jd],
action)
gn = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent, prec, jd],
action)
[gn gt] = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent, prec,
jd], action)
```

Description

`g = gravitywgs84(h, lat)` implements the mathematical representation of the geocentric equipotential ellipsoid of WGS84. Using `h`, an array of `m` altitudes in meters, and `lat`, an array of `m` geodetic latitudes in degrees, calculates `g`, an array of `m` gravity values in the direction normal to the Earth's surface at a specific location. The default calculation method is Taylor Series. Gravity precision is controlled via the `method` parameter.

`g = gravitywgs84(h, lat, lon, method, [noatm, nocent, prec, jd], action)` lets you specify both latitude and longitude, as well as other optional inputs, when calculating gravity values in the direction normal to the Earth's surface. In this format, `method` can be either `'CloseApprox'`or`'Exact'`.

`gn = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent, prec, jd], action)` calculates an array of total gravity values in the direction normal to the Earth's surface.

`[gn gt] = gravitywgs84(h, lat, lon, 'Exact', [noatm, nocent, prec, jd], action)` calculates gravity values in the direction both normal and tangential to the Earth's surface.

Inputs for `gravitywgs84` are:

`h`	An array of `m` altitudes, in meters, with respect to the WGS84 ellipsoid.
`lat`	An array of `m` geodetic latitudes, in degrees, where north latitude is positive, and south latitude is negative.
`lon`	An array of `m` geodetic longitudes, in degrees, where east longitude is positive, and west longitude is negative. This input is available only with `method` specified as `'CloseApprox'`or`'Exact'`.
`method`	Method to calculate gravity: `'TaylorSeries'`, `'CloseApprox'`, or `'Exact'`. The default is `'TaylorSeries'`.
`noatm`	A logical value specifying the exclusion of Earth's atmosphere. Set to `true` for the Earth's gravitational field to exclude the mass of the atmosphere. Set to `false` for the value for the Earth's gravitational field to include the mass of the atmosphere. This option is available only with `method` specified as `'CloseApprox'`or`'Exact'`. The default is `false`.
`nocent`	A logical value specifying the removal of centrifugal effects. Set to `true` to calculate gravity based on pure attraction resulting from the normal gravitational potential. Set to `false` to calculate gravity including the centrifugal force resulting from the Earth's angular velocity. This option is available only with `method` specified as `'CloseApprox'`or`'Exact'`. The default is `false`.
`prec`	A logical value specifying the presence of a precessing reference frame. Set to `true` for the angular velocity of the Earth to be calculated using the International Astronomical Union (IAU) value of the Earth's angular velocity and the precession rate in right ascension. To obtain the precession rate in right ascension, Julian Centuries from Epoch J2000.0 is calculated using the Julian date, `jd`. If set to `false`, the angular velocity of the Earth used is the value of the standard Earth rotating at a constant angular velocity. This option is available only with `method` specified as `'CloseApprox'`or`'Exact'`. The default is `false`.

`jd`	A scalar value specifying Julian date used to calculate Julian Centuries from Epoch J2000.0. This input is available only with `method` specified as `'CloseApprox'`or`'Exact'`.
`action`	Action for out-of-range input. Specify if out-of-range input invokes a `'Warning'`, `'Error'`, or no action (`'None'`). The default is `'Warning'`.

Outputs calculated for the Earth's gravity include:

`g`	An array of `m` gravity values in the direction normal to the Earth's surface at a specific `lat lon` location. A positive value indicates a downward direction.
`gt`	An array of `m` gravity values in the direction tangential to the Earth's surface at a specific `lat lon` location. A positive value indicates a northward direction. This option is available only with `method` specified as`'Exact'`.
`gn`	An array of `m` total gravity values in the direction normal to the Earth's surface at a specific `lat lon` location. A positive value indicates a downward direction. This option is available only with `method` specified as`'Exact'`.

Examples

Calculate the normal gravity at 5000 meters and 55 degrees latitude using the Taylor Series approximation method with errors for out-of-range inputs:

```
g = gravitywgs84( 5000, 55, 'TaylorSeries', 'Error')

g =

    9.7997
```

Calculate the normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude using the Close Approximation method with atmosphere, centrifugal effects, and no precessing, with warnings for out-of-range inputs:

```
g = gravitywgs84( 15000, 45, 120, 'CloseApprox')

g =
    9.7601
```

Calculate the normal and tangential gravity at 1000 meters, 0 degrees latitude, and 20 degrees longitude using the Exact method with atmosphere, centrifugal effects, and no precessing, with warnings for out-of-range inputs:

```
[gn, gt] = gravitywgs84( 1000, 0, 20, 'Exact')

gn =

    9.7772

gt =

     0
```

Calculate the normal and tangential gravity at 1000 meters, 0 degrees latitude, and 20 degrees longitude and 11,000 meters, 30 degrees latitude, and 50 degrees longitude using the Exact method with atmosphere, centrifugal effects, and no precessing, with no actions for out-of-range inputs:

```
h = [1000; 11000];
lat = [0; 30];
lon = [20; 50];
[gn, gt] = gravitywgs84( h, lat, lon, 'Exact', 'None' )

gn =

    9.7772
    9.7594

gt =

   1.0e-04 *

         0
   -0.7751
```

Calculate the normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude and 5000 meters, 55 degrees latitude, and 100 degrees longitude using the Close Approximation method with atmosphere, no centrifugal effects, and no precessing, with warnings for out-of-range inputs:

```
h = [15000 5000];
lat = [45 55];
lon = [120 100];
g = gravitywgs84( h, lat, lon, 'CloseApprox', [false true false 0])

g =

    9.7771    9.8109
```

Calculate the normal and tangential gravity at 1000 meters, 0 degrees latitude, and 20 degrees longitude using the Exact method with atmosphere, centrifugal effects, and precessing at Julian date 2451545, with warnings for out-of-range inputs:

```
[gn, gt] = gravitywgs84( 1000, 0, 20, 'Exact', ...
              [ false false true 2451545 ], 'Warning')

gn =

    9.7772

gt =

     0
```

Calculate the normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude using the Close Approximation method with no atmosphere, with centrifugal effects, and with precessing at Julian date 2451545, with errors for out-of-range inputs:

```
g = gravitywgs84( 15000, 45, 120, 'CloseApprox', ...
        [ true false true 2451545 ], 'Error')

g =

    9.7601
```

Calculate the total normal gravity at 15,000 meters, 45 degrees latitude, and 120 degrees longitude using the Exact method with no atmosphere, with centrifugal effects, and with precessing at Julian date 2451545, with errors for out-of-range inputs:

```
gn = gravitywgs84( 15000, 45, 120, 'Exact', ...
        [ true false true 2451545 ], 'Error')

gn =

    9.7601
```

Assumptions and Limitations

The WGS84 gravity calculations are based on the assumption of a geocentric equipotential ellipsoid of revolution. Since the gravity potential is assumed to be the same everywhere on the ellipsoid, there must be a specific theoretical gravity potential that can be uniquely determined from the four independent constants defining the ellipsoid.

Use of the WGS84 Taylor Series model should be limited to low geodetic heights. It is sufficient near the surface when submicrogal precision is not necessary. At medium and high geodetic heights, it is less accurate.

Use of the WGS84 Close Approximation model should be limited to a geodetic height of 20,000.0 meters (approximately 65,620.0 feet). Below this height, it gives results with submicrogal precision.

To predict and determine a satellite orbit with high accuracy, use the EGM96 through degree and order 70.

References

NIMA TR8350.2: “Department of Defense World Geodetic System 1984, Its Definition and Relationship with Local Geodetic Systems.”

Introduced in R2006b

gravityzonal

Implement zonal harmonic representation of planetary gravity

Syntax

```
[gravityXcoord gravityYcoord, gravityZcoord] =
gravityzonal(planetCoord)
[gravityXcoord gravityYcoord, gravityZcoord] =
gravityzonal(planetCoord, degreeGravityModel)
[gravityXcoord gravityYcoord, gravityZcoord] =
gravityzonal(planetCoord, planetModel)
[gravityXcoord gravityYcoord, gravityZcoord] =
gravityzonal(planetCoord, planetModel, degreeGravityModel)
[gravityXcoord gravityYcoord, gravityZcoord] =
gravityzonal(planetCoord, planetModel, degreeGravityModel, action)
[gravityXcoord gravityYcoord, gravityZcoord] =
gravityzonal(planetCoord, 'Custom', equatorialRadius,
planetaryGravitional, zonalHarmonicCoeff, action)
```

Description

`[gravityXcoord gravityYcoord, gravityZcoord] = gravityzonal(planetCoord)` implements the mathematical representation of zonal harmonic planetary gravity based on planetary gravitational potential. For input, it takes an m-by-3 matrix that contains planet-centered planet-fixed coordinates from the center of the planet in meters. This function calculates the arrays of m gravity values in the *x*-, *y*-, and *z*-axes of the planet-centered planet-fixed coordinates. It uses the fourth order zonal coefficients for Earth by default.

`[gravityXcoord gravityYcoord, gravityZcoord] = gravityzonal(planetCoord, degreeGravityModel)` uses the degree of harmonic model.

`[gravityXcoord gravityYcoord, gravityZcoord] = gravityzonal(planetCoord, planetModel)` uses the planetary model.

`[gravityXcoord gravityYcoord, gravityZcoord] = gravityzonal(planetCoord, planetModel, degreeGravityModel)` uses the degree of harmonic model and planetary model.

`[gravityXcoord gravityYcoord, gravityZcoord] = gravityzonal(planetCoord, planetModel, degreeGravityModel, action)` specifies the action for out-of-range input.

`[gravityXcoord gravityYcoord, gravityZcoord] = gravityzonal(planetCoord, 'Custom', equatorialRadius, planetaryGravitional, zonalHarmonicCoeff, action)` uses the equatorial radius, planetary gravitational parameter, and zonal harmonic coefficients for the custom planetary model.

This function does not include the potential due planet rotation, which excludes the centrifugal effects of planetary rotation and the effects of a precessing reference frame.

Input Arguments

`planetCoord`

m-by-3 matrix that contains planet-centered planet-fixed coordinates from the center of the planet in meters. If `planetModel` has a value of `'Earth'`, this matrix contains Earth-centered Earth-fixed (ECEF) coordinates.

`planetModel`

Planetary model. Enter one:

- `'Mercury'`
- `'Venus'`
- `'Earth'`
- `'Moon'`
- `'Mars'`
- `'Jupiter'`
- `'Saturn'`
- `'Uranus'`

- `'Neptune'`
- `'Custom'`

`'Custom'` requires you to specify your own planetary model using the `equatorialRadius`, `planetaryGravitional`, and `zonalHarmonicCoeff` parameters.

Default: `'Earth'`

`degreeGravityModel`

Degree of harmonic model.

- `2` — Second degree, J2. Most significant or largest spherical harmonic term, which accounts for the oblateness of a planet. `2` is default if `planetModel` is `'Mercury'`, `'Venus'`, `'Moon'`, `'Uranus'`, or `'Neptune'`.
- `3` — Third degree, J3. `3` is default if `planetModel` is `'Mars'`.
- `4` — Fourth degree, J4 (default). Default is `4` if `planetModel` is `'Earth`, `'Jupiter'`, `'Saturn'` or `'Custom'`.

Default:

`equatorialRadius`

Planetary equatorial radius in meters. Use this parameter only if you specify `planetModel` as `'Custom'`.

`planetaryGravitional`

Planetary gravitational parameter in meters cubed per second squared. Use this parameter only if you specify `planetModel` as `'Custom'`.

`zonalHarmonicCoeff`

3-element array defining the zonal harmonic coefficients that the function uses to calculate zonal harmonics planetary gravity. Use this parameter only if you specify `planetModel` as `'Custom'`.

`action`

Action for out-of-range input. Specify one:

`'Error'`
`'Warning'`
`'None'` (default)

Output Arguments

`gravityXcoord`

Array of m gravity values in the *x*-axis of the planet-centered planet-fixed coordinates in meters per second squared.

Default:

`gravityYcoord`

Array of m gravity values in the *y*-axis of the planet-centered planet-fixed coordinates in meters per second squared.

Default:

`gravityZcoord`

Array of m gravity values in the *z*-axis of the planet-centered planet-fixed coordinates in meters per second squared.

Default:

Examples

Calculate the gravity in the *x*-axis at the equator on the surface of Earth using the fourth degree model with no warning actions:

```
gx = gravityzonal( [-6378.1363e3 0 0] )

gx =

9.8142
```

Calculate the gravity using the close approximation method at 100 m over the geographic South Pole of Earth with error actions:

```
[gx, gy, gz] = gravityzonal( [0 0 -6356.851e3], 'Error' )

gx =

     0

gy =

     0

gz =

    9.8317
```

Calculate the gravity at 15000 m over the equator and 11000 m over the geographic North Pole using a second order Mars model with warning actions:

```
p  = [2412.648e3 -2412.648e3 0; 0 0 3376.2e3]
[gx, gy, gz] = gravityzonal( p, 'Mars', 2, 'Warning' )
p =

     2412648    -2412648           0
           0           0     3376200

gx =

   -2.6224
         0

gy =

    2.6224
         0

gz =

         0
   -3.7542
```

Calculate the gravity at 15000 m over the equator and 11000 m over the geographic North Pole using a custom planetary model with no actions:

```
p= [2412.648e3 -2412.648e3 0; 0 0 3376e3]
GM      = 42828.371901e9  % m^3/s^2
Re      = 3397e3          % m
```

```
Jvalues = [1.95545367944545e-3 3.14498094262035e-5 ...
-1.53773961526397e-5]
[gx, gy, gz] = gravityzonal( p, 'custom', Re, GM, ...
Jvalues, 'None' )
```

Algorithms

`gravityzonal` is implemented using the following planetary parameter values for each planet:

Planet	**Equatorial Radius (Re) in Meters**	**Gravitational Parameter (GM) in m^3/s^2**	**Zonal Harmonic Coefficients (J Values)**
Earth	6378.1363e3	3.986004415e14	[0.0010826269 -0.0000025323 -0.0000016204]
Jupiter	71492.e3	1.268e17	[0.01475 0 -0.00058]
Mars	3397.2e3	4.305e13	[0.001964 0.000036]
Mercury	2439.0e3	2.2032e13	0.00006
Moon	1738.0e3	4902.799e9	0.0002027
Neptune	24764e3	6.809e15	0.004
Saturn	60268.e3	3.794e16	[0.01645 0 -0.001]
Uranus	25559.e3	5.794e15	0.012
Venus	6052.0e3	3.257e14	0.000027

Alternatives

Zonal Harmonic Gravity Model block

References

Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, McGraw-Hill, New York, 1997.

Fortescue, P., J. Stark, G. Swinerd, (Eds.). *Spacecraft Systems Engineering*, Third Edition, Wiley & Sons, West Sussex, 2003.

Tewari, A., *Atmospheric and Space Flight Dynamics Modeling and Simulation with MATLAB and Simulink*, Birkhäuser, Boston, 2007.

See Also

`geoidegm96` | `gravitywgs84`

Introduced in R2009b

HeadingIndicator Properties

Control heading indicator appearance and behavior

Description

Heading indicators are components that represent a heading indicator. Properties control the appearance and behavior of a heading indicator. Use dot notation to refer to a particular object and property:

```
f = uifigure;
heading = uiaeroheading(f);
heading.Value = 100;
```

The heading indicator displays measurements for aircraft heading in degrees.

The heading indicator represents values between 0 and 360 degrees.

Properties

Heading Indicator

Heading — Location of aircraft heading

`0` (default) | finite, real, and scalar numeric

Location of the aircraft heading, specified as any finite and scalar numeric, in degrees.

- Changing the value changes the direction of the heading. It displays the exact value.

Example: `60`

Dependencies

Specifying this value changes the value of `Value`.

Data Types: `double`

Value — Location of aircraft heading

`0` (default) | finite, real, and scalar numeric

Location of the aircraft heading, specified as any finite and scalar numeric, in degrees.

- Changing the value changes the direction of the heading.

Example: `60`

Dependencies

Specifying this value changes the value of `Heading`.

Data Types: `double`

Interactivity

`Visible` — Visibility of heading indicator
`'on'` (default) | `'off'`

Visibility of the heading indicator, specified as `'on'` or `'off'`. The `Visible` property determines whether the heading indicator, is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire heading indicator is hidden, but you can still specify and access its properties.

`Enable` — Operational state of header indicator
`'on'` (default) | `'off'`

Operational state of header indicator, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the header indicator indicates that the header indicator is operational.
- If you set this property to `'off'`, then the appearance of the header indicator appears dimmed, indicating that the header indicator is not operational.

Position

`Position` — Location and size of header indicator
`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the header indicator relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the header indicator
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the header indicator
`width`	Distance between the right and left outer edges of the header indicator
`height`	Distance between the top and bottom outer edges of the header indicator

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[200 120 120 120]`

`InnerPosition` — Inner location and size of heading indicator
`[100 100 120 120]` (default) | `[left bottom width height]`

Inner location and size of the heading indicator, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of heading indicator
`[100 100 120 120]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the heading indicator returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an heading indicator in the third row and second column of its parent grid.

```
g = ui([4 3]);
gauge = uiaeroheading(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the heading indicator span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this heading indicator spans columns `2` through `3`:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function
`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function
`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption
`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
    - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
    - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing

`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status

`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uiaeroheading'`

This property is read-only.

Type of graphics object, returned as `'uiaeroheading'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also

`uiaeroheading`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

hide

Class: `Aero.Animation`
Package: `Aero`

Hide animation figure

Syntax

```
hide(h)
h.hide
```

Description

`hide(h)` and `h.hide` hide (close) the figure for the animation object `h`. Use `show` to redisplay the animation object figure.

Input Arguments

`h`	Animation object.

Examples

Hide the animation object figure that the `show` method displays.

```
h=Aero.Animation;
h.show;
h.hide;
```

igrfmagm

Calculate Earth magnetic field and secular variation using International Geomagnetic Reference Field

Syntax

```
[magFieldVector,horIntensity,declination,inclination,totalIntensity,
magFieldSecVariation,secVariationHorizontal,secVariationDeclination,
secVariationInclination,secVariationTotal] = igrfmagm(height,
latitude,longitude,decimalYear,generation)
```

Description

`[magFieldVector,horIntensity,declination,inclination,totalIntensity, magFieldSecVariation,secVariationHorizontal,secVariationDeclination, secVariationInclination,secVariationTotal] = igrfmagm(height, latitude,longitude,decimalYear,generation)` calculates the Earth magnetic field and the secular variation at a specific location and time using different generations of the International Geomagnetic Reference Field.

Examples

Calculate the Magnetic Model

Calculate the magnetic model 1000 meters over Natick, Massachusetts on July 4, 2015 using IGRF-12.

```
[mag_field_vector,hor_intensity,declinatioon,inclination,total_intensity] ...
= igrfmagm(1000,42.283,-71.35,decyear(2015,7,4),12)
```

```
mag_field_vector =

   1.0e+04 *

    1.9460   -0.5091    4.8179
```

```
hor_intensity =

   2.0115e+04

declinatioon =

  -14.6612

inclination =

   67.3387

total_intensity =

   5.2209e+04
```

Input Arguments

`height` — Distance
scalar

Distance, in meters, from the surface of the Earth, specified as a scalar.

`latitude` — Geodetic latitude
scalar

Geodetic latitude, in degrees, specified as a scalar. North latitude is positive, and south latitude is negative.

`longitude` — Geodetic longitude
scalar

Geodetic longitude, in degrees, specified as a scalar. East longitude is positive, and west longitude is negative.

`decimalYear` — Year
scalar

Year, in decimal format, specified as a decimal. This value is the desired year, including any fraction of the year that has already passed.

`generation` — Generation version of International Geomagnetic Reference Field
12 (default) | 11

Generation version of the International Geomagnetic Reference Field, specified as `11` or `12`.

Output Arguments

`magFieldVector` — Magnetic field vector
vector

Magnetic field vector, in nanotesla (nT), returned as a vector. *Z* is the vertical component (+ve down).

`horIntensity` — Horizontal intensity
scalar

Horizontal intensity, in nanotesla (nT), returned as a scalar.

`declination` — Declination
scalar

Declination, in degrees (+ve east), returned as a scalar.

`inclination` — Inclination
scalar

Inclination, in degrees (+ve down), returned as a scalar.

`totalIntensity` — Total intensity
scalar

Total intensity, in nanotesla (nT), returned as a scalar.

`magFieldSecVariation` — Secular variation in magnetic field vector
vector

Secular variation in magnetic field vector, in nT/year, returned as a vector. *Z* is the vertical component (+ve down).

`secVariationHorizontal` — Secular variation in horizontal intensity
scalar

Secular variation in horizontal intensity, in nT/year, returned as a scalar.

`secVariationDeclination` — Secular variation in declination
scalar

Secular variation in declination, in minutes/year (+ve east), returned as a scalar.

`secVariationInclination` — Secular variation in inclination
scalar

Secular variation in inclination, in minutes/year (+ve down), returned as a scalar.

`secVariationTotal` — Secular variation in total intensity
scalar

Secular variation in total intensity, in nT/year, returned as a scalar.

Limitation

This function is valid between the heights of –1000 meters to 600,000 meters.

The 11th generation is valid between the years of 1900 and 2015, and 12th generation is valid between the years of 1900 and 2020.

This function has the limitations of the International Geomagnetic Reference Field (IGRF). For more information, see the IGRF website, `https://www.ngdc.noaa.gov/IAGA/vmod/igrfhw.html`.

References

[1] Blakely, R. J. *Potential Theory in Gravity & Magnetic Applications*. Cambridge, UK: Cambridge University Press, 1996.

[2] Lowes, F. J. “The International Geomagnetic Reference Field: A 'Health' Warning.” January, 2010. https://www.ngdc.noaa.gov/IAGA/vmod/igrfhw.html.

See Also

`decyear` | `wrldmagm`

External Websites

https://www.ngdc.noaa.gov/IAGA/vmod/igrfhw.html

Introduced in R2015b

initialize

Class: `Aero.Animation`
Package: `Aero`

Create animation object figure and axes and build patches for bodies

Syntax

```
initialize(h)
h.initialize
```

Description

`initialize(h)` and `h.initialize` create a figure and axes for the animation object `h`, and builds patches for the bodies associated with the animation object. If there is an existing figure, this function

1. Clears out the old figure and its patches.
2. Creates a new figure and axes with default values.
3. Repopulates the axes with new patches using the surface to patch data from each body.

Input Arguments

`h`	Animation object.

Examples

Initialize the animation object, `h`.

```
h = Aero.Animation;
h.initialize();
```

initialize (Aero.FlightGearAnimation)

Set up FlightGear animation object

Syntax

```
initialize(h)
h.initialize
```

Description

`initialize(h)` and `h.initialize` set up the FlightGear version, IP address, and socket for the FlightGear animation object `h`.

Examples

Initialize the animation object, `h`.

```
h = Aero.FlightGearAnimation;
h.initialize();
```

See Also

`GenerateRunScript` | `delete` | `play` | `update`

Introduced in R2007a

initialize (Aero.VirtualRealityAnimation)

Create and populate virtual reality animation object

Syntax

```
initialize(h)
h.initialize
```

Description

`initialize(h)` and `h.initialize` create a virtual reality animation world and populate the virtual reality animation object `h`. If a previously initialized virtual reality animation object exists, and that object has user-specified data, this function saves the previous object to be reset after the initialization.

Examples

Initialize the virtual reality animation object, h.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
```

See Also

`delete` | `play`

Introduced in R2007b

initIfNeeded

Class: `Aero.Animation`
Package: `Aero`

Initialize animation graphics if needed

Syntax

```
initIfNeeded(h)
h.initIfNeeded
```

Description

`initIfNeeded(h)` and `h.initIfNeeded` initialize animation object graphics if necessary.

Input Arguments

`h`	Animation object.

Examples

Initialize the animation object graphics of `h` as needed.

```
h=Aero.Animation;
h.initIfNeeded;
```

juliandate

Julian date calculator

Syntax

```
jd = juliandate(v)
jd = juliandate(s,f)
jd = juliandate(y,mo,d)
jd = juliandate([y,mo,d])
jd = juliandate(y,mo,d,h,mi,s)
jd = juliandate([y,mo,d,h,mi,s])
```

Description

`jd = juliandate(v)` converts one or more date vectors, `v`, into Julian date, `jd`. Input `v` can be an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. `juliandate` returns a column vector of `m` Julian dates, which are the number of days and fractions since noon Universal Time on January 1, 4713 BCE.

A date vector contains six elements, specifying year, month, day, hour, minute, and second. A partial date vector has three elements, specifying year, month, and day. Each element of `v` must be a positive double-precision number.

`jd = juliandate(s,f)` converts one or more dates, `s`, into Julian date, `jd`, using format `f`. `s` can be a character array, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. `juliandate` returns a column vector of `m` Julian dates, where `m` is the number of character vectors in `s`.

All of the dates in `s` must have the same format `f`, composed of the same date format symbols as `datestr`. `juliandate` does not accept formats containing the letter *`Q`*.

If the format does not contain enough information to compute a date number, then:

- Hours, minutes, and seconds default to 0.
- Days default to 1.

- Months default to January.
- Years default to the current year.

Dates with two-character years are interpreted to be within 100 years of the current year.

`jd = juliandate(y,mo,d)` and `jd = juliandate([y,mo,d])` return the decimal year for corresponding elements of the `y,mo,d` (year,month,day) arrays. Specify `y`, `mo`, and `d` as one-dimensional arrays of the same length or scalar values.

`jd = juliandate(y,mo,d,h,mi,s)` and `jd = juliandate([y,mo,d,h,mi,s])` return the Julian dates for corresponding elements of the `y,mo,d,h,mi,s` (year,month,day,hour,minute,second) arrays. Specify the six input arguments as either one-dimensional arrays of the same length or scalar values.

Examples

Calculate Julian date for May 24, 2005:

```
jd = juliandate('24-May-2005','dd-mmm-yyyy')
```

```
jd =

  2.4535e+006
```

Calculate Julian date for December 19, 2006:

```
jd = juliandate(2006,12,19)
```

```
jd =

  2.4541e+006
```

Calculate Julian date for October 10, 2004, at 12:21:00 p.m.:

```
jd = juliandate(2004,10,10,12,21,0)
```

```
jd =

  2.4533e+006
```

Assumptions and Limitations

This function is valid for all common era (CE) dates in the Gregorian calendar.

The calculation of Julian date does not take into account leap seconds.

See Also

`decyear` | `leapyear` | `mjuliandate`

Introduced in R2006b

leapyear

Determine leap year

Syntax

```
ly = leapyear(year)
```

Description

`ly = leapyear(year)` determines whether one or more years are leap years or not. The output, `ly`, is a logical array. `year` should be numeric.

Examples

Determine whether 2005 is a leap year:

```
ly = leapyear(2005)

ly =

  0
```

Determine whether 2000, 2005, and 2020 are leap years:

```
ly = leapyear([2000 2005 2020])

ly =

  1     0     1
```

Assumptions and Limitations

The determination of leap years is done by Gregorian calendar rules.

See Also

`decyear` | `juliandate` | `mjuliandate`

Introduced in R2006b

lla2ecef

Convert geodetic coordinates to Earth-centered Earth-fixed (ECEF) coordinates

Syntax

```
p = lla2ecef(lla)
p = lla2ecef(lla, model)
p = lla2ecef(lla, f, Re)
```

Description

`p = lla2ecef(lla)` converts an `m`-by-3 array of geodetic coordinates (latitude, longitude and altitude), `lla`, to an `m`-by-3 array of ECEF coordinates, `p`. `lla` is in [degrees degrees meters]. `p` is in meters. The default ellipsoid planet is WGS84. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

`p = lla2ecef(lla, model)` is an alternate method for converting the coordinates for a specific ellipsoid planet. Currently only `'WGS84'` is supported for `model`. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

`p = lla2ecef(lla, f, Re)` is another alternate method for converting the coordinates for a custom ellipsoid planet defined by flattening, `f`, and the equatorial radius, `Re`, in meters. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Examples

Determine ECEF coordinates at a latitude, longitude, and altitude:

```
p = lla2ecef([0 45 1000])


p =
  1.0e+006 *

    4.5107    4.5107         0
```

Determine ECEF coordinates at multiple latitudes, longitudes, and altitudes, specifying WGS84 ellipsoid model:

```
p = lla2ecef([0 45 1000; 45 90 2000], 'WGS84')


p =

  1.0e+006 *

    4.5107    4.5107         0
    0.0000    4.5190    4.4888
```

Determine ECEF coordinates at multiple latitudes, longitudes, and altitudes, specifying custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
p = lla2ecef([0 45 1000; 45 90 2000], f, Re)


p =

  1.0e+006 *

    2.4027    2.4027         0
    0.0000    2.4096    2.3852
```

See Also

`ecef2lla` | `geoc2geod` | `geod2geoc`

Introduced in R2006b

lla2eci

Convert geodetic latitude, longitude, altitude (LLA) coordinates to Earth-centered inertial (ECI) coordinates

Syntax

```
position = lla2eci(lla,utc)
```

```
position = lla2eci(lla,utc,reduction)
position = lla2eci(lla,utc,reduction,deltaAT)
position = lla2eci(lla,utc,reduction,deltaAT,deltaUT1)
position = lla2eci(lla,utc,reduction,deltaAT,deltaUT1,polarmotion)
position = lla2eci(lla,utc,reduction,deltaAT,deltaUT1,polarmotion,
Name,Value)
```

Description

`position = lla2eci(lla,utc)` converts geodetic latitude, longitude, altitude (LLA) coordinates to Earth-centered inertial (ECI) position coordinates as an *M*-by-3 array. The conversion is based on the Universal Coordinated Time (UTC) you specify.

`position = lla2eci(lla,utc,reduction)` converts geodetic latitude, longitude, altitude (LLA) coordinates to Earth-centered inertial (ECI) position coordinates as an *M*-by-3 array. The conversion is based on the specified reduction method and the Universal Coordinated Time (UTC) you specify.

`position = lla2eci(lla,utc,reduction,deltaAT)` uses the difference between International Atomic Time and UTC that you specify as `deltaAT` to calculate the ECI coordinates.

`position = lla2eci(lla,utc,reduction,deltaAT,deltaUT1)` uses the difference between UTC and Universal Time (UT1), which you specify as `deltaUT1`, in the calculation.

`position = lla2eci(lla,utc,reduction,deltaAT,deltaUT1,polarmotion)` uses the polar displacement, `polarmotion`, in the calculation.

`position = lla2eci(lla,utc,reduction,deltaAT,deltaUT1,polarmotion, Name,Value)` uses additional options specified by one or more Name,Value pair arguments.

Examples

Convert Position to ECI Coordinates Using UTC

Convert the position to ECI coordinates from LLA coordinates 6 degrees north, 75 degrees west, and 1000 meters altitude at 01/17/2010 10:20:36 UTC.

```
position = lla2eci([6 -75 1000],[2010 1 17 10 20 36])
```

```
position=

   1.0e+06 *

   -6.0744   -1.8289    0.6685
```

Convert Position to ECI coordinates Using UTC and Reduction Method IAU-76/FK5

Convert the position to ECI coordinates from LLA coordinates 55 deg south, 75 deg west, and 500 meters altitude at 01/12/2000 4:52:12.4 UTC. Specify all arguments, including optional ones such as polar motion.

```
position = lla2eci([-55 -75 500],[2000 1 12 4 52 12.4],...
'IAU-76/FK5',32,0.234,[-0.0682e-5 0.1616e-5],...
'dNutation',[-0.2530e-6 -0.0188e-6],...
'flattening',1/290,'RE',60000)
```

```
position=

   1.0e+04 *
   -1.1358    3.2875   -4.9333
```

Input Arguments

`lla` — Latitude, longitude, altitude (LLA) coordinates

M-by-3 array

Latitude, longitude, altitude (LLA) coordinates as *M*-by-3 array of geodetic coordinates, in degrees, degrees, and meters, respectively. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

`utc` — Universal Coordinated Time

1-by-6 array | *M*-by-6 matrix

Universal Coordinated Time (UTC), in the order year, month, day, hour, minutes, and seconds, for which the function calculates the conversion, specified as one of the following.

- For the year value, enter a double value that is a whole number greater than 1, such as `2013`.
- For the month value, enter a double value that is a whole number greater than 0, within the range `1` to `12`.
- For the hour value, enter a double value that is a whole number greater than 0, within the range `1` to `24`.
- For the hour value, enter a double value that is a whole number greater than 0, within the range `1` to `60`.
- For the minute and second values, enter a double value that is a whole number greater than 0, within the range `1` to `60`.

Specify these values in one of the following formats:

- 1-by-6 array

  Specify a 1-row-by-6-column array of UTC values.
- *M*-by-6 matrix

Specify an *M*-by-6 array of UTC values, where *M* is the number of transformation matrices to calculate. Each row corresponds to one set of UTC values.

This is a one row-by-6 column array of UTC values.

Example: `[2000 1 12 4 52 12.4]`

This is an *M*-by-6 array of UTC values, where *M* is 2.

Example: `[2000 1 12 4 52 12.4;2010 6 5 7 22 0]`

Data Types: `double`

`reduction` — Reduction method
`'IAU-2000/2006'` (default) | `'IAU-76/FK5'`

Reduction method to calculate the coordinate conversion, specified as one of the following:

- `'IAU-76/FK5'`

  Reduce the calculation using the International Astronomical Union (IAU)-76/Fifth Fundamental Catalogue (FK5) (IAU-76/FK5) reference system. Choose this reduction method if the reference coordinate system for the conversion is FK5. You can use the `'dNutation'` Name,Value pair with this reduction.

  Note This method uses the IAU 1976 precession model and the IAU 1980 theory of nutation to reduce the calculation. This model and theory are no longer current, but the software provides this reduction method for existing implementations. Because of the polar motion approximation that this reduction method uses, `lla2eci` performs a coordinate conversion that is not orthogonal because of the polar motion approximation.

- `'IAU-2000/2006'`

  Reduce the calculation using the International Astronomical Union (IAU)-2000/2005 reference system. Choose this reduction method if the reference coordinate system for the conversion is IAU-2000. This reduction method uses the P03 precession model to reduce the calculation. You can use the `'dCIP'` Name,Value pair with this reduction.

`deltaAT` — Difference between International Atomic Time and UTC
M-by-1 array of zeroes (default) | scalar | one-dimensional array

Difference between International Atomic Time (IAT) and UTC, in seconds, for which the function calculates the coordinate conversion.

- scalar

  Specify one difference-time value to calculate one direction cosine or transformation matrix.
- one-dimensional array

  Specify a one-dimensional array with *M* elements, where *M* is the number of ECI coordinates. Each row corresponds to one set of ECI coordinates.

Specify `32` seconds as the difference between IAT and UTC.

Example: `32`

Data Types: `double`

`deltaUT1` — Difference between UTC and Universal Time (UT1)

M-by-1 array of zeroes (default) | scalar | one-dimensional array

Difference between UTC and Universal Time (UT1), in seconds, for which the function calculates the coordinate conversion.

- scalar

  Specify one difference-time value to calculate ECI coordinates.
- one-dimensional array

  Specify a one-dimensional array with *M* elements of difference time values, where *M* is the number of ECI coordinates. Each row corresponds to one set of ECI coordinates.

Specify `0.234` seconds as the difference between UTC and UT1.

Example: `0.234`

Data Types: `double`

`polarmotion` — Polar displacement

M-by-2 array of zeroes (default) | 1-by-2 array | *M*-by-2 array

Polar displacement of the Earth, in radians, from the motion of the Earth crust, along the *x*- and *y*-axes.

- 1-by-2 array

  Specify a 1-by-2 array of the polar displacement values to convert one ECI coordinate.
- *M*-by-2 array

  Specify an *M*-by-2 array of polar displacement values, where *M* is the number of ECI coordinates to convert. Each row corresponds to one set of UTC values.

Example: `[-0.0682e-5 0.1616e-5]`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'dNutation',[-0.2530e-6 -0.0188e-6]`

`dNutation` — Adjustment to longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*)
M-by-2 array of zeroes (default) | *M*-by-2 array

Adjustment to the longitude (*dDeltaPsi*) and obliquity (*dDeltaEpsilon*), specified in radians, as the comma-separated pair consisting of `dNutation` and an *M*-by-2 array. You can use this Name,Value pair with the IAU-76/FK5 reduction.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/ Products page.

- *M*-by-2 array

  Specify *M*-by-2 array of adjustment values, where *M* is the number of LLA coordinates to be converted. Each row corresponds to one set of longitude and obliquity values.

Data Types: `double`

`dCIP` — Adjustment to the location of the celestial intermediate pole (CIP)
M-by-2 array of zeroes (default) | *M*-by-2 array

Adjustment to the location of the Celestial Intermediate Pole (CIP), in radians, specified as the comma-separated pair consisting of `dCIP` and an *M*-by-2 array. This location (*dDeltaX*, *dDeltaY*) is along the *x*- and *y*- axes. You can use this argument with the IAU-200/2006 reduction. By default, this function assumes an *M*-by-2 array of zeroes.

For historical values, see the International Earth Rotation and Reference Systems Service website (`https://www.iers.org`) and navigate to the Earth Orientation Data Data/ Products page.

- *M*-by-2 array

  Specify *M*-by-2 array of location adjustment values, where *M* is the number of LLA coordinates to be converted. Each row corresponds to one set of *dDeltaX* and *dDeltaY* values.

Example: `'dcip',[-0.2530e-6 -0.0188e-6]`

Data Types: `double`

`flattening` — Custom ellipsoid planet
1-by-1 array

Custom ellipsoid planet defined by flattening.

Example: `1/290`

Data Types: `double`

`re` — Custom planet ellipsoid radius
1-by-1 array

Custom planet ellipsoid radius, in meters.

Example: `60000`

Data Types: `double`

See Also

`dcmeci2ecef` | `ecef2lla` | `eci2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef`

Introduced in R2014a

lla2flat

Convert from geodetic latitude, longitude, and altitude to flat Earth position

Syntax

```
flatearth_pos = lla2flat(lla, llo, psio, href)
flatearth_pos = lla2flat(lla, llo, psio, href, ellipsoidModel)
flatearth_pos = lla2flat(lla, llo, psio, href, flattening,
equatorialRadius)
```

Description

`flatearth_pos = lla2flat(lla, llo, psio, href)` estimates an array of flat Earth coordinates, *`flatearth_pos`*, from an array of geodetic coordinates, *`lla`*. This function estimates the `flatearth_pos` value with respect to a reference location that `llo`, `psio`, and `href` define.

`flatearth_pos = lla2flat(lla, llo, psio, href, ellipsoidModel)` estimates the coordinates for a specific ellipsoid planet.

`flatearth_pos = lla2flat(lla, llo, psio, href, flattening, equatorialRadius)` estimates the coordinates for a custom ellipsoid planet defined by `flattening` and `equatorialRadius`.

Input Arguments

`lla`

m-by-3 array of geodetic coordinates (latitude, longitude, and altitude), in [degrees, degrees, meters]. Latitude and longitude values can be any value. However, latitude values of +90 and -90 may return unexpected values because of singularity at the poles.

Default:

`llo`

Reference location, in degrees, of latitude and longitude, for the origin of the estimation and the origin of the flat Earth coordinate system.

`psio`

Angular direction of flat Earth *x*-axis (degrees clockwise from north), which is the angle in degrees used for converting flat Earth *x* and *y* coordinates to the North and East coordinates.

`href`

Reference height from the surface of the Earth to the flat Earth frame with regard to the flat Earth frame, in meters.

`ellipsoidModel`

Specific ellipsoid planet model. This function supports only `'WGS84'`.

Default: WGS84

`flattening`

Custom ellipsoid planet defined by flattening.

`equatorialRadius`

Planetary equatorial radius, in meters.

Output Arguments

`flatearth_pos`

Flat Earth position coordinates, in meters.

Examples

Estimate coordinates at latitude, longitude, and altitude:

```
p = lla2flat( [ 0.1 44.95 1000 ], [0 45], 5, -100 )

p =

  1.0e+004 *

    1.0530   -0.6509   -0.0900
```

Estimate coordinates at multiple latitudes, longitudes, and altitudes, specifying the WGS84 ellipsoid model:

```
p = lla2flat( [ 0.1 44.95 1000; -0.05 45.3 2000 ], [0 45], 5, -100, 'WGS84' )

p =

  1.0e+004 *

    1.0530   -0.6509   -0.0900
   -0.2597    3.3751   -0.1900
```

Estimate coordinates at multiple latitudes, longitudes, and altitudes, specifying a custom ellipsoid model:

```
f = 1/196.877360;
Re = 3397000;
p = lla2flat( [ 0.1 44.95 1000; -0.05 45.3 2000 ], [0 45], 5, -100,  f, Re )

p =

  1.0e+004 *

    0.5588   -0.3465   -0.0900
   -0.1373    1.7975   -0.1900
```

Tips

- This function assumes that the flight path and bank angle are zero.
- This function assumes that the flat Earth *z*-axis is normal to the Earth only at the initial geodetic latitude and longitude. This function has higher accuracy over small distances from the initial geodetic latitude and longitude. It also has higher accuracy at distances closer to the equator. The function calculates a longitude with higher accuracy when the variations in latitude are smaller. Additionally, longitude is singular at the poles.

Algorithms

The estimation begins by finding the small changes in latitude and longitude from the output latitude and longitude minus the initial latitude and longitude.

$$d\mu = \mu - \mu_0$$

$$d_\iota = \iota - \iota_0$$

To convert geodetic latitude and longitude to the North and East coordinates, the estimation uses the radius of curvature in the prime vertical (R_N) and the radius of curvature in the meridian (R_M). R_N and R_M are defined by the following relationships:

$$R_N = \frac{R}{\sqrt{1-(2f-f^2)\sin^2\mu_0}}$$

$$R_M = R_N\frac{1-(2f-f^2)}{1-(2f-f^2)\sin^2\mu_0}$$

where (R) is the equatorial radius of the planet and f is the flattening of the planet.

Small changes in the North (dN) and East (dE) positions are approximated from small changes in the North and East positions by

$$dN = \frac{d\mu}{\operatorname{atan}\left(\frac{1}{R_M}\right)}$$

$$dE = \frac{d\iota}{\operatorname{atan}\left(\frac{1}{R_N\cos\mu_0}\right)}$$

With the conversion of the North and East coordinates to the flat Earth x and y coordinates, the transformation has the form of

$$\begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} N \\ E \end{bmatrix}$$

where

$$(\psi)$$

is the angle in degrees clockwise between the x-axis and north.

The flat Earth *z*-axis value is the negative altitude minus the reference height (h_{ref}).

$$p_z = -h - h_{ref}$$

References

Etkin, B., *Dynamics of Atmospheric Flight*. New York: John Wiley & Sons, 1972.

Stevens, B. L., and F. L. Lewis, *Aircraft Control and Simulation*, 2nd ed. New York: John Wiley & Sons, 2003.

See Also

`flat2lla`

Introduced in R2011a

load (Aero.Body)

Get geometry data from source

Syntax

```
load(h, bodyDataSrc)
h.load(bodyDataSrc)
load(h, bodyDataSrc, geometrysource)
h.load(bodyDataSrc, geometrysource)
```

Description

`load(h, bodyDataSrc)` and `h.load(bodyDataSrc)` load the graphics data from the body graphics file. This command assumes a default geometry source type set to `Auto`.

`load(h, bodyDataSrc, geometrysource)` and `h.load(bodyDataSrc, geometrysource)` load the graphics data from the body graphics file, `bodyDataSrc`, into the face, vertex, and color data of the animation body object `h`. Then, when axes `ax` is available, you can use this data to generate patches with `generatePatches`. `geometrysource` is the geometry source type for the body.

By default *`geometrysource`* is set to `Auto`, which recognizes `.mat` extensions as MAT-files, `.ac` extensions as Ac3d files, and structures containing fields of `name`, `faces`, `vertices`, and `cdata` as MATLAB variables. If you want to use alternate file extensions or file types, enter one of the following:

- `Auto`
- `Variable`
- `MatFile`
- `Ac3d`
- `Custom`

Examples

Load the graphic data from the graphic data file, `pa24-250_orange.ac`, into `b`.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
```

See Also

`generatePatches` | `move` | `update`

Introduced in R2007a

machnumber

Compute Mach number using velocity and speed of sound

Syntax

```
mach = machnumber(v, a)
```

Description

`mach = machnumber(v, a)` computes `m` Mach numbers, `mach`, from an `m`-by-3 array of velocities, `v`, and an array of `m` speeds of sound, `a`. `v` and `a` must have the same length units.

Examples

Determine the Mach number for velocity and speed of sound in feet per second:

```
mach = machnumber([84.3905 33.7562 10.1269], 1116.4505)


mach =

    0.0819
```

Determine the Mach number for velocity and speed of sound in meters per second:

```
mach = machnumber([25.7222 10.2889 3.0867], [340.2941 295.0696])


mach =

    0.0819    0.0945
```

Determine the Mach number for velocity and speed of sound in knots:

```
mach = machnumber([50 20 6; 5 0.5 2], [661.4789 573.5694])
mach =

    0.0819
    0.0094
```

See Also

`airspeed` | `alphabeta` | `dpressure`

Introduced in R2006b

mjuliandate

Modified Julian date calculator

Syntax

```
mjd = mjuliandate(v)
mjd = mjuliandate(s,f)
mjd = mjuliandate(y,mo,d)
mjd = mjuliandate([y,mo,d])
mjd = mjuliandate(y,mo,d,h,mi,s)
mjd = mjuliandate([y,mo,d,h,mi,s])
```

Description

`mjd = mjuliandate(v)` converts one or more date vectors, `v`, into modified Julian date, `mjd`. Input `v` can be an `m`-by-6 or `m`-by-3 matrix containing `m` full or partial date vectors, respectively. `mjuliandate` returns a column vector of `m` modified Julian dates. Modified Julian dates begin at midnight rather than noon, and the first two digits of its corresponding Julian date are removed.

A date vector contains six elements, specifying year, month, day, hour, minute, and second. A partial date vector has three elements, specifying year, month, and day. Each element of `v` must be a positive double-precision number.

`mjd = mjuliandate(s,f)` converts one or more dates, `s`, into modified Julian date, `mjd`, using format `f`. `s` can be a character array, where each row corresponds to one date character vector, or a one-dimensional cell array of character vectors. `mjuliandate` returns a column vector of `m` modified Julian dates, where `m` is the number of character vectors in `s`.

All of the dates in `s` must have the same format `f`, composed of the same date format symbols as the `datestr` function. `mjuliandate` does not accept formats containing the letter *`Q`*.

If a format does not contain enough information to compute a date number, then:

- Hours, minutes, and seconds default to 0.
- Days default to 1.
- Months default to January.
- Years default to the current year.

Dates with two-character years are interpreted to be within 100 years of the current year.

`mjd = mjuliandate(y,mo,d)` and `mjd = mjuliandate([y,mo,d])` return the decimal year for corresponding elements of the `y,mo,d` (year,month,day) arrays. Specify `y`, `mo`, and `d` as one-dimensional arrays of the same length or scalar values.

`mjd = mjuliandate(y,mo,d,h,mi,s)` and `mjd = mjuliandate([y,mo,d,h,mi,s])` return the modified Julian dates for corresponding elements of the `y,mo,d,h,mi,s` (year,month,day,hour,minute,second) arrays. Specify the six arguments as one-dimensional arrays of the same length or scalar values.

Examples

Calculate the modified Julian date for May 24, 2005:

```
mjd = mjuliandate('24-May-2005','dd-mmm-yyyy')
```

```
mjd =

  53514
```

Calculate the modified Julian date for December 19, 2006:

```
mjd = mjuliandate(2006,12,19)
```

```
mjd =

  54088
```

Calculate the modified Julian date for October 10, 2004, at 12:21:00 p.m.:

```
mjd = mjuliandate(2004,10,10,12,21,0)
```

```
mjd =

  5.3289e+004
```

Assumptions and Limitations

This function is valid for all common era (CE) dates in the Gregorian calendar.

The calculation of modified Julian date does not take into account leap seconds.

See Also

`decyear` | `juliandate` | `leapyear`

Introduced in R2006b

moonLibration

Moon librations

Syntax

```
angles= moonLibration(ephemerisTime)
angles= moonLibration(ephemerisTime,ephemerisModel)
angles= moonLibration(ephemerisTime,ephemerisModel,action)

[angles,rates] = moonLibration( ___ )
```

Description

`angles= moonLibration(ephemerisTime)` implements the Moon libration angles for `ephemerisTime`, expressed in Julian days.

The function uses the Chebyshev coefficients that the NASA Jet Propulsion Laboratory provides.

This function requires that you download ephemeris data with the Add-On Explorer. For more information, see `aeroDataPackage`.

`angles= moonLibration(ephemerisTime,ephemerisModel)` uses the `ephemerisModel` coefficients to implement these values.

`angles= moonLibration(ephemerisTime,ephemerisModel,action)` uses `action` to determine error reporting.

`[angles,rates] = moonLibration( ___ )` implements the Moon libration angles and rates using any combination of the input arguments in the previous syntaxes.

Examples

Implement Libration Angles of Moon

Implement libration angles of the Moon for December 1, 1990 with DE405. Use the juliandate function to calculate the input Julian date value.

```
angles = moonLibration(juliandate(1990,12,1))
```

```
angles =
   1.0e+03 *
    0.0001    0.0004    1.8010
```

Implement Libration Angles and Rates for Moon

Specify the ephemerides (DE421) and use the juliandate function for the date (January 1, 2000) to calculate both the Moon libration angles and rates.

```
[angles,rates] = moonLibration([2451544.5 0.5],'421')
```

```
angles =
   1.0e+03 *
   -0.0001    0.0004    2.5643

rates =
   -0.0001    0.0000    0.2301
```

Input Arguments

`ephemerisTime` — Julian dates
scalar | 2-element vector | column vector | *M*-by-2 matrix

Julian dates for which the positions are calculated, specified as one of the following:

- Scalar

  Specify one fixed Julian date.
- 2-element vector

  Specify the Julian date in multiple parts. The first element is the Julian date for a specific epoch that is the most recent midnight at or before the interpolation epoch. The second element is the fractional part of a day elapsed between the first element and epoch. The second element must be positive. The value of the first element plus the second element cannot exceed the maximum Julian date.

- Column vector

  Specify a column vector with *M* elements, where *M* is the number of Julian dates.
- *M*-by-2 matrix

  Specify a matrix, where *M* is the number of Julian dates and the second column contains the elapsed days (Julian epoch date/elapsed day pairs).

Data Types: `double`

`ephemerisModel` — Ephemerides coefficients
`'405'` (default) | `'421'` | `'423'` | `'430'` | `'432t'`

Ephemerides coefficients, specified as one of these ephemerides defined by the Jet Propulsion Laboratory:

- `'405'`

  Released in 1998. This ephemerides takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
- `'421'`

  Released in 2008. This ephemerides takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
- `'423'`

  Released in 2010. This ephemerides takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
- `'430'`

Released in 2013. This ephemerides takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650).

This function implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

- `'432t'`

  Released in April 2014. This ephemerides takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650).

  This function implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Data Types: `char`

`action` — Function behavior
`'Error'` (default) | `'None'` | `'Warning'`

Function behavior when inputs are out of range, specified as one of these values:

Value	Description
`'None'`	No action.
`'Warning'`	Warning in the MATLAB Command Window, model simulation continues.
`'Error'`	MATLAB returns an exception, model simulation stops.

Data Types: `char`

Output Arguments

`angles` — Moon libration angles
M-by-3 numeric array

Moon libration angles, specified as an *M*-by-3 numeric array. *M* is the number of Julian dates, in rows. The columns contain the Euler angles (φ θ ψ) for Moon attitude, in radians.

If the input arguments include multiple Julian dates or epochs, this array has the same number of rows as the `ephemerisTime` input.

`rates` — Moon libration angular rates
M-by-3 numeric array

Moon libration angular rates, specified as an *M*-by-3 numeric array. *M* is the number of Julian dates, in rows. The columns contain the Moon libration Euler angular rates (ω), in radians/day.

If the input arguments include multiple Julian dates or epochs, this array has the same number of rows as the `ephemerisTime` input.

References

[1] Folkner, W. M., J. G. Williams, D. H. Boggs, “The Planetary and Lunar Ephemeris DE 421,” *JPL Interplanetary Network Progress Report 24-178*, 2009.

[2] Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, McGraw-Hill, New York, 1997.

See Also

`earthNutation` | `juliandate` | `planetEphemeris`

External Websites

`https://ssd.jpl.nasa.gov/?planet_eph_export`
`https://syrte.obspm.fr/jsr/journees2010/powerpoint/folkner.pdf`

Introduced in R2013a

move (Aero.Body)

Change animation body position and orientation

Syntax

```
move(h, translation, rotation)
h.move(translation,rotation)
```

Description

`move(h, translation, rotation)` and `h.move(translation,rotation)` set a new position and orientation for the body object `h`. `translation` is a 1-by-3 matrix in the aerospace body *x-y-z* coordinate system. `rotation` is a 1-by-3 matrix, in radians, that specifies the rotations about the right-hand *x-y-z* sequence of coordinate axes. The order of application of the rotation is *z-y-x* (*r-q-p*).

Examples

Change animation body position to *newpos* and *newrot*.

```
h = Aero.Body;
h.load('ac3d_xyzisrgb.ac','Ac3d');
newpos = h.Position + 1.00;
newrot = h.Rotation + 0.01;
h.move(newpos,newrot);
```

See Also

`load`

Introduced in R2007a

move (Aero.Node)

Change node translation and rotation

Syntax

```
move(h,translation,rotation)
h.move(translation,rotation)
```

Description

`move(h,translation,rotation)` and `h.move(translation,rotation)` set a new position and orientation for the node object `h`. `translation` is a 1-by-3 matrix in the aerospace body *x*-*y*-*z* coordinate system or another coordinate system. In the latter case, you can use the `CoordTransformFcn` function to move it into an aerospace body. The defined aerospace body coordinate system is defined relative to the screen as *x*-left, *y*-in, *z*-down.

`rotation` is a 1-by-3 matrix, in radians, that specifies the rotations about the right-hand *x*-*y*-*z* sequence of coordinate axes. The order of application of the rotation is *z*-*y*-*x* (*r*-*q*-*p*). This function uses the `CoordTransformFcn` to apply the translation and rotation from the input coordinate system to the aerospace body. The function then moves the translation and rotation from the aerospace body to the VRML *x*-*y*-*z* coordinates. The defined VRML coordinate system is defined relative to the screen as *x*-right, *y*-up, *z*-out.

Examples

Move the Lynx body. This example uses the Simulink 3D Animation `vrnode/getfield` function to retrieve the translation and rotation. These coordinates are those used in the Simulink 3D Animation software.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
newtrans = getfield(h.Nodes{4}.VRNode,'translation') + 1.0;
```

```
newrot = getfield(h.Nodes{4}.VRNode,'rotation') + [.2 0.01 0.01 0.01];
h.Nodes{4}.move(newtrans,newrot);
```

Limitations

This function cannot get the node position in aerospace body coordinates; it needs to use the `CoordTransformFcn` to do so.

This function cannot set a viewpoint position or orientation (see `addViewpoint`).

See Also

`addNode`

Introduced in R2007b

moveBody

Class: `Aero.Animation`
Package: `Aero`

Move body in animation object

Syntax

```
moveBody(h,idx,translation,rotation)
h.moveBody(idx,translation,rotation)
```

Description

`moveBody(h,idx,translation,rotation)` and `h.moveBody(idx,translation,rotation)` set a new position and attitude for the body specified with the index `idx` in the animation object `h`. `translation` is a 1-by-3 matrix in the aerospace body coordinate system. `rotation` is a 1-by-3 matrix, in radians, that specifies the rotations about the right-hand *x-y-z* sequence of coordinate axes. The order of application of the rotation is *z-y-x (R-Q-P)*.

Input Arguments

`h`	Animation object.
`translation`	1-by-3 matrix in the aerospace body coordinate system.
`rotation`	1-by-3 matrix, in radians, that specifies the rotations about the right-hand *x-y-z* sequence of coordinate axes.
`idx`	Body specified with this index.

Examples

Move the body with the index 1 to position offset from the original by `+ [0 0 -3]` and rotation, *rot1*.

```
h = Aero.Animation;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
pos1 = h.Bodies{1}.Position;
rot1 = h.Bodies{1}.Rotation;
h.moveBody(1,pos1 + [0 0 -3],rot1);
```

Node (Aero.Node)

Create node object for use with virtual reality animation

Syntax

```
h = Aero.Node
```

Description

`h = Aero.Node` creates a node object for use with virtual reality animation.

See `Aero.Node` for further details.

See Also

`Aero.Node`

Introduced in R2007b

nodeInfo (Aero.VirtualRealityAnimation)

Create list of nodes associated with virtual reality animation object

Syntax

```
nodeInfo(h)
h.nodeInfo
n = nodeInfo(h)
n = h.nodeInfo
```

Description

`nodeInfo(h)` and `h.nodeInfo` create a list of nodes associated with the virtual reality animation object, `h`.

`n = nodeInfo(h)` and `n = h.nodeInfo` create a cell array (`n`) that contains the node information. The function stores the information in a cell array as follows:

```
N{1,n} = Node Index
N{2,n} = Node Name
N{3,n} = Node Type
```

where `n` is the number of nodes. You might want to use this function to find an existing node by name and then perform a certain action on it using the node index.

Examples

Create list of nodes associated with virtual reality animation object, `h`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
h.initialize();
h.nodeInfo;
```

See Also

`addNode`

Introduced in R2007b

planetEphemeris

Position and velocity of astronomical objects

Syntax

```
position= planetEphemeris(ephemerisTime,center,target)
position = planetEphemeris(ephemerisTime,center,target,
ephemerisModel)
position = planetEphemeris(ephemerisTime,center,target,
ephemerisModel,units)
position= planetEphemeris(ephemerisTime,center,target,
ephemerisModel,units,action)
```

```
[position,velocity] = planetEphemeris( ___ )
```

Description

`position= planetEphemeris(ephemerisTime,center,target)` implements the position of the target object relative to the specified center object for a given Julian date `ephemerisTime`. By default, the function implements the position based on the DE405 ephemerides in units of km.

The function uses the Chebyshev coefficients that the NASA Jet Propulsion Laboratory provides.

This function requires that you download ephemeris data with the Add-On Explorer. For more information, see `aeroDataPackage`.

`position = planetEphemeris(ephemerisTime,center,target, ephemerisModel)` uses the `ephemerisModel` coefficients to implement these values.

`position = planetEphemeris(ephemerisTime,center,target, ephemerisModel,units)` specifies the units for these values.

`position= planetEphemeris(ephemerisTime,center,target, ephemerisModel,units,action)` uses `action` to determine error reporting.

`[position,velocity] = planetEphemeris( ___ )` implements the position and velocity of a the target object relative to the specified center for a given Julian date `ephemerisTime` using any of the input arguments in the previous syntaxes.

Examples

Implement Position of Moon

Implement the position of the Moon with respect to the Earth for December 1, 1990 with DE405:

```
position = planetEphemeris(juliandate(1990,12,1),'Earth','Moon')
```

```
position =
   1.0e+05 *
    2.3112    2.3817    1.3595
```

Implement Position and Velocity for Saturn

Implement the position and velocity for Saturn with respect to the Solar System barycenter for noon on January 1, 2000 using DE421 and AU units:

```
[position,velocity] = planetEphemeris([2451544.5 0.5],...
'SolarSystem','Saturn','421','AU')
```

```
position =
    6.3993    6.1720    2.2738
velocity =
   -0.0043    0.0035    0.0016
```

Input Arguments

`ephemerisTime` — Julian date
scalar | 2-element vector | column vector | *M*-by-2 matrix

Julian date for which the positions are calculated, specified as one of the following:

- Scalar

  Specify one fixed Julian date.
- 2-element vector

  Specify the Julian date in multiple parts. The first element is the Julian date for a specific epoch that is the most recent midnight at or before the interpolation epoch. The second element is the fractional part of a day elapsed between the first element and epoch. The second element must be positive. The value of the first element plus the second element cannot exceed the maximum Julian date.
- Column vector

  Specify a column vector with *M* elements, where *M* is the number of fixed Julian dates.
- *M*-by-2 matrix

  Specify a matrix, where *M* is the number of Julian dates and the second column contains the elapsed days (Julian epoch date/elapsed day pairs).

Data Types: `double`

`center` — Reference body (astronomical object) or point of reference
`'Sun'` | `'Mercury'` | `'Venus'` | `'Earth'` | `'Moon'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Pluto'` | `'SolarSystem'` | `'EarthMoon'`

Reference body (astronomical object) or point of reference from which to measure the target barycenter position and velocity.

Data Types: `char`

`target` — Target body (astronomical object) or point of reference
`'Sun'` | `'Mercury'` | `'Venus'` | `'Earth'` | `'Moon'` | `'Mars'` | `'Jupiter'` | `'Saturn'` | `'Uranus'` | `'Neptune'` | `'Pluto'` | `'SolarSystem'` | `'EarthMoon'`

Target body (astronomical object) or point of reference of the barycenter position and velocity measurement.

Data Types: `char`

`ephemerisModel` — Ephemerides coefficients
`'405'` (default) | `'421'` | `'423'` | `'430'` | `'432t'`

Ephemerides coefficients, specified as one of these ephemerides defined by the Jet Propulsion Laboratory:

- `'405'`

  Released in 1998. This ephemerides takes into account the Julian date range 2305424.50 (December 9, 1599) to 2525008.50 (February 20, 2201).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
- `'421'`

  Released in 2008. This ephemerides takes into account the Julian date range 2414992.5 (December 4, 1899) to 2469808.5 (January 2, 2050).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 1.0, adopted in 1998.
- `'423'`

  Released in 2010. This ephemerides takes into account the Julian date range 2378480.5 (December 16, 1799) to 2524624.5 (February 1, 2200).

  This function calculates these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
- `'430'`

  Released in 2013. This ephemerides takes into account the Julian date range 2287184.5 (December 21, 1549) to 2688976.5 (January 25, 2650).

  This function implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.
- `'432t'`

  Released in April 2014. This ephemerides takes into account the Julian date range 2287184.5, (December 21, 1549) to 2688976.5, (January 25, 2650).

  This function implements these ephemerides with respect to the International Celestial Reference Frame version 2.0, adopted in 2010.

Data Types: `char`

`units` — Output units
`'km'` (default) | `'AU'`

Output units for position and velocity, specified as `'km'` for km and km/s or `'AU'` for astronomical units or AU/day.

Data Types: `char`

`action` — Function behavior
`'Error'` (default) | `'None'` | `'Warning'`

Function behavior when inputs are out of range.

Value	Description
`'None'`	No action.
`'Warning'`	Warning in the MATLAB Command Window, model simulation continues.
`'Error'`	MATLAB returns an exception, model simulation stops.

Data Types: `char`

Output Arguments

`position` — Barycenter position
M-by-3 vector

Barycenter position of the `target` object relative to the barycenter of the `center` object, returned as an *M*-by-3 vector, where *M* is the number of Julian dates. The 3 vector contains the *x*, *y*, and *z* of the position along the International Celestial Reference Frame (ICRF). Units are km or astronomical units (AU). If input arguments include multiple Julian dates or epochs, this vector has the same number of rows as the `ephemerisTime` input.

`velocity` — Barycenter velocity
M-by-3 vector

Barycenter velocity of the `target` object relative to the barycenter of the `center` object, returned as an *M*-by-3 vector, where *M* is the number of Julian dates. The 3 vector contains the velocity in the *x*, *y*, and *z* directions along the ICRF. Velocity of the Units are km or astronomical units (AU). If the input includes multiple Julian dates or epochs, this vector has the same number of rows as the `ephemerisTime` input.

References

[1] Folkner, W. M., J. G. Williams, D. H. Boggs, “The Planetary and Lunar Ephemeris DE 421,” *JPL Interplanetary Network Progress Report 24-178*, 2009.

[2] Ma, C. et al., “The International Celestial Reference Frame as Realized by Very Long Baseline Interferometry,” *Astronomical Journal*, Vol. 116, 516–546, 1998.

[3] Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, McGraw-Hill, New York, 1997.

See Also

`earthNutation` | `juliandate` | `moonLibration`

Topics

“Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-141
“Marine Navigation Using Planetary Ephemerides” on page 5-129

External Websites

`https://ssd.jpl.nasa.gov/?planet_eph_export`

Introduced in R2013a

play

Class: `Aero.Animation`
Package: `Aero`

Animate `Aero.Animation` object given position/angle time series

Syntax

```
play(h)
h.play
```

Description

`play(h)` and `h.play` animate the loaded geometry in `h` for the current `TimeseriesDataSource` at the specified rate given by the `'TimeScaling'` property (in seconds of animation data per second of wall-clock time) and animated at a certain number of frames per second using the `'FramesPerSecond'` property.

The time series data is interpreted according to the `'TimeseriesSourceType'` property, which can be one of:

`'Timeseries'`	MATLAB time series data with six values per time: `x y z phi theta psi` The values are resampled.
`'Simulink.Timeseries'`	Simulink.Timeseries (Simulink signal logging): • First data item `x y z` • Second data item `phi theta psi`

`'StructureWithTime'`	Simulink struct with time (for example, Simulink root outport logging `'Structure with time'`): • `signals(1).values: x y z` • `signals(2).values: phi theta psi` Signals are linearly interpolated vs. time using `interp1`.
`'Array6DoF'`	A double-precision array in `n` rows and 7 columns for 6-DoF data: `time x y z phi theta psi`. If a double-precision array of 8 or more columns is in `'TimeseriesSource'`, the first 7 columns are used as 6-DoF data.
`'Array3DoF'`	A double-precision array in `n` rows and 4 columns for 3-DoF data: `time x z theta`. If a double-precision array of 5 or more columns is in `'TimeseriesSource'`, the first 4 columns are used as 3-DoF data.
`'Custom'`	Position and angle data is retrieved from `'TimeseriesSource'` by the currently registered `'TimeseriesReadFcn'`.

The following are limitations for the `TStart` and `TFinal` values:

- `TStart` and `TFinal` must be numeric.
- `TStart` and `TFinal` cannot be Inf or NaN.
- `TFinal` must be greater than or equal to `TStart`.
- `TFinal` cannot be greater than the maximum `Timeseries` time.
- `TStart` cannot be less than the minimum `Timeseries` time.

The time advancement algorithm used by `play` is based on animation frames counted by ticks:

```
ticks = ticks + 1;
time  = tstart + ticks*FramesPerSecond*TimeScaling;
```

where

`TimeScaling`	Specify the seconds of animation data per second of wall-clock time.

`FramesPerSecond`	Specify the number of frames per second used to animate the `'TimeseriesSource'`.

For default `'TimeseriesReadFcn'` methods, the last frame played is the last time value.

Time is in seconds, position values are in the same units as the geometry data loaded into the animation object, and all angles are in radians.

Note If there is a 15% difference between the expected time advance and the actual time advance, this method will generate the following warning:

```
TimerPeriod has been set to <value>. You may wish to modify the animation
TimeScaling and FramesPerSecond properties to compensate for the
millisecond limit of the TimerPeriod.  See documentation for details.
```

Input Arguments

`h`	Animation object.

Examples

Animate the body, `idx1`, for the duration of the time series data.

```
h = Aero.Animation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
load simdata;
h.Bodies{1}.TimeSeriesSource = simdata;
h.show();
h.play();
```

play (Aero.FlightGearAnimation)

Animate FlightGear flight simulator using given position/angle time series

Syntax

```
play(h)
h.play
```

Description

`play(h)` and `h.play` animate FlightGear flight simulator using specified time series data in `h`. The time series data can be set in `h` by using the property `'TimeseriesSource'`.

The time series data, stored in the property `'TimeseriesSource'`, is interpreted according to the `'TimeseriesSourceType'` property, which can be one of:

`'Timeseries'`	MATLAB time series data with six values per time: `latitude longitude altitude phi theta psi` The values are resampled.
`'StructureWithTime'`	Simulink struct with time (for example, Simulink root outport logging `'Structure with time'`): • `signals(1).values: latitude longitude altitude` • `signals(2).values: phi theta psi` Signals are linearly interpolated vs. time using `interp1`.
`'Array6DoF'`	A double-precision array in `n` rows and 7 columns for 6-DoF data: `time latitude longitude altitude phi theta psi`. If a double-precision array of 8 or more columns is in `'TimeseriesSource'`, the first 7 columns are used as 6-DoF data.

`'Array3DoF'`	A double-precision array in `n` rows and 4 columns for 3-DoF data: `time latitude altitude theta`. If a double-precision array of 5 or more columns is in `'TimeseriesSource'`, the first 4 columns are used as 3-DoF data.
`'Custom'`	Position and angle data is retrieved from `'TimeseriesSource'` by the currently registered `'TimeseriesReadFcn'`.

The time advancement algorithm used by `play` is based on animation frames counted by ticks:

```
ticks = ticks + 1;
time  = tstart + ticks*FramesPerSecond*TimeScaling;
```

where

`TimeScaling`	Specify the seconds of animation data per second of wall-clock time.
`FramesPerSecond`	Specify the number of frames per second used to animate the `'TimeseriesSource'`.

For default `'TimeseriesReadFcn'` methods, the last frame played is the last time value.

Time is in seconds, position values are in the same units as the geometry model to be used by FlightGear (see the property `'GeometryModelName'`), and all angles are in radians. A possible result of using incorrect units is the early termination of the FlightGear flight simulator.

Note If there is a 15% difference between the expected time advance and the actual time advance, this method will generate the following warning:

```
TimerPeriod has been set to <value>. You may wish to modify the animation
TimeScaling and FramesPerSecond properties to compensate for the
millisecond limit of the TimerPeriod.  See documentation for details.
```

The `play` method supports FlightGear animation objects with custom timers.

Limitations

The following are limitations for the `TStart` and `TFinal` values:

- `TStart` and `TFinal` must be numeric.
- `TStart` and `TFinal` cannot be Inf or NaN.
- `TFinal` must be greater than or equal to `TStart`.
- `TFinal` cannot be greater than the maximum `Timeseries` time.
- `TStart` cannot be less than the minimum `Timeseries` time.

Examples

Animate FlightGear flight simulator using the given `'Array3DoF'` position/angle time series data:

```
data = [86.2667 -2.13757034184404 7050.896596 -0.135186746141248;...
        87.2833 -2.13753906554384 6872.545051 -0.117321084678936;...
        88.2583 -2.13751089592972 6719.405713 -0.145815609299676;...
        89.275  -2.13747984652232 6550.117118 -0.150635248762596;...
        90.2667 -2.13744993157894 6385.05883  -0.143124782831999;...
        91.275  -2.13742019116849 6220.358163 -0.147946202530756;...
        92.275  -2.13739055547779 6056.906647 -0.167529704309343;...
        93.2667 -2.13736104196014 5892.356118 -0.152547361677911;...
        94.2583 -2.13733161570895 5728.201718 -0.161979312941906;...
        95.2583 -2.13730231163081 5562.923808 -0.122276929636682;...
        96.2583 -2.13727405475022 5406.736322 -0.160421658944379;...
        97.2667 -2.1372440001805  5239.138477 -0.150591353731908;...
        98.2583 -2.13721598764601 5082.78798  -0.147737722951605];
h = fganimation
h.TimeseriesSource = data
h.TimeseriesSourceType = 'Array3DoF'
play(h)
```

Animate FlightGear flight simulator using the custom timer, `MyFGTimer`.

```
h.play('MyFGTimer')
```

See Also

`GenerateRunScript` | `initialize` | `update`

Introduced in R2007a

play (Aero.VirtualRealityAnimation)

Animate virtual reality world for given position and angle in time series data

Syntax

```
play(h)
h.play
```

Description

`play(h)` and `h.play` animate the virtual reality world in `h` for the current `TimeseriesDataSource` at the specified rate given by the `'TimeScaling'` property (in seconds of animation data per second of wall-clock time) and animated at a certain number of frames per second using the `'FramesPerSecond'` property.

The time series data is interpreted according to the `'TimeseriesSourceType'` property, which can be one of:

`'timeseries'`	MATLAB time series data with six values per time: `x y z phi theta psi` The values are resampled.

`'Simulink.Timeseries'`	Simulink.Timeseries (Simulink signal logging):
	• First data item
	`x y z`
	• Second data item
	`phi theta psi`

`'StructureWithTime'`	Simulink struct with time (for example, Simulink root outport logging `'Structure with time'`): • `signals(1).values: x y z` • `signals(2).values: phi theta psi` Signals are linearly interpolated vs. time using `interp1`.
`'Array6DoF'`	A double-precision array in `n` rows and 7 columns for 6-DoF data: `time x y z phi theta psi`. If a double-precision array of 8 or more columns is in `'TimeseriesSource'`, the first 7 columns are used as 6-DoF data.
`'Array3DoF'`	A double-precision array in `n` rows and 4 columns for 3-DoF data: `time x z theta`. If a double-precision array of 5 or more columns is in `'TimeseriesSource'`, the first 4 columns are used as 3-DoF data.
`'Custom'`	Position and angle data is retrieved from `'TimeseriesSource'` by the currently registered `'TimeseriesReadFcn'`.

The time advancement algorithm used by `play` is based on animation frames counted by ticks:

```
ticks = ticks + 1;
time  = tstart + ticks*FramesPerSecond*TimeScaling;
```

where

`TimeScaling`	Specify the seconds of animation data per second of wall-clock time.
`FramesPerSecond`	Specify the number of frames per second used to animate the `'TimeseriesSource'`.

For default `'TimeseriesReadFcn'` methods, the last frame played is the last time value.

Time is in seconds, position values are in the same units as the geometry data loaded into the animation object, and all angles are in radians.

Examples

Animate virtual reality world, `asttkoff`.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
load takeoffData
[~, idxPlane] = find(strcmp('Plane', h.nodeInfo));
h.Nodes{idxPlane}.TimeseriesSource = takeoffData;
h.Nodes{idxPlane}.TimeseriesSourceType = 'StructureWithTime';
h.Nodes{idxPlane}.CoordTransformFcn = @vranimCustomTransform;
h.play();
```

See Also

`initialize`

Introduced in R2007b

polarMotion

Calculate Earth polar motion

Syntax

```
polarmotion=polarMotion(utc)
[polarmotion,polarmotionError]=polarMotion(utc)

polarmotion=polarMotion(utc,Name,Value)
[polarmotion,polarmotionError]=polarMotion(utc,Name,Value)
```

Description

`polarmotion=polarMotion(utc)` calculates the movement of the rotation axis with respect to the crust of the Earth for a specific Universal Coordinated Time (UTC), specified as a modified Julian date. By default, this function uses a prepopulated list of IAU 2000A Earth orientation (IERS) data. This list contains measured and calculated (predicted) data supplied by the IERS. The IERS measures and calculates this data for a set of predetermined dates.

`[polarmotion,polarmotionError]=polarMotion(utc)` calculates the error for the movement of the rotation axis with respect to the crust of the Earth.

`polarmotion=polarMotion(utc,Name,Value)` calculates the movement of the rotation axis with respect to the crust of the Earth using additional options specified by one or more `Name,Value` pair arguments.

`[polarmotion,polarmotionError]=polarMotion(utc,Name,Value)` calculates the error for the movement of the rotation axis with respect to the crust of the Earth.

Examples

Calculate Polar Motion

Calculate the polar motion for December 28, 2015.

```
mjd = mjuliandate(2015,12,28)
polarmotion = polarMotion(mjd)
```

```
mjd =
       57384

polarmotion =
   1.0e-05 *
    0.0289    0.1233
```

Calculate Polar Motion and Error Using IERS Data

Calculate the polar motion and polar motion error for December 28, 2015 and January 10, 2016 using the `aeroiersdata.mat` file. Use the `mjuliandate` function to calculate the date as a modified Julian date.

```
mjd = mjuliandate([2015 12 28;2016 1 10])
[polarmotion,polarmotionErr] = polarMotion(mjd,'Source','aeroiersdata.mat')
```

```
mjd =
       57384
       57397

polarmotion =
   1.0e-05 *
    0.0289    0.1233
    0.0174    0.1304
```

Input Arguments

`utc` — Principal Universal Time (UT1) for UTC
M-by-1 array

Array of UTC dates, specified as an *M*-by-1 array, represented as modified Julian dates. Use the `mjuliandate` function to convert the UTC date to a modified Julian date.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Source','aeroiersdata.mat'

`Source` — Custom list of Earth orientation data
`aeroiersdata.mat` (default) | MAT-file

Custom list of Earth orientation data, specified in a MAT-file.

`action` — Out-of-range action
Warning (default) | action

Out-of-range action, specified as a string.

Action to take in case of out-of-range or predicted value dates, specified as a string:

- Warning — Displays warning and indicates that the dates were out-of-range or predicted values.
- Error — Displays error and indicates that the dates were out-of-range or predicted values.
- None — Does not display warning or error.

Data Types: `string`

Output Arguments

`polarmotion` — Movement of the rotation axis with respect to the crust of the Earth
M-by-2 array

Movement of the rotation axis with respect to the crust of the Earth, *M*-by-2 array, in radians.

`polarmotionError` — Error for movement of the rotation axis with respect to the crust of the Earth
M-by-2 array

Error for movement of the rotation axis with respect to the crust of the Earth, specified as an *M*-by-2 array, in radians.

See Also

`aeroReadIERSData` | `dcmeci2ecef` | `deltaCIP` | `deltaUT1` | `eci2aer` | `eci2lla` | `lla2eci` | `mjuliandate`

Topics

“Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation” on page 5-141

Introduced in R2018b

quat2angle

Convert quaternion to rotation angles

Syntax

```
[r1 r2 r3] = quat2angle(q)
[r1 r2 r3] = quat2angle(q, s)
```

Description

`[r1 r2 r3] = quat2angle(q)` calculates the set of rotation angles, `r1`, `r2`, `r3`, for a given quaternion, `q`. `q` is an `m`-by-4 matrix containing `m` quaternions. Each element of `q` must be a real number. `q` has its scalar number as the first column.

Rotation angles are output in radians.

`r1`

Returns an `m` array of first rotation angles.

`r2`

Returns an `m` array of second rotation angles.

`r3`

Returns an `m` array of third rotation angles.

`[r1 r2 r3] = quat2angle(q, s)` calculates the set of rotation angles, `r1`, `r2`, `r3`, for a given quaternion, `q`, and a specified rotation sequence, `s`.

The default rotation sequence is `'ZYX'`, where `r1` is *z*-axis rotation, `r2` is *y*-axis rotation, and `r3` is *x*-axis rotation.

Supported rotation sequences are `'ZYX'`, `'ZYZ'`, `'ZXY'`, `'ZXZ'`, `'YXZ'`, `'YXY'`, `'YZX'`, `'YZY'`, `'XYZ'`, `'XYX'`, `'XZY'`, and `'XZX'`.

Examples

Determine the rotation angles from q = [1 0 1 0].

```
[yaw, pitch, roll] = quat2angle([1 0 1 0])
yaw =
     0
pitch =
    1.5708
roll =
     0
```

Determine the rotation angles from multiple quaternions.

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
 [pitch, roll, yaw] = quat2angle(q, 'YXZ')
pitch =
    1.5708
    0.8073
roll =
         0
    0.7702
yaw =
         0
    0.5422
```

Assumptions and Limitations

The limitations for the `'ZYX'`, `'ZXY'`, `'YXZ'`, `'YZX'`, `'XYZ'`, and `'XZY'` implementations generate an `r2` angle that lies between ±90 degrees, and `r1` and `r3` angles that lie between ±180 degrees.

The limitations for the `'ZYZ'`, `'ZXZ'`, `'YXY'`, `'YZY'`, `'XYX'`, and `'XZX'` implementations generate an `r2` angle that lies between 0 and 180 degrees, and `r1` and `r3` angles that lie between ±180 degrees.

See Also

`angle2dcm` | `angle2quat` | `dcm2angle` | `dcm2quat` | `quat2dcm`

Introduced in R2007b

quat2dcm

Convert quaternion to direction cosine matrix

Syntax

```
n = quat2dcm(q)
```

Description

`n = quat2dcm(q)` calculates the direction cosine matrix, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns a 3-by-3-by-`m` matrix of direction cosine matrices. The direction cosine matrix performs the coordinate transformation of a vector in inertial axes to a vector in body axes. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

Examples

Determine the direction cosine matrix from `q = [1 0 1 0]`:

```
dcm = quat2dcm([1 0 1 0])


dcm =

         0         0   -1.0000
         0    1.0000         0
    1.0000         0         0
```

Determine the direction cosine matrices from multiple quaternions:

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
dcm = quat2dcm(q)


dcm(:,:,1) =
         0         0   -1.0000
         0    1.0000         0
    1.0000         0         0


dcm(:,:,2) =

    0.8519    0.3704   -0.3704
    0.0741    0.6148    0.7852
    0.5185   -0.6963    0.4963
```

See Also

`angle2dcm` | `angle2quat` | `dcm2angle` | `dcm2quat` | `quat2angle` | `quatrotate`

Introduced in R2006b

quat2rod

Convert quaternion to Euler-Rodrigues vector

Syntax

`rod=quat2rod(quat)`

Description

`rod=quat2rod(quat)` function calculates the Euler-Rodrigues vector, `rod`, for a given quaternion `quat`.

Examples

Determine Euler-Rodrigues Vector from Quaternion

Determine the Euler-Rodrigues vector from the quaternion.

```
q = [-0.7071 0 0.7071 0]
r = quat2rod( q )
```

```
q =

   -0.7071         0    0.7071         0
r =

         0   -1.0000         0
```

Input Arguments

`quat` — Quaternion
M-by-4 array

M-by-4 array of quaternions. `quat` has its scalar number as the first column.

Data Types: `double`

Output Arguments

`rod` — Euler-Rodrigues vector
M-by-3 matrix

M-by-3 matrix containing *M* Euler-Rodrigues vectors.

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x \; b_y \; b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

See Also

`angle2rod` | `dcm2rod` | `rod2angle` | `rod2dcm` | `rod2quat`

Introduced in R2017a

quatconj

Calculate conjugate of quaternion

Syntax

```
n = quatconj(q)
```

Description

`n = quatconj(q)` calculates the conjugate, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns an `m`-by-4 matrix of conjugates. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

The quaternion conjugate has the form of

$$q' = q_0 - \mathbf{i}q_1 - \mathbf{j}q_2 - \mathbf{k}q_3$$

Examples

Determine the conjugate of `q = [1 0 1 0]`:

```
conj = quatconj([1 0 1 0])


conj =

     1     0    -1     0
```

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, Wiley–Interscience, 2nd Edition.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for this function requires the Aerospace Blockset software.

See Also

`quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnorm` | `quatnormalize` | `quatrotate`

Introduced in R2006b

quatdivide

Divide quaternion by another quaternion

Syntax

```
n = quatdivide(q,r)
```

Description

`n = quatdivide(q,r)` calculates the result of quaternion division, `n`, for two given quaternions, `q` and `r`. Inputs `q` and `r` can each be either an `m`-by-4 matrix containing `m` quaternions, or a single 1-by-4 quaternion. `n` returns an `m`-by-4 matrix of quaternion quotients. Each element of `q` and `r` must be a real number. Additionally, `q` and `r` have their scalar number as the first column.

The quaternions have the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

and

$$r = r_0 + \mathbf{i}r_1 + \mathbf{j}r_2 + \mathbf{k}r_3$$

The resulting quaternion from the division has the form of

$$t = \frac{q}{r} = t_0 + \mathbf{i}t_1 + \mathbf{j}t_2 + \mathbf{k}t_3$$

where

$$t_0 = \frac{(r_0 q_0 + r_1 q_1 + r_2 q_2 + r_3 q_3)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

$$t_1 = \frac{(r_0 q_1 - r_1 q_0 - r_2 q_3 + r_3 q_2)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

$$t_2 = \frac{(r_0 q_2 + r_1 q_3 - r_2 q_0 - r_3 q_1)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

$$t_3 = \frac{(r_0 q_3 - r_1 q_2 + r_2 q_1 - r_3 q_0)}{r_0^2 + r_1^2 + r_2^2 + r_3^2}$$

Examples

Determine the division of two 1-by-4 quaternions:

```
q = [1 0 1 0];
r = [1 0.5 0.5 0.75];
d = quatdivide(q, r)


d =

    0.7273    0.1212    0.2424   -0.6061
```

Determine the division of a 2-by-4 quaternion by a 1-by-4 quaternion:

```
q = [1 0 1 0; 2 1 0.1 0.1];
r = [1 0.5 0.5 0.75];
d = quatdivide(q, r)


d =

    0.7273    0.1212    0.2424   -0.6061
    1.2727    0.0121   -0.7758   -0.4606
```

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, Wiley–Interscience, 2nd Edition.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for this function requires the Aerospace Blockset software.

See Also

`quatconj` | `quatinv` | `quatmod` | `quatmultiply` | `quatnorm` | `quatnormalize` | `quatrotate`

Introduced in R2006b

quatexp

Exponential of quaternion

Syntax

```
qe=quatexp(q)
```

Description

`qe=quatexp(q)` calculates the exponential, `qe`, for the specified quaternion, `q`.

Examples

Calculate the Exponential of Quaternion

Calculate the exponentials of quaternion matrix `[0 0 0.7854 0]`.

```
qe = quatexp([0 0 0.7854 0])
```

```
qe =
    0.7071         0    0.7071         0
```

Input Arguments

q — Quaternions
M-by-4 matrix

Quaternions for which to calculate exponentials, specified as an *M*-by-4 matrix containing *M* quaternions.

Data Types: `double`

Output Arguments

qe — Exponential of quaternion
M-by-4 matrix

Exponential of quaternion.

See Also

`quatconj` | `quatdivide` | `quatinterp` | `quatinv` | `quatlog` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatpower` | `quatrotate`

External Websites

http://web.mit.edu/2.998/www/QuaternionReport1.pdf

Introduced in R2016a

quatinterp

Quaternion interpolation between two quaternions

Syntax

```
qi=quatinterp(p,q,f,method)
```

Description

`qi=quatinterp(p,q,f,method)` calculates the quaternion interpolation between two normalized quaternions `p` and `q` by interval fraction `f`.

`p` and `q` are the two extremes between which the function calculates the quaternion.

Examples

Quaternion Interpolation Between Two Quaternions

Use interpolation to calculate quaternion between two quaternions `p=[1.0 0 1.0 0]` and `q=[-1.0 0 1.0 0]` using the `SLERP` method. This example uses the `quatnormalize` function to first-normalize the two quaternions to `pn` and `qn`.

```
pn = quatnormalize([1.0 0 1.0 0])
qn = quatnormalize([-1.0 0 1.0 0])
qi = quatinterp(pn,qn,0.5,'slerp')
```

```
pn =

    0.7071         0    0.7071         0

qn =

   -0.7071         0    0.7071         0

qi =
     0     0     1     0
```

Input Arguments

p — First-normalized quaternion
M-by-4 matrix

First normalized quaternion for which to calculate the interpolation, specified as an *M*-by-4 matrix containing *M* quaternions. This quaternion must be a normalized quaternion.

Data Types: `double`

q — Quaternions
M-by-4 matrix

Second normalized quaternion for which to calculate the interpolation, specified as an *M*-by-4 matrix containing *M* quaternions. This quaternion must be a normalized quaternion.

Data Types: `double`

`f` — Interval fraction
M-by-1 matrix

Interval fraction by which to calculate the quaternion interpolation, specified as an *M*-by-1 matrix containing *M* fractions (scalar). `f` varies between 0 and 1. It represents the intermediate rotation of the quaternion to be calculated.

$q_i=(q_p,q_n,q_f)$, where:

- If f equals `0`, q_i equals q_p.
- If f is between `0` and `1`, q_i equals `method`.
- If f equals `1`, q_i equals q_n.

Data Types: `double`

`method` — Quaternion interpolation method
`'slerp'` (default) | `'lerp'` | `'nlerp'`

Quaternion interpolation method to calculate the quaternion interpolation. These methods have different rotational velocities, depending on the interval fraction. For more information on interval fractions, see http://web.mit.edu/2.998/www/QuaternionReport1.pdf.

- `slerp`

  Quaternion slerp. Spherical linear quaternion interpolation method. This method is most accurate, but also most computation intense.

  $Slerp(p, q, h) = p(p^{*}q)^{h}$ with $h \in [0, 1]$.
- `lerp`

  Quaternion lerp. Linear quaternion interpolation method. This method is the quickest, but is also least accurate. The method does not always generate normalized output.

  $LERP(p, q, h) = p(1 - h) + qh$ with $h \in [0, 1]$.
- `nlerp`

  Normalized quaternion linear interpolation method.

  With $r = LERP(p, q, h)$, $NLERP(p, q, h) = \frac{r}{|r|}$.

Data Types: `char`

Output Arguments

`qi` — Interpolation of quaternion
M-by-4 matrix

Interpolation of quaternion.

See Also

`quatconj` | `quatdivide` | `quatexp` | `quatinv` | `quatlog` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatpower` | `quatrotate`

External Websites

http://web.mit.edu/2.998/www/QuaternionReport1.pdf

Introduced in R2016a

quatinv

Calculate inverse of quaternion

Syntax

```
n = quatinv(q)
```

Description

`n = quatinv(q)` calculates the inverse, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns an `m`-by-4 matrix of inverses. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

The quaternion inverse has the form of

$$q^{-1} = \frac{q_0 - \mathbf{i}q_1 - \mathbf{j}q_2 - \mathbf{k}q_3}{q_0^2 + q_1^2 + q_2^2 + q_3^2}$$

Examples

Determine the inverse of `q = [1 0 1 0]`:

```
qinv = quatinv([1 0 1 0])


qinv =

    0.5000         0   -0.5000         0
```

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, Wiley–Interscience, 2nd Edition.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for this function requires the Aerospace Blockset software.

See Also

`quatconj` | `quatdivide` | `quatmod` | `quatmultiply` | `quatnorm` | `quatnormalize` | `quatrotate`

Introduced in R2006b

quatlog

Natural logarithm of quaternion

Syntax

```
ql=quatlog(q)
```

Description

`ql=quatlog(q)` calculates the natural logarithm, `ql`, for a normalized quaternion, `q`.

This function uses the relationships.

For $q = [\cos(\theta), \sin(\theta)v]$, with $\log(q) = [0, \theta v]$.

Examples

Calculate the Natural Logarithm of Quaternion

Calculate the natural logarithm of quaternion matrix `q=[1.0 0 1.0 0]`.

```
qlog = quatlog(quatnormalize([1.0 0 1.0 0]))
```

```
qlog =

         0         0    0.7854         0
```

Input Arguments

q — Quaternions
M-by-4 matrix

Quaternions for which to calculate the natural logarithm, specified as an *M*-by-4 matrix containing *M* quaternions. This quaternion must be a normalized quaternion.

Data Types: `double`

Output Arguments

`ql` — Natural logarithm of quaternion
M-by-4 matrix

Natural logarithm of quaternion.

See Also

`quatconj` | `quatdivide` | `quatexp` | `quatinterp` | `quatinv` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatpower` | `quatrotate`

External Websites

http://web.mit.edu/2.998/www/QuaternionReport1.pdf

Introduced in R2016a

quatmod

Calculate modulus of quaternion

Syntax

```
n = quatmod(q)
```

Description

`n = quatmod(q)` calculates the modulus, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns a column vector of `m` moduli. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

The quaternion modulus has the form of

$$|q| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}$$

Examples

Determine the modulus of `q = [1 0 0 0]`:

```
mod = quatmod([1 0 0 0])


mod =

     1
```

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, Wiley–Interscience, 2nd Edition.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for this function requires the Aerospace Blockset software.

See Also

`quatconj` | `quatdivide` | `quatinv` | `quatmultiply` | `quatnorm` | `quatnormalize` | `quatrotate`

Introduced in R2006b

quatmultiply

Calculate product of two quaternions

Syntax

```
quatprod = quatmultiply(q,r)
```

Description

`quatprod = quatmultiply(q,r)` calculates the quaternion product, `quatprod`, for two quaternions, `q` and `r`.

Note Quaternion multiplication is not commutative.

Examples

Determine the Product of Two Quaternions

This example shows how to determine the product of two 1-by-4 quaternions.

```
q = [1 0 1 0];
r = [1 0.5 0.5 0.75];
mult = quatmultiply(q, r)
```

```
mult = 1×4

    0.5000    1.2500    1.5000    0.2500
```

Determine Product of a Quaternion with Itself

This example shows how to determine the product of a 1-by-4 quaternion with itself.

```
q = [1 0 1 0];
mult = quatmultiply(q)
```

```
mult = 1×4

     0     0     2     0
```

Determine the Product of Two Different Quaternions

This example shows how to determine the product of 1-by-4 with two 1-by-4 quaternions.

```
q = [1 0 1 0];
r = [1 0.5 0.5 0.75; 2 1 0.1 0.1];
mult = quatmultiply(q, r)
```

```
mult = 2×4

    0.5000    1.2500    1.5000    0.2500
    1.9000    1.1000    2.1000   -0.9000
```

Input Arguments

q — First quaternion
m-by-4 matrix | 1-by-4 quaternion | real

First quaternion or set of quaternions, specified as an *m*-by-4 matrix or 1-by-4 quaternion. Each element must be real.

`q` must have its scalar number as the first column.

Data Types: `double` | `single`

r — Second quaternion
m-by-4 matrix | 1-by-4 quaternion | real

Second quaternionor set of quaternions, specified as an *m*-by-4 matrix or 1-by-4 quaternion. Each element must be real.

`r` must have its scalar number as the first column.

Data Types: `double` | `single`

Output Arguments

`quatprod` — Output quaternion product
m-by-4 matrix

Output quaternion product, returned as a *m*-by-4 matrix.

More About

q and r

Input quaternions `q` and `r` have the form:

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

and

$$r = r_0 + \mathbf{i}r_1 + \mathbf{j}r_2 + \mathbf{k}r_3$$

quatprod

Output quaternion product `quatprod` has the form of

$$n = q \times r = n_0 + \mathbf{i}n_1 + \mathbf{j}n_2 + \mathbf{k}n_3$$

where

$$\begin{aligned}
n_0 &= (r_0q_0 - r_1q_1 - r_2q_2 - r_3q_3)\\
n_1 &= (r_0q_1 + r_1q_0 - r_2q_3 + r_3q_2)\\
n_2 &= (r_0q_2 + r_1q_3 + r_2q_0 - r_3q_1)\\
n_3 &= (r_0q_3 - r_1q_2 + r_2q_1 + r_3q_0)
\end{aligned}$$

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, 2nd Edition. Hoboken, NJ: John Wiley & Sons, 2003.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for this function requires the Aerospace Blockset software.

See Also

`quatconj` | `quatdivide` | `quatinv` | `quatmod` | `quatnorm` | `quatnormalize` | `quatrotate`

Introduced in R2006b

quatnorm

Calculate norm of quaternion

Syntax

```
n = quatnorm(q)
```

Description

`n = quatnorm(q)` calculates the norm, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns a column vector of `m` norms. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

The quaternion norm has the form of

$$norm(q) = q_0^2 + q_1^2 + q_2^2 + q_3^2$$

Examples

Determine the norm of `q = [1 0 0 0]`:

```
norm=quatnorm([.5 -.5 .5 0])

norm =

    0.7500
```

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, Wiley–Interscience, 2nd Edition.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for this function requires the Aerospace Blockset software.

See Also

`quatconj` | `quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatrotate`

Introduced in R2006b

quatnormalize

Normalize quaternion

Syntax

```
n = quatnormalize(q)
```

Description

`n = quatnormalize(q)` calculates the normalized quaternion, `n`, for a given quaternion, `q`. Input `q` is an `m`-by-4 matrix containing `m` quaternions. `n` returns an `m`-by-4 matrix of normalized quaternions. Each element of `q` must be a real number. Additionally, `q` has its scalar number as the first column.

The quaternion has the form of

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

The normalized quaternion has the form of

$$normal(q) = \frac{q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3}{\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2}}$$

Examples

Normalize `q = [1 0 1 0]`:

```
normal = quatnormalize([1 0 1 0])


normal =

    0.7071         0    0.7071         0
```

References

[1] Stevens, Brian L., Frank L. Lewis, *Aircraft Control and Simulation*, Wiley–Interscience, 2nd Edition.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation for this function requires the Aerospace Blockset software.

See Also

`quatconj` | `quatdivide` | `quatinv` | `quatmod` | `quatmultiply` | `quatnorm` | `quatrotate`

Introduced in R2006b

quatpower

Power of quaternion

Syntax

```
qp=quatpower(q,pow)
```

Description

`qp=quatpower(q,pow)` calculates `q` to the power of `pow` for a normalized quaternion, `q`.

$q^t = \exp(t \cdot \log(q))$, with $t \in R$.

Examples

Calculate the Power of Quaternion

Calculate the power of 2 of quaternion `q = [1.0 0 1.0 0]`.

```
qp = quatpower(quatnormalize([1.0 0 1.0 0]),2)
```

```
qp =

   -0.0000         0    1.0000         0
```

Input Arguments

q — Quaternions
M-by-4 matrix

Quaternions for which to calculate exponentials, specified as an *M*-by-4 matrix containing *M* quaternions.

Data Types: `double`

pow — Power
M-by-1 vector

Power to which to calculate quaternion power, specified as an *M*-by-1 vector containing *M* power scalars.

Data Types: `double`

Output Arguments

qp — Power of quaternion
M-by-4 matrix

Power of quaternion.

See Also

`quatconj` | `quatdivide` | `quatexp` | `quatinterp` | `quatinv` | `quatlog` | `quatmod` | `quatmultiply` | `quatnormalize` | `quatrotate`

External Websites

http://web.mit.edu/2.998/www/QuaternionReport1.pdf

Introduced in R2016a

quatrotate

Rotate vector by quaternion

Syntax

```
n = quatrotate(q,r)
```

Description

`n = quatrotate(q,r)` calculates the rotated vector, `n`, for a quaternion, `q`, and a vector, `r`. If quaternions are not yet normalized, the function normalizes them.

Examples

Rotate a 1-by-3 Vector

This example shows how to rotate a 1-by-3 vector by a 1-by-4 quaternion.

```
q = [1 0 1 0];
r = [1 1 1];
n = quatrotate(q, r)
```

```
n = 1×3

   -1.0000    1.0000    1.0000
```

Rotate Two 1-by-3 Vectors by a 1-by-4 Quaternion

This example shows how to rotate two 1-by-3 vectors by a 1-by-4 quaternion.

```
q = [1 0 1 0];
r = [1 1 1; 2 3 4];
n = quatrotate(q, r)

n = 2×3

   -1.0000    1.0000    1.0000
   -4.0000    3.0000    2.0000
```

Rotate a 1-by-3 Vector by Two 1-by-4 Quaternions

This example shows how to rotate a 1-by-3 vector by two 1-by-4 quaternions.

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
r = [1 1 1];
n = quatrotate(q, r)

n = 2×3

   -1.0000    1.0000    1.0000
    0.8519    1.4741    0.3185
```

Rotate Multiple Vectors by Multiple Quaternions

This example shows how to rotate multiple vectors by multiple quaternions.

```
q = [1 0 1 0; 1 0.5 0.3 0.1];
r = [1 1 1; 2 3 4];
n = quatrotate(q, r)

n = 2×3

   -1.0000    1.0000    1.0000
    1.3333    5.1333    0.9333
```

Input Arguments

q — Quaternion
m-by-4 matrix | 1-by-4 array

Quaternion or set of quaternions, specified as an *m*-by-4 matrix containing *m* quaternions, or a single 1-by-4 quaternion. Each element must be real.

`q` must have its scalar number as the first column.

Data Types: `double` | `single`

r — Vector
m-by-3 matrix | 1-by-3 array

Vector or set of vectors to be rotated, specified as an *m*-by-3 matrix, containing *m* vectors, or a single 1-by-3 array. Each element must be real.

Data Types: `double` | `single`

Output Arguments

n — Rotated vector
m-by-3 matrix

Rotated vector, returned as an *m*-by-3 matrix.

More About

q

Quaternion `q` has the form:

$$q = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

r

Vector `r` has the form:

$$v = \mathbf{i}v_1 + \mathbf{j}v_2 + \mathbf{k}v_3$$

n

Rotated vector `n` has the form:

$$v' = \begin{bmatrix} v_1' \\ v_2' \\ v_3' \end{bmatrix} = \begin{bmatrix} (1 - 2q_2^2 - 2q_3^2) & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & (1 - 2q_1^2 - 2q_3^2) & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & (1 - 2q_1^2 - 2q_2^2) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

The direction cosine matrix for this equation expects a normalized quaternion.

References

[1] Stevens, Brian L., Frank L. Lewis. *Aircraft Control and Simulation*, 2nd Edition. Hoboken, NJ: John Wiley & Sons, 2003.

[2] Diebel, James. "Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors." Stanford University, Stanford, California, 2006.

See Also

`quatconj` | `quatinv` | `quatmod` | `quatmultiply` | `quatnorm` | `quatnormalize`

Introduced in R2006b

read (Aero.Geometry)

Read geometry data using current reader

Syntax

```
read(h, source)
```

Description

`read(h, source)` reads the geometry data of the geometry object `h`. `source` can be:

- `'Auto'`

  Selects default reader.
- `'Variable'`

  Selects MATLAB variable of type structure structures that contains the fields`name`, `faces`, `vertices`, and `cdata` that define the geometry in the Handle Graphics patches.
- `'MatFile'`

  Selects MAT-file reader.
- `'Ac3dFile'`

  Selects Ac3d file reader.
- `'Custom'`

  Selects a custom reader.

Examples

Read geometry data from Ac3d file, `pa24-250_orange.ac`.

```
g = Aero.Geometry;
g.Source = 'Ac3d';
g.read('pa24-250_orange.ac');
```

Introduced in R2007a

removeBody

Class: `Aero.Animation`
Package: `Aero`

Remove one body from animation

Syntax

```
h = removeBody(h,idx)
h = h.removeBody(idx)
```

Description

`h = removeBody(h,idx)` and `h = h.removeBody(idx)` remove the body specified by the index `idx` from the animation object `h`.

Input Arguments

`h`	Animation object.
`idx`	Body specified with this index.

Examples

Remove the body identified by the index, 1.

```
h = Aero.Animation;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
h = removeBody(h,1)
```

removeNode (Aero.VirtualRealityAnimation)

Remove node from virtual reality animation object

Syntax

`removeNode(h,node)`
`h.removeNode(node)`

Description

`removeNode(h,node)` and `h.removeNode(node)` remove the node specified by `node` from the virtual reality animation object `h`. `node` can be either the node name or the node index. This function can remove only one node at a time.

Note You can use only this function to remove a node added by `addNode`. If you need to remove a node from a previously defined `.wrl` file, use a VRML editor.

Examples

Remove the node, Lynx1.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
h.addNode('Lynx1',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
h.removeNode('Lynx1');
```

See Also

`addNode`

Introduced in R2007b

removeViewpoint (Aero.VirtualRealityAnimation)

Remove viewpoint node from virtual reality animation

Syntax

```
removeViewpoint(h,viewpoint)
h.removeViewpoint(viewpoint)
```

Description

`removeViewpoint(h,viewpoint)` and `h.removeViewpoint(viewpoint)` remove the viewpoint specified by `viewpoint` from the virtual reality animation object `h`. `viewpoint` can be either the viewpoint name or the viewpoint index. This function can remove only one viewpoint at a time.

Note You can use this function to remove a viewpoint added by `addViewpoint`. If you need to remove a viewpoint from a previously defined `.wrl` file, use a VRML editor.

Examples

Remove the node, Lynx1.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
h.addViewpoint(h.Nodes{2}.VRNode,'children','chaseView','View From Helicopter');
h.removeViewpoint('chaseView');
```

See Also

`addViewpoint`

Introduced in R2007b

rod2angle

Convert Euler-Rodrigues vector to rotation angles

Syntax

```
[R1 R2 R3]=rod2angle(rod)
[R1 R2 R3]=rod2angle(rod,S)
```

Description

`[R1 R2 R3]=rod2angle(rod)` function calculates the set of rotation angles, `R1`, `R2`, and `R3`, for a given Euler-Rodrigues (also known as Rodrigues) vector, `rod`.

`[R1 R2 R3]=rod2angle(rod,S)` function calculates the set of rotation angles for a given Rodrigues vector and a specified rotation sequence, `S`.

Examples

Determine Rotation Angles from One Vector

Determine rotation angles from vector, `[.1 .2 -.1]`.

```
r = [.1 .2 -.1];
[yaw, pitch, roll] = rod2angle(r)
```

```
yaw =

   -0.1651

pitch =

    0.4074

roll =
    0.1651
```

Input Arguments

rod — Rodrigues vector
M-by-3 matrix

M-by-3 matrix containing *M* Rodrigues vector.

Data Types: `double`

S — Rotation sequence
ZYX (default) | ZYZ | ZXY | ZXZ | YXZ | YXY | YZX | YZY | XYZ | XYX | XZY | XZX

Rotation angles, in radians, from which to determine Rodrigues vector. For the default rotation sequence, ZYX, the rotation angle order is:

- R1 — *z*-axis rotation
- R2 — *y*-axis rotation
- R3 — *x*-axis rotation

Data Types: `char` | `string`

Output Arguments

R1 — First rotation angles
M-by-1 array

M-by-1 array of first rotation angles, in radians.

R2 — Second rotation angles
M-by-1 array

M-by-1 array of second rotation angles, in radians.

R3 — Third rotation angles
M-by-1 array

M-by-1 array of third rotation angles, in radians.

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x\ b_y\ b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

See Also

`angle2rod` | `dcm2rod` | `quat2rod` | `rod2dcm` | `rod2quat`

Introduced in R2017a

rod2dcm

Convert Euler-Rodrigues vector to direction cosine matrix

Syntax

`dcm=rod2dcm(R)`

Description

`dcm=rod2dcm(R)` function calculates the direction cosine matrix, for a given Euler-Rodrigues (also known as Rodrigues) vector, `R`.

Examples

Determine Direction Cosine Matrix from Euler-Rodrigues Vector

Determine the direction cosine matrix from the Euler-Rodrigues vector.

```
r = [.1 .2 -.1];
DCM = rod2dcm(r)
```

```
DCM =

    0.9057   -0.1509   -0.3962
    0.2264    0.9623    0.1509
    0.3585   -0.2264    0.9057
```

Input Arguments

R — Rodrigues vector
M-by-3 matrix

M-by-3 matrix containing *M* Rodrigues vectors.

Data Types: `double`

Output Arguments

`dcm` — Direction cosine matrix
3-by-3-by-*M* matrix

3-by-3-by-*M* containing *M* direction cosine matrices.

Algorithms

An Euler-Rodrigues vector $\overrightarrow{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\overrightarrow{b} = [b_x \; b_y \; b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\overrightarrow{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

See Also

`angle2rod` | `dcm2rod` | `quat2rod` | `rod2angle` | `rod2quat`

Introduced in R2017a

rod2quat

Convert Euler-Rodrigues vector to quaternion

Syntax

`quat=rod2quat(R)`

Description

`quat=rod2quat(R)` function calculates the quaternion, `quat`, for a given Euler-Rodrigues (also known as Rodrigues) vector, `R`.

Examples

Determine Quaternion from Rodrigues Vector

Determine the quaternion from Rodrigues vector.

```
r = [.1 .2 -.1];
q = rod2quat(r)
```

```
q =

    0.9713    0.0971    0.1943   -0.0971
```

Input Arguments

R — Rodrigues vector
M-by-1 matrix

M-by-1 array of Rodrigues vectors.

Data Types: `double`

Output Arguments

quat — Rodrigues vector
M-by-4 matrix

M-by-4 matrix of *M* quaternions. `quat` has its scalar number as the first column.

Algorithms

An Euler-Rodrigues vector $\vec{b}$ represents a rotation by integrating a direction cosine of a rotation axis with the tangent of half the rotation angle as follows:

$$\vec{b} = [b_x\ b_y\ b_z]$$

where:

$$b_x = \tan\left(\frac{1}{2}\theta\right)s_x,$$

$$b_y = \tan\left(\frac{1}{2}\theta\right)s_y,$$

$$b_z = \tan\left(\frac{1}{2}\theta\right)s_z$$

are the Rodrigues parameters. Vector $\vec{s}$ represents a unit vector around which the rotation is performed. Due to the tangent, the rotation vector is indeterminate when the rotation angle equals ±pi radians or ±180 deg. Values can be negative or positive.

References

[1] Dai, J.S. "Euler-Rodrigues formula variations, quaternion conjugation and intrinsic connections." *Mechanism and Machine Theory*, 92, 144-152. Elsevier, 2015.

See Also

`angle2rod` | `dcm2rod` | `quat2rod` | `rod2angle` | `rod2dcm`

Introduced in R2017a

RPMIndicator Properties

Control revolutions per minute (RPM) indicator appearance and behavior

Description

RPM indicators are components that represent an RPM indicator. Properties control the appearance and behavior of an RMP indicator. Use dot notation to refer to a particular object and property:

```
f = uifigure;
rpm = uiaerorpm(f);
rpm.Value = 100;
```

The RPM indicator displays measurements for engine revolutions per minute in percentage of RPM.

The range of values for RPM goes from 0 to 110%. Minor ticks represent increments of 5% RPM and major ticks represent increments of 10% RPM.

Properties

RMP Indicator

`Limits` — Minimum and maximum indicator scale values

[0 110] (default) | two-element finite, real, and scalar numeric array

Minimum and maximum indicator scale values, specified as a two-element numeric array. The first value in the array must be less than the second value. This value is read-only.

If you change `Limits` such that the `Value` property is less than the new lower limit, or more than the new upper limit, then the indicator needle points to a location off the scale.

For example, suppose `Limits` is `[0 100]` and the `Value` property is 20. If the `Limits` changes to `[50 100]`, then the needle points to a location off the scale, slightly less than 50.

`RPM` — Location of RPM indicator needle

`0` (default) | finite, real, and scalar numeric

Location of the RPM indicator needle, specified a finite and scalar numeric rev/min.

- Changing the value changes the location of the to align with the corresponding value on the indicator.

Example: `60`

Dependencies

Specifying this value changes the value of `Value`.

Data Types: `double`

`ScaleColors` — Scale colors

[ ] (default) | n-by-3 array of RGB triplets | cell array

Scale colors, specified one of the following arrays:

- An n-by-3 array of RGB triplets
- A cell array containing RGB triplets, any of the color options listed in the table below, or a combination of both.

An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range `[0,1]`; for example, `[0.4 0.6 0.7]`. Alternatively, you can specify some common colors by name. This table lists the long and short color name options and the equivalent RGB triplet values.

Option	Description	Equivalent RGB Triplet
`'red'` or `'r'`	Red	`[1 0 0]`
`'green'` or `'g'`	Green	`[0 1 0]`
`'blue'` or `'b'`	Blue	`[0 0 1]`
`'yellow'` or `'y'`	Yellow	`[1 1 0]`
`'magenta'` or `'m'`	Magenta	`[1 0 1]`
`'cyan'` or `'c'`	Cyan	`[0 1 1]`
`'white'` or `'w'`	White	`[1 1 1]`

Option	Description	Equivalent RGB Triplet
`'black'` or `'k'`	Black	`[0 0 0]`

Each color of the `ScaleColors` array corresponds to a colored section of the gauge. Set the `ScaleColorLimits` property to map the colors to specific sections of the gauge.

If you do not set the `ScaleColorLimits` property, MATLAB distributes the colors equally over the range of the gauge.

`ScaleColorLimits` — Scale color limits

[ ] (default) | n-by-2 array

Scale color limits, specified as an n-by-2 array of numeric values. For every row in the array, the first element must be less than the second element.

When applying colors to the gauge, MATLAB applies the colors starting with the first color in the `ScaleColors` array. Therefore, if two rows in `ScaleColorLimits` array overlap, then the color applied later takes precedence.

The gauge does not display any portion of the `ScaleColorLimits` that falls outside of the `Limits` property.

If the `ScaleColors` and `ScaleColorLimits` property values are different sizes, then the gauge shows only the colors that have matching limits. For example, if the `ScaleColors` array has three colors, but the `ScaleColorLimits` has only two rows, then the gauge displays the first two color/limit pairs only.

`Value` — Location of RPM indicator needle

`0` (default) | finite, real, and scalar numeric

Location of the RPM indicator needle, specified a finite and scalar numeric rev/min.

- Changing the value changes the location of the to align with the corresponding value on the indicator.

Example: `60`

Dependencies

Specifying this value changes the value of `RPM`.

Data Types: `double`

Interactivity

`Visible` — Visibility of RPM indicator
`'on'` (default) | `'off'`

Visibility of the RPM indicator, specified as `'on'` or `'off'`. The `Visible` property determines whether the RPM indicator is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire RPM indicator is hidden, but you can still specify and access its properties.

`Enable` — Operational state of RPM indicator
`'on'` (default) | `'off'`

Operational state of RPM indicator, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the RPM indicator indicates that the RPM indicator is operational.
- If you set this property to `'off'`, then the appearance of the RPM indicator appears dimmed, indicating that the RPM indicator is not operational.

Position

`Position` — Location and size of RPM indicator
`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the RPM indicator relative to the parent container, specified as the vector, `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the RPM indicator
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the RPM indicator
`width`	Distance between the right and left outer edges of the RPM indicator
`height`	Distance between the top and bottom outer edges of the RPM indicator

All measurements are in pixel units.

The Position values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: [200 120 120 120]

InnerPosition — Inner location and size of RPM indicator
[100 100 120 120] (default) | [left bottom width height]

Inner location and size of the RPM indicator, specified as [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

OuterPosition — Outer location and size of RPM indicator
[100 100 120 120]] (default) | [left bottom width height]

This property is read-only.

Outer location and size of the RPM indicator returned as [left bottom width height]. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the Position property.

Layout — Layout options
empty LayoutOptions array (default) | GridLayoutOptions object

Layout options, specified as a GridLayoutOptions object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the Row and Column properties on the GridLayoutOptions object.

For example, this code places an RPM indicator in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
gauge = uiaerorpm(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the RPM indicator span multiple rows or columns, specify the Row or Column property as a two-element vector. For example, this RPM indicator spans columns 2 through 3:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

`BusyAction` — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

`BeingDeleted` — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`HandleVisibility` — Visibility of object handle
`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

`Parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Identifiers

Type — Type of graphics object
`'uiaerorpm'`

This property is read-only.

Type of graphics object, returned as `'uiaerorpm'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also

`uiaerorpm`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

rrdelta

Compute relative pressure ratio

Syntax

```
d = rrdelta(p0, mach, g)
```

Description

`d = rrdelta(p0, mach, g)` computes `m` pressure relative ratios, `d`, from `m` static pressures, `p0`, `m` Mach numbers, `mach`, and `m` specific heat ratios, `g`. `p0` must be in pascals.

Examples

Determine the relative pressure ratio for three pressures:

```
delta = rrdelta([101325 22632.0672 4328.1393], 0.5, 1.4)


delta =

    1.1862    0.2650    0.0507
```

Determine the relative pressure ratio for three pressures and three different heat ratios:

```
delta = rrdelta([101325 22632.0672 4328.1393], 0.5, [1.4 1.35 1.4])


delta =

    1.1862    0.2635    0.0507
```

Determine the relative pressure ratio for three pressures at three different conditions:

```
delta = rrdelta([101325 22632.0672 4328.1393], [0.5 1 2], [1.4 1.35 1.4])
delta =

    1.1862    0.4161    0.3342
```

Assumptions and Limitations

For cases in which total pressure ratio is desired (Mach number is nonzero), the total pressures are calculated assuming perfect gas (with constant molecular weight, constant pressure specific heat, and constant specific heat ratio) and dry air.

References

Aeronautical Vestpocket Handbook, United Technologies Pratt & Whitney, August, 1986

See Also

`rrsigma` | `rrtheta`

Introduced in R2006b

rrsigma

Compute relative density ratio

Syntax

```
s = rrsigma(rho, mach, g)
```

Description

`s = rrsigma(rho, mach, g)` computes `m` density relative ratios, `s`, from `m` static densities, `rho`, `m` Mach numbers, `mach`, and `m` specific heat ratios, `g`. `rho` must be in kilograms per meter cubed.

Examples

Determine the relative density ratio for three densities:

```
sigma = rrsigma([1.225 0.3639 0.0953], 0.5, 1.4)


sigma =

    1.1297    0.3356    0.0879
```

Determine the relative density ratio for three densities and three different heat ratios:

```
sigma = rrsigma([1.225 0.3639 0.0953], 0.5, [1.4 1.35 1.4])


sigma =

    1.1297    0.3357    0.0879
```

Determine the relative density ratio for three densities at three different conditions:

```
sigma = rrsigma([1.225 0.3639 0.0953], [0.5 1 2], [1.4 1.35 1.4])
sigma =

    1.1297    0.4709    0.3382
```

Assumptions and Limitations

For cases in which total density ratio is desired (Mach number is nonzero), the total density is calculated assuming perfect gas (with constant molecular weight, constant pressure specific heat, and constant specific heat ratio) and dry air.

References

Aeronautical Vestpocket Handbook, United Technologies Pratt & Whitney, August, 1986

See Also

`rrdelta` | `rrtheta`

Introduced in R2006b

rrtheta

Compute relative temperature ratio

Syntax

```
th = rrtheta(t0, mach, g)
```

Description

`th = rrtheta(t0, mach, g)` computes `m` temperature relative ratios, `th`, from `m` static temperatures, `t0`, `m` Mach numbers, `mach`, and `m` specific heat ratios, `g`. `t0` must be in kelvin.

Examples

Determine the relative temperature ratio for three temperatures:

```
th = rrtheta([273.15 310.9278 373.15], 0.5, 1.4)


th =

    0.9953    1.1330    1.3597
```

Determine the relative temperature ratio for three temperatures and three different heat ratios:

```
th = rrtheta([273.15 310.9278 373.15], 0.5, [1.4 1.35 1.4])


th =

    0.9953    1.1263    1.3597
```

Determine the relative temperature ratio for three temperatures at three different conditions:

```
th = rrtheta([273.15 310.9278 373.15], [0.5 1 2], [1.4 1.35 1.4])

th =

    0.9953    1.2679    2.3310
```

Assumptions and Limitations

For cases in which total temperature ratio is desired (Mach number is nonzero), the total temperature is calculated assuming perfect gas (with constant molecular weight, constant pressure specific heat, and constant specific heat ratio) and dry air.

References

Aeronautical Vestpocket Handbook, United Technologies Pratt & Whitney, August, 1986

See Also

`rrdelta` | `rrsigma`

Introduced in R2006b

saveas (Aero.VirtualRealityAnimation)

Save virtual reality world associated with virtual reality animation object

Syntax

```
saveas(h, filename)
h.saveas(filename)
saveas(h, filename, '-nothumbnail')
h.saveas(filename, '-nothumbnail')
```

Description

`saveas(h, filename)` and `h.saveas(filename)` save the world associated with the virtual reality animation object, `h`, into the `.wrl` file name specified in the `filename` variable. After saving, this function reinitializes the virtual reality animation object from the saved world.

`saveas(h, filename, '-nothumbnail')` and `h.saveas(filename, '-nothumbnail')` suppress creating a thumbnail image used for virtual world preview.

Examples

Save the world associated with `h`.

```
h = Aero.VirtualRealityAnimation;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,asttkoff.wrl'];
h.initialize();
h.saveas([tempdir,'my_asttkoff.wrl']);
```

Introduced in R2007b

SetTimer (Aero.FlightGearAnimation)

Set name of timer for animation of FlightGear flight simulator

Syntax

```
SetTimer(h)
h.SetTimer
SetTimer(h, MyFGTimer)
h.SetTimer('MyFGTimer')
```

Description

`SetTimer(h)` and `h.SetTimer` set the name of the MATLAB timer for the animation of the FlightGear flight simulator. `SetTimer(h, MyFGTimer)` and `h.SetTimer('MyFGTimer')` set the name of the MATLAB timer for the animation of the FlightGear flight simulator and assign a custom name to the timer.

You can use this function to customize your FlightGear animation object. This customization allows you to simultaneously run multiple FlightGear objects if you want to use

- Multiple FlightGear sessions
- Different ports to connect to those sessions

Examples

Set the MATLAB timer for animation of the FlightGear animation object, `h`:

```
h = Aero.FlightGearAnimation
h.SetTimer
```

Set the MATLAB timer used for animation of the FlightGear animation object, `h`, and assign a custom name, *`MyFGTimer`*, to the timer:

```
h = Aero.FlightGearAnimation
h.SetTimer('MyFGTimer')
```

See Also

`ClearTimer`

Introduced in R2008b

show

Class: `Aero.Animation`
Package: `Aero`

Show animation object figure

Syntax

```
show(h)
h.show
```

Description

`show(h)` and `h.show` create the figure graphics object for the animation object `h`. Use the `hide` function to close the figure.

Input Arguments

`h`	Animation object.

Examples

Show the animation object, `h`.

```
h = Aero.Animation;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
h.show;
```

tdbjuliandate

Convert from Barycentric Dynamical Time Estimate to Julian date

Syntax

```
jdtdb = tdbjuliandate(terrestrial_time)
[jdtdb,tttdb] = tdbjuliandate(terrestrial_time)
```

Description

`jdtdb = tdbjuliandate(terrestrial_time)` returns an estimate of the Julian date for Barycentric Dynamical Time (TDB). These estimations are valid for the years 1980 to 2050.

`[jdtdb,tttdb] = tdbjuliandate(terrestrial_time)` additionally returns an array of Julian dates for the Barycentric Dynamical Time (TDB) based on the Terrestrial Time (TT).

Examples

Estimate Julian Date for Barycentric Dynamical Time

Estimate the Julian date for the Barycentric Dynamical Time for the Terrestrial Time 2014/10/15 16:22:31.

```
jdtdb = tdbjuliandate([2014,10,15,16,22,31])
```

```
jdtdb =

   2.4569e+06
```

Estimate Julian Dates for the Barycentric Dynamical Time and TT-TDB

Estimate the Julian dates for the Barycentric Dynamical Time and TT-TDB in seconds for the terrestrial time 2014/10/15 16:22:31 and 2010/7/22 1:57:17.

```
[jdtdb,tttdb] = tdbjuliandate([2014,10,15,16,22,31;2010,7,22,1,57,17])
```

```
JDTDB =

   1.0e+06 *

    2.4569
    2.4554


TTTTDB =

    0.0016
    0.0005
```

Input Arguments

`terrestrial_time` — Terrestrial Time
1-by-6 array | *M*-by-6 array

Terrestrial Time (TT) in year, month, day, hour, minutes, and seconds for which the function calculates the Julian date for Barycentric Dynamical Time. *M* is the number of Julian dates, one for each TT date. Specify values for year, month, day, hour, and minutes as whole numbers.

Output Arguments

`jdtdb` — Julian date
M-by-1 array

Julian date for the Barycentric Dynamical Time, returned as an *M*-by-1 array. *M* is the number of rows, one for each Terrestrial Time input.

`tttdb` — Difference in seconds
M-by-1 array

Difference in seconds between Terrestrial Time and Barycentric Dynamical Time (TT-TDB), returned as an *M*-by-1 array. *M* is the number of rows, one for each Terrestrial Time input.

Limitations

Fundamentals of Astrodynamics and Applications[1] indicates an accuracy of 50 microseconds, which this function cannot achieve due to numerical issues with the values involved.

References

[1] Vallado, D. A., *Fundamentals of Astrodynamics and Applications*, New York: McGraw-Hill, 1997.

See Also

`dcmeci2ecef` | `ecef2lla` | `geoc2geod` | `geod2geoc` | `lla2ecef` | `lla2eci`

Introduced in R2015a

TurnCoordinator Properties

Control turn coordinator appearance and behavior

Description

Turn coordinators are components that represent a turn coordinator. Properties control the appearance and behavior of a turn coordinator. Use dot notation to refer to a particular object and property:

```
f = uifigure;
turn = uiaeroturn(f);
turn.Turn = 100;
```

The turn coordinator displays measurements on a turn coordinator and inclinometer. These measurements help determine if the turn is coordinated, slipped, or skidded. The turn is a coordinated turn that combines the rolling and yawing of a turn. The turn indicator signal turns the airplane in the gauge, in degrees. The inclinometer turns the ball in the gauge, in degrees. Together, these signals show the slip and skid of an airplane as it turns. Both values cannot exceed +/–15 degrees. If values exceed 15 degrees, the gauge stays fixed at the minimum or maximum value.

Properties

Turn Coordinator

`Slip` — Slip
`0` (default) | finite, real, and scalar numeric

Slip value, specified as any finite and scalar numeric. The slip value controls the direction of the inclinometer ball. A negative value moves the ball to the right, a positive value moves the ball to the left, in degrees. This value cannot exceed +/–15 degrees. If it exceed 15 degrees, the gauge stays fixed at the minimum or maximum value.

Example: 10

Dependencies

Specifying this value changes the second element of the `Value` vector. Conversely, changing the second element of the `Value` vector changes the `Slip` value.

Data Types: `double`

`Turn` — Turn

`0` (default) | finite, real, and scalar numeric

Turn value, specified as any finite and scalar numeric. The turn value determines the aircraft heading rate of change. This value cannot exceed +/–15 degrees. If it exceed 15 degrees, the gauge stays fixed at the minimum or maximum value.

Example: 10

Dependencies

Specifying this value changes the first element of the `Value` vector. Conversely, changing the first element of the `Value` vector changes the `Turn` value.

Data Types: `double`

`Value` — Turn and slip

`[0 0]` (default) | two-element vector of finite, real, and scalar numerics

Turn and slip values, specified as a vector (`[Turn Slip]`).

- The turn value determines the aircraft heading rate of change.
- The slip value controls the direction of the inclinometer ball. A negative value moves the ball to the right, a positive value moves the ball to the left.

Example: [100 -200]

Dependencies

- Specifying the `Turn` value changes the first element of the `Value` vector. Conversely, changing the first element of the `Value` vector changes the `Turn` value.
- Specifying the `Slip` value changes the second element of the `Value` vector. Conversely, changing the second element of the `Value` vector changes the `Slip` value.

Data Types: `double`

Interactivity

`Visible` — Visibility of turn coordinator
`'on'` (default) | `'off'`

Visibility of the turn coordinator, specified as `'on'` or `'off'`. The `Visible` property determines whether the turn coordinator, is displayed on the screen. If the `Visible` property is set to `'off'`, then the entire turn coordinator is hidden, but you can still specify and access its properties.

`Enable` — Operational state of turn coordinator
`'on'` (default) | `'off'`

Operational state of turn coordinator, specified as `'on'` or `'off'`.

- If you set this property to `'on'`, then the appearance of the turn coordinator indicates that the turn coordinator is operational.
- If you set this property to `'off'`, then the appearance of the turn coordinator appears dimmed, indicating that the turn coordinator is not operational.

Position

`Position` — Location and size of turn coordinator
`[100 100 120 120]` (default) | `[left bottom width height]`

Location and size of the turn coordinator relative to the parent container, specified as the vector `[left bottom width height]`. This table describes each element in the vector.

Element	Description
`left`	Distance from the inner left edge of the parent container to the outer left edge of an imaginary box surrounding the turn coordinator
`bottom`	Distance from the inner bottom edge of the parent container to the outer bottom edge of an imaginary box surrounding the turn coordinator
`width`	Distance between the right and left outer edges of the turn coordinator
`height`	Distance between the top and bottom outer edges of the turn coordinator

All measurements are in pixel units.

The `Position` values are relative to the drawable area of the parent container. The drawable area is the area inside the borders of the container and does not include the area occupied by decorations such as a menu bar or title.

Example: `[200 120 120 120]`

`InnerPosition` — Inner location and size of turn coordinator
`[100 100 120 120]` (default) | `[left bottom width height]`

Inner location and size of the turn coordinator, specified as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`OuterPosition` — Outer location and size of turn coordinator
`[100 100 120 120]]` (default) | `[left bottom width height]`

This property is read-only.

Outer location and size of the turn coordinator returned as `[left bottom width height]`. Position values are relative to the parent container. All measurements are in pixel units. This property value is identical to the `Position` property.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `GridLayoutOptions` object

Layout options, specified as a `GridLayoutOptions` object. This property specifies options for components that are children of grid layout containers. If the component is not a child of a grid layout container (for example, it is a child of a figure or panel), then this property is empty and has no effect. However, if the component is a child of a grid layout container, you can place the component in the desired row and column of the grid by setting the `Row` and `Column` properties on the `GridLayoutOptions` object.

For example, this code places an turn coordinator in the third row and second column of its parent grid.

```
g = uigridlayout([4 3]);
gauge = uiaeroturn(g);
gauge.Layout.Row = 3;
gauge.Layout.Column = 2;
```

To make the turn coordinator span multiple rows or columns, specify the `Row` or `Column` property as a two-element vector. For example, this turn coordinator spans columns 2 through 3:

```
gauge.Layout.Column = [2 3];
```

Callbacks

`CreateFcn` — Creation function

`''` (default) | function handle | cell array | character vector

Object creation function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB creates the object. MATLAB initializes all property values before executing the `CreateFcn` callback. If you do not specify the `CreateFcn` property, then MATLAB executes a default creation function.

Setting the `CreateFcn` property on an existing component has no effect.

If you specify this property as a function handle or cell array, you can access the object that is being created using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

`DeleteFcn` — Deletion function

`''` (default) | function handle | cell array | character vector

Object deletion function, specified as one of these values:

- Function handle.
- Cell array in which the first element is a function handle. Subsequent elements in the cell array are the arguments to pass to the callback function.
- Character vector containing a valid MATLAB expression (not recommended). MATLAB evaluates this expression in the base workspace.

For more information about specifying a callback as a function handle, cell array, or character vector, see “Write Callbacks in App Designer” (MATLAB).

This property specifies a callback function to execute when MATLAB deletes the object. MATLAB executes the `DeleteFcn` callback before destroying the properties of the object. If you do not specify the `DeleteFcn` property, then MATLAB executes a default deletion function.

If you specify this property as a function handle or cell array, you can access the object that is being deleted using the first argument of the callback function. Otherwise, use the `gcbo` function to access the object.

Callback Execution Control

`Interruptible` — Callback interruption

`'on'` (default) | `'off'`

Callback interruption, specified as `'on'` or `'off'`. The `Interruptible` property determines if a running callback can be interrupted.

There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt the running callback (if one exists). The `Interruptible` property of the object owning the running callback determines if interruption is allowed. The `Interruptible` property has two possible values:

- `'on'` — Allows other callbacks to interrupt the object's callbacks. The interruption occurs at the next point where MATLAB processes the queue, such as when there is a `drawnow`, `figure`, `uifigure`, `getframe`, `waitfor`, or `pause` command.
  - If the running callback contains one of those commands, then MATLAB stops the execution of the callback at that point and executes the interrupting callback. MATLAB resumes executing the running callback when the interrupting callback completes.
  - If the running callback does not contain one of those commands, then MATLAB finishes executing the callback without interruption.
- `'off'` — Blocks all interruption attempts. The `BusyAction` property of the object owning the interrupting callback determines if the interrupting callback is discarded or put into a queue.

Note Callback interruption and execution behave differently in these situations:

- If the interrupting callback is a `DeleteFcn`, `CloseRequestFcn` or `SizeChangedFcn` callback, then the interruption occurs regardless of the `Interruptible` property value.
- If the running callback is currently executing the `waitfor` function, then the interruption occurs regardless of the `Interruptible` property value.
- `Timer` objects execute according to schedule regardless of the `Interruptible` property value.

When an interruption occurs, MATLAB does not save the state of properties or the display. For example, the object returned by the `gca` or `gcf` command might change when another callback executes.

BusyAction — Callback queuing
`'queue'` (default) | `'cancel'`

Callback queuing, specified as `'queue'` or `'cancel'`. The `BusyAction` property determines how MATLAB handles the execution of interrupting callbacks. There are two callback states to consider:

- The running callback is the currently executing callback.
- The interrupting callback is a callback that tries to interrupt the running callback.

Whenever MATLAB invokes a callback, that callback attempts to interrupt a running callback. The `Interruptible` property of the object owning the running callback determines if interruption is permitted. If interruption is not permitted, then the `BusyAction` property of the object owning the interrupting callback determines if it is discarded or put in the queue. These are possible values of the `BusyAction` property:

- `'queue'` — Puts the interrupting callback in a queue to be processed after the running callback finishes execution.
- `'cancel'` — Does not execute the interrupting callback.

BeingDeleted — Deletion status
`'off'` | `'on'`

This property is read-only.

Deletion status, returned as `'off'` or `'on'`. MATLAB sets the `BeingDeleted` property to `'on'` when the `DeleteFcn` callback begins execution. The `BeingDeleted` property remains set to `'on'` until the component object no longer exists.

Check the value of the `BeingDeleted` property to verify that the object is not about to be deleted before querying or modifying it.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If no container is specified, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'callback'` | `'off'`

Visibility of the object handle, specified as `'on'`, `'callback'`, or `'off'`.

This property controls the visibility of the object in its parent's list of children. When an object is not visible in its parent's list of children, it is not returned by functions that obtain objects by searching the object hierarchy or querying properties. These functions include `get`, `findobj`, `clf`, and `close`. Objects are valid even if they are not visible. If you can access an object, you can set and get its properties, and pass it to any function that operates on objects.

HandleVisibility Value	Description
`'on'`	The object is always visible.
`'callback'`	The object is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command-line, but allows callback functions to access it.

HandleVisibility Value	Description
`'off'`	The object is invisible at all times. This option is useful for preventing unintended changes to the UI by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the object during the execution of that function.

Identifiers

Type — Type of graphics object
`'uiaeroturn'`

This property is read-only.

Type of graphics object, returned as `'uiaeroturn'`.

Tag — Object identifier
`''` (default) | character vector | string scalar

Object identifier, specified as a character vector or string scalar. You can specify a unique `Tag` value to serve as an identifier for an object. When you need access to the object elsewhere in your code, you can use the `findobj` function to search for the object based on the `Tag` value.

`UserData` — User data
`[]` (default) | array

User data, specified as any MATLAB array. For example, you can specify a scalar, vector, matrix, cell array, character array, table, or structure. Use this property to store arbitrary data on an object.

If you are working in App Designer, create public or private properties in the app to share data instead of using the `UserData` property. For more information, see “Share Data Within App Designer Apps” (MATLAB).

See Also

`uiaeroturn`

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

uiaeroairspeed

Package: `Aero.ui.control`

Create airspeed indicator component

Syntax

```
airspeed = uiaeroairspeed
egt = uiaeroairspeed(parent)
egt = uiaeroairspeed( ___ ,Name,Value)
```

Description

`airspeed = uiaeroairspeed` creates an airspeed indicator in a new figure. MATLAB calls the `uifigure` function to create the figure.

The airspeed indicator displays measurements for aircraft airspeed in knots.

By default, minor ticks represent 10-knot increments and major ticks represent 40-knot increments. The parameters **Minimum** and **Maximum** determine the minimum and maximum values on the gauge. The number and distribution of ticks is fixed, which means that the first and last tick display the minimum and maximum values. The ticks in between distribute evenly between the minimum and maximum values. For major ticks, the distribution of ticks is (**Maximum**-**Minimum**)/9. For minor ticks, the distribution of ticks is (**Maximum**-**Minimum**)/36.

The airspeed indicator has scale color bars that allow for overlapping for the first bar, displayed at a different radius. This different radius lets the gauge represent V_{FE} (maximum speed with flap extended) and V_{SO} (stall speed with flap extended) accurately for aircraft airspeed and stall speed.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`egt = uiaeroairspeed(parent)` specifies the object in which to create the airspeed indicator.

`egt = uiaeroairspeed( ___ ,Name,Value)` specifies airspeed indicator properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

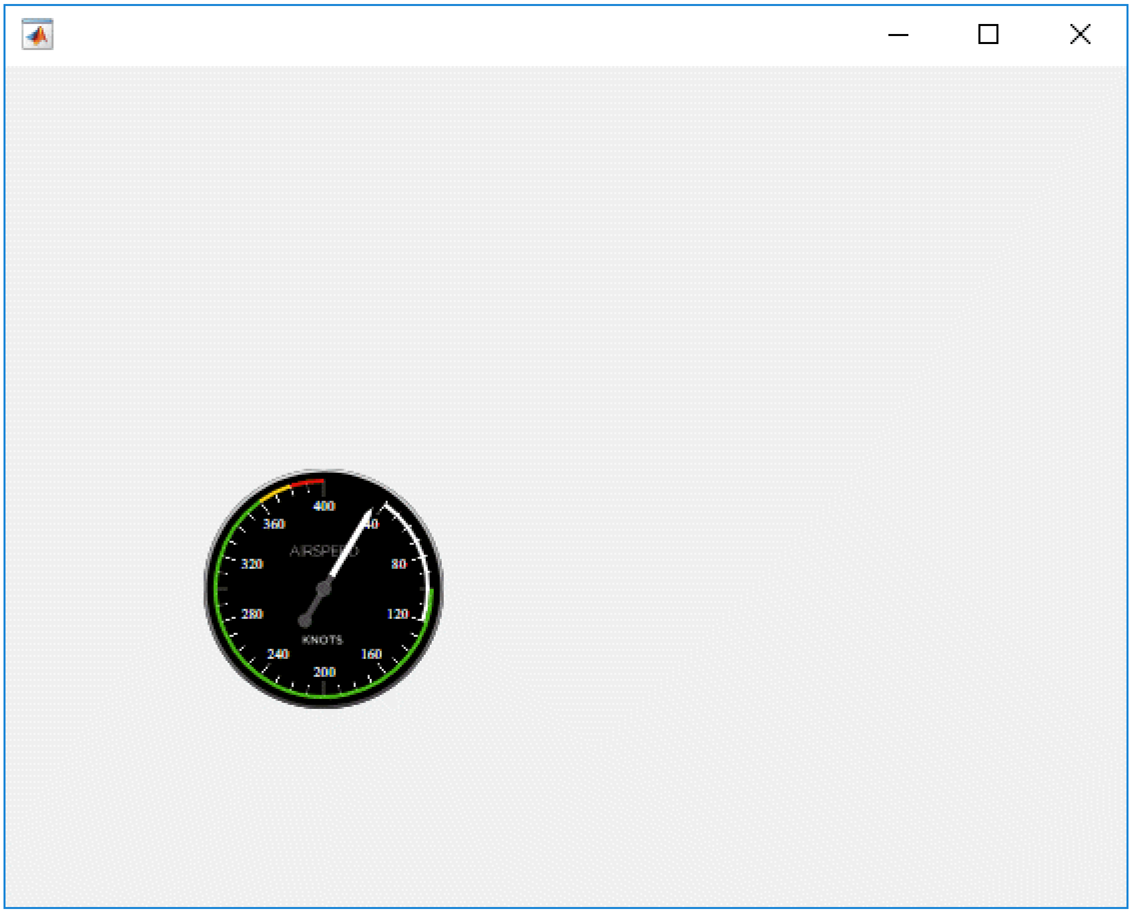
Examples

Create Airspeed Indicator Component

Create an airspeed indicator component named `airspeed`. By default, the function creates a `uifigure` object for the indicator object.

```
airspeed = uiaeroairspeed
```

```
airspeed = 

  AirspeedIndicator (0) with properties:

           Airspeed: 0
        ScaleColors: [4×3 double]
   ScaleColorLimits: [4×2 double]
             Limits: [40 400]
           Position: [100 100 120 120]

  Show all properties
```



Create Figure Window and Airspeed Indicator Component

Create a figure window to contain the airspeed indicator component, then create an airspeed indicator component named `airspeed`.

```
f = uifigure;
airspeed = uiaeroairspeed(f)
```

```
airspeed =
  AirspeedIndicator (0) with properties:

            Airspeed: 0
         ScaleColors: [4×3 double]
    ScaleColorLimits: [4×2 double]
              Limits: [40 400]
            Position: [100 100 120 120]

  Show all properties
```

Input Arguments

parent — Parent container
Figure object (default) | Panel object | Tab object | ButtonGroup object | GridLayout object

Parent container, specified as a Figure object created using the uifigure function, or one of its child containers: Tab, Panel, ButtonGroup, or GridLayout. If you do not specify a parent container, MATLAB calls the uifigure function to create a new Figure object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

For a full list of airspeed indicator properties and descriptions for each type, see AirspeedIndicator Properties.

Output Arguments

airspeed — Airspeed indicator component
object

Airspeed indicator component, returned as an object.

See Also

AirspeedIndicator Properties

Topics

"Create and Configure Flight Instrument Component and an Animation Object" on page 2-65
"Display Flight Trajectory Data Using Flight Instruments and Flight Animation" on page 5-145

Introduced in R2018b

uiaeroaltimeter

Package: `Aero.ui.control`

Create altimeter component

Syntax

```
altimeter = uiaeroaltimeter
altimeter = uiaeroaltimeter(parent)
altimeter = uiaeroaltimeter( ___ ,Name,Value)
```

Description

`altimeter = uiaeroaltimeter` creates an altimeter in a new figure. MATLAB calls the `uifigure` function to create the figure.

The altimeter displays the altitude above sea level in feet, also known as the pressure altitude. It displays the altitude value with needles on a gauge and a numeric indicator.

- The gauge has 10 major ticks. Within each major tick are five minor ticks. This gauge has three needles. Using the needles, the altimeter can display accurately only altitudes between 0 and 100,000 feet.
  - For the longest needle, an increment of a small tick represents 20 feet and a major tick represents 100 feet.
  - For the second longest needle, a minor tick represents 200 feet and a major tick represents 1,000 feet.
  - For the shortest needle a minor tick represents 2,000 feet and a major tick represents 10,000 feet.
- For the numeric display, the gauge shows values as numeric characters between 0 and 9,999 feet. When the numeric display value reaches 10,000 feet, the gauge displays the value as the remaining values below 10,000 feet. For example, 12,345 feet displays as 2,345 feet. When a value is less than 0 (below sea level), the gauge displays `0`. The needles show the appropriate value except for when the value is below sea level or

over 100000 feet. Below sea level, the needles set to `0`, over 100,000, the needles stay set at 100,000.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`altimeter = uiaeroaltimeter(parent)` specifies the object in which to create the altimeter.

`altimeter = uiaeroaltimeter( ___ ,Name,Value)` specifies altimeter properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

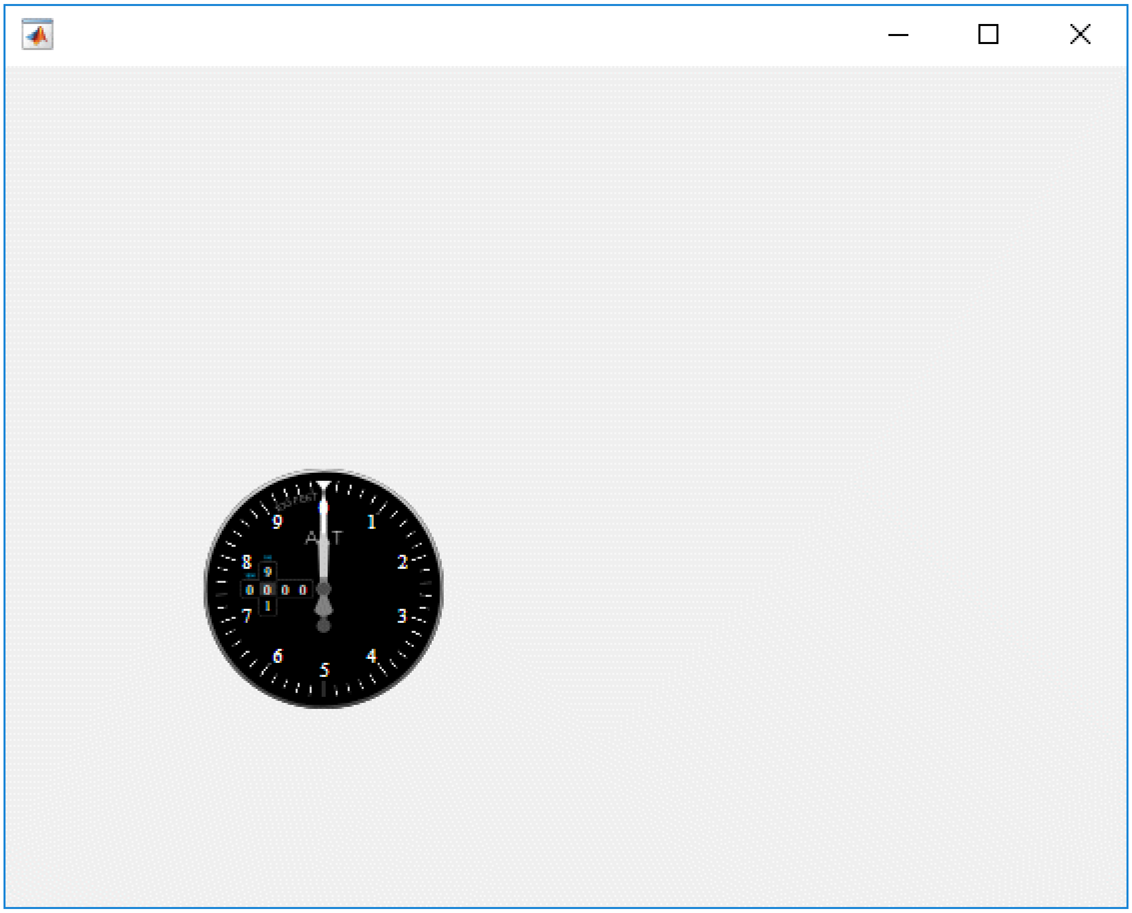
Examples

Create Altimeter Component

Create an altimeter component named `altimeter`. By default, the function creates a `uifigure` object for the indicator object.

```
altimeter = uiaeroaltimeter
```

```
altimeter = 

Altimeter (0) with properties:

    Altitude: 0
    Position: [100 100 120 120]

Show all properties
```



Create Figure Window and Altimeter Component

Create a figure window to contain the altimeter component, then create a altimeter component named `altimeter`.

```
f = uifigure;
altimeter = uiaeroaltimeter(f)
```

```
altimeter =
  Altimeter (0) with properties:

    Altitude: 0
    Position: [100 100 120 120]

  Show all properties
```

Input Arguments

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For a full list of Altimeter properties and descriptions for each type, see Altimeter Properties.

Output Arguments

`altimeter` — Altimeter component
object

Altimeter component, returned as an object.

See Also

Altimeter Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

uiaeroclimb

Package: `Aero.ui.control`

Create climb rate indicator component

Syntax

```
climbrate = uiaeroclimb
climbrate = uiaeroclimb(parent)
climbrate = uiaeroclimb( ___ ,Name,Value)
```

Description

`climbrate = uiaeroclimb` creates a climb rate indicator in a new figure. MATLAB calls the `uifigure` function to create the figure.

The climb rate indicator displays measurements for an aircraft climb rate in ft/min.

The needle covers the top semicircle, if the velocity is positive, and the lower semicircle, if the climb rate is negative. The range of the indicator is from –**Maximum** feet per minute to **Maximum** feet per minute. Major ticks indicate **Maximum**/4. Minor ticks indicate **Maximum**/8 and **Maximum**/80.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`climbrate = uiaeroclimb(parent)` specifies the object in which to create the climb rate indicator.

`climbrate = uiaeroclimb( ___ ,Name,Value)` specifies climb rate indicator properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

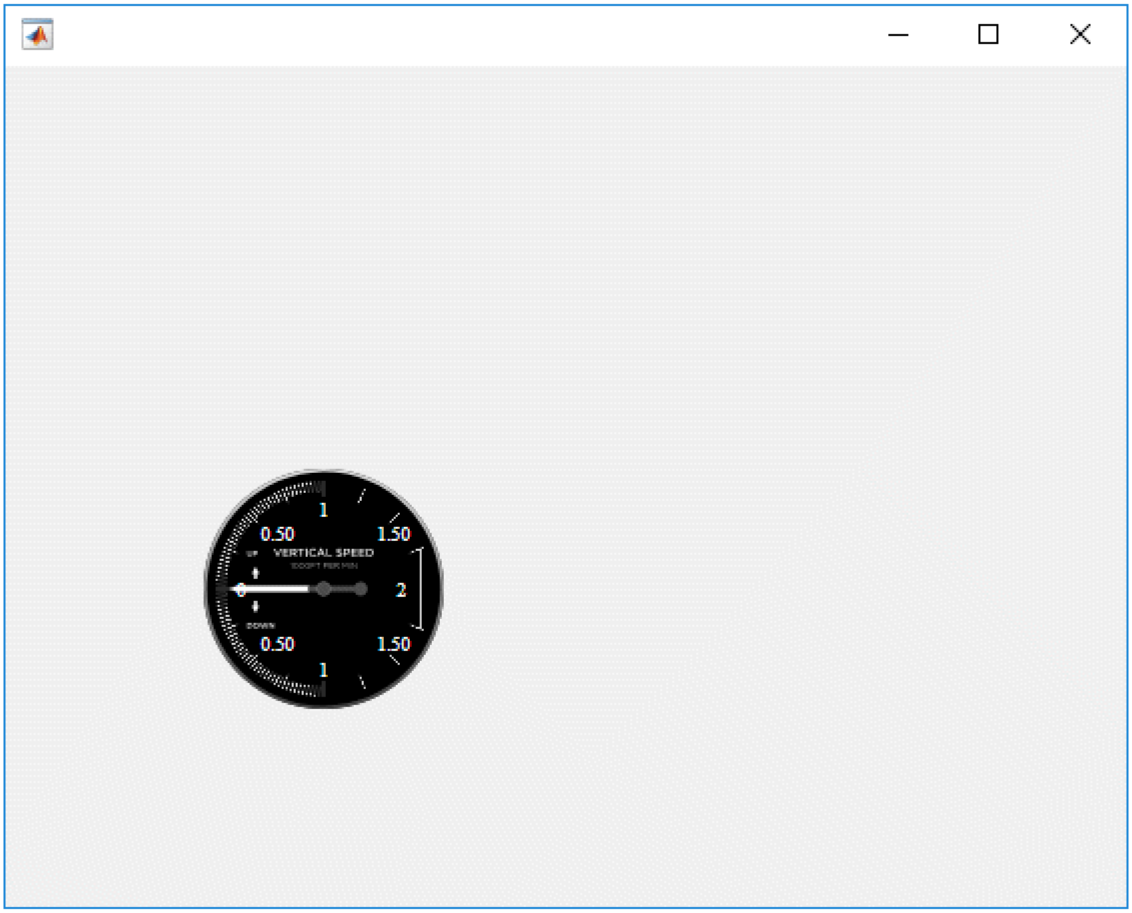
Examples

Create Climb Rate Indicator Component

Create a climb rate indicator indicator component named `climbrate`. By default, the function creates a `uifigure` object for the indicator object.

```
climbrate = uiaeroclimb
```

```
climbrate = 

  ClimbIndicator (0) with properties:

      ClimbRate: 0
    MaximumRate: 2000
       Position: [100 100 120 120]

  Show all properties
```



Create Figure Window and Climb Rate Indicator Component

Create a figure window to contain the climb rate indicator component, then create an climb rate indicator component named `climbrate`.

```
f = uifigure;
climbrate = uiaeroclimb(f)
```

```
climbrate =
  ClimbIndicator (0) with properties:

      ClimbRate: 0
    MaximumRate: 2000
       Position: [100 100 120 120]

  Show all properties
```

Input Arguments

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For a full list of climb rate indicator properties and descriptions for each type, see ClimbIndicator Properties.

Output Arguments

`climbrate` — Climb rate indicator component
object

Climb rate indicator component, returned as an object.

See Also

ClimbIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

uiaeroegt

Package: `Aero.ui.control`

Create exhaust gas temperature (EGT) indicator component

Syntax

```
egt = uiaeroegt
egt = uiaeroegt(parent)
egt = uiaeroegt( ___ ,Name,Value)
```

Description

`egt = uiaeroegt` creates an EGT indicator in a new figure. MATLAB calls the `uifigure` function to create the figure.

The EGT indicator displays temperature measurements for engine exhaust gas temperature (EGT) in Celsius.

This gauge displays values using both:

- A needle on a gauge. A major tick is (**Maximum**-**Minimum**)/1,000 degrees, a minor tick is (**Maximum**-**Minimum**)/200 degrees Celsius.
- A numeric indicator. The operating range for the indicator goes from **Minimum** to **Maximum** degrees Celsius.

If the value of the signal is under **Minimum**, the needle displays 5 degrees under the **Minimum** value, the numeric display shows the **Minimum** value. If the value exceeds the **Maximum** value, the needle displays 5 degrees over the maximum tick, and the numeric displays the **Maximum** value.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`egt = uiaeroegt(parent)` specifies the object in which to create the EGT indicator.

`egt = uiaeroegt( ___ ,Name,Value)` specifies EGT indicator properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

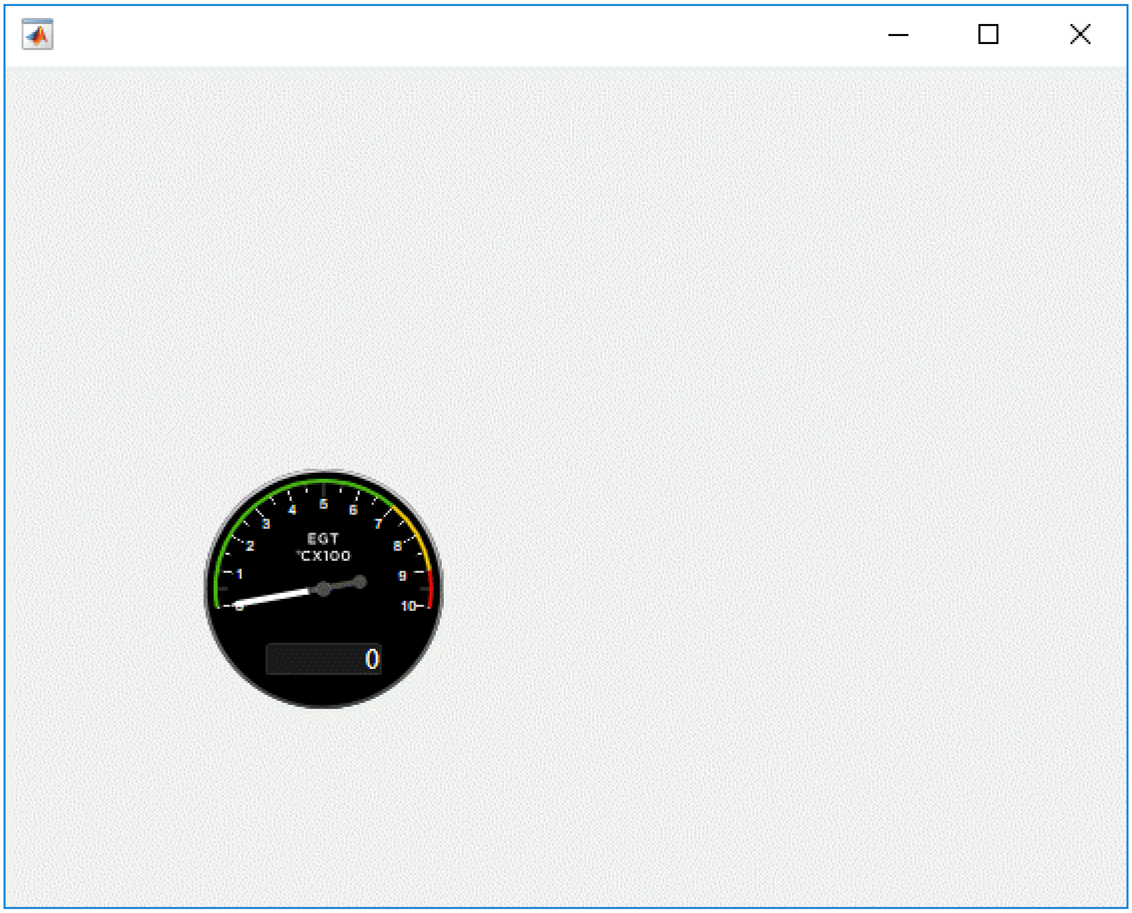
Examples

Create EGT Indicator Component

Create an EGT indicator component named `egt`. By default, the function creates a `uifigure` object for the indicator object.

```
egt = uiaeroegt
```

```
egt = 

  EGTIndicator (0) with properties:

         Temperature: 0
         ScaleColors: [3×3 double]
    ScaleColorLimits: [3×2 double]
              Limits: [0 1000]
            Position: [100 100 120 120]

  Show all properties
```



Create Figure Window and EGT Indicator Component

Create a figure window to contain the EGT indicator component, then create an EGT indicator component named `egt`.

```
f = uifigure;
egt = uiaeroegt(f)
```

```
egt =
EGTIndicator (0) with properties:

       Temperature: 0
       ScaleColors: [3×3 double]
  ScaleColorLimits: [3×2 double]
            Limits: [0 1000]
          Position: [100 100 120 120]

Show all properties
```

Input Arguments

parent — Parent container
Figure object (default) | Panel object | Tab object | ButtonGroup object | GridLayout object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For a full list of EGT indicator properties and descriptions for each type, see EGTIndicator Properties.

Output Arguments

egt — EGT indicator component
object

EGT indicator component, returned as an object.

See Also
EGTIndicator Properties

Topics
“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

uiaeroheading

Package: `Aero.ui.control`

Create heading indicator component

Syntax

```
heading = uiaeroheading
heading = uiaeroheading(parent)
heading = uiaeroheading( ___ ,Name,Value)
```

Description

`heading = uiaeroheading` creates a heading indicator in a new figure. MATLAB calls the `uifigure` function to create the figure.

The heading indicator displays measurements for aircraft heading in degrees.

The heading indicator represents values between 0 and 360 degrees.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`heading = uiaeroheading(parent)` specifies the object in which to create the heading indicator.

`heading = uiaeroheading( ___ ,Name,Value)` specifies heading indicator properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

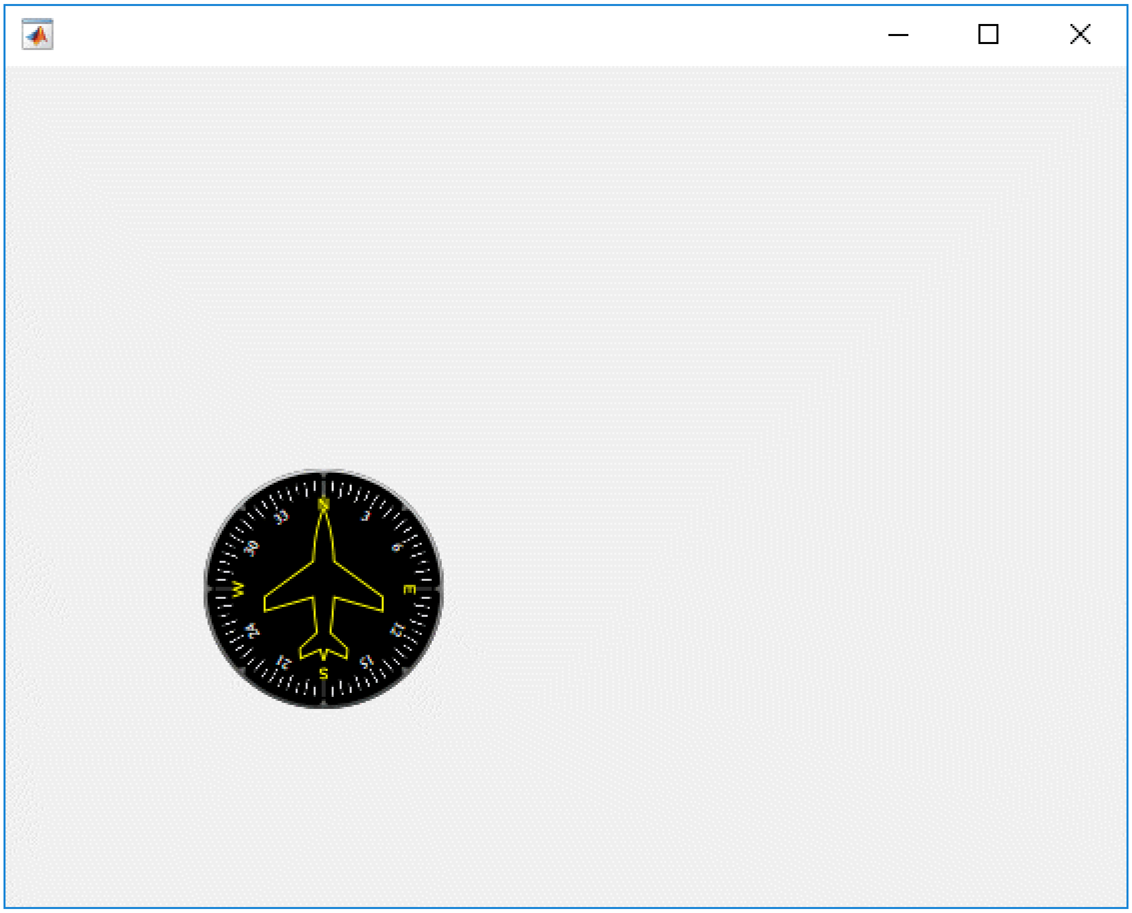
Examples

Create Heading Indicator Component

Create a heading indicator component named `heading`. By default, the function creates a `uifigure` object for the indicator object.

```
heading = uiaeroheading
```

```
heading = 

HeadingIndicator (0) with properties:

     Heading: 0
    Position: [100 100 120 120]
```



Create Figure Window and Heading Indicator Component

Create a figure window to contain the heading component, then create a heading indicator component named `heading`.

```
f = uifigure;
heading = uiaeroheading(f)
```

```
heading =
HeadingIndicator (0) with properties:
     Heading: 0
    Position: [100 100 120 120]

Show all properties
```

Input Arguments

parent — Parent container
Figure object (default) | Panel object | Tab object | ButtonGroup object | GridLayout object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For a full list of heading indicator properties and descriptions for each type, see HeadingIndicator Properties.

Output Arguments

heading — Heading indicator component
object

Heading indicator component, returned as an object.

See Also

HeadingIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

uiaerohorizon

Package: `Aero.ui.control`

Create artificial horizon component

Syntax

```
horizon = uiaerohorizon
horizon = uiaerohorizon(parent)
horizon = uiaerohorizon( ___ ,Name,Value)
```

Description

`horizon = uiaerohorizon` creates an artificial horizon in a new figure. MATLAB calls the `uifigure` function to create the figure.

The artificial horizon represents aircraft attitude relative to horizon and displays roll and pitch in degrees:

- Values for roll cannot exceed +/– 90 degrees.
- Values for pitch cannot exceed +/– 30 degrees.

If the values exceed the maximum values, the gauge maximum and minimum values do not change.

Changes in roll value affect the gauge semicircles and the ticks located on the black arc turn accordingly. Changes in pitch value affect the scales and the distribution of the semicircles.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`horizon = uiaerohorizon(parent)` specifies the object in which to create the artificial horizon.

`horizon = uiaerohorizon( ___ ,Name,Value)` specifies artificial horizon properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

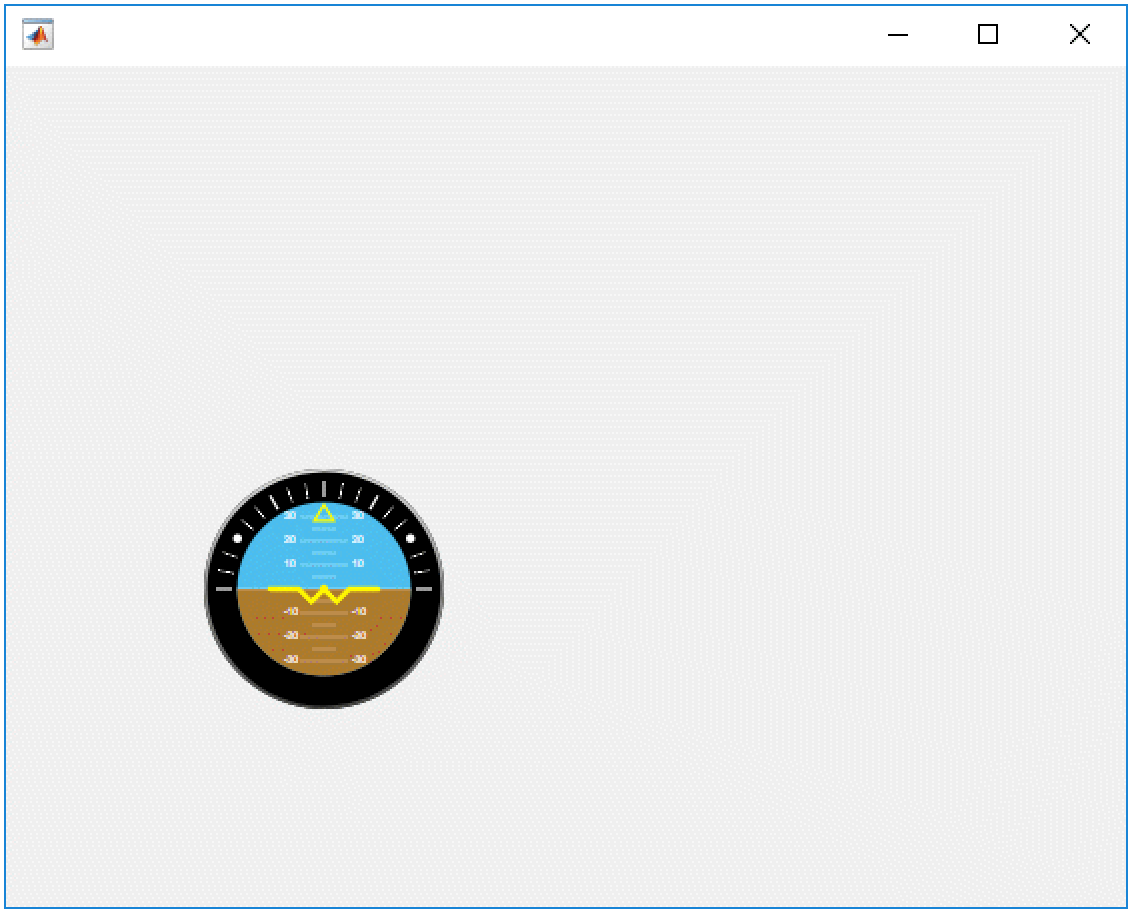
Examples

Create Artificial Horizon Component

Create an artificial horizon component named `horizon`. By default, the function creates a `uifigure` object for the indicator object.

```
horizon = uiaerohorizon
```

```
horizon = 

  ArtificialHorizon ([0  0]) with properties:

       Pitch: 0
        Roll: 0
    Position: [100 100 120 120]

  Show all properties
```



Create Figure Window and Artificial Horizon Component

Create a figure window to contain the artificial horizon component, then create an artificial horizon component named `horizon`.

```
f = uifigure;
egt = uiaeroegt(f)
```

```
horizon =
  ArtificialHorizon ([0  0]) with properties:

        Pitch: 0
         Roll: 0
     Position: [100 100 120 120]

  Show all properties
```

Input Arguments

`parent` — Parent container
`Figure` object (default) | `Panel` object | `Tab` object | `ButtonGroup` object | `GridLayout` object

Parent container, specified as a `Figure` object created using the `uifigure` function, or one of its child containers: `Tab`, `Panel`, `ButtonGroup`, or `GridLayout`. If you do not specify a parent container, MATLAB calls the `uifigure` function to create a new `Figure` object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

For a full list of artificial horizon properties and descriptions for each type, see ArtificialHorizon Properties.

Output Arguments

`horizon` — Artificial horizon component
object

Artificial horizon component, returned as an object.

See Also

ArtificialHorizon Properties

Topics

"Create and Configure Flight Instrument Component and an Animation Object" on page 2-65
"Display Flight Trajectory Data Using Flight Instruments and Flight Animation" on page 5-145

Introduced in R2018b

uiaerorpm

Package: `Aero.ui.control`

Create revolutions per minute (RPM) indicator component

Syntax

```
rpm = uiaerorpm
rpm = uiaerorpm(parent)
rpm = uiaerorpm( ___ ,Name,Value)
```

Description

`rpm = uiaerorpm` creates an RPM indicator in a new figure. MATLAB calls the `uifigure` function to create the figure.

The RPM indicator displays measurements for engine revolutions per minute in percentage of RPM.

The range of values for RPM goes from 0 to 110%. Minor ticks represent increments of 5% RPM and major ticks represent increments of 10% RPM.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`rpm = uiaerorpm(parent)` specifies the object in which to create the RPM indicator.

`rpm = uiaerorpm( ___ ,Name,Value)` specifies RPM indicator properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

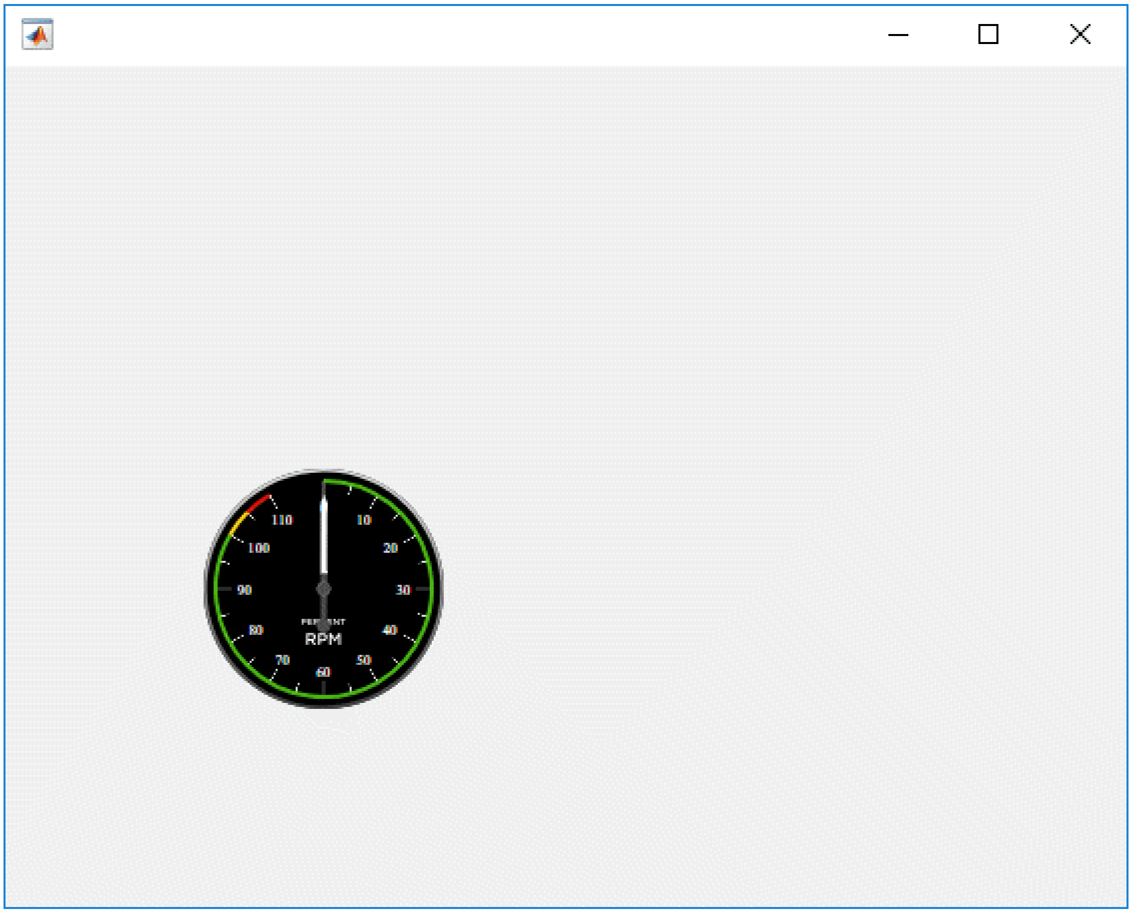
Examples

Create RPM Indicator Component

Create an RPM indicator component named `rpm`. By default, the function creates a `uifigure` object for the indicator object.

```
rpm = uiaerorpm
```

```
rpm = 

  RPMIndicator (0) with properties:

                 RPM: 0
         ScaleColors: [3×3 double]
    ScaleColorLimits: [3×2 double]
            Position: [100 100 120 120]

  Show all properties
```



Create Figure Window and RPM Indicator Component

Create a figure window to contain the RPM indicator component, then create an RPM indicator component named `rpm`.

```
f = uifigure;
rpm = uiaerorpm(f)
```

```
rpm =
  RPMIndicator (0) with properties:

                 RPM: 0
         ScaleColors: [3×3 double]
    ScaleColorLimits: [3×2 double]
            Position: [100 100 120 120]

  Show all properties
```

Input Arguments

parent — Parent container
Figure object (default) | Panel object | Tab object | ButtonGroup object | GridLayout object

Parent container, specified as a Figure object created using the uifigure function, or one of its child containers: Tab, Panel, ButtonGroup, or GridLayout. If you do not specify a parent container, MATLAB calls the uifigure function to create a new Figure object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

For a full list of RPM indicator properties and descriptions for each type, see RPMIndicator Properties.

Output Arguments

rpm — RPM indicator
object

RPM indicator component, returned as an object.

See Also

RPMIndicator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

uiaeroturn

Package: `Aero.ui.control`

Create turn coordinator component

Syntax

```
turn = uiaeroturn
turn = uiaeroturn(parent)
turn = uiaeroturn( ___ ,Name,Value)
```

Description

`turn = uiaeroturn` creates a turn coordinator in a new figure. MATLAB calls the `uifigure` function to create the figure.

The turn coordinator displays measurements on a turn coordinator and inclinometer. These measurements help determine if the turn is coordinated, slipped, or skidded. The turn is a coordinated turn that combines the rolling and yawing of a turn. The turn indicator signal turns the airplane in the gauge, in degrees. The inclinometer turns the ball in the gauge, in degrees. Together, these signals show the slip and skid of an airplane as it turns. Both values cannot exceed +/–15 degrees. If values exceed 15 degrees, the gauge stays fixed at the minimum or maximum value.

Note Use this function only with figures created using the `uifigure` function. Apps created using GUIDE or the `figure` function do not support flight instrument components.

`turn = uiaeroturn(parent)` specifies the object in which to create the turn coordinator.

`turn = uiaeroturn( ___ ,Name,Value)` specifies turn coordinator properties using one or more `Name,Value` pair arguments. Use this option with any of the input argument combinations in the previous syntaxes.

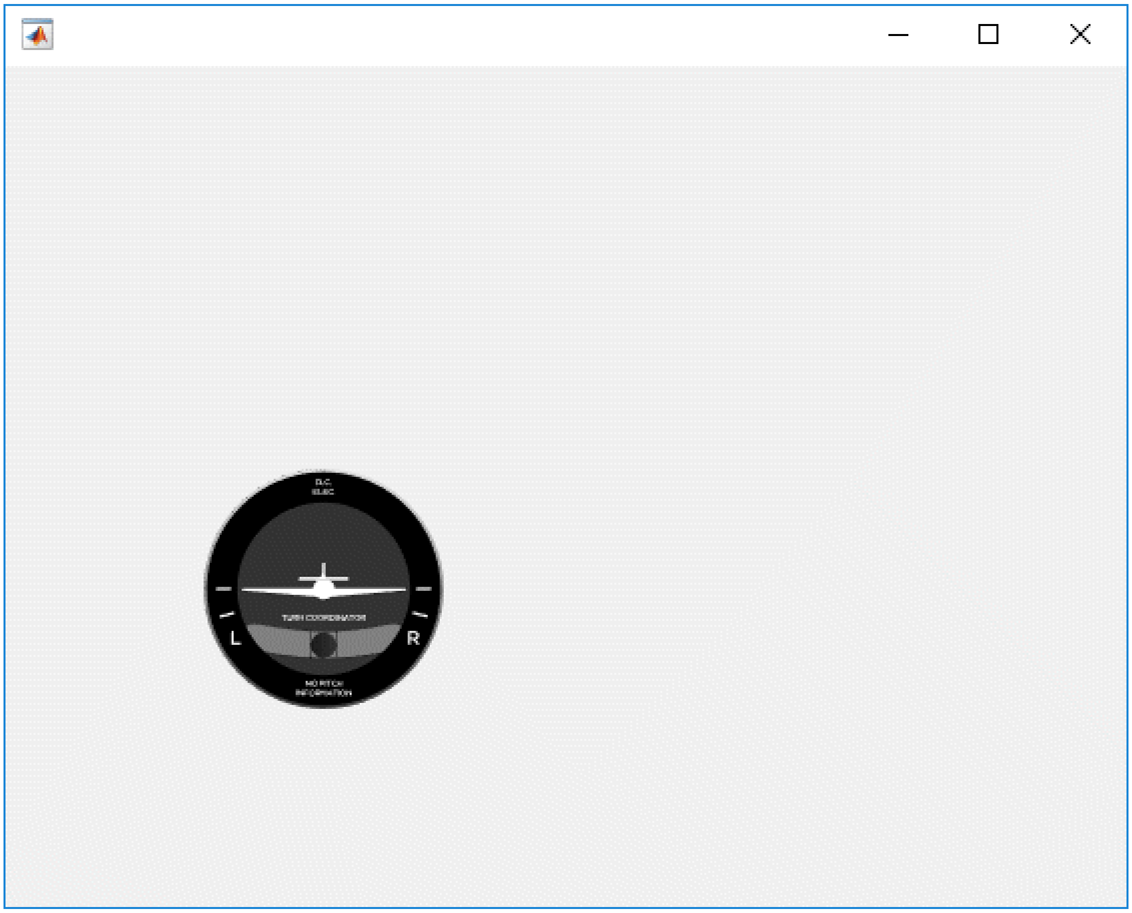
Examples

Create Turn Coordinator Component

Create a turn coordinator component named `turn`.

```
turn = uiaeroturn
```

```
turn = 

  TurnCoordinator ([0  0]) with properties:

        Turn: 0
        Slip: 0
    Position: [100 100 120 120]

  Show all properties
```



Create Figure Window and Turn Coordinator Component

Create a figure window to contain the turn coordinator component, then create a turn coordinator component named `turn`.

```
f = uifigure;
turn = uiaeroturn(f)
```

```
turn =
  TurnCoordinator ([0  0]) with properties:

        Turn: 0
        Slip: 0
    Position: [100 100 120 120]

  Show all properties
```

Input Arguments

parent — Parent container
Figure object (default) | Panel object | Tab object | ButtonGroup object | GridLayout object

Parent container, specified as a Figure object created using the uifigure function, or one of its child containers: Tab, Panel, ButtonGroup, or GridLayout. If you do not specify a parent container, MATLAB calls the uifigure function to create a new Figure object that serves as the parent container.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

For a full list of turn coordinator properties and descriptions for each type, see TurnCoordinator Properties.

Output Arguments

turn — Turn coordinator component
object

Turn coordinator component, returned as an object.

See Also

TurnCoordinator Properties

Topics

“Create and Configure Flight Instrument Component and an Animation Object” on page 2-65
“Display Flight Trajectory Data Using Flight Instruments and Flight Animation” on page 5-145

Introduced in R2018b

update (Aero.Body)

Change body position and orientation as function of time

Syntax

```
update(h,t)
h.update(t)
```

Description

`update(h,t)` and `h.update(t)` change body position and orientation of body `h` as a function of time `t`. *t* is a scalar in seconds.

Note This function requires that you load the body geometry and time series data first.

Examples

Update the body `b` with time in seconds of 5.

```
b=Aero.Body;
b.load('pa24-250_orange.ac','Ac3d');
tsdata = [ ...
    0,  1,1,1, 0,0,0; ...
    10  2,2,2, 1,1,1; ];
b.TimeSeriesSource = tsdata;
b.update(5);
```

See Also

`load`

Introduced in R2007a

update (Aero.Camera)

Update camera position based on time and position of other Aero.Body objects

Syntax

```
update(h,newtime,bodies)
h.update(newtime,bodies)
```

Description

`update(h,newtime,bodies)` and `h.update(newtime,bodies)` update the camera object, `h`, position and aim point data based on the new time, `newtime`, and position of other `Aero.Body` objects, `bodies`. This function updates the camera object `PrevTime` property to `newtime`.

See Also

`play`

Introduced in R2007a

update (Aero.FlightGearAnimation)

Update position data to FlightGear animation object

Syntax

```
update(h,time)
h.update(time)
```

Description

`update(h,time)` and `h.update(time)` update the position data to the FlightGear animation object via UDP. It sets the new position and attitude of body `h`. `time` is a scalar in seconds.

Note This function requires that you load the time series data and run FlightGear first.

Examples

Configure a body with `TimeSeriesSource` set to `simdata`, then update the body with time *`time`* equal to 0.

```
h = Aero.FlightGearAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
load simdata;
h.TimeSeriesSource = simdata;
t = 0;
h.update(t);
```

See Also

`GenerateRunScript` | `initialize` | `play`

Introduced in R2007a

update (Aero.Node)

Change node position and orientation versus time data

Syntax

```
update(h,t)
h.update(t)
```

Description

`update(h,t)` and `h.update(t)` change node position and orientation of node `h` as a function of time `t`. *t* is a scalar in seconds.

Note This function requires that you load the node and time series data first.

Examples

Move the Lynx body.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
load takeoffData
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWithTime';
h.Nodes{7}.update(5);
```

See Also

`updateNodes`

Introduced in R2007b

updateBodies

Class: `Aero.Animation`
Package: `Aero`

Update bodies of animation object

Syntax

```
h = updateBodies(time)
h.updateBodies(time)
```

Description

`h = updateBodies(time)` and `h.updateBodies(time)` set the new position and attitude of movable bodies in the animation object `h`. This function updates the bodies contained in the animation `object h`. `time` is a scalar in seconds.

Examples

Configure a body with `TimeSeriesSource` set to `simdata`, then update the body with time *t* equal to 0.

```
h = Aero.Animation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
load simdata;
h.Bodies{1}.TimeSeriesSource = simdata;
t = 0;
h.updateBodies(t);
```

updateCamera

Class: `Aero.Animation`
Package: `Aero`

Update camera in animation object

Syntax

```
updateCamera(h,time)
h.updateCamera(time)
```

Description

`updateCamera(h,time)` and `h.updateCamera(time)` update the camera in the animation object `h`. `time` is a scalar in seconds.

Note The `PositionFcn` property of a camera object controls the camera position relative to the bodies in the animation. The default camera `PositionFcn` follows the path of a first order chase vehicle. Therefore, it takes a few steps for the camera to position itself correctly in the chase plane position.

Input Arguments

`h`	Animation object.
`time`	Scalar in seconds.

Examples

Configure a body with `TimeSeriesSource` set to `simdata`, then update the camera with time *t* equal to 0.

```
h = Aero.Animation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
load simdata;
h.Bodies{1}.TimeSeriesSource = simdata;
t = 0;
h.updateCamera(t);
```

updateNodes (Aero.VirtualRealityAnimation)

Change virtual reality animation node position and orientation as function of time

Syntax

```
updateNodes(h,t)
h.updateNotes(t)
```

Description

`updateNodes(h,t)` and `h.updateNotes(t)` change node position and orientation of body `h` as a function of time `t`. *t* is a scalar in seconds.

Note This function requires that you load the node and time series data first.

Examples

Update the node `h` with time in 5 seconds.

```
h = Aero.VirtualRealityAnimation;
h.FramesPerSecond = 10;
h.TimeScaling = 5;
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
h.initialize();
load takeoffData
h.Nodes{7}.TimeseriesSource = takeoffData;
h.Nodes{7}.TimeseriesSourceType = 'StructureWithTime';
h.Nodes{7}.CoordTransformFcn = @vranimCustomTransform;
h.updateNodes(5);
```

See Also

`addNode` | `update`

Introduced in R2007b

Viewpoint (Aero.Viewpoint)

Create viewpoint object for use in virtual reality animation

Syntax

```
h = Aero.Viewpoint
```

Description

`h = Aero.Viewpoint` creates a viewpoint object for use with virtual reality animation.

See `Aero.Viewpoint` for further details.

Introduced in R2007b

VirtualRealityAnimation (Aero.VirtualRealityAnimation)

Construct virtual reality animation object

Syntax

```
h = Aero.VirtualRealityAnimation
```

Description

`h = Aero.VirtualRealityAnimation` constructs a virtual reality animation object. The animation object is returned to `h`.

See `Aero.VirtualRealityAnimation` for further details.

See Also

`Aero.VirtualRealityAnimation`

Introduced in R2007b

wrldmagm

Use World Magnetic Model

Syntax

```
[xyz,h,dec,dip,f] = wrldmagm(height,lat,lon,dyear)
[xyz,h,dec,dip,f] = wrldmagm(height,lat,lon,dyear,model)
[xyz,h,dec,dip,f] = wrldmagm(height,lat,lon,dyear,'Custom',filename)
```

Description

`[xyz,h,dec,dip,f] = wrldmagm(height,lat,lon,dyear)` calculates the Earth magnetic field at a specific location and time using World Magnetic Model (WMM) model WMM2015v2, which is valid from January 1, 2015 to December 31, 2019.

Note WMM2015v2 supersedes WMM2015 and corrects performance degradation issues present in WMM2015 above 55-degrees latitude in the Northern hemisphere.

`[xyz,h,dec,dip,f] = wrldmagm(height,lat,lon,dyear,model)` calculates the Earth magnetic field using World Magnetic Model `model`.

`[xyz,h,dec,dip,f] = wrldmagm(height,lat,lon,dyear,'Custom',filename)` calculates the Earth magnetic field using the World Magnetic Model defined in the `WMM.cof` file. The `WMM.cof` files must be in their original form as provided by `NOAA`.

Examples

Calculate Magnetic Model Using WMM2015v2

Calculate the magnetic model 1000 meters over Natick, Massachusetts, on July 4, 2015, using the WMM2015v2 model.

```
[XYZ, H, DEC, DIP, F] = wrldmagm(1000, 42.283, -71.35, decyear(2015,7,4),'2015')
```

```
XYZ =
   1.0e+04 *
    1.9455
   -0.5087
    4.8169

H =
   2.0110e+04

DEC =
  -14.6533

DIP =
   67.3405

F =
   5.2198e+04
```

Calculate Magnetic Model for Downloaded `WMM.COF` file

Calculate the magnetic model 1000 meters over Natick, Massachusetts, on July 4, 2015, using a downloaded `WMM.COF` file.

```
[XYZ, H, DEC, DIP, F] = wrldmagm(1000, 42.283, -71.35, decyear(2015,7,4),'Custom','WMM.COF')
```

```
XYZ =
   1.0e+04 *
    1.9455
   -0.5087
    4.8169

H =
   2.0110e+04

DEC =
  -14.6533

DIP =
   67.3405
```

```
F =
   5.2198e+04
```

Input Arguments

`height` — Height
scalar

Height of the magnetic field, specified as a scalar, in meters.

Data Types: `double`

`lat` — Geodetic latitude
scalar

Geodetic latitude, specified as a scalar, in degrees. North latitude is positive and south latitude is negative.

Data Types: `double`

`lon` — Geodetic longitude
scalar

Geodetic longitude, specified as a scalar, in degrees. East longitude is positive and west latitude is negative.

Data Types: `double`

`dyear` — Decimal year
scalar

Decimal year is the desired year in a decimal format, specified as a scalar. This value can have any fraction of the year that has already passed.

Data Types: `double`

`model` — World Magnetic Model
`'2015v2'` (default) | `'2015'` | `'2015v1'` | `'2010'` | `'2005'` | `'2000'` | character vector

World Magnetic Model, specified as a character vector or string.

Model	Description
`'2015v2'` or `'2015'`	WMM2015v2 (epoch 2015–2020).
`'2015v1'`	WMM2015 (epoch 2015–2020). This version is not recommended. Use `'2015v2'` (WMM2015v2) instead.
`'2010'`	WMM2010 (epoch 2010–2015).
`'2005'`	WMM2005 (epoch 2005–2010).
`'2000'`	WMM2000 (epoch 2000–2005).

Data Types: `char` | `string`

`'Custom',filename` — Coefficient file
coefficient file name

Coefficient file, `WMM.COF`, downloaded from https://www.ngdc.noaa.gov/geomag/WMM/DoDWMM.shtml.

Example: `'Custom','WMM.COF'`

Data Types: `char` | `string`

Output Arguments

xyz — Magnetic field vector
vector

Magnetic field vector, returned as a vector, in nanotesla.

h — Horizontal intensity
scalar

Horizontal intensity, returned as a scalar, in nanotesla.

dec — Declination
scalar

Declination, returned as a scalar, in degrees.

dip — Inclination
scalar

Inclination, returned as a scalar, in degrees.

f — Total intensity
scalar

Total intensity, returned as a scalar, in nanotesla.

Limitations

- The WMM specification produces data that is reliable five years after the epoch of the model, which begins January 1 of the model year selected. The WMM specification describes only the long-wavelength spatial magnetic fluctuations due to the Earth core. Intermediate and short-wavelength fluctuations, contributed from the crustal field (the mantle and crust), are not included. Also, the substantial fluctuations of the geomagnetic field, which occur constantly during magnetic storms and almost constantly in the disturbance field (auroral zones), are not included.
- WMM2015v2 was released by NOAA in February, 2019 to correct performance degradation issues in the Arctic region from January 1, 2015 to December 31, 2019. WMM2015v2 supersedes WMM2015. Consider replacing WMM2015 with WMM2015v2 for use with navigation and other systems. It is still acceptable to use WMM2015 in systems below 55 degrees latitude in the Northern hemisphere.

See Also

`decyear` | `igrfmagm`

External Websites

https://www.ngdc.noaa.gov/geomag/WMM/DoDWMM.shtml

Introduced in R2006b

Bodies property

Class: `Aero.Animation`
Package: `Aero`

Specify name of animation object

Values

`MATLAB array`

Default: `[ ]`

Description

This property specifies the bodies that the animation object contains.

Camera property

Class: `Aero.Animation`
Package: `Aero`

Specify camera that animation object contains

Values

`handle`

Default: `[ ]`

Description

This property specifies the camera that the animation object contains.

Figure property

Class: `Aero.Animation`
Package: `Aero`

Specify name of figure object

Values

`MATLAB array`

Default: `[ ]`

Description

This property specifies the name of the figure object.

FigureCustomizationFcn property

Class: `Aero.Animation`
Package: `Aero`

Specify figure customization function

Values

`MATLAB array`

Default: `[ ]`

Description

This property specifies the figure customization function.

FramesPerSecond property

Class: `Aero.Animation`
Package: `Aero`

Animation rate

Values

`MATLAB array`

Default: `12`

Description

This property specifies rate in frames per second.

Name property

Class: `Aero.Animation`
Package: `Aero`

Specify name of animation object

Values

`Character vector | string`

Default: `' '`

Description

This property specifies the name of the animation object.

TCurrent property

Class: `Aero.Animation`
Package: `Aero`

Current time

Values

`double`

Default: `0`

Description

This property specifies the current time.

TFinal property

Class: `Aero.Animation`
Package: `Aero`

End time

Values

`double`

Default: `NaN`

Description

This property specifies the end time.

TimeScaling property

Class: `Aero.Animation`
Package: `Aero`

Scaling time

Values

`double`

Default: `1`

Description

This property specifies the time, in seconds.

TStart property

Class: `Aero.Animation`
Package: `Aero`

Start time

Values

`double`

Default: `NaN`

Description

This property specifies the start time.

VideoCompression property

Class: `Aero.Animation`
Package: `Aero`

Video recording compression file type

Values

`'Archival'`

Create Motion JPEG 2000 format file with lossless compression.

`'Motion JPEG AVI'`

Create compressed AVI format file using Motion JPEG codec.

`'Motion JPEG 2000'`

Create compressed Motion JPEG 2000 format file.

`'MPEG-4'`

Create compressed MPEG-4 format file with H.264 encoding (Windows 7 systems only).

`'Uncompressed AVI'`

Create uncompressed AVI format file with RGB24 video.

Data type: `Aero.VideoProfileTypeEnum`

Default: `'Archival'`

Description

This property specifies the compression file type to create. For more information on video compression, see `VideoWriter`.

VideoFileName property

Class: `Aero.Animation`
Package: `Aero`

Video recording file name

Values

filename

Data type: character vector | string

Default: `temp`

Description

This property specifies the file name for the video recording.

VideoQuality property

Class: `Aero.Animation`
Package: `Aero`

Video recording quality

Values

Value between `0` and `100`

Data type: double

Default: `75`

Description

This property specifies the recording quality. For more information on video quality, see the `Quality` property in `VideoWriter`.

VideoRecord property

Class: `Aero.Animation`
Package: `Aero`

Video recording

Values

`‘on’`

Enable video recording.

`‘off’`

Disable video recording.

`‘scheduled’`

Schedule video recording. Use this setting with the `VideoTStart` and `VideoTFinal` properties.

Data type: character vector | string

Default: `'off'`

Description

This property enables video recording of animation objects.

If you are capturing frames of a plot that takes a long time to generate or are repeatedly capturing frames in a loop, make sure that your computer's screen saver does not activate and that your monitor does not turn off for the duration of the capture; otherwise one or more of the captured frames can contain graphics from your screen saver or nothing at all.

Note In situations where MATLAB software is running on a virtual desktop that is not currently visible on your monitor, it may capture a region on your monitor that corresponds to the position occupied by the figure or axes on the hidden desktop. Therefore, make sure that the window to be captured exists on the currently active desktop.

Examples

Record Animation Object Simulation

Simulate and record flight data. Create an animation object.

```
h = Aero.Animation;
```

Control the frame display rate.

```
h.FramesPerSecond = 10;
```

Set the time-scaling (`TimeScaling`) property on the animation object to specify the data per second.

```
h.TimeScaling = 5;
```

The combination of `FramesPerSecond` and `TimeScaling` properties determines the time step of the simulation. These settings result in a time step of approximately 0.5 s.

Create and load a body for the animation object.

```
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
```

Load simulated flight trajectory data (`simdata`), located in *`matlabroot`*`\toolbox\aero\astdemos`.

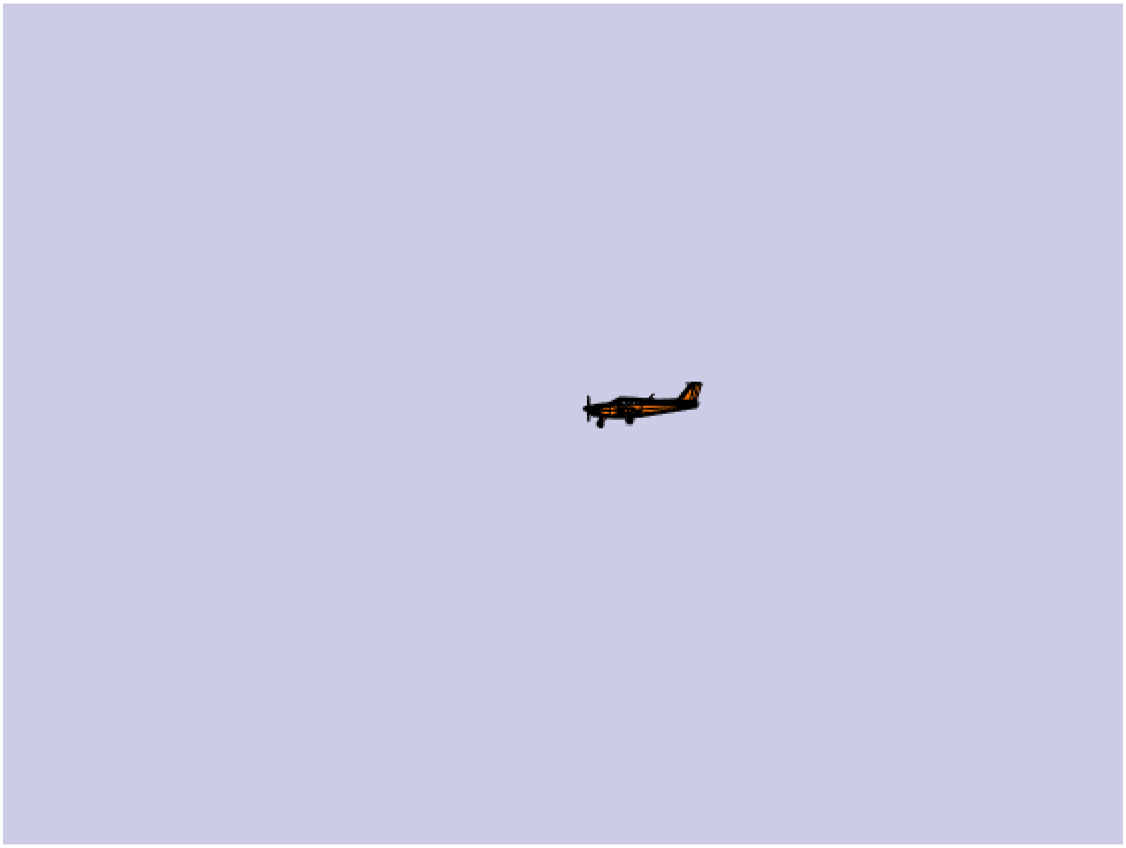
```
load simdata;
```

Set the time series data for the body.

```
h.Bodies{1}.TimeSeriesSource = simdata;
```

Create a figure object for the animation object.

```
h.show();
```



Set up recording properties.

```
h.VideoRecord = 'on';
h.VideoQuality = 50;
h.VideoCompression = 'Motion JPEG AVI'
```

```
h = 
  Animation with properties:

                      Name: ''
                    Figure: [1×1 Figure]
    FigureCustomizationFcn: []
                    Bodies: {[1×1 Aero.Body]}
                    Camera: [1×1 Aero.Camera]
```

```
           TimeScaling: 5
                TStart: NaN
                TFinal: NaN
              TCurrent: 0
       FramesPerSecond: 10
           VideoRecord: 'on'
         VideoFileName: 'temp'
      VideoCompression: 'Motion JPEG AVI'
          VideoQuality: 50
           VideoTStart: NaN
           VideoTFinal: NaN
```

```
h.VideoFilename = 'astMotion_JPEG';
```

Play the animation.

```
h.play();
```

Verify that a file named `astMotion_JPEG.avi` was created in the current folder.

Disable recording to preserve the file.

```
h.VideoRecord = 'off';
```

Record Animation for Four Seconds

Simulate flight data for four seconds. Create an animation object.

```
h = Aero.Animation;
```

Control the frame display rate.

```
h.FramesPerSecond = 10;
```

Configure the animation object to set the seconds of animation data per second time-scaling (`TimeScaling`) property.

```
h.TimeScaling = 5;
```

The combination of `FramesPerSecond` and `TimeScaling` properties determines the time step of the simulation (`TimeScaling`/|FramesPerSecond|). These settings result in a time step of approximately 0.5 s.

Create and load a body for the animation object.

```
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
```

Load simulated flight trajectory data (`simdata`), located in *`matlabroot`*`\toolbox\aero\astdemos`.

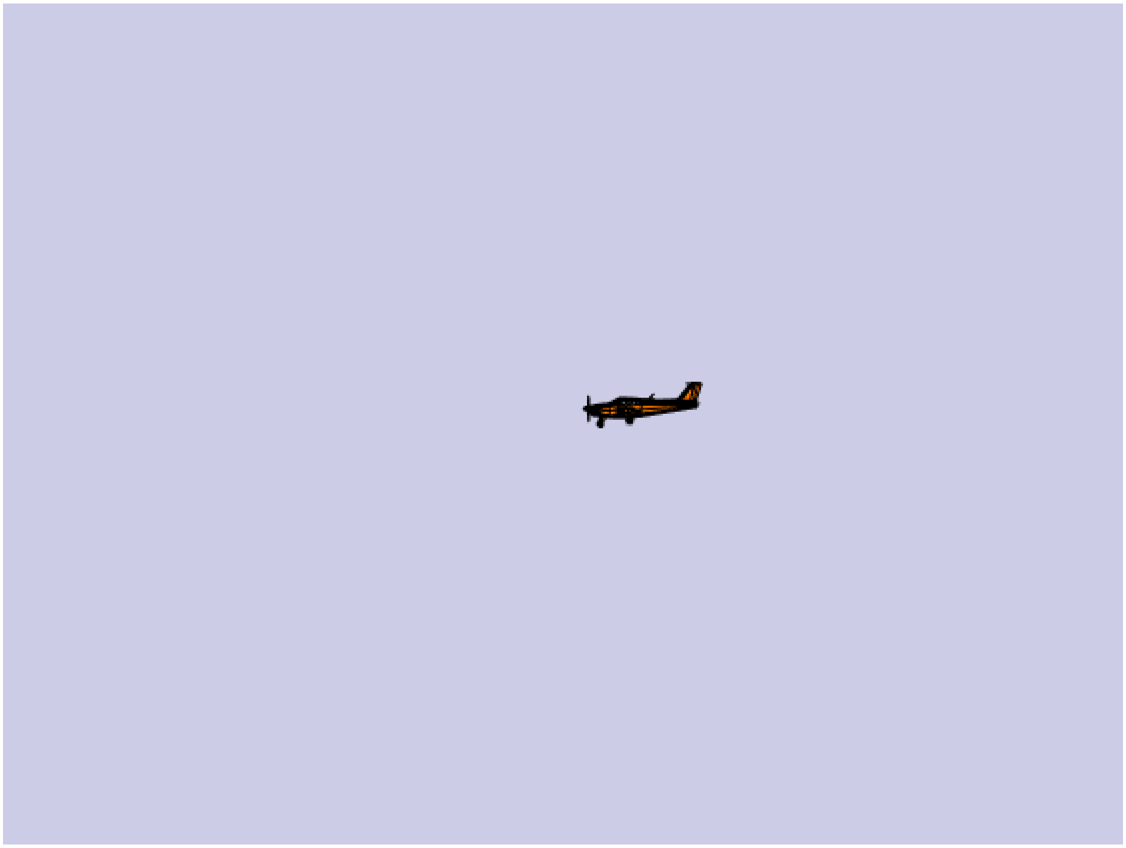
```
load simdata;
```

Set the time series data for the body.

```
h.Bodies{1}.TimeSeriesSource = simdata;
```

Create a figure object for the animation object.

```
h.show();
```



Set up recording properties.

```
h.VideoRecord='on';
h.VideoQuality = 50;
h.VideoCompression = 'Motion JPEG AVI';
h.VideoFilename = 'astMotion_JPEG';
```

Play the animation from `TFinal` to `TStart`.

```
h.TSTart = 1;
h.TFinal = 5;
h.play();
```

Verify that a file named `astMotion_JPEG.avi` was created in the current folder. When you rerun the recording, notice that the play time is shorter than that in the previous example when you record for the length of the simulation time.

Disable recording to preserve the file.

```
h.VideoRecord = 'off';
```

Schedule Three Second Recording of Simulation

Schedule three second recording of animation object simulation.

Create an animation object.

```
h = Aero.Animation;
```

Control the frame display rate.

```
h.FramesPerSecond = 10;
```

Configure the animation object to set the seconds of animation data per second time-scaling (`TimeScaling`) property.

```
h.TimeScaling = 5;
```

The combination of `FramesPerSecond` and `TimeScaling` properties determines the time step of the simulation (`TimeScaling`/|`FramesPerSecond`|). These settings result in a time step of approximately 0.5 s.

Create and load a body for the animation object.

```
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
```

Load simulated flight trajectory data (`simdata`), located in `matlabroot\toolbox\aero\astdemos`.

```
load simdata;
```

Set the time series data for the body.

```
h.Bodies{1}.TimeSeriesSource = simdata;
```

Create a figure object for the animation object.

```
h.show();
```

VideoTFinal property

Class: `Aero.Animation`
Package: `Aero`

Video recording stop time for scheduled recording

Values

Value between `TStart` and `TFinal`

Data type: double

Default: `NaN`, which uses the value of `TFinal`

Description

This property specifies the stop time of scheduled recording.

Use when `VideoRecord` is set to `'scheduled'`. Use `VideoTStart` to set the start time of the recording.

VideoTStart property

Class: `Aero.Animation`
Package: `Aero`

Video recording start time for scheduled recording

Values

Value between `TStart` and `TFinal`

Data type: double

Default: `NaN`, which uses the value of `TStart`.

Description

This property specifies the start time of the scheduled recording.

Use when `VideoRecord` is set to `'scheduled'`. Use `VideoTFinal` to set the end time of the recording.

Aerospace Toolbox Examples

Importing from USAF Digital DATCOM Files

This example shows how to bring United States Air Force (USAF) Digital DATCOM files into the MATLAB® environment using the Aerospace Toolbox™ software.

Example USAF Digital DATCOM File

Here's a sample input file for USAF Digital DATCOM for a wing-body-horizontal tail-vertical tail configuration running over 5 alphas, 2 Mach numbers, and 2 altitudes and calculating static and dynamic derivatives:

```
type astdatcom.in
```

```
 $FLTCON NMACH=2.0,MACH(1)=0.1,0.2$
 $FLTCON NALT=2.0,ALT(1)=5000.0,8000.0$
 $FLTCON NALPHA=5.,ALSCHD(1)=-2.0,0.0,2.0,
  ALSCHD(4)=4.0,8.0,LOOP=2.0$
 $OPTINS SREF=225.8,CBARR=5.75,BLREF=41.15$
 $SYNTHS XCG=7.08,ZCG=0.0,XW=6.1,ZW=-1.4,ALIW=1.1,XH=20.2,
   ZH=0.4,ALIH=0.0,XV=21.3,ZV=0.0,VERTUP=.TRUE.$
 $BODY NX=10.0,
   X(1)=-4.9,0.0,3.0,6.1,9.1,13.3,20.2,23.5,25.9,
   R(1)=0.0,1.0,1.75,2.6,2.6,2.6,2.0,1.0,0.0$
 $WGPLNF CHRDTP=4.0,SSPNE=18.7,SSPN=20.6,CHRDR=7.2,SAVSI=0.0,CHSTAT=0.25,
   TWISTA=-1.1,SSPNDD=0.0,DHDADI=3.0,DHDADO=3.0,TYPE=1.0$
NACA-W-6-64A412
 $HTPLNF CHRDTP=2.3,SSPNE=5.7,SSPN=6.625,CHRDR=0.25,SAVSI=11.0,
   CHSTAT=1.0,TWISTA=0.0,TYPE=1.0$
NACA-H-4-0012
 $VTPLNF CHRDTP=2.7,SSPNE=5.0,SSPN=5.2,CHRDR=5.3,SAVSI=31.3,
   CHSTAT=0.25,TWISTA=0.0,TYPE=1.0$
NACA-V-4-0012
CASEID SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG
DAMP
NEXT CASE
```

Here's the output file generated by USAF Digital DATCOM for the same wing-body-horizontal tail-vertical tail configuration running over 5 alphas, 2 Mach numbers, and 2 altitudes:

```
type astdatcom.out
```

```
 THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION
 IS RELEASED "AS IS".  THE U.S. GOVERNMENT MAKES NO
 WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, CONCERNING
 THIS SOFTWARE AND ANY ACCOMPANYING DOCUMENTATION,
 INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF
 MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
 IN NO EVENT WILL THE U.S. GOVERNMENT BE LIABLE FOR ANY
 DAMAGES, INCLUDING LOST PROFITS, LOST SAVINGS OR OTHER
 INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
 USE, OR INABILITY TO USE, THIS SOFTWARE OR ANY
 ACCOMPANYING DOCUMENTATION, EVEN IF INFORMED IN ADVANCE
 OF THE POSSIBILITY OF SUCH DAMAGES.








                                       **********************************************
                                       *    USAF STABILITY AND CONTROL  DIGITAL DATCO
                                       *    PROGRAM REV. JAN 96   DIRECT INQUIRIES TO
                                       *   WRIGHT LABORATORY  (WL/FIGC)  ATTN: W. BLA
                                       *         WRIGHT PATTERSON AFB, OHIO  45433
                                       *    PHONE (513) 255-6764,   FAX (513) 258-405
                                       **********************************************
1                        CONERR - INPUT ERROR CHECKING
0 ERROR CODES - N* DENOTES THE NUMBER OF OCCURENCES OF EACH ERROR
0 A - UNKNOWN VARIABLE NAME
0 B - MISSING EQUAL SIGN FOLLOWING VARIABLE NAME
0 C - NON-ARRAY VARIABLE HAS AN ARRAY ELEMENT DESIGNATION - (N)
0 D - NON-ARRAY VARIABLE HAS MULTIPLE VALUES ASSIGNED
0 E - ASSIGNED VALUES EXCEED ARRAY DIMENSION
0 F - SYNTAX ERROR

0******************************  INPUT DATA CARDS  ******************************

  $FLTCON NMACH=2.0,MACH(1)=0.1,0.2$
  $FLTCON NALT=2.0,ALT(1)=5000.0,8000.0$
  $FLTCON NALPHA=5.,ALSCHD(1)=-2.0,0.0,2.0,
   ALSCHD(4)=4.0,8.0,LOOP=2.0$
  $OPTINS SREF=225.8,CBARR=5.75,BLREF=41.15$
  $SYNTHS XCG=7.08,ZCG=0.0,XW=6.1,ZW=-1.4,ALIW=1.1,XH=20.2,
    ZH=0.4,ALIH=0.0,XV=21.3,ZV=0.0,VERTUP=.TRUE.$
  $BODY NX=10.0,
    X(1)=-4.9,0.0,3.0,6.1,9.1,13.3,20.2,23.5,25.9,
    R(1)=0.0,1.0,1.75,2.6,2.6,2.6,2.0,1.0,0.0$
  $WGPLNF CHRDTP=4.0,SSPNE=18.7,SSPN=20.6,CHRDR=7.2,SAVSI=0.0,CHSTAT=0.25,
    TWISTA=-1.1,SSPNDD=0.0,DHDADI=3.0,DHDADO=3.0,TYPE=1.0$
 NACA-W-6-64A412
  $HTPLNF CHRDTP=2.3,SSPNE=5.7,SSPN=6.625,CHRDR=0.25,SAVSI=11.0,
    CHSTAT=1.0,TWISTA=0.0,TYPE=1.0$
 NACA-H-4-0012
  $VTPLNF CHRDTP=2.7,SSPNE=5.0,SSPN=5.2,CHRDR=5.3,SAVSI=31.3,
    CHSTAT=0.25,TWISTA=0.0,TYPE=1.0$
 NACA-V-4-0012
 CASEID SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG
 DAMP
 NEXT CASE
1          THE FOLLOWING IS A LIST OF ALL INPUT CARDS FOR THIS CASE.
0
  $FLTCON NMACH=2.0,MACH(1)=0.1,0.2$
  $FLTCON NALT=2.0,ALT(1)=5000.0,8000.0$
  $FLTCON NALPHA=5.,ALSCHD(1)=-2.0,0.0,2.0,
   ALSCHD(4)=4.0,8.0,LOOP=2.0$
  $OPTINS SREF=225.8,CBARR=5.75,BLREF=41.15$
  $SYNTHS XCG=7.08,ZCG=0.0,XW=6.1,ZW=-1.4,ALIW=1.1,XH=20.2,
    ZH=0.4,ALIH=0.0,XV=21.3,ZV=0.0,VERTUP=.TRUE.$
  $BODY NX=10.0,
    X(1)=-4.9,0.0,3.0,6.1,9.1,13.3,20.2,23.5,25.9,
    R(1)=0.0,1.0,1.75,2.6,2.6,2.6,2.0,1.0,0.0$
  $WGPLNF CHRDTP=4.0,SSPNE=18.7,SSPN=20.6,CHRDR=7.2,SAVSI=0.0,CHSTAT=0.25,
    TWISTA=-1.1,SSPNDD=0.0,DHDADI=3.0,DHDADO=3.0,TYPE=1.0$
 NACA-W-6-64A412
  $HTPLNF CHRDTP=2.3,SSPNE=5.7,SSPN=6.625,CHRDR=0.25,SAVSI=11.0,
    CHSTAT=1.0,TWISTA=0.0,TYPE=1.0$
 NACA-H-4-0012
  $VTPLNF CHRDTP=2.7,SSPNE=5.0,SSPN=5.2,CHRDR=5.3,SAVSI=31.3,
    CHSTAT=0.25,TWISTA=0.0,TYPE=1.0$
 NACA-V-4-0012
 CASEID SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG
 DAMP
 NEXT CASE
0 INPUT DIMENSIONS ARE IN FT, SCALE FACTOR IS 1.0000

1                             AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VE
                                                          WING SECTION DEFINITION
0                                 IDEAL ANGLE OF ATTACK =   0.00000 DEG.
```

```
                                ZERO LIFT ANGLE OF ATTACK =  -3.09292 DEG.

                                   IDEAL LIFT COEFFICIENT =   0.40000

                     ZERO LIFT PITCHING MOMENT COEFFICIENT =  -0.08719

                               MACH ZERO LIFT-CURVE-SLOPE =   0.09654 /DEG.

                                    LEADING EDGE RADIUS =   0.00993 FRACTION CHORD

                                MAXIMUM AIRFOIL THICKNESS =   0.12000 FRACTION CHORD

                                                  DELTA-Y =   2.46808 PERCENT CHORD


0                        MACH= 0.1000 LIFT-CURVE-SLOPE =   0.09693 /DEG.      XAC =
0                        MACH= 0.2000 LIFT-CURVE-SLOPE =   0.09811 /DEG.      XAC =
1                            AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VE
                                                 HORIZONTAL TAIL SECTION DEFINITION
0                                  IDEAL ANGLE OF ATTACK =   0.00000 DEG.

                                ZERO LIFT ANGLE OF ATTACK =   0.00000 DEG.

                                   IDEAL LIFT COEFFICIENT =   0.00000

                     ZERO LIFT PITCHING MOMENT COEFFICIENT =   0.00000

                               MACH ZERO LIFT-CURVE-SLOPE =   0.09596 /DEG.

                                    LEADING EDGE RADIUS =   0.01587 FRACTION CHORD

                                MAXIMUM AIRFOIL THICKNESS =   0.12000 FRACTION CHORD

                                                  DELTA-Y =   3.16898 PERCENT CHORD


0                        MACH= 0.1000 LIFT-CURVE-SLOPE =   0.09636 /DEG.      XAC =
0                        MACH= 0.2000 LIFT-CURVE-SLOPE =   0.09761 /DEG.      XAC =
1                            AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976 VE
                                                  VERTICAL TAIL SECTION DEFINITION
0                                  IDEAL ANGLE OF ATTACK =   0.00000 DEG.

                                ZERO LIFT ANGLE OF ATTACK =   0.00000 DEG.
```

```
                                   IDEAL LIFT COEFFICIENT =   0.00000

                    ZERO LIFT PITCHING MOMENT COEFFICIENT =   0.00000

                               MACH ZERO LIFT-CURVE-SLOPE =   0.09596 /DEG.

                                    LEADING EDGE RADIUS =   0.01587 FRACTION CHORD

                               MAXIMUM AIRFOIL THICKNESS =   0.12000 FRACTION CHORD

                                                DELTA-Y =   3.16898 PERCENT CHORD


0                        MACH= 0.1000 LIFT-CURVE-SLOPE =   0.09636 /DEG.      XAC =
0                        MACH= 0.2000 LIFT-CURVE-SLOPE =   0.09761 /DEG.      XAC =
1                              AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                        CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDE
                                       WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
                               SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

 -----------------------  FLIGHT CONDITIONS  -----------------------           -------
  MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS            REF.
 NUMBER                                                      NUMBER              AREA
              FT       FT/SEC     LB/FT**2       DEG R         1/FT              FT**2
0 0.100    5000.00     109.70   1.7609E+03     500.843     6.1507E+05          225.800
0                                                                 -------------------DERI
0 ALPHA     CD       CL       CM       CN       CA       XCP       CLA         CMA
0
   -2.0    0.032    0.113   -0.0340   0.112    0.035   -0.304    8.926E-02   -2.105E-02
    0.0    0.035    0.296   -0.0752   0.296    0.035   -0.254    9.350E-02   -2.034E-02
    2.0    0.042    0.487   -0.1153   0.488    0.025   -0.236    9.732E-02   -1.971E-02
    4.0    0.052    0.685   -0.1541   0.687    0.004   -0.224    1.005E-01   -1.927E-02
    8.0    0.084    1.098   -0.2304   1.099   -0.069   -0.210    1.059E-01   -1.890E-02
0                                  ALPHA     Q/QINF    EPSLON  D(EPSLON)/D(ALPHA)
0
                                   -2.0      1.000      0.953       0.571
                                    0.0      1.000      2.094       0.583
                                    2.0      1.000      3.284       0.606
                                    4.0      1.000      4.520       0.610
                                    8.0      1.000      6.897       0.594
1                              AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                                 DYNAMIC DERIVATIVES
                                       WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
```

```
                                SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

  -----------------------  FLIGHT CONDITIONS  ------------------------           -------
   MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.
  NUMBER                                                       NUMBER              AREA
              FT       FT/SEC     LB/FT**2       DEG R          1/FT               FT**2
0 0.100    5000.00     109.70   1.7609E+03     500.843     6.1507E+05          225.800
                                                                   DYNAMIC DERIVATIVES (PER DEGREE)
0           -------PITCHING-------    -----ACCELERATION------    --------------ROLLING-
0   ALPHA       CLQ          CMQ           CLAD         CMAD         CLP          CYP
0
    -2.00    9.739E-02   -8.918E-02     1.874E-02   -4.247E-02   -7.824E-03   -1.516E-0
     0.00                               1.913E-02   -4.336E-02   -8.226E-03   -1.649E-0
     2.00                               1.991E-02   -4.512E-02   -8.599E-03   -1.792E-0
     4.00                               2.003E-02   -4.540E-02   -8.890E-03   -1.942E-0
     8.00                               1.952E-02   -4.424E-02   -9.387E-03   -2.262E-0
1                               AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                         CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDE
                                        WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
                                SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

  -----------------------  FLIGHT CONDITIONS  ------------------------           -------
   MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.
  NUMBER                                                       NUMBER              AREA
              FT       FT/SEC     LB/FT**2       DEG R          1/FT               FT**2
0 0.200    5000.00     219.39   1.7609E+03     500.843     1.2301E+06          225.800
0                                                                       -------------------DERI
0 ALPHA     CD       CL       CM       CN       CA       XCP        CLA         CMA
0
   -2.0    0.028    0.114   -0.0335   0.113    0.032   -0.297    9.000E-02   -2.124E-02
    0.0    0.031    0.298   -0.0751   0.298    0.031   -0.252    9.421E-02   -2.051E-02
    2.0    0.038    0.491   -0.1155   0.492    0.021   -0.235    9.800E-02   -1.987E-02
    4.0    0.048    0.690   -0.1546   0.692    0.000   -0.223    1.011E-01   -1.943E-02
    8.0    0.081    1.105   -0.2316   1.106   -0.074   -0.209    1.065E-01   -1.906E-02
0                                    ALPHA     Q/QINF    EPSLON  D(EPSLON)/D(ALPHA)
0
                                     -2.0      1.000      0.957        0.573
                                      0.0      1.000      2.103        0.585
                                      2.0      1.000      3.297        0.609
                                      4.0      1.000      4.537        0.612
                                      8.0      1.000      6.923        0.596
1                               AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                                     DYNAMIC DERIVATIVES
                                        WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
                                SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

  ----------------------- FLIGHT CONDITIONS  ------------------------           -------
   MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.
  NUMBER                                                       NUMBER              AREA
             FT        FT/SEC     LB/FT**2       DEG R         1/FT                FT**2
 0 0.200    5000.00     219.39   1.7609E+03     500.843     1.2301E+06          225.800
                                                          DYNAMIC DERIVATIVES (PER DEGREE)
 0           -------PITCHING-------    -----ACCELERATION------    --------------ROLLING-
 0   ALPHA       CLQ          CMQ           CLAD         CMAD         CLP          CYP
 0
    -2.00    9.840E-02   -8.993E-02     1.900E-02   -4.307E-02   -7.877E-03   -1.525E-0
     0.00                               1.940E-02   -4.398E-02   -8.276E-03   -1.659E-0
     2.00                               2.018E-02   -4.574E-02   -8.646E-03   -1.802E-0
     4.00                               2.030E-02   -4.602E-02   -8.934E-03   -1.953E-0
     8.00                               1.978E-02   -4.483E-02   -9.423E-03   -2.273E-0
 1                               AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                          CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDE
                                         WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
                                SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

  ----------------------- FLIGHT CONDITIONS  ------------------------           -------
   MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS             REF.
  NUMBER                                                       NUMBER              AREA
             FT        FT/SEC     LB/FT**2       DEG R         1/FT                FT**2
 0 0.100    8000.00     108.52   1.5721E+03     490.151     5.6457E+05          225.800
 0                                                                    -------------------DERI
 0 ALPHA     CD       CL       CM       CN       CA       XCP       CLA         CMA
 0
   -2.0    0.032    0.113   -0.0340   0.112    0.036   -0.305    8.926E-02   -2.106E-02
    0.0    0.035    0.296   -0.0753   0.296    0.035   -0.254    9.350E-02   -2.034E-02
    2.0    0.042    0.487   -0.1154   0.488    0.025   -0.236    9.732E-02   -1.971E-02
    4.0    0.052    0.685   -0.1541   0.687    0.004   -0.224    1.005E-01   -1.927E-02
    8.0    0.085    1.098   -0.2304   1.099   -0.069   -0.210    1.059E-01   -1.891E-02
 0                                    ALPHA     Q/QINF    EPSLON  D(EPSLON)/D(ALPHA)
 0
                                      -2.0      1.000      0.953       0.571
                                       0.0      1.000      2.094       0.583
                                       2.0      1.000      3.284       0.606
                                       4.0      1.000      4.520       0.610
                                       8.0      1.000      6.897       0.594
 1                               AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                                     DYNAMIC DERIVATIVES
                                         WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
```

```
                               SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

  -----------------------  FLIGHT CONDITIONS  ------------------------           -------
   MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS            REF.
  NUMBER                                                       NUMBER             AREA
              FT       FT/SEC     LB/FT**2       DEG R         1/FT               FT**2
 0 0.100    8000.00     108.52   1.5721E+03     490.151     5.6457E+05          225.800
                                                   DYNAMIC DERIVATIVES (PER DEGREE)
 0          -------PITCHING-------    -----ACCELERATION------    --------------ROLLING-
 0   ALPHA       CLQ          CMQ           CLAD         CMAD         CLP          CYP
 0
    -2.00    9.739E-02   -8.918E-02     1.874E-02   -4.247E-02   -7.824E-03   -1.516E-0
     0.00                               1.913E-02   -4.336E-02   -8.226E-03   -1.649E-0
     2.00                               1.991E-02   -4.512E-02   -8.599E-03   -1.792E-0
     4.00                               2.003E-02   -4.540E-02   -8.890E-03   -1.942E-0
     8.00                               1.952E-02   -4.424E-02   -9.387E-03   -2.262E-0
 1                              AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                         CHARACTERISTICS AT ANGLE OF ATTACK AND IN SIDE
                                        WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
                               SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

  -----------------------  FLIGHT CONDITIONS  ------------------------           -------
   MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS            REF.
  NUMBER                                                       NUMBER             AREA
              FT       FT/SEC     LB/FT**2       DEG R         1/FT               FT**2
 0 0.200    8000.00     217.04   1.5721E+03     490.151     1.1291E+06          225.800
 0                                                                 -------------------DERI
 0 ALPHA     CD       CL       CM        CN       CA       XCP        CLA         CMA
 0
   -2.0    0.028    0.114   -0.0335   0.113    0.032   -0.297    9.000E-02   -2.124E-02
    0.0    0.031    0.298   -0.0751   0.298    0.031   -0.252    9.421E-02   -2.051E-02
    2.0    0.038    0.491   -0.1156   0.492    0.021   -0.235    9.800E-02   -1.987E-02
    4.0    0.049    0.690   -0.1546   0.692    0.000   -0.223    1.011E-01   -1.943E-02
    8.0    0.081    1.105   -0.2316   1.106   -0.073   -0.209    1.065E-01   -1.906E-02
 0                                ALPHA     Q/QINF    EPSLON  D(EPSLON)/D(ALPHA)
 0
                                  -2.0      1.000      0.957        0.573
                                   0.0      1.000      2.103        0.585
                                   2.0      1.000      3.297        0.609
                                   4.0      1.000      4.537        0.612
                                   8.0      1.000      6.923        0.596
 1                              AUTOMATED STABILITY AND CONTROL METHODS PER APRIL 1976
                                                      DYNAMIC DERIVATIVES
                                        WING-BODY-VERTICAL TAIL-HORIZONTAL TAIL CONFIGU
```

```
                                SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG

  -----------------------  FLIGHT CONDITIONS  ------------------------           -------
   MACH    ALTITUDE   VELOCITY    PRESSURE    TEMPERATURE     REYNOLDS            REF.
  NUMBER                                                       NUMBER             AREA
              FT       FT/SEC     LB/FT**2       DEG R          1/FT              FT**2
 0 0.200    8000.00     217.04   1.5721E+03     490.151     1.1291E+06           225.800
                                                       DYNAMIC DERIVATIVES (PER DEGREE)
 0           -------PITCHING-------    -----ACCELERATION------    --------------ROLLING-
 0   ALPHA       CLQ          CMQ           CLAD         CMAD         CLP          CYP
 0
    -2.00    9.840E-02   -8.993E-02     1.900E-02   -4.307E-02   -7.877E-03   -1.525E-0
     0.00                               1.940E-02   -4.398E-02   -8.276E-03   -1.659E-0
     2.00                               2.018E-02   -4.574E-02   -8.646E-03   -1.802E-0
     4.00                               2.030E-02   -4.602E-02   -8.934E-03   -1.953E-0
     8.00                               1.978E-02   -4.483E-02   -9.424E-03   -2.273E-0
 1          THE FOLLOWING IS A LIST OF ALL INPUT CARDS FOR THIS CASE.
 0
 1 END OF JOB.
```

Import Data from DATCOM Files

Use the **datcomimport** function to bring the Digital DATCOM data into MATLAB.

```
alldata = datcomimport('astdatcom.out', true, 0);
```

Examining Imported DATCOM Data

The **datcomimport** function creates a cell array of structures containing the data from the Digital DATCOM output file.

```
data = alldata{1}
```

```
data = 

  struct with fields:

        case: 'SKYHOGG BODY-WING-HORIZONTAL TAIL-VERTICAL TAIL CONFIG'
        mach: [0.1000 0.2000]
         alt: [5000 8000]
       alpha: [-2 0 2 4 8]
       nmach: 2
        nalt: 2
      nalpha: 5
       rnnub: []
      hypers: 0
        loop: 2
        sref: 225.8000
        cbar: 5.7500
       blref: 41.1500
         dim: 'ft'
       deriv: 'deg'
      stmach: 0.6000
      tsmach: 1.4000
        save: 0
       stype: []
        trim: 0
        damp: 1
       build: 1
        part: 0
     highsym: 0
     highasy: 0
     highcon: 0
        tjet: 0
      hypeff: 0
          lb: 0
         pwr: 0
        grnd: 0
       wsspn: 18.7000
       hsspn: 5.7000
      ndelta: 0
       delta: []
      deltal: []
      deltar: []
         ngh: 0
      grndht: []
      config: [1x1 struct]
     version: 1976
          cd: [5x2x2 double]
          cl: [5x2x2 double]
          cm: [5x2x2 double]
          cn: [5x2x2 double]
          ca: [5x2x2 double]
         xcp: [5x2x2 double]
         cma: [5x2x2 double]
         cyb: [5x2x2 double]
         cnb: [5x2x2 double]
         clb: [5x2x2 double]
```

```
             cla: [5x2x2 double]
           qqinf: [5x2x2 double]
             eps: [5x2x2 double]
        depsdalp: [5x2x2 double]
             clq: [5x2x2 double]
             cmq: [5x2x2 double]
            clad: [5x2x2 double]
            cmad: [5x2x2 double]
             clp: [5x2x2 double]
             cyp: [5x2x2 double]
             cnp: [5x2x2 double]
             cnr: [5x2x2 double]
             clr: [5x2x2 double]
```

Filling in Missing DATCOM Data

By default, missing data points are set to 99999 and data points are set to NaN where no DATCOM methods exist or where the method is not applicable.

It can be seen in the Digital DATCOM output file and examining the imported data that

$$C_{Y\beta},\ C_{n\beta},\ C_{Lq},\ \text{and}\ C_{mq}$$

have data only in the first alpha value. Here are the imported data values.

```
data.cyb
```

```
ans(:,:,1) =

   1.0e+04 *

   -0.0000   -0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999


ans(:,:,2) =

   1.0e+04 *

   -0.0000   -0.0000
```

```
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
```

```
data.cnb
```

```
ans(:,:,1) =

   1.0e+04 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999


ans(:,:,2) =

   1.0e+04 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
```

```
data.clq
```

```
ans(:,:,1) =

   1.0e+04 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
```

```
ans(:,:,2) =

   1.0e+04 *

    0.0000    0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
```

```
data.cmq
```

```
ans(:,:,1) =

   1.0e+04 *

   -0.0000   -0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999


ans(:,:,2) =

   1.0e+04 *

   -0.0000   -0.0000
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
    9.9999    9.9999
```

The missing data points will be filled with the values for the first alpha, since these data points are meant to be used for all alpha values.

```
aerotab = {'cyb' 'cnb' 'clq' 'cmq'};

for k = 1:length(aerotab)
    for m = 1:data.nmach
        for h = 1:data.nalt
            data.(aerotab{k})(:,m,h) = data.(aerotab{k})(1,m,h);
        end
    end
end
```

Here are the updated imported data values:

```
data.cyb


ans(:,:,1) =

   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035


ans(:,:,2) =

   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035
   -0.0035   -0.0035


data.cnb


ans(:,:,1) =

   1.0e-03 *

    0.9142    0.8781
    0.9142    0.8781
    0.9142    0.8781
    0.9142    0.8781
    0.9142    0.8781


ans(:,:,2) =

   1.0e-03 *
```

```
    0.9190    0.8829
    0.9190    0.8829
    0.9190    0.8829
    0.9190    0.8829
    0.9190    0.8829
```

```
data.clq
```

```
ans(:,:,1) =

    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984


ans(:,:,2) =

    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
    0.0974    0.0984
```

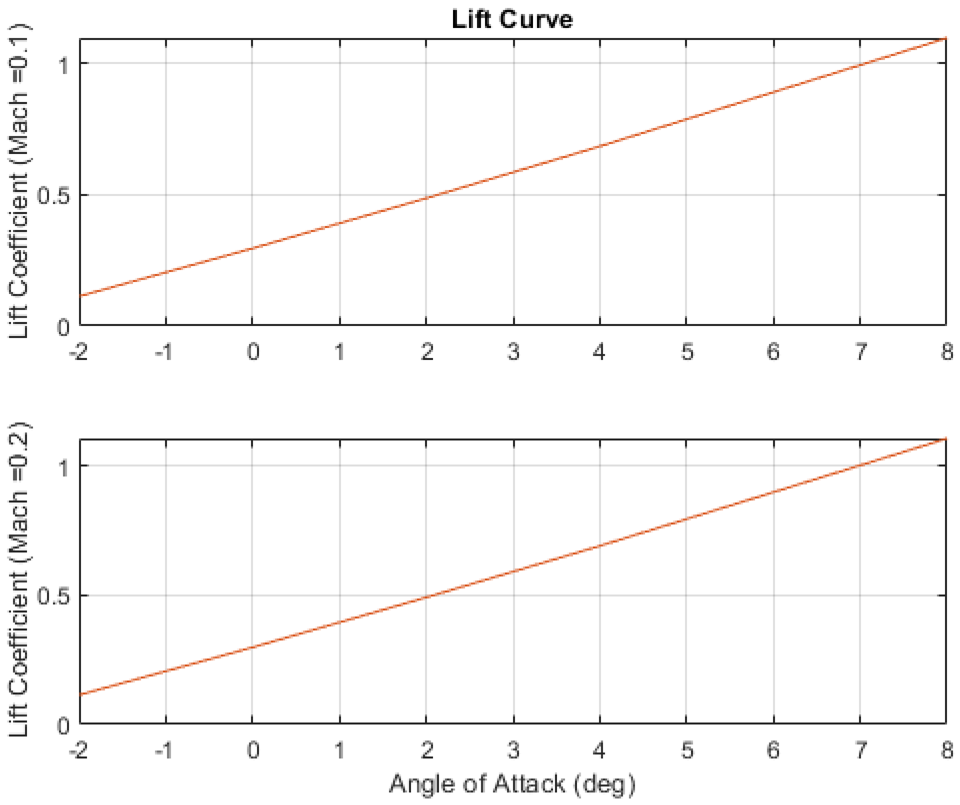
```
data.cmq
```

```
ans(:,:,1) =

   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899


ans(:,:,2) =

   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
   -0.0892   -0.0899
```

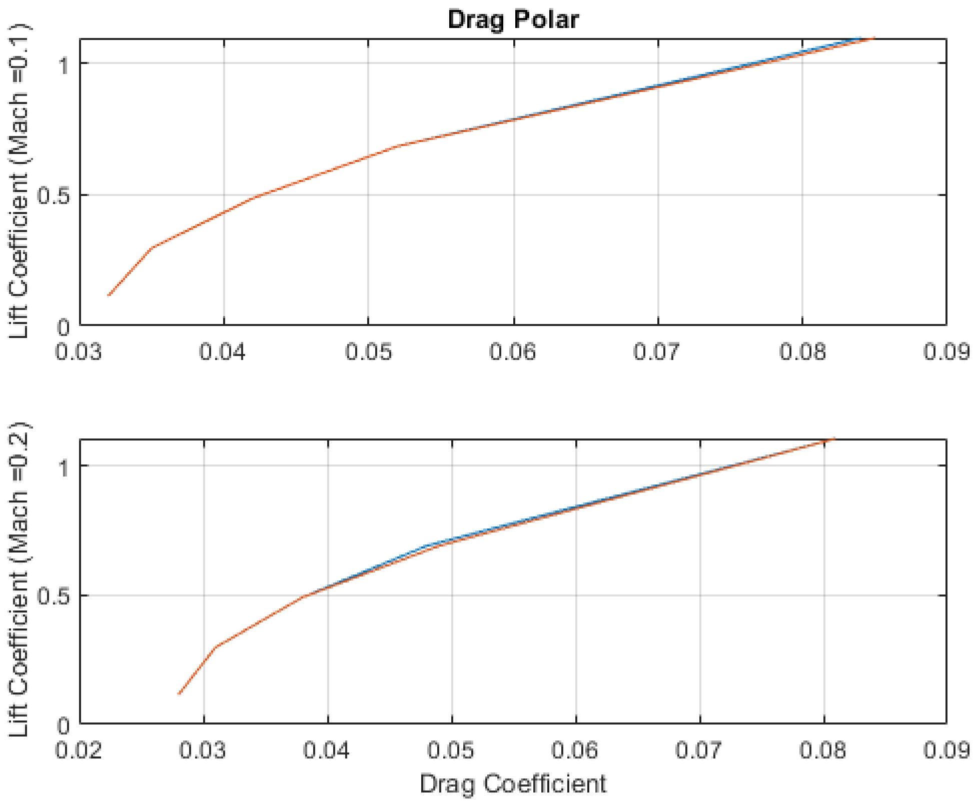
Plotting Aerodynamic Coefficients

Plot lift curve, drag polar and pitching moments.

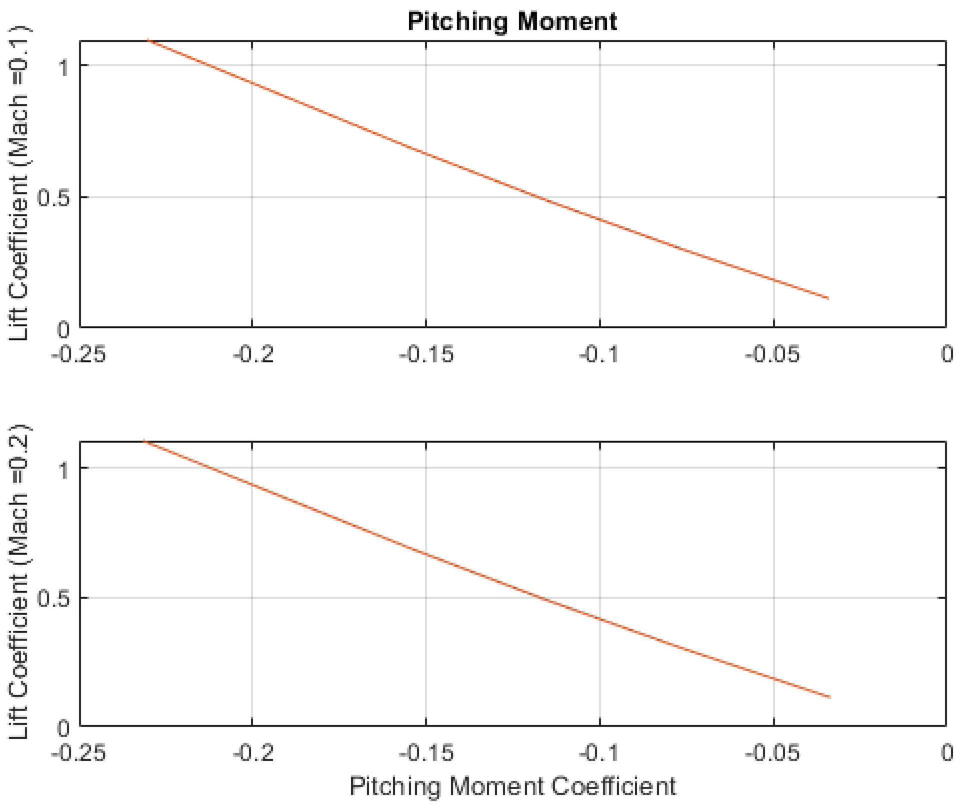
```
h1 = figure;
figtitle = {'Lift Curve' ''};
for k=1:2
    subplot(2,1,k)
    plot(data.alpha,permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach =' num2str(data.mach(k)) ')'])
    title(figtitle{k});
end
xlabel('Angle of Attack (deg)')
```



```
h2 = figure;
figtitle = {'Drag Polar' ''};
for k=1:2
    subplot(2,1,k)
    plot(permute(data.cd(:,k,:),[1 3 2]),permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach =' num2str(data.mach(k)) ')'])
    title(figtitle{k})
end
xlabel('Drag Coefficient')
```



```
h3 = figure;
figtitle = {'Pitching Moment' ''};
for k=1:2
    subplot(2,1,k)
    plot(permute(data.cm(:,k,:),[1 3 2]),permute(data.cl(:,k,:),[1 3 2]))
    grid
    ylabel(['Lift Coefficient (Mach =' num2str(data.mach(k)) ')'])
    title(figtitle{k})
end
xlabel('Pitching Moment Coefficient')
```



```
close(h1,h2,h3);
%#ok<*NOPTS>
```

Create a Flight Animation from Trajectory Data

This example shows how to create a flight animation for a trajectory using a FlightGear Animation object.

Note: When running this example within the product, you must customize the example with your FlightGear installation and uncomment the GenerateRunScript, system and play commands. You must also copy the $MATLAB/toolbox/aero/astdemos/HL20 folder into the $FLIGHTGEAR/data/Aircraft/ folder.

Load Recorded Flight Trajectory Data

The flight trajectory data for this example is stored in a comma separated value formatted file. Use **dlmread** to read the data from the file starting at row 1 and column 0 skipping the header information.

```
tdata = dlmread('asthl20log.csv',',',1,0);
```

Create a Time Series Object from Trajectory Data

Use the MATLAB® **timeseries** command to create the time series object, ts, from the latitude, longitude, altitude, and Euler angle data along with the time array in tdata. To convert the latitude, longitude, and Euler angles from degrees to radians use the **convang** function.

```
ts = timeseries([convang(tdata(:,[3 2]),'deg','rad') ...
                 tdata(:,4) convang(tdata(:,5:7),'deg','rad')],tdata(:,1));
```

You can create imported data from this data using other valid formats, such as 'Array6DoF', For example:

ts = [tdata(:,1) convang(tdata(:,[3 2]),'deg','rad') tdata(:,4) ...
convang(tdata(:,5:7),'deg','rad')];

and 'Array3DoF'.

ts = [tdata(:,1) convang(tdata(:,3),'deg','rad') tdata(:,4) ... convang(tdata(:,6),'deg','rad')];

Use FlightGearAnimation Object to Initialize Flight Animation

Open a FlightGearAnimation object.

```
h = Aero.FlightGearAnimation;
```

Set FlightGearAnimation object properties for timeseries.

```
h.TimeseriesSourceType = 'Timeseries';
h.TimeseriesSource = ts;
```

Set FlightGearAnimation object properties about FlightGear

These properties include the path to the installation folder, the version number, the aircraft geometry model, and the network information for FlightGear flight simulator.

```
h.FlightGearBaseDirectory = 'C:\Program Files\FlightGear';
h.FlightGearVersion = '2018.3';
h.GeometryModelName = 'HL20';
h.DestinationIpAddress = '127.0.0.1';
h.DestinationPort = '5502';
```

Set the desired initial conditions (location and orientation) for FlightGear flight simulator.

```
h.AirportId = 'KSFO';
h.RunwayId = '10L';
h.InitialAltitude = 7224;
h.InitialHeading = 113;
h.OffsetDistance = 4.72;
h.OffsetAzimuth = 0;
```

Enable "just in time" scenery installation for FlightGear flight simulator. Required scenery will be downloaded while the simulator is running. For Windows® systems, you may encounter an error message while launching FlightGear with this option enabled. For more information, see `Installing Additional FlightGear Scenery`.

```
h.InstallScenery = true;
```

Disable FlightGear Shaders.

```
h.DisableShaders = true;
```

Set the seconds of animation data per second of wall-clock time.

```
h.TimeScaling = 5;
```

Use get(h) to check the FlightGearAnimation object properties and their values.

```
get(h)
```

```
           TimeseriesSource: [1x1 timeseries]
       TimeseriesSourceType: 'Timeseries'
```

```
          TimeseriesReadFcn: @TimeseriesRead
                TimeScaling: 5
             FramesPerSecond: 12
           FlightGearVersion: '2018.3'
              OutputFileName: 'runfg.bat'
     FlightGearBaseDirectory: 'C:\Program Files\FlightGear'
           GeometryModelName: 'HL20'
        DestinationIpAddress: '127.0.0.1'
             DestinationPort: '5502'
                   AirportId: 'KSFO'
                    RunwayId: '10L'
             InitialAltitude: 7224
              InitialHeading: 113
              OffsetDistance: 4.7200
               OffsetAzimuth: 0
              InstallScenery: 1
              DisableShaders: 1
                      TStart: NaN
                      TFinal: NaN
                Architecture: 'Default'
```

Create a Run Script to Launch FlightGear Flight Simulator

To start FlightGear with the desired initial conditions (location, date, time, weather, and operating modes), create a run script with the **GenerateRunScript** command. By default, **GenerateRunScript** saves the run script as a text file named 'runfg.bat'.

GenerateRunScript(h)

You do not need to generate this file each time the data is viewed. Generate it only when the desired initial conditions or FlightGear information changes.

Start FlightGear Flight Simulator

To start FlightGear from the MATLAB command prompt, type the **system** command to execute the run script created by **GenerateRunScript**.

system('runfg.bat &');

Tip: With the FlightGear window in focus, press the V key to alternate between the different aircraft views: cockpit view, helicopter view, and chase view.

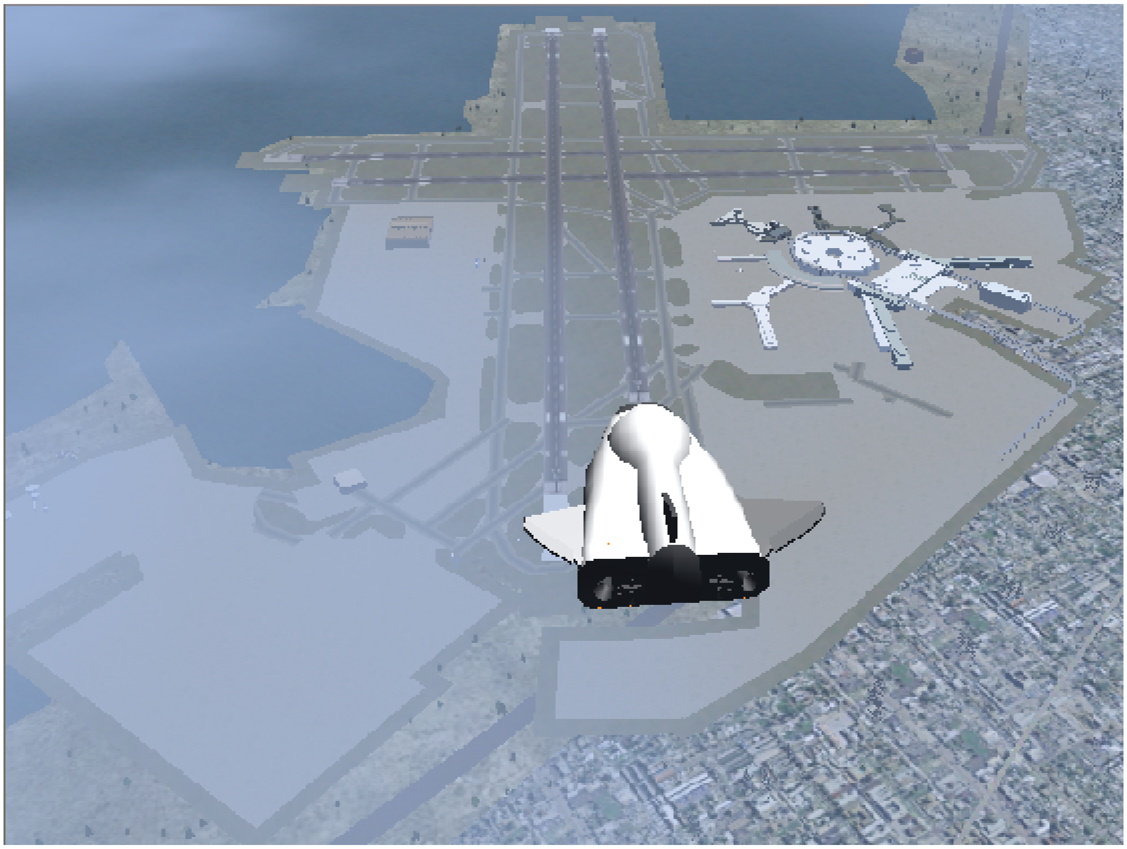
Play the Flight Animation of Trajectory Data

Once FlightGear is up and running, the FlightGearAnimation object can start to communicate with FlightGear. To display the flight animation with FlightGear, use the **play** command.

play(h)

To display a screenshot of the flight animation, use the MATLAB **image** command.

```
image(imread([matlabroot filesep fullfile('toolbox','aero','astdemos','figures','astfga
axis off;
set(gca,'Position',[ 0 0 1 1 ]);
set(gcf,'MenuBar','none');
```



Estimating G Forces for Flight Data

This example shows how to load flight data and estimate G forces during the flight.

Load Recorded Flight Data for Analysis

The recorded data contains the following flight parameters:

- angle of attack (alpha) in radians,
- sideslip angle (beta) in radians,
- indicated airspeed (IAS) in knots,
- body angular rates (omega) in radians/second,
- downrange and crossrange positions in feet, and
- altitude (alt) in feet.

```
load('astflight.mat');
```

Extract Flight Parameters from Loaded Data

MATLAB® variables are created for angle of attack (alpha), sideslip angle (beta), body angular rates (omega), and altitude (alt) from recorded data. The `convangvel` function is used to convert body angular rates from radians per second (rad/s) to degrees per second (deg/s).

```
alpha = fltdata(:,2);
beta  = fltdata(:,3);
omega = convangvel( fltdata(:,5:7), 'rad/s', 'deg/s' );
alt   = fltdata(:,10);
```

Compute True Airspeed from Indicated Airspeed

In this set of flight data, indicated airspeed (IAS) was recorded. Indicated airspeed (IAS) is displayed in the cockpit instrumentation. To perform calculations, true airspeed (TAS), the airspeed without measurement errors, is typically used.

Measurement errors are introduced through the pitot-static airspeed indicators used to determine airspeed. These measurement errors are *density error*, *compressibility error* and *calibration error*. Applying these errors to true airspeed results in indicated airspeed.

- *Density error* occurs due to lower air density at altitude. The effect is an airspeed indicator reads lower than true airspeed at higher altitudes. When the difference or error in air density at altitude from air density on a standard day at sea level is applied to true airspeed, it results in equivalent airspeed (EAS). Equivalent airspeed is true airspeed modified with the changes in atmospheric density which affect the airspeed indicator.

- *Compressibility error* occurs because air has a limited ability to resist compression. This ability is reduced by an increase in altitude, an increase in speed, or a restricted volume. Within the airspeed indicator, there is a certain amount of trapped air. When flying at high altitudes and higher airspeeds, calibrated airspeed (CAS) is always higher than equivalent airspeed. Calibrated airspeed is equivalent airspeed modified with compressibility effects of air which affect the airspeed indicator.
- *Calibration error* is specific to a given aircraft design. Calibration error is the result of the position and placement of the static vent(s) to maintain a pressure equal to atmospheric pressure inside the airspeed indicator. Position and placement of the static vent along with angle of attack and velocity of the aircraft will determine the pressure inside the airspeed indicator and thus the amount of calibration error of the airspeed indicator. A calibration table is usually given in the pilot operating handbook (POH) or in other aircraft specifications. Using this calibration table, the indicated airspeed (IAS) is determined from calibrated airspeed by modifying it with calibration error of the airspeed indicator.

The following data is the airspeed calibration table for the airspeed indicator of the aircraft with zero flap deflection. The airspeed calibration table converts indicated airspeed (IAS) to calibrated airspeed (CAS) by removing the calibration error.

```
flaps0IAS = 40:10:140;
flaps0CAS = [43 51 59 68 77 87 98 108 118 129 140];
```

The indicated airspeed (IAS) from the flight and airspeed calibration table are used to determine the calibrated airspeed (CAS) for the flight.

```
CAS = interp1( flaps0IAS, flaps0CAS, fltdata(:,4) );
```

The atmospheric properties, temperature (T), speed of sound (a), pressure (P), and density (rho), are determined at altitude for standard day using the `atmoscoesa` function.

```
[T, a, P, rho]= atmoscoesa( alt );
```

Once the calibrated airspeed (CAS) and the atmospheric properties are determined, the true airspeed (Vt) can be calculated using the `correctairspeed` function.

```
Vt = correctairspeed( CAS, a, P, 'CAS', 'TAS' );
```

Import Digital DATCOM Data for Aircraft

Use the `datcomimport` function to bring the Digital DATCOM data into MATLAB. The units for this aerodynamic information are feet and degrees.

```
data = datcomimport( 'astflight.out', true, 0 );
```

It can be seen in the Digital DATCOM output file and examining the imported data that

$$C_{Y\beta},\ C_{n\beta},\ C_{lq},\ \text{and}\ C_{mq}$$

have data only in the first alpha value. By default, missing data points are set to 99999. The missing data points are filled with the values for the first alpha, since these data points are meant to be used for all alpha values.

```
aerotab = {'cyb' 'cnb' 'clq' 'cmq'};

for k = 1:length(aerotab)
    for m = 1:data{1}.nmach
        for h = 1:data{1}.nalt
            data{1}.(aerotab{k})(:,m,h) = data{1}.(aerotab{k})(1,m,h);
        end
    end
end
```

Interpolate Stability and Dynamic Derivatives at Flight Conditions

The stability and dynamic derivatives in the digital DATCOM structure are 3-D tables which are functions of Mach number, angle of attack in degrees, and altitude in feet. In order to perform 3-D linear interpolation (`interp3`), the indices for the derivative tables are required to be monotonic and plaid. Indices of this form are generated by the `meshgrid` function.

```
[mnum, alp, h] = meshgrid( data{1}.mach, data{1}.alpha, data{1}.alt );
```

Since the angular units of the derivatives are in degrees, the units of angle of attack (alpha) are converted from radians from degrees by the function `convang`.

```
alphadeg = convang( alpha, 'rad', 'deg' );
```

The Mach numbers for the flight are calculated by the function `machnumber` using speed of sound (a) and airspeed (Vt).

```
Mach = machnumber( convvel( [Vt zeros(size(Vt,1),2)], 'kts', 'm/s' ), a );
```

Looping through the flight conditions, allows `interp3` to be used to linearly interpolate derivative tables to find static and dynamic derivatives at those flight conditions.

```
for k = length(alt):-1:1
    cd(k,:)   = interp3( mnum, alp, h, data{1}.cd,   Mach(k), alphadeg(k), alt(k), 'lin
    cyb(k,:)  = interp3( mnum, alp, h, data{1}.cyb,  Mach(k), alphadeg(k), alt(k), 'lin
    cl(k,:)   = interp3( mnum, alp, h, data{1}.cl,   Mach(k), alphadeg(k), alt(k), 'lin
    cyp(k,:)  = interp3( mnum, alp, h, data{1}.cyp,  Mach(k), alphadeg(k), alt(k), 'lin
    clad(k,:) = interp3( mnum, alp, h, data{1}.clad, Mach(k), alphadeg(k), alt(k), 'lin
end
```

Compute Aerodynamic Coefficients

Once the derivatives are found for the flight conditions, aerodynamic coefficients can be calculated.

Reference lengths and areas used in the aerodynamic coefficient computation are extracted from the digital DATCOM structure.

```
cbar = data{1}.cbar;
Sref = data{1}.sref;
bref = data{1}.blref;
```

The angular units of the derivatives are in degrees, so the units of sideslip angle (beta) are converted from radians from degrees by the function `convang`.

```
betadeg  = convang( beta,  'rad', 'deg' );
```

In order to calculate the aerodynamic coefficients, the body angular rates (omega) need to be given in the stability axes, like the derivatives. The function `dcmbody2wind` generates the direction cosine matrix for body axes to stability axes (Tsb) when the sideslip angle (beta) is set to zero.

```
Tsb = dcmbody2wind( alpha, zeros(size(alpha)) );
```

The rate of change in angle of attack (alpha_dot) is also needed to find angular rates in stability axis (omega_stab). The function `diff` is used on alpha in degrees divided by data sample time (0.50 seconds) to approximate the rate of change in angle of attack (alpha_dot).

```
alpha_dot = diff( alphadeg/0.50 );
```

The last value of alpha_dot is held to keep the length of alpha_dot consistent with other arrays in this calculation. This is needed because the `diff` function returns an array that is one value shorter that the input

```
alpha_dot = [alpha_dot; alpha_dot(end)];
```

The angular rates in stability axis (omega_stab) are computed for the flight data. The angular rates are reshaped into a 3-D matrix to be multiplied with the 3-D matrix for the direction cosine matrix for body axes to stability axes (Tsb).

```
omega_temp = reshape((omega - [zeros(size(alpha)) alpha_dot zeros(size(alpha))])',3,1,

for k = length(omega):-1:1
    omega_stab(k,:) = (Tsb(:,:,k)*omega_temp(:,:,k))';
end
```

Compute the drag coefficient (CD), the side force coefficient (CY) and the lift coefficient (CL). The `convvel` function is used to get the units of airspeed (Vt) consistent with those of the derivatives.

```
CD = cd;
CY = cyb.*betadeg + cyp.*omega_stab(:,1)*bref/2./convvel(Vt,'kts','ft/s');
CL = cl + clad.*alpha_dot*cbar/2./convvel(Vt,'kts','ft/s');
```

Compute Forces

Aerodynamic coefficients for drag, side force and lift are used to compute aerodynamic forces.

Dynamic pressure is needed to calculate the aerodynamic forces. The function `dpressure` compute dynamic pressure from the airspeed (Vt) and density (rho). The `convvel` function is used to get the units of airspeed (Vt) consistent with those of density (rho).

```
qbar = dpressure( convvel( [Vt zeros(size(Vt,1),2)], 'kts', 'm/s' ), rho );
```

To find forces in body axes, the direction cosine matrix for stability axes to body axes (Tbs) is needed. Direction cosine matrix for stability axes to body axes (Tbs) is the transpose of direction cosine matrix for body axes to stability axes (Tsb). To take the transpose of an 3-D array, the `permute` function is used.

```
Tbs = permute( Tsb, [2 1 3] );
```

Looping through the flight data points, aerodynamic forces are computed and converted from stability to body axes. The `convpres` function is used to get the units of dynamic pressure (qbar) consistent with those of the reference area (Sref).

```
for k = length(qbar):-1:1
    forces_lbs(k,:) = Tbs(:,:,k)*(convpres(qbar(k),'Pa','psf')*Sref*[-CD(k); CY(k); -Cl
end
```

A constant thrust is estimated in the body axes.

```
thrust = ones(length(forces_lbs),1)*[200 0 0];
```

The constant thrust estimate is added to aerodynamic forces and units are converted to metric.

```
forces = convforce((forces_lbs + thrust),'lbf','N');
```

Estimate G Forces

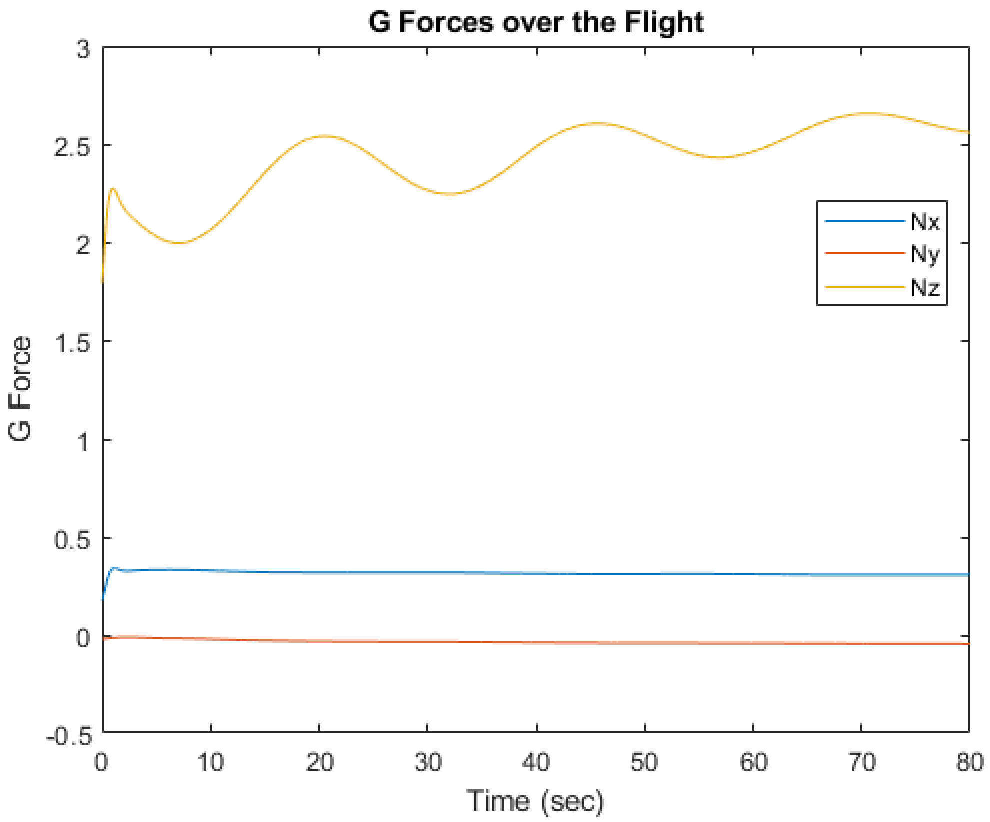
Using the calculated forces, estimate G forces during the flight.

Accelerations are estimated using the calculated forces and mass converted into kilograms using `convmass`. Accelerations are converted to G forces using `convacc`.

```
N = convacc(( forces/convmass(84.2,'slug','kg') ),'m/s^2','G''s');
N = [N(:,1:2) -N(:,3)];
```

G forces are plotted over the flight.

```
h1 = figure;
plot(fltdata(:,1), N);
xlabel('Time (sec)')
ylabel('G Force')
title('G Forces over the Flight')
legend('Nx','Ny','Nz','Location','Best')
```


G Forces over the Flight
G Force
Time (sec)
Nx
Ny
Nz
3
2.5
2
1.5
1
0.5
0
-0.5
0
10
20
30
40
50
60
70
80

```
close(h1);
```

Calculating Best Glide Quantities

This example shows how to perform glide calculations for a Cessna 172 following Example 9.1 in reference 1 using the Aerospace Toolbox software.

Best glide calculations provide values (velocity and glide angle) that minimize drag and maximize lift-drag ratio (also called the glide ratio).

Aircraft Specifications

The aircraft parameters are declared as follows.

```
W = 2400; % weight, lbf
S = 174;  % wing reference area, ft^2;
A = 7.38; % wing aspect ratio
C_D0 = 0.037; % flaps up parasite drag coefficient
e = 0.72; % airplane efficiency factor
```

Conditions

Set the current aircraft conditions. The bank angle (phi) is zero for this case.

```
h = 4000; % altitude, ft
phi = 0; % bank angle, deg
```

Convert altitude to meters using `convlength`. The atmospheric calculations in the next step require values in metric units.

```
h_m = convlength(h,'ft','m');
```

Calculate atmospheric parameters based on altitude using `atmoscoesa`:

```
[T, a, P, rho] = atmoscoesa(h_m, 'Warning');
```

Convert density from metric to English units using `convdensity`:

```
rho = convdensity(rho,'kg/m^3','slug/ft^3');
```

Best Glide Data

Best glide velocity is calculated using the following equation. TAS (true airspeed in feet per second) is the velocity of the aircraft relative to the surrounding air mass.

$$TAS_{bg} = \sqrt{\frac{2W}{\rho S}} \times [\frac{1}{4{C_{D_0}}^2 + C_{D_0}\pi e A \cos^2\phi}]^{\frac{1}{4}}$$

```
TAS_bg = sqrt((2*W) / (rho*S))...
         *(1./(4*C_D0.^2 + C_D0.*pi*e*A*cos(phi)^2)).^(1/4); % TAS, fps
```

Convert velocity from fps to kts using `convvel`. KTAS is true airspeed in knots.

```
KTAS_bg = convvel(TAS_bg,'ft/s','kts')';
```

Convert KTAS to KCAS using `correctairspeed`. KCAS (calibrated airspeed in knots) is the velocity corrected for instrument error and position error. This position error comes from inaccuracies in static pressure measurements at different points in the flight envelope.

```
KCAS_bg = correctairspeed(KTAS_bg,a,P,'TAS','CAS')';
```

Best glide angle is calculated using:

$$\sin \gamma_{bg} = -\sqrt{\frac{4C_{D_0}}{\pi e A \cos^2 \phi + 4C_{D_0}}}$$

This is the angle between the flight path and the ground that provides the highest L/D ratio.

```
gamma_bg_rad = asin( -sqrt((4.*C_D0')./(pi*e*A*cos(phi)^2 + 4.*C_D0')) );
```

Convert glide angle from radians to degrees using `convang`:

```
gamma_bg = convang(gamma_bg_rad,'rad','deg');
```

Best glide drag is calculated using:

$$D_{min} = D_{bg} = \frac{1}{2}\rho(TAS_{bg}^2)S(2C_{D_0}) = -W \sin \gamma_{bg}$$

```
D_bg = -W*sin(gamma_bg_rad);
```

Best glide lift is calculated using:

$$L_{bg} = L_{max} = W \cos \gamma_{bg} = \sqrt{W^2 - D_{bg}^2}$$

```
L_bg =  W*cos(gamma_bg_rad);
```

Calculate dynamic pressure using `dpressure`:

```
qbar = dpressure([TAS_bg' zeros(size(TAS_bg,2),2)], rho);
```

Calculate drag and lift coefficients using:

$$C_{D_{bg}} = \frac{D_{bg}}{\bar{q}S}$$

$$C_{L_{bg}} = \frac{L_{bg}}{\bar{q}S}$$

```
C_D_bg = D_bg./(qbar*S);
C_L_bg = L_bg./(qbar*S);
```

Summary of Best Glide Values

Here are the best glide values:

$$KCAS_{bg} = 71.9\,KCAS$$

$$\gamma_{bg} = -5.38\,\mathrm{deg}$$

$$C_{D_{bg}} = 0.074$$

$$C_{L_{bg}} = 0.7859$$

$$D_{bg} = 224.9\,lbf$$

$$L_{bg} = 2389.4\,lbf$$

Verification

These plots show drag and lift-drag ratio plots for the aircraft as a function of KCAS. The plots are used to verify the best glide calculations.

Set range of airspeeds and convert to KCAS using `convvel` and `correctairspeed`:

```
TAS = (70:200)'; % true airspeed, fps
KTAS = convvel(TAS,'ft/s','kts')'; % true airspeed, kts
KCAS = correctairspeed(KTAS,a,P,'TAS','CAS')'; % corrected airspeed, kts
```

Calculate dynamic pressure for new airspeeds using `dpressure`:

```
qbar = dpressure([TAS zeros(size(TAS,1),2)], rho);
```

Calculate parasite drag using:

$$D_p = \frac{1}{2}\rho S C_{D_0}(TAS^2)$$

```
Dp = qbar*S.*C_D0;
```

Calculate induced drag using:

$$D_i = \frac{2W^2}{\rho S \pi e A}\frac{1}{(TAS^2)}$$

```
Di = (2*W^2)/(rho*S*pi*e*A).*(TAS.^-2);
```

Calculate total drag using:

$$D = D_p + D_i$$

```
D = Dp + Di;
```

Approximate lift as weight (assuming small glide angle and small angle of attack). At this speed, assuming

$$C_L = 2\pi\alpha$$

and using

$$C_{L_{bg}}$$

from above, the angle of attack is about 7 degrees. Adding the flight path angle (i.e. best glide angle) from above shows the fuselage pitch (attitude angle theta) to be about 2 degrees.

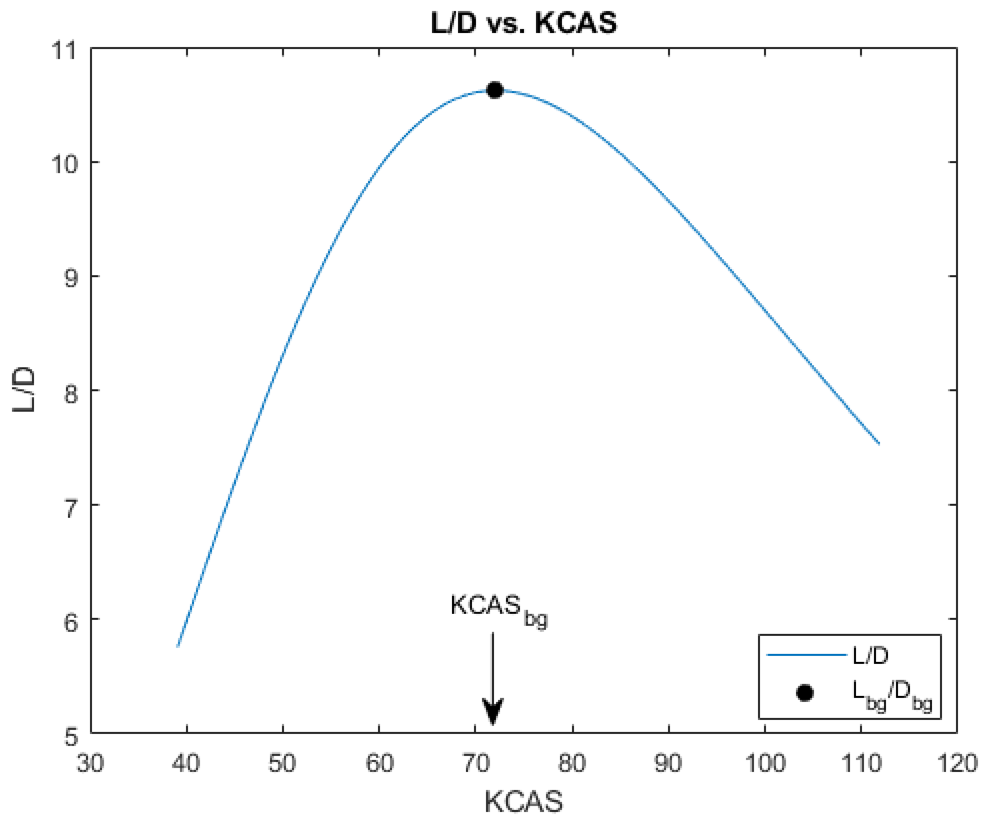
```
L = W;
```

Plot L/D versus KCAS

As expected, the maximum L/D occurs at approximately the best glide velocity calculated above.

```
h1 = figure;
plot(KCAS,L./D);
title('L/D vs. KCAS');
```

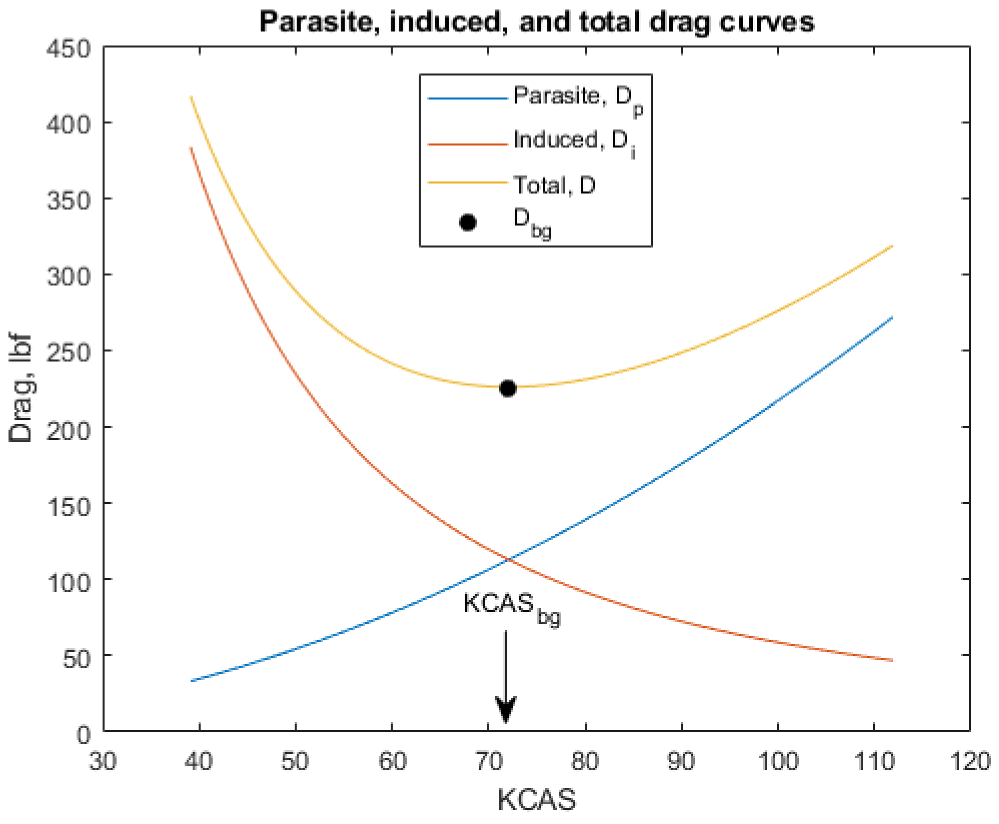
```
xlabel('KCAS'); ylabel('L/D');
hold on
plot(KCAS_bg,L_bg/D_bg,'Marker','o','MarkerFaceColor','black',...
    'MarkerEdgeColor','black','Color','white');
hold off
legend('L/D','L_{bg}/D_{bg}','Location','Best');
annotation('textarrow',[0.49 0.49],[0.23 0.12],'String','KCAS_{bg}');
```



Plot parasite, induced, and total drag curves

Notice the minimum total drag (i.e. D_bg) occurs at approximately the same best glide velocity calculated above.

```
h2 = figure;
plot(KCAS,Dp,KCAS,Di,KCAS,D);
title('Parasite, induced, and total drag curves');
xlabel('KCAS'); ylabel('Drag, lbf');
hold on
plot(KCAS_bg,D_bg,'Marker','o','MarkerFaceColor','black',...
    'MarkerEdgeColor','black','Color','white');
hold off
legend('Parasite, D_p','Induced, D_i','Total, D','D_{bg}','Location','Best');
annotation('textarrow',[0.49 0.49],[0.23 0.12],'String','KCAS_{bg}');
```



```
close(h1,h2);
```

Reference

[1] Lowry, J. T., "Performance of Light Aircraft", AIAA(R) Education Series, Washington, DC, 1999.

Overlaying Simulated and Actual Flight Data

This example shows how to visualize simulated versus actual flight trajectories with the animation object (Aero.Animation) while showing some of the animation object functionality. In this example, you can use the Aero.Animation object to create and configure an animation object, then use that object to create, visualize, and manipulate bodies for the flight trajectories.

Create the Animation Object

This code creates an instance of the `Aero.Animation` object.

```
h = Aero.Animation;
```

Set the Animation Object Properties

This code sets the number of frames per second. This controls the rate at which frames are displayed in the figure window.

```
h.FramesPerSecond = 10;
```

This code sets the seconds of animation data per second time scaling. This property and the `'FramesPerSecond'` property determine the time step of the simulation. The settings in this example result in a time step of approximately 0.5s. The equation is (1/FramesPerSecond)*TimeScaling along with some extra terms to handle for sub-second precision.

```
h.TimeScaling = 5;
```

Create and Load Bodies

This code loads the bodies using `createBody` for the animation object, `h`. This example will use these bodies to work with and display the simulated and actual flight trajectories. The first body is orange and will represent simulated data. The second body is blue and will represent the actual flight data.

```
idx1 = h.createBody('pa24-250_orange.ac','Ac3d');
idx2 = h.createBody('pa24-250_blue.ac','Ac3d');
```

Load Recorded Data for Flight Trajectories

Using the bodies from the previous code, this code provides simulated and actual recorded data for flight trajectories in the following files:

- The simdata file contains logged simulated data. `simdata` is set up as a 6DoF array, which is one of the default data formats.
- The fltdata file contains actual flight test data. In this example, `fltdata` is set up in a custom format. The example must create a custom read function and set the `'TimeSeriesSourceType'` parameter to `'Custom'`.

To load the simdata and fltdata files:

```
load simdata
load fltdata
```

To work with the custom flight test data, this code sets the second body `'TimeSeriesReadFcn'`. The custom read function is located here: `matlabroot/toolbox/aero/astdemos/CustomReadBodyTSData.m`

```
h.Bodies{2}.TimeseriesReadFcn = @CustomReadBodyTSData;
```

Set the bodies' timeseries data.

```
h.Bodies{1}.TimeSeriesSource = simdata;
h.Bodies{2}.TimeSeriesSource = fltdata;
h.Bodies{2}.TimeSeriesSourceType = 'Custom';
```

Camera Manipulation

This code illustrates how you can manipulate the camera for the two bodies.

The `'PositionFcn'` property of a camera object controls the camera position relative to the bodies in the animation. The default camera `'PositionFcn'` follows the path of a first order chase vehicle. Therefore, it takes a few steps for the camera to position itself correctly in the chase plane position. The default `'PositionFcn'` is located here: `matlabroot/toolbox/aero/aero/doFirstOrderChaseCameraDynamics.m`

Set `'PositionFcn'`.

```
h.Camera.PositionFcn = @doFirstOrderChaseCameraDynamics;
```

Display Body Geometries in Figure

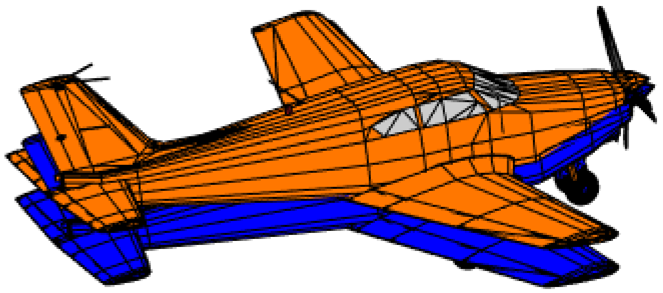
This code uses the `show` method to create the figure graphics object for the animation object.

```
h.show();
```

Use the Animation Object to Play Back Flight Trajectories

This code uses the `play` method to animate bodies for the duration of the timeseries data. Using this method will illustrate the slight differences between the simulated and flight data.

```
h.play();
```



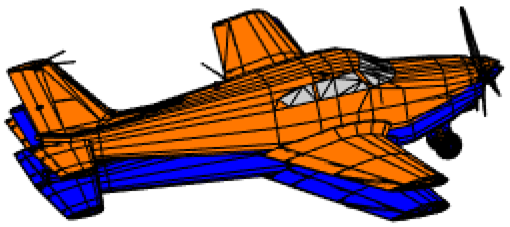
The code can also use a custom, simplified `'PositionFcn'` that is a static position based on the position of the bodies (i.e., no dynamics). The simplified `'PositionFcn'` is located here: `matlabroot/toolbox/aero/astdemos/staticCameraPosition.m`

Set the new `'PositionFcn'`.

```
h.Camera.PositionFcn = @staticCameraPosition;
```

Run the animation with new `'PositionFcn'`.

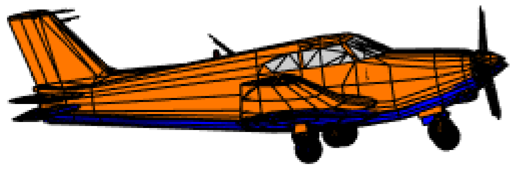
```
h.play();
```



Move Bodies

This code illustrates how to move the bodies to the starting position (based on timeseries data) and update the camera position according to the new `'PositionFcn'`. This code uses `updateBodies` and `updateCamera`.

```
t = 0;
h.updateBodies(t);
h.updateCamera(t);
```



Reposition Bodies

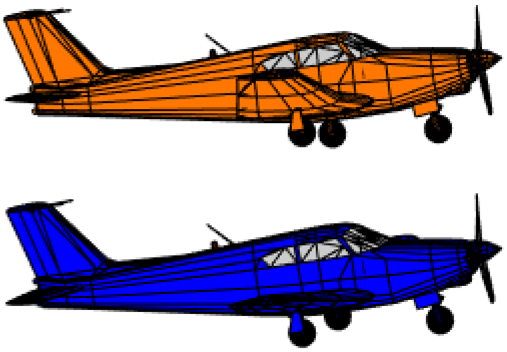
This code illustrates how to reposition the bodies by first getting the current body position and then separating the bodies.

Get current body positions and rotations from the body objects.

```
pos1 = h.Bodies{1}.Position;
rot1 = h.Bodies{1}.Rotation;
pos2 = h.Bodies{2}.Position;
rot2 = h.Bodies{2}.Rotation;
```

Separate bodies using `moveBody`. This code separates and repositions the two bodies.

```
h.moveBody(1,pos1 + [0 0 -3],rot1);
h.moveBody(2,pos1 + [0 0  0],rot2);
```



Create Transparency in the First Body

This code illustrates how to create transparency in the first body. The code does this by changing the body patch properties via `'PatchHandles'`. (For more information on patches in MATLAB®, see the `Introduction to Patch Objects` section in the MATLAB documentation.)

Note: On some platforms utilizing software OpenGL® rendering, the transparency may cause a decrease in animation speed.

See the `opengl` documentation for more information on OpenGL in MATLAB.

To create a transparency, the code gets the patch handles for the first body.

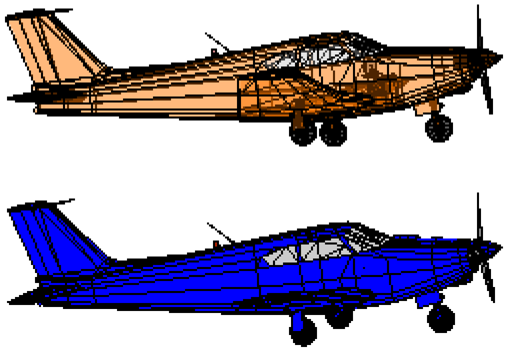
```
patchHandles2 = h.Bodies{1}.PatchHandles;
```

Set desired face and edge alpha values.

```
desiredFaceTransparency = .3;
desiredEdgeTransparency = 1;
```

This code gets the current face and edge alpha data and changes all values to desired alpha values. In the figure, notice the first body now has a transparency.

```
for k = 1:size(patchHandles2,1)
    tempFaceAlpha = get(patchHandles2(k),'FaceVertexAlphaData');
    tempEdgeAlpha = get(patchHandles2(k),'EdgeAlpha');
    set(patchHandles2(k),...
        'FaceVertexAlphaData',repmat(desiredFaceTransparency,size(tempFaceAlpha)));
    set(patchHandles2(k),...
        'EdgeAlpha',repmat(desiredEdgeTransparency,size(tempEdgeAlpha)));
end
```



Change Color of the Second Body

This code illustrates how to change the body color of the second body. The code does this by changing the body patch properties via `'PatchHandles'`.

```
patchHandles3 = h.Bodies{2}.PatchHandles;
```

This code sets the desired patch color (red).

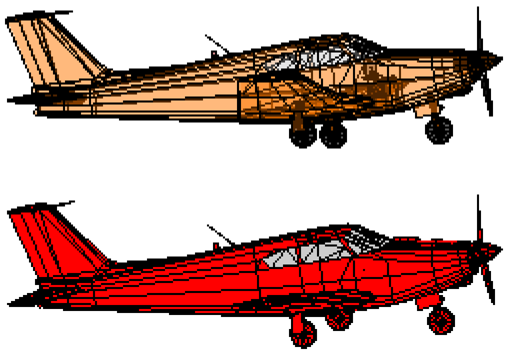
```
desiredColor = [1 0 0];
```

The code can now get the current face color data and change all values to desired color values. Note the following points on the code:

- The `if` condition keeps the windows from being colored.
- The name property is stored in the geometry data of the body (h.Bodies{2}.Geometry.FaceVertexColorData(k).name).
- The code only changes the indices in `patchHandles3` with non-window counterparts in the body geometry data.

The name property might not always be available to determine various parts of the vehicle. In these cases, you will need to use an alternative approach to selective coloring.

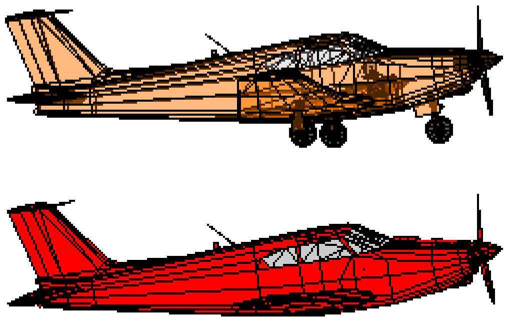
```
for k = 1:size(patchHandles3,1)
    tempFaceColor = get(patchHandles3(k),'FaceVertexCData');
    tempName = h.Bodies{2}.Geometry.FaceVertexColorData(k).name;
    if ~contains(tempName,'Windshield') &&...
       ~contains(tempName,'front-windows') &&...
       ~contains(tempName,'rear-windows')
    set(patchHandles3(k),...
        'FaceVertexCData',repmat(desiredColor,[size(tempFaceColor,1),1]));
    end
end
```



Turn Off Landing Gear on Second Body

The following code turns off the landing gear for the second body. To do this, it turns off the visibility of all vehicle parts associated with the landing gear. Note the indices into the `patchHandles3` vector were determined from the name property in the geometry data. Other data sources might not have this information available. In these cases, you will need to know which indices correspond to particular parts of the geometry.

```
for k = [1:8,11:14,52:57]
    set(patchHandles3(k),'Visible','off')
end
```



Close and Delete Animation Object

To close and delete

```
h.delete();

%#ok<*REPMAT>
```

Comparing Zonal Harmonic Gravity Model to Other Gravity Models

This example shows how to examine the zonal harmonic, spherical and 1984 World Geodetic System (WGS84) gravity models for latitudes from +/- 90 degrees at the surface of the Earth.

Determine Earth-Centered Earth-Fixed (ECEF) Position

Since ECEF coordinate system is geocentric, you can use spherical equations to determine the position from the latitude, longitude and geocentric radius.

Calculate the geocentric radii in meters at an array of latitudes from +/- 90 degrees using `geocradius`.

```
lat = -90:90;
r = geocradius( lat );
rlat = convang( lat, 'deg', 'rad');
z = r.*sin(rlat);
```

Because longitude has no effect for zonal harmonic gravity model, assume that the y position is zero.

```
x = r.*cos(rlat);
y = zeros(size(x));
```

Compute Zonal Harmonic Gravity for Earth

Use `gravityzonal` to calculate array of zonal harmonic gravity in ECEF coordinates for array of ECEF positions in meters per seconds squared.

```
[gx_zonal, gy_zonal, gz_zonal] = gravityzonal([x' y' z']);
```

Calculate WGS84 Gravity

Use `gravitywgs84` to compute WGS84 gravity in down-axis and north-axis at the Earth's surface, an array of geodetic latitudes in degrees and 0 degrees longitude using the exact method with atmosphere, no centrifugal effects, and no precessing.

```
lat_gd = geoc2geod(lat, r);
long_gd = zeros(size(lat));
[gd_wgs84, gn_wgs84] = gravitywgs84( zeros(size(lat)), lat_gd, long_gd, 'Exact', [false
```

Determine Gravity for Spherical Earth

Compute the array of spherical gravity for the array of geocentric radii in meters per second squared using the Earth's gravitational parameter in meters cubed per second squared.

```
GM  = 3.986004415e14;
gd_sphere = -GM./(r.*r);
```

Comparison Plots for Different Gravity Models

To compare the gravity models, their outputs must be in the same coordinate system. You can transform zonal gravity from ECEF coordinates to NED coordinates by using the Direction Cosine Matrix from `dcmecef2ned`.

```
dcm_ef = dcmecef2ned( lat_gd, long_gd );
gxyz_zonal = reshape([gx_zonal gy_zonal gz_zonal]', [3 1 181]);
for m = length(lat):-1:1
    gned_zonal(m,:) = (dcm_ef(:,:,m)*gxyz_zonal(:,:,m))';
end

figure(1)
plot(lat_gd, gned_zonal(:,3), lat, -gd_sphere, '--', lat_gd, gd_wgs84, '-.')
legend('Zonal Harmonic', 'Spherical', 'WGS84','Location','North')
xlabel('Geodetic Latitude (degrees)')
ylabel('Down gravity (m/s^2)')
grid
```

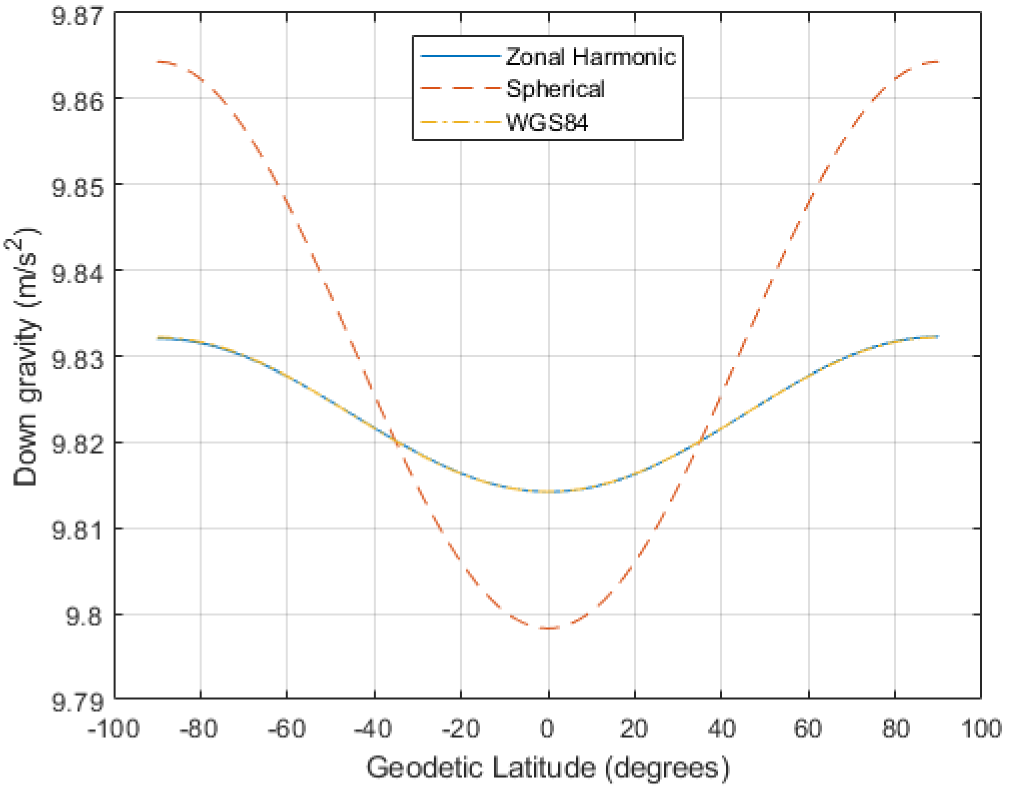


Figure 1: Gravity in the Down-axis in meters per second squared

```
figure(2)
plot( lat_gd, gned_zonal(:,1), lat_gd, gn_wgs84, 'r-.' )
hold on
plot([lat(1) lat(end)], [0 0], 'g--' )
legend('Zonal Harmonic', 'WGS84', 'Spherical','Location','Best')
xlabel('Geodetic Latitude (degrees)')
ylabel('North gravity (m/s^2)')
grid
hold off
```

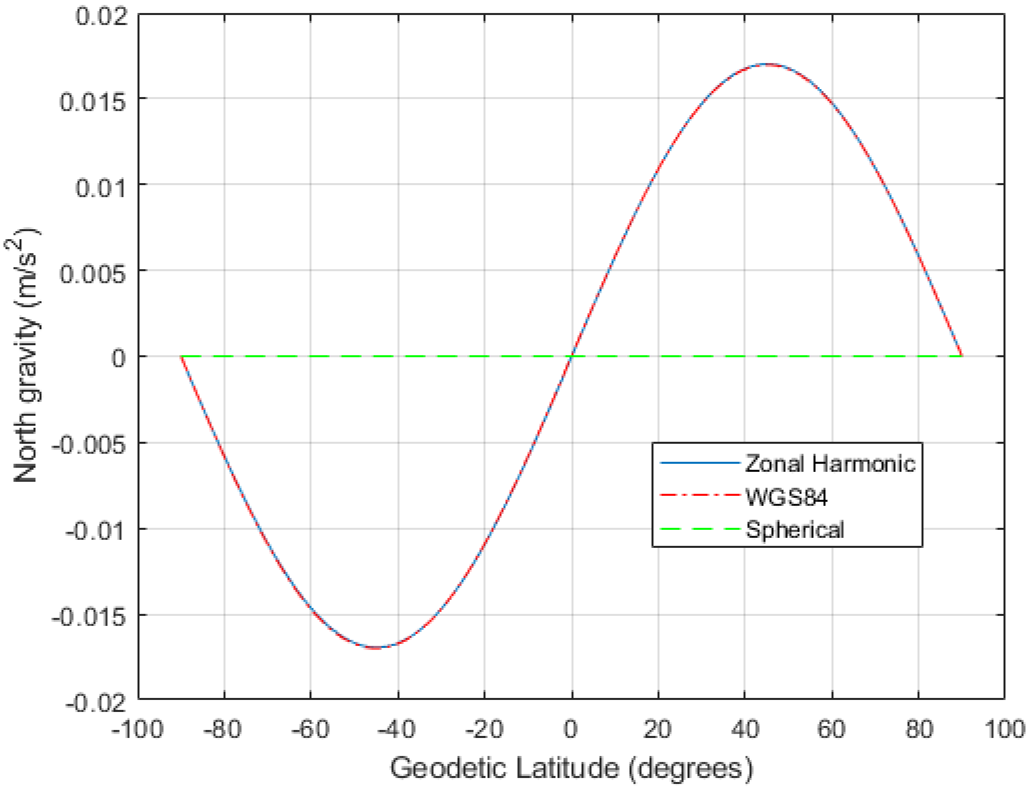


Figure 2: Gravity in the North-axis in meters per second squared

Calculate total gravity for WGS84 and from zonal gravity vector in meters per second squared.

```
gtotal_wgs84 = gravitywgs84( zeros(size(lat)), lat_gd, zeros(size(lat)), 'Exact', [fals
gtotal_zonal = sqrt(sum([gx_zonal.^2 gy_zonal.^2 gz_zonal.^2],2));

figure(3)
plot( lat, gtotal_zonal, lat_gd, gtotal_wgs84, '-.' )
legend('Zonal Harmonic', 'WGS84','Location','North')
xlabel('Geodetic Latitude (degrees)')
ylabel('Total gravity (m/s^2)')
grid
```

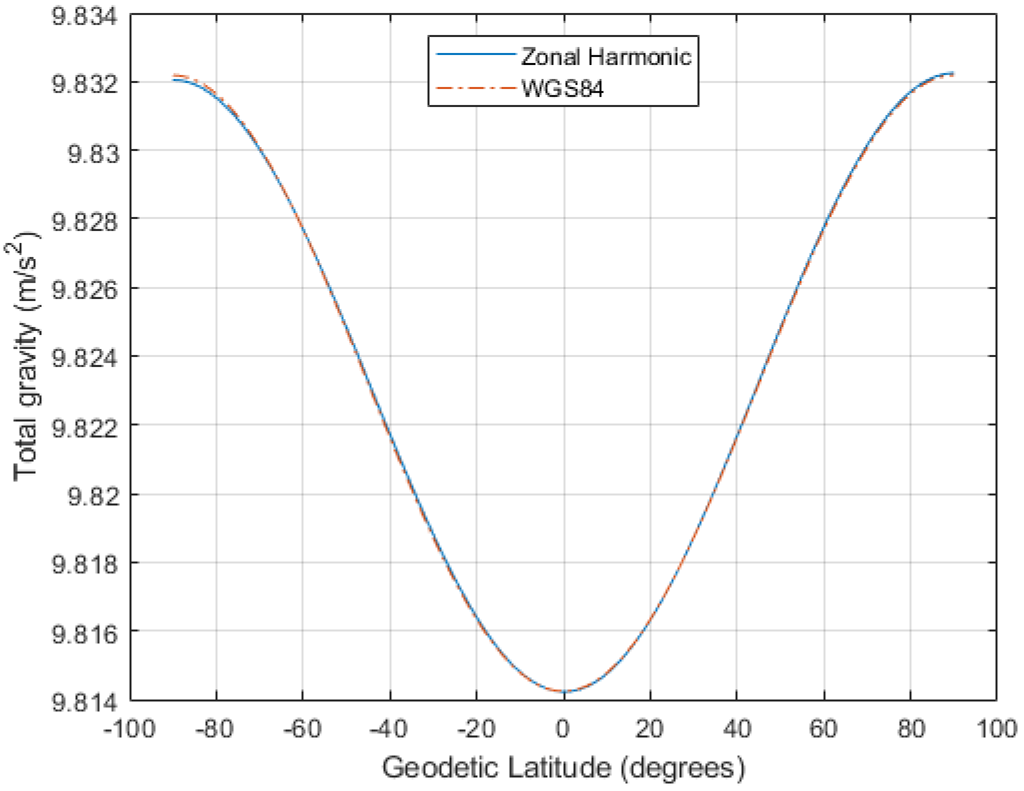


Figure 3: Total gravity in meters per second squared

Compare Gravity Models with Centrifugal Effects

Now, you have seen the gravity comparisons of a non-rotating Earth. Examine the centrifugal effects from the Earth's rotation on the gravity models.

Compute Gravity Centrifugal Effects for Earth

Use `gravitycentrifugal` to calculate array of centrifugal effects in ECEF coordinates for array of ECEF positions in meters per seconds squared.

```
[gx_cent, gy_cent, gz_cent] = gravitycentrifugal([x' y' z']);
```

Add centrifugal effects to zonal harmonic gravity.

```
gx_cent_zonal = gx_zonal + gx_cent;
gy_cent_zonal = gy_zonal + gy_cent;
gz_cent_zonal = gz_zonal + gz_cent;
```

Calculate WGS84 Gravity with Centrifugal Effects

Use `gravitywgs84` to compute WGS84 gravity in down-axis and north-axis at the Earth's surface, an array of geodetic latitudes in degrees and 0 degrees longitude using the exact method with atmosphere, centrifugal effects, and no precessing.

```
[gd_cent_wgs84, gn_cent_wgs84] = gravitywgs84( zeros(size(lat)), lat_gd, long_gd, 'Exac
```

Calculate total gravity with centrifugal effects for WGS84 and from zonal gravity vector in meters per second squared.

```
gtotal_cent_wgs84 = gravitywgs84( zeros(size(lat)), lat_gd, zeros(size(lat)), 'Exact',
gtotal_cent_zonal = sqrt(sum([gx_cent_zonal.^2 gy_cent_zonal.^2 gz_cent_zonal.^2],2));
```

Comparison Plots for Different Gravity Models with Centrifugal Effects

To compare the gravity models, their outputs must be in the same coordinate system. You can transform zonal gravity from ECEF coordinates to NED coordinates by using the Direction Cosine Matrix from `dcmecef2ned`. In figure 5, you can see there is some difference between zonal harmonic gravity with centrifugal effects and WGS84 gravity with centrifugal effects. The majority of difference is due to differences between the zonal harmonic gravity and WGS84 gravity calculations.

```
gxyz_cent_zonal = reshape([gx_cent_zonal gy_cent_zonal gz_cent_zonal]', [3 1 181]);
for m = length(lat):-1:1
    gned_cent_zonal(m,:) = (dcm_ef(:,:,m)*gxyz_cent_zonal(:,:,m))';
end

figure(4)
plot(lat_gd, gned_cent_zonal(:,3), lat_gd, gd_cent_wgs84, '-.')
legend('Zonal Harmonic', 'WGS84','Location','North')
xlabel('Geodetic Latitude (degrees)')
ylabel('Down gravity (m/s^2)')
grid
```

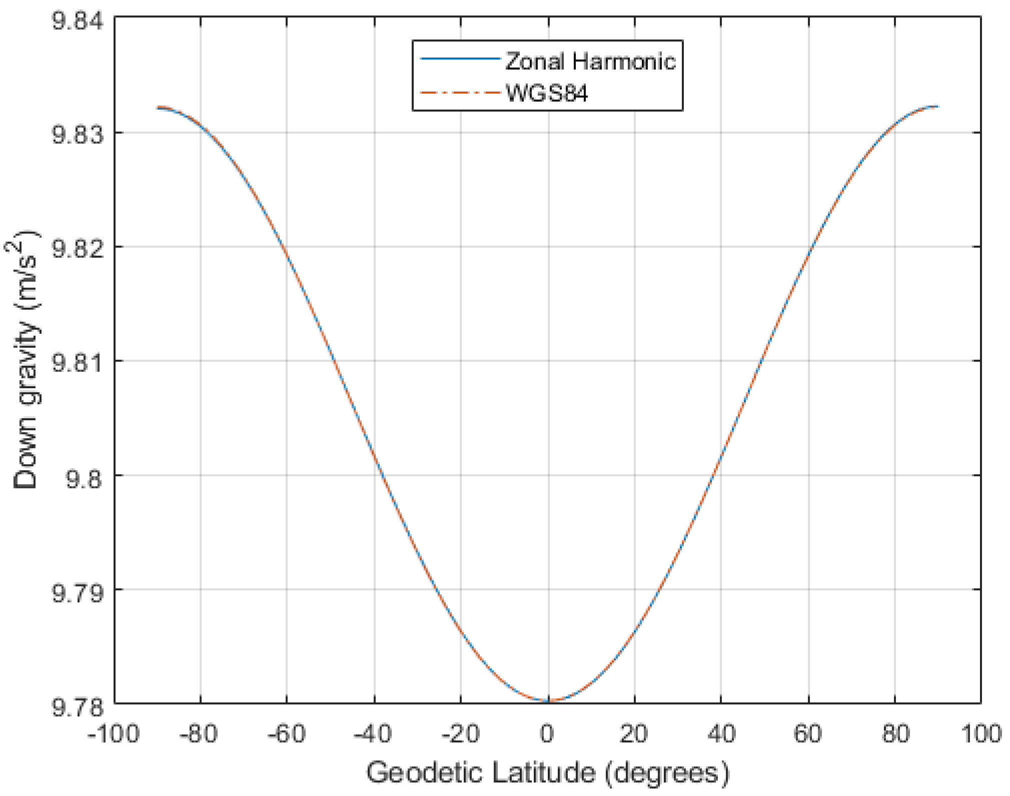


Figure 4: Gravity with centrifugal effects in the Down-axis in meters per second squared

```
figure(5)
plot( lat_gd, gned_cent_zonal(:,1), lat_gd, gn_cent_wgs84, '--', lat_gd, (gned_zonal(:,
axis([-100 100 -0.0002 0.0002])
legend('Zonal Harmonic', 'WGS84', 'Error Between Models w/o Centrifugal Effects', 'Loca
xlabel('Geodetic Latitude (degrees)')
ylabel('North gravity (m/s^2)')
grid
```

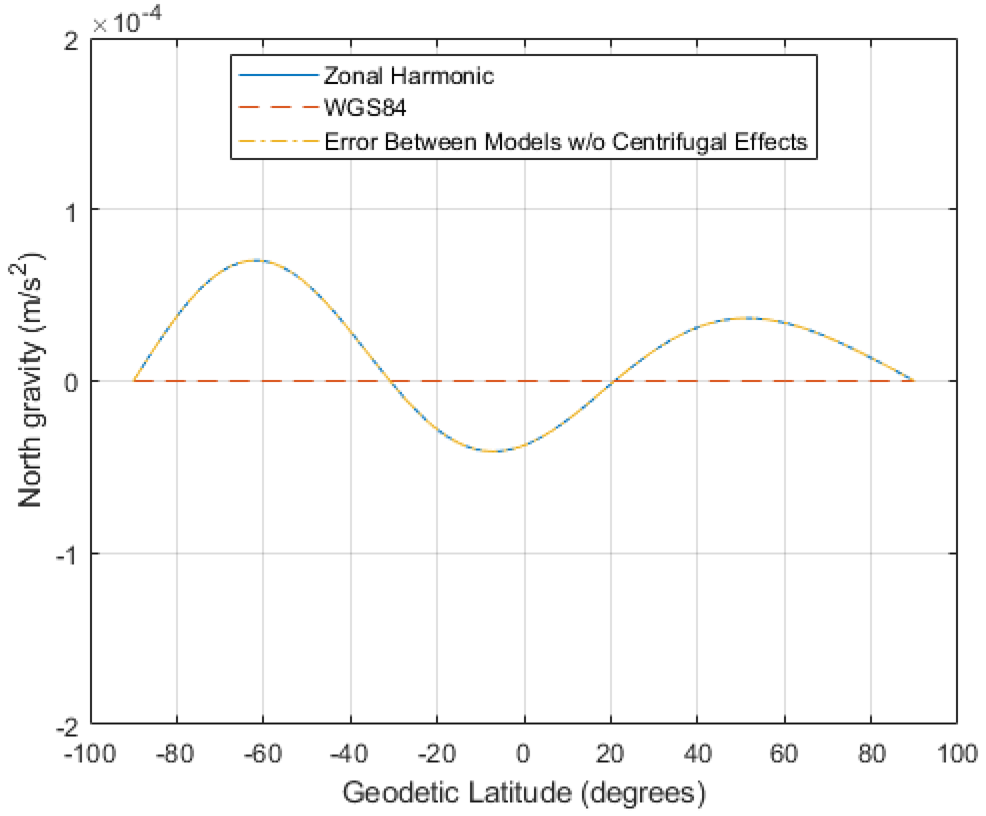


Figure 5: Gravity in the North-axis in meters per second squared

```
figure(6)
plot( lat, gtotal_cent_zonal, lat_gd, gtotal_cent_wgs84, '-.' )
legend('Zonal Harmonic', 'WGS84','Location','North')
xlabel('Geodetic Latitude (degrees)')
ylabel('Total gravity (m/s^2)')
grid
```

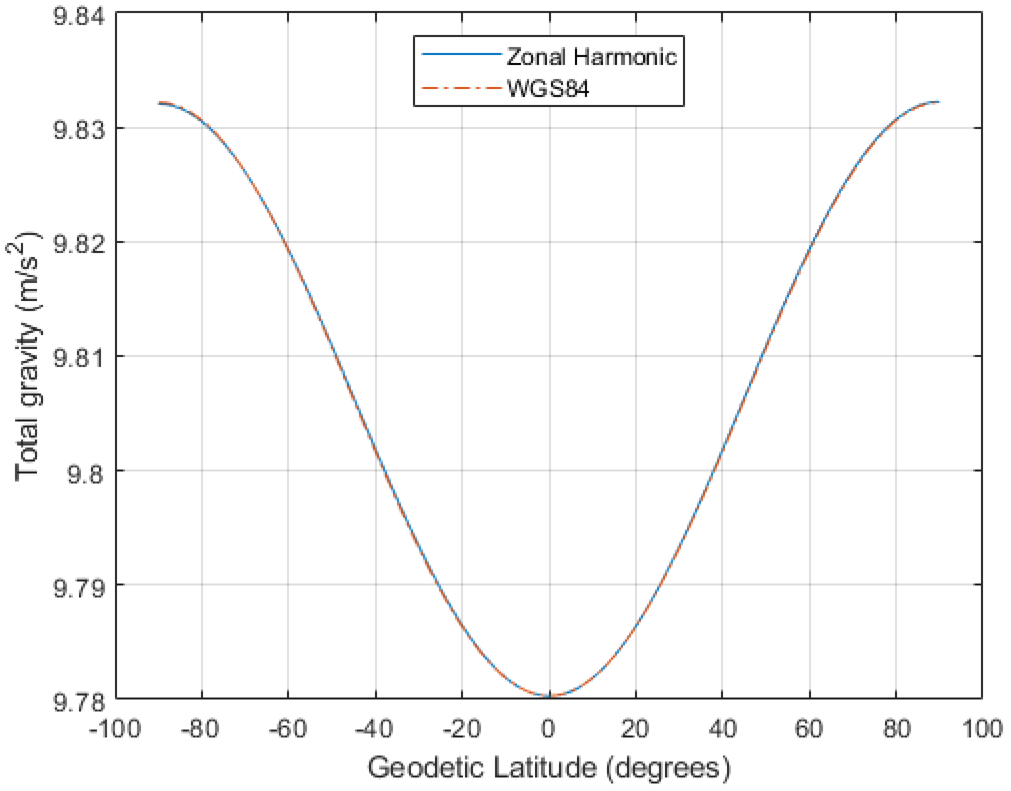


Figure 6: Total gravity with centrifugal effects in meters per second squared

Visualize Aircraft Takeoff via Virtual Reality Animation Object

This example shows how to visualize aircraft takeoff and chase helicopter with the virtual reality animation object. In this example, you can use the Aero.VirtualRealityAnimation object to set up a virtual reality animation based on the asttkoff.wrl file. The scene simulates an aircraft takeoff. The example adds a chase vehicle to the simulation and a chase viewpoint associated with the new vehicle.

Create the Animation Object

This code creates an instance of the `Aero.VirtualRealityAnimation` object.

```
h = Aero.VirtualRealityAnimation;
```

Set the Animation Object Properties

This code sets the number of frames per second and the seconds of animation data per second time scaling. `'FramesPerSecond'` controls the rate at which frames are displayed in the figure window. `'TimeScaling'` is the seconds of animation data per second time scaling.

The `'TimeScaling'` and `'FramesPerSecond'` properties determine the time step of the simulation. The settings in this example result in a time step of approximately 0.5s. The equation is:

(1/FramesPerSecond)*TimeScaling + extra terms to handle for sub-second precision.

```
h.FramesPerSecond = 10;
h.TimeScaling = 5;
```

This code sets the .wrl file to be used in the virtual reality animation.

```
h.VRWorldFilename = [matlabroot,'/toolbox/aero/astdemos/asttkoff.wrl'];
```

Change Directory

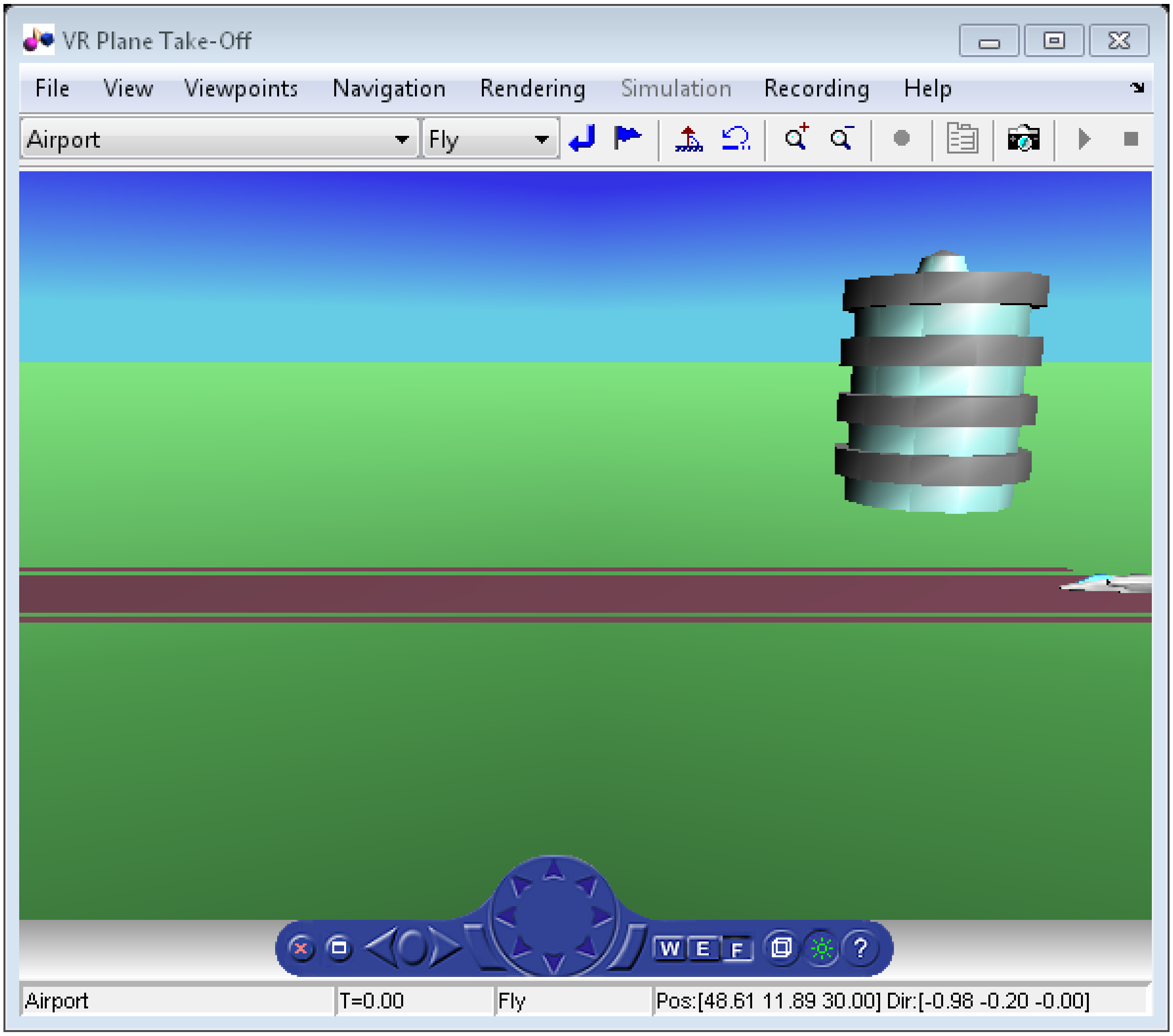
The VirtualRealityAnimation object methods use temporary .wrl files to keep track of changes to the world. This requires the directory containing the original .wrl file to be writable. This code runs the example from a temporary directory to ensure there are no issues with directory permissions. Note, a license for Simulink® 3D Animation™ is required to run this example.

```
% Copy file to temporary directory
copyfile(h.VRWorldFilename,[tempdir,'asttkoff.wrl'],'f');
% Set world filename to the copied .wrl file.
h.VRWorldFilename = [tempdir,'asttkoff.wrl'];
```

Initialize the Virtual Reality Animation Object

The `initialize` method loads the animation world described in the `'VRWorldFilename'` field of the animation object. When parsing the world, node objects are created for existing nodes with DEF names. The `initialize` method also opens the Simulink 3D Animation viewer.

```
h.initialize();
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F
?

Airport T=0.00 Fly Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.00]

Set Additional Node Information

This code sets simulation timeseries data. `takeoffData.mat` contains logged simulated data. `takeoffData` is set up as a `'StructureWithTime'`, which is one of the default data formats.

```
load takeoffData
[~, idxPlane] = find(strcmp('Plane', h.nodeInfo));
h.Nodes{idxPlane}.TimeseriesSource = takeoffData;
h.Nodes{idxPlane}.TimeseriesSourceType = 'StructureWithTime';
```

Set Coordinate Transform Function

The virtual reality animation object expects positions and rotations in aerospace body coordinates. If the input data is different, you must create a coordinate transformation function in order to correctly line up the position and rotation data with the surrounding objects in the virtual world. This code sets the coordinate transformation function for the virtual reality animation.

In this particular case, if the input translation coordinates are [x1,y1,z1], they must be adjusted as follows: [X,Y,Z] = -[y1,x1,z1]. The custom transform function can be seen here: `matlabroot/toolbox/aero/astdemos/vranimCustomTransform.m`

```
h.Nodes{idxPlane}.CoordTransformFcn = @vranimCustomTransform;
```

Add a Chase Helicopter

This code shows how to add a chase helicopter to the animation object.

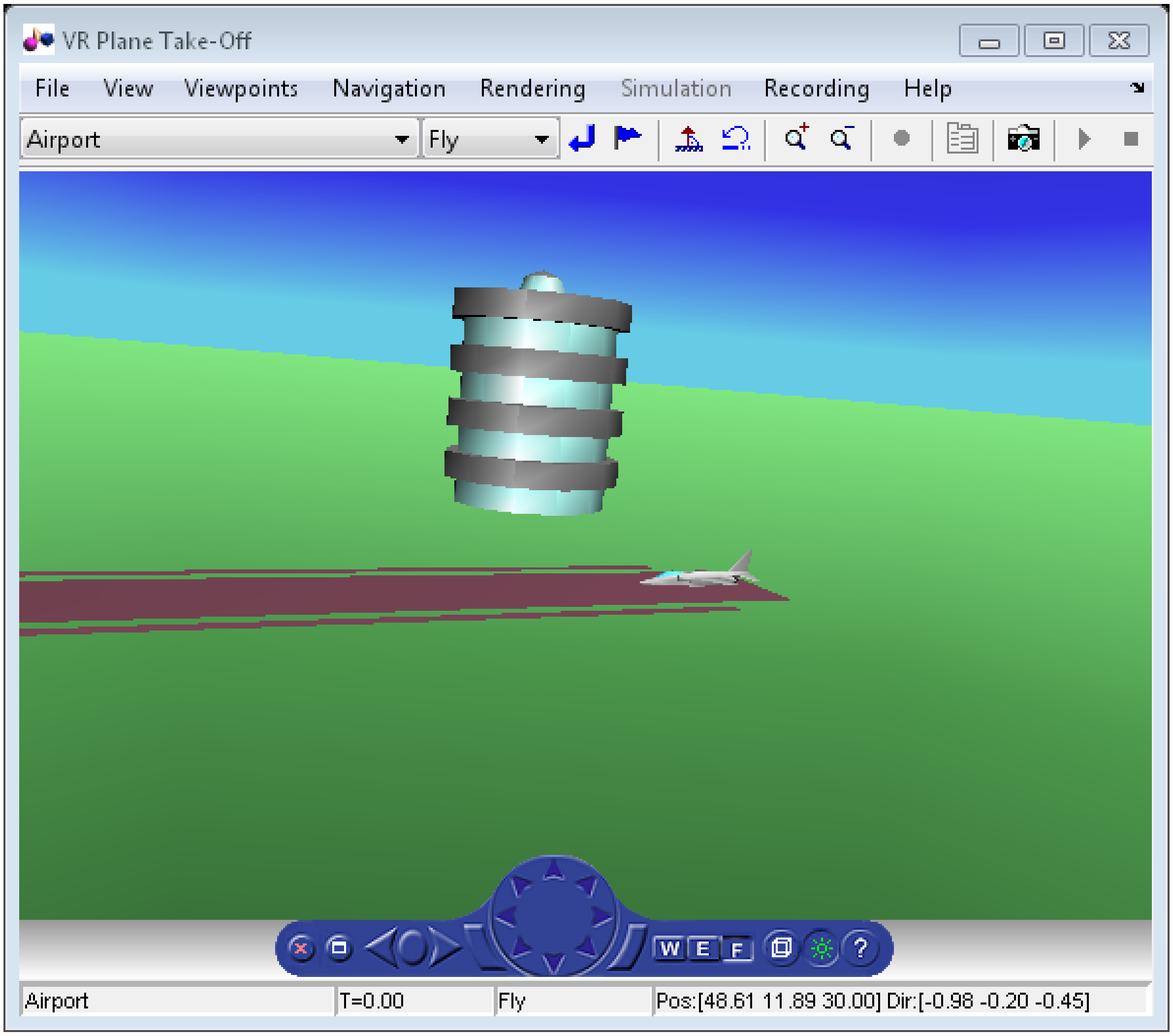
You can view all the nodes currently in the virtual reality animation object by using the `nodeInfo` method. When called with no output argument, this method prints the node information to the command window. With an output argument, the method sets node information to that argument.

```
h.nodeInfo;
```

```
Node Information
1    Camera1
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
```

This code moves the camera angle of the virtual reality figure to view the aircraft.

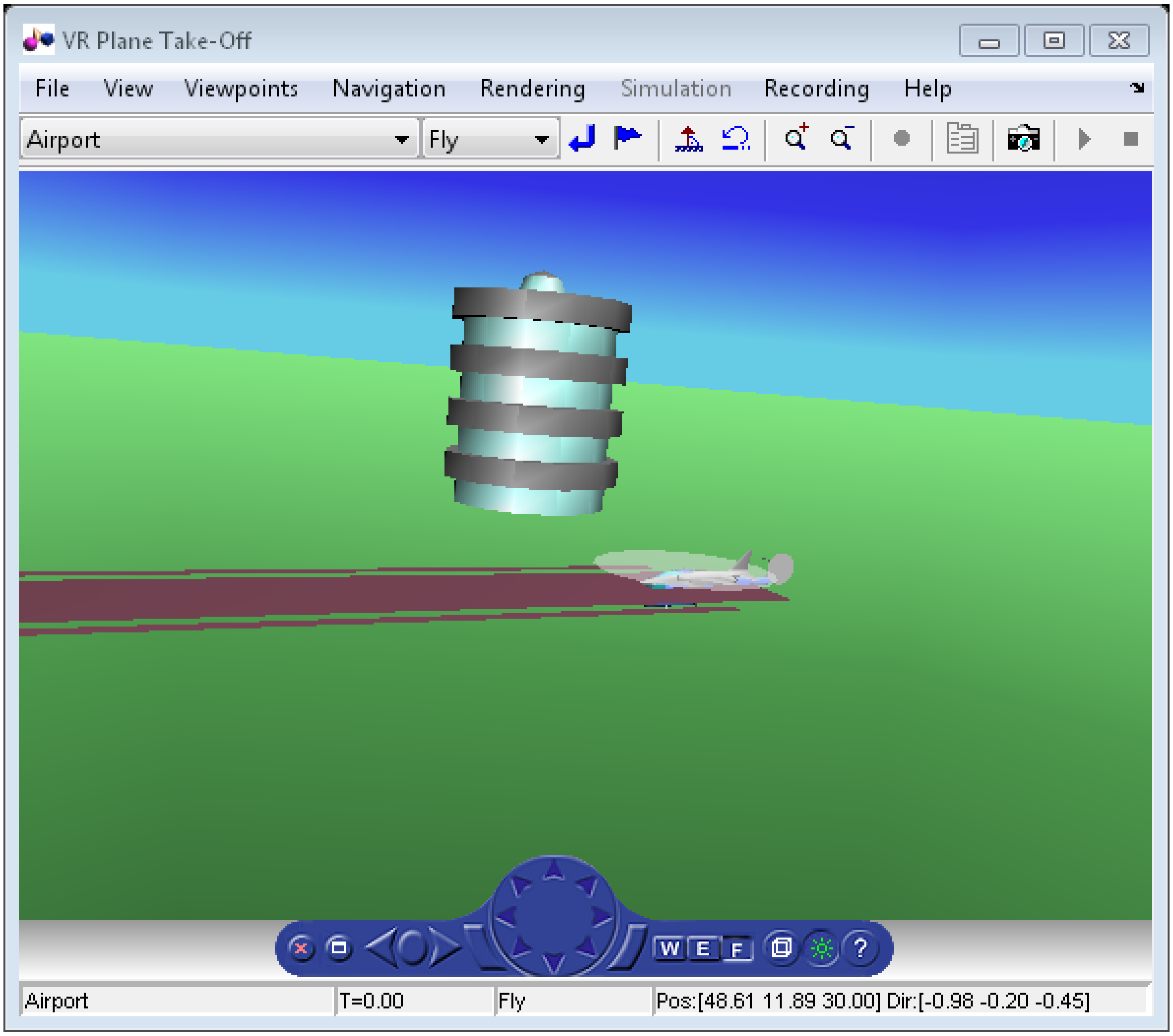
```
set(h.VRFigure,'CameraDirection',[0.45 0 -1]);
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F

Airport T=0.00 Fly Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.45]

Use the addNode method to add another node to the object. By default, each time you add or remove a node or route, or when you call the saveas method, Aerospace Toolbox displays a message about the current .wrl file location. To disable this message, set the 'ShowSaveWarning' property in the VirtualRealityAnimation object.

```
h.ShowSaveWarning = false;
h.addNode('Lynx',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F
Airport
T=0.00
Fly
Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.45]

Another call to `nodeInfo` shows the newly added Node objects.

```
h.nodeInfo
```

```
Node Information
1    Camera1
```

```
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
9    Lynx
10    Lynx_Inline
```

Adjust newly added helicopter to sit on runway.

```
[~, idxLynx] = find(strcmp('Lynx',h.nodeInfo));
h.Node{idxLynx}.VRNode.translation = [0 1.5 0];
```

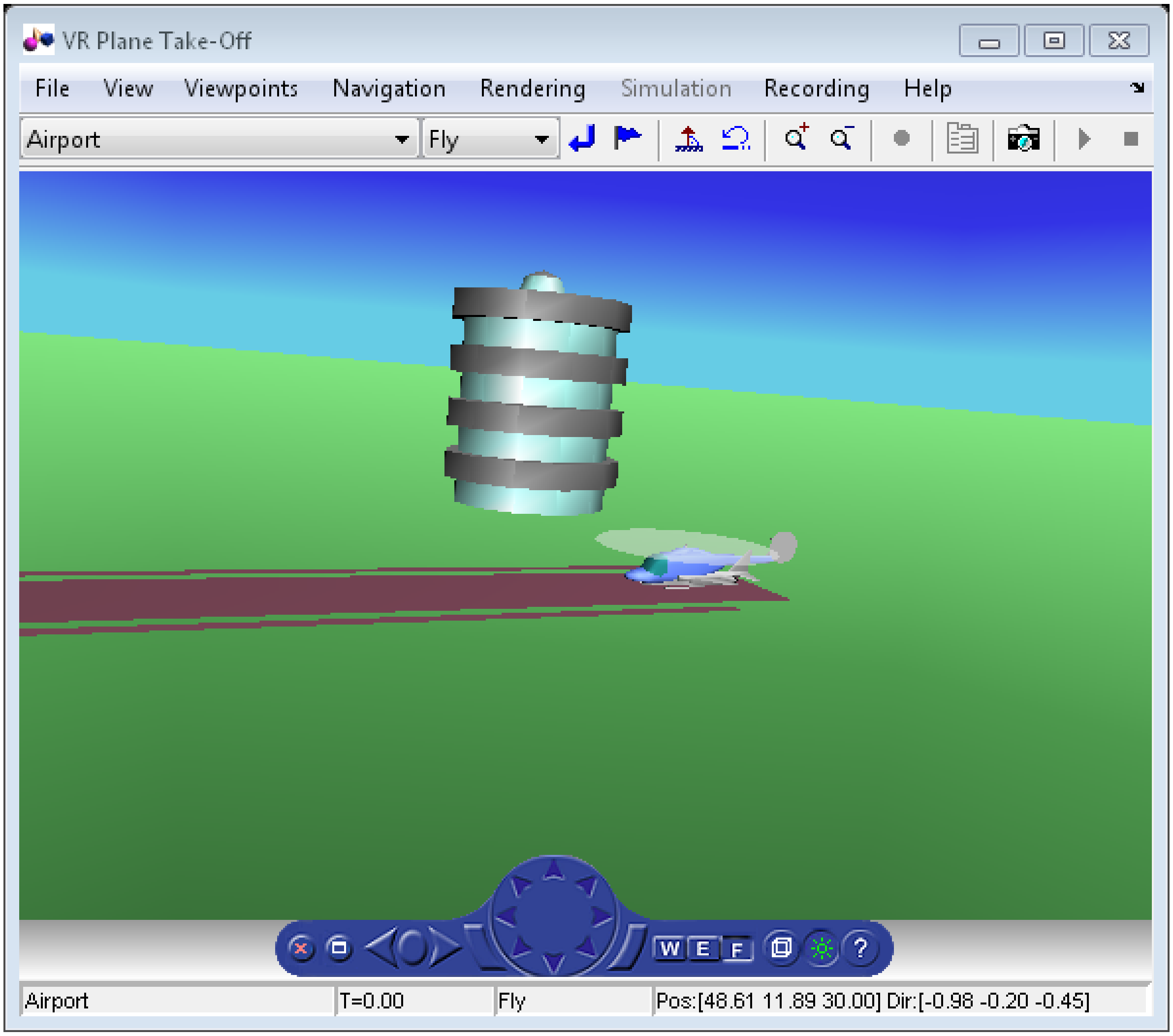
This code sets data properties for the chase helicopter. The `'TimeseriesSourceType'` is the default `'Array6DoF'`, so no additional property changes are needed. The same coordinate transform function (`vranimCustomTransform`) is used for this node as the preceding node. The previous call to `nodeInfo` returned the node index (2).

```
load chaseData
h.Nodes{idxLynx}.TimeseriesSource = chaseData;
h.Nodes{idxLynx}.CoordTransformFcn = @vranimCustomTransform;
```

Create New Viewpoint

This code uses the `addViewpoint` method to create a new viewpoint named 'chaseView'. The new viewpoint will appear in the viewpoint pulldown menu in the virtual reality window as "View From Helicopter". Another call to `nodeInfo` shows the newly added node objects. The node is created as a child of the chase helicopter.

```
h.addViewpoint(h.Nodes{idxLynx}.VRNode,'children','chaseView','View From Helicopter');
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F

Airport | T=0.00 | Fly | Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.45]

Play Animation

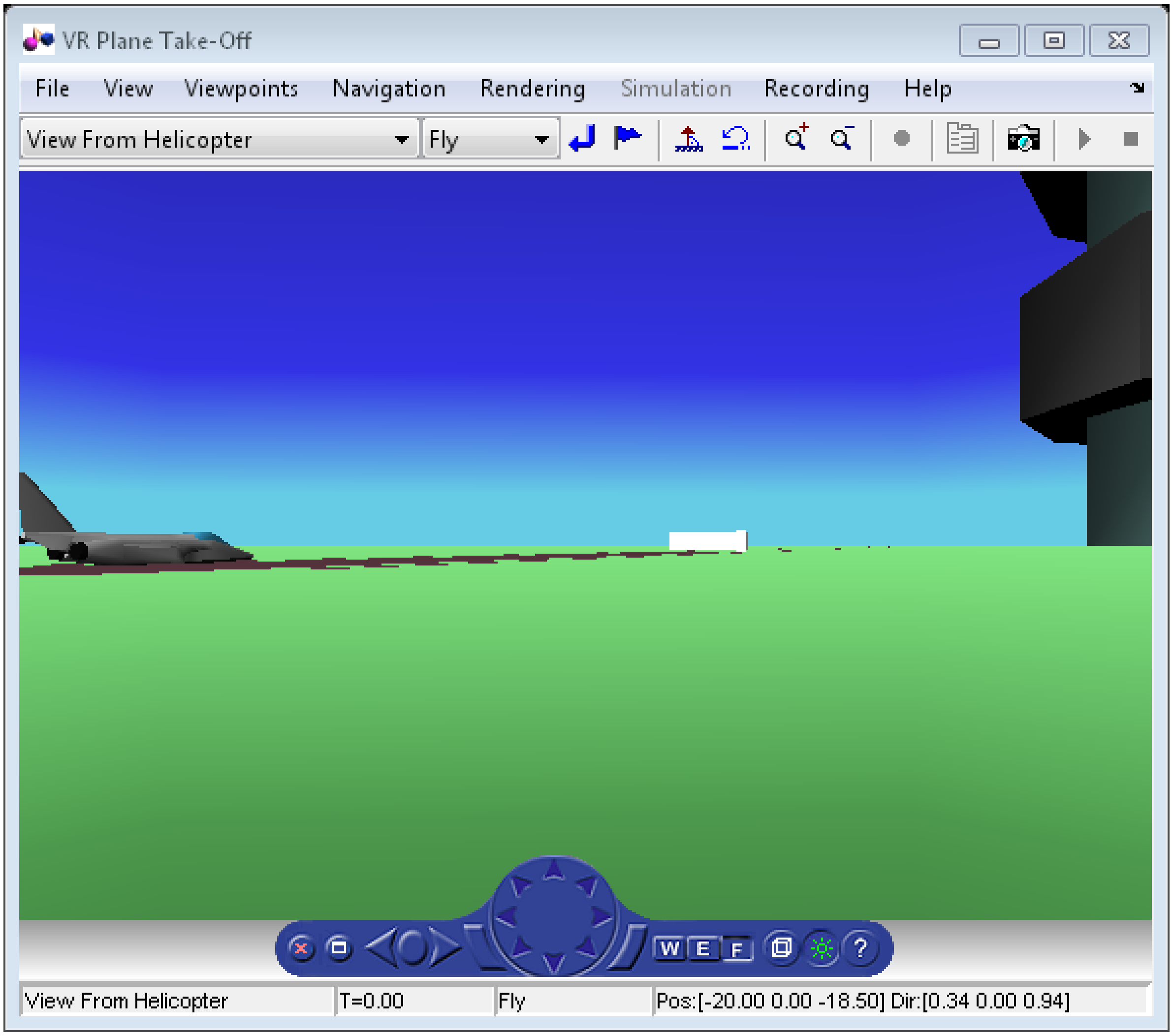
The `play` method runs the simulation for the specified timeseries data.

```
h.play();
```

Play Animation From Helicopter

This code sets the orientation of the viewpoint via the vrnode object associated with the node object for the viewpoint. In this case, it will change the viewpoint to look out the left side of the helicopter at the plane.

```
[~, idxChaseView] = find(strcmp('chaseView',h.nodeInfo));
h.Nodes{idxChaseView}.VRNode.orientation = [0 1 0 convang(200,'deg','rad')];
set(h.VRFigure,'Viewpoint','View From Helicopter');
```

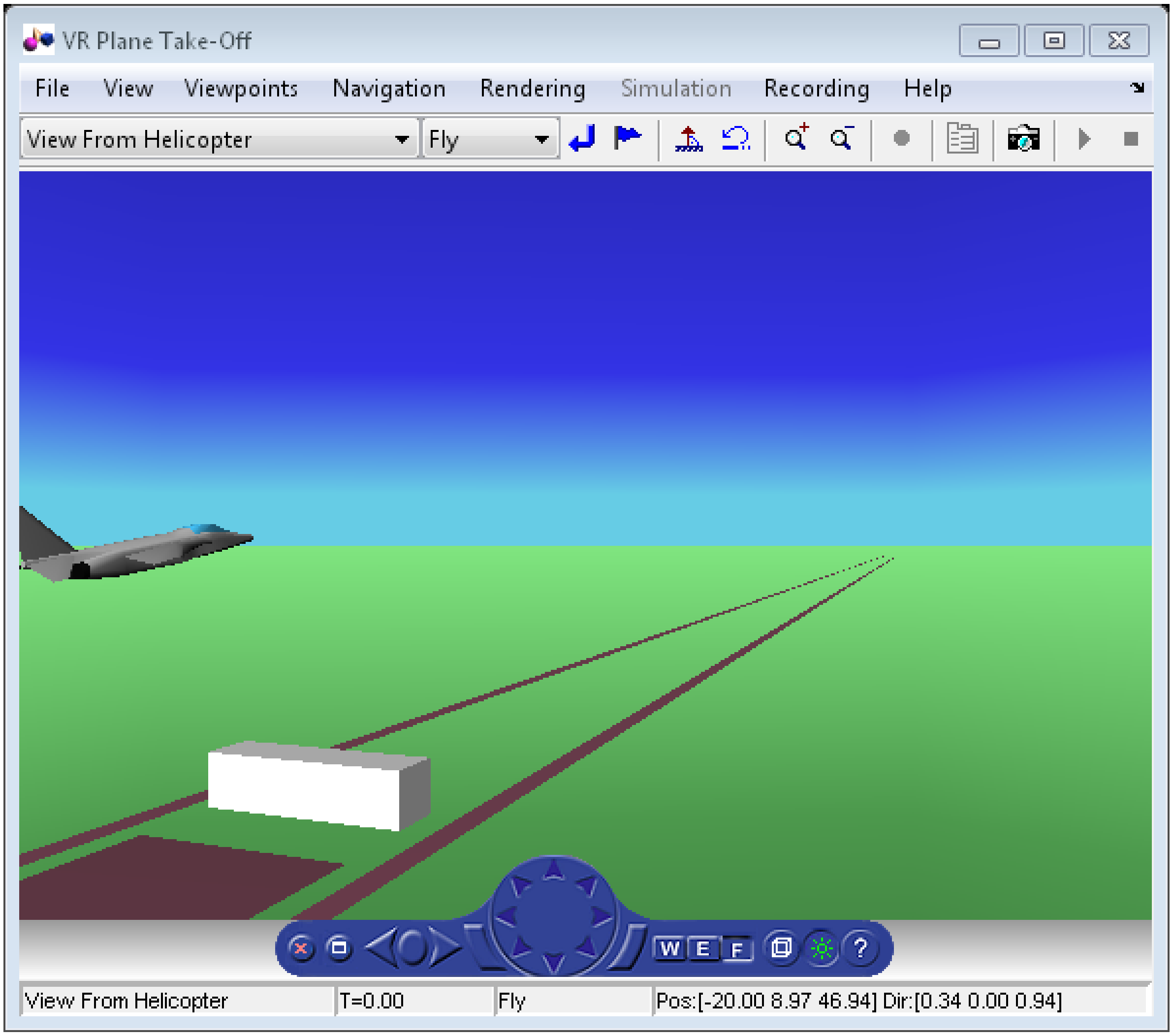


View From Helicopter T=0.00 Fly Pos:[-20.00 0.00 -18.50] Dir:[0.34 0.00 0.94]

Add ROUTE

This code calls the `addRoute` method to add a ROUTE command to connect the plane position to the Camera1 node. This will allow for the "Ride on the Plane" viewpoint to function as intended.

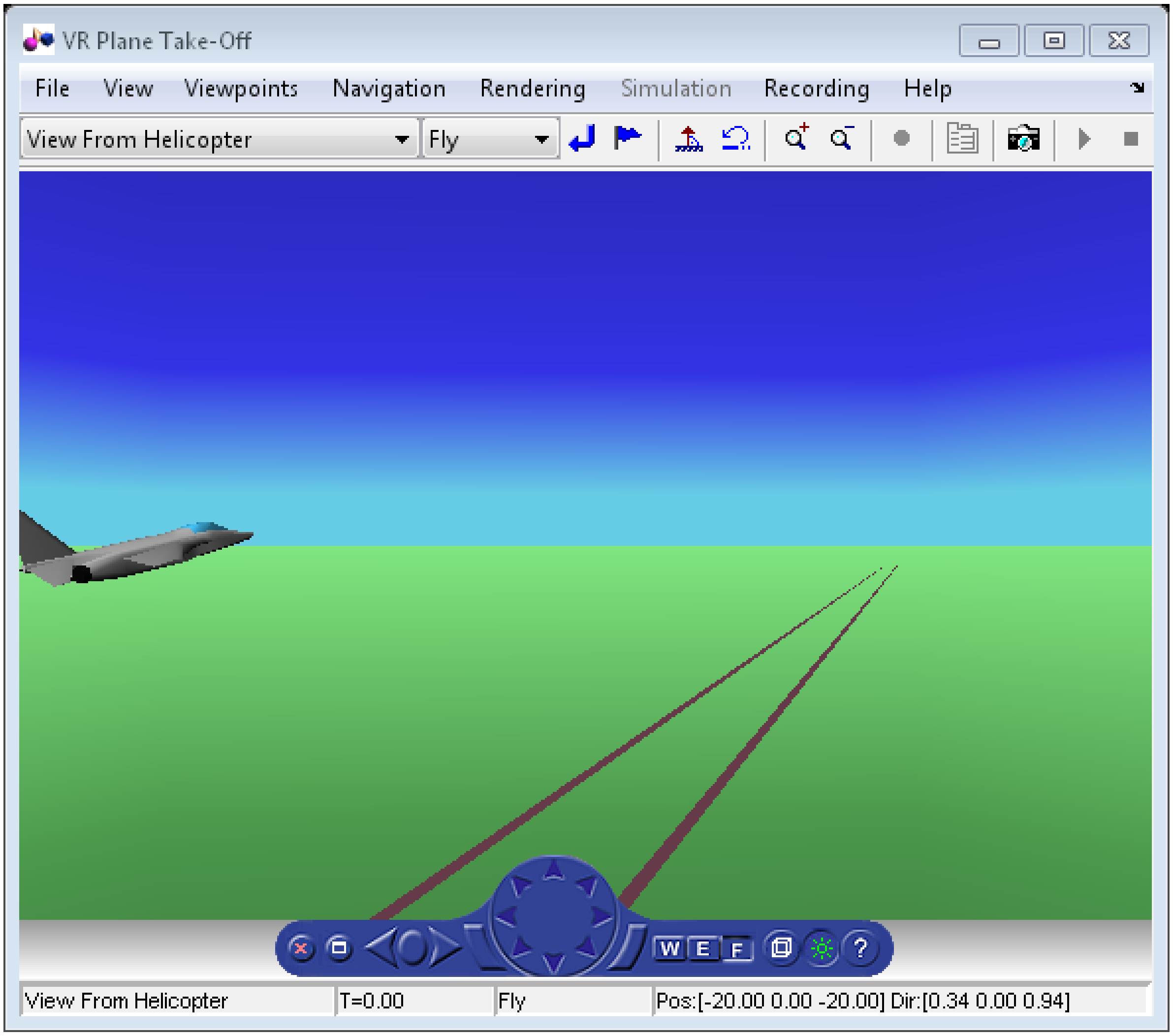
```
h.addRoute('Plane','translation','Camera1','translation');
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
View From Helicopter
Fly
W E F
View From Helicopter
T=0.00
Fly
Pos:[-20.00 8.97 46.94] Dir:[0.34 0.00 0.94]

The scene from the helicopter viewpoint

This code plays the animation.

```
h.play();
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
View From Helicopter
Fly
W E F

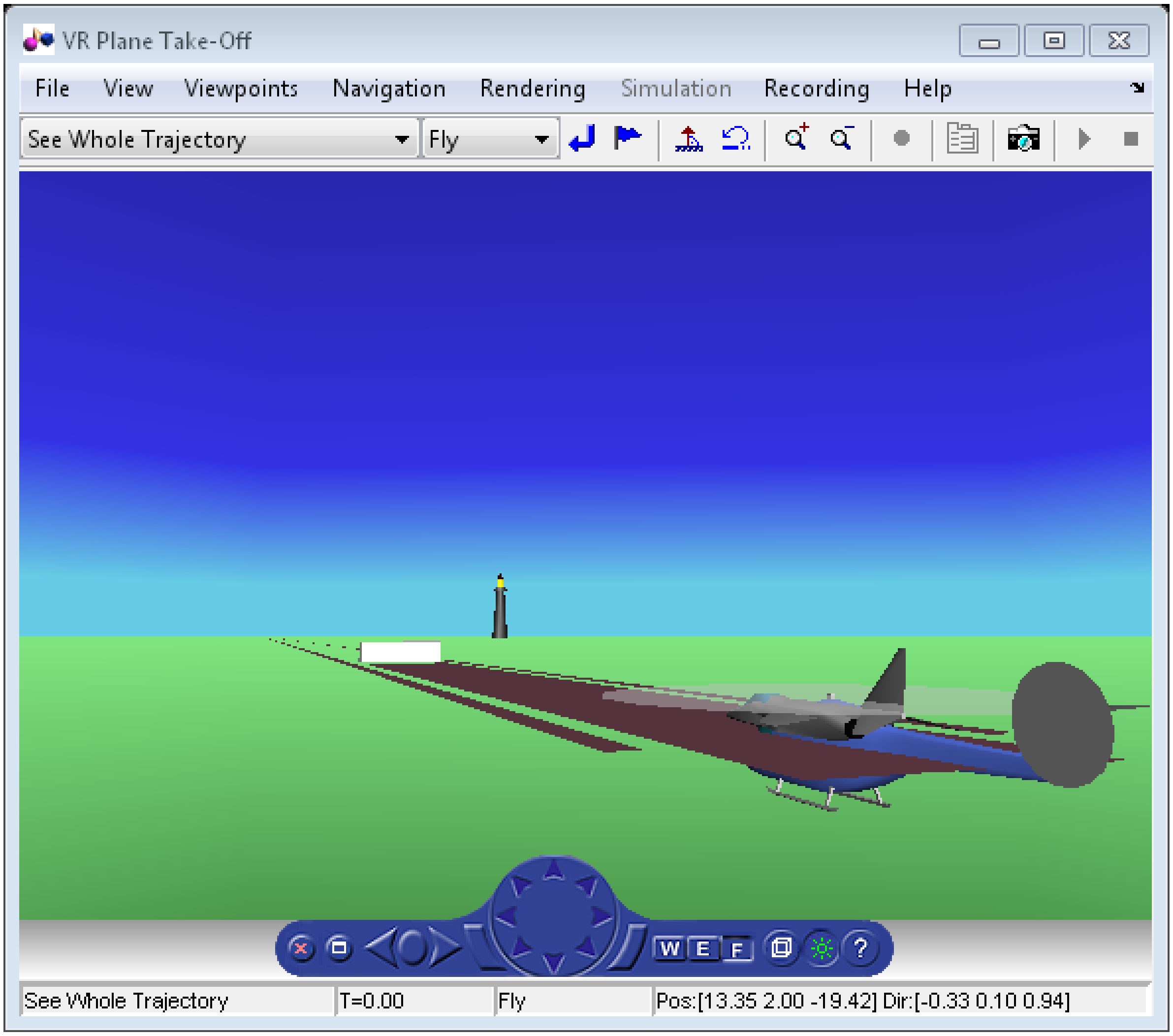
View From Helicopter | T=0.00 | Fly | Pos:[-20.00 0.00 -20.00] Dir:[0.34 0.00 0.94]

Add Another Body

This code adds another helicopter to the scene. It also changes to another viewpoint to view all three bodies in the scene at once.

```
set(h.VRFigure,'Viewpoint','See Whole Trajectory');
h.addNode('Lynx1',[matlabroot,'/toolbox/aero/astdemos/chaseHelicopter.wrl']);
h.nodeInfo
```

```
Node Information
1    Camera1
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
9    Lynx
10    Lynx_Inline
11    chaseView
12    Lynx1
13    Lynx1_Inline
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
See Whole Trajectory
Fly
W E F

See Whole Trajectory | T=0.00 | Fly | Pos:[13.35 2.00 -19.42] Dir:[-0.33 0.10 0.94]

Adjust newly added helicopter to sit above runway.

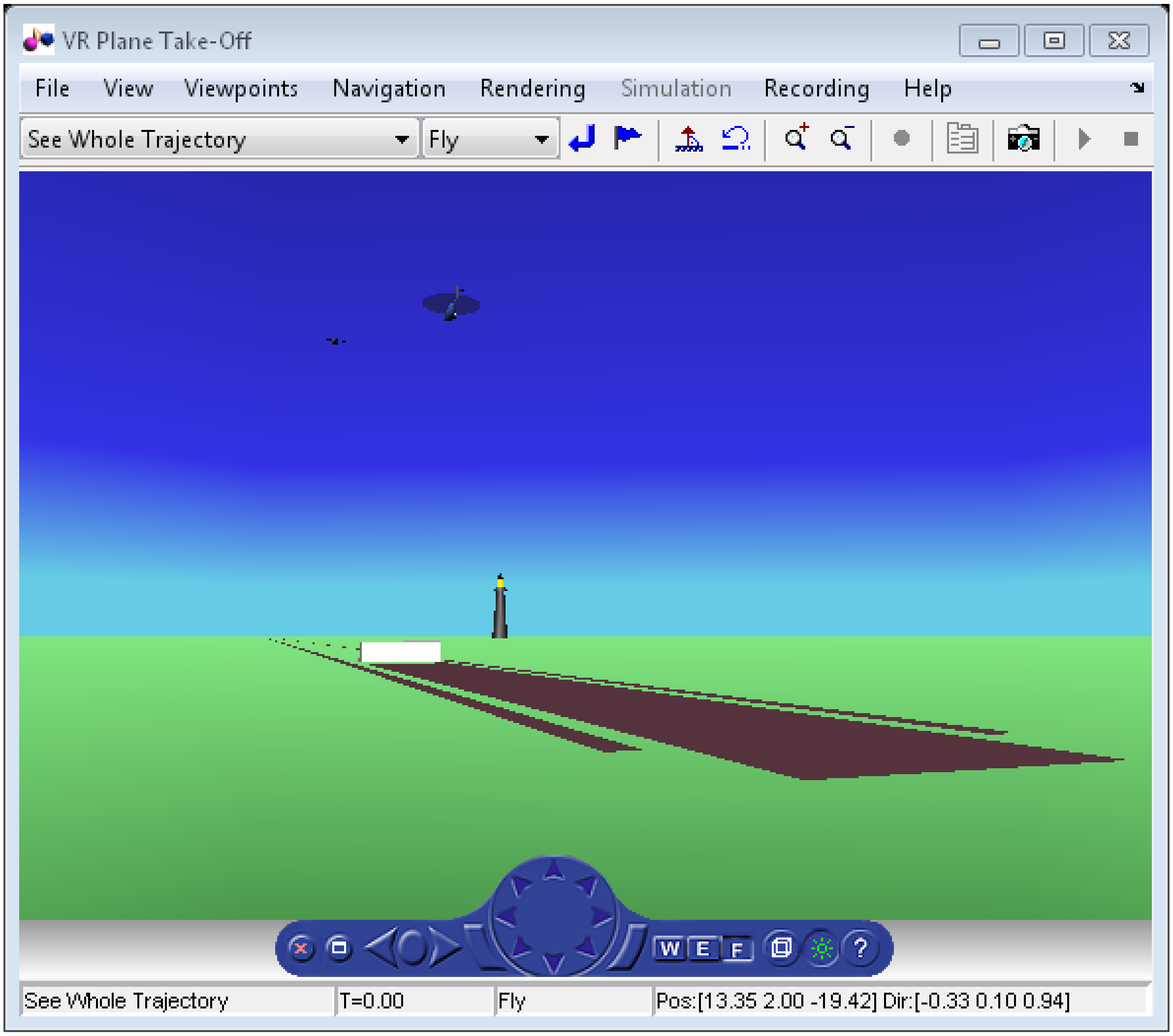
```
[~, idxLynx1] = find(strcmp('Lynx1',h.nodeInfo));
h.Node{idxLynx1}.VRNode.translation = [0 1.3 0];
```

Remove Body

This code uses the `removeNode` method to remove the second helicopter. `removeNode` takes either the node name or node index (as obtained from `nodeInfo`). The associated inline node is removed as well.

```
h.removeNode('Lynx1');
h.nodeInfo
```

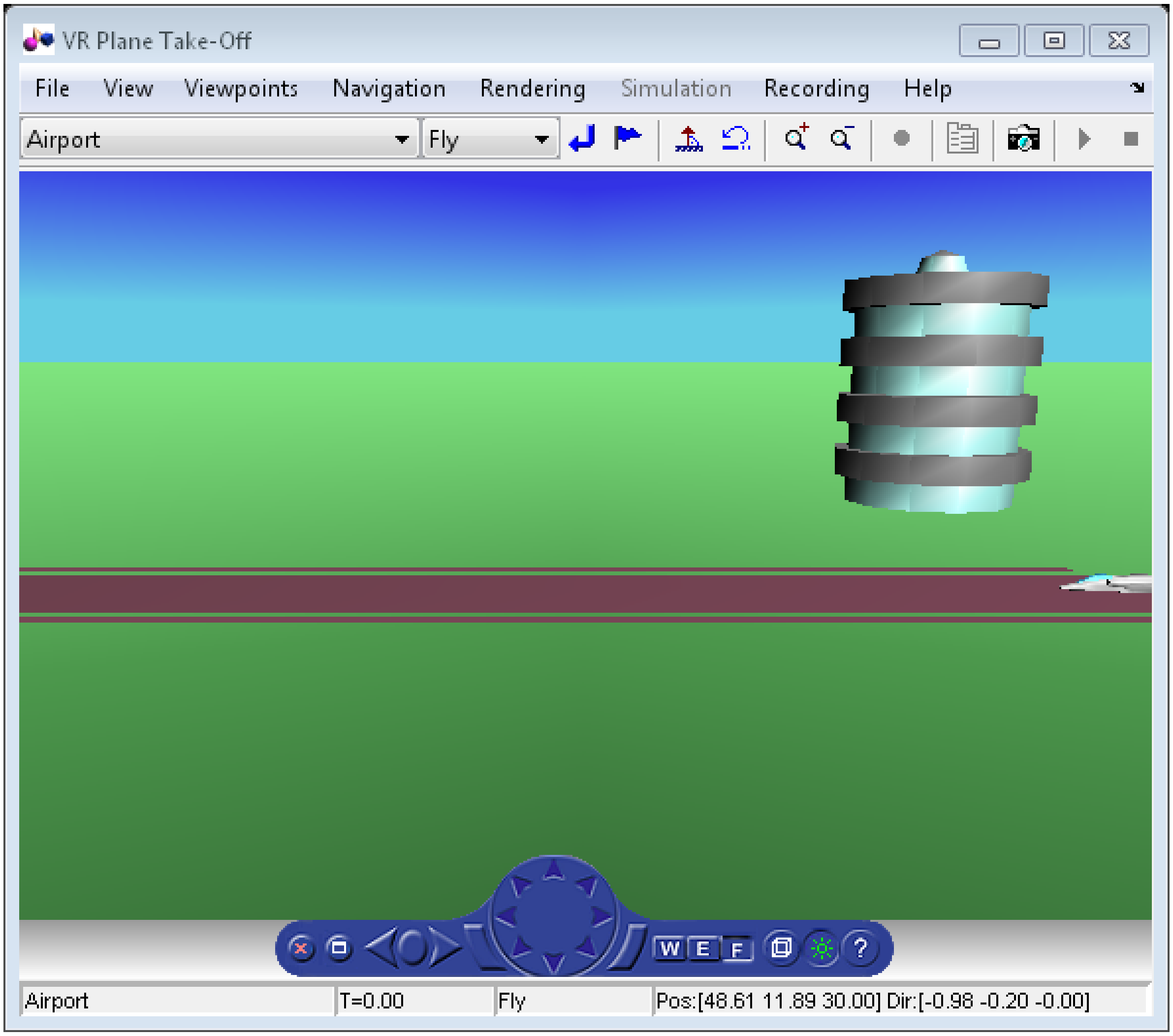
```
Node Information
1    Camera1
2    Plane
3    _V2
4    Block
5    Terminal
6    _v3
7    Lighthouse
8    _v1
9    Lynx
10    Lynx_Inline
11    chaseView
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
See Whole Trajectory
Fly
W E F
See Whole Trajectory
T=0.00
Fly
Pos:[13.35 2.00 -19.42] Dir:[-0.33 0.10 0.94]

Revert To Original World

The original filename is stored in the `'VRWorldOldFilename'` property of the animation object. To bring up the original world, set `'VRWorldFilename'` to the original name and reinitializing it.

```
h.VRWorldFilename = h.VRWorldOldFilename{1};
h.initialize();
```


VR Plane Take-Off
File View Viewpoints Navigation Rendering Simulation Recording Help
Airport
Fly
W E F

Airport | T=0.00 | Fly | Pos:[48.61 11.89 30.00] Dir:[-0.98 -0.20 -0.00]

Close and Delete World

To close and `delete`

```
h.delete();
```

Calculating Compressor Power Required in a Supersonic Wind Tunnel

This example shows how to calculate the required compressor power in a supersonic wind tunnel.

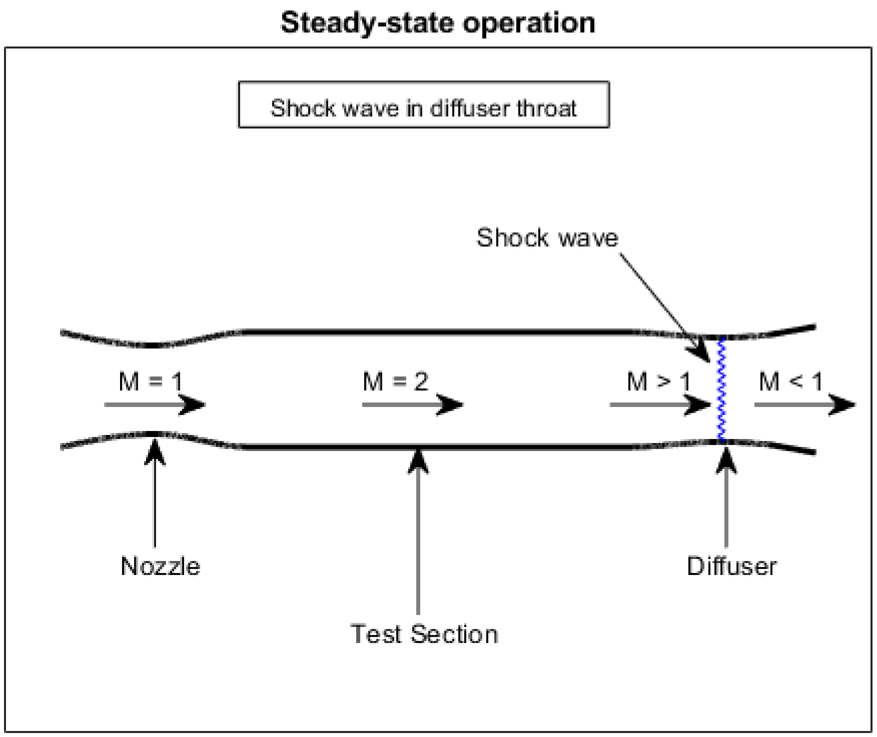
Problem Definition

This section describes the problem to be solved. It also provides necessary equations and known values.

Calculate how much compressor power is required to run a fixed geometry supersonic wind tunnel at steady-state and startup to simulate operating conditions of Mach 2 flow at an altitude of 20 kilometers.

The test section is circular with a diameter of 25 centimeters. After the test section is a fixed-area diffuser. The wind tunnel uses a cooler to reject extra energy that is added to the system by the compressor. Therefore, the compressor inlet and the test section have the same stagnation temperature. Assume the compressor is isentropic and friction effects are negligible.

```
steadyPicture = astsswtschematic('steady');
```



The given information in the problem is:

```
diameter = 25/100;  % Diameter of the cross-section [m]
height   = 20e+03;    % Design altitude [m]
testMach = 2.0;     % Mach number in the test section [dimensionless]
```

The fluid is assumed to be air and therefore it has the following properties.

```
k  = 1.4;        % Specific heat ratio [dimensionless]
cp = 1.004;     % Specific heat at constant pressure [kJ / (kg * K)]
```

The cross-section area of the test section is needed from the diameter.

```
testSectionArea = pi * (diameter)^2 / 4 ; % [m^2]
```

Because the design altitude is given, solve for the flight conditions at that altitude. The Aerospace Toolbox has several functions to calculate the conditions at various altitudes. One such function, `atmosisa`, uses the International Standard Atmosphere to calculate the flight conditions on the left hand side given an altitude input:

```
[testSectionTemp, testSectionSpeedOfSound, testSectionPressure, testSectionDensity] = a
```

This function uses the following units:

```
testSectionTemp = Static temperature in the test section        [K]
testSectionSpeedOfSound = Speed of sound in the test section    [m / s]
testSectionPressure = Static pressure in the test section       [kPa]
testSectionDensity = Density of the fluid in the test section   [kg / m^3]
```

Calculation of the Stagnation Quantities

You must calculate many of the stagnation (total) quantities in the test section. The ratios of local static conditions to the stagnation conditions can be calculated with `flowisentropic`.

```
[~,tempRatioIsen, presRatioIsen, ~, areaRatioIsen] = flowisentropic(k, testMach);
```

All of the left hand side quantities are dimensionless ratios. Now we can use the ratio of static temperature to stagnation temperature to calculate the stagnation temperature.

```
testSectionStagTemp = testSectionTemp / tempRatioIsen;
```

The optimum condition for steady-state operation of a supersonic wind tunnel with a fixed-area diffuser occurs when a normal shock is present at the diffuser throat. For optimum condition, the area of the diffuser throat must be smaller than the area of the nozzle throat. Assuming a perfect gas with constant specific heats, calculate the factor by which the diffuser area must be smaller than the nozzle area. This calculation is from a simplified form of the conservation of mass equation involving total pressures and cross-sectional areas:

$$p_{t_{nozzle}} A^*_{nozzle} = p_{t_{diffuser}} A^*_{diffuser}$$

where

$$p_{t_{nozzle}} = Total\ pressure\ at\ the\ nozzle$$

$$p_{t_{diffuser}} = Total\ pressure\ at\ the\ diffuser$$

$$A^*_{nozzle} = Reference\ area\ for\ sonic\ flow\ at\ the\ nozzle$$

$$A^*_{diffuser} = Reference\ area\ for\ sonic\ flow\ at\ the\ diffuser$$

Rearrange the equation:

$$\frac{A^*_{diffuser}}{A^*_{nozzle}} = \frac{p_{t_{nozzle}}}{p_{t_{diffuser}}}$$

This example assumes the nozzle throat area, the test section, and the region of flow at the diffuser throat before the shock to be upstream. Because the shock wave is at the throat of the diffuser, the diffuser throat area can be considered either upstream or downstream of the shock. This example assumes the diffuser throat area to be downstream. Since the upstream flow is isentropic until the shock wave, you can use the test section Mach number as the upstream Mach number. Doing this enables you to calculate the total pressure ratio through the shock and then the area ratio between the nozzle and the diffuser area.

The total pressure ratio is:

$$stagPressRatio = \frac{p_{t_{diffuser}}}{p_{t_{nozzle}}}$$

Calculate the total pressure ratio using the normal shock function from the Aerospace Toolbox:

```
[~, ~, ~, ~, ~, stagPressRatio] = flownormalshock(k, testMach);
```

The area ratio at the shock is:

$$areaRatioShock = \frac{A^*_{nozzle}}{A^*_{diffuser}}$$

We have the following expression using the conservation of mass as discussed previously.

```
areaRatioShock = stagPressRatio;
```

Calculate the area of the diffuser:

```
diffuserArea = testSectionArea / (areaRatioShock * areaRatioIsen);
```

Because the diffuser throat area is smaller than the test section area, the Mach number of the flow must converge toward unity. Using `flowisentropic` with the area ratio as the input, calculate the Mach number just upstream of the shock:

```
diffuserMachUpstreamOfShock = flowisentropic(k, (1 / areaRatioShock), 'sup');
```

Use `flownormalshock` to calculate the flow properties through the shock wave. Note, here again, we will only need the total pressure ratio:

```
[~, ~, ~, ~, ~, P0] = flownormalshock(k, diffuserMachUpstreamOfShock);
```

Calculation of Work and Power Required for the Steady-State Case

The work done by the compressor per unit mass of fluid equals the enthalpy change through the compressor. From the definition of enthalpy, calculate the specific work done by knowing the temperature change and the specific heat of the fluid at constant pressure:

$$specificWork = h_{out} - h_{in} = c_p(T_{out} - T_{in})$$

For an isentropic compressor,

$$\frac{T_{out}}{T_{in}} = \left(\frac{p_{out}}{p_{in}}\right)^{\frac{k-1}{k}}$$

Rearrange the above equation to solve for the temperature difference. Recall that the temperature into the compressor is the same as the test section stagnation temperature.

$$T_{out} - T_{in} = T_{in}\left[\left(\frac{p_{out}}{p_{in}}\right)^{\frac{k-1}{k}} - 1\right]$$

```
tempDiff = testSectionStagTemp * ((1 / P0)^((k - 1) / k) - 1); % [K]
```

Now the specific work can be found.

```
specificWork = cp * tempDiff; % [kJ / kg]
```

The power required equals the specific work times the mass flow rate. During steady-state operation, the mass flow rate through the test section is given by:

$$\dot{m} = \rho A V = \rho A M a$$

where all flow quantities are the values in the test section:

$$\dot{m} = Mass\ flow\ rate$$

$$\rho = Density$$

$$A = Cross - sectional\ area\ of\ test\ section$$

$$V = Velocity$$

$$M = Mach\ number$$

$$a = Local\ speed\ of\ sound$$

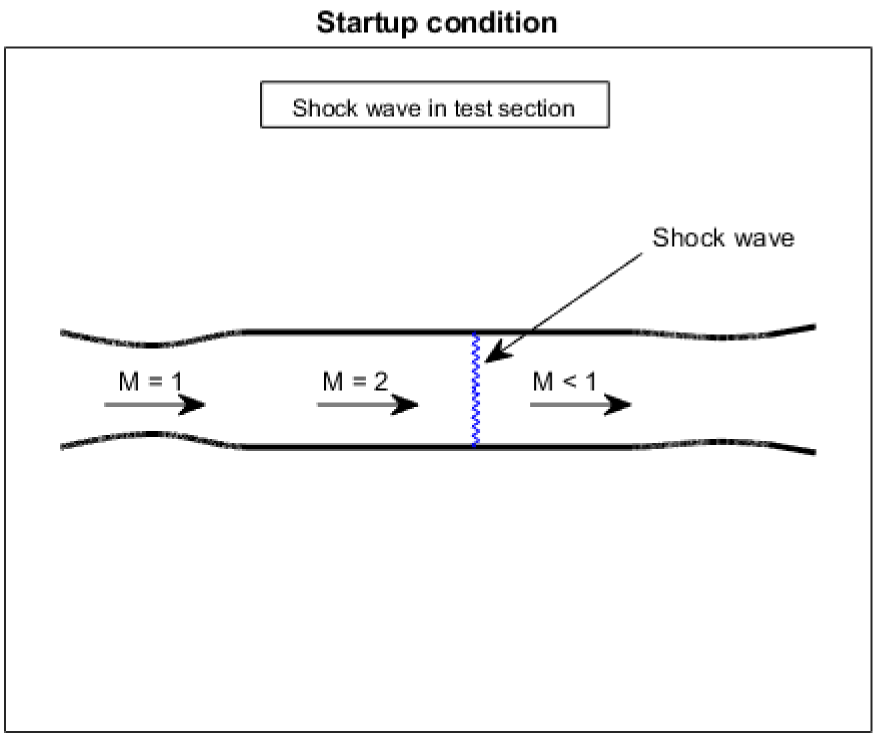
```
massFlowRate = testSectionDensity * testSectionArea * testMach * testSectionSpeedOfSour
```

Finally, calculate the power required by the compressor during steady-state operation.

```
powerSteadyState = specificWork * massFlowRate;  % [kW]
```

Calculating Work and Power Required During Startup

```
startupPicture = astsswtschematic('startup');
```



For the startup condition the shock wave is in the test section. The Mach number immediately before the shock wave is the test section Mach number.

```
[~, ~, ~, ~, ~, stagPressRatioStartup] = flownormalshock(k, testMach);
```

Now, calculate the specific work of the isentropic compressor.

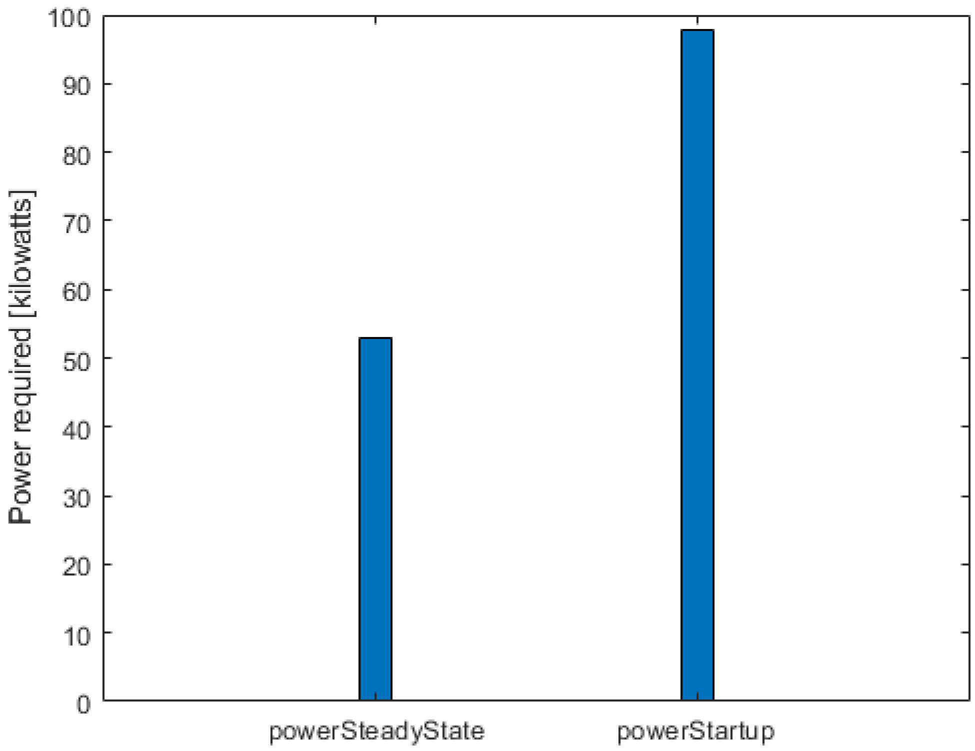
```
specificWorkStartup = cp * testSectionStagTemp * ((1 / stagPressRatioStartup)^((k - 1)
```

Then, calculate the power required during startup:

```
powerStartup = specificWorkStartup * massFlowRate;    % [kW]
```

The power required during steady-state operation (53.1 kW) is much lower than that required by the compressor during startup (97.9 kW) These power required results represent the optimum and worst-case operation conditions, respectively.

```
power = [powerSteadyState powerStartup];
barGraph = figure('name','barGraph');
bar(power,0.1);
ylabel('Power required [kilowatts]')
set(gca,'XTickLabel',{'powerSteadyState', 'powerStartup'})
```



```
close(steadyPicture, startupPicture, barGraph)
```

Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

Analyzing Flow with Friction Through an Insulated Constant Area Duct

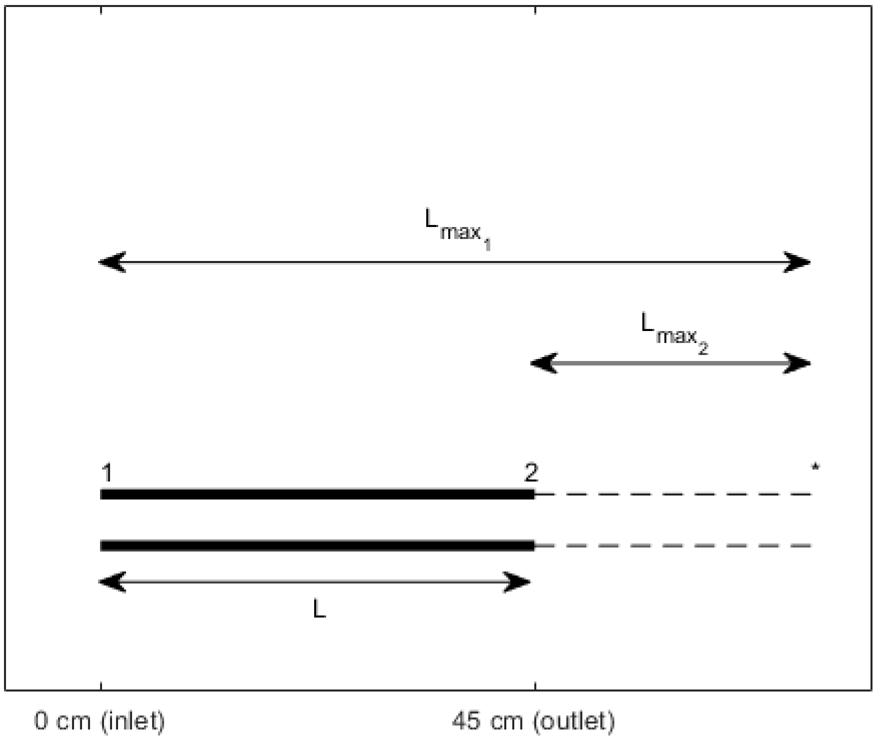
This example shows how to implement a steady, viscous flow through an insulated, constant-area duct using the Aerospace Toolbox™ software. This flow is also called Fanno line flow.

Problem Definition

This section describes the problem to be solved. It also provides necessary equations and known values.

Fanno line flow is the modeling of perfect gas flow through a constant-area duct that does not change with time and which is adiabatic. Wall friction is the main mechanism for the change of the flow variables. This example looks at Fanno flow of air entering a 3 centimeter diameter pipe that is 45 centimeters long at a Mach number of 0.6. The conditions at the inlet, also called station 1, are static pressure of 150 kilopascals and static temperature of 300 Kelvin. The duct is assumed to have a coefficient of friction of 0.02. Calculate the Mach number, static pressure and static temperature at the exit of the duct or station 2.

```
ductPicture = astfrictionduct;
```



The given information in the problem is:

```
ductLength       = 0.45;    % Length of the duct [m]
diameter         = 0.03;    % Diameter of the duct [m]
inletMach        = 0.6;     % Mach number at the duct inlet [dimensionless]
inletPressure    = 150;     % Static pressure at the duct inlet [kPa]
inletTemperature = 300;     % Static temperature at the duct input [K]
frictionCoeff    = 0.02;    % Duct friction coefficient [dimensionless]
```

The fluid is air which has the following specific heat ratio.

```
k = 1.4;    % Specific heat ratio [dimensionless]
```

Understanding the Fanno Parameter

The Fanno parameter is a dimensionless quantity that indicates how much influence friction will have while the fluid is flowing through the duct. For a given duct, the Fanno parameter is defined as

$$Fanno\ parameter = \frac{fL}{D_h}$$

Where

$$L = Length\ of\ the\ duct$$

$$D_h = \frac{4 * Cross - sectional\ area}{Perimeter\ of\ cross - section} = Hydraulic\ diameter$$

For the circular pipe, assume the hydraulic diameter is the inner diameter of the pipe. The friction coefficient, f, is given by the following expression:

$$f = \frac{4\tau_f}{\frac{1}{2}\rho V^2}$$

where

$$\tau_f = Shear\ stress\ due\ to\ wall\ friction$$

$$\rho = Density$$

$$V = Velocity$$

Note, this example uses the convention in which a factor of four is not visible in the Fanno parameter. This convention defines the friction coefficient as four times the skin friction over the dynamic pressure. Friction will have a greater effect on the flow in a long duct than in a short duct because the flow is impeded by more wall surface friction. Additionally, friction is more dominant when the duct is narrow. This is because the boundary layer affects a larger portion of the flow along the walls than when the duct width is large.

Friction is an energy loss that generates entropy (irreversibility) in the system. The increase in entropy causes the flow to tend towards the choked condition (Mach = 1). The choked condition occurs if the length of the duct is long enough. For a given Mach number and specific heat ratio, the reference Fanno parameter for choked flow is

$$\frac{fL_{max}}{D_h} = \frac{1-M^2}{\gamma M^2} + \frac{\gamma+1}{2\gamma} ln\left(\frac{M^2}{\frac{2}{\gamma+1}[1+\frac{\gamma-1}{2}M^2]}\right)$$

where

$$\gamma = k = Specific\ heat\ ratio$$

Using the Fanno Parameter and FLOWFANNO to Solve for the Flow Properties at the Inlet

This example provides the length of the duct, the diameter of the duct, and the friction coefficient. Therefore, the actual Fanno parameter of the duct can be computed as follows:

```
fannoParameter = frictionCoeff * ductLength / diameter;
```

This example also provides the Mach number and specific heat ratio. This enables you to calculate the reference Fanno parameter for the inlet condition, the inlet temperature ratio and inlet the pressure ratio. Use the `flowfanno` function from the Aerospace Toolbox:

```
[~, inletTempRatio, inletPresRatio, ~, ~, ~, inletFannoRef] = flowfanno(k, inletMach);
```

$$inletTempRatio = \frac{T_1}{T_1^*}$$

$$inletPresRatio = \frac{p_1}{p_1^*}$$

$$inletFannoRef = \left(\frac{fL_{max}}{D_h}\right)_1$$

where:

- The subscript indicates the flow station.
- Unstarred quantities are the local values of the given variables.

- Starred quantities are referenced value of the given variables if the flow is brought to the choked condition.

The length of the inlet reference Fanno parameter, also called inlet max length, is the length that the pipe needs to be to have choked flow for a given inlet condition. If the actual length is less than the inlet max length, an extension to the pipe is needed for choked flow. This choked flow corresponds to a reference Fanno parameter for the outlet. Since the diameter and friction coefficient are given in the problem statement, only the lengths vary in the following equation for the outlet reference Fanno parameter:

$$\left(\frac{fL_{max}}{D_h}\right)_2 = \left(\frac{fL_{max}}{D_h}\right)_1 - \frac{fL}{D_h}$$

```
outletFannoRef = inletFannoRef - fannoParameter;
```

Calculating the Flow Properties at the Outlet with the FLOWFANNO Function

Next, use `flowfanno` to calculate the flow ratios at the outlet station. The third input, 'fannosub', indicates that the second input, outletFannoRef, is a subsonic Fanno parameter input.

```
[outletMach, outletTempRatio, outletPresRatio] = flowfanno(k, outletFannoRef, 'fannosub
```

$$outletTempRatio = \frac{T_2}{T_2^*}$$

$$outletPresRatio = \frac{p_2}{p_2^*}$$

Use the temperature ratios found at the inlet and outlet to calculate the temperature and pressure at the outlet. The reference conditions are the same at both stations because the duct is insulated. In addition, assume that the effects of friction act on both stations in the same manner. As a result, we have

$$T_1^* = T_2^*$$

$$p_1^* = p_2^*$$

Therefore, the temperature at the outlet and the pressure at the outlet are

$$T_2 = T_1 \frac{T_1^*}{T_1} \frac{T_2}{T_2^*}$$

```
outletTemperature = inletTemperature / inletTempRatio * outletTempRatio;
```

$$p_2 = p_1 \frac{p_1^*}{p_1} \frac{p_2}{p_2^*}$$

```
outletPressure = inletPressure / inletPresRatio * outletPresRatio;
```

The values that we wanted to calculate are

```
outletMach          % [dimensionless]
outletTemperature   % [K]
outletPressure      % [kPa]
```

```
outletMach =

    0.7093


outletTemperature =

  292.2018


outletPressure =

  125.2332
```

For Fanno line flow where the inlet flow is subsonic, the temperature and pressure always decrease through the duct. For all Fanno line flow cases the Mach number moves closer to one.

```
close(ductPicture)
```

Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

```
%#ok<*NOPTS>
```

Determining Heat Transfer and Mass Flow Rate in a Ramjet Combustion Chamber

This example shows how to use the Aerospace Toolbox™ functions to determine heat transfer and mass flow rate in a ramjet combustion chamber.

The Ramjet Engine

When calculating the thrust of a ramjet engine, it is important to optimize the amount of heat added and the mass flow rate through an air breathing engine. This optimization is important because the thrust generated by the engine is governed by these parameters. The ramjet thrust equation is the following:

$$Thrust = \dot{m}_{exit}V_{exit} - \dot{m}_{enter}V_{enter} + (p_{exit} - p_{enter})A_{exit}$$

where

$$\dot{m} = Mass\ flow\ rate\ [kg/s]$$

$$V = Velocity\ [m/s]$$

$$p = pressure\ [kPa]$$

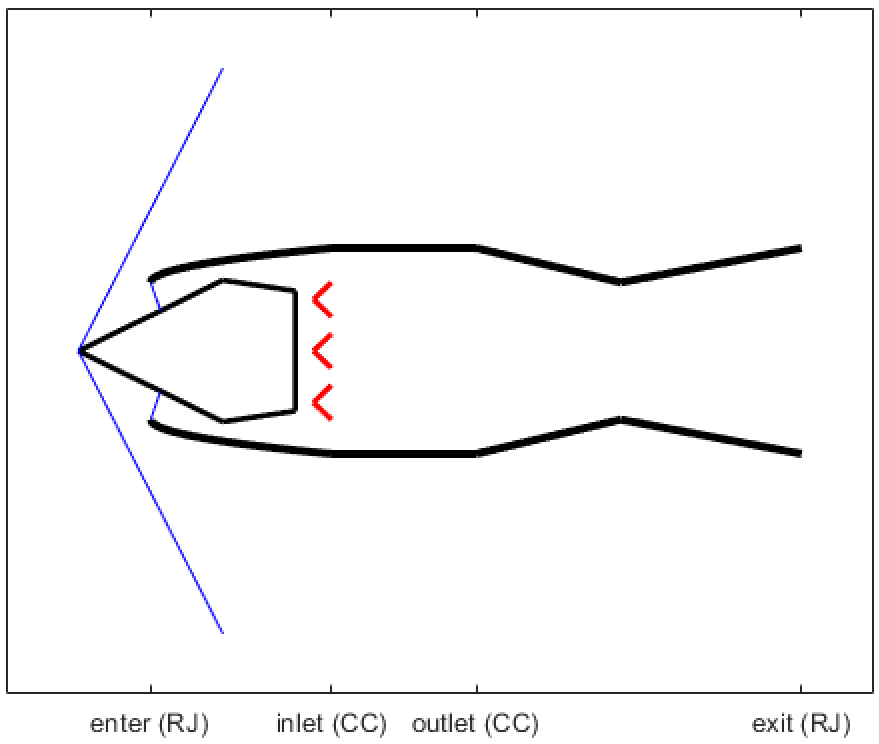
$$A = Cross - sectional\ area\ [m^2]$$

In the ramjet thrust equation, the subscripts denote the location of the parameter.

- enter - Denotes the entrance of the entire ramjet.
- exit - Denotes the exit of the ramjet engine.
- inlet - Used for the beginning of the combustion chamber.
- outlet - Used for the end of the combustion chamber.

This difference is illustrated in the following figure, (RJ) stands for the entire ramjet engine and (CC) refers to the combustion chamber.

```
ramjetPicture = astramjet;
```



Note, the thrust equation takes directly into account the mass flow rate. Heat addition correlates to higher exit velocity from the energy equation; higher exit velocity means more thrust. Modeling the ramjet combustion chamber as a constant area duct where heat addition is the main driver for the change in the flow variables enables the use of Rayleigh line flow principles.

Problem Definition

This section describes the problem to be solved. It also provides necessary equations and known values.

After a series of shock waves, flow enters the combustion at a velocity of 100 m/s and static temperature of 400K. We want to:

- Maximize the amount of heat added in the combustion chamber without decreasing the mass flow rate.
- Calculate the fuel-air ratio associated with the maximum allowable heat added.

The heating value of the fuel is 40 megajoules per kilogram and the mass of the fuel is negligible compared to the mass of the air. We assume that the working fluid behaves like a perfect gas with constant specific heat ratio and specific heat at constant pressure given as:

$$\gamma = k = Specific\ heat\ ratio = 1.4\ [dimensionless]$$

$$c_p = Specific\ heat\ at\ constant\ pressure = 1.004\ [kJ/(kgK)]$$

Given data for problem is listed below.

```
inletVelocity       = 100;      % Velocity of fluid at combustor intake [m/s]
inletTemperature    = 400;      % Temperature of fluid at combustor intake [K]
heatingValue        = 40e+03;   % Heating value of the fuel [kJ/kg]
k                   = 1.4;      % Specific heat ratio [dimensionless]
cp                  = 1.004;    % Specific heat at constant pressure [kJ/(kg*K)]
```

Because the fluid is air, it also has the following gas constant:

```
R = 287; % Gas constant of air [J/(kg*K)]
```

Therefore, the speed of sound is:

```
speedOfSound = sqrt(k * R * inletTemperature); % [m/s]
```

The inlet Mach number is

```
inletMach = inletVelocity/speedOfSound; % [dimensionless]
```

Solving for the Stagnation Quantities and Reference Values

To apply the energy equation in order to find the heat transfer rate, calculate the stagnation temperature at the inlet. Use isentropic flow ratios and the static temperature at that point to calculate this temperature. The `flowisentropic` function calculates the ratio of the static temperature to the total (stagnation) temperature.

$$\frac{T_{inlet}}{T_{t_{inlet}}} = inletTempRatio$$

Where

$$T_{t_{inlet}} = Total\ temperature\ (at\ the\ inlet)$$

```
[~, inletTempRatio, inletPresRatio] = flowisentropic(k, inletMach);
```

With the temperature ratio at the inlet, calculate the total temperature at the inlet. Be careful. Note that the form in which we need the temperature ratio is inverted from the form as given in the `flowisentropic` function.

$$T_{t_{inlet}} = T_{inlet}\frac{T_{t_{inlet}}}{T_{inlet}}$$

```
inletTotalTemp = inletTemperature / inletTempRatio;
```

Use the energy equation to describe the flow in the combustion chamber:

$$\dot{q} = \dot{m}_{air}c_p(T_{t_{outlet}} - T_{t_{inlet}}) = \dot{m}_{fuel} * heatingValue$$

where

$$\dot{q} = rate\ of\ heat\ transfer\ [kW]$$

To maximize the rate of heat transfer, the stagnation temperature at the outlet station must be the reference stagnation temperature:

$$T_{t_{outlet}} = T_t^*$$

Use the `flowrayleigh` function to calculate the total temperature ratio at the inlet. After this, you can calculate the reference total temperature.

```
[~,~,~,~,~,totalTempRatio] = flowrayleigh(k, inletMach);
```

In this equation, note that this ratio is found by the function as the local value over the reference value.

$$\frac{T_{t_{inlet}}}{T_t^*} = totalTempRatio$$

Now calculate the reference total temperature. Note that the total temperature ratio has been inverted to allow the proper cancellation of terms.

$$T_t^* = T_{t_{inlet}} \frac{T_t^*}{T_{t_{inlet}}}$$

```
inletTotalTempRef = inletTotalTemp / totalTempRatio;
```

Calculating the Fuel-to-Air Ratio and Maximum Heat Added

Calculate the fuel-to-air ratio by rearranging the energy equation.

$$\frac{\dot{m}_{fuel}}{\dot{m}_{air}} = \frac{c_p(T_t^* - T_{t_{inlet}})}{heatingValue}$$

```
fuelAirRatio = cp * (inletTotalTempRef - inletTotalTemp) / heatingValue
```

```
fuelAirRatio =

    0.0296
```

The maximum heat added is:

$$q_{max} = c_p * (T_t^* - T_{t_{inlet}})$$

```
heatMax = cp * (inletTotalTempRef - inletTotalTemp)
```

```
heatMax =

   1.1826e+03
```

Accounting for an Increase in the Fuel-Air Ratio

Consider the case where there is a 10% increase in the fuel-air ratio. Calculate how much the mass flow rate decreases with a 10% increase in fuel-to-air ratio, holding the stagnation temperature and pressure constant. The new fuel-air ratio is:

```
fuelAirRatio10 = 1.1 * fuelAirRatio;
```

Note, any variable that ends with "10" indicate that the given value is related to the 10% increase in fuel-to-air ratio. Rearrange the energy equation to calculate the difference in total temperatures from the inlet to the outlet of the combustion chamber:

$$T_{t_{outlet}} - T_{t_{inlet}} = \frac{\frac{\dot{m}_{fuel}}{\dot{m}_{air}} * heatingValue}{c_p}$$

```
totalTempDiff = fuelAirRatio10 * heatingValue / cp;
```

The maximum heating condition is where the flow is choked at the outlet:

$$T_{t_{outlet}} = T_t^*$$

Therefore, inlet reference total temperature and the ratio of total temperature to the reference value are:

$$T_{t_{inlet}}^* = T_{t_{outlet}} - T_{t_{inlet}} + T_t^*$$

$$\frac{T_{t_{inlet}}}{T_{t_{inlet}}^*} = totalTempRatio$$

```
inletTotalTempRef10 = totalTempDiff + inletTotalTemp;
```

```
totalTempRatio10 = inletTotalTemp / inletTotalTempRef10;
```

Calculating the Decrease in Mass Flow Rate

Given the total temperature ratio, `flowrayleigh` calculates the Mach number at the inlet of the combustion chamber:

```
inletMach10 = flowrayleigh(k, totalTempRatio10, 'totaltsub');
```

In this equation, the string input causes the function to use the subsonic total temperature ratio input mode. We know that the flow will be subsonic entering the combustion chamber because the flow will have gone through several shock waves leading up to the combustion chamber. With this Mach number at the inlet, use `flowisentropic` to find the isentropic temperature ratio and pressure ratio at the inlet:

```
[~, inletTempRatio10, inletPresRatio10] = flowisentropic(k, inletMach10);
```

The static temperature at the inlet is:

```
inletTemperature10 = inletTotalTemp * inletTempRatio10;
```

From the equation of state, the mass flow rate can be written as:

$$\dot{m} = \frac{p}{RT} A(M\sqrt{\gamma RT})$$

With a 10% fuel-air ratio increase, relate a ratio showing a decrease in mass flow from a 10% increase in the mass flow rate to the ratio of decreasing Mach number. The increase in pressure ratio contributes to increasing the mass flow rate, but not as much as the decrease in Mach number decreases the mass flow rate. All other variables are constant between the two cases.

$$\frac{\dot{m}_{10}}{\dot{m}} = \frac{M_{10}}{M} \frac{\left(\frac{p_{inlet}}{p_{t_{inlet}}}\right)_{10}}{\frac{p_{inlet}}{p_{t_{inlet}}}}$$

```
massFlowRateRatio = inletMach10 / inletMach * inletPresRatio10 / inletPresRatio;
```

This ratio represents the percentage of the mass flow rate of the case with a 10% increase in fuel-to-air ratio. It uses the original mass flow rate as a whole. The percentage decrease in mass flow rate is just one minus the above ratio (times 100):

```
percentageDecrease = ( 1 - massFlowRateRatio ) * 100 % [percent]
```

```
percentageDecrease =

    3.7665
```

These results show that adding fuel to the fuel-air mixture decreases the mass flow rate. This in turn makes the thrust decrease. This means that once a certain amount of fuel is added to the combustion chamber, adding more produces an inefficient result. Therefore, preemptive calculations such as these help engineers maximize fuel efficiency around the design conditions of an engine.

```
close(ramjetPicture)
```

Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

```
%#ok<*NOPTS>
```

Solving for the Exit Flow of a Supersonic Nozzle

This example shows how to use the method of characteristics and Prandtl-Meyer flow theory to solve a problem in supersonic flow involving expansions. Solve for the flow field downstream of the exit of a supersonic nozzle.

Problem Definition

This section describes the problem to be solved. It also provides necessary equations and known values.

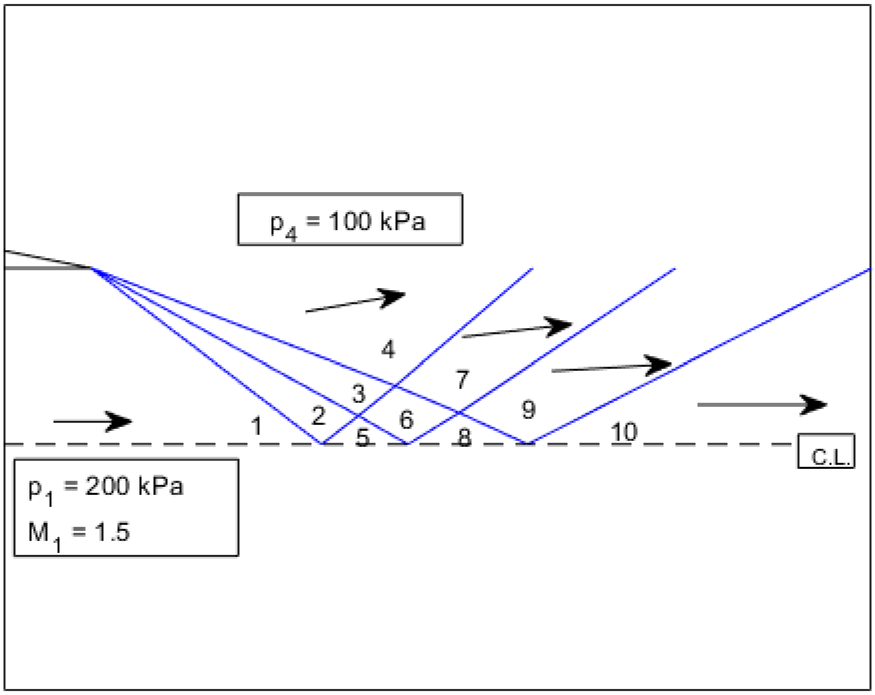
Solve for the flow field downstream of a supersonic nozzle using the method of characteristics. The Mach number at the exit plane is 1.5 and the pressure at the exit plane is 200 kilopascals. The back pressure is 100 kilopascals.

Assumptions:

- Flow is isentropic
- Variation in flow properties depend on the interaction of expansion waves that occur throughout the wake of the nozzle.
- The geometry of the nozzle and the flow is symmetric.

Model the expansion fan as three characteristics. Due to symmetry, arbitrarily choose to work only on the top half of the flow. Following is a figure of the nozzle exit.

```
upperNozzle = astexpandschematic('uppernozzle');
```



The given information in the problem is:

```
exitMach = 1.5; % Mach number at the exit plane [dimensionless]
exitPres = 200; % Static pressure at the exit plane [kPa]
backPres = 100; % Pressure downstream of the nozzle, outside of the expansion wake
```

The fluid is assumed to be air that behaves like a perfect gas with the following constant specific heat ratio.

```
k = 1.4; % Specific heat ratio [dimensionless]
```

Method of Characteristics

The method of characteristics is a theory for supersonic flow that analyzes the irrotational potential flow equation in fully nonlinear form. Isentropic flow is assumed. The definition of characteristics are the curves in the flow where the velocity is continuous but the first derivative of velocity is discontinuous.

In the previous figure, the blue lines in the are approximate characteristics. Characteristics of type I make a negative acute angle with the flow direction. Characteristics of type II make a positive acute angle with the flow direction. A detailed derivation of the method is outside the scope of this example analysis. This example analysis uses the region-to-region procedure. It is assumed that you are familiar with this procedure.

In Prandtl-Meyer flow and the method of characteristics, calculate the important angles for all regions of the flow.

- Flow angle is the direction in which the air is moving.

  $$\theta_n = Flow\ angle\ in\ the\ n^{th}\ region$$

- The Prandtl-Meyer angle is the angle at which the flow changes direction from one region to another.

  $$\nu_n = Prandtl - Meyer\ angle\ in\ the\ n^{th}\ region$$

- Mach angle is the angle between the local flow direction and the weak pressure waves that emanate from a given point.

  $$\mu_n = Mach\ angle\ in\ the\ n^{th}\ region$$

Compute the Mach number in each region and solve for the angles of both types of characteristic in all of the regions. Solve for the geometric boundary of all of the regions by calculating the slopes of all of the characteristics and locate of all of the intersections of characteristics.

Computing the Flow Properties Through the First Expansion Fan

Determine the Mach number outside the wake (region 4). The Mach number at this location can be found using isentropic ratios for pressure and the given values for pressure. The pressure ratio at the exit plane is easily solved for using `flowisentropic`.

```
[~, ~, exitPresRatio] = flowisentropic(k, exitMach);
```

The back pressure ratio is the ratio of the back pressure to the stagnation pressure. The isentropic pressure ratio at the outer wake region is:

$$\frac{p_4}{p_0} = \frac{p_4}{p_1}\frac{p_1}{p_0}$$

```
backPresRatio = backPres / exitPres * exitPresRatio;
```

Calculate the Mach number in region 4 using `flowisentropic`.

```
backMach = flowisentropic(k, backPresRatio, 'pres');
```

The 'pres' string input indicates that the function is in pressure ratio input mode. The flow angle in the back pressure region is the difference in Prandtl-Meyer angles from the exit plane region (region 1) to the back pressure region (region 4).

$$\theta_4 = \nu_4 - \nu_1$$

```
[~, nu_1]    = flowprandtlmeyer(k, exitMach);
[~, nu_4]    = flowprandtlmeyer(k, backMach);
theta_4 = nu_4 - nu_1;
```

Because we are approximating the flow with three characteristics, the calculate the change in flow angle in crossing type I characteristics from region 1 to region 4:

$$\Delta\theta_I = \frac{\theta_4}{3}$$

```
deltaThetaI = theta_4 / 3;
```

Note that flow in region 1 is parallel to the horizontal and therefore:

```
theta_1 = 0;
```

In fact, the flow in any region that straddles the centerline is parallel to the centerline. This is because the centerline is considered to be a boundary for this symmetric flow. In addition, there are no sources nor sinks at the boundary.

```
theta_5  = 0;
theta_8  = 0;
theta_10 = 0;
```

The flow angles of regions 2 and 3 follow simply.

```
theta_2 = theta_1 + deltaThetaI;
theta_3 = theta_2 + deltaThetaI;
```

Across type I characteristics, the change in Prandtl-Meyer angle is equal to the change in flow angle:

$$\Delta\nu_I = \Delta\theta_I$$

```
deltaNuI = deltaThetaI;
```

Calculate the Prandtl-Meyer angle in region 2 by using the Prandtl-Meyer angle in region 1 and deltaNuI, the change in Prandtl-Meyer angle through type I characteristics. Calculate the Prandtl-Meyer angle in region 3 in a similar manner to that of region 2.

$$\Delta\nu_I = \nu_2 - \nu_1$$

$$\nu_2 = \nu_1 + \Delta\nu_I$$

```
nu_2 = nu_1 + deltaNuI;
nu_3 = nu_2 + deltaNuI;
```

Calculating Flow Properties in the Interference Regions

The flow angle in region 5 is known to be zero from the centerline boundary condition. Therefore, the change in angle from region 2 to region 5 is

$$\Delta\theta_{II} = \theta_5 - \theta_2$$

```
deltaThetaII = theta_5 - theta_2;
```

Calculate the change in Prandtl-Meyer angle across type II characteristics:

$$\Delta\nu_{II} = -\Delta\theta_{II}$$

```
deltaNuII = -deltaThetaII;
```

Then, calculate the Prandtl-Meyer angle in region 5. You already know the region 2 Prandtl-Meyer angle and the change in Prandtl-Meyer angle across type II characteristics.

```
nu_5 = nu_2 + deltaNuII;
```

To calculate the properties in region 6, use the fact that the properties in region 3 and region 5 are known. Note also that which characteristic the flow crosses define the changes in properties. From region 5 to region 6, a type I characteristic is crossed. Therefore,

$$\Delta\theta_I = \Delta\nu_I = \theta_6 - \theta_5 = \nu_6 - \nu_5$$

Rearranged this as:

$$\nu_6 - \theta_6 = \nu_5 - \theta_5 \qquad (1)$$

A type II characteristic is crossed in going from region 3 to region 6. Therefore,

$$\Delta\nu_{II} = -\Delta\theta_{II}$$

$$\nu_6 - \nu_3 = \theta_3 - \theta_6$$

Rearranging this as:

$$\nu_6 + \theta_6 = \nu_3 + \theta_3 \qquad (2)$$

Add equations (1) and (2) together, then solve for the Prandtl-Meyer angle in region 6. This yields the following expression.

$$\nu_6 = \frac{(\nu_5 - \theta_5) + (\nu_3 + \theta_3)}{2}$$

In MATLAB®, use:

```
nu_6 = ((nu_5 - theta_5) + (nu_3 + theta_3))/2;
```

From equation (1), the flow angle in region 6 is

```
theta_6 = nu_6 - (nu_5 - theta_5);
```

For region 7 a type one characteristic is crossed and all information is available in region 6.

```
nu_7    = nu_6 + deltaNuI;
theta_7 = theta_6 + deltaThetaI;
```

Region 8 is on the centerline; its flow angle is zero. Going from region 6 to region 8 requires crossing a type II characteristic. Therefore, calculate the Prandtl-Meyer angle in region 8 as:

```
nu_8 = nu_6 + deltaNuII;
```

Calculate the Prandtl-Meyer angle and flow angle in region 9 just like the way you did for region 6. Region 8 is the upstream region across the type I characteristic. Region 7 is the upstream region across the type II characteristic.

```
nu_9    = ((nu_8 - theta_8) + (nu_7 + theta_7))/2;
theta_9 = nu_9 - (nu_8 - theta_8);
```

Region ten is on the centerline. The flow is parallel and so the flow angle is zero. Use the Prandtl-Meyer angle in region 9 and the crossing over a type II characteristic to calculate the Prandtl-Meyer angle in region 10.

```
nu_10 = nu_9 + deltaNuII;
```

Preparing and Tabulating the Flow Parameter Results

For upcoming calculations, combine the flow angles into one vector and the Prandtl-Meyer angles into another vector.

```
flowAngles         = [theta_1 theta_2 theta_3 theta_4 theta_5 theta_6 theta_7 theta_8 t

prandtlMeyerAngles = [nu_1 nu_2 nu_3 nu_4 nu_5 nu_6 nu_7 nu_8 nu_9 nu_10];
```

To calculate the Mach numbers and Mach angles in each region, using the `flowprandtlmeyer` function with the prandtlMeyerAngles as the input. You can use the results from this function to find the angle that the type I and type II characteristics make with the horizontal inside each region. You can then use these angles to calculate slopes in the x-y plane, where the centerline is the x-axis and the exit plane of the nozzle is the y-axis. For type I characteristics and type II characteristics, respectively, the slopes are:

$$\left(\frac{dy}{dx}\right)_I = tan(\theta - \mu)$$

$$\left(\frac{dy}{dx}\right)_{II} = tan(\theta + \mu)$$

Note, the values in the following table for type I and type II are the angles with the horizontal, not the slopes.

```
% Preallocation for speed
machNumbers = zeros(1,10);
machAngles  = zeros(1,10);
typeOne     = zeros(1,10);
typeTwo     = zeros(1,10);

for i = 1:10
    [machNumbers(i), ~, machAngles(i)] = flowprandtlmeyer(k, prandtlMeyerAngles(i), 'nu
    typeOne(i) = flowAngles(i) - machAngles(i);
    typeTwo(i) = flowAngles(i) + machAngles(i);
end


clear table;
table(1,:) = 'Region   theta        nu        Mach        mu       type I    type II';
table(2,:) = '         (Deg)      (Deg)                  (Deg)     (Deg)      (Deg) ';
for m=1:length(machNumbers)
table(m+3,:) = sprintf('%3.0d   %8.2f      %5.2f   %8.3f    %8.2f   %8.2f  %8.2f ', ..
                m, flowAngles(m), prandtlMeyerAngles(m), machNumbers(m), ...
                machAngles(m), typeOne(m), typeTwo(m));
end

disp(table)
```

```
Region   theta        nu        Mach        mu       type I    type II
         (Deg)      (Deg)                  (Deg)     (Deg)      (Deg)

  1       0.00      11.91      1.500       41.81     -41.81      41.81
  2       4.45      16.35      1.650       37.29     -32.85      41.74
  3       8.89      20.80      1.803       33.70     -24.80      42.59
  4      13.34      25.24      1.959       30.69     -17.35      44.03
  5       0.00      20.80      1.803       33.70     -33.70      33.70
  6       4.45      25.24      1.959       30.69     -26.25      35.14
  7       8.89      29.69      2.122       28.11     -19.22      37.00
  8       0.00      29.69      2.122       28.11     -28.11      28.11
  9       4.45      34.14      2.294       25.84     -21.40      30.29
 10       0.00      38.58      2.477       23.81     -23.81      23.81
```

Note the following:

- The flow angles increase away from the centerline.
- The Prandtl-Meyer angles increase as the flow moves downstream.
- The Mach number also increases as the flow moves downstream.

Solving for the Flow Geometry

The flow properties are known in all regions, but in order to solve for the flow field you must calculate the actual geometry of each region. The last two columns of the above table contain the angles that each type of characteristic makes with the horizontal. Because straight lines approximate the characteristics of the flow in each region, the boundary between any two regions is approximated by the average of the angles that each makes in the bordering regions. Because the waves bend through the expansion fan, begin the analysis from the point from which the characteristics originate. The characteristics originate at the lip of the nozzle and work downstream.

Assume the intersection of the centerline and the exit plane of the nozzle is the origin of our coordinate system. Also assume lengths are normalized to half of the exit height of the nozzle. The positive x axis is taken horizontal along the centerline in the downstream direction. The positive y axis is vertically up in the exit plane of the nozzle. The lip of the nozzle is at the point (0,1).

All three characteristics that propagate from the upper lip are type I characteristics. Analyze the steepest sloping characteristic first because no waves interfere with the steepest wave until the steepest wave intersects the centerline. In the symmetric half-nozzle model, the steepest wave reflects back into the fan and interferes with the other waves in the expansion fan.

This wave that "reflects" from the centerline is actually the steepest sloping type II characteristic that propagates from the bottom lip. However, the analysis considers the centerline to be a boundary due to symmetry. This produces the same results that you would get if you worked both halves of the nozzle.

The steepest sloping line from the lip is a type I characteristic that separates region 1 and region 2. To calculate the angle that steepest sloping wave makes with the horizontal use the average of angles that the type I characteristics make in each region. To calculate the slope use trigonometry.

```
avgAngle12 = (typeOne(1) + typeOne(2)) / 2;
slope12    = tand(avgAngle12);
```

With the following information known:

- Slope of the first type I wave in x-y space.
- The y-intercept of the wave (y = 1 at the lip).
- The wave intersects the centerline (y = 0) without interference.

Calculate the x-location of the point using the equation of a line in slope-intercept form. Rearrange y = m*x+b for the x-location with y = 0 to produce x = -b / m. This is the x-location of the first downstream point, point 1.

```
y1 = 0; % On the centerline
x1 = -1 / slope12;
```

From point 1, the first type II characteristic propagates and interferes with the fan. The other type I characteristics that originate from the nozzle lip are disturbed by the type II wave, but not before reaching that wave. Therefore, calculate the points of intersection of the steepest type II characteristic and the flatter type I waves from the lip. The type II characteristic coming up from the centerline separates region 2 and region 5. The average of the two angles and associated slopes are given by:

```
avgAngle25 = (typeTwo(2) + typeTwo(5)) / 2;
slope25    = tand(avgAngle25);
```

The second steepest type I characteristic is the from the nozzle lip, separates region 2 and region 3. The average angle with the horizontal and the associated slope of the wave are given by:

```
avgAngle23 = (typeOne(2) + typeOne(3)) / 2;
slope23    = tand(avgAngle23);
```

Calculate the point of intersection of the region 2-3 boundary and the region 2-5 boundary. You need this point because the characteristics interfere with each other at this point. The slopes of both boundaries and a point on each line are known. Point 1 and the nozzle lip (to be referenced point 0 are known." Solve for the unknown x-coordinate of the point of intersection. Use that x-location in the equation of either of the two lines to find the y-location of the point of intersection. The point-slope form equation of a line through point p with slope m is:

$$y - y_p = m(x - x_p)$$

The advantage of this form of a line is that you need only one point and the slope to completely define the line. The x and y without subscripts can be any point on the line. However, the point of intersection of two lines must be unique. Calling this point of intersection point 2, the equation of both lines are the following.

$$y_2 - y_0 = m_{2,3}(x_2 - x_0) \quad (3)$$

$$y_2 - y_1 = m_{2,5}(x_2 - x_1) \quad (4)$$

where

$$m_{i,j} = Slope\ of\ the\ boundary\ between\ region\ i\ and\ region\ j$$

Subtract and rearrange:

$$x_2 = \frac{x_1 m_{2,5} - x_0 m_{2,3} + y_0 - y_1}{m_{2,5} - m_{2,3}}$$

Knowing some of the values exactly due to the axes intercepts simplifies this expression to.

```
x2 = (x1 * slope25 + 1) / (slope25 -slope23);
```

Below the y-location of point 2 is found by plugging the x-location of point 2 into equation (4) above, but plugging into equation (3) works just as well.

```
y2 = (x2 - x1) * slope25;
```

Use the slope-intercept formula and the procedure above to calculate all points. To calculate the third point in the flow, first calculate the intersection of the region 3-4 boundary and the 3-6 boundary. The angles of the boundary lines are calculated using the average of the angles with the horizontal. You can then use trigonometry to find the slope, which is now computed in one step.

```
slope34 = tand( (typeOne(3) + typeOne(4)) / 2 );
slope36 = tand( (typeTwo(3) + typeTwo(6)) / 2 );
```

Because the boundary between region 3 and region 4 is a type I characteristic and the boundary between region 3 and region 6 is a type II characteristic, be careful to take the angles for the appropriate type. Use the point-slope form of these boundary lines subtracted from each other to calculate the x-location of point 3.

```
x3 = (y2 - 1 - x2 * slope36) / (slope34 - slope36);
y3 = (x3 - x2) * slope36 + y2;
```

The first type II characteristic now propagates beyond the fan and does not interfere with any other characteristics. The angle that defines the direction in which the steepest type II propagates is an angle which is the average of type II waves in regions 4 and 7.

```
slope47 = tand( (typeTwo(4) + typeTwo(7)) / 2);
```

Solve also the second steepest type I characteristic to continue downstream. Start from the known location of point 2 to calculate the x-intercept of the second type I characteristic. Again, the solution uses the point-slope form of the line. However, there is no interference until the centerline boundary is reached. We only need to consider one line to find the x-location of "point 4". The slope of the boundary between region 5 and region 6 is a type I wave:

```
slope56 = tand( (typeOne(5) + typeOne(6)) / 2 );
```

The rearranged point-slope form (knowing y = 0 at the centerline) is used to find point 4.

```
x4 = ( slope56 * x2 - y2 ) / slope56;
y4 = 0;
```

Calculate point 5 in the same manner as point 2. The region 6-7 boundary is type I and the region 6-8 boundary is type II.

```
slope67 = tand( ( typeOne(6) + typeOne(7)) / 2);
slope68 = tand( (typeTwo(6) + typeTwo(8)) / 2);
```

The known point on the region 6-7 boundary is point 3. The known point on the region 6-8 boundary is point 4. Use this information in the slope-intercept form, subtracting the equations, and rearrange to yield the location of the next point.

```
x5 = (-x4 * slope68 + x3 * slope67 + y4 -y3) / (slope67 - slope68);
y5 = (x5 - x4) * slope68 + y4;
```

The second type II characteristic propagates beyond the fan at an angle averaged between the region 7 and region 9 angles for type II waves.

```
slope79 = tand( (typeTwo(7) + typeTwo(9)) / 2);
```

The last point of interest is the x-intercept of the flattest type I wave. Calculate this point by knowing the location of point 5 and finding the slope of the type I wave between region 8 and region 9.

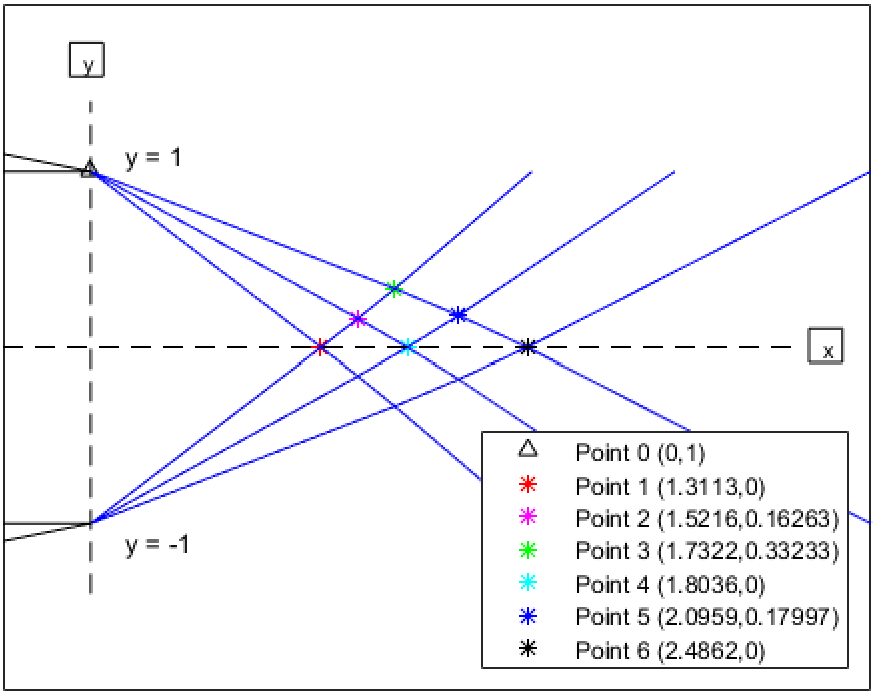
```
slope89 = tand( (typeOne(8) + typeOne(9)) / 2);
y6      = 0;
x6      = (slope89 * x5 - y5) / slope89;
```

The final type II wave propagates away at an angle averaged between region 9 and region 10.

```
slope910 = tand( (typeTwo(9) + typeTwo(10)) /2);
```

With all the points calculated and the slope of the freely propagating lines known, connect the dots:

```
points = astexpandschematic('nozzlepoints');
```



The method of characteristics is a potent method for solving supersonic gas dynamics problems. Note, that this method represents an approximation for the characteristic lines. The approximation approaches the exact case for an infinite number of characteristic lines.

```
close(upperNozzle,points)
```

Reference

[1] James, J. E. A., "Gas Dynamics, Second Edition", Allyn and Bacon, Inc, Boston, 1984.

Visualizing World Magnetic Model Contours for 2015v2 Epoch

This example shows how to visualize contour plots of the calculated values for the Earth's magnetic field using World Magnetic Model 2015v2 (WMM2015v2) overlaid on maps of the Earth. WMM2015v2, released by NOAA in February of 2019, addresses degraded performance of WMM2015(v1) in the Arctic region and supersedes it. For more information about the WMM2015v2 release, see https://www.ncei.noaa.gov/news/world-magnetic-model-out-cycle-release.

The Mapping Toolbox™ is required to generate the maps of the Earth.

Generating Values for Earth's Magnetic Field

Calculate values for the Earth's magnetic field using `wrldmagm` function to implement the World Magnetic Model 2015v2 (WMM2015v2):

- X - North component of magnetic field vector in nanotesla (nT)
- Y - East component of magnetic field vector in nanotesla (nT)
- Z - Down component of magnetic field vector in nanotesla (nT)
- H - Horizontal intensity in nanotesla (nT)
- DEC - Declination in degrees
- DIP - Inclination in degrees
- F - Total intensity in nanotesla (nT)

Based on the `wrldmagm` inputs:

- model_epoch - Epoch of WMM model.
- decimal_year - Scalar value representing the decimal year within the epoch for which the data was generated.

```
model_epoch = '2015v2';
decimal_year = 2015;
```

For a given epoch and decimal year, use the following code to generate 13021 data points for calculating values of Earth's magnetic field using `wrldmagm`. To reduce overhead calculation, this model includes a mat-file that contains this data for epoch 2015v2 and decimal year 2015.

```
% % Assume zero height
% height = 0;
%
% % Geodetic Longitude value in degrees to use for latitude sweep.
% geod_lon = -180:1:180;      %degrees
%
% % Geodetic Latitude values to sweep.
% geod_lat = -89.5:.5:89.5;       %degrees
%
% % Loop through longitude values for each array of latitudes -89.5:89.5.
% for lonIdx = size(geod_lon,2):-1:1
%     for latIdx = size(geod_lat,2):-1:1
%
%     % Use WRLDMAGM function to obtain magnetic parameters for each lat/lon
%     % value.
%     [xyz, h, dec, dip, f] = wrldmagm(height, geod_lat(latIdx),geod_lon(lonIdx), deci
%
%     % Store results
%     WMMResults(latIdx,1:7,lonIdx) = [xyz' h dec dip f];
%
%     end
% end
```

Load data saved in mat-file.

```
WMMFileName = 'WMMResults_Epoch_2015v2_decyear_2015.mat';
load(WMMFileName);
```

Read in continent land areas for plot overlay using Mapping Toolbox function, `shaperead`.

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

Plotting Earth's Magnetic Field Overlaid on Earth Maps

Load plot formatting data for each of the magnetic parameters.

```
plotWMM = load('astPlotWMM.mat');
```

```
hX = figure;
set(hX,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 1, landAreas, geod_lat, geod_lon, decimal_yea
```

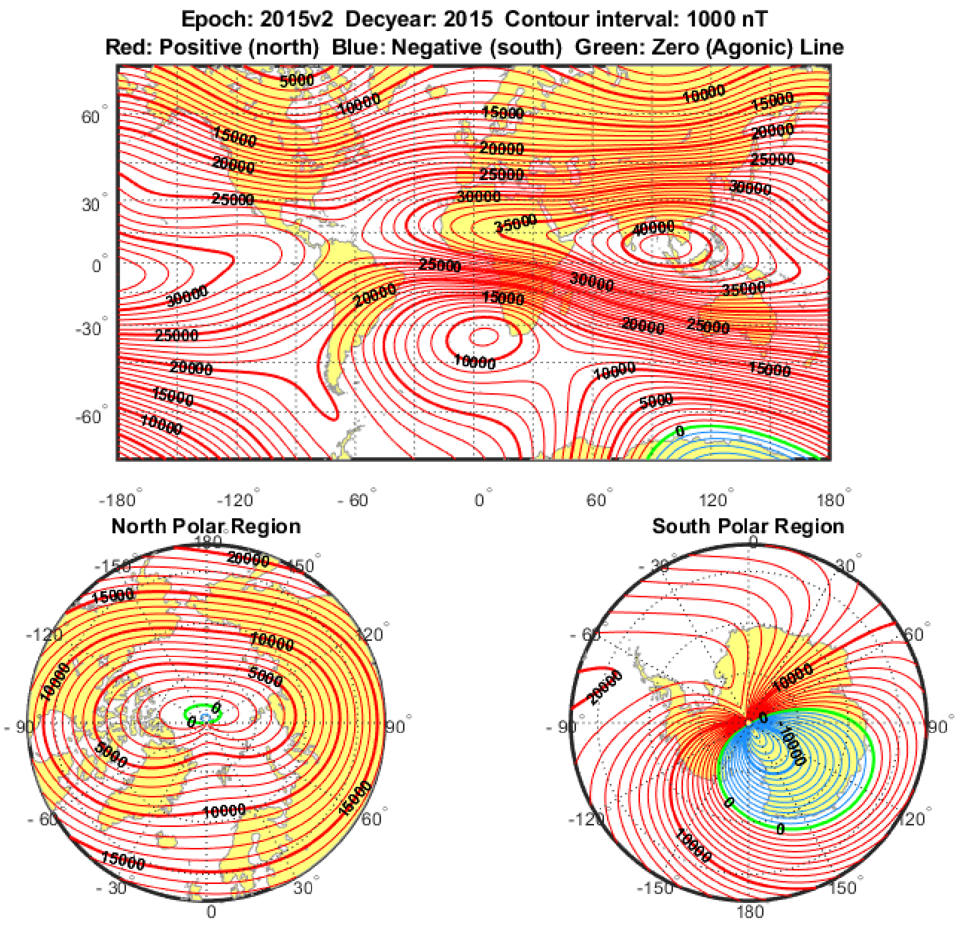

Epoch: 2015v2 Decyear: 2015 Contour interval: 1000 nT
Red: Positive (north) Blue: Negative (south) Green: Zero (Agonic) Line
North Polar Region
South Polar Region

Figure 1: North Component of Magnetic Field Vector, X (nT)

```
hY = figure;
set(hY,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 2, landAreas, geod_lat, geod_lon, decimal_yea
```

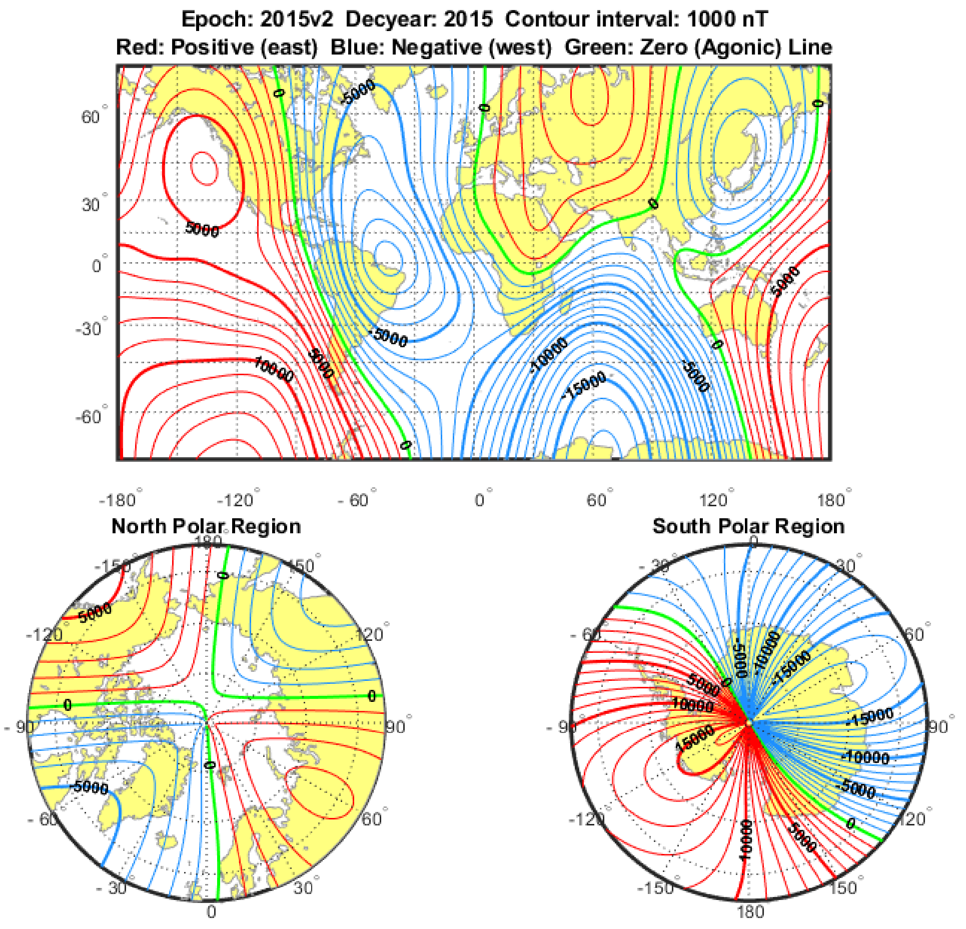

Epoch: 2015v2 Decyear: 2015 Contour interval: 1000 nT
Red: Positive (east) Blue: Negative (west) Green: Zero (Agonic) Line
North Polar Region
South Polar Region
5000
-5000
10000
-10000
-15000
0
60°
30°
0°
-30°
-60°
-180°
-120°
-60°
60°
120°
180°

Figure 2: East Component of Magnetic Field Vector, Y (nT)

```
hZ = figure;
set(hZ,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 3, landAreas, geod_lat, geod_lon, decimal_year
```

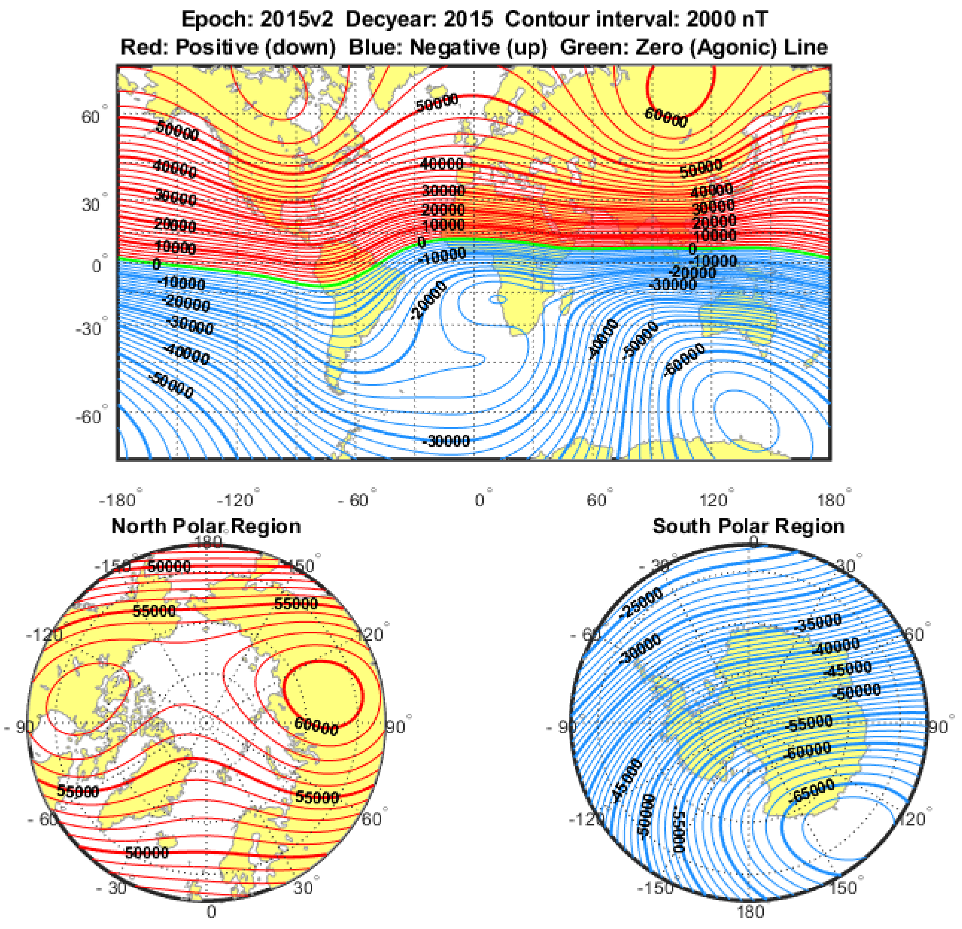

Epoch: 2015v2 Decyear: 2015 Contour interval: 2000 nT
Red: Positive (down) Blue: Negative (up) Green: Zero (Agonic) Line
North Polar Region
South Polar Region

Figure 3: Down Component of Magnetic Field Vector, Z (nT)

```
hH = figure;
set(hH,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 4, landAreas, geod_lat, geod_lon, decimal_yea
```

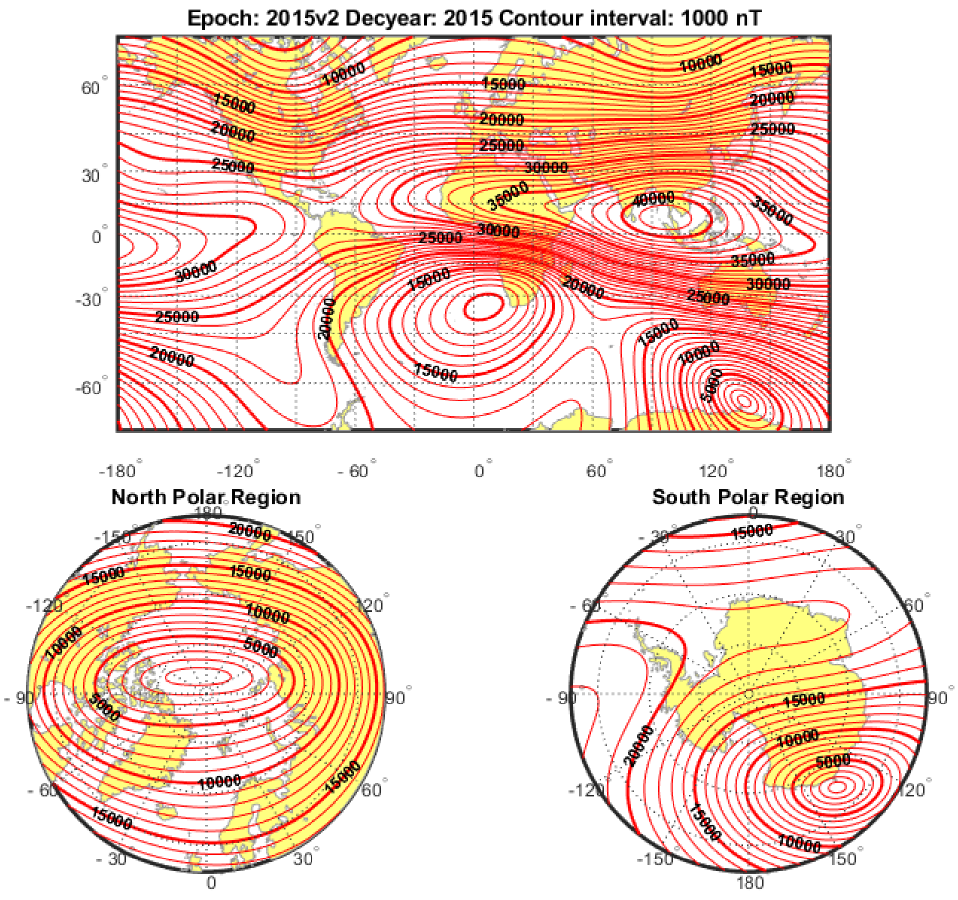


Figure 4: Horizontal Intensity, H (nT)

```
hDEC = figure;
set(hDEC,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 5, landAreas, geod_lat, geod_lon, decimal_year
```

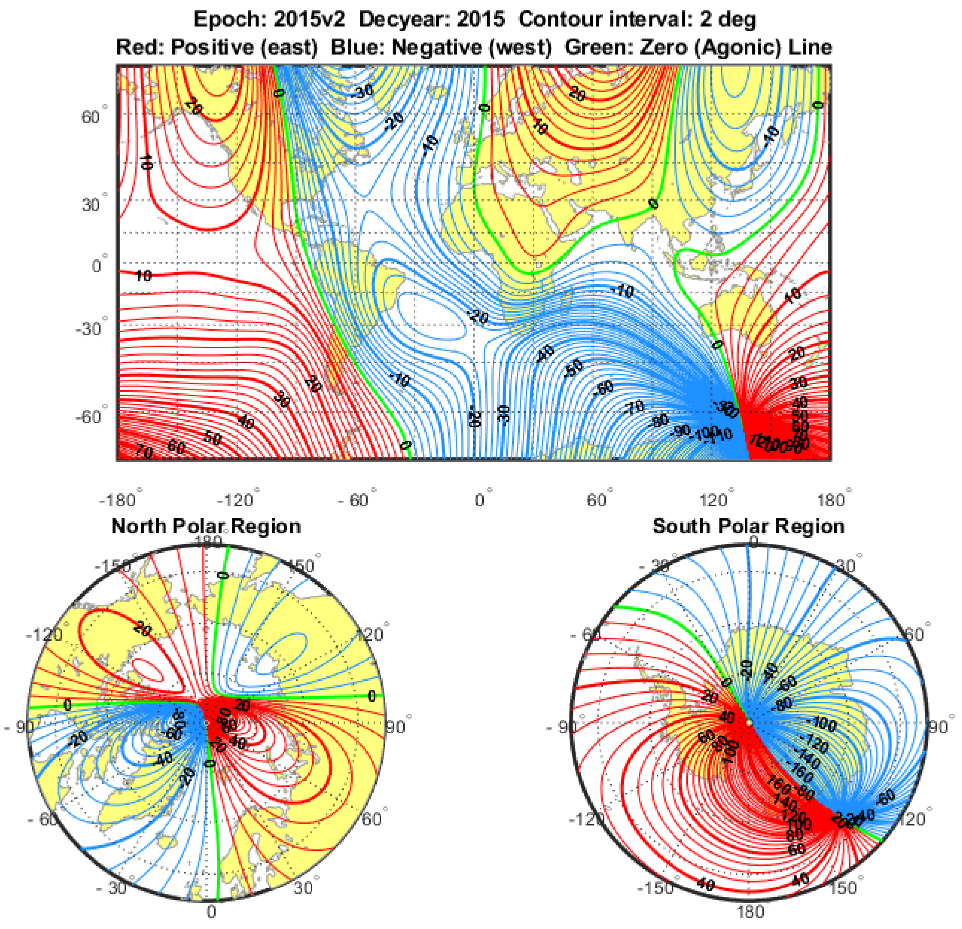

Epoch: 2015v2 Decyear: 2015 Contour interval: 2 deg
Red: Positive (east) Blue: Negative (west) Green: Zero (Agonic) Line
North Polar Region
South Polar Region

Figure 5: Declination, DEC (deg)

```
hDIP = figure;
set(hDIP,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 6, landAreas, geod_lat, geod_lon, decimal_yea
```

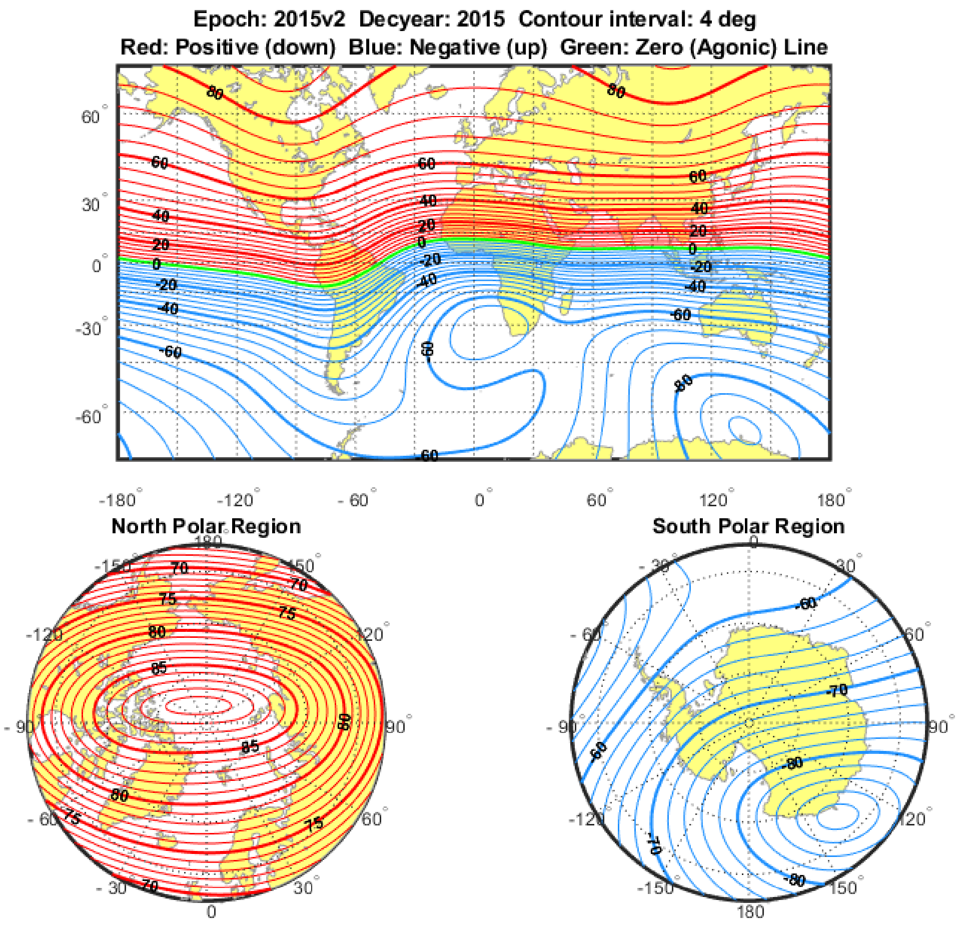

Epoch: 2015v2 Decyear: 2015 Contour interval: 4 deg
Red: Positive (down) Blue: Negative (up) Green: Zero (Agonic) Line
North Polar Region
South Polar Region

Figure 6: Inclination, DIP (deg)

```
hF = figure;
set(hF,'Position',[0 0 827 620],'Color','white')
astPlotWMMContours( WMMResults, plotWMM, 7, landAreas, geod_lat, geod_lon, decimal_yea
```

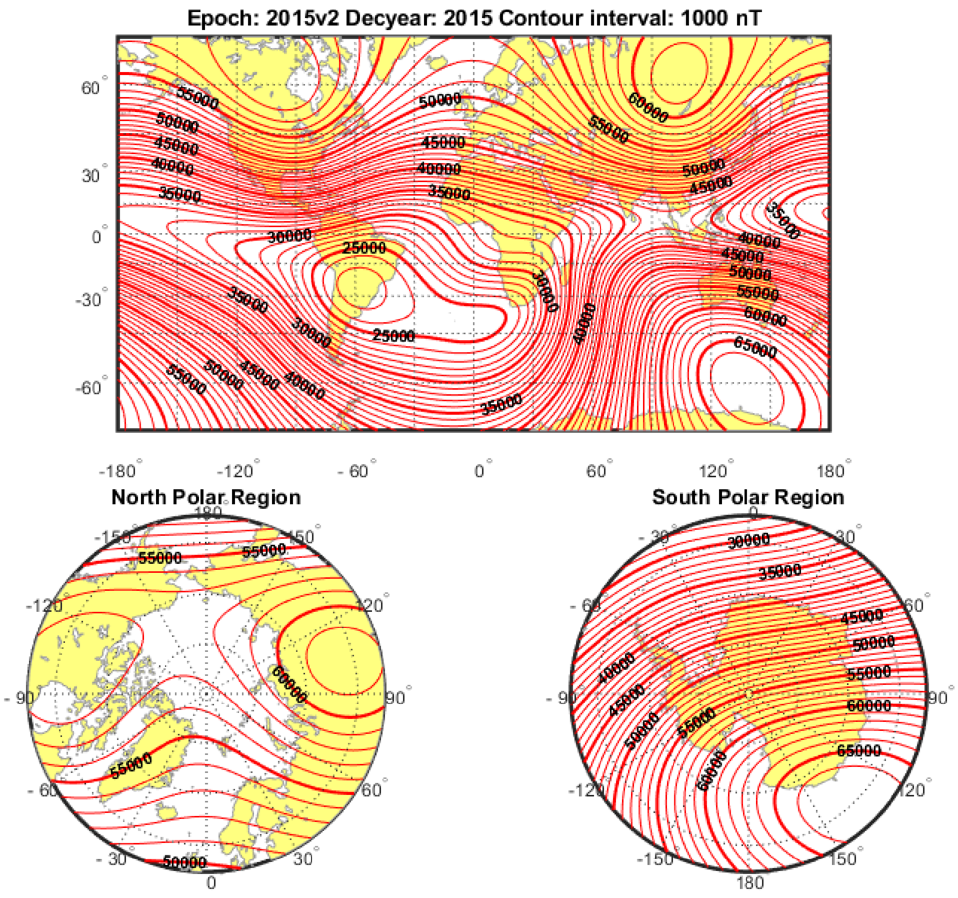


Figure 7: Total Intensity, F (nT)

```
close (hX, hY, hZ, hH, hDEC, hDIP, hF)
```

Visualizing Geoid Height for Earth Geopotential Model 1996

This example shows how to calculate the Earth's Geoid height using the EGM96 Geopotential Model of the Aerospace Toolbox™ software. It also shows how to visualize the results with contour maps overlaid on maps of the Earth. The Mapping Toolbox™ and Simulink® 3D Animation™ are required to generate the visualizations.

Generating Values for Earth Geopotential Model 1996

Calculate values for the Earth's Geopotential using the `geoidheight` function to implement the EGM96 Geopotential Model.

The following code can be used to generate 260281 data points for calculating values of the Earth's Geoid height using `geoidheight`. To reduce computational overhead, this example includes a mat-file that contains this data.

```
% % Set amount of increment between degrees
% gridDegInc = 0.5;                   %degrees
%
% % Geocentric Longitude value in degrees to use for latitude sweep.
% geoc_lon =-180:gridDegInc:180;      %degrees
%
% % Geodetic Latitude values to sweep.
% geod_lat = -90:gridDegInc:90;       %degrees
%
% % Convert to geocentric to obtain geoid height.
% Re = 6378137;
% f = 1/298.256415099;
% geoc_lat = geod2geoc(geod_lat, 0, f, Re);
%
% % Loop through longitude values for each array of latitudes -90:90.
% for lonIdx = size(geoc_lon,2):-1:1
%
%     % Longitude must be the same dimension as the latitude array
%     lon = geoc_lon(lonIdx)*ones(1,numLatitude);      % degrees
%     geoidResults(1:end,lonIdx) = geoidheight(geoc_lat,lon,'None');
%
% end
```

Loading Geoid Data File and Coastal Data

```
geoidFileName = 'GeoidResults_05deg_180.mat';
load(geoidFileName);
coast = load('coast');
```

Plot 2-D View of Geoid Height

```
% Create 2-D plot using |meshm|
h2D = figure;
set(h2D,'Position',[20 75 700 600],'Toolbar','figure');

% Reference matrix for mapping geoid heights to lat/lon on globe.
RRR = makerefmat('RasterSize',size(geoidResults), ...
    'Latlim', [-90 90], 'Lonlim', [-180 180] );

ast2DGeoidPlot(RRR,geoidResults,coast,gridDegInc)

% Viewing Geoid height using VR canvas
www2D = vrworld('astGeoidHeights.wrl');
open(www2D)

% Actual geoid heights for reference
geoidGrid = vrnode(www2D,'EGM96_Grid');
actualHeights = getfield(geoidGrid,'height'); %#ok<GFLD>

% Initialize heights to 0 for slider control
geoidGrid.height = 0*actualHeights;

% Size canvas for plotting and set parameters
geoidcanvas2D = vr.canvas(www2D,'Parent',h2D,...
    'Antialiasing', 'on','NavSpeed','veryslow',...
    'NavMode','Examine','Units', 'normalized',...
    'Viewpoint','Perspective','Position',[.15 .04 .7 .42]);

% Create slider
slid=astGeoidSlider(geoidcanvas2D);
```

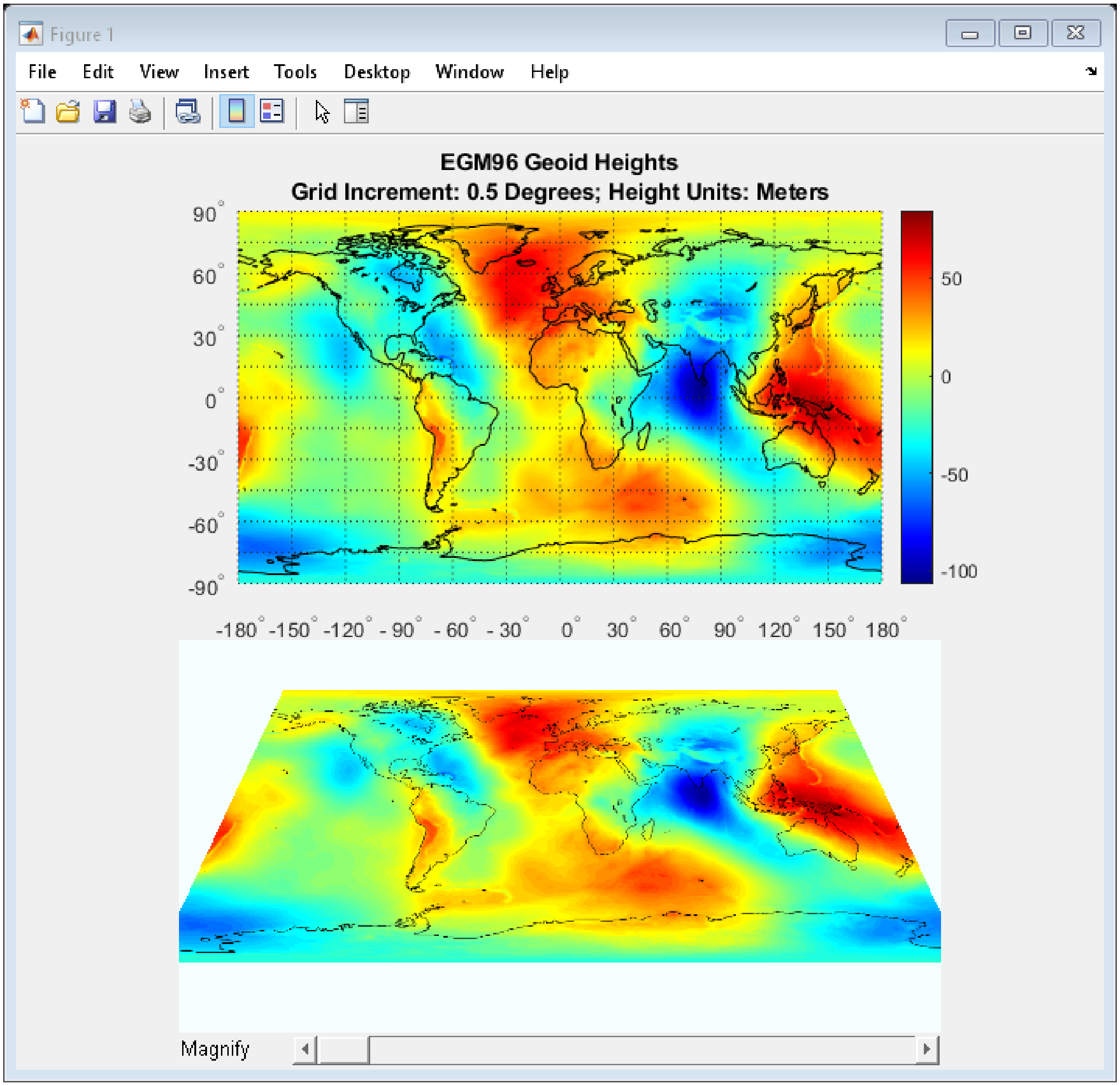


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
EGM96 Geoid Heights
Grid Increment: 0.5 Degrees; Height Units: Meters
90°
60°
30°
0°
-30°
-60°
-90°
-180° -150° -120° - 90° - 60° - 30° 0° 30° 60° 90° 120° 150° 180°
50
0
-50
-100
Magnify

Plot 3-D View of Geoid Height

```
h3D = figure;
set(h3D,'Position',[20 75 700 600]);

% Set up axes
hmapaxis = axesm ('globe','Grid', 'on');
set(hmapaxis,'Position',[.1 .5 .8 .4])

view(85,0)
axis off

% Plot data on 3-D globe
meshm(geoidResults,RRR)

% Plot land mass outline
plotm(coast.lat,coast.long,'Color','k')
colormap('jet');

% Plot Title
title({'EGM96 Geoid Heights';['Grid Increment: ' ,num2str(gridDegInc), ' Degrees; Heigh

colorbar;

% 3-D Globe: Geoid Height Using VR Canvas
www3D = vrworld('astGeoidSphere.wrl');
open(www3D)

% Position canvas
geoidcanvas3D = vr.canvas(www3D,'Parent',h3D,...
    'Antialiasing', 'on','NavSpeed','veryslow',...
    'NavMode','Examine','Units', 'normalized',...
    'Position',[.15 .04 .7 .4]);
vrdrawnow;
```

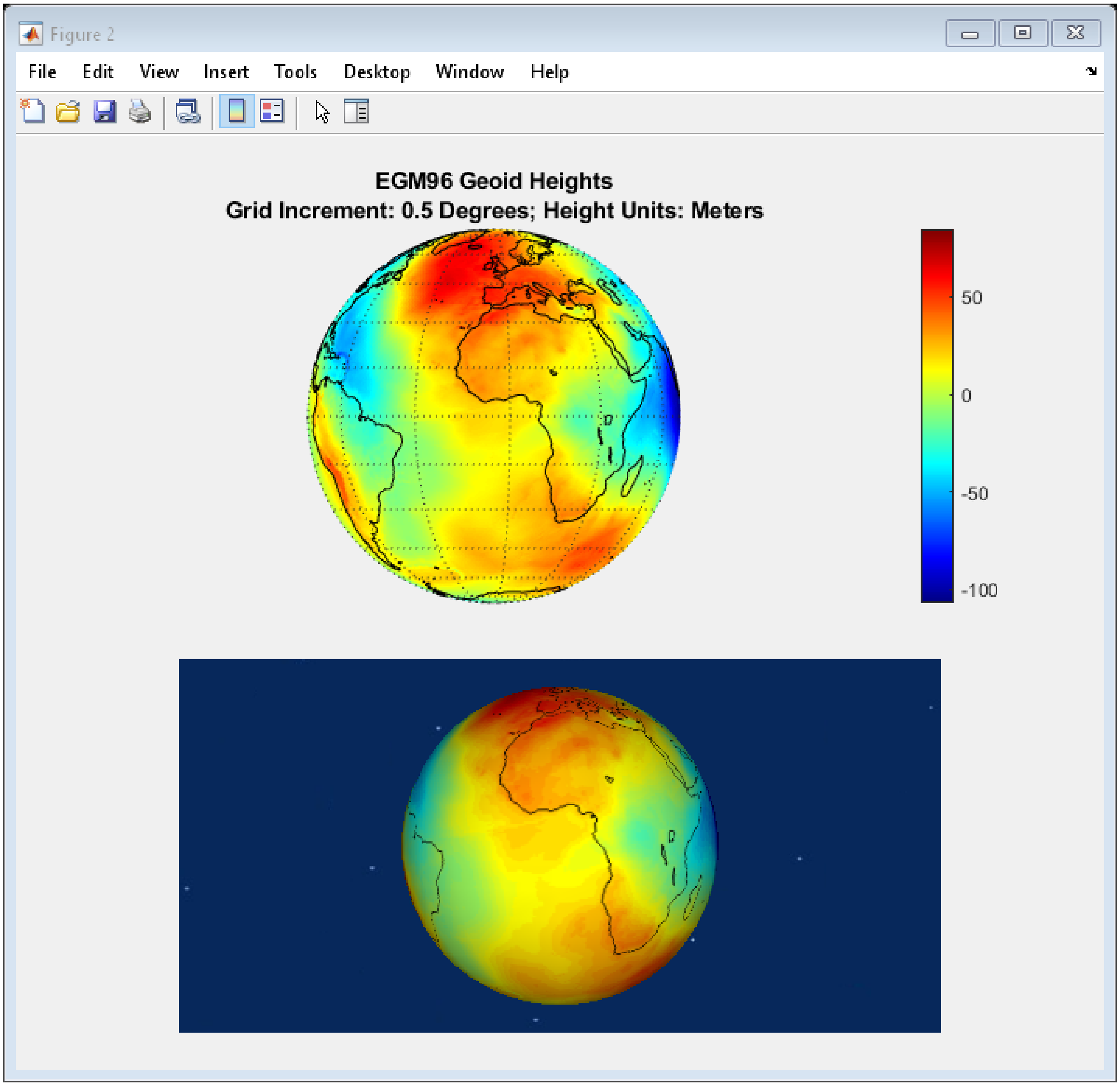


Figure 2
File
Edit
View
Insert
Tools
Desktop
Window
Help
EGM96 Geoid Heights
Grid Increment: 0.5 Degrees; Height Units: Meters
50
0
-50
-100

Clean Up

```
close(h2D,h3D)
close(www2D);close(www3D);
delete(www2D);delete(www3D);
```

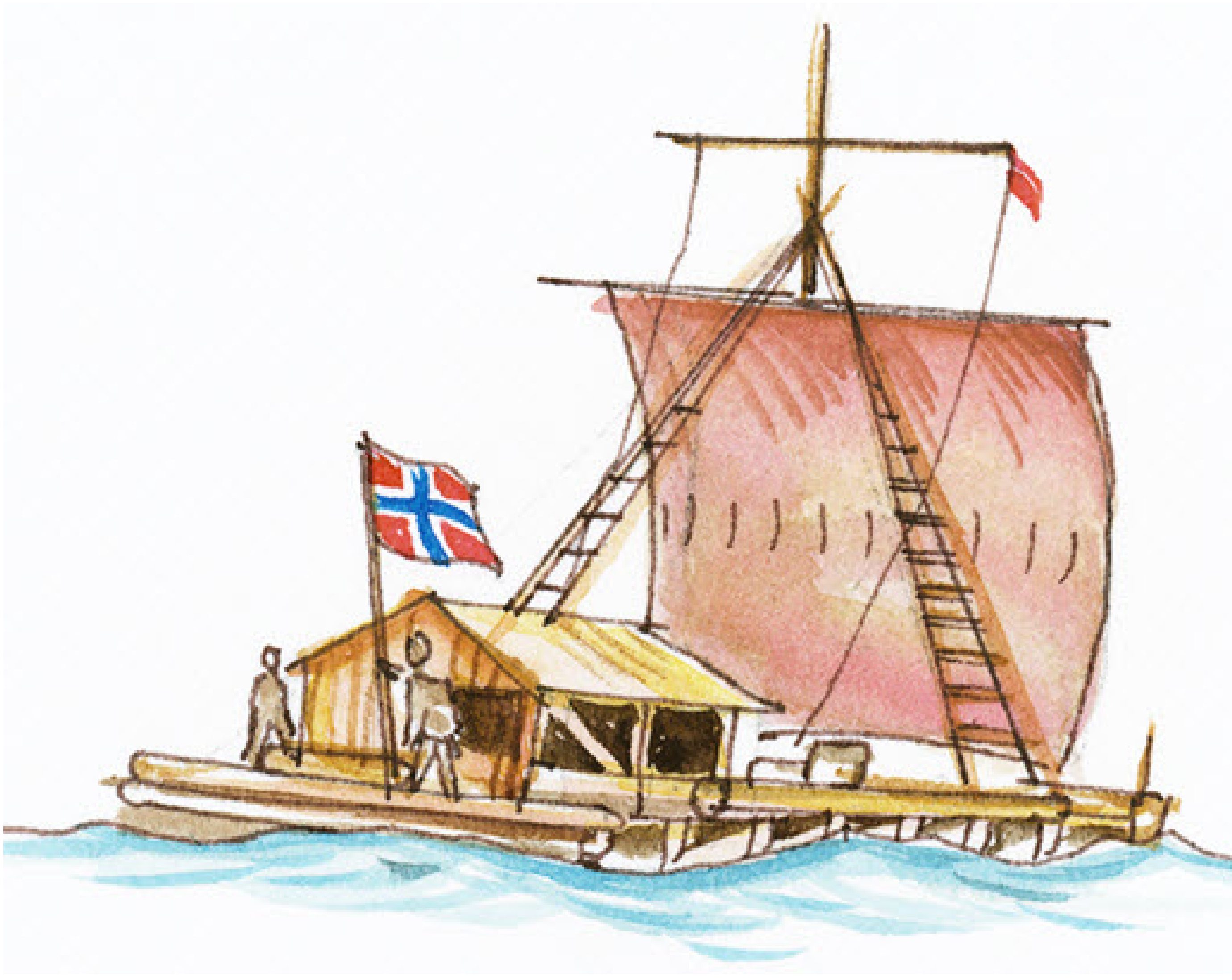
Marine Navigation Using Planetary Ephemerides

This example shows how to use the planetary ephemerides and a Earth Centered Inertial to Earth Centered Earth Fixed (ECI to ECEF) transformation to perform celestial navigation of a marine vessel.

This example uses the Mapping Toolbox™. You must also download data for the example using the aeroDataPackage command.

POLYNESIA
MUANA PAKIPIKA AKAU
PAKIPIKA HEMA

This example uses the route followed by the 1947 expedition across the Pacific Ocean Kon-Tiki. The expedition, led by Thor Heyerdahl, aimed to prove the theory that the Polynesian islands were populated by people from South America in pre-Columbian times. The expedition took 101 days and sailed from the port of Callao, Peru to the Raroia atoll, French Polynesia.



Notes: This example loosely recreates the expedition route. It takes some liberties to show the planetary ephemerides and ECI to ECEF transformation simply.

Load Vessel Track

Load the astKonTikiData.mat file. It contains the ship trajectory, velocity and course for this example. This file stores, the latitude and longitude for the different track points in the variables "lat" and "long", respectively. The variables contain enough data for one track point per day from the port of Callao to the Raroia atoll. Additionally, this file also stores values for each day's vessel velocity in knots per day "V" and course in deg "T".

```
load astKonTikiData
```

Create an Observation Structure

The nautical reduction process is a series of steps that a navigator follows to determine the latitude and longitude of his vessel. It is based on the theory described in The American Practical Navigator[1], the Nautical Almanac[2], and the Explanatory Supplement to the Astronomical Almanac[3]. The process uses observational data obtained from a sextant, clock, compass, and navigational charts. It returns the intercept (p) and true azimuth (Z) for each of the observed objects. This example uses an observation structure array, obs, to contain the observational data. The fields for the structure array are:

- h: Height of eye level of the observer, in m.
- IC: Sextant's index correction, in deg.
- P: Local ambient pressure, in mb.
- T: Local temperature, in C.
- year: Local year at the time of the observation.
- month: Local month at the time of the observation.
- day: Local day at the time of the observation.
- hour: Local hour at the time of the observation.
- UTC: Coordinated Universal Time for the observation, represented as a six element vector with year, month, day, hour, minutes, and seconds.
- Hs: Sextant altitude of the celestial object above the horizon, in deg.
- object: Celestial object used for the measurement (i.e., Jupiter, Neptune, Saturn, etc).
- latitude: Estimated latitude of the vessel at the time of the observation, in deg.

- longitude: Estimated longitude of the vessel at the time of the observation, in deg.
- declination: Declination of the celestial object, in deg.
- altitude: Distance from the surface of the earth to the celestial object, in km.
- GHA: Greenwich Hour Angle, which is the angle in degrees of the celestial object relative to the Greenwich meridian.

For simplicity, assume that all measurements are taken at the same location in the boat, with the same sextant, at the same ambient temperature and pressure:

```
obs.h = 4;
obs.IC = 0;
obs.P = 982;
obs.T = 15;
```

The expedition departed on April 28th 1947. As a result, initialize the structure for the observation for this date:

```
obs.year = 1947;
obs.month = 4;
obs.day = 28;
```

Initialize Dead Reckoning Process for Navigation

To start the dead reckoning process, define the initial conditions for the position of the vessel. Store the latitude for a fixed solution for the latitude and longitude in the latFix and longFix variables, respectively. In this example, for the first fix location, use the latitude and longitude of Callao, Peru:

```
longFix = zeros(size(long));
latFix = zeros(size(lat));
longFix(1) = long(1);
latFix(1) = lat(1);
```

Daily Dead Reckoning

For this example, assume that a fix is obtained daily using the observational data. Therefore, this example uses a "for loop" for each observation. The variable "m" acts as a counter representing every elapsed day since the departure from the port:

```
for m = 1:size(lat,1)-1
```

Increment the day and make day adjustments for the months of June and April, both of which have only 30 days:

```
    obs.day = obs.day + 1;
    [obs.month,obs.day] = astHelperDayCheck(obs.year,obs.month,obs.day);
```

Actual Latitude and Longitude

Extract the vessel actual position for each day from the track points loaded earlier. The example uses this value to calculate the local time zone and the position of the selected planets in the sky:

```
    longActual = long(m+1);
    latActual = lat(m+1);
```

Planet Selection

Select the planets for observation if they are visible to the vessel for the given latitude and longitude. The following code uses precalculated data:

```
    if longActual>-90
        obs.object = {'Saturn';'Neptune'};
    elseif longActual<=-90 && longActual>-95
        obs.object = {'Saturn';'Neptune';'Jupiter'};
    elseif longActual<=-95
        obs.object = {'Neptune';'Jupiter'};
    end
```

UTC Time Calculation

Adjust local time to UTC depending on the assumed longitude. For this example, assume that all observations are taken at the same time every day at 8 p.m. local time.

```
    obs.hour = 20;
```

For the dead reckoning process, update the observation structure with the estimate of the current location. In this case, the location is estimated using the previous fix, the vessel's velocity V, and course T.

```
    obs.longitude = longFix(m)+(1/60)*V(m)*sind(T(m))/cosd(latFix(m));
    obs.latitude = latFix(m)+(1/60)*V(m)*cosd(T(m));
```

Adjust the local time to UTC using the helper function astHelperLongitudeHour. This function adjusts the UTC observation time depending on the estimated longitude of the vessel.

```
    obs.UTC = astHelperLongitudeHour(obs);
```

Sextant Altitude Calculation

For each of the planets, the astHelperNauticalCalculation helper function calculates the sextant measurement that the crew in the Kon-Tiki would have measured. This function models the actual behavior of the planets, while compensating for the local conditions. This function uses the planetary ephemeris and the ECI to ECEF transformation matrix. The analysis doesn't include planetary aberration, gravitational light deflection, and aberration of light phenomena.

```
obs.Hs = astHelperNauticalCalculation(obs,latActual,longActual);
```

Calculate position

The following calculations replace the use of the nautical almanac. They include the use of the planetary ephemerides and the ECI to ECEF transformation matrix.

Initialize declination, Greenwich Hour Angle (GHA), and altitude for the observed object:

```
obs.declination = zeros(size(obs.Hs));
obs.GHA = zeros(size(obs.Hs));
obs.altitude = zeros(size(obs.Hs));
```

Calculate the modified Julian date for the measurement time:

```
mjd = mjuliandate(obs.UTC);
```

Calculate the difference between UT1 and UTC:

```
dUT1 = deltaUT1(mjd,'Action','None');
```

Calculate the ECI to ECEF transformation matrix using the values of TAI-UTC (dAT) from the U.S. Naval Observatory:

```
dAT = 1.4228180;
TM = dcmeci2ecef('IAU-76/FK5',obs.UTC,dAT,dUT1);
```

Calculate Julian date for Terrestrial Time to approximate the Barycentric Dynamical Time:

```
jdTT = juliandate(obs.UTC)+(dAT+32.184)/86400;
```

Calculate declination, Greenwich Hour Angle, and altitude for each of the celestial objects:

```
for k =1:length(obs.object)
```

Calculate the ECI position for every planet:

```
        posECI = planetEphemeris(jdTT,'Earth',obs.object{k},'405','km');
```

Calculate the ECEF position of every planet:

```
        posECEF = TM*posECI';
```

Calculate the Greenwich Hour Angle (GHA) and declination using the ECEF position:

```
        obs.GHA(k) = -atan2d(posECEF(2),posECEF(1));
        obs.declination(k) = atan2d(posECEF(3),sqrt(posECEF(1)^2 + ...
            posECEF(2)^2));
```

Calculate the distance from the surface of the Earth to the center of the planet using the ECEF to LLA transformation function:

```
        posLLA = ecef2lla(1000*posECEF');
        obs.altitude(k) = posLLA(3);

    end
```

Sight Reduction for Planetary Objects

Reduce sight for each of the planets specified in the observation structure array:

```
    [p,Z] = astHelperNauticalReduction(obs);
```

Calculate the increments to the latitude and longitude from the last fix for the current fix using the following equations. These equations are based on the Nautical Almanac.

```
    Ap = sum(cosd(Z).^2);
    Bp = sum(cosd(Z).*sind(Z));
    Cp = sum(sind(Z).^2);
    Dp = sum(p.*cosd(Z));
    Ep = sum(p.*sind(Z));
    Gp = Ap*Cp-Bp^2;
```

Calculate the increments for latitude and longitude according to the reduction:

```
    deltaLongFix = (Ap*Ep-Bp*Dp)/(Gp*cosd(latFix(m)));
    deltaLatFix = (Cp*Dp-Bp*Ep)/Gp;
```

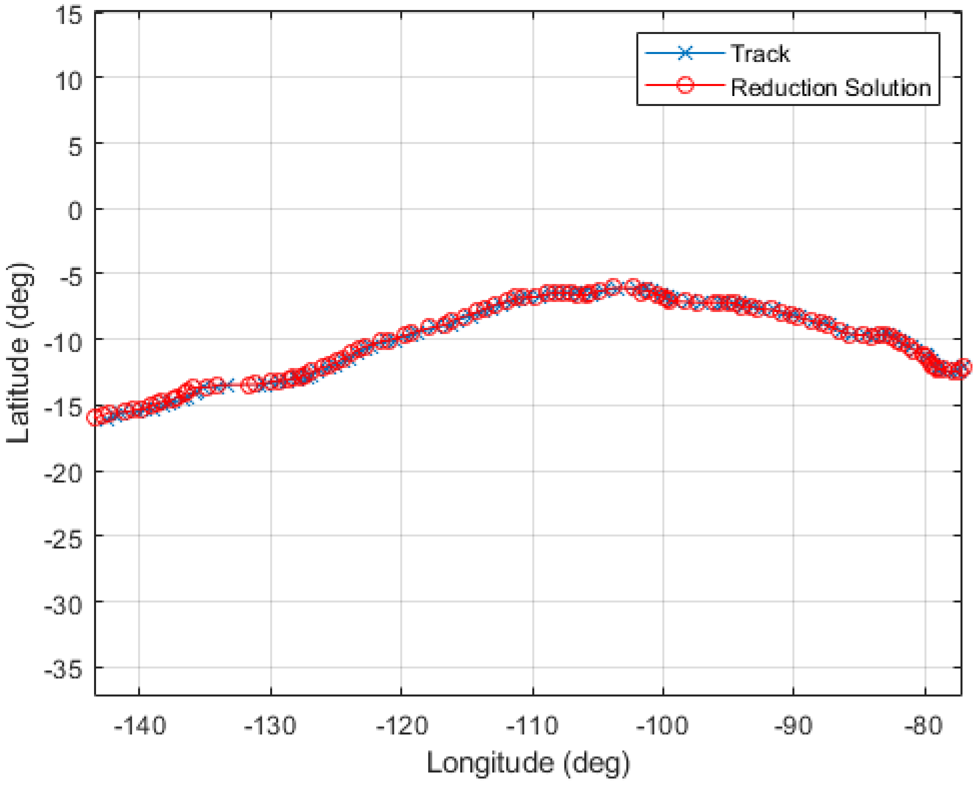
After the increments for the latitude and longitude are calculated, add them to the estimated location, obtaining the fix for the time of observation:

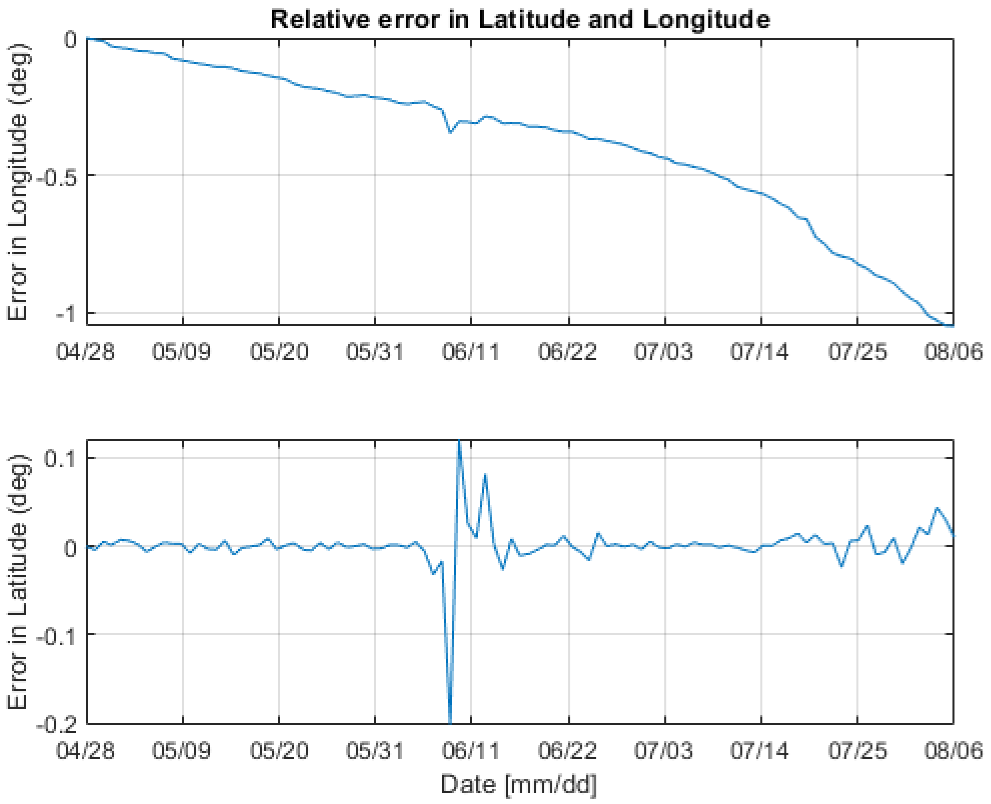
```
    longFix(m+1) = obs.longitude + deltaLongFix;
    latFix(m+1) = obs.latitude + deltaLatFix;

end
```

Navigation Solution Visualization

The following figure shows the actual track and sight reduction solutions:

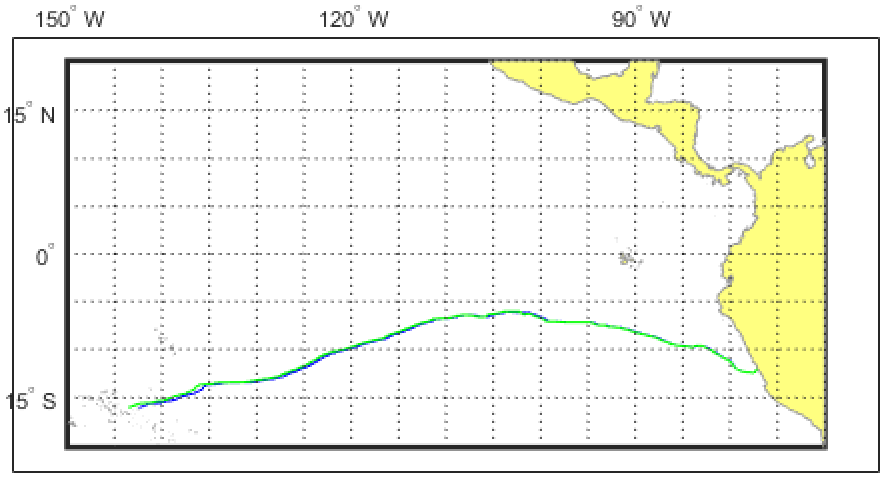
```
astHelperVisualization(long,lat,longFix,latFix,'Plot')
```





You can use the Mapping Toolbox to obtain a more detailed graph depicting the sight reduction solutions with the American continent and French Polynesia.

```
astHelperVisualization(long,lat,longFix,latFix,'Map')
```



The relative error in longitude and latitude is accumulated as the vessel sails from Callao to Raroia. This error is due to small errors in the measurement of the sextant altitude. For June 9th, the reduction method calculates a true azimuth (Z) for Neptune and Jupiter. The true azimuths for Neptune and Jupiter are close to 180 deg apart. This difference causes a small peak in the relative error. This error, however, is still within the reduction method error boundaries.

References

[1] Bowditch, N. The American Practical Navigator. National Geospatial Intelligence Agency, 2012.

[2] United Kingdom Hydrographic Office. Nautical Almanac 2012 Commercial Edition. Paradise Publications, Inc. 2011.

[3] Urban, Sean E. and P. Kenneth Seidelmann. Explanatory Supplement to the Astronomical Almanac. 3rd Ed., University Science Books, 2013.

[4] United States Naval Observatory. https://www.usno.navy.mil.

Estimate Sun Analemma Using Planetary Ephemerides and ECI to AER Transformation

This example shows how to estimate the analemma of the Sun. The analemma is the curve that represents the variation of the angular offset of the Sun from its mean position on the celestial sphere relative to a specific geolocation on the Earth surface. In this example, the analemma is estimated relative to the Royal Observatory at Greenwich, United Kingdom. After the estimation, the example plots the analemma.

This example uses data that you can download using the aeroDataPackage command.

Identify the Dates of a Year over Which to Calculate the Analemma of the Sun

Specify the dates for which to calculate the analemma. In this example, these dates range for the year 2014 from January 1st to December 31st at noon UTC.

```
dv = datetime(2014,1,1:365,12,0,0);
dvUTC = [dv.Year' dv.Month' dv.Day' dv.Hour' dv.Minute' dv.Second'];
```

Calculate the Position of the Sun

Use the planetEphemeris function to calculate the position of the Sun. In this example:

- The tdbjuliandate function calculates the Julian date for the dynamic barycentric time (TDB).
- The tdbjuliandate function requires the terrestrial time (TT).

The calculation of the terrestrial time in seconds from UTC requires the difference in Coordinated Universal Time (UTC) and International Atomic Time (TAI).

- For 2014, this difference (dAT) is 35 seconds.
- The approximate terrestrial time (secTT) is the dAT + 32.184 seconds.
- The terrestrial time in year, month, day, hour, minutes, and seconds is contained in the dvTT array.

```
dAT = 35;
secTT = dAT + 32.184;
dvTT = dv + secTT/86400;
```

Estimate the Julian date for the dynamic barycentric time based on the terrestrial time using the array dvTT:

```
jdTDB = tdbjuliandate([dvTT.Year' dvTT.Month' dvTT.Day' dvTT.Hour' dvTT.Minute' dvTT.Se
```

Determine the position of the Sun for these dates:

```
posSun = planetEphemeris(jdTDB,'Earth','Sun')*1000;
```

Calculate Difference Between UTC and Principal Universal Time (UT1)

Calculate the difference between UTC and UT1, deltaUT1, using the modified Julian dates for UTC.

```
mjdUTC = mjuliandate(dvUTC);
dUT1 = deltaUT1(mjdUTC);
```

Calculate Polar Motion and Displacement of the Celestial Intermediate Pole (CIP)

Calculate the polar motion and displacement of the CIP using the modified Julian dates for UTC.

```
PM = polarMotion(mjdUTC);
dCIP = deltaCIP(mjdUTC);
```

Specify the Geopotential Position of the Royal Observatory at Greenwich, United Kingdom

Specify the geopotential location for the position against which to estimate the analemma. In this example, this location is the latitude, longitude, and altitude for the Royal Observatory at Greenwich (51.48 degrees North, 0.0015 degrees West, 0 meters altitude).

```
LLAGreenwich = [51.48,-0.0015,0];
aer = eci2aer(posSun,dvUTC,repmat(LLAGreenwich,length(jdTDB),1),...
              'deltaAT',dAT*ones(length(jdTDB),1),'deltaUT1',dUT1,...
              'PolarMotion',PM,'dCIP',dCIP);
```

Specify Days Within the Year of the Analemma That You Want to Plot

On the analemma, you can plot days of interest within the year of the analemma. This example plots:

- The first day of each month in 2014.
- The summer and winter solstices.
- The spring and fall equinoxes.

To get the first day of each month in 2014:

```
aerFirstMonth = aer(dvUTC(:,3)==1,:);
```

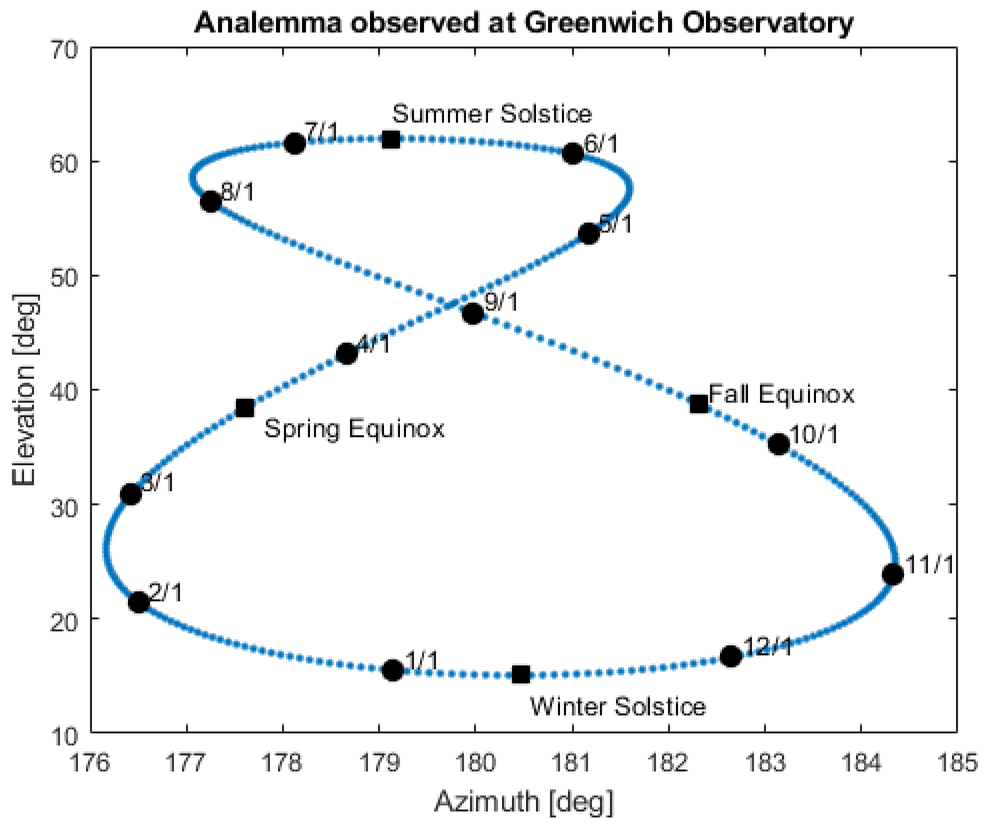
To get solstices and equinoxes (for 2014 are 3/20, 6/21, 9/22, 12/21):

```
solsticeEquinox = [ aer(and(dvUTC(:,2)==3,dvUTC(:,3)==20),1) aer(and(dvUTC(:,2)==3,dvU
                    aer(and(dvUTC(:,2)==6,dvUTC(:,3)==21),1) aer(and(dvUTC(:,2)==6,dvU
                    aer(and(dvUTC(:,2)==9,dvUTC(:,3)==22),1) aer(and(dvUTC(:,2)==9,dvU
                    aer(and(dvUTC(:,2)==12,dvUTC(:,3)==21),1) aer(and(dvUTC(:,2)==12,dv
```

Plot Results

Plot the analemma. Along the analemma, plot points for the whole year, first days of the month, equinoxes, and solstices.

```
for ii = 12:-1:1
    firstDays{ii} = [num2str(ii) '/' num2str(1)];
end

f = figure;
plot(aer(:,1),aer(:,2),'.',...
    solsticeEquinox(:,1),solsticeEquinox(:,2),'ks',...
    aerFirstMonth(:,1),aerFirstMonth(:,2),'ko',...
    'MarkerSize',8,'MarkerFaceColor','k');
title('Analemma observed at Greenwich Observatory');
xlabel('Azimuth [deg]');
ylabel('Elevation [deg]');
axis([176,185,10,70])
text(aerFirstMonth(:,1)+.1, aerFirstMonth(:,2)+1.2, firstDays, 'Color', 'k','Horizonta
text(solsticeEquinox(1,1)+.2, solsticeEquinox(1,2)-1.5, 'Spring Equinox', 'Color', 'k',
text(solsticeEquinox(2,1), solsticeEquinox(2,2)+2.5, 'Summer Solstice', 'Color', 'k','H
text(solsticeEquinox(3,1)+.1, solsticeEquinox(3,2)+1.2, 'Fall Equinox', 'Color', 'k','H
text(solsticeEquinox(4,1)+.1, solsticeEquinox(4,2)-2.5, 'Winter Solstice', 'Color', 'k
```


Analemma observed at Greenwich Observatory
Elevation [deg]
Azimuth [deg]
70
60
50
40
30
20
10
176
177
178
179
180
181
182
183
184
185
Summer Solstice
7/1
6/1
8/1
5/1
9/1
4/1
Fall Equinox
Spring Equinox
10/1
3/1
11/1
2/1
12/1
1/1
Winter Solstice

Display Flight Trajectory Data Using Flight Instruments and Flight Animation

This example shows how to visualize flight trajectories in a UI figure window using flight instrument components. In this example, you will create and configure standard flight instruments in conjunction with the Aero.Animation object.

Load Recorded Data for Flight Trajectories and Instrument Display

Load logged aircraft position, attitude, and time to the workspace.

```
load simdata
```

Create Animation Interface

To display the flight trajectories stored in the flight trajectory data, create an Aero.Animation object. The aircraft used in this example is the Piper PA24-250 Comanche.

```
h = Aero.Animation;
h.createBody('pa24-250_orange.ac','Ac3d');
h.Bodies{1}.TimeSeriesSource = simdata;
h.Camera.PositionFcn = @staticCameraPosition;
h.Figure.Position(1) = h.Figure.Position(1) + 572/2;
h.show();
```

Create Flight Instruments

Create a UI figure window to contain the flight instruments.

```
fig = uifigure('Name','Flight Instruments',...
    'Position',[h.Figure.Position(1)-572 h.Figure.Position(2)+h.Figure.Position(4)-502
    'Color',[0.2667 0.2706 0.2784],'Resize','off');
```

Load panel image into an axis:

```
imgPanel = imread('astFlightInstrumentPanel.png');
ax = uiaxes('Parent',fig,'Visible','off','Position',[10 30 530 460],...
    'BackgroundColor',[0.2667 0.2706 0.2784]);
image(ax,imgPanel);
disableDefaultInteractivity(ax);
```

Create standard flight instruments for navigation:

Create altimeter:

```
alt = uiaeroaltimeter('Parent',fig,'Position',[369 299 144 144]);
```

Create heading indicator:

```
head = uiaeroheading('Parent',fig,'Position',[212 104 144 144]);
```

Create airspeed indicator:

```
air = uiaeroairspeed('Parent',fig,'Position',[56 299 144 144]);
```

Change the airspeed indicator limits according to the Piper PA 24-250 Comanche capabilities:

```
air.Limits = [20 200];
air.ScaleColorLimits = [0 50;40 160;160 190;190 200];
```

Create artificial horizon:

```
hor = uiaerohorizon('Parent',fig,'Position',[212 299 144 144]);
```

Create climb rate indicator:

```
climb = uiaeroclimb('Parent',fig,'Position',[369 104 144 144]);
```

Change the climb indicator maximum climb rate according to the aircraft capabilities:

```
climb.MaximumRate = 4000;
```

Create turn coordinator:

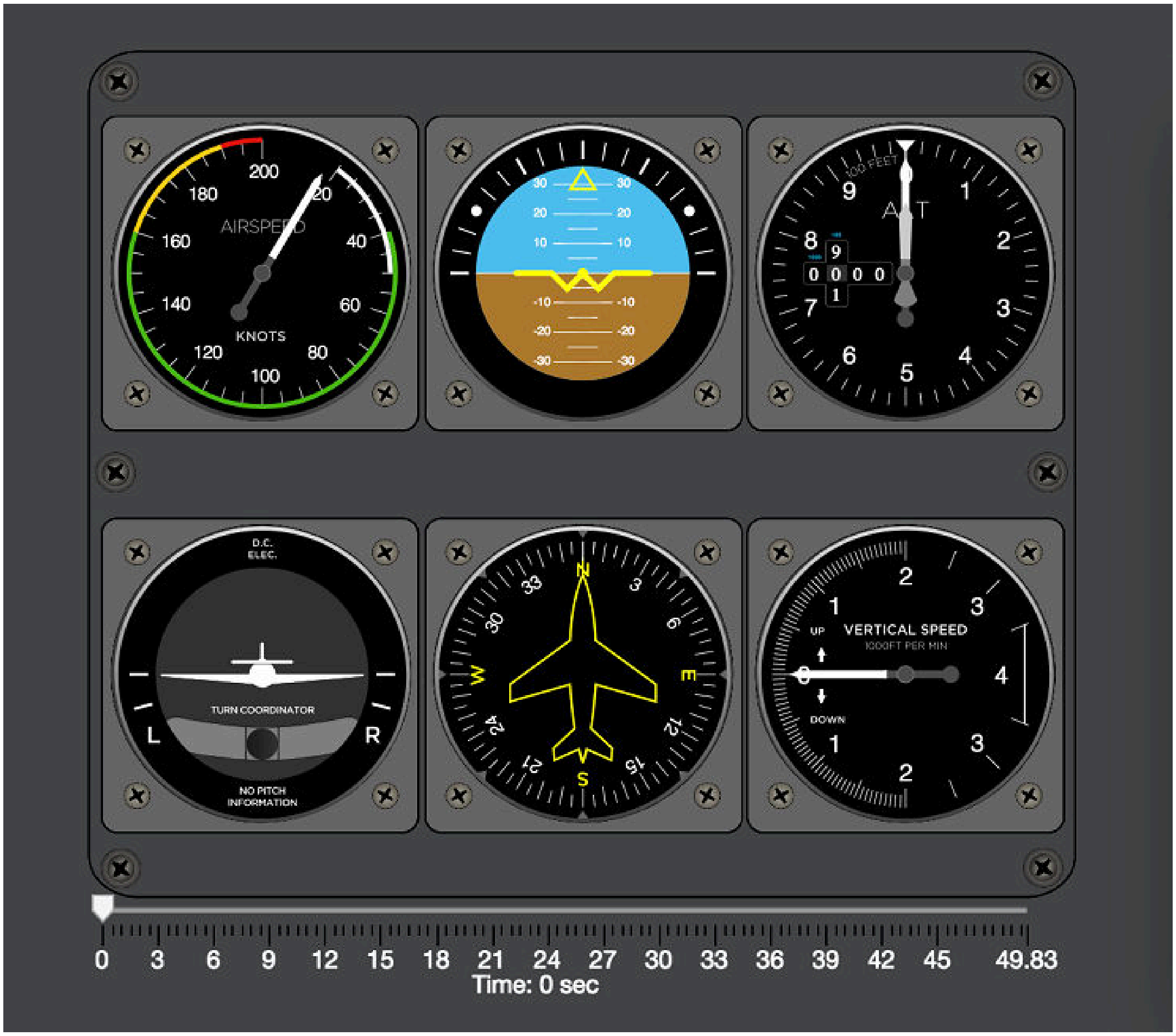
```
turn = uiaeroturn('Parent',fig,'Position',[56 104 144 144]);
```

To update the flight instruments and animation figure, assign the `ValueChangingFcn` callback to the `astHelperFlightInstrumentsAnimation` helper function. Then, when a time is selected on the slider, the flight instruments and animation figure will be updated according to the selected time value.

```
sl = uislider('Parent',fig,'Limits',[simdata(1,1) simdata(end,1)],'FontColor','white')
sl.Position = [50 60 450 3];
sl.ValueChangingFcn = @(sl,event) astHelperFlightInstrumentsAnimation(fig,simdata,h,eve
```

To display the time selected in the slider, create a label component.

```
lbl = uilabel('Parent',fig,'Text',['Time: ' num2str(sl.Value,4) ' sec'],'FontColor','wh
lbl.Position = [230 10 90 30];
```

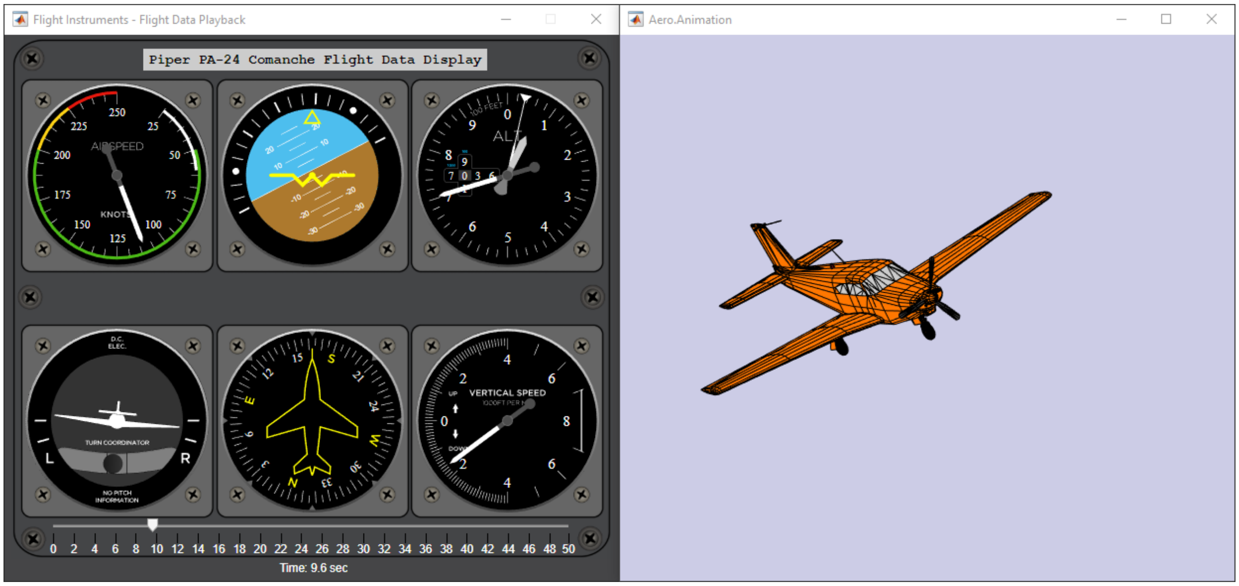


Aerospace Flight Instruments in App Designer

This app shows how to display flight status information with standard cockpit instrumentation using Aerospace Toolbox flight instruments in App Designer. On startup, the app loads saved flight data from a MAT-file and starts a new `Aero.Animation` figure window. The app uses six flight instruments to display flight data corresponding with the time selected in the slider. The animation window updates to reflect the aircraft orientation at the selected time.

This example demonstrates the following app building tasks:

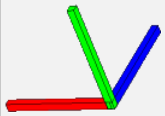
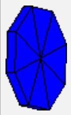
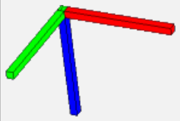
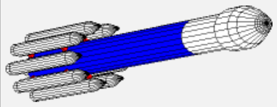
- Use a `StartupFcn` callback to load data from a file and create an `Aero.Animation` object.
- Use aerospace flight instrument components to visualize flight status information: Airspeed Indicator, Artificial Horizon, Altimeter, Turn Coordinator, Heading Indicator, and Climb Indicator
- Use the slider component `ValueChangingFcn` callback to set aerospace flight instrument component properties and interact with an `Aero.Animation` object.



AC3D Files and Thumbnails

AC3D Files and Thumbnails Overview

Aerospace Toolbox demos use the following AC3D files, located in the `matlabroot\toolbox\aero\astdemos` folder. For other AC3D files, see `https://www.flightgear.org/download/download-aircraft/` and click the **Download Aircraft** link.

Thumbnail	AC3D File
	`ac3d_xyzisrgb.ac`
	`blueoctagon.ac`
	`bluewedge.ac`
	`body_xyzisrgb.ac`
	`delta2.ac`
	`greenarrow.ac`
	`pa24–250_blue.ac`
	`pa24–250_orange.ac`

Thumbnail	AC3D File
	`redwedge.ac`
	`testrocket.ac`